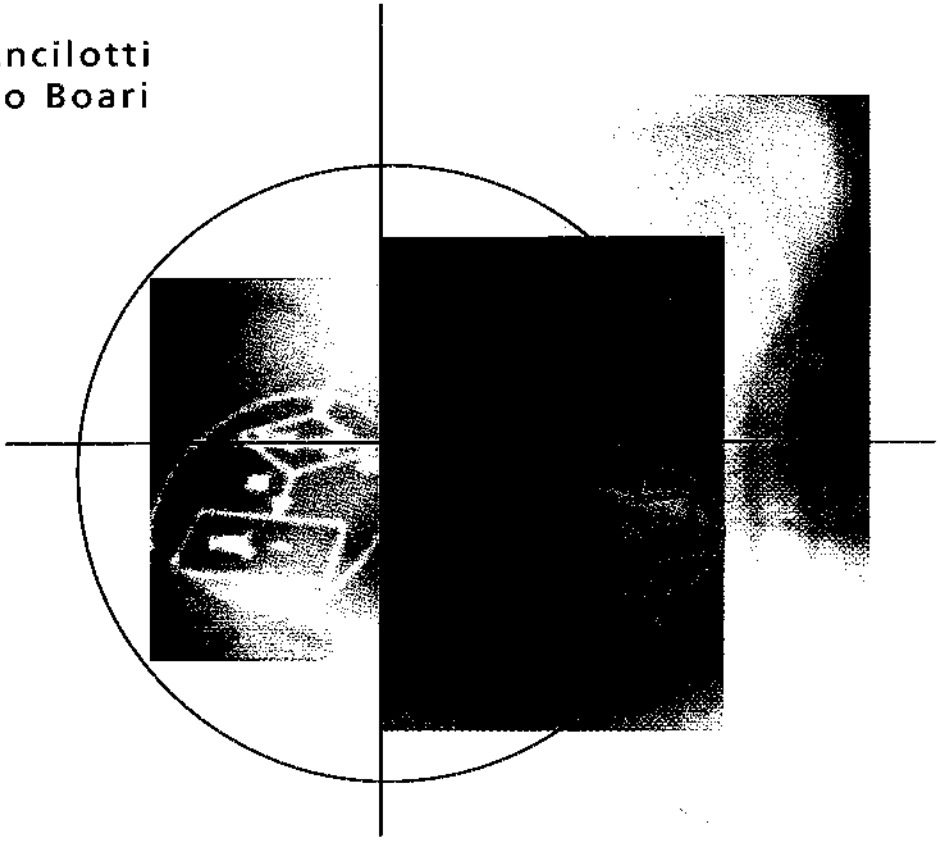
Paolo Ancilotti
Maurelio Boari



Programmazione concorrente e distribuita

McGraw-Hill

1
Concetti fondamentali

Il principale concetto alla base della programmazione concorrente è quello di *processo*. Infatti, un programma concorrente, e come vedremo anche un programma distribuito, può essere definito come un programma che, durante la sua esecuzione, sviluppa due o più processi destinati a svolgere, congiuntamente, un determinato compito. È proprio questa caratteristica che distingue un programma concorrente, o distribuito, da un tradizionale programma sequenziale che, viceversa, durante la sua esecuzione sviluppa un unico processo (un unico *thread*, nell'accezione normalmente utilizzata, soprattutto nell'ambito dei sistemi operativi, per indicare l'esecuzione di una sequenza di istruzioni).

Scopo di questo capitolo è quello di riconsiderare alcune semplici nozioni fondamentali che sono alla base della programmazione, cercando di estenderle, a partire dal tradizionale ambito della programmazione sequenziale, a quello più ampio e generale della programmazione concorrente e distribuita.

1.1 Il concetto di processo

Partiamo, prima di tutto, dal concetto di *processore* (spesso indicato, nel seguito, anche con l'acronimo CPU da *Central Processing Unit*), inteso come unità di elaborazione in grado di eseguire, una dopo l'altra e in modo prettamente sequenziale, le istruzioni contenute in un programma. Un *programma* non è altro che la descrizione di un *algoritmo* mediante un formalismo, il linguaggio di programmazione, che consente di descrivere i singoli passi dell'algoritmo in termini delle istruzioni che il processore è in grado di interpretare ed eseguire. L'esecuzione di un'istruzione da parte del processore corrisponde a un'*azione* che produce un ben determinato *ef-*

le) intenderemo, infine, la sequenza di azioni corrispondenti all'esecuzione della sequenza di istruzioni di un programma da parte del processore.

Esiste, quindi, una distinzione concettuale tra programma e processo. Mentre il programma è un'entità statica che descrive un algoritmo, il processo è un'entità dinamica che identifica l'attività svolta dal processore durante l'esecuzione del programma. È quindi chiaro che uno stesso programma può dar luogo, durante due diverse esecuzioni, a due processi completamente diversi poiché la sequenza di istruzioni che il processore eseguirà dipende, in generale, dai dati di ingresso. In pratica, uno stesso programma può evocare durante la sua esecuzione un numero di processi, diversi tra loro, spesso molto elevato e in certi casi infinito.

Essendo il processo un'entità dinamica, la sua definizione come sequenza di azioni fa implicitamente riferimento al concetto di tempo. In particolare, chiameremo *durata di un processo* (indicata spesso anche con il termine di *vita di un processo*) l'intervallo temporale che intercorre tra l'istante in cui il processore inizia l'esecuzione del programma e l'istante in cui tale esecuzione termina. Indicheremo, inoltre, con *stato del processore*, in un particolare istante della vita di un processo, l'insieme dei valori contenuti in tutti gli oggetti coinvolti.

Un modo spesso utilizzato per rappresentare un processo è quello di identificarlo con la sequenza degli stati del processore così come vengono generati durante l'esecuzione del programma. Questa sequenza prende anche il nome di *storia del processo* o di *traccia dell'esecuzione del programma*.

Facciamo un semplicissimo esempio per chiarire ulteriormente questi concetti. Supponiamo di voler scrivere un programma che calcola il Massimo Comun Divisore (MCD) di due interi positivi x e y e supponiamo di voler utilizzare il seguente algoritmo, descritto informalmente come segue:

. controllare se i due interi x e y sono uguali, nel qual caso il MCD coincide con il loro comune valore;
. se x e y sono diversi tra loro, valutare la loro differenza;
. tornare al punto 1, prendendo in considerazione il più piccolo tra i due e la loro differenza.

'ale algoritmo si basa sulle ben note proprietà del MCD, proprietà che possiamo descrivere con le seguenti relazioni:

) MCD(i,i)==i
) MCD(i,j)==MCD(j,i)
) Se i>j allora MCD(i,j)==MCD(i-j,j)

escriviamo, quindi, il programma come una semplice funzione a cui i valori d'ingresso x e y sono passati come parametri e che restituisce il risultato desiderato:

```
nt MCD(int x,int y) {
  int a,b;
  a=x;
  b=y;
  while(a!=b)
     if(a>b) a=a-b;
     else b=b-a;
  return a;
```

	stato iniziale	S1	S2	S3	S4	S5	stato finale
a	--	18	18	18	6	6	6
b	--	--	30	12	12	6	6
MCD	--	--	--	--	--	--	6

tempo

Figura 1.1 Traccia dell'esecuzione del programma.

Il processo generato dal processore durante l'esecuzione di questa funzione, supponendo che i due valori d'ingresso x e y siano rispettivamente 18 e 30, può essere rappresentato tramite la traccia riportata nella figura 1.1.

Nella figura sono indicati tutti gli stati attraverso i quali passa il processore: a partire dallo stato iniziale, nel quale tutte le variabili coinvolte sono ancora non specificate, si passa nello stato S1 con l'esecuzione della prima istruzione (a=x;) e così di seguito, fino allo stato finale.

Un diverso modo di rappresentare graficamente un processo è quello di fare ricorso a un grafo orientato, detto *grafo di precedenza del processo*. I nodi del grafo rappresentano i singoli eventi generati dal processore (cioè i cambiamenti di stato prodotti dalle sue azioni) nell'ordine in cui questi si verificano. Gli archi del grafo identificano le precedenze temporali tra tali eventi. Essendo il processo strettamente sequenziale, il grafo di precedenza sarà necessariamente un *grafo a ordinamento totale*. Con tale termine si intende il fatto che ogni nodo ha esattamente un predecessore e un successore eccettuato il nodo iniziale (senza predecessore) e quello finale (senza successore). Nella figura 1.2 è rappresentato il grafo di precedenza del processo precedentemente illustrato nella figura 1.1.

1.2 Test e verifica di un programma

La differenza tra il concetto di programma e quello di processo acquista una particolare importanza quando, terminata la scrittura di un programma, ci poniamo il problema di verificarne la correttezza e cioè di verificare che il suo comportamento corrisponda esattamente a quanto previsto in fase di progetto. È sicuramente questa la fase più critica e complessa di tutto il ciclo di vita di un programma e, come vedremo, lo è ancora di più per un programma concorrente.

La prassi che usualmente viene seguita per controllare la correttezza di un programma (nota col termine di *program testing*) è quella di eseguire lo stesso un certo numero di volte, cambiando di volta in volta i dati d'ingresso e verificando che i risultati ottenuti siano quelli attesi. In caso negativo sarà necessario individuare la causa del malfunzionamento (*debugging*) e, una volta corretto l'errore, riprovare il programma.

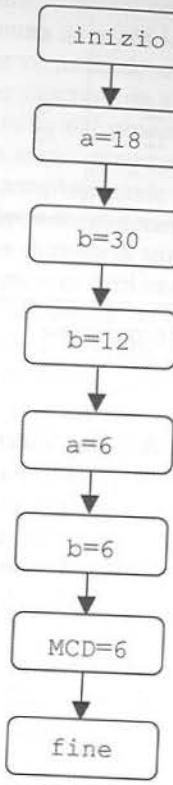


Figura 1.2 Grafo di precedenza del processo.

Dopo un sufficiente numero di prove terminate con successo si acquisisce una sufficiente convinzione che il programma sia corretto, convinzione tanto più fondata quanto maggiore è il numero delle prove effettuate e quanto più accurata sarà stata la scelta dei dati di prova utilizzati per effettuare queste verifiche. Nonostante tutto, però, la tecnica del testing non potrà mai dare la certezza assoluta che il programma sia del tutto corretto e che non generi mai malfunzionamenti. In effetti questa tecnica consente di verificare la correttezza di alcuni processi, quelli generati dalle esecuzioni del programma con i dati di test, ma questo, ovviamente, non significa che risulti corretto un qualunque altro processo che quel programma può generare cambiando i dati d'ingresso.

Per raggiungere questo grado di garanzia, invece di verificare la correttezza di un sottoinsieme dei processi che il programma può generare, dovremmo cercare di verificare la correttezza dello stesso programma (*program verification*) e con essa, quindi, quella di un qualunque suo processo.

Per ottenere questo obiettivo è necessario astrarre dai singoli processi e, in base alla struttura del programma, cercare di convincerci della sua correttezza qualunque sia la sua particolare esecuzione. A questo fine, possiamo cercare di ricavare una traccia dell'esecuzione del programma analoga a quella ricavata da una particolare esecuzione durante la fase di test (vedi per esempio la figura 1.1), ma che sia indipendente dai particolari valori assunti dalle variabili. In questo senso, invece di registrare gli specifici valori assunti dalle variabili durante l'esecuzione, è necessario registrare i *range* di valori che queste possono assumere e, successivamente, ricavare condizioni logiche che legano tra loro i valori delle variabili e che dovranno essere vere nei vari punti del programma durante ogni sua possibile esecuzione. Chiameremo *traccia simbolica del programma* la traccia così ricavata. Si dice "simbolica" in quanto i singoli stati del processore saranno caratterizzati non da specifici valori delle variabili, ma dai range di valori che le variabili possono assumere e dalle specifiche condizioni logiche che, nei singoli stati, legano tra loro le variabili del programma. In altri termini, si tratta di verificare "a mano" il programma cercando di ipotizzare una sua "esecuzione simbolica".

Per esemplificare, possiamo fare riferimento al precedente programma relativo al calcolo del Massimo Comun Divisore. Nel costruire la traccia simbolica del programma (vedi figura 1.3) possiamo fare riferimento alla figura 1.1 già vista precedentemente.

Dopo l'esecuzione della prima istruzione il processore passa nello stato S1 caratterizzato dal fatto che la variabile `a` ha assunto il valore `x`, qualunque questo sia. Analogamente, dopo l'esecuzione della seconda istruzione lo stato S2 sarà caratterizzato dalla seguente condizione logica:

S2: `{(a==x)&&(b==y)}`

Nella figura 1.1, sono poi riportati gli stati S2, S3, S4 e S5 che sono relativi all'esecuzione delle istruzioni contenute nel ciclo `while` del programma. In particolare, questi stati che precedono lo stato finale sono quattro poiché, per lo specifico processo illustrato nella figura, sono quattro le iterazioni del ciclo. Ovviamente, se i dati d'ingresso fossero stati diversi sarebbe stato diverso anche il numero di iterazioni e quindi il numero di stati intermedi prima dello stato finale. Il nostro compito, però, è quello di descrivere il comportamento del programma indipendentemente dai particolari valori e quindi dal numero di iterazioni del ciclo. È quindi necessario astrarre dal numero di iterazioni del ciclo cercando di individuare una relazione che caratterizzi tutti gli stati del processore quando inizia l'esecuzione di una qualunque iterazione del ciclo. Indicando questo generico stato con SI (vedi figura 1.3), chiameremo *invariante del ciclo* la relazione logica che lo caratterizza. "Invariante" in quanto deve valere indipendentemente dal numero di volte che il ciclo viene eseguito.

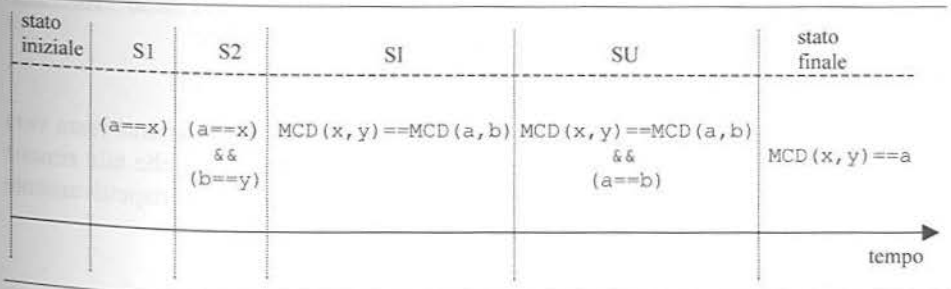


Figura 1.3 Traccia dell'esecuzione del programma.

Nel nostro esempio, in base all'algoritmo utilizzato per scrivere il programma, è semplice ricavare tale relazione. Per esempio possiamo asserire che la seguente relazione costituisce l'invariante del ciclo:

SI: {MCD(x,y)==MCD(a,b)}

Per convincerci di questa affermazione è sufficiente verificare che:

a) la relazione è vera la prima volta che si esegue il `while`;
b) per il principio di induzione, supposta vera all'inizio del corpo del `while`, allora è ancora vera al termine della sua esecuzione e quindi è ancora vera all'inizio del ciclo successivo.

Il punto a) è ovviamente verificato in base alla relazione che caratterizza lo stato S2 in cui si trova il processore quando esegue il `while` per la prima volta.

Per verificare il punto b) supponiamo che la relazione sia vera all'inizio di una qualunque esecuzione del corpo del ciclo. L'istruzione successiva a essere eseguita è la seguente:

```
if(a>b)a=a-b;
else b=b-a;
```

Qui abbiamo due alternative: (a>b) oppure (a<b) poiché, essendo nel corpo del `while`, sicuramente `a` è diverso da `b`. Supponiamo vero il primo caso, per cui il processore esegue il ramo `then` dell'istruzione `if`. Ricordando che abbiamo ipotizzato vera la relazione invariante, questa sarà ovviamente vera anche all'inizio dell'esecuzione di questo ramo dell'`if`. Inoltre, avendo supposto di eseguire il ramo `then`, sarà anche vera la relazione (a>b). Per cui, lo stato del processore prima che venga eseguita l'istruzione `a=a-b` (indichiamolo con ST1) può essere così caratterizzato:

ST1: {(a>b)&&MCD(x,y)==MCD(a,b)}

Ma, in base alla proprietà c) del `MCD` richiamata nel precedente paragrafo, ciò equivale anche alla seguente relazione:

ST1: {(a>b)&&MCD(x,y)==MCD(a-b,b)}

Se questa relazione è vera prima dell'istruzione di assegnamento `a=a-b` allora, dopo la sua esecuzione, sarà vera una relazione nella quale al posto di (a-b) possiamo sostituire `a`. Per cui, lo stato del processore dopo l'esecuzione del ramo `then` dell'`if` (indicato con ST2) sarà caratterizzato dalla seguente relazione:

ST2: {(a>0)&&MCD(x,y)==MCD(a,b)}

Ciò dimostra che eseguendo il ramo `then` dell'`if` la relazione invariante resta vera alla fine del ciclo. Con analogo ragionamento, possiamo verificare che tale rimane anche eseguendo il ramo `else`, ricavando le seguenti relazioni vere rispettivamente prima e dopo l'esecuzione dell'istruzione `b=b-a`:

SF1: {(b>a)&&MCD(x,y)==MCD(a,b-a)}

SF2: {(b>0)&&MCD(x,y)==MCD(a,b)}

A questo punto possiamo caratterizzare lo stato del processore in uscita dal ciclo, e cioè quando (a==b) (vedi stato SU nella figura 1.3):

SU: {MCD(x,y)==MCD(a,b)&&(a==b)}

e quindi, in base alla proprietà a) del `MCD` richiamata nel precedente paragrafo, sarà vera la seguente relazione che caratterizza lo stato finale:

{MCD(x,y)==a}

che dimostra la correttezza del programma.

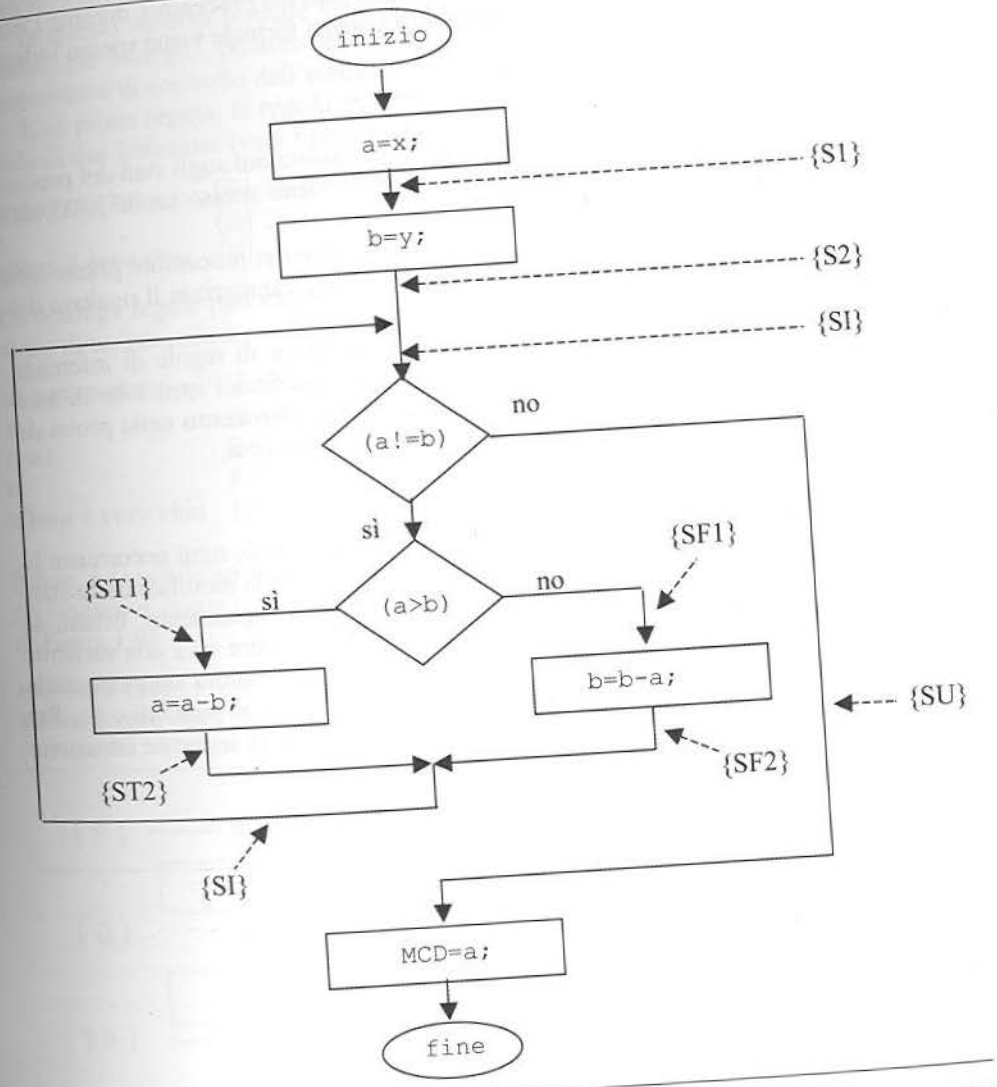


Figura 1.4 Grafo di flusso e asserzioni relative alla funzione `MCD`.

Per meglio illustrare gli stati attraverso i quali passa il processore durante l'esecuzione del programma possiamo ricorrere a una descrizione grafica, riportando il grafo di flusso del programma e indicando le relazioni che caratterizzano i valori delle variabili nei singoli punti del programma. Tali relazioni, note anche con il termine di *asserzioni*, caratterizzano i singoli stati del processore. Nella figura 1.4 vengono illustrati il grafo di flusso della funzione `MCD` e le asserzioni relative ai singoli punti del programma.

Questo approccio alla verifica delle proprietà di un programma tramite una sua analisi astratta è stato introdotto alla fine degli anni '60, con l'obiettivo di definire un metodo formale di verifica basato sulla logica dei predicati. In pratica, le asserzioni sono dei predicati che caratterizzano i singoli stati del processore durante l'esecuzione del programma. Una formula in questo sistema formale viene spesso indicata nella seguente maniera:

```
{ P } S { Q }
```

dove `S` è un'istruzione del programma e `P` e `Q` sono asserzioni sugli stati del processore rispettivamente prima e dopo l'esecuzione di `S` (dette spesso anche *precondizione* e *postcondizione* di `S`, vedi figura 1.5).

La precondizione di un'istruzione caratterizza lo stato del processore prima della sua esecuzione, mentre la corrispondente postcondizione caratterizza il risultato dell'esecuzione dell'istruzione stessa.

Vengono quindi stabiliti un certo numero di assiomi e di regole di inferenza. L'assioma più importante è sicuramente quello che caratterizza l'istruzione di assegnamento (a cui abbiamo implicitamente fatto più volte riferimento nella prova del precedente programma del `MCD`) e che potremmo descrivere così:

$P_{var <- val}$ `} var=val; { P }`

ove $P_{var <- val}$ rappresenta lo stesso predicato `P` in cui, però, ogni occorrenza libera di `var` viene sostituita da `val`. Questo assioma descrive la modifica dello stato del processore come conseguenza dell'esecuzione di un assegnamento. Infatti, se nello stato che precede l'esecuzione dell'assegnamento del valore `val` alla variabile `var` vale una certa condizione in cui compare il valore `val`, allora sicuramente la stessa condizione è vera dopo l'assegnamento sostituendo `var` al posto di `val`. Per esempio, supponiamo che in un certo programma sia presente la seguente istruzione di assegnamento:

```
a=b+c;
```

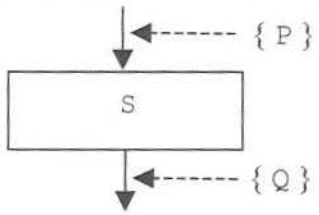


Figura 1.5 Precondizione e postcondizione.

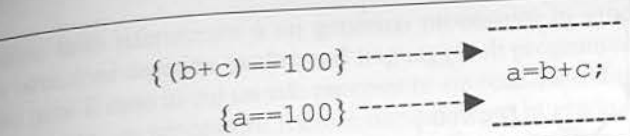


Figura 1.6 Regola di assegnamento.

e supponiamo che prima della sua esecuzione sia possibile provare che `{(b+c)==100}` (precondizione): allora l'ovvia postcondizione dell'assegnamento sarà necessariamente `{a==100}` (vedi figura 1.6).

In questo sistema formale vengono poi definite alcune regole di inferenza. Fra queste, per esempio, possono essere definite le regole che caratterizzano gli usuali meccanismi di controllo dell'esecuzione di istruzioni in un programma.

Una prima regola, la *regola di composizione*, caratterizza l'esecuzione sequenziale di due istruzioni (vedi figura 1.7):

se è vero che: `{P} Sa; {Q}`
e: `{Q} Sb; {R}`
allora è vero che: `{P} Sa; Sb; {R}`

Un'analoga regola può essere descritta nel seguente modo relativamente all'esecuzione di un'istruzione *condizionale* (vedi figura 1.8):

se è vero che: `{P&&C} Sa; {Q}`
e: `{P&&!C} Sb; {R}`
e se: `Q => W`
e: `R => W`
allora è vero che: `{P} if(C)Sa; else Sb; {W}`

Infine, è possibile specificare una regola di inferenza relativamente a istruzioni di tipo *ripetitivo*, come indicato di seguito dove `I` rappresenta l'invariante del ciclo (vedi figura 1.9):

se è vero che: `{I&&C} S {I}`
allora è vero che: `{I} while(C) S; {I&&!C}`

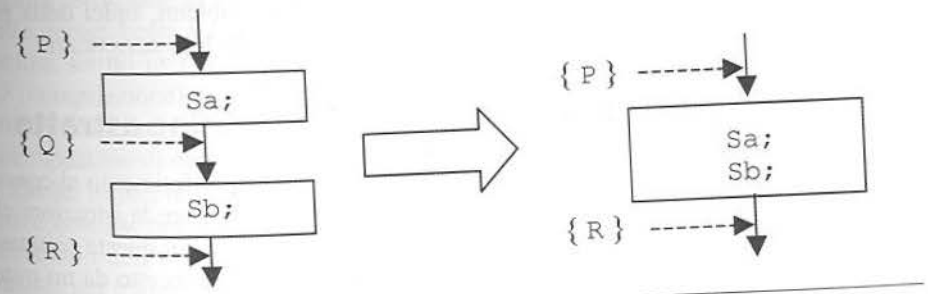


Figura 1.7 Regola di composizione.

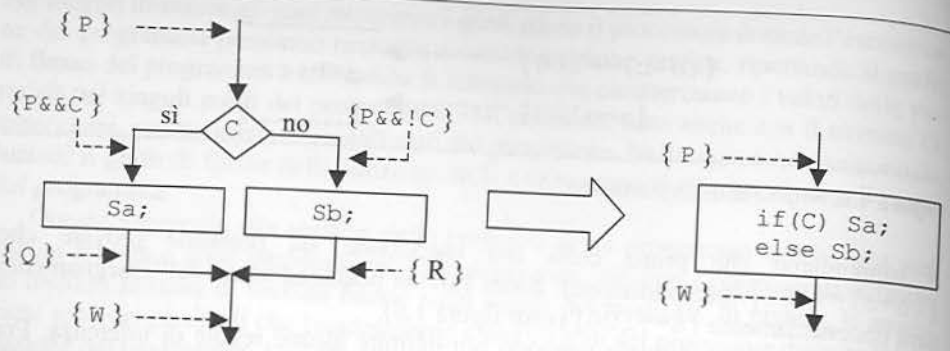


Figura 1.8 Regola condizionale.

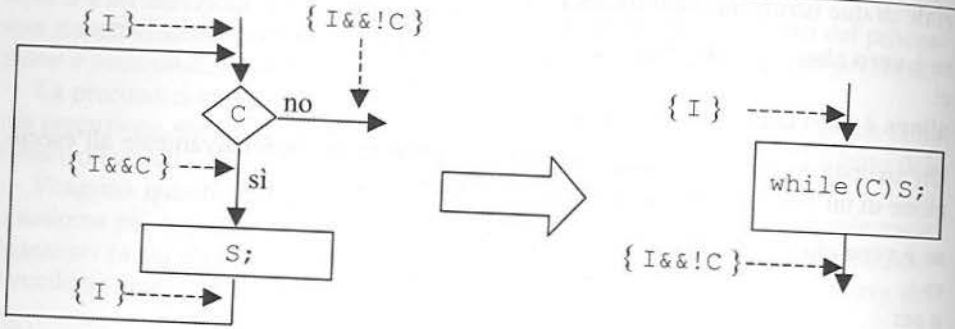


Figura 1.9 Regola ripetitiva.

Lo scopo di assiomatizzare il significato delle varie istruzioni di un linguaggio di programmazione è stato quello di definire un sistema formale da utilizzare per verificare matematicamente la correttezza di un programma e, al tempo stesso, di fornire un aiuto per la sintesi di programmi. In realtà, siamo ancora lontani dal raggiungimento di questi obiettivi. Nel seguito cercheremo di utilizzare i precedenti concetti, spesso in modo informale, non tanto per effettuare dimostrazioni di correttezza quanto, piuttosto, come aiuto nelle definizione di soluzioni ad alcuni problemi, tipici della programmazione concorrente, con l'obiettivo di convincerci della loro correttezza.

1.3 Linguaggi di programmazione e macchine astratte

Nell'introdurre il concetto di processo si è fatto esplicitamente riferimento al concetto di processore inteso come macchina che, in grado di interpretare le istruzioni che compongono un programma, le esegue rispettando l'ordine con cui queste compaiono nel programma. Fino a ora, però, abbiamo considerato tale concetto da un punto di vista del tutto generale, astraendo da possibili realizzazioni fisiche del processore. Per esempio, con riferimento al precedente programma del `MCD` e ai relativi processi, abbiamo fatto riferimento a un generico processore in grado di interpretare ed eseguire istruzioni descritte mediante il linguaggio di programmazione C.

È raro però il caso in cui un tale processore sia realizzato direttamente in hardware. Come è noto, un processore fisico è normalmente in grado di eseguire istruzioni molto più semplici ed elementari di quelle fornite da un linguaggio di programmazione ad alto livello. La possibilità di eseguire programmi scritti in tali linguaggi viene assicurata, in questi casi, da un insieme di programmi di sistema: gli interpreti e/o i compilatori.

In generale, dato un qualunque linguaggio di programmazione *LS* (*linguaggio sorgente*), indicheremo con *PS* il processore astratto che è in grado di interpretare ed eseguire le istruzioni proprie di *LS*. Le tecniche di realizzazione di *PS* possono essere molto diverse tra loro, ciascuna con pregi e difetti nei confronti delle altre, soprattutto per quanto riguarda aspetti relativi alla flessibilità di implementazione e all'efficienza di esecuzione. Le varie soluzioni, però, sono del tutto equivalenti dal punto di vista di chi, utilizzando il linguaggio *LS*, deve scrivere un programma e verificarne la correttezza, almeno fino a quando si rimane nel contesto della tradizionale programmazione sequenziale. In questo senso, almeno per il momento, faremo genericamente riferimento a un *PS* astraendo dai suoi dettagli realizzativi.

Richiamiamo comunque, brevemente, le principali tecniche di realizzazione di un processore astratto *PS* relativo a un generico linguaggio di programmazione *LS*. Indichiamo con *PO* il processore fisico utilizzato per la realizzazione di *PS*: *PO* è ovviamente in grado di interpretare ed eseguire soltanto le istruzioni macchina proprie del linguaggio oggetto *LO*.

La prima tecnica, ovvia ma del tutto teorica, è quella che prevede di realizzare *PS* direttamente in hardware. In questo caso *PS* coincide con *PO* e le istruzioni di *LS* sono direttamente interpretate dalla macchina fisica. Questa è sicuramente la soluzione più efficiente ma anche quella estremamente meno flessibile e più difficile da implementare.

Una diversa tecnica è quella che consiste nel realizzare su *PO* un programma in grado di interpretare le istruzioni di *LS* (interprete), simulando così il processore *PS* sulla macchina fisica *PO*. Questa soluzione è relativamente semplice da realizzare ma soffre di scarsa efficienza a causa del fatto che l'interprete del linguaggio viene simulato.

Un'alternativa del tutto diversa è quella in cui il processore *PS* è del tutto virtuale, nel senso che non esiste nessun interprete, né hardware né software, in grado di eseguire istruzioni di *LS*. Le uniche istruzioni che possono essere eseguite sono quelle appartenenti a *LO*. In questo caso si utilizza la tecnica di tradurre ogni programma scritto in *LS* in un programma funzionalmente equivalente ma scritto in *LO* (compilazione) da far eseguire direttamente su *PO*. Questa soluzione consente di ottenere elevati livelli di efficienza di esecuzione anche se comporta una maggiore occupazione di memoria relativamente al codice tradotto.

Esiste, infine, un'alternativa che si pone in posizione intermedia fra le due precedenti e consiste nel definire un processore astratto di livello intermedio tra *PS* e *PO*: indichiamo con *PI* tale processore e con *LI* il suo linguaggio. Questa alternativa consiste nel tradurre il programma scritto in linguaggio sorgente *LS* in un programma equivalente scritto nel linguaggio intermedio *LI* e nel realizzare su *PO* un interprete di *LI* che simuli il processore *PI* (vedi figura 1.10).

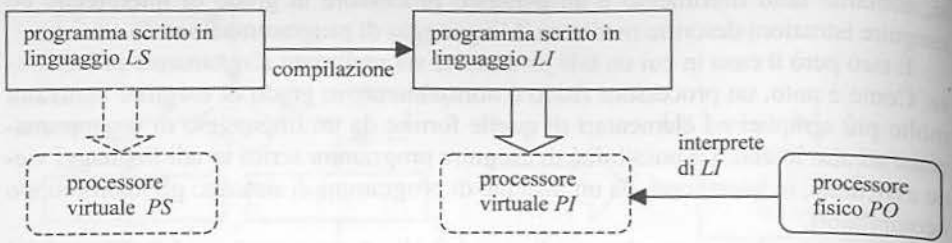


Figura 1.10 Compilazione e interpretazione.

Il compito di *PI* è quello di estendere le funzionalità della macchina fisica *PO* in modo tale da semplificare la traduzione dal linguaggio sorgente e, al tempo stesso, di facilitare la portabilità di un linguaggio su macchine diverse. Si tratta, in questo caso di riscrivere esclusivamente l'interprete del linguaggio intermedio. Questa soluzione è sicuramente la più utilizzata. Un tipico esempio è quello del linguaggio Java con la definizione del *byte code* come linguaggio intermedio e della *Java Virtual Machine*.

La figura 1.10 contiene, come casi particolari, anche le due soluzioni precedenti: la soluzione completamente interpretativa, nel caso in cui *PI* coincida con *PS*, e la soluzione completamente compilativa nel caso in cui *PI* coincida con PO.

Una soluzione completamente compilativa è comunque raramente utilizzata. In genere il compilatore genera sempre il codice per una macchina *PI* che possiamo considerare coma un'estensione della macchina fisica *PO*. La differenza tra le due è costituita da quell'insieme di strutture dati e di funzioni che, caricate in memoria insieme al codice tradotto, costituiscono il cosiddetto supporto a tempo di esecuzione (RTS, *Run Time Support*) del linguaggio *LS*. Il linguaggio intermedio *LI* coincide col linguaggio oggetto *LO*, esteso con le chiamate alle funzioni del RTS. Esempi tipici di funzioni appartenenti al RTS sono le funzioni di ingresso/uscita e quelle relative alla gestione della pila dei record di attivazione e della memoria dinamica (*heap*).

Come già indicato precedentemente, il programmatore vede il processore astratto *PS* senza preoccuparsi di come questo venga realizzato. Con l'introduzione della programmazione concorrente sarà però necessario rivedere questa affermazione. Per chiarire questo aspetto richiamiamo, nel paragrafo successivo, i concetti che stanno alla base dei meccanismi di astrazione.

1.4 Meccanismi di astrazione

Le tecniche di astrazione rappresentano uno strumento, valido in tutti i settori dell'ingegneria, di fondamentale importanza per gestire la complessità di un sistema durante il processo della sua progettazione. Tali tecniche rappresentano lo strumento mediante il quale il progettista può dedicare la propria attenzione agli aspetti importanti del fenomeno da trattare, ignorando tutti quegli aspetti che, viceversa, sono trascurabili relativamente all'obiettivo da raggiungere.

Le tecniche di astrazione permeano l'intero ambito della programmazione. Come indicato precedentemente, gli stessi linguaggi di programmazione rappresentano delle astrazioni realizzate "al di sopra" della macchina fisica. È l'applicazione delle tecniche di astrazione che, come è noto, ha dato luogo ai due paradigmi di progettazione del software noti come *top-down* e *bottom-up*.

Il primo, noto anche come paradigma *stepwise refinement*, consiste nell'applicare, ricorsivamente, il criterio di decomposizione di un modulo software in componenti più semplici da trattare fino ad arrivare a componenti di limitata complessità in grado di essere facilmente realizzati.

Il secondo, noto anche come paradigma di *information hiding*, consiste, viceversa, nel partire dalla macchina su cui il sistema deve essere realizzato estendendo le sue funzionalità con alcuni moduli software che permettano di definire una macchina astratta più potente e sulla quale possano essere più facilmente implementati ulteriori e più potenti moduli e così via, ricorsivamente, fino ad arrivare alla definizione di una macchina astratta sufficientemente potente, sulla quale realizzare più semplicemente l'intero sistema.

I due paradigmi non sono antitetici e vengono spesso utilizzati congiuntamente, dando luogo a sistemi software organizzati secondo un insieme di livelli di astrazione strutturati gerarchicamente e dove ogni livello definisce una nuova macchina astratta che, utilizzando le funzionalità della macchina di livello più basso, implementa nuove funzionalità necessarie per l'implementazione della macchina di livello superiore.

Come è noto dalla teoria dei sistemi operativi, la strutturazione di un sistema a livelli mediante una gerarchia di macchine astratte è una tecnica spesso utilizzata nel progetto di un nuovo sistema operativo. In questo caso, si parte dalla macchina fisica estendendo le sue funzionalità mediante l'aggiunta di moduli software che implementano nuove funzionalità astratte. La nuova macchina viene quindi utilizzata per implementare una seconda macchina astratta mediante l'aggiunta di ulteriori funzionalità che vengono definite e implementate disponendo di tutte le funzionalità della precedente macchina e astraendo dai loro dettagli implementativi.

Per capire meglio cosa significa "estendere le funzionalità di una macchina" mediante l'aggiunta di moduli software, cerchiamo di riflettere su quali sono le funzionalità che una macchina deve mettere a disposizione di un programmatore. Parafrasando il titolo di un famoso testo di Wirth [1], un programma può essere definito come la codifica di un algoritmo e un insieme di strutture dati che codificano le informazioni da elaborare. In questo senso, le funzionalità che una macchina deve mettere a disposizione di un programmatore sono di tre tipologie diverse:

a) un insieme di operazioni mediante le quali codificare i singoli passi dell'algoritmo;
b) un insieme di meccanismi per il controllo del flusso di esecuzione delle istruzioni, mediante il quale il programmatore definisce l'ordine con cui le istruzioni che compongono il programma devono essere eseguite;
c) un insieme di tipi di dati[1] mediante i quali codificare le informazioni da elaborare.

[1] Come è noto dalla teoria dei linguaggi di programmazione, possiamo definire il concetto di tipo di dato come una coppia di insiemi: l'insieme di tutti e soli i valori che una variabile del tipo può assumere e un insieme di operazioni, le sole che possono essere utilizzate per operare su una variabile del tipo.

Per esempio, con riferimento a un tradizionale processore fisico l'insieme a) è costituito da tutte le istruzioni macchina (`load`, `store`, `add`, `sub` ecc.) eccettuate quelle utilizzate per il controllo del flusso di esecuzione che, viceversa, fanno parte dell'insieme b). Fra quest'ultime ci sono le istruzioni di salto e di salto condizionato, quelle di salto a sottoprogramma e di ritorno da sottoprogramma, il meccanismo di interruzione ecc. L'insieme c) è costituito dai tipi di dati elementari (bit, byte ecc.) per i quali sono presenti, fra le istruzioni macchina, istruzioni che hanno operandi appartenenti a questi tipi. Nel seguito, faremo riferimento alle funzionalità offerte dalla macchina fisica identificandole come funzionalità *primitive* o, spesso, come funzionalità *atomiche*, per distinguerle da quelle implementate mediante moduli software che, viceversa, saranno identificate come funzionalità astratte. Il significato dell'aggettivo "atomico" riferito alle funzionalità primitive sarà chiarito nel seguito.

Vediamo adesso che tipi di moduli software vengono utilizzati per implementare nuove funzionalità astratte e che tipi di costrutti linguistici vengono forniti a questo scopo dai linguaggi di programmazione. Limitiamo, per il momento, la nostra attenzione ai moduli utilizzati per definire nuove operazioni astratte e nuovi tipi di dati astratti mentre nuovi meccanismi astratti per il controllo del flusso saranno ampiamente analizzati nel seguito con riferimento, in particolare, ai meccanismi propri della concorrenza.

Generalmente, un modulo software destinato a introdurre una nuova funzionalità astratta è costituito da due parti: la prima, nota come *interfaccia*, destinata a specificare le funzionalità offerte dal modulo, la seconda, nota come *corpo del modulo*, contiene il codice che realizza le funzionalità specificate nell'interfaccia. Normalmente, le regole di visibilità del linguaggio garantiscono che l'utilizzatore di un modulo possa "vedere" esclusivamente l'interfaccia del modulo, ma non il codice contenuto nel corpo. Conformemente ai criteri di modularità, ciò garantisce che una qualunque modifica dell'implementazione di un modulo, che non modifichi la relativa interfaccia, non influenzi gli utilizzatori dello stesso.

1.4.1 Operazioni astratte

I meccanismi linguistici destinati alla definizione di operazioni astratte sono noti fin dai primi linguaggi ad alto livello e sono costituiti dai meccanismi per la definizione di procedure e funzioni.

Mediante questi costrutti, il programmatore può definire una nuova operazione astratta definendone l'interfaccia costituita dal nome della nuova operazione e della lista dei parametri su cui opera, più un corpo che realizza la nuova operazione come una sequenza di operazioni primitive o di operazioni astratte precedentemente definite. È così possibile estendere l'insieme delle operazioni disponibili in modo tale che il programmatore possa utilizzare questo insieme più ampio astraendo dai dettagli implementativi delle varie operazioni, cioè indipendentemente dal fatto che le stesse siano fornite in hardware o tramite moduli software. In effetti, se si prescinde dai evidenti problemi di efficienza di esecuzione, il fatto che una certa operazione sia primitiva o astratta non influenza minimamente la funzionalità del programma che la usa. Questo è sicuramente vero nel contesto della programmazione sequenziale ma, come anticipato precedentemente, non è sempre vero quando passiamo a uno più generale quale quello della programmazione concorrente, la quale prevede la possibilità di scrivere programmi che, durante la loro esecuzione, diano luogo a più processi eseguiti contemporaneamente.

La differenza semantica che esiste tra le operazioni primitive e quelle astratte è data da una proprietà che è tipica delle prime e che, viceversa, non è posseduta dalle seconde. Tale proprietà è quella dell'*atomicità*, termine con cui si intende che ogni istruzione macchina è "non divisibile". Concettualmente possiamo considerare come istantanea la sua esecuzione, nel senso che il processore non produce nessun effetto visibile tra l'istante in cui essa ha inizio e quello in cui termina. Ciò è dovuto al fatto che un'eventuale interruzione è servita dal processore soltanto prima o dopo l'esecuzione di un'istruzione. Questa caratteristica resta valida anche nel caso in cui si considerino macchine con più processori (architetture multiprocessore). Tali architetture consentono di eseguire più programmi sequenziali contemporaneamente, per esempio uno su un processore e uno su un altro. È quindi ovvio, in questo caso, che più processi vengono svolti contemporaneamente e, perciò, due o più istruzioni possono essere eseguite nello stesso istante, ovviamente su processori distinti. Nel caso però che due processori tentino di eseguire contemporaneamente una stessa istruzione con lo stesso operando, la caratteristica di atomicità (garantita, in questo caso, dal meccanismo di arbitraggio del bus) ci garantisce che le due istruzioni non potranno essere eseguite insieme. La loro istantaneità implica che una sarà eseguita prima dell'altra o viceversa.

In base alla sua definizione, un'operazione astratta, in quanto composta da una sequenza di istruzioni primitive, non è certamente indivisibile e, quindi, atomica. Infatti, il processore può essere interrotto durante la sua esecuzione e, a causa di questa interruzione, può iniziare l'esecuzione di una diversa operazione prima che quella interrotta sia stata completata. Questa caratteristica contraddistingue le operazioni astratte da quelle primitive e, come vedremo, può comportare malfunzionamenti durante l'esecuzione di un programma concorrente se non se ne tiene opportunamente conto.

1.4.2 Tipi di dati astratti

La possibilità di definire nuovi tipi di dati è una caratteristica ormai propria dei moderni linguaggi di programmazione. I costrutti linguistici messi a disposizione del programmatore per questo scopo sono diversi da linguaggio a linguaggio ma, indipendentemente da queste differenze e in conformità con la definizione di tipo di dato, consentono al programmatore di specificare sia l'insieme dei valori che quello delle operazioni che caratterizzano il nuovo tipo. Inoltre, come ogni meccanismo di astrazione, consentono di separare la specifica dell'interfaccia da quella del corpo del modulo. L'interfaccia, visibile a chi dovrà utilizzare il nuovo tipo, è costituita dal nome del tipo e dalle specifiche delle operazioni che potranno essere utilizzate per operare su variabili del nuovo tipo. Il corpo, non visibile all'esterno del modulo, specifica l'implementazione del nuovo tipo, cioè definisce sia la struttura dati, che caratterizza la codifica dei valori del nuovo tipo in termini di tipi di dati primitivi o di tipi astratti precedentemente definiti, sia l'implementazione di tutte le operazioni definite per il nuovo tipo. Come esempio di costrutto linguistico utilizzato per questo scopo, faremo riferimento al costrutto `class` proprio di molti altri linguaggi di programmazione. In particolare, nel seguito utilizzeremo la seguente notazione sintattica:

```
class   <identificatore del nuovo tipo astratto> {
        <struttura dati, non visibile all'esterno della classe>;
        {<eventuale istruzione di inizializzazione>};
public  <dichiarazioni delle funzioni che rappresentano i metodi di accesso agli
        oggetti della classe. Le specifiche di queste funzioni sono visibile all'e-
        sterno, come indicato dalla parola chiave public>;
}
```

Riportiamo nella figura 1.11 la definizione di un tipo di dati astratto relativo a una coda di valori logici (*boolean*), per esempio una coda che possa contenere al più sedici valori logici.

La classe definisce il nuovo tipo specificando il suo identificatore, che in questo caso è `coda_di_sedici_boolean`, la parte privata non visibile all'utilizzatore, che definisce come viene rappresentata in memoria ogni nuova istanza del tipo, e la parte pubblica, costituita da tutte le funzioni che possono essere invocate per operare sulle istanze del tipo. In questo caso, come esempio di implementazione del nuovo tipo è stata scelta la nota rappresentazione di una coda mediante un buffer circolare (vedi figura 1.12).

```
class coda_di_sedici_boolean {
   boolean vettore[16];
   int primo=0; ultimo=0; cont=0;

   public boolean vuota() {
      return(cont==0);
   }

   public boolean piena() {
      return(cont==16);
   }

   public void inserisci(boolean x) {
      if(cont==16) {<solleva l'eccezione Overflow>;}
      vettore [ultimo]=x;
      cont ++;
      ultimo=(ultimo+1)%16;
   }

   public boolean estrai(){
      if(cont==0) {<solleva l'eccezione Underflow>;}
      boolean temp=vettore [primo];
      cont -;
      primo=(primo+1)% 16;
      return temp;
   }
}
```

Figura 1.11 Coda di valori logici.

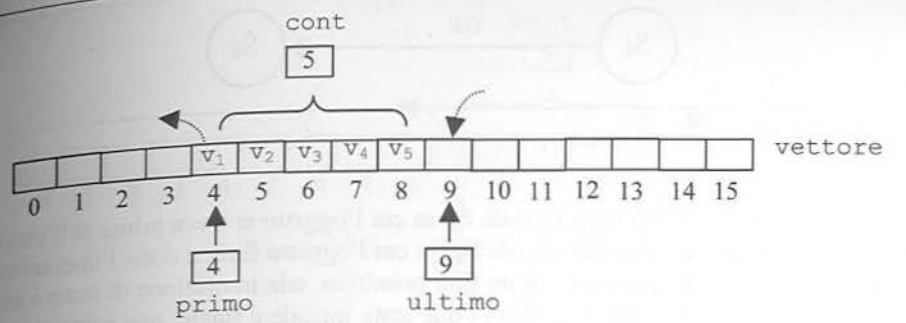


Figura 1.12 Coda di sedici boolean.

Il buffer è costituito dal vettore di valori boolean `vettore`. I tre campi interi `primo`, `ultimo` e `cont` identificano rispettivamente: l'indice dell'elemento di `vettore` contenente il valore che dovrà essere estratto per primo (cioè quello che è stato inserito per primo fra quelli presenti nella coda); l'indice dell'elemento di `vettore` nel quale inserire il prossimo valore; il numero di valori attualmente presenti nella coda. Quando la coda è vuota (`cont==0`), i valori dei due indici `primo` e `ultimo` coincidono e tornano a coincidere dopo 16 inserimenti consecutivi. In questo caso, però, il valore di `cont` è uguale a 16.

I due metodi `vuota()` e `piena()` sono definiti come funzioni che restituiscono un `boolean` e che servono per verificare se, in un certo istante, la coda è rispettivamente vuota o piena. Infine, sono forniti i due metodi `inserisci` ed `estrai`, da utilizzare per inserire un nuovo elemento nella coda, o per estrarre il primo fra quelli presenti. Il metodo `inserisci` verifica che la coda non sia piena (altrimenti solleva l'eccezione di `overflow`). L'inserimento viene effettuato nell'elemento del `vettore` il cui indice è contenuto nel campo `ultimo`. Quindi vengono incrementati il campo `cont`, per indicare che nella coda è presente un nuovo valore, e, infine, anche il campo `ultimo`, per predisporlo al prossimo inserimento. Quest'ultimo incremento viene effettuato modulo 16 operando, quindi, su `vettore` in maniera circolare. Il metodo `estrai` è realizzato in maniera del tutto analoga operando sul campo `primo`.

Una volta definita la classe possono essere dichiarati oggetti del nuovo tipo come, per esempio, il seguente oggetto `var`:

```
coda_di_sedici_boolean var;
```

su cui operare con i metodi precedentemente definiti:

```
var.inserisci(true);
```

Come già visto a proposito delle astrazioni procedurali, anche per gli oggetti esiste una differenza semantica fra quelli di tipo primitivo e quelli di tipo astratto.

Un oggetto di tipo primitivo è atomico, intendendo con tale aggettivo il seguente significato: l'oggetto può essere in un numero finito di stati, tanti quanti sono i valori che può assumere, e ogni operazione `OP` eseguita sull'oggetto implica una sua

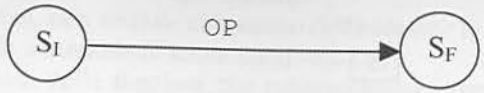


Figura 1.13 Transizione di stato.

transizione di stato, dallo stato iniziale S_I, in cui l'oggetto si trova prima dell'esecuzione dell'operazione, allo stato finale S_F, in cui l'oggetto finisce dopo l'operazione. Essendo atomiche le operazioni di un tipo primitivo, tale transizione di stato è essa stessa atomica, non divisibile, cioè tra i due stati, iniziale e finale, non esistono stati intermedi attraverso i quali l'oggetto passa (vedi figura 1.13).

Facciamo un esempio: prendiamo in considerazione un tipo primitivo, come il tipo bit o il tipo boolean a esso equivalente. Ogni oggetto di questo tipo può essere in uno dei due stati `true` e `false` (zero e uno). Se, per esempio, l'oggetto si trova nello stato `true` e su di esso viene eseguita l'operazione di `negazione`, l'oggetto subisce una transizione che lo porta, atomicamente, nello stato `false`.

Anche per un oggetto di tipo astratto valgono considerazioni analoghe. Per esempio, un oggetto di tipo `coda_di_sedici_boolean` può essere in un numero finito di stati corrispondenti ai valori che può assumere. Può essere nello stato *coda vuota*, oppure in uno dei due stati *coda con un valore* (valore `true` o valore `false`), oppure ancora in uno dei quattro stati *coda con due valori*, e così via fino ai 2^{16} stati di *coda piena*. Anche per un tale oggetto l'esecuzione di un'operazione implica una transizione di stato. Per esempio, se l'oggetto si trova in uno dei due stati *coda con un valore* e su di esso viene eseguito il metodo `inserisci(a)`, l'oggetto transita in uno stato *coda con due valori*. Purtroppo però, a differenza di quanto avviene per gli oggetti primitivi, questa transizione di stato non è atomica. Infatti, un oggetto di tipo astratto è, in generale, composto da più oggetti di tipo primitivo (o di tipi astratti precedentemente definiti). In questo senso, lo stato in cui si trova un oggetto astratto, può essere definito in due diversi modi, uno significativo per chi usa il tipo astratto, l'altro per chi lo definisce: possiamo indicarli, rispettivamente, come *stato esterno* e *stato interno*. Lo stato esterno corrisponde ai valori che l'oggetto può assumere come entità unica, concettualmente distinta dai suoi componenti. Gli stati esterni sono quelli visibili a chi utilizza il tipo astratto, senza conoscere i dettagli relativi alla sua implementazione: nel caso della `coda_di_sedici_boolean`, corrispondono agli stati precedentemente elencati. Viceversa, lo stato interno, significativo per che implementa il nuovo tipo, lo possiamo definire come l'aggregazione degli stati (esterni) degli oggetti componenti. Per le regole di visibilità del linguaggio, tale stato non è quindi visibile all'esterno del modulo che implementa il tipo astratto.

Per esempio, nella figura 1.12 viene illustrato il caso di un oggetto di tipo `coda_di_sedici_boolean` contenente cinque valori. Lo stato esterno è, quindi, uno di quelli *coda con cinque valori*, mentre lo stato interno corrisponde all'aggregazione dei seguenti stati degli oggetti componenti: `primo`==4, `ultimo`==9, `cont`==5, `vettore[4]`==v_1, `vettore[5]`==v_2, ..., `vettore[8]`==v_5, e dove v_1, v_2, ..., v_5 denotano gli specifici valori `boolean` contenuti nella coda.

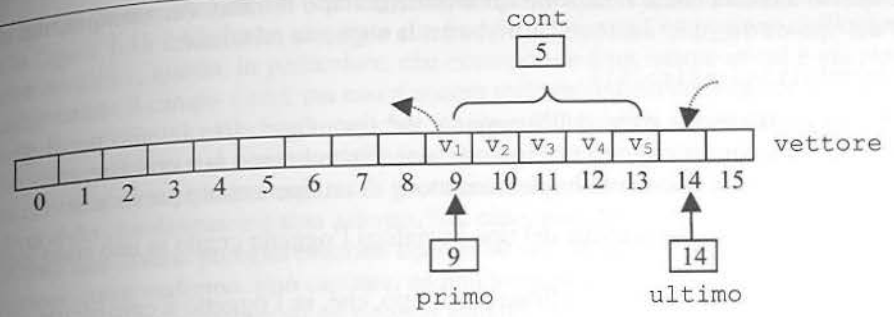


Figura 1.14 Coda con cinque valori.

Esiste, ovviamente, una corrispondenza fra stati interni e stati esterni, relativa alla funzione concettuale di *astrazione*. Tale corrispondenza non è però biunivoca. Infatti, a diversi stati interni può corrispondere lo stesso stato esterno. Per esempio, nella figura 1.14 è riportato il caso di un diverso stato interno della coda che corrisponde però allo stesso stato esterno *coda con cinque valori*, illustrato nella figura 1.12. Si suppone, in particolare, che i cinque valori v_1, v_2, ..., v_5 siano contenuti negli elementi del `vettore` rispettivamente di indici 9, 10, ..., 13. Quindi, anche i valori contenuti nei campi `primo` ed `ultimo` sono ovviamente diversi da prima.

La corrispondenza tra stati interni e stati esterni è inoltre parziale, cioè definita soltanto per un sottoinsieme degli stati interni. Per esempio, lo stato interno della coda illustrato nella figura 1.15 non ha un corrispondente stato esterno. Infatti, il valore 4 del campo `cont`, che denota il numero di valori contenuti nella coda, farebbe supporre che lo stato esterno fosse uno di quelli *coda con quattro valori*, mentre la differenza fra i valori contenuti nei campi `ultimo` e `primo`, che individuano rispettivamente il primo elemento vuoto e il primo elemento pieno del `vettore`, è cinque, che equivale a uno stato della coda *coda con cinque valori*.

Nel seguito, gli stati interni a cui corrispondono stati esterni significativi verranno indicati come *stati interni consistenti*. Essi sono caratterizzati dalla validità di una relazione fra i valori dei campi componenti l'oggetto astratto: tale relazione vie-

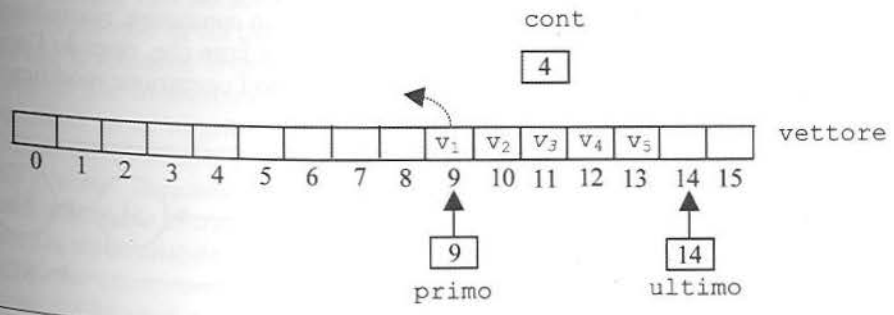


Figura 1.15 Stato interno inconsistente.

ne spesso indicata come *relazione invariante* del tipo astratto. Per esempio, nel caso del tipo `coda_di_sedici_boolean` la seguente relazione:

```
cont==(ultimo-primo)%16
```

deve, ovviamente, far parte dell'invariante del tipo. Ogni stato interno per il quale tale relazione non è verificata corrisponde a uno stato interno non consistente.

La verifica di una corretta implementazione di un tipo astratto prevede di:

a) provare che ogni costruttore del tipo inizializzi l'oggetto creato in uno stato interno consistente;
b) provare, per ogni metodo definito per il tipo, che, se l'oggetto è consistente all'inizio della sua esecuzione, lo sia anche alla fine della stessa.

Se le due precedenti condizioni sono verificate, abbiamo la garanzia che un oggetto è sempre in stato consistente, o almeno tale condizione è sempre vera quando nessuna operazione è in esecuzione su di esso. Per cui, si ha la garanzia che, all'atto dell'invocazione di un metodo per operare sull'oggetto, questo si trova in uno stato consistente. Questa verifica, nel caso relativo al precedente esempio della `coda_di_sedici_boolean`, è molto semplice. Infatti, il valore iniziale dei campi `primo`, `ultimo` e `cont` è zero. Quindi la relazione invariante è vera dopo l'inizializzazione e perciò ogni oggetto viene creato in stato consistente. Se supponiamo che un oggetto sia consistente all'inizio dell'esecuzione del metodo `inserisci`, l'oggetto torna consistente alla fine della sua esecuzione. Infatti, nel corpo del metodo vengono incrementati sia il valore del campo `cont` che quello del campo `ultimo`, e quindi i due termini della relazione invariante restano uguali. Analogamente, se la relazione è vera all'inizio dell'esecuzione del metodo `estrai`, l'oggetto torna consistente alla fine della sua esecuzione. Infatti, nel corpo del metodo viene decrementato il valore del campo `cont` e incrementato quello del campo `primo`, e quindi, di nuovo, i due termini della relazione invariante restano uguali.

Con riferimento a quanto già visto per gli oggetti primitivi, anche per gli oggetti astratti l'esecuzione di un'operazione corrisponde a una transizione di stato, in particolare, tra gli stati interni (consistenti) iniziale SI_I e finale SI_F a cui corrispondono, rispettivamente, i due stati esterni iniziale SE_I e finale SE_F. Tenendo conto, però, che le operazioni di tipo astratto non sono atomiche e che operano sui campi componenti l'oggetto, si può notare come questo passi, durante l'esecuzione dell'operazione, attraverso tutta una sequenza di stati interni SI_1, SI_2, ..., SI_n (vedi figura 1.16). Mentre, come già visto, gli stati interni iniziale e finale sono consistenti, normalmente gli stati intermedi sono non consistenti. Ciò dipende dal fatto che, essendo l'operazione non atomica, le singole istruzioni che compongono l'operazione modificano i vari campi dell'oggetto uno alla volta.

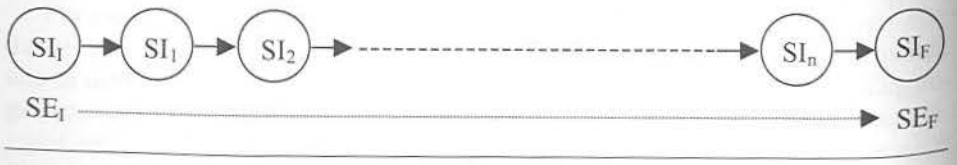


Figura 1.16 Transizioni fra stati interni.

Per esempio, con riferimento a un oggetto di tipo `coda_di_sedici_boolean`, nella figura 1.15 è illustrato uno stato intermedio relativo all'esecuzione dell'operazione `estrai`, quello, in particolare, che corrisponde a un istante in cui è già stato decrementato il campo `cont` ma non è ancora stato incrementato il campo `primo`.

Questa differenza di comportamento tra oggetti primitivi e oggetti astratti non comporta problemi nel contesto della programmazione sequenziale in quanto, per il meccanismo di incapsulamento, tutti gli stati intermedi non sono visibili all'esterno del modulo che definisce il tipo astratto. Nel caso, però, della programmazione concorrente due diversi processi possono operare su uno stesso oggetto astratto. In questo caso, come vedremo, può capitare, se non vengono prese opportune misure, che un processo inizi a operare su un oggetto astratto mentre è in corso una precedente operazione sullo stesso da parte di un altro processo e, quindi, quando lo stesso è in stato inconsistente. Se questa eventualità si verifica, la precondizione della seconda operazione, che prevede di trovare l'oggetto in stato consistente, è violata.

Il problema degli stati inconsistenti è proprio di tutti gli oggetti di tipo astratto, anche di quelli più semplici, costituiti da un unico valore di tipo primitivo, nel caso in cui le operazioni definite per il tipo non siano primitive. Supponiamo di dover definire un tipo `contatore` costituito da un solo campo di tipo `int` e sul quale operare con la sola operazione di `incremento`, che prende il valore del `contatore` e ne restituisce il valore incrementato: normalmente questo tipo è primitivo. L'oggetto è rappresentato da una locazione in memoria (`contatore`) e l'operazione di incremento dall'istruzione macchina `inc contatore`. Ma supponiamo, per esempio, che la macchina su cui si opera non disponga dell'istruzione di incremento di una locazione di memoria, ma soltanto di quella di incremento del valore di un registro di macchina. In questo caso, l'operazione `incremento` deve essere realizzata mediante una semplice funzione che, in un ipotetico linguaggio assembler, potrebbe essere costituita dalle seguenti istruzioni macchina:

```
load R, contatore
inc R
store R, contatore
```

In questo caso, pur non essendo atomica l'operazione di `incremento`, l'oggetto è costituito da un solo campo, la locazione di memoria `contatore`, e ciò può indurre a pensare che non esistano, per esso, stati intermedi inconsistenti. In realtà, anche in questo caso, possiamo definire una relazione invariante per il tipo e verificare che tale relazione non sia soddisfatta durante l'esecuzione dell'operazione di `incremento` nel caso in cui questa, come nel nostro esempio, non sia atomica. Ricordiamo che la relazione invariante per un tipo astratto non è altro che un esempio di asserzione, che caratterizza lo stato del processore prima e dopo l'esecuzione di un'operazione su un oggetto del tipo. Per definire l'invariante di questo semplice tipo possiamo utilizzare, oltre che la stessa variabile `contatore`, anche una *metavariabile* ("meta" in quanto non corrisponde a una variabile del programma, ma caratterizza comunque lo stato di avanzamento dello stesso). Per esempio, possiamo introdurre la metavariabile `ni` che rappresenta, in ogni istante, il numero di volte che l'operazione `incremento` è stata eseguita. Utilizzando questa metavariabile, l'invariante del tipo può essere rappresentata dalla seguente relazione:

```
{ ni==contatore }
```

Questa specifica che, in ogni istante, il valore della variabile `contatore` (che si suppone inizializzata al valore zero), corrisponde al numero di volte che l'operazione `incremento` è stata eseguita. Se l'operazione `incremento` è costituita da un'istruzione macchina, e quindi atomica, la relazione è ovviamente sempre valida. Se però, come nel nostro caso, l'operazione non è atomica, per eseguire l'operazione `incremento` è necessario invocare la corrispondente funzione. In questo caso, possiamo quindi identificare il valore della metavariabile `ni` col numero di invocazioni della funzione `incremento` eseguite. Fra l'istante in cui la funzione viene invocata e l'istante in cui la sua esecuzione termina esiste un intervallo temporale, durante il quale la relazione invariante non è verificata e quindi il `contatore` è in stato inconsistente. Infatti, supponiamo che in un certo istante l'oggetto `contatore` sia in stato consistente; per esempio, supponiamo che la variabile `contatore` abbia un certo valore `n` e, quindi, anche la metaviariabile `ni` valga `n`. Se, in tale istante, viene invocata la funzione `incremento`, il valore di `ni`, dopo l'invocazione, è pari a `n+1` mentre la variabile `contatore` continua ad avere il valore `n`. Soltanto dopo l'esecuzione dell'ultima istruzione della funzione, anche `contatore` assumerà il valore `n+1` e, quindi, la relazione invariante tornerà a essere vera (vedi figura 1.17).

Di nuovo, nel caso della programmazione sequenziale, la presenza di un intervallo temporale durante il quale l'oggetto `contatore` è in stato inconsistente non crea problemi. Nel caso, però, della programmazione concorrente e se l'oggetto `contatore` viene condiviso tra due o più processi, potrebbe accadere che uno di essi inizi l'esecuzione della funzione quando non è ancora terminata quella precedente di un altro processo e, quindi, quando l'oggetto è inconsistente. In particolare, durante l'esecuzione di `incremento`, come illustrato nella figura 1.17, vale la relazione `{ ni==contatore+1 }`, che denota l'inconsistenza del contatore. Se, prima che la funzione termini, un altro processo lo invoca da quel momento vale la relazione

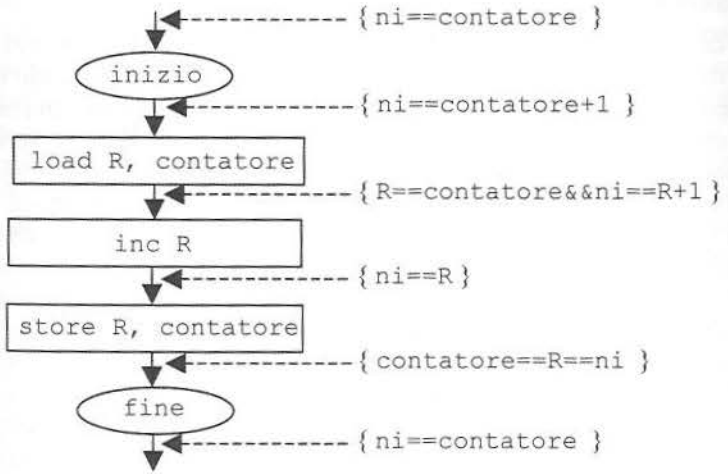


Figura 1.17 Funzione `incremento`.

`{ ni==contatore+2 }` che, ovviamente, invalida tutte le asserzioni previste per una corretta esecuzione della stessa. Per illustrare come ciò possa creare malfunzionamenti, nella figura 1.18 viene illustrato il caso in cui, durante l'esecuzione della funzione `incremento` da parte di un processo, in particolare dopo che lo stesso ne ha eseguito la prima istruzione, un secondo processo inizia a sua volta l'esecuzione della stessa. Nella figura lo stato iniziale del contatore è consistente (si suppone, per esempio, che il comune valore di `ni` e di `contatore` sia `n`).

Ovviamente, l'uso del registro di macchina `R` all'interno della funzione corrisponde a una variabile locale della funzione stessa (ogni processo possiede un proprio insieme di registri). Per chiarezza, nella figura tale variabile è stata denotata, all'interno del corpo della funzione eseguita dai due processi, rispettivamente con `Ra` e `Rb`. La figura, infine, descrive come le esecuzioni delle due funzioni, così concatenate, modifichino gli stati e come, alla fine delle due esecuzioni, non valga più la relazione invariante ma la relazione `{ ni==contatore+1}`, che testimonia la perdita di uno dei due incrementi.

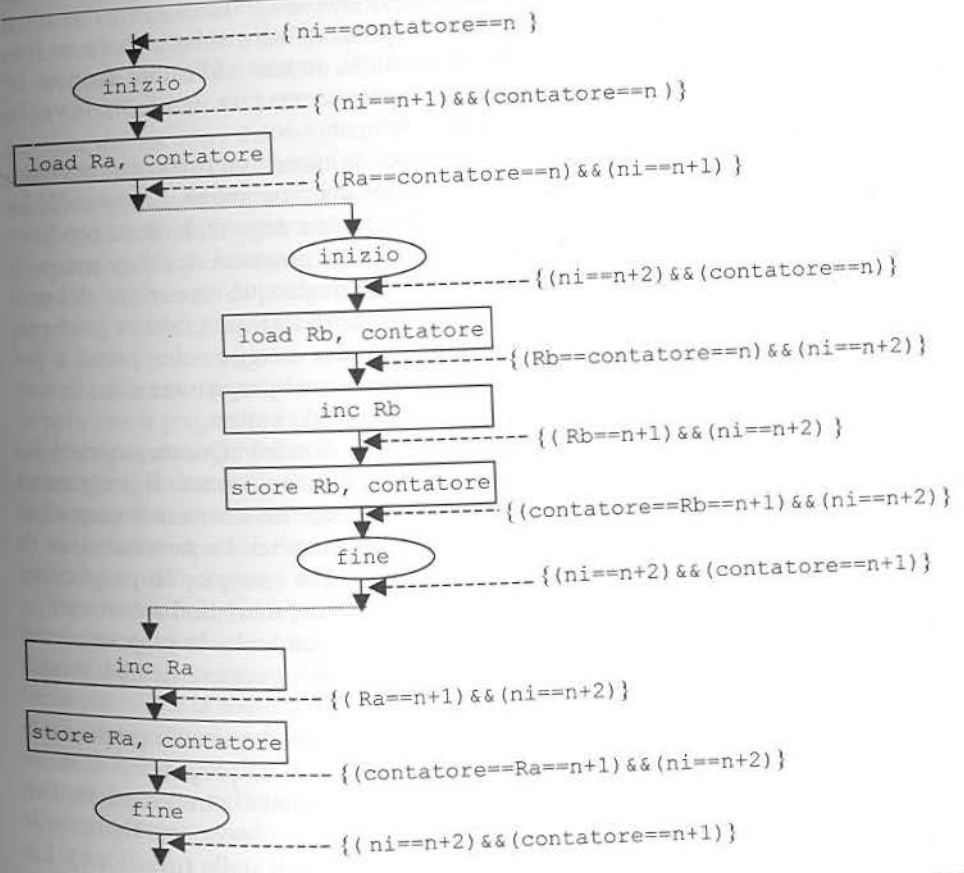


Figura 1.18 Esecuzioni concorrenti.

Possiamo concludere che il fatto che un oggetto astratto non goda della proprietà di atomicità e che, quindi, quando viene utilizzato passi attraverso stati inconsistenti, pone severi problemi di correttezza nel caso di programmi concorrenti. In particolare, ciò avviene quando un processo inizia a operare su un oggetto quando lo stesso è in stato inconsistente poiché sta già operando su di esso un altro processo. La causa del malfunzionamento è da attribuirsi al fatto che le operazioni del tipo sono state realizzate in modo tale che prevedano di trovare l'oggetto in stato consistente all'inizio della loro esecuzione.

1.5 Proprietà di un programma

Come è stato mostrato in un precedente paragrafo, la verifica di correttezza di un programma costituisce una delle fasi più importanti in tutto il ciclo della sua vita. Per il momento, abbiamo accennato alla correttezza di un programma solo dal punto di vista funzionale. In realtà, almeno per certe categorie di programmi, la correttezza può prevedere anche altri aspetti, non necessariamente funzionali. Per esempio, in un sistema in tempo reale è necessario che un programma fornisca i propri risultati entro determinati limiti temporali (*deadline*), altrimenti tali risultati, anche se funzionalmente corretti, possono essere di scarsa utilità, se non addirittura dannosi. In generale, quindi, la verifica di correttezza di un programma può significare la verifica che esso goda di certe proprietà (funzionali, temporali ecc.).

Con il termine di *proprietà di un programma* si intende un particolare attributo che deve essere vero per ogni possibile esecuzione del programma. Lamport [2] ha classificato le proprietà di un programma in due distinte categorie, indicate con i termini di proprietà di *safety* e proprietà di *liveness*. Una proprietà di *safety* asserisce che niente di erroneo si può verificare durante una qualunque esecuzione del programma, cioè nessun processo relativo all'esecuzione di quel programma potrà mai entrare in uno stato erroneo. Una proprietà di *liveness* asserisce che, prima o poi (*eventually*), ogni processo relativo all'esecuzione di quel programma entra in uno stato corretto, uno stato cioè in cui le variabili hanno i valori attesi.

La *correttezza parziale* è un esempio di proprietà di safety. Questa proprietà asserisce che, se il programma termina, allora i risultati sono corretti. Il programma potrebbe non terminare e, in questo caso, non produrrebbe nessun risultato, ma sicuramente non potrà mai terminare producendo risultati erronei. La *terminazione* di un programma è un esempio di proprietà di liveness. Per esempio, un programma che entra in un ciclo da cui non esce non gode di questa proprietà. La *correttezza totale* caratterizza un programma se lo stesso gode di entrambe le proprietà, cioè ogni processo di un programma totalmente corretto termina sempre e termina producendo i risultati corretti.

La correttezza parziale e la terminazione sono le due più importanti proprietà di safety e liveness di un programma sequenziale. Nel caso di un programma concorrente, oltre a queste, è però necessario verificare anche molte altre proprietà. Per esempio, due importanti proprietà di safety di un programma concorrente riguardano la garanzia di *mutua esclusione* e l'assenza di condizioni di stallo (*deadlock*). La prima stabilisce che, durante l'esecuzione del programma, nessun processo potrà mai accedere a un oggetto quando lo stesso è in stato inconsistente, cioè quando sullo stesso sta operando un diverso processo. La seconda stabilisce che non si può mai verificare uno stato in cui i processi sono bloccati in attesa di una condizione che non si può verificare.

Un esempio di proprietà di liveness di un programma concorrente è quella che garantisce che un processo che voglia accedere a un oggetto condiviso, operando quindi in mutua esclusione sullo stesso rispetto ad altri processi, sia abilitato prima o poi a operarci. Più in generale appartiene alle proprietà di liveness l'assenza di ogni possibilità di *starvation*: con questo termine si intende lo stato in cui un processo P è in attesa, insieme ad altri processi, che si verifichi una certa condizione, la quale abilita uno solo dei processi in attesa a proseguire. Si ha starvation di P se, fra i processi in attesa, viene effettuata una scelta che non prende mai in considerazione P.

1.6 Sommario

Il capitolo si sofferma, nella parte iniziale, sulla definizione del concetto di processo, sulle sue proprietà e sulla sua relazione con il programma. Vengono precisate le modalità di rappresentazione dei processi, utilizzando la sua traccia di esecuzione sul processore e un grafo a ordinamento totale. Poiché un processo è l'esecuzione di un programma con un particolare insieme di dati di ingresso, il problema della verifica della correttezza di un processo si riconduce alla verifica della correttezza del programma da cui esso trae origine. Viene introdotto un metodo formale di verifica, realizzato alla fine degli anni '60, basato sulla logica dei predicati e viene mostrato come esso possa essere utilizzato per la verifica dei programmi.

Precisata la differenza tra oggetti primitivi e oggetti astratti, viene sottolineata l'importanza di quest'ultimi nella strutturazione dei programmi e viene messa in evidenza la differenza di comportamento rispetto agli oggetti primitivi per quanto riguarda la verifica della loro consistenza, cioè il rispetto della condizione di invarianza. Il fatto che un oggetto astratto non goda della proprietà di atomicità e quindi, quando utilizzato, passi attraverso stati inconsistenti, pone seri problemi di correttezza nel caso di programmi concorrenti. In particolare ciò avviene quando un processo inizia a operare su un oggetto in stato inconsistente, in quanto su di esso sta operando un altro processo. Come si vedrà nei capitoli successivi, risulta quindi necessario proteggere l'accesso a un oggetto astratto da parte di un processo, impedendo che altri vi possano operare finché esso non abbia terminato la sua operazione (mutua esclusione nell'accesso a risorse comuni).

In conclusione, viene precisato come la verifica della correttezza di un programma possa equivalere ad accertarsi che esso goda di certe proprietà e come queste proprietà possano essere classificate nelle due categorie, *safety* e *liveness*. La correttezza parziale e la terminazione sono, rispettivamente, le due più importanti proprietà di safety e liveness di un programma sequenziale. Nel caso di programmi concorrente oltre a queste due, devono essere aggiunte altre proprietà, quali per esempio la *mutua esclusione*, l'assenza di condizioni di stallo (*deadlock*) e di attesa indefinita di un processo in grado di proseguire la sua esecuzione (*starvation*).

1.7 Note bibliografiche

Un'ottima introduzione alle metodologie della programmazione sequenziale, ai concetti di algoritmo, programma, processo nonché di tipo di dati è presentata da Wirth [3] [1].

Per quanto riguarda i meccanismi di astrazione, vanno citati tra i più significativi i lavori fondamentali di Parnas sulla metodologia dell'information hiding [4] [5] [6] e i lavori di Liskov e Zilles [7] [8] sulla definizione di tipo di dato astratto.

Un esauriente trattazione sul ruolo dell'astrazione nei linguaggi di programmazione, con riferimento sia all'astrazione dei dati che a quella di controllo è contenuta nel testo di Ghezzi [9].

Robert Floyd [10] è stato tra i primi a proporre una tecnica basata sui predicati per provare la correttezza dei programmi. Ispirato dal lavoro di Floyd, Hoare [11] ha sviluppato la prima logica formale per provare le proprietà di correttezza parziale dei programmi sequenziali. Sempre Hoare [12] ha esteso la sua logica di correttezza parziale per i programmi sequenziali includendo regole di inferenza per la concorrenza e la sincronizzazione.

I termini safety e liveness sono stati introdotti da Leslie Lamport [2].

Il problema di provare proprietà di correttezza formale per programmi concorrenti è trattato in [13] [14] [15].

2
Programmazione concorrente e distribuita

La programmazione concorrente, intesa come l'insieme delle tecniche, delle metodologie e degli strumenti necessari per fornire il supporto all'esecuzione di applicazioni software come un insieme di attività svolte simultaneamente, nasce agli inizi degli anni '60. Inizialmente è stata introdotta come una tecnica da utilizzare nel progetto di un sistema operativo, con l'obiettivo di sfruttare la disponibilità, nei grossi calcolatori dell'epoca (*main frame*), dei *canali di I/O*, appositi dispositivi utilizzati per controllare le operazioni d'ingresso/uscita: si trattava di veri e propri processori special purpose, dotati di un proprio linguaggio macchina e in grado di fornire il supporto all'esecuzione di programmi speciali, dedicati a controllare il trasferimento dei dati (*programmi di canale*) ed eseguiti concorrentemente col programma in esecuzione sulla CPU. Successivamente, l'introduzione del meccanismo di interruzione e dei canali di accesso diretto alla memoria (DMA) ha fornito il supporto hardware all'introduzione della tecnica della multiprogrammazione. Con tale tecnica, la programmazione concorrente è stata progressivamente sviluppata, in stretto rapporto con lo sviluppo dei sistemi operativi. La multiprogrammazione, prima forma di programmazione concorrente, fornisce il supporto per l'esecuzione intercalata di programmi diversi (*interleaving*). Come è noto dalla teoria dei sistemi operativi, la multiprogrammazione è una tecnica che consente di estendere le potenzialità di una macchina fisica monoelaboratore in modo tale da generare una macchina astratta che dispone di più processori virtuali, uno per ogni processo che il sistema può supportare. La concorrenza tra le esecuzioni dei singoli processori virtuali viene simulata mediante il meccanismo di schedulazione, proprio del nucleo del sistema operativo, che, assegnando il processore reale a ciascun processore virtuale per brevi intervalli temporali, garantisce che tutti procedano concorrentemente e in modo tale da rispettare i requisiti propri del sistema operativo. Tali requisiti sono ovviamente diversi a seconda della tipologia del sistema (sistemi *batch*, *time-sharing* o *real-time*).

Successivamente, con l'introduzione di architetture multielaboratore, la concorrenza ha usufruito anche del supporto hardware, che ha consentito l'esecuzione realmente in parallelo (*overlapping*) di più processi sequenziali. Infine, l'introduzione delle reti di calcolatori e dei sistemi distribuiti ha consentito di estendere la programmazione concorrente in modo tale da permettere l'esecuzione di più processi contemporaneamente su processori distribuiti su una rete (*programmazione distribuita*).

Oggi le tecniche e le metodologie proprie della programmazione concorrente e distribuita trovano utilizzazione anche nel progetto e realizzazione di varie tipologie di sistemi applicativi, pur continuando a essere il supporto indispensabile al progetto e alla realizzazione dei sistemi operativi. È per questo che i moderni linguaggi di programmazione, come vedremo, sono spesso dotati di un insieme di costrutti linguistici dedicati a fornire il supporto a questo tipo di tecniche e di metodologie.

Di seguito, verranno presentati alcuni semplicissimi esempi di concorrenza, con lo scopo di introdurre le principali problematiche inerenti la programmazione concorrente e distribuita.

2.1 Elaborazioni concorrenti

Precedentemente abbiamo identificato col termine di processo (o thread) l'attività del processore quando esegue un programma sequenziale. Al fine di evitare ambiguità, nel seguito identificheremo col termine di *elaborazione* (*computation*) l'attività svolta da una macchina composta da più elaboratori (sia essa fisica o astratta) quando la stessa esegue un programma concorrente.

Come già indicato, nel caso dei processi utilizzeremo i grafi di precedenza, anche per rappresentare elaborazioni concorrenti. In questo caso, poiché l'attività svolta dalla macchina non è più sequenziale, il grafo non sarà più a ordinamento totale, ma bensì a ordinamento parziale; in questo modo sarà possibile evidenziare che tra certe azioni dell'elaborazione complessiva non esiste più un vincolo di stretta precedenza temporale.

Per chiarire questi aspetti facciamo riferimento a un semplice esempio. Supponiamo di dover scrivere le istruzioni che codificano l'algoritmo di valutazione di una semplice espressione aritmetica, per esempio:

```
(a-b)*(c+d)+(e*f)
```

Pur senza scrivere il programma, è facile rendersi conto che il grafo di precedenza del corrispondente processo assume l'aspetto riportato nella figura 2.1, dove `r1`, `r2`, `r3` e `r4` rappresentano risultati parziali e `ris` il risultato finale. Abbiamo adottato il semplice algoritmo sequenziale basato sull'usuale tecnica di scansione dell'espressione da sinistra verso destra e svolto prima le operazioni all'interno delle parentesi e poi quelle all'esterno delle stesse, rispettando la priorità delle operazioni, ovvero prima moltiplicazioni e divisioni e successivamente somme e sottrazioni.

La logica del problema non impone, tuttavia, un ordinamento totale fra le operazioni da eseguire. Per esempio, è indifferente che `(a-b)` venga eseguito prima di `(c+d)` o viceversa; è invece necessario che entrambe le operazioni siano state eseguite prima di poter eseguire il prodotto fra i loro risultati. Ci troviamo quindi di fronte a un caso in cui l'elaborazione complessiva può essere più convenientemente

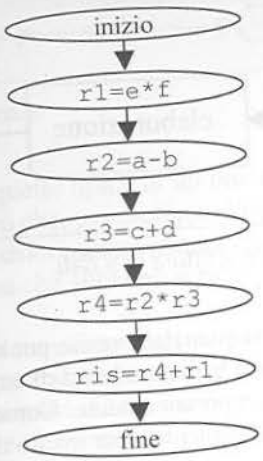


Figura 2.1 Grafo di precedenza a ordinamento totale.

rappresentata mediante un grafo di precedenza a ordinamento parziale, come quello illustrato nella figura 2.2, che mette in evidenza esclusivamente le precedenze tra gli eventi dell'elaborazione che è necessario rispettare in base alla semantica dell'espressione stessa.

Dal grafo risulta che certi eventi dell'elaborazione sono tra loro scorrelati da qualunque relazione di precedenza temporale; ciò significa che il risultato dell'elaborazione è indipendente dall'ordine con cui gli eventi avvengono. Imporre che l'elaborazione abbia la struttura di un processo sequenziale, come quello illustrato nella figura 2.1, è quindi un vincolo dettato esclusivamente dalla natura sequenziale del processore.

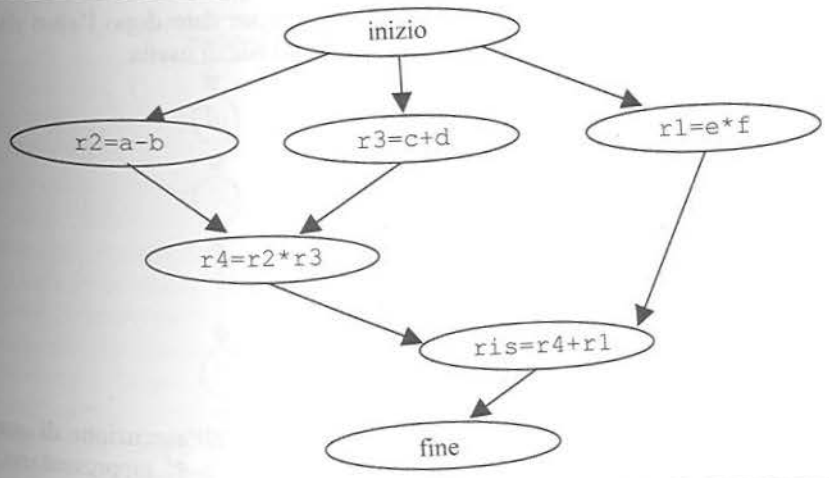


Figura 2.2 Grafo di precedenza a ordinamento parziale.

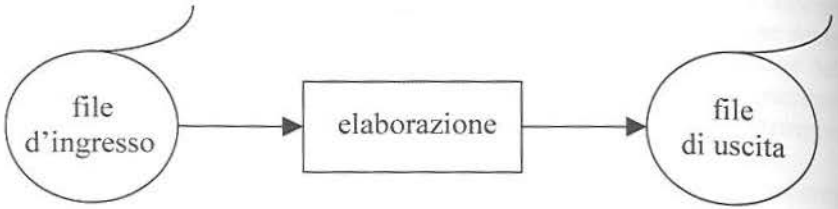


Figura 2.3 Lettura, elaborazione e scrittura di N dati.

Per eseguire elaborazioni non sequenziali, come per esempio quella illustrata nella figura 2.2, è necessario ipotizzare la disponibilità di una macchina (astratta) in grado di eseguire più operazioni contemporaneamente. Come già indicato precedentemente, nel seguito faremo sempre riferimento a un livello di concorrenza che è quello tipico di una macchina che abbia a disposizione più processori e che sia, quindi, in grado di eseguire elaborazioni composte da più processi sequenziali eseguiti concorrentemente, ciascuno sul proprio elaboratore.

Illustriamo ulteriormente questi concetti con un semplice esempio. Supponiamo di dover scrivere un programma che deve leggere i dati contenuti in un file sequenziale, elaborare tali dati, per esempio eseguendo su ciascuno di essi una determinata funzione e, alla fine, produrre con i risultati delle elaborazioni un file sequenziale contenente i dati finali (vedi figura 2.3). Supponiamo inoltre che `N` siano i dati contenuti nel file d'ingresso e quindi anche il numero dei dati elaborati, contenuti nel file di uscita. Per semplicità, supponiamo infine che tutti i dati contenuti nel file d'ingresso, e anche tutti i risultati, siano di uno stesso tipo generico `T`. Tralasciando alcuni inutili dettagli, il seguente frammento di codice potrebbe rappresentare un esempio di programma sequenziale che risolve il problema. Le funzioni `leggi()`, `elabora()` e `scrivi()`, che si suppone siano state precedentemente dichiarate, sono quelle che servono, rispettivamente, per leggere un dato dopo l'altro dal file d'ingresso, elaborarlo e scrivere il relativo risultato sul file di uscita.

```
{ ..........
  T buffer;
  ..........
  for (int i = 0; i < N; i ++){
     leggi (buffer);
     elabora (buffer);
     scrivi (buffer);
  }
  ..........
}
```

Il grafo di precedenza, che rappresenta il processo relativo all'esecuzione di questo programma, è quello riportato nella figura 2.4, dove L_i, E_i e S_i rappresentano, rispettivamente, gli eventi relativi alle esecuzioni della lettura, elaborazione e scrittura del dato `i`-esimo.

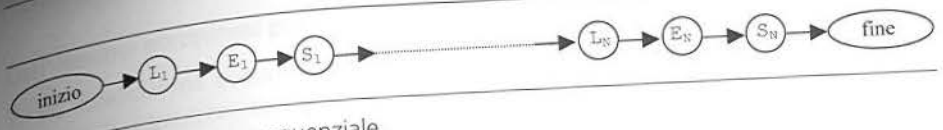


Figura 2.4 Elaborazione sequenziale.

Questo grafo di precedenza è quello tipico di un processo sequenziale ed è quindi a ordinamento totale. Il fatto però che, per esempio, l'operazione relativa alla seconda lettura debba essere preceduta dalla prima scrittura dipende esclusivamente dalla natura sequenziale del programma che abbiamo scritto e non dalla natura del problema da risolvere.

Proviamo adesso a riconsiderare il precedente problema svincolandoci dal particolare linguaggio con cui scrivere il programma e dal numero di processori presenti sulla macchina che lo deve eseguire. Manteniamo però i vincoli legati alla sequenzialità dei due file d'ingresso e di uscita, che impongono la lettura e la scrittura dei dati uno alla volta. Per esempio, per quanto riguarda la lettura non è possibile leggere il secondo dato se non dopo che è stato letto il primo, e così via. In questo caso, potremmo dedicare due processi sequenziali autonomi rispettivamente alla lettura dei dati e alla scrittura dei risultati e un terzo processo a elaborare un dato alla volta producendo il relativo risultato, come indicato nella figura 2.5.

I tre processi di lettura, elaborazione e scrittura non sono però tra loro indipendenti. Infatti, affinché il processo di elaborazione possa elaborare il primo dato è necessario che il processo di lettura abbia trasferito il dato in memoria e cioè che sia terminata la prima lettura. Analogamente, affinché sia possibile scrivere il primo risultato, questo deve essere disponibile, cioè è necessario che sia terminata la prima elaborazione. È quindi necessario imporre alcuni vincoli di precedenza tra le azioni dei tre processi, come indicato nella figura 2.6 dove viene illustrato il grafo di precedenza della elaborazione complessiva.

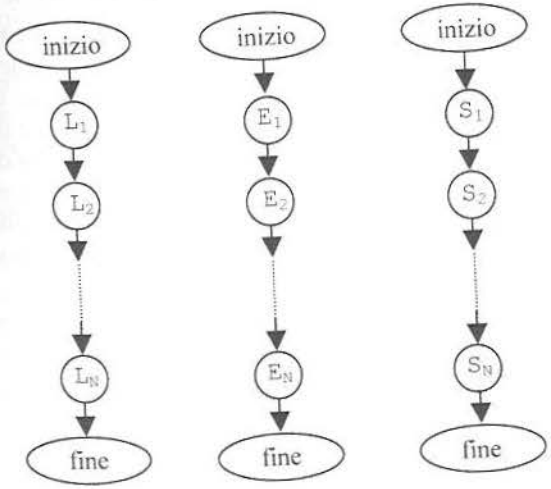


Figura 2.5 Processi di lettura elaborazione e scrittura.

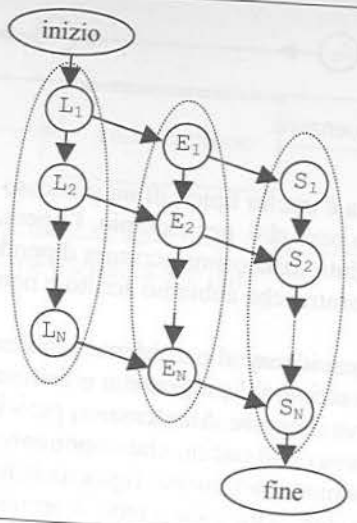


Figura 2.6 Elaborazione complessiva.

Il grafo di precedenza riportato nella figura 2.6 è a ordinamento parziale, in quanto esistono azioni, come per esempio S_1 e L_2, che non sono correlate tra loro da relazioni di precedenza temporale. Ciò significa che possono avvenire in qualsiasi ordine, l'una prima dell'altra o viceversa, oppure ancora contemporaneamente. Tale grafo descrive quindi un'elaborazione concorrente.

Una caratteristica peculiare di un grafo che rappresenta un'elaborazione concorrente è quella di contenere degli archi, che denotano vincoli di precedenza temporale tra azioni di processi diversi; per esempio, l'arco tra l'azione di lettura di un generico dato e quella di elaborazione dello stesso, e l'arco tra quest'ultima azione e quella di scrittura del relativo risultato (vedi figura 2.7). Questi archi, che impongono precedenze temporali tra azioni di processi diversi, verranno identificati come *vincoli di sincronizzazione*. La necessità di tali vincoli deriva dal fatto che i tre processi non sono tra loro indipendenti, ma bensì interagenti l'uno con l'altro. In particolare, l'interazione è relativa al fatto che fra i processi componenti avviene uno scambio di dati. È il processo di lettura che fornisce i dati da elaborare a quello di elaborazione, ed è quest'ultimo che fornisce i risultati da scrivere al processo di scrittura. Non essendo a priori note le velocità dei processori, i processi che questi sviluppano sono asincroni. È quindi ovvio che nel momento in cui due processi devono interagire sia necessario sincronizzare le loro velocità.

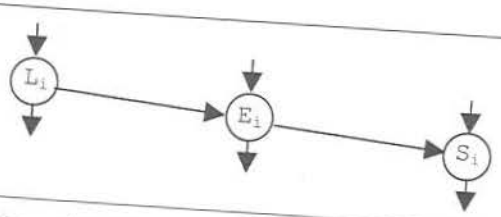


Figura 2.7 Vincoli di sincronizzazione.

Dall'esempio precedente possiamo derivare alcune considerazioni generali:

1. un'elaborazione concorrente è composta da un insieme di processi sequenziali eseguiti concorrentemente. Una caratteristica peculiare di questi processi è quella di essere tra loro *asincroni*. Questa caratteristica deriva dal fatto che le velocità, con cui le singole azioni di ciascun processo vengono eseguite, non sono note a priori e comunque non sono generalmente uguali tra loro. Per esempio, la velocità con cui un processo legge i caratteri, battuti dall'utente sulla tastiera del proprio calcolatore, dipende dalla velocità con cui egli batte sui tasti della tastiera stessa.
2. i processi componenti un'elaborazione concorrente non sono tra loro indipendenti; in quanto componenti della stessa elaborazione, sono tra loro *interagenti*. Tutte le volte che si verifica un'interazione tra processi, gli stessi devono sincronizzare le proprie velocità.
3. per descrivere un algoritmo che durante la sua esecuzione dia luogo a un'elaborazione concorrente, è necessario un linguaggio di programmazione che, contrariamente a un tradizionale linguaggio sequenziale, deve consentire al programmatore di descrivere il programma non come un'unica sequenza di istruzioni, ma come più sequenze da eseguire concorrentemente tra di loro (*linguaggi concorrenti*).
4. per eseguire un programma concorrente è necessaria una macchina in grado di sviluppare più processi sequenziali contemporaneamente, cioè una macchina che disponga di più processori fisici o virtuali (*macchina concorrente*).

2.2 Interazioni tra processi

Come è stato messo in evidenza nel precedente paragrafo, un'elaborazione concorrente si contraddistingue, rispetto a un'elaborazione sequenziale, per essere composta da più processi sequenziali eseguiti concorrentemente; identificheremo tali processi come i *componenti* dell'elaborazione complessiva. Tali componenti concorrono alla produzione dei risultati dell'elaborazione e, per questo scopo, è necessario prevedere che in certi istanti i processi componenti interagiscano tra loro.

Normalmente, le possibili interazioni tra processi vengono classificate in tre categorie diverse: *cooperazione*, *competizione* e *interferenza*.

2.2.1 Cooperazione

Il primo tipo di interazione, la cooperazione, corrisponde in senso lato a uno scambio di informazioni tra processi. Un esempio di scambio di informazioni è quello messo in evidenza nel precedente paragrafo. In generale, col termine di cooperazione vengono identificate tutte quelle interazioni che vengono spesso indicate come "prevedibili e desiderate", nel senso che la loro presenza è necessaria in quanto insita nella logica del particolare programma: è il caso dello scambio di informazioni. Infatti, se un processo P invia informazioni a un processo Q, l'uno non avrebbe senso in assenza dell'altro.

Il caso più semplice di cooperazione riguarda il semplice scambio di segnali temporali tra processi. Un segnale temporale rappresenta un puro segnale di sincronizza-

zione che viene scambiato tra due processi. Ciò avviene tutte le volte che è necessario garantire che una certa operazione op_2 sia eseguita da un processo Q solo dopo che una diversa operazione op_1 sia stata eseguita da un altro processo P.

Un caso tipico di scambio di segnali è quello che avviene nell'ambito di un sistema in tempo reale dove l'applicazione è spesso costituita da un insieme di processi periodici. Per esempio, in un sistema per il controllo di un'apparecchiatura, o di un impianto industriale, il compito del sistema è quello di monitorare l'andamento di alcune grandezze fisiche come temperature, pressioni, portate ecc. In questi casi, è possibile strutturare l'applicazione come un insieme di processi sequenziali, ciascuno dei quali è dedicato a monitorare una delle grandezze fisiche d'interesse. La struttura di ogni processo è normalmente molto semplice: si tratta di leggere il valore della grandezza, verificare se tale valore rientra nell'intervallo di valori ammessi e, in caso contrario, intervenire per mezzo di appositi attuatori per riportare il valore nell'intervallo desiderato. Tutti i processi applicativi avanzano concorrentemente. Ogni processo è inoltre periodico cioè, una volta terminata la propria esecuzione, il processo viene di nuovo attivato con una cadenza che dipende dalla costante tempo della grandezza da controllare, al fine di mantenerla continuamente sotto controllo. Una possibile realizzazione di questo schema è quella che corrisponde a un'applicazione costituita da un insieme di processi applicativi, ciascuno dei quali viene periodicamente attivato tramite il timer del sistema. Per esempio, nella figura 2.8 viene rappresentato il grafo di precedenza di una semplice applicazione costituita da due processi applicativi indicati come processo A e processo B: il primo è costituito da tre azioni (a_1, a_2 e a_3), da eseguire con periodicità di un secondo, l'altro da quattro azioni (b_1, b_2, b_3 e b_4), da eseguire ogni quattro secondi. Il processo riportato all'estrema sinistra rappresenta il `timer` del sistema: le relative azioni (t_1, t_2, t_3, ...) rappresentano i secondi scanditi dall'orologio e corrispondono alle esecuzioni del driver delle interruzioni del `timer`. Gli archi, che collegano i nodi del processo `timer` a quelli dei processi A e B, rappresentano i vincoli di precedenza corrispondenti ai segnali di attivazione. Per cui, per esempio, la seconda attivazione del processo B (azione b_1'') non può che avvenire dopo che è stato scandito dal `timer` il quarto secondo (azione t_4).

Questa tipologia di interazioni, come mostrato dall'esempio, si estrinseca nello scambio di segnali di sincronizzazione tra processi, segnali che sul grafo di precedenza sono rappresentati dagli archi che collegano le azioni del processo `timer` alle azioni dei processi applicativi, imponendo precisi vincoli temporali. Ciò significa che il linguaggio di programmazione con cui esprimere il programma concorrente deve possedere appositi costrutti in grado di esplicitare questi vincoli di sincronizzazione e, al tempo stesso, la macchina concorrente, sulla quale il programma dovrà essere eseguito, deve possedere appositi meccanismi primitivi in grado di fornire il supporto a tali costrutti.

Come già indicato, un esempio più generale di cooperazione corrisponde a quello messo in evidenza nel precedente paragrafo, dove i processi non si scambiano soltanto dei segnali di sincronizzazione ma più in generale dei dati. È il caso del processo di lettura che *invia* il dato letto al processo di elaborazione affinché questo possa svolgere il proprio compito, cioè elaborare il dato stesso. Anche nel caso di questo tipo di interazione, è necessario imporre vincoli di precedenza analoghi al caso precedente.

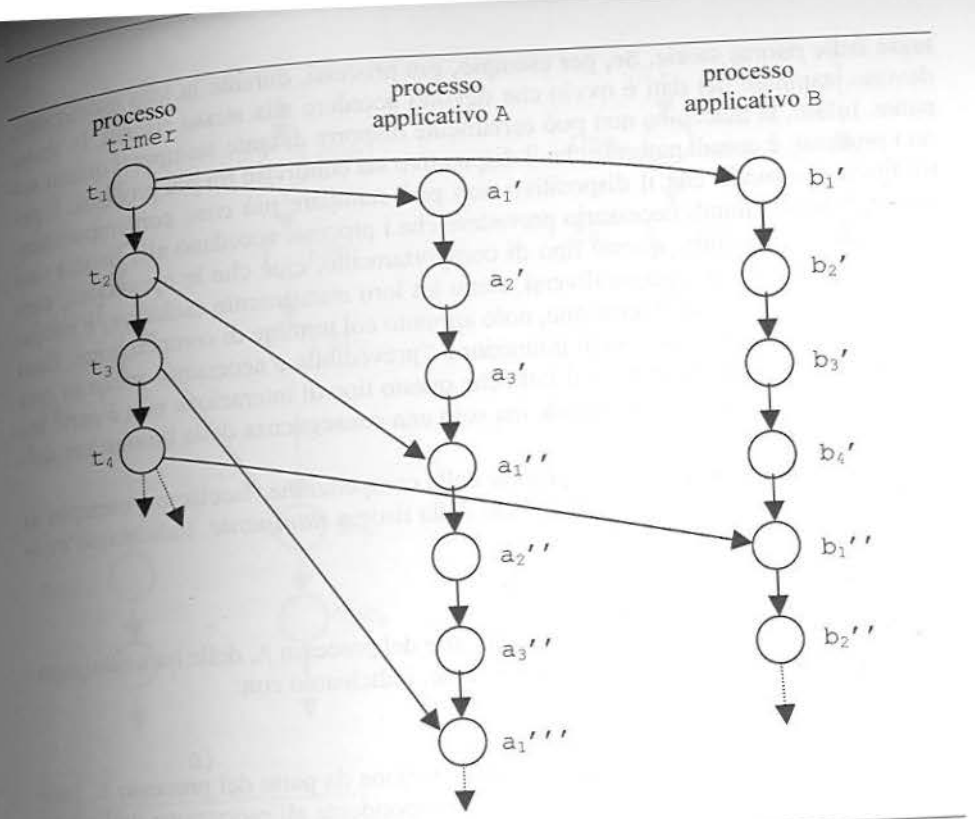


Figura 2.8 Esempio di scambio di segnali temporali.

Infatti, la ricezione del dato da elaborare non può che avvenire successivamente all'invio del dato stesso. È quindi evidente che, per consentire anche questo tipo più generale di cooperazione, deve essere possibile la specifica di vincoli di sincronizzazione, ma allo stesso tempo devono essere disponibili, sia a livello di linguaggio di programmazione che a livello di macchina concorrente, meccanismi per la comunicazione tra processi (*Inter Process Communication mechanism*).

2.2.2 Competizione

La seconda tipologia di interazioni, la competizione, riguarda la necessità di coordinare gli accessi, da parte di processi diversi, alle stesse risorse condivise. Nel seguito verrà chiarito con maggiori dettagli il concetto di risorsa; fin da ora però, possiamo indicare con tale termine un qualunque oggetto su cui un processo deve operare per portare a termine il proprio compito. Soddisfano questa generica definizione tanto le risorse fisiche della macchina, come per esempio i dispositivi periferici per l'ingesso/uscita dei dati, quanto oggetti logici, come strutture dati o, più in generale, istanze di tipi di dati astratti.

Il fatto che le risorse fisiche messe a disposizione da parte della macchina concorrente debbano essere, in generale, condivise è un'ovvia conseguenza della limita-

tezza delle risorse stesse. Se, per esempio, più processi, durante la loro esecuzione, devono stampare dei dati è ovvio che devono accedere alla stessa risorsa, la stampante. Infatti, la macchina non può certamente disporre di tante stampanti quanti sono i processi: è quindi naturale che il dispositivo sia condiviso tra più processi. È però altrettanto ovvio che il dispositivo non può stampare più cose contemporaneamente; si rende quindi necessario prevedere che i processi accedano alla risorsa uno alla volta. Per garantire questo tipo di comportamento, cioè che le operazioni, eseguite sulla risorsa da processi diversi, siano tra loro mutuamente esclusive, è necessario imporre un tipo di interazione, noto appunto col termine di competizione. Esso è indicato anche come un tipo di interazione "prevedibile e necessaria anche se non desiderata", intendendo con ciò il fatto che questo tipo di interazione non è parte integrante della logica del programma, ma solo una conseguenza della limitatezza delle risorse disponibili.

Per illustrare le caratteristiche proprie della competizione, facciamo l'esempio di due processi A e B che condividono l'uso della risorsa *stampante*. Indichiamo inoltre con:

$Sa_1, Sa_2, ..., Sa_n$

le azioni corrispondenti all'esecuzione, da parte del processo A, delle istruzioni componenti la funzione di `stampa`. Analogamente, indichiamo con:

$Sb_1, Sb_2, ..., Sb_m$

le azioni corrispondenti all'esecuzione della funzione da parte del processo B. Indichiamo, infine, con Opa_u (Opb_u) l'azione corrispondente all'esecuzione dell'ultima istruzione da parte di A (B) prima dell'esecuzione della stampa e con Opa_p (Opb_p) l'azione corrispondente all'esecuzione della prima istruzione dopo la stampa. Per garantire una corretta competizione sulla stampante, è necessario assicurare la mutua esclusione fra le esecuzioni della funzione `stampa` da parte dei due processi. Ciò significa che se, per esempio, è il processo B che inizia per primo l'esecuzione di `stampa`, allora l'inizio della stessa esecuzione da parte di A deve avvenire dopo che è terminata quella da parte di B (vedi figura 2.9a). È altrettanto accettabile però una elaborazione dove è il processo A che inizia per primo l'esecuzione di `stampa`. In questo secondo caso, è l'esecuzione di `stampa` da parte di B che deve iniziare solo dopo che è terminata quella da parte di A (vedi figura 2.9b).

Come nel caso della cooperazione, anche una corretta competizione presuppone che i processi interagenti sincronizzino le loro velocità nel momento in cui entrano in competizione tra di loro. La differenza fra le due tipologie di interazione è legata al fatto che, mentre una cooperazione prevede una sincronizzazione fissa (l'operazione di invio di un'informazione da parte di un processo deve sempre precedere la ricezione da parte di un altro), nel caso della competizione sono consentite due diverse forme di sincronizzazione, come illustrato nella figura 2.9. Entrambe le forme di sincronizzazione sono corrette ed è possibile che, durante una particolare esecuzione del programma, si verifichi una delle due forme mentre, durante una seconda esecuzione, si verifichi l'altra. In particolare, ciò accade se, cambiando il rapporto di velocità fra i processi, il processo che inizia per primo la `stampa` è diverso da quello relativo alla prima esecuzione.

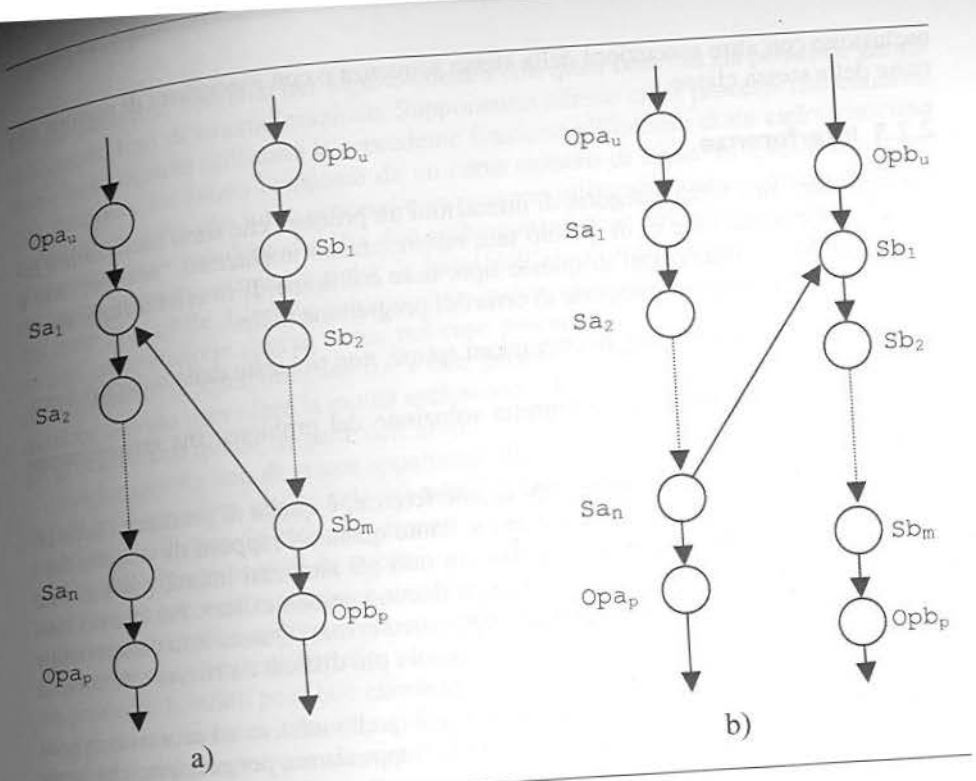


Figura 2.9 Esempio di competizione.

Essendo diversa la natura delle due tipologie di interazione, è quindi naturale che siano diverse anche le modalità con cui i processi interagenti si sincronizzano. La sincronizzazione legata a problemi di cooperazione viene spesso indicata anche come *sincronizzazione diretta* o *esplicita*, mentre quella legata a problemi di competizione come *sincronizzazione indiretta* o *implicita*. Nel seguito, illustrando i costrutti linguistici che un linguaggio concorrente offre per consentire la specifica dei vari tipi di interazione, vedremo come, in certi casi, siano riservati costrutti diversi per le due tipologie di sincronizzazione.

La mutua esclusione degli accessi a una risorsa condivisa, è necessaria non solo nel caso di risorse fisiche ma anche nel caso in cui la risorsa condivisa sia una risorsa logica, come per esempio l'istanza di un tipo di dati astratto. Abbiamo già visto come, quando un processo opera su un oggetto astratto, questo passi attraverso una sequenza di stati interni inconsistenti. È quindi evidente che, se un processo P inizia a operare su un oggetto O mentre su di esso sta operando un altro processo Q e, quindi, mentre O è in stato inconsistente, viene violata la precondizione necessaria per la corretta operazione su O da parte di P. Quindi, anche in questo caso, è necessaria una sincronizzazione indiretta tra i processi che condividono l'uso di O.

L'insieme delle operazioni, che possono essere eseguite su un oggetto condiviso tra più processi, viene anche indicato come una *classe di sezioni critiche*. Il termine *sezione critica* indica una sequenza di istruzioni, che deve essere eseguita in mutua

esclusione con altre esecuzioni della stessa sequenza o con esecuzioni di sezioni critiche della stessa classe.

2.2.3 Interferenze

Esiste, infine, una terza categoria di interazioni tra processi che sono ascrivibili a errori di programmazione e, in quanto tali, rappresentano interazioni "non previste e non desiderate". Interazioni di questo tipo, note col nome di interferenze, sono riconducibili alle seguenti categorie di errori di programmazione:

a) presenza, nel programma, di interazioni spurie, non richieste dalla natura del problema;

b) interazioni necessarie per la corretta soluzione del problema ma erroneamente programmate.

Una caratteristica, comune a tutti i tipi di interferenze, è quella di produrre i loro effetti erronei sui risultati dell'elaborazione soltanto quando i rapporti di velocità fra i processi assumono ben determinati valori: in tutti gli altri casi infatti, pur essendo presenti nel programma, questi errori non producono nessun effetto. Per questo motivo sono noti anche col termine di *time-dependent errors*. Questa loro caratteristica li rende particolarmente insidiosi, in quanto ancora più difficili da rilevare durante la fase di test rispetto ad altri errori di programmazione.

Un tipico esempio di interferenze del tipo a) è quello relativo ad accessi non previsti a una risorsa `R` da parte di un processo P_i. Supponiamo, per esempio, che la risorsa `R` sia necessaria esclusivamente al processo P_j e che nessun altro processo debba quindi operare su di essa. In questo caso P_j può operare su `R` senza preoccuparsi di accedervi in mutua esclusione, non dovendo competere con nessun altro processo. Può accadere, però, che nello scrivere il programma relativo al processo P_i si commetta un errore, inserendo nel suo codice un'operazione su `R`. Tale operazione non prevista può generare un errore nel risultato dell'intero programma, se l'accesso a `R` da parte di P_i avviene concorrentemente con quello corretto da parte di P_j. Infatti, per quanto precedentemente osservato, i due accessi concorrenti produrrebbero errori dovuti agli stati inconsistenti di R. Analogamente, anche se i due processi operano su R l'uno indipendentemente dall'altro, se P_i è il primo a operare su R e tale operazione modifica lo stato della risorsa, successivamente P_j, quando effettuerà il proprio accesso alla risorsa, la troverà in uno stato diverso da quello in cui l'avrebbe dovuta trovare. Ovviamente questo errore non produrrebbe nessun effetto se l'accesso erroneo avvenisse dopo che P_j ha correttamente svolto la propia operazione sulla risorsa. In questo senso, come già detto, tale errore appartiene alla categoria dei time-dependent errors.

Per fare un esempio di interferenze del tipo b) possiamo pensare al caso in cui due processi debbano operare su una stessa risorsa, per esempio la stampante. Come è già stato messo in evidenza, è necessario garantire che gli accessi alla stampante siano fra loro mutuamente esclusivi. Per risolvere questo tipo di competizione possiamo programmare la funzione di accesso alla stampante, per esempio la funzione `print(..)`, utilizzando il meccanismo di sincronizzazione indiretta, in modo tale da garantire che non siano consentite esecuzioni concorrenti della funzione. Supponiamo anche che la funzione `print(..)` abbia come parametro la stringa di caratteri da stampare; vedremo nel seguito come e con quali costrutti sia possibile garantire questo tipo di sincronizzazione. Supponiamo adesso che i processi che condividono la stampante utilizzino la precedente funzione all'interno di un ciclo, ciascuno per stampare un listato composto da un certo numero di righe. In questo caso, pur essendo garantito che i due processi non possono utilizzare contemporaneamente la stampante, può accadere che, fra due righe consecutive del listato stampato da un processo, vengano stampate righe da parte dell'altro, producendo un stampa finale del tutto illeggibile. L'errore commesso in questo caso non consiste nell'aver inserito una competizione spuria, come nel caso precedente, ma nell'aver erroneamente programmato la competizione fra i due processi. Una corretta programmazione avrebbe dovuto prevedere la mutua esclusione dell'intero ciclo di stampe da parte di un processo con quello da parte dell'altro.

Anche questo tipo di errore appartiene alla categoria dei time-dependent errors. Infatti, può accadere che le velocità relative fra i processi siano tali per cui un processo inizia il proprio ciclo di stampe solo dopo che è terminata l'esecuzione di quello dell'altro processo: in questo caso non si verifica nessun effetto erroneo. Se però, in una seconda esecuzione, cambiano i rapporti di velocità e l'esecuzione di un ciclo inizia mentre è ancora in atto quella dell'altro processo, l'errore si manifesta.

È proprio questa caratteristica che rende particolarmente insidiose le interferenze tra processi. È infatti possibile effettuare molti test di un programma contenente questi tipi di errori senza rilevare nessun malfunzionamento ed essere quindi indotti a pensare che il programma sia corretto mentre in realtà non lo è e l'errore può manifestarsi dopo che il programma è stato validato. È quindi necessario porre la massima attenzione alla prevenzione di questo tipo di errori.

Fra le due categorie di interferenze esaminate, quelle del primo tipo sono costituite da errori di programmazione che, come illustrato nell'esempio visto precedentemente, dipendono da accessi erronei a una risorsa da parte di un processo che su quella risorsa non dovrebbe operare. Pur non potendo eliminare a priori questa categoria di errori, è possibile ipotizzare l'esistenza di un meccanismo, noto col termine di *meccanismo di controllo degli accessi*, normalmente offerto dalla macchina concorrente e che è in grado di rilevare quando si verificano accessi non consentiti a una risorsa da parte di un processo; accenneremo nel seguito ad alcuni esempi di tali meccanismi. La loro presenza consente di intercettare l'errore quando si verifica, permettendo così di circoscrivere i suoi effetti al solo processo erroneo senza che questi si trasmettano anche ad altri.

Per quanto riguarda la seconda categoria di errori, purtroppo, non esiste nessun ausilio per la loro rilevazione. Anche la presenza dei meccanismi di controllo degli accessi non fornisce nessun contributo. Infatti, essendo queste interferenze relative a errori commessi nel programmare interazioni necessarie, un eventuale meccanismo di controllo degli accessi non potrebbe rilevare alcun errore. Nell'esempio illustrato precedentemente, i due processi che desiderano stampare devono avere il diritto di accedere alla stampante. Il problema, in quel caso, è legato all'errore nelle modalità di accesso e non alla possibilità o meno di accedere alla stampante. È quindi a quest'ultima categoria di interferenze che dedicheremo particolare attenzione nei prossimi capitoli, cercando di verificare la correttezza delle soluzioni proposte ai vari problemi di interazione tra processi sia mediante metodi di validazione delle soluzioni,

come per esempio quelli a cui è stato accennato nel precedente capitolo, sia ipotizzando la disponibilità di appositi costrutti linguistici che abilitino il compilatore a rilevare, in fase di compilazione, il maggior numero possibile di questi errori.

2.3 Macchina concorrente

Come è già stato indicato, il tipo di concorrenza a cui faremo riferimento è quello relativo all'esecuzione contemporanea di più attività sequenziali, normalmente interagenti fra di loro e che rappresentano le componenti di un'unica elaborazione complessiva. Intenderemo nel seguito col termine *macchina concorrente* la macchina dedicata a fornire il supporto per l'esecuzione di un programma concorrente, macchina che deve quindi essere in grado di sviluppare, durante l'esecuzione del programma, l'attività corrispondente all'elaborazione concorrente. Il concetto di macchina a cui facciamo riferimento è del tutto generale e indipendente da possibili realizzazioni fisiche. In altri termini, esso coincide col concetto di macchina astratta nel senso già indicato nel precedente capitolo e la cui realizzazione è, in generale, costituita sia da componenti hardware sia da moduli software, che forniscono tutte le funzionalità che la macchina deve offrire e che non sono realizzate direttamente in hardware.

Sulla base delle indicazioni ricavate dagli esempi illustrati nel precedente paragrafo, cerchiamo adesso di delineare le caratteristiche che la macchina concorrente deve possedere al fine di fornire un adeguato supporto per l'esecuzione di programmi concorrenti.

Una prima ovvia caratteristica, che deriva dalla stessa definizione di elaborazione concorrente, è che, nella macchina, devono essere disponibili tanti processori quante sono le attività sequenziali che compongono l'elaborazione complessiva. L'architettura della macchina concorrente si presenta quindi come un'architettura multiprocessore o multicomputer, come vedremo più in dettaglio nel seguito.

Il numero di attività sequenziali che compongono un'elaborazione varia da programma a programma e, anche per lo stesso programma, varia in generale durante la sua esecuzione, poiché attività sequenziali possono terminare e altre possono essere attivate dinamicamente. Il numero di processori che devono fornire il supporto all'esecuzione di un programma concorrente varia quindi da programma a programma e, per lo stesso programma, nel tempo. È quindi ovvio che, quasi mai, la macchina concorrente è equipaggiata con un numero di processori fisici corrispondente al massimo numero di attività sequenziali a cui un programma concorrente può dar luogo. Il concetto di processore a cui faremo riferimento è piuttosto quello di processore virtuale, con lo stesso significato che a questo termine viene dato nel contesto di un sistema operativo multiprogrammato. In altri termini, la macchina concorrente *MC* a cui faremo riferimento è una macchina astratta realizzata aggiungendo, a una macchina fisica *MO*, un insieme di moduli software che corrispondono al nucleo di un sistema operativo multiprogrammato (vedi figura 2.10).

Come è noto dalla teoria dei sistemi operativi, nel nucleo vengono normalmente implementati tre tipi di meccanismi primitivi:

- il *meccanismo di multiprogrammazione*, che ha il compito di generare i processori virtuali;

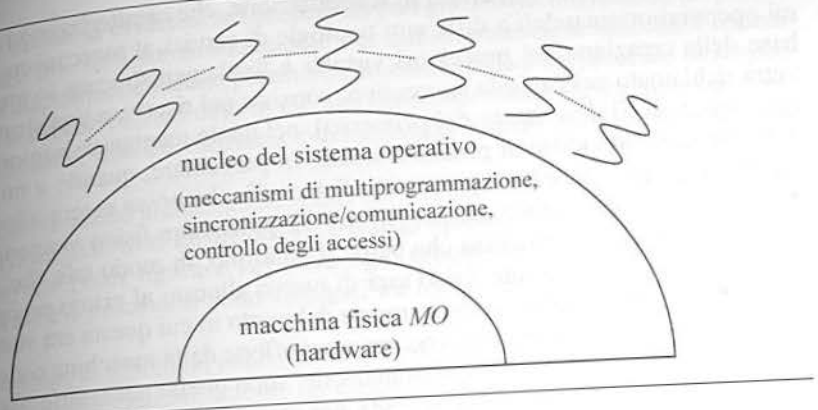


Figura 2.10 Macchina concorrente *MC*.

- il *meccanismo di sincronizzazione e comunicazione*, mediante il quale i processori possono interagire;
- il *meccanismo di controllo degli accessi*, utile, come è stato già messo in evidenza, per rilevare alcuni tipi di interferenze.

L'aggettivo *primitivo* associato a tali meccanismi ha il significato già messo in evidenza nel precedente capitolo. Infatti, tali meccanismi, in quanto funzionalità messe a disposizione dalla macchina concorrente, devono corrispondere a funzionalità atomiche. Il fatto che queste siano spesso realizzate come moduli software implica che, all'interno del nucleo di un sistema operativo, devono essere utilizzati appositi meccanismi, forniti direttamente dalla macchina fisica, che consentano di abilitare l'esecuzione di una sequenza di istruzioni macchina in modo atomico, non divisibile. In altri termini, tali meccanismi devono agire in modo tale che l'esecuzione di una sequenza di istruzioni macchina corrisponda a quella di una singola istruzione e quindi, qualora tale sequenza operi su una struttura dati del nucleo, in modo tale da non rendere visibili eventuali stati intermedi inconsistenti della stessa struttura. In questo modo, la macchina concorrente *MC* può essere utilizzata come supporto all'esecuzione di un programma concorrente, astraendo dal fatto che la stessa sia implementata completamente in hardware o, più in generale, come estensione software di una macchina fisica *MO*. Alla fine del prossimo capitolo verranno brevemente richiamati i meccanismi utilizzati per garantire l'atomicità delle funzionalità offerte dal nucleo di un sistema operativo.

Un'elaborazione concorrente corrisponde quindi a un insieme di processi sequenziali (sequenze di azioni atomiche) sviluppati concorrentemente dai singoli processori della macchina *MC*. Ogni azione atomica legge e/o modifica, in maniera non divisibile, lo stato della macchina e corrisponde all'esecuzione di una singola istruzione macchina o all'esecuzione di una funzione del nucleo (primitiva), resa atomica mediante appositi meccanismi di atomicità.

Compito del meccanismo di multiprogrammazione del nucleo del sistema operativo è quello di creare i processori virtuali, allocando i processori fisici della macchina hardware *MO* ai vari processori virtuali, mediante l'algoritmo di schedulazione.

Pur nella diversità dei vari criteri di schedulazione, che caratterizzano i diversi sistemi operativi orientandoli a differenti tipologie di servizi, il meccanismo che sta alla base della creazione dei processori virtuali è fondamentalmente lo stesso e, come verrà richiamato nel capitolo successivo, consiste nel riservare una struttura dati per ogni processo (il *descrittore del processo*), nel quale mantenere aggiornato lo stato del processore associato al processo stesso. In particolare, quando a un processo in esecuzione viene revocato il processore fisico affinché possa essere allocato a un altro processo, i valori contenuti nei registri del processore fisico vengono salvati nel descrittore relativo al processo che perde il controllo, in modo tale che, successivamente, quando il processore fisico sarà di nuovo allocato al primo processo, questo possa riprendere l'esecuzione esattamente dal punto in cui questa era stata precedentemente interrotta. Le primitive, che vengono offerte dalla macchina concorrente tramite il meccanismo di multiprogrammazione, sono quelle necessarie per la creazione e terminazione di processi (o thread), per esempio le primitive `fork` e `exit` di UNIX. Come è noto, un processo eseguendo la `fork` crea un processo figlio mentre eseguendo la `exit` il processo termina la propria esecuzione. Spesso viene fornita anche una primitiva (`join` o `wait`) che consente a un processo di sincronizzarsi con la terminazione di un processo figlio.

La necessità di consentire ai processi di interagire implica che alcune sequenze di azioni, sviluppate da processi diversi, debbano avvenire in un ben determinato ordine temporale. Per questo motivo, compito del nucleo del sistema è anche quello di fornire i meccanismi primitivi per la sincronizzazione e la comunicazione fra i processi, meccanismi che, come abbiamo visto, sono necessari per garantire la correttezza delle varie interazioni di tipo sia competitivo che cooperativo. Da questo punto di vista, esistono due diverse tipologie di meccanismi che abilitano i processi a competere e/o a comunicare seguendo due approcci completamente diversi e che dipendono dal modello architetturale della macchina concorrente.

Un primo modello architetturale, che è noto col termine di *modello a memoria comune*, è quello tipico di un'architettura multiprocessore (vedi figura 2.11), dove i vari processori virtuali sono tutti collegati tramite un bus a un'unica memoria principale. In questo caso, la memoria rappresenta l'unico dispositivo condiviso tra i vari processori e quindi l'unico dispositivo attraverso il quale processi diversi possono interagire. In particolare, ogni forma di comunicazione tra processi viene implementata tramite un'area di memoria condivisa, nella quale un processo `A` può scrivere i valori da trasferire a un processo `B` e quest'ultimo, in un secondo tempo, può leggere tali valori.

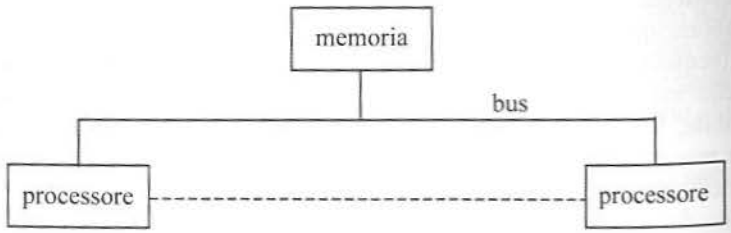


Figura 2.11 Modello architetturale a memoria comune.

Come è stato messo in evidenza nei precedenti paragrafi, ogni interazione tra processi implica però una loro sincronizzazione. Per questo motivo, la macchina concorrente mette a disposizione alcune primitive che consentono a un processo di sospendere la propria esecuzione nel caso in cui una certa condizione logica, necessaria per la sua prosecuzione, non sia verificata e di riattivarla quando la condizione attesa si verifica.

Un secondo modello architetturale, noto col termine di *modello a memoria locale* o *modello a scambio di messaggi*, è quello tipico di un'architettura distribuita multicomputer (vedi figura 2.12) dove ogni processore virtuale ha una propria memoria locale e i processori sono collegati tra loro tramite una rete di comunicazione.

In questo caso, la macchina concorrente offre le primitive necessarie a un processo `A` per inviare informazioni, tramite un canale della rete, a un secondo processo `B` (`send`) e a quest'ultimo per ricevere tali informazioni tramite lo stesso canale (`receive`). Adesso, sono le stesse primitive di comunicazione che impongono ai due processi comunicanti la necessaria sincronizzazione in modo tale che, per esempio, non sia possibile eseguire la ricezione prima che sia terminato l'invio.

È opportuno ricordare che i due modelli caratterizzano l'architettura della macchina astratta *MC* e non quella della macchina fisica *MO*. In altri termini, è il nucleo del sistema operativo che caratterizza il modello architetturale della macchina concorrente e non la struttura della macchina fisica. Per esempio, su una tradizionale macchina fisica monoelaboratore è possibile realizzare sia un sistema secondo il modello a memoria sia un sistema orientato al modello a scambio di messaggi.

È comunque ovvio che il modello di macchina astratta a memoria comune è più adatto per macchine fisiche multiprocessore, mentre il modello a scambio di messaggi si presta meglio per architetture fisicamente distribuite.

Il fatto che, nel modello a memoria comune, i processori condividano lo spazio di memoria implica che le attività concorrenti sviluppate dalla macchina astratta *MC* corrispondano al concetto di thread e non a quello di processo che, viceversa, è più idoneo a rappresentare le attività concorrenti in un sistema a scambio di messaggi. Nel seguito, parleremo di *programmazione concorrente* con particolare riferimento a una macchina astratta strutturata secondo il modello a memoria comune e di *programmazione distribuita* nel caso in cui la macchina astratta sia strutturata secondo l'altro modello.

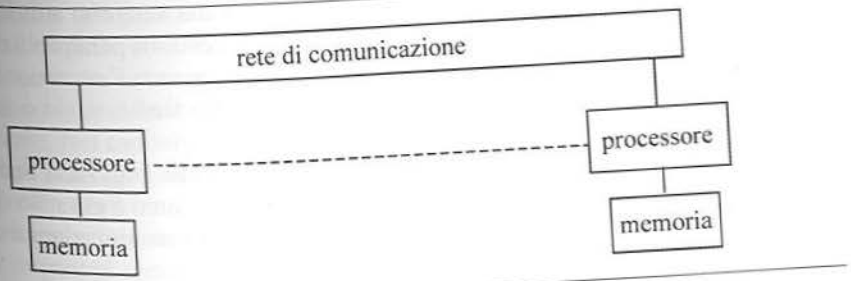


Figura 2.12 Modello architetturale a scambio di messaggi.

Esiste, infine, anche il caso in cui la macchina astratta sia strutturata secondo un modello architetturale che contempla le due diverse modalità di interazione. In questo caso, il kernel del sistema prevede sia la possibilità di definire processi sia quella di definire thread all'interno di un processo, come nel caso del sistema operativo Linux. Nel seguito, useremo comunque, genericamente, il termine processo per indicare un'attività sequenziale, qualunque sia il modello di macchina astratta.

Come illustrato nella figura 2.10, un ulteriore meccanismo, spesso fornito dal sistema operativo, è infine quello per il controllo degli accessi. Nel precedente paragrafo è già stata messa in evidenza l'importanza di tale meccanismo, in particolare con riferimento alla rilevazione di alcuni tipi di interferenze. Come noto dalla teoria dei sistemi operativi, due sono gli approcci al progetto di tali meccanismi: il primo noto come *motodo delle liste di controllo degli accessi*, il secondo come *metodo delle liste di capability*. Una trattazione approfondita di tali metodi esula dagli scopi di questo testo e si rimanda quindi alla bibliografia per un approfondimento dell'argomento.

2.4 Linguaggi concorrenti

Se la macchina concorrente rappresenta il necessario supporto per l'esecuzione di un programma concorrente, altrettanto necessaria è la disponibilità di un linguaggio di programmazione che consenta di descrivere il programma come un insieme di moduli sequenziali da eseguire concorrentemente sui processori virtuali della macchina concorrente. Tali linguaggi, noti come *linguaggi concorrenti*, si differenziano rispetto a un tradizionale linguaggio sequenziale per la necessità di fornire al programmatore, oltre agli usuali costrutti linguistici tipici della programmazione sequenziale, alcuni costrutti specifici per la concorrenza. In particolare, dovranno essere presenti nel linguaggio due insiemi di costrutti: quelli dedicati alla *specifica della concorrenza* e quelli dedicati alla *specifica delle interazioni fra processi*.

Al primo insieme appartengono tutti quei costrutti dedicati alla dichiarazione di moduli di programma da eseguire come processi sequenziali autonomi. Normalmente non è necessario dichiarare un diverso modulo per ogni processo dell'elaborazione complessiva; non è infatti raro il caso in cui più processi svolgano la stessa identica sequenza di azioni, anche se ovviamente su dati diversi. È sufficiente, in questo caso, dichiarare un solo modulo di programma come codice condiviso tra i diversi processi. Inoltre, poiché in generale i processi costituenti un'elaborazione concorrente non vengono eseguiti tutti contemporaneamente ma vengono attivati e terminati dinamicamente, è necessario disporre di appositi costrutti per specificare quando un processo deve essere attivato, cioè quando deve iniziare l'esecuzione del modulo di programma corrispondente a quel processo; allo stesso modo è necessario poter specificare la terminazione di un processo.

Al secondo insieme appartengono tutti quei meccanismi linguistici dedicati alla specifica delle interazioni tra processi e quindi, in base a quanto è già stato detto precedentemente, i costrutti necessari per la specifica delle sincronizzazioni e delle comunicazioni tra processi.

A livello della macchina concorrente sono stati messi in evidenza due diversi metodi per consentire le interazioni tra processi: tramite una memoria comune ai vari processori virtuali o tramite lo scambio di messaggi a seconda del modello architetturale della macchina. La stessa distinzione è presente anche a livello di linguaggio di programmazione. Esistono quindi linguaggi che forniscono costrutti per la specifica delle interazioni secondo il modello a memoria comune e linguaggi i cui costrutti sono concepiti secondo il modello a scambio di messaggi. Tale distinzione è comunque limitata ai costrutti per la specifica delle interazioni mentre quelli dedicati alla specifica della concorrenza sono del tutto indipendenti dal modello architetturale.

È poi compito del compilatore del linguaggio tradurre i due insiemi di costrutti in termini dei meccanismi primitivi offerti dalla macchina concorrente su cui il programma dovrà essere eseguito.

Nel prossimo capitolo verranno illustrati i principali costrutti che sono stati proposti per la specifica della concorrenza e che, come è stato detto, non dipendono dal modello architetturale della macchina concorrente. Successivamente, verranno illustrati i principali costrutti per la specifica delle interazioni, separatamente per ognuno dei due modelli architetturali.

2.5 Sommario

Il capitolo si propone di approfondire il tema della programmazione concorrente intesa come l'insieme delle tecniche, delle metodologie e degli strumenti necessari per fornire il supporto all'esecuzione di un'applicazione software come un insieme di attività svolte simultaneamente. Oggi la programmazione concorrente, pur continuando a essere il supporto indispensabile al progetto e alla realizzazione dei sistemi operativi, è largamente utilizzata anche nel progetto e nella realizzazione di varie tipologie di sistemi applicativi. Per questo motivo, i moderni linguaggi di programmazione (per esempio, Java) sono spesso dotati di un insieme di costrutti linguistici dedicati a fornire il supporto a questo tipo di strumenti e di metodologie.

Il grafo di precedenza cui dà luogo un'applicazione concorrente è a ordinamento parziale; gli archi che collegano i vari nodi del grafo rappresentano vincoli di precedenza temporale (vincoli di sincronizzazione) tra azioni di processi diversi. Dal grafo si può enucleare un insieme di processi sequenziali concorrenti e interagenti.

Vengono introdotti i due tipi di interazione possibili tra i processi: cooperazione e competizione. Per ciascuna di esse, sono discusse le proprietà fondamentali anche con l'aiuto di alcuni esempi. Esiste una terza categoria di interazione, l'interferenza, che è ascrivibile a errori di programmazione. Si tratta quindi di un tipo di interazione "non prevista e non desiderata", il cui verificarsi dà luogo a errori time-dependent, dipendenti cioè dalla velocità relativa dei processi.

Viene mostrato come questo tipo di errori risulti particolarmente insidioso, in quanto ancora più difficili da rilevare in fase di test rispetto ad altri errori di programmazione. Nei capitoli successivi verranno presentate le tecniche e gli strumenti, da utilizzare nella progettazione e scrittura di programmi concorrenti, per prevenire questo tipo di errori.

La scrittura e l'esecuzione di un programma concorrente richiede che siano disponibili linguaggi concorrenti e macchine concorrenti. Un linguaggio concorrente deve possedere due insiemi di costrutti: quelli dedicati alla specifica della concorrenza e quelli dedicati alla specifica delle interazioni tra processi. La macchina concor-

rente, cioè la macchina dedicata a fornire il supporto per l'esecuzione di un programma concorrente, è un'astrazione che trova la sua realizzazione aggiungendo a una macchina fisica un insieme di moduli software, che corrispondono al nucleo di un sistema operativo multiprogrammato, in grado di fornire il meccanismo di multiprogrammazione, quello di comunicazione e sincronizzazione e quello di controllo degli accessi.

Vengono presentati i due modelli architetturali possibili per la macchina concorrente, quello a memoria comune e quello a scambio di messaggi. Come si vedrà nei capitoli successivi, per ognuno dei due modelli la macchina concorrente metterà a disposizione due diverse tipologie di meccanismi che abilitano i processi a competere e/o a cooperare.

2.6 Note bibliografiche

Sulla definizione di programmazione concorrente e sui diversi tipi di interazione tra i processi si veda [16] e [17].

Una definizione formale del concetto di processo e delle modalità di interazione dei processi è contenuta nel lavoro di Horning e Randell [18]. Di particolare interesse risultano anche i primi lavori sull'argomento di Dijkstra [19] [20].

Metodologie e tecniche di programmazione concorrente sono discusse nei due testi [21] [22] e nei lavori dello stesso autore [23] [24].

Per quanto riguarda il meccanismo di controllo degli accessi, introdotto come uno dei meccanismi fondamentali della macchina concorrente, si può fare riferimento a [25] e [26].

3

Costrutti linguistici per la specifica della concorrenza

Nel paragrafo 2.4 abbiamo messo in evidenza la necessità che un linguaggio ad alto livello per la programmazione concorrente fornisca un insieme di costrutti linguistici che consentano di dichiarare, creare, attivare e terminare processi sequenziali e di permettere la sincronizzazione e la comunicazione tra gli stessi.

In questo capitolo illustreremo le notazioni proposte nei moderni linguaggi per esprimere la concorrenza, illustrando brevemente come tali costrutti possano essere realizzati in termini di meccanismi primitivi offerti dal supporto a tempo di esecuzione del linguaggio.

Dopo aver brevemente richiamato alcuni costrutti, che per primi sono stati utilizzati per esprimere la concorrenza (`fork/join`, `cobegin/coend`), verrà introdotto il costrutto `process`, usato per individuare quali parti di un programma devono essere eseguite come processi sequenziali.

Verranno inoltre introdotti due tipi di rappresentazione utilizzati per specificare la concorrenza in termini di `thread`. Il primo fa riferimento all'utilizzo, nell'ambito di un linguaggio sequenziale (C), della libreria `pthread` mentre il secondo alla rappresentazione, utilizzata nell'ambito di un linguaggio concorrente (Java), che supporta direttamente il `thread` a livello di linguaggio.

Al termine del capitolo verranno introdotti gli elementi essenziali del supporto a tempo di esecuzione di un sistema a processi (*nucleo* o *kernel*), con riferimento alle operazioni di creazione e terminazione dei processi.

3.1 Fork/Join

Tra le prime notazioni linguistiche proposte per esprimere la concorrenza, facendo quindi riferimento implicitamente a un supporto a tempo di esecuzione che consenta uno schema a processi, vi è l'istruzione `fork` [27] [28], utilizzata per specificare la

```
    process  p;
    =======
A:  ........;
    p=fork fun;
B:  ........;
    =======
    =======

void  fun() {
    C:  ........;
    =======
}
```

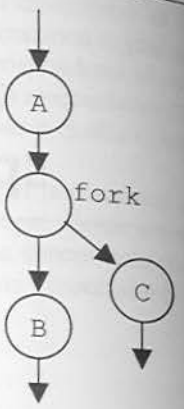


Figura 3.1 Istruzione `fork`.

creazione di un nuovo processo (o thread). Di tale istruzione esistono varie interpretazioni, fra le quali è nota quella implementata come primitiva di sistema nel Sistema Operativo UNIX. Nel seguito illustriamo la `fork` facendo riferimento al linguaggio Mesa [29] (vedi figura 3.1); in quest'ultimo essa viene definita come una funzione primitiva che crea un nuovo processo, destinato a eseguire la funzione il cui identificatore è passato come parametro, e che restituisce l'identificatore del processo creato (un valore del tipo predefinito `process`).

L'istruzione `fork` ha un comportamento simile a quello di una chiamata di procedura (`call`). Tuttavia, mentre quest'ultima implica l'attivazione del programma chiamato e la sospensione del programma chiamante, la `fork` prevede che il programma chiamante prosegua concorrentemente con l'esecuzione della funzione chiamata. L'esecuzione di una `fork` coincide quindi con la creazione e l'attivazione di un processo che inizia la propria esecuzione in parallelo con quella del processo chiamante.

In termini grafici potremmo rappresentare l'esecuzione di una `fork` mediante un nodo del grafo di precedenza che possiede più di un successore (figura 3.1).

L'esecuzione di una `fork` coincide quindi con una biforcazione del flusso di controllo del programma. Il nuovo processo, corrispondente all'esecuzione della funzione chiamata, viene eseguito in modo asincrono rispetto ai processi esistenti, in particolare nei confronti del processo chiamante.

È opportuno anche prevedere un'istruzione che consenta di determinare quando il processo, creato tramite una `fork`, ha terminato il suo compito, sincronizzandosi con tale evento. L'istruzione `join` è stata introdotta per questo motivo e, concettualmente, coincide con la congiunzione di più flussi di controllo indipendenti.

In termini di grafo di precedenza, l'esecuzione di una `join` corrisponde a un nodo del grafo che possiede più predecessori (figura 3.2).

Nella figura sono rappresentati `m` processi indipendenti, ognuno dei quali termina eseguendo un'istruzione A_i (`i=1..m`) e sincronizzandosi con la terminazione degli

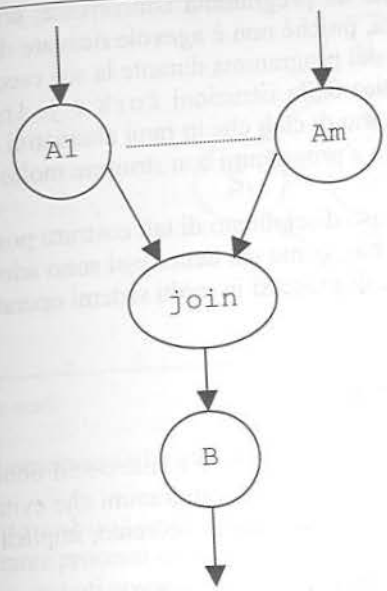


Figura 3.2 Istruzione `join`.

altri `m-1` processi. Quando tutti i processi sono completati, il programma continua come un unico processo eseguendo l'istruzione `B`.

La realizzazione più nota della `join` è quella che consente di denotare, in modo esplicito, il processo con la cui terminazione ci si vuole sincronizzare, specificando tale processo come parametro della `join` (figura 3.3).

```
    process  p;
    =======
A:  ........;
    p=fork fun;
B:  ........;
    =======
    join p;
D:  ........;
    =======

void  fun() {
    C:  ........;
    =======
}
```

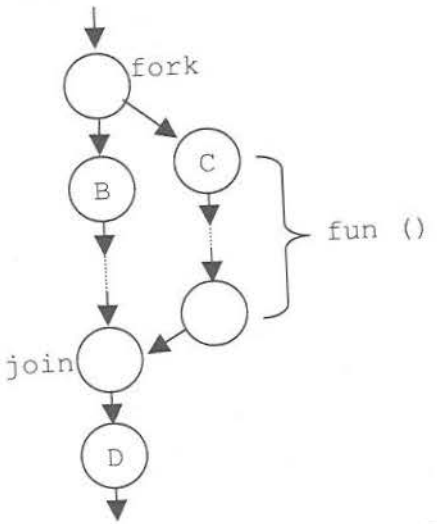


Figura 3.3 `join` con tipo `process`.

È facilmente intuibile che un programma concorrente, scritto con `fork` e `join`, non risulti di facile lettura, poiché non è agevole ricavare dal testo quali siano i processi attivi in ogni punto del programma durante la sua esecuzione. Questa difficoltà è intrinseca nella semantica delle istruzioni `fork` e `join`; in particolare, potendo esse comparire sia all'interno di cicli che in rami alternativi di istruzioni tipo `if... else`, possono dare luogo a programmi con strutture molto complicate e quindi difficili da controllare.

Ciò non toglie che un uso disciplinato di tali costrutti possa dar luogo a programmi concorrenti ben strutturati; come già detto, essi sono adottati, per esempio, come strumento per la creazione di processi in molti sistemi operativi quali quelli della famiglia UNIX.

3.2 Cobegin/Coend

Proposto da Dijkstra [30], il costrutto `cobegin/coend` obbliga il programmatore a seguire uno schema di strutturazione dei programmi che evita i problemi visti con il costrutto `fork/join`, anche se ciò, come vedremo, implica una minore flessibilità di uso.

La concorrenza viene espressa nel modo seguente:

```
cobegin
   S1;
   S2;
   ..
   Sn;
coend;
```

Le istruzioni S_1, S_2, ..., S_n sono eseguite in parallelo. Ovviamente, ogni S_i può essere un'istruzione comunque complessa; essa può contenere altre istruzioni `cobegin/coend` al suo interno. Ciò consente di nidificare strutture parallele: l'esecuzione di una struttura parallela non è terminata se non dopo che sono terminate le esecuzioni di tutte le istruzioni componenti.

In altri termini `coend`, come `join`, rappresenta un punto di sincronizzazione, in quanto implica un *rendez-vous* tra tutte le istruzioni componenti la struttura parallela.

L'interpretazione di `cobegin/coend` in termini di grafo di precedenza è immediata, come illustrato nella figura 3.4

Si noti che non è possibile, in questo caso, rappresentare un'elaborazione corrispondente a un qualunque grafo di precedenza, come invece è possibile con il costrutto `fork/join`. Infatti la tipica struttura a blocchi, indotta nel programma dall'uso del blocco parallelo `cobegin/coend`, implica che i vari blocchi paralleli possono comparire in un programma, o in sequenza o nidificati l'uno dentro l'altro.

Una differenza tra i costrutti `fork/join` e `cobegin/coend` riguarda la creazione e terminazione dei processi durante un'elaborazione. Nel caso dei costrutti `fork/join`, un processo può creare altri processi mediante la `fork`. Il processo padre (quello che ha eseguito la `fork`) e il processo figlio (quello creato) continuano la loro esecuzione in parallelo. Il processo padre si sospende soltanto se esegue la `join` quando il processo figlio non ha ancora terminato la sua esecuzione, evento

```
S0
cobegin
   S1;
   S2;
   S3;
coend
S4;
```

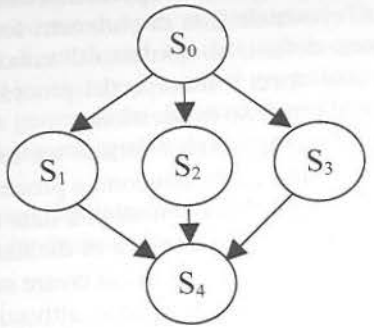


Figura 3.4 `cobegin/coend`.

che viene segnalato dal processo figlio tramite l'esecuzione di un'apposita primitiva, per esempio `quit`.

Nel caso del costrutto `cobegin/coend` il processo che esegue una `cobegin` (processo padre) crea tanti processi quante sono le istruzioni componenti il blocco parallelo (processi figli), quindi sospende la sua esecuzione in attesa che tutti i processi figli abbiano terminato. A questo punto il processo padre può riprendere la sua esecuzione.

3.3 Processi

Programmi di grandi dimensioni sono spesso costituiti da un elevato numero di moduli sequenziali, che possono essere eseguiti concorrentemente. Tali moduli possono essere programmati come procedure da attivare concorrentemente, secondo la struttura del grafo di precedenza corrispondente al programma. Questa attivazione concorrente può essere specificata nel programma mediante le istruzioni `fork` o `cobegin` viste nei paragrafi precedenti.

Si ottiene però una più chiara struttura del programma se, durante la dichiarazione di un modulo, è possibile distinguere se lo stesso debba essere eseguito concorrentemente con altri moduli o costituisca una normale procedura da chiamare in modo sincrono. Per questo motivo, molti linguaggi concorrenti moderni forniscono un costrutto linguistico mediante il quale il programmatore individua, in maniera sintatticamente precisa, quali siano quei moduli che durante l'esecuzione del programma sono destinati a essere eseguiti come processi autonomi.

Una dichiarazione di processo è simile a una dichiarazione di procedura, con la differenza che la prima denota una parte del programma che verrà eseguita concorrentemente con le altre.

Al di là delle differenze sintattiche da linguaggio a linguaggio, potremmo illustrare le caratteristiche di questo costrutto nel modo seguente:

```
process <identificatore> (<parametri formali>) {
   <dichiarazione di variabili locali> ;
   <corpo del processo> ;
}
```

Tale modulo di programma rappresenta una dichiarazione di processo e ne specifica il nome, un'eventuale lista di parametri formali (i cui corrispondenti argomenti attuali vengono definiti al momento di creazione del processo), le variabili locali su cui il processo opera e il corpo del processo, cioè il programma la cui esecuzione corrisponde al processo in questione.

In alcuni linguaggi viene fornito un costruttore di tipo, atto a definire oggetti astratti le cui istanze corrispondono a processi. Ciò è utile quando, in un'elaborazione, si possano individuare più attività tutte uguali tra loro, ma ciascuna operante su diversi dati. In questi casi, invece di dichiarare processi tutti uguali, è più semplice dichiarare un tipo `process`, di cui creare successivamente diverse istanze.

Per quanto riguarda la creazione, attivazione e terminazione dei processi, possiamo distinguere tra linguaggi che consentono la sola programmazione di elaborazioni statiche, cioè di elaborazioni nelle quali i processi componenti sono noti a priori e creati tutti all'inizio dell'esecuzione del programma, e linguaggi che consentono anche la creazione dinamica di processi.

Negli esempi che verranno mostrati nel seguito, si farà riferimento al costrutto `thread`, anziché al costrutto `process`; in particolare verrà presentata la dichiarazione dei thread tramite la libreria `pthread`, utilizzata nell'ambito di un linguaggio sequenziale e in quello di Java, dove il costrutto thread fa parte integrante del linguaggio stesso.

Si ricordi che un thread (*processo leggero*) rappresenta un flusso di esecuzione all'interno di un processo (*processo pesante*). All'interno di un processo è possibile definire più thread (*multithreading*), ciascuno dei quali condivide le risorse del processo, risiede nello stesso spazio di indirizzamento e accede agli stessi dati. Non possedendo risorse, i thread possono essere creati e distrutti più facilmente rispetto ai processi e, inoltre, il cambio di contesto tra due thread risulta molto più efficiente [31].

3.4 La libreria Pthread

Definita in ambito POSIX (*Portable Operating System Interface*), la libreria `pthread` offre un insieme sufficientemente ricco di primitive per la programmazione di applicazioni *multithreaded* realizzate nel linguaggio C.[1]

Ogni programma in esecuzione nel sistema operativo Linux è rappresentato da uno o più thread, ognuno dei quali è individuato univocamente da un *identificatore* (*Thread Identifier*, o TID). A questo scopo, la libreria introduce il tipo `pthread` per riferire i thread all'interno di programmi concorrenti.

Lo standard POSIX prevede che i thread vengano creati all'interno di un processo (*task*).

[1] Esistono diverse versioni della libreria per differenti Sistemi Operativi. Nel seguito faremo riferimento alla libreria *LinuxThreads* integrata nella libreria *glibc*, sulla quale sono basate le versioni più recenti di Linux. Pertanto, in queste versioni, *LinuxThreads* è parte integrante del Sistema Operativo.

Tipi e funzioni definiti nella libreria *pthread* sono utilizzabili previa inclusione dell'*header file* <*pthread.h*>.

Questa caratteristica non è mantenuta nell'implementazione `LinuxThreads`, perché il sistema operativo non realizza il concetto di task; in particolare, l'esecuzione di un programma determina la creazione di un thread iniziale, che esegue il codice specificato all'interno del `main`. Il thread iniziale può essere il capostipite di una gerarchia che viene a formarsi attraverso la generazione di nuovi thread; questa gerarchia è un insieme di thread che condividono uno spazio di indirizzi e quindi concettualmente equivale al task.

La creazione di un nuovo thread viene effettuata mediante la chiamata della primitiva `pthread_create` che osserva la sintassi, (facendo riferimento al linguaggio C):

```
int pthread_create(pthread_t*T,pthread_attr_t*A,
  void*(*cod)(void*arg))
```

dove:

- `T`: è il puntatore alla variabile che raccoglierà il TID del nuovo thread;
- `A`: può essere usato per specificare eventuali attributi da associare al thread (per esempio la priorità del thread), oppure `NULL`;
- `cod`: è il puntatore alla funzione che contiene il codice del nuovo thread;
- `arg`: è il puntatore all'eventuale vettore contenente i valori dei parametri da passare alla funzione codice.

La primitiva `pthread_create` restituisce zero in caso di successo, altrimenti un codice di errore. Ogni nuovo thread esegue concorrentemente con il padre e condivide con esso le variabili globali del programma nel quale è definito.

Un thread può terminare chiamando:

```
void pthread_exit(void*retval)
```

dove `retval` è il puntatore alla variabile che contiene il valore eventualmente restituito dal thread.

Un thread padre può sospendersi in attesa della terminazione di un thread figlio con:

```
int pthread_join(pthread_t th,void**retval)
```

dove `th` è il TID del particolare thread da attendere e `retval` è il puntatore alla variabile nella quale verrà memorizzato il valore eventualmente restituito dal thread (con `pthread_exit`).

L'uso delle primitive di gestione dei thread è esemplificato nel programma riportato nella figura 3.5.

Nell'esempio, il processo iniziale crea due thread figli che eseguono lo stesso codice (contenuto nella funzione `codice_T`): i tre thread (padre e due figli) eseguono concorrentemente e condividono le variabili globali. Il padre, una volta generati i figli (identificati rispettivamente dalle variabili `th1` e `th2`), si pone in attesa di essi con `pthread_join` e poi termina.

```
include <pthread.h>
/*codice dei thread: */
void *codice_T(void*arg) {
   <corpo del thread>
}
main(){
   pthread_t th1,th2;
   /* creazione primo thread: */
   pthread_create(&th1,NULL,codice_T,"1");
   /* creazione secondo thread: */
   pthread_create(&th2,NULL,codice_T,"2");
   <.......>
   pthread_join(th1,NULL);
   pthread_join(th2,NULL);
}
```

Figura 3.5 Esempio `pthread`.

3.5 I thread in Java

Ogni programma Java contiene almeno un singolo thread, corrispondente all'esecuzione del metodo `main()` sulla JVM (*Java Virual Machine*). È possibile però creare dinamicamente ulteriori thread, attivando le loro esecuzioni concorrentemente all'interno del programma. I thread Java sono oggetti che derivano dalla classe `Thread` fornita dal package `java.lang`.

Esistono due diversi modi in Java per creare nuovi thread:

- il primo consiste nel derivare nuove classi dalla classe `Thread` mediante la normale tecnica di estensione tipica dell'ereditarietà;
- il secondo consiste invece nel definire una nuova classe che implementa l'interfaccia `Runnable` (anch'essa offerta dal package `java.lang`).

Esaminiamo adesso i due metodi.

3.5.1 Creazione di thread mediante estensione della classe Thread

La classe di libreria `Thread` definisce e implementa i thread Java fornendo un insieme di metodi necessari per il controllo delle loro esecuzioni. Uno di questi metodi, il metodo `run()`, definisce ciò che ogni oggetto della classe eseguirà come thread separato (una qualunque sequenza di statements Java), concorrentemente con gli altri thread del programma. La classe `Thread` implementa un thread generico che, per default, non fa rigorosamente niente, cioè l'implementazione del suo metodo `run()` è vuota. Per creare quindi qualcosa di utile si può definire una sottoclasse di `Thread` ridefinendone (*override*) il metodo `run()` in modo tale da fargli eseguire ciò che è richiesto dal programma.

Nella successiva figura 3.6 viene illustrato questo approccio. La classe `AltriThreads` (estensione di `Thread`) implementa i nuovi thread ridefinendo il metodo `run()`.

```
class AltriThreads extends Thread {
  public void run() {
    <corpo del programma eseguito da ogni thread di questa classe>;
    <.......>
  }
}
public class PrimoEsempioConDueThreads {
  public static void main (String[] args) {
    AltriThreads t1=new AltriThreads();
    t1.start();
    <resto del programma eseguito dal thread main>;
  }
}
```

Figura 3.6 Creazione di un thread mediante estensione della classe `Thread`.

La successiva classe è quella che fornisce il `main`, nel quale viene creato il thread `t1` come oggetto derivato dalla classe `Thread`. È da notare che la creazione dell'oggetto `t1` nel primo statement del metodo `main` si limita a creare un thread vuoto, cioè senza nessuna risorsa allocata al thread (senza un processore virtuale in grado di eseguirlo). Per attivare il thread assegnandogli un processore virtuale, e quindi per farlo transire in stato pronto per l'esecuzione, è necessario eseguire il metodo `start()`, come indicato nel secondo statement del `main`. È il metodo `start()` che invoca il metodo `run()`, attivando l'esecuzione del nuovo thread. Il metodo `run()` non può essere chiamato direttamente ma solo tramite lo `start()`.

Nell'esempio della figura 3.6, la JVM gestisce due thread concorrenti: il thread principale, associato al `main`, e il `thread t1`, creato dinamicamente dal precedente con l'esecuzione dello statement `t1.start()` che lancia, in concorrenza, l'esecuzione del metodo `run()` del nuovo thread.

3.5.2 Creazione di thread mediante implementazione dell'interfaccia Runnable

Un diverso modo di fornire il metodo `run()` a un thread, rispetto a quello visto precedentemente, è quello di definire una nuova classe che implementa l'interfaccia `Runnable`, così definita:

```
public interface Runnable public abstract void run();
```

ne risulta ovviamente che ogni classe che implementa questa interfaccia deve definire il metodo `run()`. Nella figura 3.7 viene illustrato questo approccio.

In questo caso, la nuova classe che implementa `Runnable` non estende `Thread` e, quindi, non ha accesso al metodo `start()`, necessario per attivare un nuovo thread. È quindi obbligatorio anche ora creare un oggetto della classe `Thread` (`t1` nell'esempio della figura 3.7, nel secondo statement del metodo `main`). Il criterio con cui viene specificato il metodo `run()`, che il nuovo thread dovrà eseguire, è quello di creare un oggetto `Runnable`, come nella prima istruzione del `main` nel quale l'oggetto esecutore è creato come istanza della classe `AltriThread`, che implementa il nuovo metodo `run()`.

```
class AltriThread implements Runnable {
   public void run() {
      <corpo del programma eseguito da ogni thread di questa classe>;
   }
}
public class SecondoEsempioConDueThread {
   public static void main (String[] args) {
      Runnable esecutore=new AltriThread();
      Thread t1=new Thread(esecutore);
      t1.start();
      <resto del programma eseguito dal thread main>;
   }
}
```

Figura 3.7 Creazione di un thread mediante implementazione dell'interfaccia `Runnable`.

Tale oggetto è poi passato come parametro al costruttore del `thread t1` nel secondo statement del `main`. In questo modo, quando `t1` è attivato mediante il metodo `start()` (terzo statement), inizia l'esecuzione del metodo `run()` dell'oggetto `Runnable`.

La necessità di avere in Java anche questo secondo criterio di creazione di un thread è dovuto all'impossibilità di derivazione multipla di una classe. Per cui, per esempio, se una classe è già derivata da un'altra non è possibile estendere contemporaneamente anche la classe `Thread`. In questo caso, si può allora consentire alla classe derivata di implementare contemporaneamente l'interfaccia `Runnable`.

3.6 Realizzazione delle primitive di creazione e terminazione dei processi

Come indicato nel capitolo 1, il compito della realizzazione delle unità centrali virtuali, sulle quali i processi vengono eseguiti, e dei meccanismi di comunicazione e sincronizzazione, necessari per le interazioni tra i processi, è affidato a un particolare componente di sistema, chiamato *nucleo* (o *kernel*).

Il nucleo, oltre a rappresentare il supporto a tempo di esecuzione di un linguaggio di programmazione che utilizza il concetto di processo, rappresenta in generale la parte più interna di un sistema operativo. Le sue proprietà vengono quindi presentate e discusse nell'ambito di un corso introduttivo ai sistemi operativi [31]. In questo paragrafo, ci concentreremo sulla realizzazione, in ambiente sia monoprocessore che multiprocessore, delle primitive `fork`, usate da un processo per creare un altro processo (figlio), di quelle `join`, usate per consentire a un processo di attendere la terminazione di un processo figlio (equivalente alla primitiva `wait` di UNIX), e di quelle `quit`, usate per eliminare un processo (equivalente alla primitiva `exit` di UNIX). L'implementazione riportata è del tutto esemplificativa e non corrisponde a come le precedenti primitive vengono realizzate all'interno del kernel di UNIX.

Come noto dalla teoria dei sistemi operativi, tutte le funzioni del nucleo vengono implementate in modo tale da comportarsi come operazioni primitive e quindi atomiche. La tecnica, utilizzata per garantire l'atomicità delle funzioni del nucleo, consiste nell'eseguirle a interruzioni disabilitate e, nel caso di sistemi multielaboratori, facendo riferimento anche ai meccanismi di `lock` e `unlock`, realizzati mediante particolari istruzioni macchina, come la `test_and_set`, e che verranno brevemente richiamati nel paragrafo 5.8.2. Inoltre, le funzioni del nucleo girano con la CPU in stato privilegiato mentre le istruzioni eseguite da un qualunque processo girano con la CPU in stato non privilegiato. Come è noto, ciò implica che il meccanismo di passaggio tra l'ambiente dei processi e l'ambiente del nucleo è costituito dal meccanismo di interruzione. In altri termini, un processo invoca una primitiva di sistema eseguendo, invece di una normale chiamata a sottoprogramma, un'opportuna istruzione del tipo *chiamata a supervisore* (`SVC`), che genera un'interruzione producendo un effetto del tutto analogo all'arrivo di un segnale esterno da una periferica. Si parla infatti di *interruzioni interne* o *sincrone*, per distinguerle dalle *interruzioni esterne* o *asincrone* generate dai dispositivi esterni di I/O. Terminata l'esecuzione di una primitiva, così come avviene al termine dell'esecuzione di una funzione di risposta alle interruzioni esterne, il controllo torna dall'ambiente di nucleo all'ambiente dei processi; tale trasferimento avviene eseguendo un'istruzione di *ritorno da interruzione* (`IRET`). Possiamo quindi schematizzare il passaggio dall'ambiente dei processi all'ambiente di nucleo, e viceversa, come indicato nella figura 3.8.

L'esecuzione di una primitiva da parte di un processo corrisponde all'inserimento, nello stack dello stesso, dei parametri richiesti dalla primitiva stessa e, come è stato detto, all'esecuzione di una istruzione di tipo `SVC` che genera un'interruzione di tipo sincrono. La gestione dell'interruzione prevede il salvataggio del contesto del processo, cioè dei valori contenuti nei registri di macchina (funzione `salvataggio_stato` che vedremo di seguito). Viene quindi eseguita la procedura di risposta all'interruzione (`interrupt_handler`) e, successivamente, si torna al processo ripristinando nei registri i valori prima salvati (`ripristino_stato`) ed eseguendo l'istruzione di ritorno da interruzione. La funzione di risposta all'interruzione, sulla base dell'operando della `SVC`, provvede a chiamare la funzione primitiva richiesta, passandogli i parametri che preleva dallo stack del processo.

Nel caso in cui l'esecuzione della primitiva comporti il blocco del processo chiamante, il nucleo provvederà alla scelta di un nuovo processo (tra quelli pronti) da mettere in esecuzione. Ciò avviene indicando un diverso contesto da cui ripristinare i valori dei registri di macchina, nel momento in cui il controllo viene restituito alla funzione di risposta all'interruzione e quindi alla funzione `ripristino_stato`.

Figura 3.8 Passaggio di controllo fra ambiente dei processi e ambiente di nucleo.

Così facendo, viene ripristinato il contesto del nuovo processo e restituito, quindi, il controllo a un processo diverso da quello che ha invocato la primitiva.

Nel seguito del testo illustreremo ogni primitiva di sistema indicandola come una normale funzione. È però necessario ricordare, come indicato nella figura 3.8, che la sua esecuzione avviene dopo che è stato salvato lo stato del processo chiamante e che, una volta terminata, viene ripristinato lo stato del processo che deve andare in esecuzione (lo stesso processo chiamante in assenza di una commutazione di contesto o, in caso contrario, il processo scelto dall'algoritmo di schedulazione).

Ciascun processo è rappresentato nel nucleo da una struttura dati detta `des_processo` (descrittore del processo).

Nel descrittore sono contenuti diversi tipi di informazioni, fra i quali: il nome del processo (cioè l'identificatore con cui è noto al sistema); le informazioni necessarie per definire le sue modalità di servizio (per esempio, la sua *priorità* o il *quanto di tempo*, che possono essere usati in algoritmi di scelta (*scheduling*) su base prioritaria o a divisione di tempo); il `contesto` (cioè l'area di memoria in cui salvare le informazioni contenute nei registri del processore all'atto della sospensione del processo e che quindi rappresenta il processore virtuale del processo stesso); lo `stato` in cui si trova il processo; l'identificatore del processo padre; il numero e lo stato dei processi figli (se sono già terminati o ancora in esecuzione); l'identificazione del processo `successivo` nella coda nella quale, come vedremo, il processo è inserito a seconda del suo stato (pronto o bloccato). Possono poi esserci ulteriori campi diversi da sistema a sistema.

Definiamo, quindi, il tipo `des_processo` e, poiché le code in cui un descrittore può essere inserito vengono realizzate mediante liste concatenate, anche il tipo `p_des` (puntatore a descrittore):

```
typedef des_processo*p_des;
typedef struct {
   PID nome;
   modalità_di_servizio servizio;
   tipo_contesto contesto;
   tipo_stato stato;
   PID padre;
   int N_figli;
   des_figlio prole[max_figli];
   p_des successivo;
}des_processo;
```

Vediamo adesso di dettagliare alcuni dei campi di questa struttura, lasciandone non specificati altri che hanno una struttura dipendente dal particolare tipo di architettura fisica, come il `tipo_contesto` del campo `contesto`. È questo il campo in cui vengono salvati i valori dei registri di macchina e, come tale, cambia da processore a processore. Analogamente, per semplicità di esposizione, lasciamo solo parzialmente specificati altri campi del descrittore, come per esempio `tipo_stato` che, normalmente, è un tipo enumerazione i cui valori identificano lo stato in cui si trova il processo (se attivo o bloccato e, in questo caso, in attesa di quale evento).

Ogni processo viene univocamente identificato nel sistema tramite un numero intero. Con `PID` (*Process IDentification number*) denotiamo quindi un alias del tipo `int` utilizzato per identificare processi:

```
typedef int PID;
```

Faremo inoltre l'ipotesi di avere a disposizione una funzione (`assegna_nome`) che, chiamata all'atto della creazione di un processo, restituisce il suo nome unico:

```
PID assegna_nome();
```

Per il tipo `modalità_di_servizio` si sono previsti, come esempio, due campi, utilizzati rispettivamente in algoritmi di scelta dei processi basati sulla loro priorità e sulla modalità a divisione di tempo:

```
typedef struct {
   int priorità;
   int delta_t;
}modalità_di_servizio;
```

Sono poi riportati tre campi che caratterizzano il processo nell'ambito delle relazioni padre-figlio: il primo (`padre`) riporta il nome (`PID`) del processo padre; il secondo (l'intero `N_figli`) contiene il valore che corrisponde al numero di processi figli che in un certo istante il processo ha già creato; il terzo (`prole`) è un vettore di strutture, una per ciascuno dei processi figli. Ogni elemento di questo vettore viene utilizzato per caratterizzare se il corrispondente figlio è ancora in esecuzione o se è già terminato, informazione necessaria per implementare la primitiva `join`. La costante `max_figli` rappresenta il numero massimo di figli che un processo può creare. Il tipo `des_figlio` di questa struttura (descrittore del figlio) può essere così definito:

```
typedef struct {
   PID figlio;
   boolean terminato;
}des_figlio;
```

dove il campo `figlio` denota il `PID` di un processo figlio e il campo boolean `terminato` (inizializzato a `false`) viene posto al valore `true` quando il corrispondente processo termina la sua esecuzione.

L'insieme di tutti i descrittori dei processi viene rappresentato dal vettore `descrittori`:

```
des_processo descrittori[num_max_proc];
```

dove la costante `num_max_proc` rappresenta il numero massimo di processi che il nucleo è in grado di gestire. Supporremo disponibile la funzione:

```
p_des descrittore(PID x);
```

che può essere invocata passandogli, tramite il parametro `x`, il `PID` di un processo e che, scandendo il vettore `descrittori`, restituisce il puntatore al descrittore del processo: in caso di processo non esistente restituisce `null`.

Come è noto dalla teoria dei sistemi operativi, il numero dei processi è maggiore del numero dei processori fisici, per cui soltanto un sottoinsieme di questi può essere in *esecuzione* in ogni istante. Gli altri processi sono o *bloccati*, in attesa di qualche evento, o *pronti* per essere eseguiti non appena si libera uno dei processori fisici. I processi pronti vengono normalmente mantenuti in diverse code FIFO, una per ciascun livello di priorità e ciascuna organizzata come una lista nella quale i descrittori vengono concatenati tramite il campo `successivo`. A ogni coda possiamo quindi associare una struttura (`des_coda`), contenente una coppia di valori che identificano, rispettivamente, il puntatore al `primo` e all'`ultimo` descrittore di processo nella coda. Essa è costituita dalla lista ottenuta partendo dal descrittore indicato da `primo` e utilizzando il campo `successivo` per indicare i descrittori successivi fino al descrittore indicato da `ultimo`. Per ciascuna coda verrà adottata l'ipotesi che il campo primo assuma il valore `null` se la coda è vuota. Analogamente verrà assegnato il valore `null` al campo successivo dell'ultimo descrittore della coda:

```
typedef struct {
   p_des primo,ultimo;
} des_coda;
```

Supporremo, nel seguito, di utilizzare le seguenti funzioni: la prima

```
void inserimento (p_des pro, des_coda coda)
```

per inserire un descrittore di processo il cui puntatore è passato tramite il parametro `pro` nella coda FIFO `coda`, mentre la seconda

```
p_des prelievo(des_coda coda)
```

per prelevare da `coda` un descrittore di processo il cui puntatore viene restituito dalla funzione.

La coda dei processi pronti può essere quindi rappresentata da un array di code FIFO, una per ciascun livello di priorità:

```
des_coda coda_processi_pronti[num_min_priorità];
```

È stata fatta l'ipotesi che gli indici dei livelli di priorità siano i numeri interi compresi tra zero (massimo livello di priorità) e `num_min_priorità-1` (minimo livello di priorità).

La variabile `descrittori_liberi` rappresenta una coda nella quale sono concatenati i descrittori disponibili per la creazione di nuovi processi:

```
des_coda descrittori_liberi;
```

3.6.1 Ambiente monoprocessore

In un ambiente monoelaboratore, in ogni istante, tra tutti i processi pronti uno solo è realmente in esecuzione. Oltre alle strutture dati definite precedentemente, il nucleo necessita quindi dell'informazione di quale sia il processo in esecuzione. Questa informazione viene mantenuta nella variabile puntatore `processo_in_esecuzione` che contiene l'indirizzo del descrittore del processo in esecuzione.

```
p_des processo_in_esecuzione;
```

Quando il processo chiama una primitiva di sistema (o viene interrotto da un'interruzione esterna), come visto nella figura 3.8, il nucleo provvede a copiare i contenuti dei registri di macchina nell'area `contesto` del suo descrittore (il cui indirizzo è contenuto in `processo_in_esecuzione`) tramite la funzione `salvataggio_stato`:

```
void salvataggio_stato() {
  p_des esec=processo_in_esecuzione;
  esec -> contesto=<valori dei registri della CPU>;
}
```

Quando l'esecuzione di una primitiva termina, il controllo torna al processo mediante la funzione inversa `ripristino_stato`, che ripristina i valori nei registri di macchina prelevandoli dall'area `contesto` del descrittore il cui indirizzo è contenuto in `processo_in_esecuzione`:

```
void ripristino_stato() {
  p_des esec=processo_in_esecuzione;
  <registri della CPU>=esec -> contesto;
}
```

Una primitiva molto semplice, spesso fornita dal nucleo di un sistema operativo, è quella che, quando invocata, restituisce il `PID` del processo in esecuzione (`PIE`):

```
PID PIE() {
  PID nome_processo_in_esecuzione;
  p_des esec=processo_in_esecuzione;
  nome_processo_in_esecuzione=esec -> nome;
  return(nome_processo_in_esecuzione);
}
```

Durante l'esecuzione di una primitiva, un processo può però perdere il controllo, come vedremo più in dettaglio nel paragrafo 5.8, o anche in seguito all'esecuzione di una `join` con un processo figlio non ancora terminato. In questo caso, prima di completare la primitiva, verrà eseguita la funzione `assegnazione_CPU`, che sceglie un diverso processo da mandare in esecuzione al posto di quello che si blocca, inserendo nella variabile `processo_in_esecuzione` l'indirizzo del suo descrittore in modo tale che, quando la primitiva termina e viene eseguita la funzione `ripristino_stato`, si abbia una commutazione di contesto:

```
void assegnazione_CPU {
   int k=0;
   p_des p;
   while(coda_processi_pronti[k].primo==null)
      k++;
   p=prelievo(coda_processi_pronti[k]);
   processo_in_esecuzione=p;
   <registro-temporizzatore>=p -> servizio.delta_t;
}
```

Questa funzione sceglie il processo da attivare ricercando quello a maggior priorità nella coda dei processi pronti. Nell'ipotesi che il sistema funzioni a divisione di tem-

, la funzione termina caricando il registro del temporizzatore con il valore del-
t_t dell'intervallo di esecuzione previsto per il processo che viene attivato. Ciò im-
ca, come vedremo, che una commutazione di contesto si ha anche quando arriva
'interruzione dal temporizzatore. La funzione prevede inoltre che esista sempre al-
no un processo pronto. Come è noto, ciò viene ottenuto mediante l'uso di un pro-
so *fittizio* (*dummy*) che ha la priorità più bassa ed è sempre nello stato di pronto.
In un sistema in cui i processi sono gestiti in base alla loro priorità, è anche ne-
sario effettuare un controllo ogni volta che un processo deve essere attivato inse-
dolo in coda pronti (per esempio quando il processo viene creato o quando, come
remo nei successivi capitoli, dallo stato bloccato torna in stato pronto). Tale con-
lo ha il compito di confrontare la priorità del processo in esecuzione con quella
processo da attivare. Se la priorità del processo da attivare è minore o uguale di
lla del processo in esecuzione allora il processo viene effettivamente inserito in
a pronti e in particolare nella coda FIFO relativa alla sua priorità. In caso contra-
però, il processo da attivare è più importante del processo in esecuzione e quindi
lest'ultimo che viene inserito in coda pronti revocandogli l'uso della CPU e, al
posto, viene indicato come `processo_in_esecuzione` il processo da atti-
(*cambio di contesto*). Per questo motivo, faremo riferimento alla seguente fun-
e `attiva`, da eseguire ogni volta che un processo deve essere attivato, passan-
i come parametro il puntatore al suo descrittore (`proc`):

```
attiva (p_des proc) {
_des esec=processo_in_esecuzione;
nt pri_esec=esec -> servizio.priorità;
nt pri_proc=proc -> servizio.priorità;
oc -> stato=<"processo attivo">;
f(pri_esec > pri_pro) {
 inserimento(esec.coda_processi_pronti[pri_esec]);
 processo_in_esecuzione=proc);
 }
se
 inserimento(proc.coda_processi_pronti[pri_proc]);
```

funzione, i due interi `pri_proc` e `pri_esec` indicano, rispettivamente, la
rità del processo da attivare e del processo in esecuzione. Ricordiamo che a
aggiore priorità corrisponde un minor valore dell'intero `priorità`.
ora possibile definire il comportamento delle primitive `fork`, `join` e `quit`.
no per prima la primitiva `fork`:

```
ef enum {OK,eccezione} risultato;

ato fork(des_processo inizializzazione) {
es p;
NF;
es esec=processo_in_esecuzione;
descrittori_liberi.primo==NULL)
eturn eccezione; /*non ci sono descrittori liberi*/
e {
=prelievo(descrittori_liberi);
*p=inizializzazione;
p -> nome=assegna_nome();
p -> padre=esec -> nome;
esec -> N_figli++;
NF=esec->N_figli;
esec -> prole[NF].figlio=p -> nome;
esec -> prole[NF].terminato=false;
attiva(p);
return ok; }
}
```

dove il tipo enumerazione `risultato` è stato definito per caratterizzare il valore restituito dalla primitiva: si ha il valore `OK` se la primitiva termina correttamente e il valore `eccezione` se non è possibile terminare la creazione del processo per mancanza di descrittori liberi. Il parametro `inizializzazione` definisce il valore con cui inizializzare il descrittore del processo da creare, in particolare i valori del campo `servizio` (col valore della priorità del processo), del campo `contesto` (con i valori iniziali da caricare nei registri di macchina al momento dell'attivazione del processo), del campo `N_figli` (inizializzato a zero). Il campo `nome` viene inizializzato esplicitamente, chiamando la funzione `assegna_nome` che restituisce il `PID` del processo creato. Infine, viene inizializzato il campo `padre`, che coincide col nome del processo che ha invocato la `fork`, cioè del `processo_in_esecuzione`. Terminata l'inizializzazione del descrittore del processo creato, è necessario modificare anche alcune informazioni del descrittore del processo in esecuzione. In particolare viene incrementato il numero dei figli e inizializzato l'elemento del campo `prole`, che corrisponde al processo figlio che è stato creato. Tale elemento è una struttura che identifica il nome del figlio che è stato creato e il suo stato di esecuzione `terminato` che viene posto al valore `false`. Completate le operazioni di inizializzazione e modifica dei descrittori dei due processi (padre e figlio), il processo creato viene attivato mediante la funzione `attiva` decidendo a quale processo restituire il controllo: al padre o al figlio in base alle loro priorità.

Vediamo adesso la funzione `join` che richiede, come parametro, il nome del processo figlio con cui sincronizzarsi. Per semplicità, supponiamo disponibile la funzione:

```
int indice_figlio(p_des padre,PID nome_figlio)
```

che, dato il nome unico del processo figlio (`nome_figlio`), scandisce il vettore `prole` nel descrittore del processo padre, il cui puntatore è passato come primo parametro (`padre`). La funzione restituisce l'indice dell'elemento del vettore `prole`, il cui campo `figlio` corrisponde al nome passato come secondo parametro:

```
void join(PID nome_figlio) {
   p_des esec=processo_in_esecuzione;
   int k=indice_figlio(esec.nome_figlio);
   if(esec -> prole[k].terminato==false) {
         /*figlio non terminato*/
      esec -> stato=<"sospeso in attesa che il figlio termini">;
      Assegnazione_CPU();
   }
}
```

La primitiva `join` verifica se il processo figlio è terminato; in caso contrario indica, nel campo stato del padre (il processo in esecuzione), che il processo passa in stato sospeso in attesa della terminazione del figlio. Quindi la primitiva termina scegliendo un nuovo processo da mandare in esecuzione (cambio di contesto). In caso affermativo il controllo ritorna al processo chiamante (`processo_in_esecuzione` non viene modificato) che quindi può proseguire l'esecuzione.

Infine, la primitiva `quit`:

```
void quit(){
   p_des esec=processo_in_esecuzione;
   PID nome_padre=esec -> padre;
   p_des p_padre=descrittore(nome_padre);
   int k indice_figlio(p_padre,esec -> nome);
   p_padre -> prole[k].terminato=true;
   inserimento(esec,descrittori_liberi);
   if(p_padre -> stato==<"in attesa che questo figlio termini">){
      int pri=p_padre -> servizio.priorità;
      inserimento(p_padre,coda_processi_pronti[pri]);
      p_padre -> stato=<"processo attivo">;
   }
   assegnazione_CPU();
}
```

Per prima cosa si ricava il nome del padre di chi sta terminando (cioè del processo in esecuzione) prelevandolo dal campo `padre` del suo descrittore. Quindi, mediante la funzione `descrittore`, si ricava il puntatore al descrittore del padre (`d_padre`). In questo descrittore, mediante la funzione `indice_figlio`, si cerca l'indice del figlio corrispondente al processo che sta terminando e, quindi, si assegna il valore `true` al suo campo `terminato`. A questo punto possiamo liberare il descrittore del processo che ha terminato inserendolo nella coda `descrittori_liberi`. Infine, se il padre era sospeso in attesa di questo evento, possiamo riattivarlo inserendo il suo descrittore in coda pronti. È da notare che in questo caso non viene invocata la funzione `attiva`, poiché il processo in esecuzione non esiste più e quindi la funzione termina invocando `assegnazione_CPU`, per scegliere il processo a più alta priorità da mandare in esecuzione.

Per consentire la realizzazione di modalità di servizio a divisione di tempo, è necessario che il nucleo gestisca le interruzioni provenienti dal dispositivo temporizzatore. Lo scopo è quello di revocare, a intervalli fissati di tempo, la CPU al processo in esecuzione e di assegnarla a un nuovo processo pronto. Si ha:

```
void interruzione_temporizzatore() {
   p_des esec=processo_in_esecuzione;
   int k;
   k=esec -> servizio.priorità;
   Inserimento(esec,coda_processi_pronti[k]);
   Assegnazione_CPU();
}
```

3.6.2 Ambiente multiprocessore

L'architettura multiprocessore presa in considerazione prevede che i diversi processori abbiano accesso a una memoria comune nella quale sono contenute le strutture dati e le funzioni del nucleo. Ogni processo può inoltre operare su ogni processore.

Per evitare interferenze tra processi operanti su processori diversi, sarebbe necessario considerare il nucleo come una sezione critica, garantendo quindi che un singolo processo alla volta possa accedere a esso. Questa soluzione, che ha il pregio della semplicità, presenta tuttavia l'inconveniente di limitare il grado di parallelismo del sistema, escludendo a priori ogni possibilità di esecuzione contemporanea di più funzioni del nucleo. Infatti solo alcune strutture del nucleo, come per esempio le liste dei descrittori liberi e dei processi pronti, devono essere accedute in mutua esclusione. Rinviando la soluzione di questo problema a quando verranno introdotte le primitive per la sincronizzazione dei processi (paragrafo 5.8.2), nel seguito si vogliono evidenziare solamente alcuni aspetti che caratterizzano la realizzazione delle funzioni per la creazione e terminazione dei processi rispetto al caso monoprocessore.

Il codice per le funzioni `fork`, `join` e `quit` rimane sostanzialmente lo stesso, fatta eccezione per il fatto che le liste `coda_processi_pronti` e `descrittori_liberi` devono essere utilizzate in modo esclusivo da parte dei processi.

Inoltre sarà necessario introdurre il vettore `processo_in_esecuzione[N]` (anziché la variabile `processo_in_esecuzione`) dove `N` rappresenta il numero di processori fisici; ogni componente del vettore è associato a un processore fisico e il suo valore rappresenta l'indirizzo del descrittore del processo che quel processore sta eseguendo (o è `null` se il processore è libero).

La differenza fondamentale rispetto a quanto visto precedentemente riguarda la funzione `assegnazione_CPU`. Quando viene creato un nuovo processo tramite la `fork` può succedere che esistano uno o più processori liberi a cui indifferentemente il processo può essere assegnato. La funzione `fork` può analizzare il vettore `processo_in_esecuzione` e assegnare a un suo elemento, corrispondente a un processore libero, l'indirizzo del descrittore del processo da eseguire. Se tutti i processori sono occupati e il nuovo processo ha priorità maggiore di almeno uno di quelli attualmente in esecuzione, è necessario che il nucleo, mediante la funzione `attiva`, provveda a revocare la CPU al processo con priorità più bassa e assegnarla a quello nuovo. Per questo motivo, la funzione `attiva`, vista precedentemente, deve essere modificata in modo tale che non si limiti a verificare la priorità del processo, in esecuzione sul processore dove viene eseguita la `fork`, ma controlli la priorità dei processi in esecuzione su tutti i processori, per scegliere quello con priorità più bassa. Inoltre, è necessario un meccanismo di interruzione interprocessore.

Indichiamo, infatti, con P_i il processo che esegue la `fork` operando sul processore U_i, con P_j il nuovo processo e con P_k il processo a più bassa priorità che opera sull'unità U_k. Il nucleo, attualmente eseguito dall'unità U_i, provvede a inviare un segnale di interruzione all'unità U_k che, utilizzando le funzioni del nucleo stesso per il cambio di contesto, risponde all'interruzione inserendo il processo P_k nella coda dei processi pronti e mettendo in esecuzione il processo P_j.

Si noti, per concludere, che l'ipotesi che un processo possa essere eseguito su ogni processore non sempre risulta possibile: in alcuni sistemi multiprocessori, pro-

cessi quali driver per dispositivi o file server, possono essere eseguiti solo sul processore cui è collegato il dispositivo periferico.

Un problema analogo si presenta nel caso di sistemi multiprocessori dotati di un sistema di memoria non uniforme; in questi casi è più conveniente che un processo venga eseguito sul processore la cui memoria locale contiene il suo codice. Infine, in presenza di memorie cache, è più opportuno che un processo venga riassegnato al processore nel quale era stato prima eseguito, in quanto parte del suo stato può essere contenuto nella cache.

3.7 Sommario

L'obiettivo di questo capitolo è stato quello di introdurre i meccanismi linguistici per la specifica della concorrenza. Tali meccanismi sono comuni ai due tipi di architetture di macchine concorrenti che vedremo nel seguito e quindi anche ai due tipi di linguaggi concorrenti ad alto livello.

Sono stati introdotti i meccanismi linguistici `fork/join` e `cobegin/coend`, discutendone le caratteristiche e le principali differenze. Sono stati poi introdotti i costrutti `process` e `thread`. Di quest'ultimo è stata illustrata la realizzazione nell'ambito della libreria `pthread`. Analogamente è stata illustrata la realizzazione dei thread nel linguaggio Java.

Alla fine del capitolo è stata mostrata la realizzazione del meccanismo di multiprogrammazione offerto dal nucleo di un sistema operativo a supporto della concorrenza. Per questo motivo, sono state introdotte le strutture dati e le funzioni primitive del nucleo che, nei prossimi capitoli, verranno estese con quelle necessarie per offrire il supporto ai meccanismi di sincronizzazione e comunicazione tra processi.

3.8 Note bibliografiche

I costrutti `fork` e `join` sono stati introdotti per la prima volta da Conway [27] e, successivamente, da Dennis [28]; sono stati implementati, come primitive di sistema, nel Sistema Operativo UNIX [32] e, come meccanismo linguistico, nel linguaggio di programmazione Mesa [29].

Il costrutto `cobegin/coend` è stato introdotto da Dijkstra [30] con il nome di `parbegin/parend`. È stato poi ripreso e ampiamente discusso da Brinch Hansen in [23] e [24]. È stato quindi introdotto nel linguaggio Edison, sviluppato dallo stesso Brinch Hansen [33] per la programmazione di applicazioni in tempo reale su architetture multimicroprocessore.

Il costrutto `process`, introdotto nel linguaggio Concurrent Pascal [22], è uno dei più usati per specificare la concorrenza in un linguaggio ad alto livello.

Per la programmazione concorrente mediante la libreria `Pthread` si può fare riferimento a [34] e [35].

Per un buon tutorial relativo al linguaggio Java si può fare riferimento a [37] mentre in [38] si trova una trattazione completa della concorrenza in Java.

Per quanto riguarda, infine, i meccanismi di nucleo a supporto della concorrenza, si può fare riferimento a un buon testo generale sui sistemi operativi [31] [39] [40] [26].

Parte seconda

4
Modello a memoria comune

Come indicato nel paragrafo 2.4 e nel precedente capitolo, mentre i costrutti linguistici per la specifica della concorrenza non dipendono dal modello architetturale della macchina concorrente, i costrutti per la specifica delle interazioni e della sincronizzazione tra processi sono strettamente dipendenti dalle caratteristiche dello stesso. Per questo motivo, questa seconda parte del testo è riservata all'analisi dei costrutti adatti per macchine organizzate secondo il modello a memoria comune, riservando l'ultima parte del testo all'esame dei costrutti relativi al modello a scambio di messaggi.

È quindi opportuno caratterizzare più in dettaglio il modello architetturale a memoria comune con lo scopo di definire le proprietà che devono essere possedute dai costrutti linguistici adatti a programmare le interazioni tra processi su macchine organizzate secondo questo modello. Vedremo poi, nei due successivi capitoli, due diversi esempi di costrutti specifici: vedremo per primo il meccanismo dei *semafori*, sicuramente il più noto fra i meccanismi di sincronizzazione di "basso livello", cioè di quelli primitivi offerti direttamente dalla macchina concorrente; successivamente vedremo il costrutto linguistico dei *monitor*, sicuramente il più usato fra i meccanismi di "alto livello".

4.1 Aspetti caratterizzanti il modello

Come è stato indicato nel paragrafo 2.3, il modello a memoria comune è quello tipico di un'architettura multielaboratore dove i vari processori (virtuali) sono tutti collegati, tramite un bus, a un'unica memoria principale. Questo tipo di organizzazione logica impone che ogni interazione tra processi possa avvenire esclusivamente tramite l'unica entità condivisa tra i processori virtuali, cioè tramite la memoria comune. La macchina concorrente, e quindi il supporto a tempo di esecuzione di un linguaggio che segue questo modello, non fornisce, infatti, alcun meccanismo primitivo di interazione diretta tra processo e processo.

In base a queste caratteristiche, vediamo adesso come può essere strutturata un'applicazione da far eseguire sulla macchina concorrente e, quindi, che tipo di costrutti dovrà fornire un linguaggio che segue questo modello per consentire la specifica delle interazioni tra processi.

Da un punto di vista macroscopico, possiamo indicare che ogni applicazione concorrente viene strutturata in termini di due insiemi disgiunti di componenti: i *componenti attivi*, ossia i processi, e i *componenti passivi*, ossia le risorse. Mentre il concetto di processo è stato sufficientemente descritto nei precedenti capitoli, vediamo adesso di caratterizzare più in dettaglio il concetto di risorsa, al fine di descrivere come tale concetto sia stato recepito a livello di linguaggio di programmazione.

Di seguito, indicheremo col termine di risorsa qualunque oggetto (passivo), sia esso fisico o logico, sul quale un processo deve operare per portare a termine il compito a lui affidato nell'ambito di un programma applicativo. In altri termini, questo concetto di risorsa corrisponde a quello di operando di una qualunque operazione eseguita dal processo. In questo senso, possiamo classificare le risorse in base al loro tipo, secondo la definizione di tipo vista nel paragrafo 1.4 (vedi nota 2).

Con riferimento a un sistema software organizzato secondo un insieme di livelli di astrazione strutturati gerarchicamente, possiamo quindi parlare di *risorsa primitiva* e di *risorsa astratta*, a seconda che il tipo a cui essa appartiene sia un tipo primitivo o un tipo astratto.

Così definito, il concetto di risorsa coincide con quello di struttura dati allocata nella memoria comune della macchina concorrente. Ciò è ovviamente vero nel caso gli oggetti di tipo logico ma, come è noto dalla teoria dei sistemi operativi, questa considerazione vale anche nel caso di oggetti fisici, come per esempio le risorse periferiche per l'ingresso e l'uscita dei dati, la memoria e il processore. Ogni dispositivo periferico è infatti rappresentato in memoria da una struttura dati, il *descrittore del dispositivo*, su cui operano tutte le funzioni definite dal sistema operativo per accedere al dispositivo stesso. Analogamente, sia il processore virtuale di un processo che la sua memoria virtuale altro non sono che strutture dati allocate nella memoria del sistema (vedi per esempio il *descrittore del processo* per quanto riguarda il processore virtuale o le varie tabelle delle pagine e/o dei segmenti per la memoria).

Possiamo concludere, quindi, che ogni risorsa corrisponde a una struttura dati su cui i processi possono operare mediante le operazioni definite dal tipo della struttura, operazioni macchina (virtuale) nel caso di risorse di tipo primitivo e funzioni definite nell'ambito della definizione di tipo, nel caso di risorse di tipo astratto.

In base a queste considerazioni, possiamo anche ricapitolare che in una macchina concorrente, strutturata secondo il modello a memoria comune, ogni interazione tra processi, sia essa di tipo competitivo o cooperativo, non può che avvenire consentendo ai processi di accedere alle stesse risorse allocate nella memoria comune. Come è stato mostrato nel paragrafo 2.2, per evitare interferenze tra i processi, è però necessario regolare gli accessi alle risorse definendo i criteri in base ai quali sia possibile stabilire quando un processo possa legalmente accedere a una risorsa e quando viceversa, non gli debba essere consentito. Ciò anche al fine di fornire le indicazioni che consentano al meccanismo di controllo degli accessi di effettuare i propri controlli di correttezza per prevenire una delle cause di interferenza tra processi. Per questo motivo, nel prossimo paragrafo verrà introdotto il concetto di *diritto di accesso* di un processo a una risorsa e verranno illustrati i criteri in base ai quali tali diritti vengono assegnati ai processi. Sarà così possibile abilitare il meccanismo di controllo degli accessi a svolgere il proprio ruolo di verifica, come indicato nella figura 4.1. In pratica, ogni volta che un processo `P` tenta di accedere a una risorsa `R`, per esempio invocando l'operazione `op` (una delle operazioni definite dal tipo di `R`), il meccanismo verifica se, in quell'istante, `P` possiede il diritto di accesso per operare su `R`. In caso positivo, l'operazione viene eseguita e portata a termine altrimenti la stessa viene abortita, sollevando un'eccezione al fine di evitare possibili interferenze con altri processi.

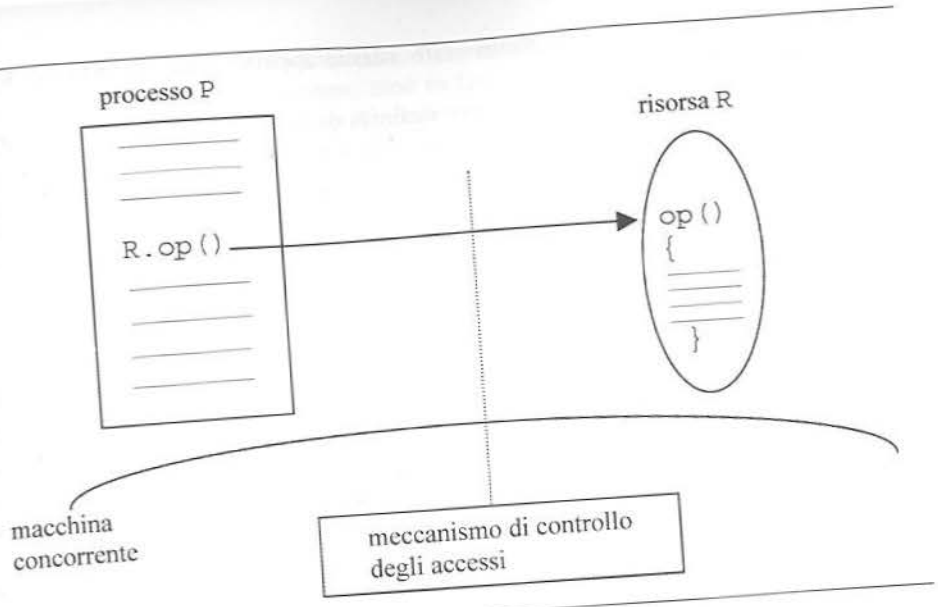


Figura 4.1 Meccanismo di controllo degli accessi.

4.2 Gestore di una risorsa

Al fine di distribuire i diritti di accesso alle risorse tra i vari processi di un'applicazione concorrente, per ogni risorsa `R` viene introdotto il concetto di *gestore* (o *allocatore*) della risorsa, il cui compito è quello di indicare, istante per istante, quali processi hanno il diritto di operare su `R` e quali no. Formalmente possiamo definire, per ogni risorsa `R`, questa nuova entità, il suo gestore G_R, il cui compito è quello di definire, in ogni istante t, l'insieme $S_R(t)$ dei processi che possiedono, in tale istante, il diritto di operare su `R`.

Se la cardinalità dell'insieme $S_R(t)$ è sempre minore o uguale a 1, cioè quando non più di un processo alla volta ha il diritto di operare su `R`, definiamo `R` come *risorsa dedicata* (all'unico processo che in ogni istante può operare su di essa). Viceversa, definiamo `R` come *risorsa condivisa* se $S_R(t)$ viene definito dal gestore in modo tale da poter contenere più processi contemporaneamente. In quest'ultimo ca-

so, infatti, i vari processi, che in un certo istante appartengono all'insieme $S_R(t)$, condividono l'uso della risorsa.

Infine, se l'insieme S_R è una costante definita da G_R all'istante t_0 (istante iniziale dell'elaborazione), indicheremo che R viene *allocata staticamente*. Infatti, i processi che hanno il diritto di operare su R vengono specificati prima che l'applicazione inizi la propria esecuzione. Viceversa, definiremo R come risorsa *allocata dinamicamente* nel caso in cui $S_R(t)$ venga definito come una funzione del tempo. In questo caso l'insieme $S_R(t)$ viene definito da G_R all'istante t_0 come l'insieme vuoto:

$$S_R(t_0) = \{\Phi\}$$

Lo schema che viene seguito per consentire a un processo di entrare nell'insieme (per acquisire il diritto di operare su R) è quello di obbligare il processo a richiedere il diritto di accesso al gestore G_R. Sarà poi compito del gestore accettare tale richiesta, ritardarla o rifiutarla, in base alla strategia con cui la risorsa viene allocata.

Lo schema riportato nella figura 4.2 riepiloga i vari criteri di allocazione di una risorsa precedentemente definiti.

In particolare, ogni risorsa allocata con i criteri indicati con la lettera A nella figura 4.2 rappresenta una *risorsa privata* di un processo essendo accessibile a un solo processo (in quanto risorsa dedicata) e tale rimane per tutta la durata dell'esecuzione del programma (in quanto risorsa allocata staticamente). In tutti gli altri casi siamo in presenza di *risorse comuni* a più processi: comuni ai processi che ne condividono l'uso (risorse allocate secondo le modalità B e D), oppure comuni ai processi che dinamicamente, anche se uno alla volta, possono acquisire il diritto di operare su esse, nel caso di risorse allocate con modalità C.

Per il momento il gestore di una risorsa è stato definito, in modo molto generico, come quell'entità che ha il compito di allocare i diritti di accesso alla risorsa. Vediamo adesso di definire in maniera più precisa questo concetto a seconda delle modalità di allocazione della risorsa stessa.

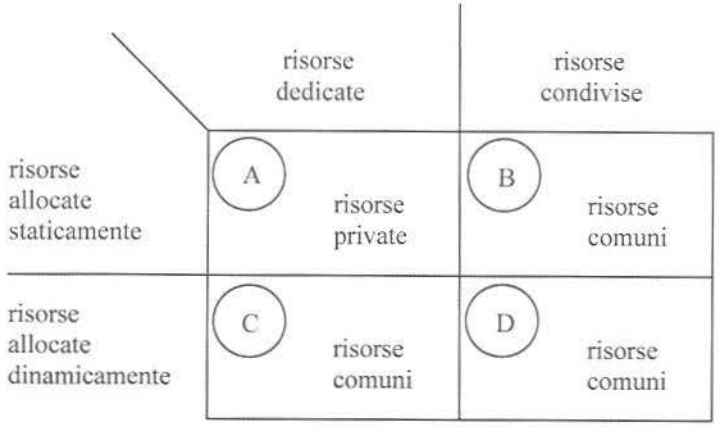


Figura 4.2 Tecniche di allocazione delle risorse.

Per ogni risorsa R allocata staticamente, l'insieme $S_R(t)$ è definito prima che il programma inizi la propria esecuzione, cioè in fase di stesura dello stesso programma. È quindi ovvio che in questo caso il gestore della risorsa è lo stesso programmatore che, in base alle regole di visibilità del linguaggio, stabilisce quale processo può "vedere" e quindi operare su R. Viceversa, per ogni risorsa R allocata dinamicamente, il relativo gestore G_R definisce l'insieme $S_R(t)$ in fase di esecuzione e, quindi, deve necessariamente essere un componente della stessa applicazione destinato a svolgere i seguenti compiti:

1. mantenere aggiornato l'insieme $S_R(t)$, cioè lo stato di allocazione della risorsa;
2. fornire i meccanismi che un processo può utilizzare per acquisire il diritto di operare sulla risorsa, entrando a far parte dell'insieme $S_R(t)$, e per rilasciare tale diritto quando non è più necessario;
3. implementare la strategia di allocazione della risorsa, cioè definire quando, a chi e per quanto tempo allocare la risorsa.

Se il gestore deve essere un componente dell'applicazione, può essere soltanto o un processo o, esso stesso, una risorsa. Vedremo di seguito che è proprio questa distinzione a caratterizzare i due modelli di macchina concorrente. In particolare, nel caso di architetture che seguono il modello a scambio di messaggi, il gestore di una risorsa è un processo mentre in quelle che seguono il modello a memoria comune, è una risorsa.

Il fatto che nel modello a memoria comune il gestore di una risorsa debba essere, esso stesso, una risorsa deriva dalla seguente considerazione: quando un processo P deve richiedere il diritto di accesso a una risorsa R, esso deve interagire col suo gestore G_R e, se questo fosse un processo, avremmo quindi un'interazione diretta tra i due processi P e G_R. Ma, come abbiamo visto, in questo tipo di architetture non è prevista nessuna forma di interazione diretta tra processi, ma sempre attraverso una risorsa comune a essi.

Il gestore di una risorsa viene quindi definito come l'istanza di un nuovo tipo astratto di risorse e, come tale, costituito da una struttura dati e da un insieme di funzioni. La struttura dati è destinata a mantenere aggiornato lo stato della risorsa gestita (vedi il precedente punto 1), mentre le funzioni vengono invocate da un processo per acquisire o rilasciare il diritto di operare sulla risorsa gestita (precedente punto 2). Infine, la strategia di allocazione della risorsa (vedi precedente punto 3) dipende dal modo in cui le funzioni di richiesta e rilascio dei diritti di accesso vengono implementate.

Fino a ora abbiamo parlato di gestore di una risorsa. È però opportuno mettere in evidenza che, in alcuni casi, più risorse possono essere allocate da uno stesso gestore. Ciò si verifica, in particolare, quando le risorse sono tutte istanze di uno stesso tipo e, quindi, in grado di svolgere tutte le stesse funzioni. In questi casi, può capitare che un processo applicativo abbia necessità di operare su una di queste risorse senza necessariamente specificare quale di esse in particolare, essendo tutte equivalenti. Per esempio, se in un sistema esistono più dispositivi di stampa e se ogni processo che deve stampare non ha necessità di specificare una particolare stampante, allora è opportuno gestire l'intero insieme delle stampanti mediante un solo gestore. In tutti questi casi parleremo di gestione di un *pool di risorse equivalenti*.

Stabilito che il gestore di una risorsa è esso stesso una risorsa, resta da chiarire con quali criteri questo viene a sua volta allocato. Come si può facilmente capire, un gestore viene allocato normalmente in modo statico e ciò per evitare che lo stesso abbia a sua volta bisogno di un gestore. D'altra parte, non può che essere una risorsa condivisa (in particolare tra tutti i processi che possono richiedere diritti sulla risorsa gestita). Con riferimento alla figura 4.2, la modalità di gestione indicata con la lettera B rappresenta quindi la tipica modalità con cui viene programmato un gestore.

Come abbiamo messo in evidenza all'inizio di questo capitolo, in architetture a memoria comune ogni interazione tra processi avviene tramite accessi a risorse comuni. Vediamo, quindi, che tipo di interazioni possono avvenire tra i processi che operano su una risorsa comune e che tipo di sincronizzazione deve essere prevista fra i processi quando gli stessi accedono alla risorsa, a seconda delle varie modalità di gestione della risorsa stessa indicate nella figura 4.2.

Indichiamo quindi con P un processo che, a un certo istante, deve operare su una risorsa R. Supponiamo anche che T sia il tipo della risorsa e che op_1, op_2, ..., op_n siano le n diverse funzioni, specifiche nell'ambito del tipo T, che possono essere utilizzate da P per accedere a R. Vediamo adesso i diversi paradigmi che devono essere seguiti nello scrivere il codice di P, relativamente a suoi accessi a R, a seconda delle diverse modalità di gestione di R.

Se R è allocata staticamente a P (modalità A e B di figura 4.2), il processo possiede il diritto per operare su R in qualunque istante del proprio ciclo di vita. Per cui, ogni volta che deve eseguire, per esempio, l'operazione op_i su R può correttamente limitarsi a invocare la funzione:

```
R.op_i(...);
```

Viceversa, se R è allocata dinamicamente a P (modalità C e D di figura 4.2), è necessario prevedere il gestore di R, cioè l'ulteriore risorsa G_R. Supponiamo, in questo caso, che `richiesta()` e `rilascio()` siano le funzioni definite per accedere a G_R, funzioni che P deve invocare rispettivamente prima di iniziare a operare su R e dopo aver terminato tali operazioni. In questo caso, il processo P deve seguire il seguente protocollo per operare sulla risorsa R:

```
G_R.richiesta(...);
R.op_i(...);
G_R.rilascio(...);
```

Ovviamente, dopo aver terminato l'esecuzione della funzione `richiesta`, quindi una volta ottenuto dal gestore il diritto per operare su R, e prima di invocare la funzione `rilascio`, il processo P può accedere a R tante volte quanto è necessario.

Per quanto riguarda gli accessi a R da parte di P, se R è allocata come risorsa condivisa (modalità B e D di figura 4.2), è necessario garantire che tali accessi avvengano in modo non divisibile essendo la risorsa accessibile contemporaneamente da parte di più processi. Se la risorsa è di tipo primitivo, ciò è garantito dall'atomicità delle operazioni eseguite sulla risorsa. Nel caso però che la risorsa sia di tipo astratto, è necessario evitare che due diversi processi che ne condividono l'uso accedano contemporaneamente a R, al fine di evitare le interferenze dovute agli stati inconsistenti attraverso i quali la risorsa passa mentre un processo sta operando su di lei. In altri termini, le funzioni di accesso alla risorsa devono essere programmate, in questo caso, come una *classe di sezioni critiche*, come indicato nel precedente paragrafo 2.2, utilizzando il meccanismo di sincronizzazione indiretta, che deve essere offerto dal linguaggio di programmazione e supportato dalla macchina concorrente.

Per quanto riguarda, infine, gli accessi a R da parte di P, se R è allocata come risorsa dedicata (modalità A e C di figura 4.2), essendo P l'unico processo che accede alla risorsa quando possiede il diritto per operarci, non è necessario prevedere nessuna forma di sincronizzazione tra P e gli altri processi durante l'accesso a R.

Concludendo, le risorse allocate con modalità A di figura 4.2, in quanto risorse private, non consentono nessuna forma di interazione tra processi. Per questo motivo, non saranno prese ulteriormente in considerazione nel corso del testo. Analogamente, trascureremo le risorse allocate con modalità D, in quanto raramente utilizzate in pratica: comunque per esse valgono tutte le considerazioni che vedremo per le altre due categorie che, viceversa, costituiscono quelle più utilizzate per quanto riguarda le interazioni tra processi. Di seguito, quindi, cerchiamo di mettere in evidenza qualche caratteristica ulteriore delle due principali modalità di gestione di una risorsa: la modalità B (risorse condivise e allocate staticamente) e la modalità C (risorse dedicate e allocate dinamicamente). In entrambe i casi l'uso di tali risorse da parte di più processi implica delle interazioni tra i processi stessi e, di conseguenza, la necessità di sincronizzazione.

Il primo tipo di interazione è sicuramente costituito dalla competizione. Nel caso di una risorsa R condivisa, come è stato messo in evidenza precedentemente, i processi devono operare su R in mutua esclusione, competendo quindi tra di loro negli accessi a R. Anche nel caso di una risorsa R gestita con modalità C sorge la competizione tra processi. In questo caso essa non avviene tanto durante gli accessi a R che, in quanto risorsa dedicata, come è già stato detto, non presuppone nessuna forma di sincronizzazione, quanto nel momento in cui gli stessi devono accedere al gestore G_R per ottenere i diritti di accesso a R, essendo il gestore, come abbiamo visto, una risorsa astratta di tipo condiviso su cui è quindi necessario operare in mutua esclusione.

In entrambe i casi la competizione ha lo scopo di garantire che sulla risorsa R operi un solo processo alla volta. Nel caso di risorse allocate con modalità B, è il meccanismo di mutua esclusione, utilizzato nel programmare le funzioni di accesso a R, che garantisce gli accessi esclusivi alla risorsa; nell'altro caso è il gestore G_R che, opportunamente programmato, svolge questa funzione, assegnando il diritto di operare su R a uno solo dei processi richiedenti alla volta. Può quindi nascere il dubbio del perché sono previste due diverse tecniche per ottenere lo stesso risultato. In effetti, in molti casi le due modalità, come vedremo anche nel seguito, sono del tutto equivalenti e quando ciò si verifica è sicuramente da privilegiare la modalità B, più semplice dal un punto di vista implementativo, in quanto non richiede la programmazione esplicita dell'ulteriore componente costituito dal gestore G_R. Ci sono però almeno due casi in cui questa equivalenza non vale e per i quali è la modalità C la più conveniente.

Il primo caso è quello relativo alla gestione di un pool di n risorse equivalenti, per esempio le risorse R_1, R_2, ..., R_n tutte dello stesso tipo T. Supponiamo infatti che ciascuna di queste risorse sia gestita con modalità B, cioè allocata staticamente come

una risorsa condivisa fra tutti i processi che durante la loro esecuzione devono operare su una qualunque risorsa del pool. In questo caso, quando uno qualunque di processi, per esempio P, deve operare su una delle risorse del pool eseguendo su essa la funzione op(), avendo già il diritto di operare su ciascuna di esse, dovrà sceglierne una qualunque, per esempio quella di indice i. Ciò significa che nel codice eseguito di P comparirà l'istruzione:

R_i.op();

Può però accadere che, quando il processo esegue questa istruzione, sulla risorsa R_i stia già operando un altro fra i processi che ne condividono l'uso. In questo caso, in seguito al meccanismo di mutua esclusione, P deve attendere che su R_i non operi più nessuno. Questa attesa può risultare del tutto inutile se, in quel momento, fossero disponibili altre risorse del pool. L'inconveniente nasce dal fatto che la scelta della risorsa da usare viene effettuata durante la scrittura del codice relativo al processo P e, ovviamente, non è possibile in quella fase prevedere quale fra le risorse del pool sarà libera e quale occupata nel momento in cui l'istruzione verrà eseguita. Viceversa, allocando tutte le risorse del pool come risorse dedicate mediante un unico gestore, il problema viene automaticamente risolto. Infatti, adesso P, quando ha bisogno di una risorsa, chiede il permesso al gestore, il quale può registrare nella propria struttura dati quale risorsa del pool è libera e quale occupata e, quindi, allocare al processo richiedente una qualunque fra le risorse disponibili, ovviamente se ce ne sono.

Il secondo caso riguarda la gestione di una risorsa R (o anche di un pool risorse), ma in modo tale da allocarla ai vari processi che ne richiedono l'uso, implementando una qualche strategia di priorità fra i processi stessi. Anche in questo caso la gestione con modalità B non si presta a risolvere il problema. Per esempio, supponiamo che R venga gestita come risorsa condivisa tra i processi P_a, P_b e P_c e che, negli accessi a R, si voglia privilegiare P_a rispetto a P_b e quest'ultimo rispetto a P_c. Facciamo quindi l'ipotesi che, in un certo istante, la risorsa sia libera e che P_c invochi un'operazione su R. Non essendo in esecuzione altre operazioni sulla risorsa, il processo può iniziare la sua operazione. Se però, durante questa esecuzione, sia P_a che P_b invocano a loro volta un'operazione su R, le esecuzioni di queste operazioni devono essere ritardate per mutua esclusione e quindi i processi P_a e P_b si devono bloccare in attesa che termini l'operazione eseguita da P_c. Quando questa termina, il meccanismo di mutua esclusione può abilitare l'esecuzione di una delle due operazioni sospese, o quella invocata da P_a o quella invocata da P_b. Per rispettare la priorità specificata dal problema, dovrebbe essere garantito che per prima venga abilitata l'esecuzione dell'operazione invocata da P_a. Purtroppo però il meccanismo di mutua esclusione, come avremo modo di vedere nel seguito, viene programmato utilizzando il meccanismo di sincronizzazione primitivo offerto dalla macchina concorrente, il quale non può garantire che, quando più processi sono in attesa di un evento e questo si verifica, venga abilitato a proseguire per primo un processo piuttosto che un altro. Chiaramente una scelta viene effettuata, in base alla tecnica con cui il meccanismo stesso è stato implementato. Ma questa scelta non può essere quella più adatta a tutti i diversi casi. In particolare, se, come nel nostro esempio, è prevista una ben precisa regola di priorità, è necessario di nuovo ricorrere alla tecnica di gestione di tipo C, programmando esplicitamente un gestore della risorsa. In questo caso sarà compito di chi scrive il codice del gestore G_R garantire che, nel momento in cui un processo rilascia la risorsa al gestore, se ci sono altri processi in attesa, venga svegliato quello a più alta priorità.

Infine, entrambe le modalità B e C di gestione di una risorsa R possono anche consentire ai processi di cooperare, per esempio scambiando informazioni. Nel caso di una risorsa condivisa, la cooperazione può avvenire se uno dei processi memorizza, nella struttura dati di R, delle informazioni che possono essere successivamente lette da un altro dei processi che condividono R. Ma la stessa forma di cooperazione può avvenire nel caso in cui la risorsa R sia gestita con modalità C. In questo caso, un processo, una volta acquisito da G_R il diritto di operare su R, può memorizzare informazioni nella struttura dati di R. Successivamente il diritto di accesso può essere acquisito da un altro processo che può quindi leggere tali informazioni. Questo tipo di interazione implica però che i processi si sincronizzino anche in forma diretta per evitare, per esempio, che un processo tenti di leggere delle informazioni che non sono state ancora memorizzate dal processo mittente.

Possiamo quindi concludere questa breve rassegna delle varie modalità di interazione, e quindi di sincronizzazione, tra processi con la seguente affermazione: *ogni interazione tra processi avviene sempre ed esclusivamente per il tramite di una risorsa condivisa.*

Nel caso di risorse gestite con modalità B, ciò è ovvio per la stessa definizione di questa modalità. Ma, come abbiamo visto, la stessa cosa si verifica anche nel caso di allocazione di una risorsa con modalità C, in quanto ogni forma di sincronizzazione tra i processi avviene, in questo caso, durante gli accessi al gestore della risorsa, gestore che a sua volta è una risorsa condivisa. Nei successivi capitoli possiamo quindi limitare l'esame dei meccanismi di interazione e sincronizzazione tra processi agli accessi a risorse condivise.

4.3 Specifica della sincronizzazione

Come abbiamo visto nel precedente paragrafo, processi diversi che accedono a una stessa risorsa condivisa devono sincronizzarsi per due diverse ragioni. La prima riguarda la necessità di eseguire le funzioni di accesso in mutua esclusione (sincronizzazione indiretta). Nel caso però in cui un processo utilizzi la risorsa per cooperare con altri processi, è necessaria anche una seconda forma di sincronizzazione diretta per garantire la corretta cooperazione tra i processi. Per esempio, abbiamo visto che, nel caso di una comunicazione, il processo destinato a ricevere le informazioni deve attendere che tali informazioni siano state inviate da parte del processo mittente prima di poterle ricevere. Come avremo modo di verificare nei successivi capitoli, questa seconda forma di sincronizzazione viene sempre specificata in modo tale da garantire che l'esecuzione di certe operazioni sulla risorsa condivisa siano consentite esclusivamente quando la stessa si trova in alcuni stati interni che soddisfano ben determinate condizioni.

Per chiarire questo aspetto, facciamo riferimento a un semplice esempio. Supponiamo di riservare in memoria la risorsa condivisa M tra due processi, mittente e

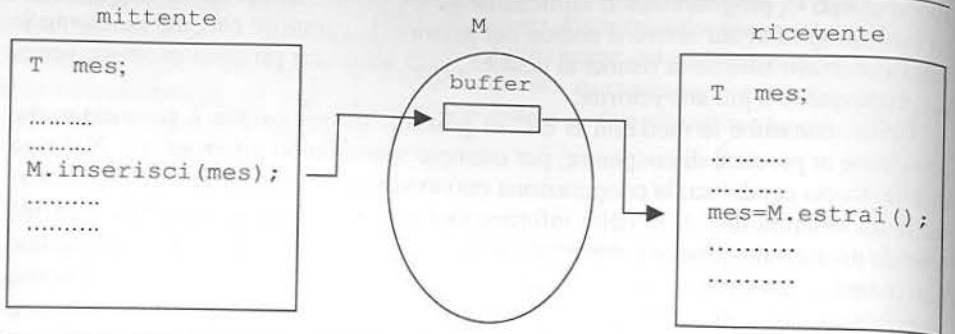


Figura 4.3 Scambio di informazioni tra processi.

`ricevente`, e che tale risorsa sia utilizzata dai due processi per abilitare `mittente` a inviare periodicamente dati di un certo tipo `T` a `ricevente`. Per esempio, potremmo definire il tipo di `M` come un nuovo tipo astratto, una classe la cui struttura dati coincida con un `buffer` di tipo `T`, in cui il processo `mittente` possa inserire il dato `mes` da inviare, eseguendo la funzione `inserisci`, e `ricevente` possa leggere tale dato, eseguendo la funzione `estrai` (vedi figura 4.3).

In questo caso però, per assicurare una corretta interazione tra i due processi, non è sufficiente garantire la mutua esclusione tra le esecuzioni di `inserisci` e `estrai`. È infatti necessario evitare anche che `estrai` venga eseguita da `ricevente` prima che `mittente` abbia a sua volta eseguito `inserisci`, altrimenti il dato estratto da `buffer` sarebbe privo di significato. È quindi necessario aggiungere alla struttura dati di `M` anche altri campi, per esempio il campo `pieno` di tipo `boolean` (inizializzato al valore `false`), che deve essere posto al valore `true` alla fine dell'esecuzione della funzione `inserisci`, per indicare la presenza di un dato da ricevere nel `buffer` e che deve essere posto al valore `false` alla fine dell'esecuzione della funzione `estrai`, per indicare che il dato presente nel `buffer` è già stato letto. Con questa modifica possiamo quindi specificare la sincronizzazione diretta tra i due processi, indicando che la funzione `estrai` può essere eseguita esclusivamente quando la struttura dati di `M` è in uno stato interno in cui è verificata la condizione (`pieno==true`) e che, analogamente, la funzione `inserisci` può essere eseguita soltanto quando lo stato interno di `M` verifica la condizione (`pieno==false`). Ogni tentativo di eseguire una funzione quando la risorsa non si trova in uno stato interno che verifica la condizione per la sua corretta esecuzione deve implicare la sospensione del processo che la esegue, in attesa che la condizione diventi vera.

L'esempio precedente, pur essendo un caso particolare, mette in evidenza una caratteristica del tutto generale dei meccanismi di sincronizzazione diretta, cioè che i vincoli imposti da questi meccanismi possono essere sempre specificati in termini di condizioni relative agli stati interni della risorsa condivisa utilizzata per far cooperare i processi.

Introduciamo adesso un formalismo del tutto generale che ci consentirà, nei successivi capitoli, di specificare i vari vincoli di sincronizzazione a cui dovranno essere soggetti i processi che, operando su una risorsa condivisa, dovranno interagire per risolvere correttamente un qualunque problema, sia di cooperazione che di competizione. A questo fine faremo riferimento a una variante del formalismo delle regioni critiche, originariamente introdotto da Hoare [12]. Tale formalismo è molto generale ed è in grado quindi di consentire la specifica di qualunque vincolo di sincronizzazione anche se, proprio in virtù di questa generalità, è particolarmente difficile da implementare in modo efficiente come costrutto di un linguaggio di programmazione. Per questo motivo, useremo tale meccanismo esclusivamente come strumento di specifica dei vari problemi di sincronizzazione.

Supponiamo che `R` sia una risorsa di un certo tipo `T` condivisa fra alcuni processi che tramite `R` devono interagire. In questo caso, ciascuna delle operazioni che i processi possono invocare per operare su `R` verrà specificata mediante il seguente formalismo:

`region R<<`S_a`: when(C)` S_b`:>>`

che specifica i seguenti vincoli si sincronizzazione:

a) quanto contenuto all'interno delle doppie parentesi angolari (che chiameremo il corpo della `region`), rappresenta un'operazione da eseguire sulla risorsa condivisa `R` e, quindi, costituisce una regione critica che deve essere eseguita in mutua esclusione con le altre operazioni definite su `R`.

b) il corpo di tale operazione è costituito da due istruzioni da eseguire in sequenza: l'istruzione S_a e, successivamente, l'istruzione S_b. Inoltre, tale specifica indica anche che, una volta terminata l'esecuzione di S_a, viene valutata la condizione `C` e, soltanto se questa è vera, l'esecuzione continua con S_b mentre, se `C` è falsa, il processo che ha invocato l'operazione si sospende in attesa che la condizione `C` diventi vera. A quel punto, l'esecuzione della `region` può riprendere ed essere completata mediante l'esecuzione di S_b.

La condizione `C`, che deve essere verificata per eseguire S_b, viene anche indicata come *condizione di sincronizzazione* e rappresenta un'espressione di tipo `boolean` che è vera soltanto quando la risorsa `R` si trova in particolari stati interni.

Il formalismo indicato precedentemente rappresenta una `region` nella sua forma più generale. Non è necessario però che il corpo della `region` sia sempre costituito dalle tre componenti S_a, S_b e `C`; in alcuni casi alcune di queste componenti possono mancare. Per esempio, il caso più semplice di `region` è il seguente:

`region R<<S:>>`

che rappresenta un'operazione da eseguire su `R` in mutua esclusione senza nessuna ulteriore specifica di sincronizzazione diretta. Questa specifica è tipica di ogni operazione su una risorsa condivisa, tramite la quale non viene attuata nessuna forma di cooperazione tra processi. Per esempio, un'operazione di stampa su una stampante condivisa che prevede esclusivamente la competizione per mutua esclusione.

Anche nel caso in cui sia prevista una qualche forma di sincronizzazione diretta, per cui è necessario specificare la condizione di sincronizzazione, la forma più usuale è quella in cui manca la prima istruzione S_a:

`region R<<when(C) S:>>`

Infatti, la condizione di sincronizzazione, che caratterizza lo stato in cui deve trovarsi la risorsa R al fine di poter eseguire una certa operazione, viene normalmente valutata prima di iniziare l'esecuzione dell'operazione stessa.

Infine, la forma:

```
region R<<when(C)>>
```

in cui è presente la sola condizione C, indica un semplice vincolo di sincronizzazione e specifica che il processo deve attendere che la condizione C sia verificata per proseguire la propria esecuzione.

Il fatto che la clausola when(C) di una region sia contenuta all'interno della regione critica implica che la valutazione della condizione C deve avvenire, essa stessa, in mutua esclusione. Ciò è necessario in quanto tale condizione rappresenta un'espressione di tipo boolean relativa allo stato interno della risorsa condivisa R e, se non fosse valutata in mutua esclusione, la sua valutazione potrebbe entrare in collisione con una qualunque altra operazione su R che ne modifichi concorrentemente lo stato.

Con riferimento al precedente esempio, relativo allo scambio di informazioni tra i processi mittente e ricevente, possiamo adesso specificare la sincronizzazione indotta nei due processi quando gli stessi accedono alla risorsa M. Si tratta, in questo caso, di specificare i vincoli di sincronizzazione che sono associati alle due funzioni inserisci e estrai, che rappresentano i due metodi di accesso a M. In particolare, se la struttura dati di M è costituita dai due campi:

```
T buffer;
boolean pieno;
```

con il significato precedentemente indicato, possiamo specificare le due funzioni nel seguente modo:

```
void inserisci(T dato):region M<<when(pieno==false)
   buffer=dato;pieno=true;>>
T estrai(): region M<<when(pieno==true)pieno=false;
   return buffer;>>
```

La semantica della clausola when di una region è molto simile a quella di un ciclo while. Infatti, per esempio, la seguente region:

```
region R<<when(C) S;>>
```

specifica che, prima di eseguire l'istruzione S, la condizione C deve essere verificata. In altri termini, ciò significa che C fa parte della precondizione di S. Lo stesso risultato, allora, lo potremmo ottenere con la seguente region:

```
region R<<
   while(!C) continue;
   S;>>
```

il cui comportamento in termini di diagramma di flusso è riportato nella figura 4.4, dove l'attesa del processo che esegue tali istruzioni, quando la condizione C non è vera, viene implementata mediante un ciclo di attesa attiva, cioè mantenendo il processore virtuale del processo in attesa tramite un ciclo al cui interno la condizione C viene continuamente valutata fino a quando non diventa vera.

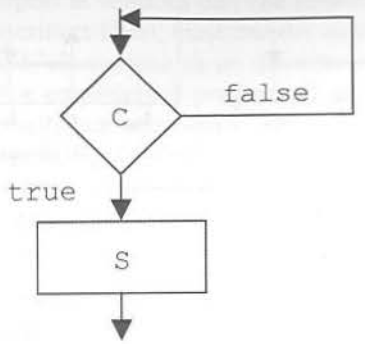


Figura 4.4 Attesa attiva.

In realtà, l'equivalenza fra la clausola when e il ciclo di attesa attiva while è soltanto apparente. Esiste, infatti, una differenza semantica sostanziale tra i due costrutti. Infatti, il fatto che la condizione C debba essere valutata in mutua esclusione implica che la valutazione di C e, se C è vera, la successiva esecuzione di S debbano avvenire in modo non divisibile rispetto ad altre operazioni su R. Analogamente, la valutazione di C e la decisione di non proseguire se C è falsa deve essere non divisibile. Però, in quest'ultimo caso, la mutua esclusione su R deve essere rilasciata, altrimenti nessun altro processo potrà operare su R e rendere quindi vera C abilitando il processo in attesa a proseguire. Nel caso del ciclo while, viceversa, essendo quest'ultimo all'inteno della regione critica, ciò non si verifica e quindi il processo che lo esegue entra in un ciclo da cui non esce più e cioè entra in una condizione di stallo. In altri termini la clausola when corrisponde semanticamente a un while ma con rilascio della mutua esclusione nell'ipotesi in cui C non sia vera e, successivamente, con il rientro in mutua esclusione quando C sarà di nuovo valutata. Vedremo, nei successivi capitoli, che questa necessità di uscire dalla mutua esclusione per rientrarci successivamente può essere causa di errori dovuti a corse critiche fra i processi che competono per operare su R, se l'implementazione di questo schema non viene effettuata prendendo opportuni accorgimenti.

4.4 Tecniche di allocazione delle risorse

Nella figura 4.2 sono state schematizzate tutte le possibili modalità di gestione delle risorse. Ciò, però, è vero limitatamente alle risorse di cui deve occuparsi chi progetta e scrive il codice di un'applicazione concorrente. Infatti, se prendiamo in considerazione anche altri tipi di risorse, in particolare quelle che normalmente vengono gestite dal sistema operativo, cioè dal supporto offerto dalla macchina concorrente, allora è necessario estendere quanto indicato precedentemente. Per rendersi conto di ciò, riportiamo di seguito due affermazioni che sono state fatte nel paragrafo 4.2 e che cadono in difetto se facciamo riferimento a risorse gestite direttamente dal sistema operativo come, per esempio, le risorse fisiche o anche le strutture dati del nucleo del sistema:

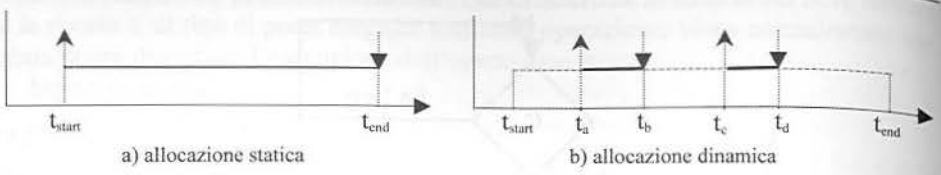


Figura 4.5 Allocazione a) statica e b) dinamica di una risorsa.

1. solo per le risorse allocate dinamicamente è necessaria la disponibilità di moduli di programmna che rappresentano i gestori. In particolare, come abbiamo visto, tali moduli sono delle risorse nel caso del modello a memoria comune.
2. se una risorsa è allocata dinamicamente, un processo acquisisce il diritto di operarci soltanto richiedendolo esplicitamente al gestore e mantiene tale diritto fino a quando non decide, ancora esplicitamente, di rilasciarlo.

Per capire perché tali affermazioni cadono in difetto, rivediamo i significati di allocazione statica e allocazione dinamica facendo riferimento alla figura 4.5.

Nella figura 4.5a) viene schematizzata l'allocazione statica di una risorsa R a un processo P, mentre nella figura 4.5b) è schematizzato un esempio di allocazione dinamica. In entrambi i casi, l'attività del processo viene rappresentata mediante una linea orizzontale che inizia all'istante t_{start}, istante in cui il processo viene creato, e termina all'istante t_{end}, in cui il processo completa la sua esecuzione. La linea che rappresenta il processo è inoltre riportata in forma tratteggiata quando P non possiede il diritto di operare sulla risorsa, mentre è riportata in forma continua quando può operare su R. Gli istanti di allocazione e deallocazione della risorsa sono rappresentati mediante una piccola freccia rivolta, rispettivamente, verso l'alto e verso il basso.

Conformemente al significato di allocazione statica, nella figura 4.5a) il processo è rappresentato da un linea unita poiché possiede il diritto di operare su R per tutta la sua durata. Infatti l'istante di allocazione di R coincide con la creazione del processo e quello di deallocazione con la sua terminazione. Viceversa, nella figura 4.5b) che rappresenta lo schema di allocazione dinamica, il processo nasce senza il diritto di operare su R e solo dopo aver acquisito tale diritto (istante t_a) può operarci. Successivamente, (istante t_b) la risorsa viene deallocata e il processo perde il diritto di operarci per riacquisirlo successivamente (istante t_c) e poi perderlo di nuovo (istante t_d).

Prendiamo adesso in considerazione la precedente affermazione 1 con riferimento a una risorsa gestita staticamente ma direttamente dal sistema operativo. Facciamo per esempio il caso di un descrittore di processo. Come è noto e come è stato descritto anche nel precedente capitolo, il descrittore di un processo P è una struttura dati, e quindi una risorsa, che viene allocata a P quando lo stesso nasce e che rimane a sua esclusiva disponibilità fino a quando P termina. Ciò però non significa che non sia presente un modulo di programma nel nucleo del sistema che ha proprio il compito di gestire i descrittori di processo. Infatti, quando un processo termina il suo descrittore non viene eliminato ma deve essere gestito al fine di poterlo utilizzare per creare, successivamente, un nuovo processo. Come è stato mostrato nel precedente capitolo, esiste infatti una struttura dati, quella che abbiamo chiamato descrittori_liberi, che è proprio la struttura dati che caratterizza il gestore di un pool di risorse equivalenti, i descrittori liberi, ciascuno dei quali può essere utilizzato per creare un nuovo processo. L'allocazione di un descrittore a un processo P avviene durante la creazione di P e consiste nel prelievo di un elemento dalla lista descrittori_liberi e nella sua successiva inizializzazione con i dati relativi al processo da creare. Analogamente, la deallocazione del descrittore avviene durante l'esecuzione della primitiva di terminazione del processo e consiste nel reinserire il descrittore nella lista descrittori_liberi.

Esistono, in effetti, anche alcuni casi particolari di risorse che non prevedono nessun tipo di gestore. Queste sono esclusivamente quelle risorse che vengono allocate staticamente ai processi di sistema (i cosiddetti *deamon*) processi che non terminano mai, cioè che vengono creati in fase di inizializzazione del sistema operativo (*bootstrap*) e restano in vita fino allo *shut down* del sistema. È ovvio che per queste particolari risorse (allocate staticamente a processi che non terminano mai) non è necessaria nessuna forma di gestione. Per questi casi molto particolari parleremo di *allocazione assoluta* per distinguerli da quelli di allocazione statica vera e propria.

Vediamo adesso l'affermazione 2, secondo cui, nel caso di allocazione dinamica di una risorsa R, un processo acquisisce il diritto di operarci esclusivamente mediante un'esplicita richiesta al gestore di R. Anche in questo caso, se R è gestita direttamente dal sistema operativo tale affermazione può cadere in difetto. Ciò si verifica, in particolare, per quelle risorse fisiche, come il processore o la memoria principale, senza le quali perde di significato lo stesso concetto di processo. Come è noto, in un sistema multiprogrammato queste risorse sono allocate dinamicamente ai processi. Non avrebbe senso, però, dire che un processo richiede e rilascia esplicitamente al rispettivo gestore l'uso del processore o quello della memoria. Infatti, per richiedere tale uso il processo deve essere già in esecuzione e cioè deve già usare il processore e avere già il suo programma allocato in memoria. In questi casi, quindi, è il sistema operativo che, in base a certe considerazioni a cui accenneremo, decide di allocare la risorsa R a un processo ed è sempre il sistema operativo che decide quando revocare il diritto di accesso al processo per poterlo allocare ad altri. Per distinguere le due diverse tecniche di allocazione dinamica, parleremo di *allocazione dinamica senza revoca* con riferimento alle modalità di allocazione già esaminate nei precedenti paragrafi, mentre parleremo di *allocazione dinamica con revoca* se è il sistema operativo a decidere quando allocare la risorsa e quando revocarla.

Quest'ultimo tipo di allocazione dinamica crea però un'apparente contraddizione. Supponiamo che la risorsa R venga allocata ai processi con questa tecnica. Ciò significa che ogni processo P a cui R viene allocata non deve richiederla esplicitamente e quindi che il programma che P deve eseguire viene scritto senza fare mai riferimento al gestore di R. In altri termini, per chi scrive tale programma è come se R fosse sempre disponibile per P. Ma, in realtà ciò non accade. Come si può vedere dalla figura 4.5b), R è disponibile per P solo negli intervalli temporali $t_a - t_b$ e $t_c - t_d$. Questa contraddizione viene eliminata, per ogni risorsa fisica R gestita con tale tecnica, creando e gestendo un ulteriore tipo di risorsa, la *risorsa virtuale*, che viceversa viene allocata in modo statico a P. Una risorsa virtuale non è altro che una struttura dati che rappresenta la risorsa fisica R e di cui esiste una istanza per ogni processo, il quale quindi può essere programmato come se la risorsa R fosse sempre a sua

disposizione. In realtà ciò che è sempre a sua disposizione è la risorsa virtuale mentre la corrispondente risorsa fisica viene allocata dinamicamente con revoca. Quando la risorsa fisica R è allocata a P significa che R offre il supporto alla corrispondente risorsa virtuale di P. Quando, viceversa, R non è allocata a P, il processo non può essere eseguito e quindi viene sospeso, in modo del tutto trasparente al programmatore, fino a quando R non gli viene nuovamente allocata. Per questo motivo, l'allocazione dinamica con revoca è nota anche come tecnica di virtualizzazione delle risorse. Per esempio, il campo *contesto* di un descrittore di processo identifica la risorsa processore virtuale così come le varie tabelle delle pagine e/o dei segmenti rappresentano la memoria virtuale di un processo. Ogni istante in cui la risorsa reale viene revocata a un processo per essere allocata a una altro corrisponde a un cambio di contesto.

Infine, nel caso di allocazione dinamica con revoca, possiamo identificare due diverse tipologie di allocazione, se invece di considerare una sola risorsa reale siamo in presenza di un pool di risorse equivalenti. Per comprendere questa ulteriore distinzione, supponiamo che all'istante t_a di figura 4.5b) sia allocata al processo P una particolare istanza R_i del pool di risorse equivalenti, istanza revocata successivamente all'istante t_b. Quando poi, all'istante t_c una risorsa del pool deve di nuovo essere allocata al processo, ci possiamo chiedere se è necessario che venga allocata la stessa istanza R_i, relativa alla prima allocazione, o se, viceversa, è possibile allocare una qualunque altra istanza R_j in quel momento disponibile. Se siamo costretti ad allocare la stessa istanza relativa alla prima allocazione parleremo di allocazione, *senza rilocabilità* altrimenti parleremo di *rilocabilità dinamica*. Nella figura 4.6 viene riportato schematicamente l'intero insieme di tecniche di allocazione di una risorsa.

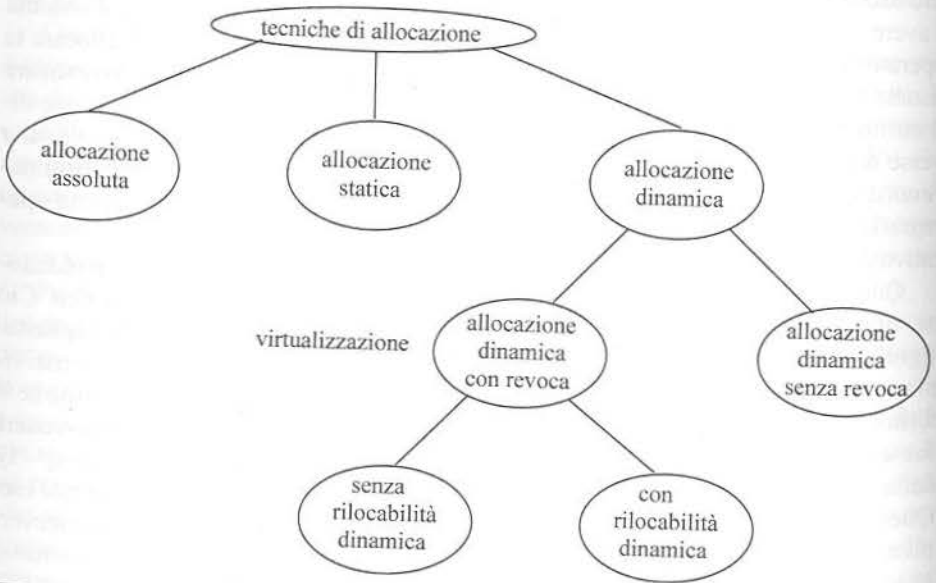


Figura 4.6 Tecniche di allocazione delle risorse.

Chiaramente, la rilocabilità dinamica consente una maggiore efficienza nella gestione di un pool di risorse equivalenti. Per esempio, nel caso di un sistema multiprocessore a processori anonimi, cioè con processori tutti uguali, è possibile rilocare dinamicamente il processore allocato a un singolo processo, nel senso che in prima allocazione può essere allocato uno specifico processore e, dopo un cambio di contesto, a una successiva allocazione, può essere allocato un qualunque altro processore del pool.

Nel caso della memoria non sempre ciò è possibile. Supponiamo, per esempio, di allocare la memoria con la tecnica delle partizioni e che, in fase di prima allocazione, venga utilizzata per creare il processo una particolare partizione disponibile e di dimensioni sufficienti. Supponiamo inoltre che, a un successivo istante, sia revocata la partizione al processo. In fase di successiva riallocazione, il sistema è costretto a riallocare la stessa partizione e non una qualunque fra quelle disponibili a meno che non esista un particolare dispositivo hardware, noto con l'acronimo MMU (*Memory Management Unit*), che viene utilizzato per consentire la rilocabilità dinamica. Per esempio, come è noto dalla teoria dei sistemi operativi, la coppia di registri base-limite.

Per concludere, mettiamo in evidenza alcune caratteristiche delle precedenti tecniche di allocazione.

Una prima osservazione è che la tecnica di allocazione statica può essere considerata come un caso limite della tecnica di allocazione dinamica con revoca. Questo caso limite coincide con quello in cui la risorsa reale e quella virtuale coincidono.

Nel caso di allocazione dinamica esistono sempre almeno due funzioni del gestore, quelle utilizzate, rispettivamente, per allocare e deallocare la risorsa. Se l'allocazione è senza revoca, tali funzioni sono invocate direttamente dal processo richiedente e quella di allocazione risulta bloccante per il processo, che la invoca se la risorsa richiesta non è al momento disponibile. Nel caso di allocazione con revoca tali funzioni, come abbiamo visto, non sono invocate dal processo e fanno parte delle funzioni del sistema operativo. Come tali vengono eseguite in seguito al verificarsi di alcuni eventi di sistema. Per esempio, nel caso del processore l'allocazione viene eseguita al cambio di contesto e, quindi, quando il processore viene revocato al processo in esecuzione, o perchè questo si blocca o, per esempio nei sistemi time-sharing, allo scadere del quanto di tempo del timer. Inoltre, quando la risorsa da allocare non è disponibile, invece di bloccare il processo a cui vogliamo allocare la risorsa, questa viene tolta per revoca a chi la possiede per allocarla quindi al processo prescelto. Si pensi, per esempio, al rimpiazzamento delle pagine o dei segmenti o, ancora, alla revoca del processore al processo in esecuzione quando, in un sistema gestito su base prioritaria, viene attivato un processo a priorità più alta rispetto a quella del processo in esecuzione.

Infine, anche un gestore che opera con la tecnica con revoca implementa i tre compiti a cui abbiamo accennato nel precedente paragrafo 4.2. Per esempio, nel caso del gestore del processore la struttura dati che mantiene aggiornato lo stato della risorsa coincide con la variabile di sistema `processo_in_esecuzione` (vedi capitolo precedente). I meccanismi per allocare e deallocare il processore coincidono con le funzioni `salvataggio_stato` e `ripristino_stato`. Infine, la strategia di allocazione viene implementata nella funzione di schedulazione `assegnazione_CPU`.

.5 Sommario

questo capitolo sono state esaminate le principali caratteristiche relative al model-
architetturale di macchina concorrente a memoria comune, con lo scopo identifi-
re le modalità utilizzate dai processi che girano su tali architetture virtuali per inte-
gire tra di loro, sia per competere sull'uso di risorse comuni sia per cooperare.
ueste caratteristiche, come vedremo nei due successivi capitoli, influenzano in ma-
ra sostanziale la tipologia di costrutti che un linguaggio, orientato secondo questo
odello, deve fornire per consentire la specifica delle interazioni tra processi. Sono
te introdotte le varie tipologie di gestione delle risorse e il concetto di gestore di
a risorsa. È stato introdotto il costrutto delle regioni critiche condizionali come
umento principale che verrà utilizzato, nei successivi capitoli, per la specifica del-
condizioni di sincronizzazione che devono essere verificate durante gli accessi a
orse condivise. Infine, le tipologie di allocazione delle risorse sono state estese
ricomprendere anche quelle utilizzate all'interno del nucleo di un sistema opera-
per gestire le risorse fisiche, introducendo il concetto di virtualizzazione delle
orse.

5 Note bibliografiche

introduzione alle caratteristiche del modello architetturale di macchina virtuale a
noria comune si trova in [41].
Una descrizione delle proprietà del modello e un suo confronto con il modello a
nbio di messaggi è presente nei lavori di Bryant-Dennis [42] e Lauer-Nedham
].
La definizione dei concetti di risorsa condivisa e di allocazione di una risorsa è
ritta in [44] e [45].
'introduzione dei costrutti di regione critica e di regione critica condizionale è
uta a Hoare [12] [46] e Brinch Hansen [23] [24].
Una descrizione approfondita dei concetti di virtualizzazione delle risorse e dei
ivi meccanismi di allocazione delle risorse fisiche all'interno del nucleo di un si-
a operativo si trova in [31].

5
Semafori

Scopo di questo capitolo è quello di descrivere il meccanismo di sincronizzazione dei semafori, introdotto originariamente da Dijkstra [30], che rappresenta sicuramente lo strumento di sincronizzazione più noto nell'ambito dei sistemi organizzati secondo il modello a memoria comune. È infatti il meccanismo più famoso fra quelli implementati nel nucleo di un sistema operativo. Come tale, rappresenta uno strumento di "basso livello" che ben si presta a essere offerto come meccanismo primitivo dalla macchina concorrente. Per lo stesso motivo, però, è più difficile che venga utilizzato come strumento linguistico da inserire nell'ambito di un linguaggio di alto livello.

Come vedremo, proprio in virtù della sua estrema generalità, può essere utilizzato per risolvere qualunque problema di sincronizzazione tra processi. Vedremo diverse tipologie di problemi di sincronizzazione e, per ciascuna di queste, uno schema di soluzione che prevede uno specifico paradigma di uso del meccanismo semaforico. Questi diversi paradigmi di uso dei semafori vengono spesso identificati anche con nomi specifici che ne richiamano le proprietà: *semafori binari di mutua esclusione*, *semafori evento*, *semafori binari composti*, *semafori condizione*, *semafori risorsa*, *semafori privati*.

5.1 Definizione del meccanismo semaforico

Anche se è raro il caso in cui i semafori sono utilizzati come strumento linguistico, è comunque opportuno esaminare in dettaglio questo importante strumento di sincronizzazione; da una parte perché è normalmente questo il meccanismo utilizzato a livello di macchina concorrente per implementare strumenti di sincronizzazione di più alto livello, dall'altra perché è spesso disponibile all'interno di librerie standard, che consentono di implementare programmi concorrenti anche utilizzando linguaggi sequenziali, per esempio il C o il C++, e chiamando le funzioni della libreria sia per la gestione della concorrenza che della sincronizzazione. Vedi per esempio la libreria `Pthread` definita dallo standard POSIX.

Per definire questo meccanismo è necessario introdurre un nuovo tipo di dati, il tipo `semaphore`. Per il momento definiremo questo nuovo tipo come un tipo primitivo offerto quindi dalla macchina concorrente. Forniremo perciò la specifica del funzionamento del meccanismo in modo tale da definirne la semantica in modo preciso. Ciò consentirà di descrivere il suo corretto uso per fornire la soluzione ai più usuali problemi di interazione tra processi. Vedremo poi, in un successivo paragrafo, come il meccanismo venga implementato all'interno del nucleo di un sistema operativo.

Come ogni altro tipo, anche il tipo `semaphore` è caratterizzato da due insiemi: quello dei valori che ogni oggetto del tipo può assumere e quello delle operazioni che possono essere utilizzate per operare su oggetti di questo tipo. In particolare lo possiamo definire nel seguente modo:

1) insieme dei valori = {tutti i valori interi ≥ 0}
2) insieme delle operazioni = {`P`, `V`}

Un oggetto di tipo `semaphore`, in quanto strumento di sincronizzazione tra processi, è sempre condiviso tra due o più processi che lo utilizzano per sincronizzare le loro velocità di esecuzione. Un processo esegue l'operazione `P` su un semaforo per testare se è vera una condizione, concettualmente associata al semaforo e necessaria per la sua prosecuzione, sospendendo la propria esecuzione in caso negativo e rimanendo in attesa che la condizione diventi vera. L'operazione `V` viene utilizzata per attivare l'esecuzione di un processo precedentemente sospeso su una `P`.[1]

Da un punto di vista sintattico, nel seguito di questo capitolo useremo `semaphore` come un tipo predefinito. In fase di dichiarazione di un semaforo `s` ne specificheremo il valore iniziale `i` ($i \geq 0$) assegnando tale valore a `s`:

```
semaphore s=i;
```

Le altre operazioni che potranno essere eseguite su `s` sono quindi esclusivamente la `P` e la `V`, che verranno denotate mediante la seguente notazione funzionale: `P(s)` e `V(s)`.

Lo stato interno di un oggetto `s` di tipo `semaphore` è caratterizzato dal suo valore (un intero positivo o nullo che a livello di specifica indicheremo con val_s) e dai processi che, eventualmente, hanno sospeso la loro esecuzione su `s` eseguendo su di esso l'operazione `P` (sempre a livello di specifica, il numero di tali processi verrà indicato con $bloccati_s$).

La semantica delle due operazioni può essere così specificata:

```
void P(semaphore s):region s<<when(val_s>0)val_s--;>>        (5.1)
void V(semaphore s):region s<<val_s++;>>                     (5.2)
```

[1] La lettera *P*, che denota la prima operazione sui semafori, rappresenta l'iniziale della parola olandese *proberen* che significa testare mentre la *V* è l'iniziale della parola *verhogen* che significa incrementare. In molti casi per indicare le operazioni semaforiche, invece di usare `P` e `V`, vengono usati gli identificatori wait e signal. Abbiamo preferito utilizzare `P` e `V` per evitare confusione con le funzioni `wait` e `signal` che verranno introdotte nel successivo capitolo sui monitor.

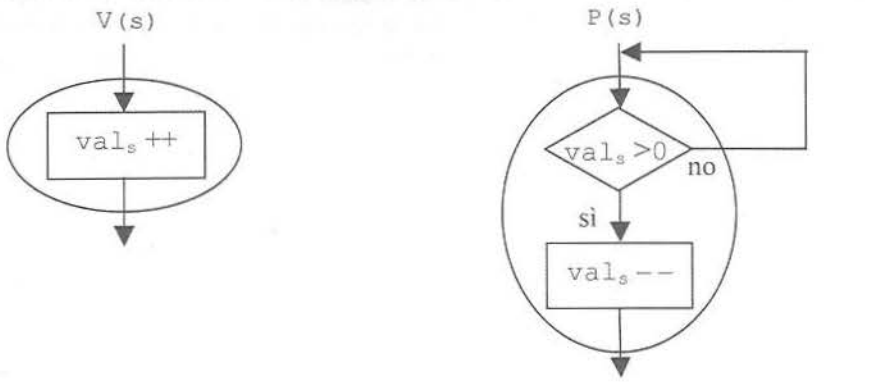


Figura 5.1 Operazioni `P` e `V`.

Essendo l'oggetto `s` condiviso, le due operazioni vengono definite come due sezioni critiche da eseguire in mutua esclusione. La mutua esclusione delle due operazioni è assicurata dal fatto che le stesse sono operazioni primitive, in quanto offerte dalla macchina concorrente, e quindi atomiche. Per eseguire la `P`, che decrementa il valore di `s`, è però necessario che val_s sia positivo altrimenti, dopo il decremento, `s` assumerebbe un valore negativo, contraddicendo il fatto che i valori di un semaforo devono essere sempre maggiori o uguali a zero. Nessuna condizione viene viceversa specificata per l'operazione `V` che, incrementando il valore di `s`, non può mai violare la precedente condizione.

Il comportamento delle due operazioni semaforiche può essere quindi rappresentato dagli schemi a blocchi riportati nella figura 5.1, dove il simbolo grafico ovale che racchiude le operazioni rappresenta il fatto che tali operazioni devono essere eseguite in forma atomica.

Come rappresentato nella figura, la `P` verifica se il valore del `semaforo` è maggiore di zero e, in questo caso, lo decrementa. La verifica e il successivo decremento devono avvenire atomicamente altrimenti due processi potrebbero trovare contemporaneamente il valore del semaforo uguale a uno e quindi entrambi decrementarne il valore portandolo, erroneamente, a -1. Se il valore è nullo l'atomicità dell'operazione viene interrotta conformemente alla semantica della clausola `when` descritta nel paragrafo 4.3. Il processo che esegue la `P` in questo caso non prosegue ma resta in attesa continuando a leggere il valore fino a quando non lo trova positivo dopo che un altro processo, eseguendo la `V`, lo ha incrementato.

Questa descrizione del funzionamento del meccanismo semaforico, come già anticipato, è puramente astratta. Descrive, cioè, il comportamento delle operazioni semaforiche così come viene percepito da chi le usa, astraendo da come le stesse vengono realizzate. L'unica ipotesi che è stata fatta è quella di considerare il meccanismo come primitivo, cioè offerto direttamente dalla macchina concorrente. La caratteristica saliente di questa macchina, come è stato già messo in evidenza, è quella di offrire tanti processori (virtuali) quanti sono i processi che su essa girano. In questo senso, ogni processo che, eseguendo una `P`, si deve sospendere resta in attesa attiva,

continuando a eseguire sul suo processore il ciclo di lettura del valore del semaforo finché non diventa positivo e gli consente di proseguire. Come è ben noto dalla teoria dei sistemi operativi, e come avremo modo di richiamare nel seguito, a livello di macchina concorrente e in particolare nel nucleo del sistema, questa attesa attiva non corrisponde alla realtà se facciamo riferimento al processore reale. Infatti, quando un processo non può proseguire, subisce una commutazione di contesto che ha il compito di togliere il processore reale al processo che sta girando e di assegnarlo ad altro processo attivo. Tutto ciò è però del tutto trasparente per il programma concorrente e, agli effetti della sua correttezza, possiamo ritenere valida la schematizzazione precedente in quanto ogni implementazione reale del meccanismo ne garantisce la validità.

Un semaforo è quindi molto simile a un contatore il cui valore viene decrementato mediante la `P` e incrementato mediante la `V`. Per completare la specifica del meccanismo introduciamo adesso alcune metavariabili per definire le relazioni invarianti e quindi le relazioni di consistenza del tipo semaforo (col significato di metavariabile e invariante di un tipo visti nel paragrafo 1.4.2):

- I_s: valore intero ≥ 0 con cui il semaforo `s` viene inizializzato;
- nv_s: numero di volte che l'operazione `V(s)` è stata eseguita;
- ni_s: numero di volte che l'operazione `P(s)` è stata iniziata;
- np_s: numero di volte che l'operazione `P(s)` è stata completata;

Le due principali relazioni invarianti, che definiscono le due grandezze val_s e $bloccati_s$, sono:

$$bloccati_s = ni_s - np_s \tag{5.3}$$

$$val_s = I_s + nv_s - np_s \tag{5.4}$$

La (5.3) stabilisce che il numero di processi sospesi sul semaforo `s`, in un certo istante, corrisponde alla differenza tra il numero di volte cha la `P(s)` è stata iniziata e quello in cui è stata terminata. Infatti, come indicato nella figura 5.1, un processo si blocca su un semaforo `s` quando, in fase di esecuzione della `P(s)`, trova il valore del semaforo uguale a zero e quindi resta in ciclo non completando tale operazione.

La (5.4), in base alla semantica delle due operazioni semaforiche, definisce il valore del semaforo in un certo istante come differenza tra il suo valore iniziale aumentato del numero di `V(s)` eseguite (ogni `V` implica un incremento del valore del semaforo) e il numero di `P(s)` completate (ogni `P` completata implica un decremento del valore del semaforo). Poiché, per definizione, il valore è sempre positivo o nullo, ne deriva che la seguente relazione è anch'essa invariante:

$$np_s \leq I_s + nv_s \tag{5.5}$$

Per completare il modello di comportamento di un semaforo, possiamo mettere in evidenza due ulteriori semplici relazioni invarianti che legano tra loro le due grandezze val_s e $bloccati_s$. In particolare, per quanto abbiamo già visto, fino a quando un semaforo ha un valore positivo allora non potranno esserci processi sospesi sullo stesso poiché ogni `P` potrebbe terminare senza attese (vedi relazione (5.6)). Per questo motivo un semaforo col valore positivo viene anche indicato come *semaforo verde* mentre uno col valore nullo come *semaforo rosso*. Analogamente, se in un certo istante ci sono processi bloccati allora è evidente che il valore del semaforo deve essere rosso:

$$val_s > 0 \Rightarrow bloccati_s == 0 \tag{5.6}$$

$$bloccati_s > 0 \Rightarrow val_s == 0 \tag{5.7}$$

Come vedremo, in base all'uso che ne viene fatto, un semaforo può essere limitato nei valori che lo stesso può assumere. Per esempio parleremo di *semaforo binario* quando il semaforo può assumere soltanto i valori zero e uno. Al contrario, un *semaforo generale* può assumere qualunque valore intero non negativo. Infine, un semaforo, che all'atto dell'esecuzione di un'operazione `P` è sempre rosso, verrà indicato come *semaforo bloccante*. Di seguito, illustreremo alcuni esempi di uso dei semafori, per risolvere sia problemi di competizione che di cooperazione, e useremo le precedenti relazioni per verificare la correttezza delle soluzioni. In particolare, mostreremo alcuni paradigmi di uso del meccanismo semaforico, ciascuno adatto a risolvere una diversa categoria di problemi di sincronizzazione tra processi, identificando ciascuno di essi con un diverso nome che ne caratterizza le proprietà e le modalità di uso.

5.2 Semafori di mutua esclusione

Se un semaforo viene inizializzato al valore uno e se ogni processo che lo utilizza esegue prima l'operazione `P` e successivamente la `V`, allora il semaforo è sicuramente binario. Infatti, non potrà mai assumere un valore maggiore di uno poiché ogni operazione `V` che ne incrementa il valore è sempre preceduta da una `P` che, terminando, lo decrementa.

Un tipico esempio di uso di un semaforo binario è quello relativo alla soluzione del caso più semplice di competizione tra processi, che riguarda la necessità di evitare accessi contemporanei a una risorsa condivisa se questa è costituita da un oggetto di un tipo astratto. Come è stato già mostrato, accessi contemporanei implicherebbero in questo caso interferenze dovute agli stati inconsistenti dell'oggetto. È quindi necessario che le operazioni definite per operare sulla risorsa costituiscano una classe di sezioni critiche. Questo problema, conosciuto come il problema della *mutua esclusione*, viene risolto molto facilmente, come è noto dalla teoria dei sistemi operativi, associando alla risorsa un semaforo `mutex` (noto anche come *semaforo di mutua esclusione*) inizializzato a uno e racchiudendo tutte le operazioni sulla risorsa tra una `P(mutex)` e una `V(mutex)`. Il semaforo `mutex` viene concettualmente associato alla seguente condizione di sincronizzazione:

<nessuna sezione critica della classe è in esecuzione>

Se `mutex` è verde la condizione è vera, se è rosso è falsa. La `V(mutex)`, eseguita alla fine di ogni operazione sulla risorsa, rende vera la precedente condizione, mentre la `P(mutex)`, se la condizione è falsa, blocca il processo che la esegue e, se è vera, abilita il processo a entrare in mutua esclusione rendendo contemporaneamente falsa la condizione.

Per esempio, se con la seguente classe `tipo_risorsa` denotiamo il tipo astratto a cui appartiene la risorsa e indichiamo con $op_1()$, $op_2()$, ..., $op_n()$ le funzioni

definite per operare su oggetti della classe, è sufficiente dichiarare il semaforo `mutex` fra i campi della struttura dati della classe e racchiudere il corpo di tutte le funzioni tra le due operazioni su `mutex`:

```
class tipo_risorsa {
        <struttura dati di ogni istanza della classe>;
        semaphore mutex=1;
    public void op_1() {
            P(mutex);
            <corpo della funzione op_1>;
            V(mutex);
        }
    ..............................
    public void op_n() {
            P(mutex);
            <corpo della funzione op_n>;
            V(mutex);
        }
}
```

Dichiarato, quindi, un oggetto `ris` di questa classe:

```
tipo_risorsa ris;
```

ogni processo che ne condivide l'uso può operare su essa invocando uno dei metodi della classe:

```
ris.op_i();
```

avendo la garanzia che l'operazione viene eseguita in mutua esclusione.

Nel riquadro 5.1 vengono verificate le proprietà di correttezza del protocollo di mutua esclusione, ipotizzando che il semaforo `mutex` venga utilizzato esclusivamente nel modo indicato precedentemente, cioè che sia un semaforo binario. Al tempo stesso, abbiamo anche implicitamente assunto che l'esecuzione di ogni sezione critica termini sempre. Se infatti un processo, una volta entrato in sezione critica, non terminasse la propria operazione non consentirebbe l'ingresso in sezione critica a nessun altro, generando quindi condizioni di stallo. Quindi, tutte le volte che un'operazione, che potenzialmente può bloccare un processo, viene eseguita all'interno di una sezione critica è necessario verificare che la corrispondente operazione sbloccante non appartenga a una sezione critica della stessa classe. Infatti, un processo che si bloccasse all'interno della propria sezione critica non rilascerebbe la mutua esclusione, precludendo quindi l'esecuzione della sezione critica contenente l'operazione sbloccante e generando inevitabilmente una condizione di stallo.

Se il semaforo `mutex`, utilizzato secondo lo schema del riquadro 5.1, fosse stato inizializzato a un valore maggiore di uno, per esempio al valore `i`, seguendo lo stesso ragionamento, potremmo facilmente mostrare che al più `i` processi potrebbero essere contemporaneamente in esecuzione all'interno delle rispettive sezioni critiche. In questo caso, ovviamente, il semaforo non sarebbe più binario ma generale.

Riquadro 5.1

VERIFICA DELLA CORRETTEZZA DEL PROTOCOLLO DI MUTUA ESCLUSIONE

Vediamo adesso come, utilizzando la tecnica delle asserzioni, anche se in maniera informale, sia possibile verificare alcune proprietà della classe prima definita. Agli effetti della correttezza degli accessi a ogni oggetto della classe, dobbiamo verificare le seguenti proprietà:

a) è garantita la mutua esclusione;
b) è garantita l'assenza di condizioni di stallo;
c) a un processo, che vuole operare su un oggetto della classe quando nessuno ci sta operando, deve essere garantito di non subire ritardi inutili;
d) devono essere evitate condizioni di attesa indefinita (*starvation*).

Le prime tre sono proprietà di safety mentre l'ultima è una proprietà di liveness.

Per verificare le proprietà di un programma che usa i semafori si devono tener presenti le seguenti considerazioni:

1. le relazioni invarianti di ogni semaforo sono sempre vere;
2. in genere, è possibile ricavare altre relazioni tra le metavariabili del modello semaforico sulla base del modo in cui ogni semaforo viene utilizzato, cioè in base a dove compaiono le operazioni `P` e `V` nel testo del programma.

Per esempio, volendo verificare la proprietà a), quella della mutua esclusione, possiamo tener conto di due relazioni: la relazione invariante del semaforo `mutex` (vedi la relazione (5.5) dove il valore iniziale adesso è 1):

$$np_{mutex} \leq 1 + nv_{mutex} \tag{5.8}$$

Inoltre, per come viene usato `mutex`, per ogni accesso a `ris` viene sempre eseguita una `P(mutex)` prima del corpo dell'operazione e una `V(mutex)` alla fine. Quindi, poiché ogni processo che opera su `ris` deve prima passare la `P` iniziale e solo dopo, alla fine, eseguire la `V`, ciò significa che il numero di `P` completate sarà sempre maggiore o uguale al numero di `V` eseguite:

$$np_{mutex} \geq nv_{mutex} \tag{5.9}$$

Più in particolare, possiamo asserire che $np_{mutex} == nv_{mutex}$ quando nessun processo opera su `ris` (cioè hanno iniziato a operare su `ris` tanti quanti sono quelli che hanno terminato). Viceversa, avremo $np_{mutex} > nv_{mutex}$ se qualcuno sta operando su `ris` e, più precisamente, se ipotizziamo che siano `j` i processi che operano contemporaneamente su `ris`, allora deve essere $np_{mutex} == nv_{mutex} + j$. Poiché però, per la precedente relazione invariante (5.8), è sempre vero che $np_{mutex} \leq 1 + nv_{mutex}$, ne segue che `j` non può che essere o zero o uno. Sarà zero quando nessuno opera su `ris`, sarà uno quando un processo lavora da solo, e quindi in mutua esclusione, sull'oggetto.

Verifichiamo adesso la proprietà b) della soluzione, quella per cui questa non introduce condizioni di stallo. Le uniche condizioni di stallo che la soluzione al problema di mutua esclusione potrebbe introdurre sono quelle in cui tutti i processi si bloccano sull'unica operazione bloccante che è presente nella soluzione stessa, cioè la `P(mutex)`, quando tentano di operare sulla risorsa condivisa. Al tempo stesso, affinché questi blocchi corrispondano a una condizione di stallo nessun processo dovrebbe essere più svegliato e questo è vero solo se nessuno sta operando sulla risorsa. Infatti, se qualcuno operasse sulla risorsa alla fine dell'esecuzione dell'operazione, eseguendo `V(mutex)`, sveglierebbe uno dei processi bloccati che quindi non sarebbe in condizione di stallo. Ricapitolando, affinché sia presente uno stallo devono essere vere due condizioni:

a) i processi si bloccano su `mutex` quando, invocando una delle operazioni su `ris`, iniziano l'operazione stessa con `P(mutex)`. Questo implica che ogni processo trovi rosso il semaforo `mutex`, cioè per esso vale la relazione $val_{mutex}==0$ quando il processo tenta di eseguire un'operazione su `ris`.

b) nessun processo sta operando sulla risorsa. Ciò implica, per quanto già visto precedentemente, che $np_{mutex}==nv_{mutex}$ e quindi, per la relazione invariante (5.4), che $val_{mutex}==I_{mutex}+nv_{mutex}-np_{mutex}==1$ essendo uno il valore iniziale di `mutex`. Tutto ciò contraddice quanto dedotto dalla condizione a).

Applicando lo stesso ragionamento possiamo verificare la terza proprietà, la c), per cui un processo, che vuole operare su `ris` quando nessuno ci sta operando, non deve subire ritardi. Infatti, dalla precedente condizione b) deriva che, se nessuno opera su `ris`, allora $val_{mutex}==1$, cioè `mutex` è verde. Ma allora un processo che voglia operare su `ris` non potrà subire ritardi poiché la `P(mutex)` iniziale non potrà essere bloccante.

Per quanto riguarda la proprietà d), cioè l'assenza di condizioni di starvation, non è possibile effettuare nessuna verifica limitandoci a esaminare il codice relativo all'uso del semaforo `mutex`. Infatti, una volta che un processo si blocca su `mutex` potrebbe permanere nello stato bloccato per un tempo indeterminato, se insieme a lui fossero bloccati anche altri processi e se il criterio con cui viene scelto il processo da sbloccare all'atto di una `V` non lo prendesse mai in considerazione. Ma ciò dipende esclusivamente da questo criterio (*scheduling*), cioè dal modo in cui il meccanismo semaforico viene implementato all'interno della macchina concorrente. Come vedremo in seguito, illustrando le tecniche di realizzazione del meccanismo semaforico all'interno del nucleo del sistema operativo, tale fenomeno può essere facilmente eliminato adottando strategie di scheduling opportune, per esempio quella *First-In-First-Out*.

5.2.1 Mutua esclusione fra gruppi di processi

Il classico problema della mutua esclusione, visto precedentemente, prevede che un solo processo alla volta possa accedere a una risorsa condivisa. Il motivo di tale limitazione, come è noto, è dovuto al fatto che, mentre è in esecuzione un'operazione op_i sulla risorsa `ris`, questa è normalmente in stato inconsistente e, quindi, non è verificata la precondizione di nessuna operazione prevista per operare su `ris`, ivi inclusa la stessa op_i. Ci possono essere, però, dei casi particolari di risorse per le quali i vincoli legati agli stati inconsistenti sono più laschi. Per esempio, può capitare che la struttura interna della risorsa e la tipologia delle operazioni previste dal suo tipo siano tali per cui l'esecuzione di una delle operazioni op_i invalidi la precondizione di ogni altra operazione op_j ma non quella della stessa operazione op_i. In questi casi è possibile consentire a processi diversi di eseguire concorrentemente la stessa operazione sulla risorsa, ma non operazioni diverse. Possiamo quindi definire un diverso problema di mutua esclusione, più generale di quello visto precedentemente: "data la risorsa condivisa `ris` e indicate con op_1, ..., op_n le `n` operazioni previste per operare su `ris`, vogliamo garantire che più processi possano eseguire concorrentemente la stessa operazione op_i mentre non devono essere consentite esecuzioni contemporanee di operazioni diverse".

Il problema può essere risolto facendo ancora ricorso ai semafori di mutua esclusione, anche se utilizzandoli in maniera leggermente diversa. Nel caso della semplice mutua esclusione, ogni operazione viene realizzata racchiudendola tra un prologo, `P(mutex)`, e un epilogo, `V(mutex)`. Anche adesso seguiremo lo stesso criterio:

```
public void op_i() {
    <prologo_i>;
    <corpo della funzione op_i>;
    <epilogo_i>;
}
```

ma differenziando il prologo e l'epilogo di ogni operazione op_i da quelli delle altre operazioni. In pratica, `<prologo_i>` deve sospendere un processo che ha invocato l'operazione op_i se sulla risorsa sono in esecuzione altre operazioni diverse da questa. Viceversa, deve consentire al processo di eseguire op_i se nessuno sta operando sulla risorsa oppure se vi stanno operando processi che hanno invocato la stessa operazione op_i. Alla fine dell'esecuzione di un'operazione op_i, il processo che sta terminando l'operazione, eseguendo `<epilogo_i>`, deve liberare la mutua esclusione solo se è l'unico processo in esecuzione sulla risorsa, oppure se è l'ultimo di un gruppo di processi che hanno eseguito la stessa operazione op_i. Per questo motivo faremo ancora riferimento a un semaforo `mutex` di mutua esclusione per realizzare, però, l'esclusione mutua non di un processo con tutti gli altri ma bensì di un gruppo di processi (quelli che devono eseguire una delle operazioni) rispetto a tutti quelli che devono eseguire operazioni diverse. Supponiamo, per esempio, di fare riferimento a chi invoca l'operazione op_i e vediamo come realizzare `<prologo_i>` e `<epilogo_i>`.

Nel prologo deve essere eseguita la `P(mutex)` per disabilitare altri processi dal procedere concorrentemente, ma solo se questi invocano operazioni diverse da op_i. Viceversa, se un secondo processo invoca di nuovo op_i questo deve essere abilitato a procedere concorrentemente. Per questo motivo, deve essere esentato da eseguire `P(mutex)`. Questo comportamento lo possiamo ottenere molto semplicemente riservando un contatore $cont_i$ inizializzato a zero e incrementandolo durante il `<prologo_i>` e, quindi, eseguendo `P(mutex)` solo se, dopo l'incremento, $cont_i$ vale uno (e cioè se chi sta eseguendo `<prologo_i>` è il primo di un gruppo di processi che devono eseguire op_i). Tutti gli altri processi del gruppo, saltando la `P(mutex)`, possono procedere concorrentemente. Chiaramente la variabile $cont_i$ è condivisa tra tutti i processi del gruppo e quindi il `<prologo_i>` diventa, esso stesso, una piccola sezione critica da eseguire in mutua esclusione fra tutti i processi del gruppo. Per questo è sufficiente riservare un semaforo m_i di mutua esclusione associato al gruppo (vedi figura 5.2).

```
public void op_i() {
        P(m_i);
        cont_i++;                          } <prologo_i>
        if (cont_i==1) P(mutex);
        V(m_i);
        <corpo della funzione op_i>;
        P(m_i);
        cont_i--;                          } <epilogo_i>
        if (cont_i==0) V(mutex);
        V(m_i);
}
```

Figura 5.2 Mutua esclusione fra gruppi di processi.

Chiaramente, se uno o più processi di un gruppo (per esempio i processi che hanno invocato op_i) sono in esecuzione allora il semaforo `mutex` è rosso. Se un processo di un altro gruppo (per esempio di coloro che devono eseguire op_j) invoca l'operazione richiesta, inizia il proprio prologo e, essendo il primo del suo gruppo, incrementa il proprio contatore $cont_j$ e si blocca su `mutex`, lasciando il semaforo di mutua esclusione del gruppo m_j al valore zero. Quindi gli altri eventuali processi del suo gruppo, eseguendo $\langle prologo_j \rangle$ si bloccheranno per mutua esclusione su m_j. Quando ciascuno dei processi che eseguono op_i termina la propria operazione, eseguendo $\langle epilogo_i \rangle$ decrementa il proprio contatore $cont_i$. In questo modo, quando l'ultimo processo del gruppo termina, $cont_i$ viene azzerato e il processo esegue una `V(mutex)` che abilita il primo processo di un altro gruppo a proseguire e così via.

Questo schema trova una pratica applicazione nella soluzione di un problema che si ritrova spesso nel progetto di un sistema operativo e in particolare di un file system. Un file è un tipico esempio di risorsa astratta che viene spesso condivisa tra processi diversi. Le operazioni che vengono definite per operare sul file sono tipicamente le due operazioni di `lettura` e di `scrittura`. Poiché la `lettura` non modifica nessuna struttura del file, non introduce nessun problema di inconsistenza del file. È quindi evidente che operazioni di `lettura` possono essere eseguite concorrentemente mentre, ovviamente, due scritture devono essere mutuamente esclusive come lo devono essere una scrittura con operazioni di lettura. Seguendo lo schema precedente, possiamo quindi associare al file un semaforo `mutex` e la coppia m_1 e $cont_1$ per implementare il prologo e l'epilogo in lettura. Ovviamente il prologo e l'epilogo in scrittura saranno costituiti dalle semplici `P(mutex)` e `V(mutex)` rispettivamente poiché le scritture devono costituire delle normalissime sezioni critiche (vedi figura 5.3).

Questo esempio è noto in letteratura col nome di problema dei *lettori/scrittori*. La soluzione descritta, pur essendo molto semplice, soffre però di un inconveniente qualora i processi siano ciclici, come è spesso il caso dei processi di sistema. Questa soluzione può infatti generare un problema di starvation se, dal momento in cui un lettore riesce a iniziare la lettura, altri processi lettori continuano a invocare l'operazione di `lettura`. In questi casi, ogni processo che voglia scrivere si blocca in attesa che l'ultimo lettore termini. Ma questo evento può non verificarsi mai se l'operazione di `lettura` continua a essere invocata, tenendo anche conto che un processo, terminata la sua lettura, se è ciclico prima o poi torna a leggere di nuovo. Come è evidente, questo inconveniente non viene eliminato neppure se ipotizziamo che la coda di attesa sul semaforo `mutex` sia gestita FIFO mentre, in questo caso, non esiste un simile problema per i processi lettori.

Vedremo in un successivo paragrafo, parlando di politiche di allocazione, come questo inconveniente possa essere eliminato adottando un diverso schema per risolvere il problema.

```
semaphore mutex=1;
semaphore m1=1
int cont1=0;
public void lettura() {
   P(m1);
   cont1++;
   if (cont1==1) P(mutex);
   V(m1);
   <lettura del file>;
   P(m1);
   cont1--;
   if (cont1==0) V(mutex);
   V(m1);
}

public void scrittura() {
   P(mutex);
   <scrittura del file>;
   V(mutex);
}
```

Figura 5.3 Problema dei *lettori/scrittori*.

5.3 Semafori evento: scambio di segnali temporali

Un diverso caso di uso di semafori binari riguarda la soluzione di problemi che prevedono lo scambio di segnali di sincronizzazione tra processi. Come semplice esempio, possiamo fare riferimento a due processi P_1 e P_2 ai quali vogliamo imporre un vincolo di sincronizzazione per cui l'operazione op_a sia eseguita da P_1 soltanto dopo che P_2 ha eseguito la sua operazione op_b (vedi grafo di precedenza riportato nella figura 5.4a).

Il vincolo di precedenza tra le operazioni op_a e op_b dei due processi può essere imposto usando un semaforo `sem` che, inizializzato a zero, sia associato all'evento "esecuzione di op_b". In questo caso P_1, subito prima di eseguire op_a, chiede, mediante l'esecuzione di una `P(sem)`, se l'evento si è già verificato, bloccandosi in caso contrario; P_2 invece, dopo aver eseguito op_b, segnala, mediante l'esecuzione di una `V(sem)`, che l'evento si è già verificato. Per questo motivo, un semaforo binario utilizzato in questo modo viene anche indicato come *semaforo evento*. Il grafo di precedenza diventa quello riportato nella figura 5.4b che ovviamente assicura il vincolo tra op_a e op_b.

Nel riquadro 5.2 viene verificato che l'uso del semaforo evento consente di rispettare correttamente il vincolo di precedenza tra op_a e op_b.

Un'applicazione pratica di questo tipo di cooperazione tra processi si verifica in alcuni esempi di sistemi in tempo reale dedicati al controllo di processi industriali. In questi casi, l'applicazione di controllo viene spesso strutturata in termini di un certo numero n di processi applicativi P_0, P_1, ..., P_{n-1}, ciascuno dei quali dedicato

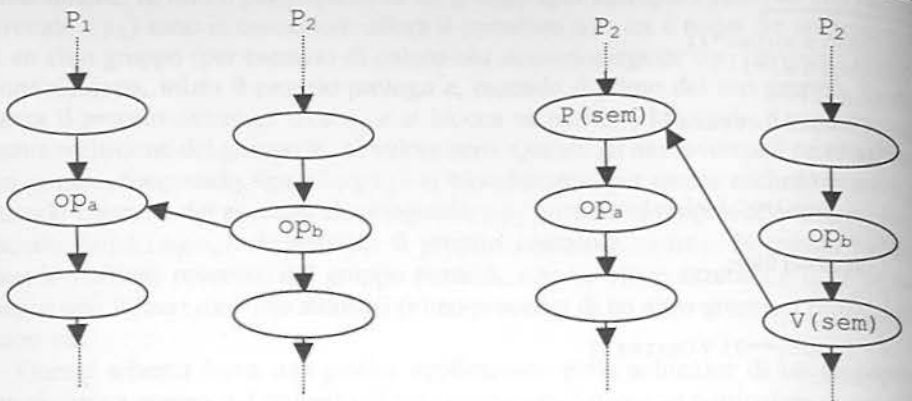


Figura 5.4 Scambio di segnali.

al controllo di una diversa grandezza fisica dell'impianto. Inoltre, normalmente i processi sono periodici, cioè vengono attivati a intervalli regolari da un processo di sistema dedicato a questo scopo (*processo schedulatore*) affinché continuino a eseguire il loro ciclo di controllo per tutta la vita del sistema. Il periodo tra due consecutive esecuzioni dello stesso processo dipende dalla costante tempo della grandezza fisica da controllare. Utilizzando il timer del sistema, è quindi compito del processo schedulatore attivare i processi applicativi inviando loro, a intervalli predefiniti, segnali temporali secondo lo schema precedente.

Un diverso esempio di uso dei semafori evento riguarda la soluzione di un classico problema di sincronizzazione tra processi, noto anche come problema del *rendez-vous*, cioè dell'appuntamento tra processi. Per esempio, supponiamo che due processi P_1 e P_2 eseguano ciascuno due operazioni: p_a e p_b il primo, q_a e q_b il secondo. Indipendentemente dalle loro velocità, vogliamo imporre che le esecuzioni di p_b da parte di P_1 e di q_b da parte di P_2 possano iniziare solo dopo che entrambi i processi hanno completato la loro prima operazione (vedi figura 5.6).

In definitiva, come illustrato nella figura 5.6, si tratta di adottare la precedente soluzione dello scambio di segnali temporali dove, in modo perfettamente simmetrico, ogni processo, quando arriva all'appuntamento, segnala di esserci arrivato e attende l'altro. Per questo è sufficiente utilizzare due semafori evento sem_1 e sem_2. Il primo associato all'evento "P_1 è arrivato all'appuntamento" il secondo all'evento "P_2 è arrivato all'appuntamento". Avremmo potuto adottare anche una diversa soluzione, non simmetrica, cioè, lasciando inalterato il codice di P_1, far eseguire a P_2 prima la $P(sem_1)$, ossia l'operazione di attesa che P_1 sia arrivato all'appuntamento, e successivamente, mediante la $V(sem_2)$, segnalare che anche lui è arrivato. Ovviamente per simmetria, questo scambio di operazioni lo avremmo potuto fare in P_1 invece che in P_2. Sarebbe però errato commutare le operazioni in entrambe i processi in quanto, se iniziassero tutti e due con una P, avremmo una condizione di stallo essendo i due semafori inizializzati a zero.

Riquadro 5.2

VERIFICA DELLA CORRETTEZZA DELLO SCAMBIO DI SEGNALI TEMPORALI

Si può dimostrare facilmente che, qualunque siano le velocità dei due processi, l'uso del semaforo evento garantisce il rispetto del vincolo di precedenza desiderato. Supponiamo infatti, per assurdo, che sia possibile che op_a venga eseguita prima di op_b. Allora, se riportiamo sull'asse dei tempi gli eventi generati da questa esecuzione (vedi figura 5.5) avremo che op_a precede op_b. Quindi `P(sem)`, che precede op_a, dovrebbe essere completata prima di `V(sem)`, la quale segue op_b.

Ma questa eventualità non potrà mai accadere. Infatti, in un qualunque istante successivo al completamento di `P(sem)`, e prima dell'inizio di `V(sem)` (vedi figura 5.5), sarebbe vera la relazione:

$$np_{sem} > nv_{sem}$$

essendo già completata la `P(sem)` e non ancora iniziata la `V(sem)`. Questa relazione però è assurda in quanto, essendo il valore iniziale del semaforo $I_{sem}=0$, viola l'invariante semaforico (±5).

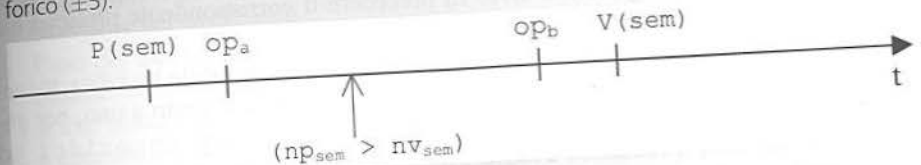


Figura 5.5 Esecuzione impossibile.

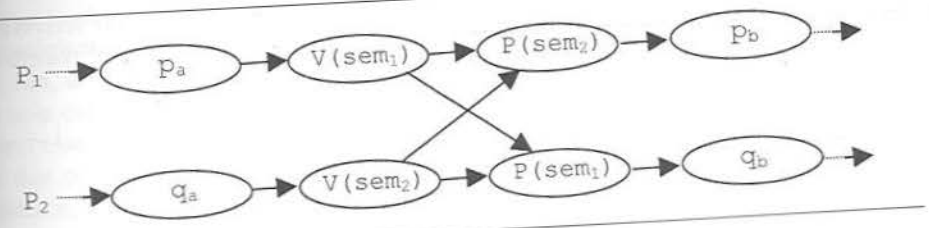


Figura 5.6 *Rendez-vous* tra processi.

5.4 Semafori binari composti: scambio di dati

Un ulteriore esempio di semafori binari riguarda la soluzione al classico problema dello scambio di dati fra processi. Supponiamo che due processi P_1 e P_2 si debbano, periodicamente scambiare un dato di un certo tipo generico `T`. Per risolvere il problema in un sistema a memoria comune è sufficiente riservare una variabile condivisa di tipo `T`, chiamiamola `buffer`, dove P_1 scrive il dato quando lo vuole inviare a P_2 e da cui P_2 lo preleva quando lo deve ricevere. Chiaramente una corretta soluzione del problema prevede che vengano rispettati alcuni vincoli di sincronizzazione:

a) gli accessi a buffer devono essere mutuamente esclusivi se T è un tipo astratto;
b) P_2 può prelevare un dato solo dopo che P_1 lo ha inserito;
c) P_1, prima di inserire un dato, deve attendere che P_2 abbia prelevato il precedente.

Nei precedenti paragrafi abbiamo visto come ciascuno dei tre vincoli può essere garantito. In particolare, per garantire il vincolo a), se con `void inserisci(T d)` e `T estrai()` indichiamo, rispettivamente, l'operazione di inserimento del dato `d` nel `buffer` da parte di P_1 e il prelievo e la restituzione di un dato da parte di P_2, è sufficiente riservare un semaforo di mutua esclusione associato a `buffer` e implementare `inserisci` ed `estrai` come due sezioni critiche. Inoltre i vincoli b) e c) sono due esempi di applicazione dei semafori evento, visti nel precedente paragrafo. Infatti, se nel grafo di precedenza di P_1 indichiamo con in_i e in_{i+1} gli eventi relativi a due consecutive esecuzioni di `inserisci` e, analogamente, con es_i e es_{i+1} indichiamo, nel grafo di P_2, i corrispondenti eventi relativi alle due consecutive esecuzioni di `estrai` (vedi figura 5.7), i vincoli di sincronizzazione b) e c) sono rappresentati dallo scambio di segnali temporali riportati nella figura tramite le frecce che impongono al generico invio di precedere il corrispondente prelievo e a questo di precedere il successivo invio.

Per risolvere il problema possiamo quindi associare alla variabile `buffer` tre semafori: il primo, il semaforo di mutua esclusione `mutex` inizializzato a uno, per garantire gli accessi esclusivi a `buffer` da parte delle funzioni `inserisci` ed `estrai`; il secondo, il semaforo evento `pn` inizializzato a zero, per garantire che il generico prelievo sia preceduto dal corrispondente inserimento; il terzo, il semaforo evento `vu` inizializzato a zero, per garantire che il generico inserimento sia preceduto dal precedente prelievo.

Usando, quindi, gli schemi visti nei precedenti paragrafi, possiamo scrivere le due funzioni `invio` e `ricezione`, che i processi P_1 e P_2 utilizzano, rispettivamente, per inviare e ricevere un dato sincronizzandosi in modo corretto:

```
void invio(T dato) {
   P(vu);
   inserisci(dato);
   V(pn);
}
```

```
T ricezione() {
   T dato;
   P(pn);
   dato=estrai();
   V(vu);
   return dato;
}
```

dove `inserisci` ed `estrai` operano su `buffer` in mutua esclusione tramite `mutex`.

In realtà è necessario apportare una modifica a questa soluzione altrimenti si va ncontro a una condizione di stallo. Infatti, se è vero che il generico inserimento de'e essere preceduto dal precedente prelievo, nel caso particolare del primo inserinento un precedente prelievo non esiste, quindi, per evitare uno stallo, il primo inseimento non deve attendere nessun evento. Se non vogliamo modificare il codice lella funzione `invio` per tener conto di questo caso particolare, possiamo sempliemente inizializzare il semaforo `vu` a uno in modo tale che la prima `P(vu)` non ia mai bloccante.

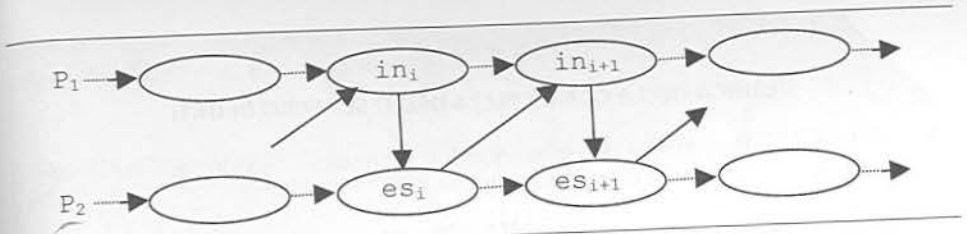


Figura 5.7 Vincoli di precedenza tra invio e ricezione di dati.

Con questa variante, nel riquadro 5.3 vengono dimostrate alcune proprietà di questa soluzione, fra cui la sua correttezza per quanto riguarda i vincoli di sincronizzazione. In particolare, usando le proprietà dei semafori, viene dimostrato che:

1. è garantito il precedente vincolo b) di sincronizzazione (assenza di condizioni di *underflow* del `buffer`);
2. è garantito il precedente vincolo c) di sincronizzazione (assenza di condizioni di *overflow* del `buffer`);
3. l'uso dei due soli semafori `vu` e `pn`, come sopra illustrato, garantisce che le esecuzioni di `inserisci` e `estrai` avvengono in maniera mutuamente esclusiva anche senza fare ricorso al semaforo `mutex` di mutua esclusione che quindi non è necessario;
4. la soluzione non genera condizioni di stallo.

Il fatto che i due semafori[2] `pn` e `vu` garantiscano da soli, per come sono usati, la mutua esclusione di `estrai` e `inserisci` si può anche dedurre, molto semplicemente, dal fatto che mai i due semafori sono contemporaneamente verdi. Infatti, inizialmente è verde `vu` e rosso `pn` e quindi solo `inserisci` può essere eseguita. Ma, dopo la sua esecuzione, è diventato verde `pn` e rosso `vu`, abilitando perciò la sola esecuzione di `estrai`, dopo di che si torna nella condizione iniziale. Possono essere ovviamente entrambi rossi, il che significa che è in esecuzione una delle due operazioni. In altri termini, questa coppia di semafori si comporta, nel suo insieme, come se fosse un unico semaforo binario di mutua esclusione anche se composto da due semafori diversi. Per questo motivo, si parla di *semaforo binario composto*. Questo particolare tipo di uso dei semafori può tornare utile in molti altri esempi di mutua esclusione, come vedremo. In pratica siamo in presenza di un semaforo binario composto tutte le volte che un insieme di semafori viene usato in modo tale che solo uno di essi venga inizializzato al valore uno e tutti gli altri al valore zero; inoltre ogni processo che usa questi semafori esegue sempre sequenze che iniziano con la `P` su uno di questi semafori e termina con la `V` su un altro. Ciò garantisce che tutte le operazioni incluse tra una `P` e la successiva `V` vengano eseguite in mutua esclusione.

[2] Il semaforo `pn` è verde quando `buffer` è pieno mentre `vu` è verde quando è vuoto.

Riquadro 5.3

VERIFICA DELLA CORRETTEZZA DELLO SCAMBIO DI DATI

Per effettuare queste verifiche è necessario ricorrere alle relazioni invarianti dei due semafori che, tenendo conto dei loro valori iniziali, sono:

$$np_{vu} \leq 1+nv_{vu} \quad (5.10)$$

$$np_{pn} \leq nv_{pn} \quad (5.11)$$

Inoltre, in base all'ordine con cui vengono eseguite le operazioni semaforiche nelle due funzioni `invio` e `ricezione`, possiamo asserire che in questa soluzione valgono le seguenti ulteriori relazioni:

$$np_{vu} \geq nv_{pn} \quad (5.12)$$

$$np_{pn} \geq nv_{vu} \quad (5.13)$$

la (5.12) perché nella `invio` si esegue prima `P(vu)` e poi `V(pn)` e la (5.13) perché nella `ricezione` si esegue prima `P(pn)` e poi `V(vu)`.

Iniziamo con la verifica delle precedenti proprietà 1 e 2. In pratica dobbiamo dimostrare che, per evitare underflow, ossia di estrarre da `buffer` un dato prima che questo sia stato inserito, il numero di inserimenti effettuati in un certo istante (che indicheremo con `nI`) sia sempre maggiore o uguale al numero di estrazioni (che indicheremo con `nE`). Analogamente, per evitare overflow del `buffer`, `nI` dovrà sempre essere minore, o al più uguale, a `nE+1`. Si tratta quindi di dimostrare che vale sempre la seguente relazione:

$$nE \leq nI \leq nE+1 \quad (5.14)$$

Poiché un inserimento viene eseguito nella funzione `invio` dopo la `P(vu)` iniziale e prima della `V(pn)` è ovvio che:

$$nv_{pn} \leq nI \leq np_{vu} \quad (5.15)$$

e, per le stesse considerazioni relative alla funzione `ricezione`:

$$nv_{vu} \leq nE \leq np_{pn} \quad (5.16)$$

Quindi, in base alla relazione (5.15), $nI \leq np_{vu}$ ma $np_{vu} \leq 1+nv_{vu}$ in base alla relazione (5.10) e $nv_{vu} \leq nE$ in base alla relazione (5.16). Mettendo insieme queste relazioni si ottiene $nI \leq np_{vu} \leq 1+nv_{vu} \leq 1+nE$ e, quindi, la parte destra della relazione (5.14) che volevamo dimostrare, cioè l'assenza di condizioni di overflow. Con analogo ragionamento, relativamente alle relazioni (5.16), (5.11) e (5.15) otteniamo che $nE \leq np_{pn} \leq nv_{pn} \leq nI$ cioè la parte sinistra della relazione (5.14) che dimostra l'assenza di condizioni di overflow.

Verifichiamo adesso la proprietà 3, per la quale la mutua esclusione fra `estrai` e `inserisci` è garantita dalla sola presenza dei due semafori `pn` e `vu`. Supponiamo infatti, per assurdo, che `estrai` e `inserisci` siano contemporaneamente in esecuzione. Se è in esecuzione `estrai`, significa che all'interno della funzione `invio` la `P` iniziale è già stata superata mentre la `V` finale non è stata ancora eseguita. Per cui la relazione (5.12), in questo caso, vale con il segno > in senso stretto (5.12'). Per lo stesso ragionamento, mentre è in esecuzione `inserisci`, anche la relazione (5.13) vale col segno > in senso stretto (5.13'). Quindi:

$$nv_{vu} \quad < \quad np_{pn} \quad \leq \quad nv_{pn} \quad < \quad np_{vu} \quad \leq \quad 1+nv_{vu}$$

relazione (5.13') relazione (5.11) relazione (5.12') relazione (5.10)

da cui risulta, tra nv_{vu} e nv_{pn}, la seguente relazione:

$$nv_{vu} < nv_{pn} < 1+nv_{vu}$$

ossia una grandezza intera (nv_{vu}) dovrebbe essere compresa tra due valori interi consecutivi (nv_{vu} e $1+nv_{vu}$), il che è palesemente assurdo.

Concludiamo con la prova della proprietà 4. Anche in questo caso, supporre, per assurdo, che i due processi siano in stallo conduce a una contraddizione. Infatti, se i due processi sono in stallo è perché ciascuno ha trovato rosso il semaforo su cui esegue la `P`. Quindi il valore dei due semafori è nullo e ciò significa che:

$$np_{vu} = 1+nv_{vu} \quad (5.10')$$

e

$$np_{pn} = nv_{pn} \quad (5.11')$$

D'altra parte, se i due processi sono bloccati, nessuno sta eseguendo le rispettive funzioni `invio` e `ricezione` e quindi il numero di volte che la `P` iniziale di ogni funzione è stata completata corrisponde al numero di volte in cui la `V` finale è stata eseguita:

$$np_{vu} = nv_{pn} \quad (5.12'')$$

$$np_{pn} = nv_{vu} \quad (5.13'')$$

Componendo queste relazioni, in particolare la (5.13'') con la (5.11'), questa con la (5.12'') e infine con la (5.10'), si ricava che dovrebbe essere:

$$nv_{vu} = 1+nv_{vu}$$

palesemente assurda.

5.5 Semafori condizione

Uno degli usi più generali dei semafori è quello legato alla specifica di condizioni di sicronizzazione. Come è stato indicato nel paragrafo 4.3, capita spesso che una (o più di una) delle operazioni per accedere a una risorsa condivisa `R` richieda che lo stato interno della risorsa soddisfi una particolare condizione logica `C` (condizione di sincronizzazione). Quando ciò si verifica, potremo specificare tale operazione come una `region`. Per esempio, indicando con op_1 tale operazione, la potremo specificare così:

`void` op_1`(): region R<<when(C)` S_1`;>>`

che indica appunto che l'operazione $op_1()$ è una regione critica, dovendo operare su una risorsa condivisa, ma anche che il corpo S_1 dell'operazione ha come precondizione la validità di `C` e che, quindi, il processo che la invoca si deve sospendere se, all'atto dell'invocazione, la risorsa è in uno stato interno in cui la condizione `C` non è vera. Come abbiamo visto nel paragrafo 4.3, l'operazione può essere quindi rappresentata come riportato nella figura 5.8a, dove usiamo ancora il simbolo grafico ovale per rappresentare il fatto che l'operazione deve essere eseguita in forma atomica. In particolare, come è indicato nella figura, anche la valutazione della condizione `C` deve avvenire in mutua esclusione, poiché la `C` è una condizione sullo stato inter-

Riquadro 5.3

VERIFICA DELLA CORRETTEZZA DELLO SCAMBIO DI DATI

Per effettuare queste verifiche è necessario ricorrere alle relazioni invarianti dei due semafori che, tenendo conto dei loro valori iniziali, sono:

$$np_{vu} \leq 1+nv_{vu} \quad (5.10)$$

$$np_{pn} \leq nv_{pn} \quad (5.11)$$

Inoltre, in base all'ordine con cui vengono eseguite le operazioni semaforiche nelle due funzioni `invio` e `ricezione`, possiamo asserire che in questa soluzione valgono le seguenti ulteriori relazioni:

$$np_{vu} \geq nv_{pn} \quad (5.12)$$

$$np_{pn} \geq nv_{vu} \quad (5.13)$$

la (5.12) perché nella `invio` si esegue prima `P(vu)` e poi `V(pn)` e la (5.13) perché nella `ricezione` si esegue prima `P(pn)` e poi `V(vu)`.

Iniziamo con la verifica delle precedenti proprietà 1 e 2. In pratica dobbiamo dimostrare che, per evitare underflow, ossia di estrarre da `buffer` un dato prima che questo sia stato inserito, il numero di inserimenti effettuati in un certo istante (che indicheremo con `nI`) sia sempre maggiore o uguale al numero di estrazioni (che indicheremo con `nE`). Analogamente, per evitare overflow del `buffer`, `nI` dovrà sempre essere minore, o al più uguale, a `nE+1`. Si tratta quindi di dimostrare che vale sempre la seguente relazione:

$$nE \leq nI \leq nE+1 \quad (5.14)$$

Poiché un inserimento viene eseguito nella funzione `invio` dopo la `P(vu)` iniziale e prima della `V(pn)` è ovvio che:

$$nv_{pn} \leq nI \leq np_{vu} \quad (5.15)$$

e, per le stesse considerazioni relative alla funzione `ricezione`:

$$nv_{vu} \leq nE \leq np_{pn} \quad (5.16)$$

Quindi, in base alla relazione (5.15), $nI \leq np_{vu}$ ma $np_{vu} \leq 1+nv_{vu}$ in base alla relazione (5.10) e $nv_{vu} \leq nE$ in base alla relazione (5.16). Mettendo insieme queste relazioni si ottiene $nI \leq np_{vu} \leq 1+nv_{vu} \leq 1+nE$ e, quindi, la parte destra della relazione (5.14) che volevamo dimostrare, cioè l'assenza di condizioni di overflow. Con analogo ragionamento, relativamente alle relazioni (5.16), (5.11) e (5.15) otteniamo che $nE \leq np_{pn} \leq nv_{pn} \leq nI$ cioè la parte sinistra della relazione (5.14) che dimostra l'assenza di condizioni di overflow.

Verifichiamo adesso la proprietà 3, per la quale la mutua esclusione fra `estrai` e `inserisci` è garantita dalla sola presenza dei due semafori `pn` e `vu`. Supponiamo infatti, per assurdo, che `estrai` e `inserisci` siano contemporaneamente in esecuzione. Se è in esecuzione `estrai`, significa che all'interno della funzione `invio` la `P` iniziale è già stata superata mentre la `V` finale non è stata ancora eseguita. Per cui la relazione (5.12), in questo caso, vale con il segno > in senso stretto (5.12'). Per lo stesso ragionamento, mentre è in esecuzione `inserisci`, anche la relazione (5.13) vale col segno > in senso stretto (5.13'). Quindi:

$$nv_{vu} < np_{pn} \leq nv_{pn} < np_{vu} \leq 1+nv_{vu}$$

relazione (5.13') relazione (5.11) relazione (5.12') relazione (5.10)

da cui risulta, tra nv_{vu} e nv_{pn}, la seguente relazione:

$$nv_{vu} < nv_{pn} < 1+nv_{vu}$$

ossia una grandezza intera (nv_{vu}) dovrebbe essere compresa tra due valori interi consecutivi (nv_{vu} e $1+nv_{vu}$), il che è palesemente assurdo.

Concludiamo con la prova della proprietà 4. Anche in questo caso, supporre, per assurdo, che i due processi siano in stallo conduce a una contraddizione. Infatti, se i due processi sono in stallo è perché ciascuno ha trovato rosso il semaforo su cui esegue la `P`. Quindi il valore dei due semafori è nullo e ciò significa che:

$$np_{vu} = 1+nv_{vu} \quad (5.10')$$

e

$$np_{pn} = nv_{pn} \quad (5.11')$$

D'altra parte, se i due processi sono bloccati, nessuno sta eseguendo le rispettive funzioni `invio` e `ricezione` e quindi il numero di volte che la `P` iniziale di ogni funzione è stata completata corrisponde al numero di volte in cui la `V` finale è stata eseguita:

$$np_{vu} = nv_{pn} \quad (5.12'')$$

$$np_{pn} = nv_{vu} \quad (5.13'')$$

Componendo queste relazioni, in particolare la (5.13'') con la (5.11'), questa con la (5.12'') e infine con la (5.10'), si ricava che dovrebbe essere:

$$nv_{vu} = 1+nv_{vu}$$

palesemente assurda.

5.5 Semafori condizione

Uno degli usi più generali dei semafori è quello legato alla specifica di condizioni di sincronizzazione. Come è stato indicato nel paragrafo 4.3, capita spesso che una (o più di una) delle operazioni per accedere a una risorsa condivisa `R` richieda che lo stato interno della risorsa soddisfi una particolare condizione logica `C` (condizione di sincronizzazione). Quando ciò si verifica, potremo specificare tale operazione come una `region`. Per esempio, indicando con op_1 tale operazione, la potremo specificare così:

```
void op1(): region R<<when(C) S1;>>
```

che indica appunto che l'operazione $op_1()$ è una regione critica, dovendo operare su una risorsa condivisa, ma anche che il corpo S_1 dell'operazione ha come precondizione la validità di `C` e che, quindi, il processo che la invoca si deve sospendere se, all'atto dell'invocazione, la risorsa è in uno stato interno in cui la condizione `C` non è vera. Come abbiamo visto nel paragrafo 4.3, l'operazione può essere quindi rappresentata come riportato nella figura 5.8a, dove usiamo ancora il simbolo grafico ovale per rappresentare il fatto che l'operazione deve essere eseguita in forma atomica. In particolare, come è indicato nella figura, anche la valutazione della condizione `C` deve avvenire in mutua esclusione, poiché la `C` è una condizione sullo stato inter-

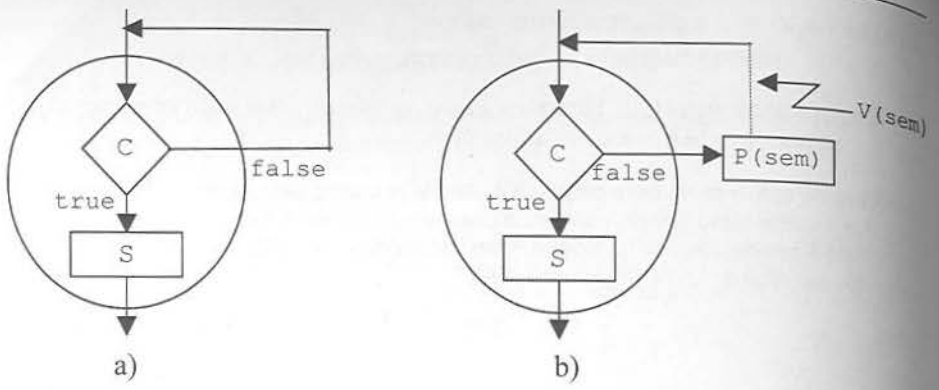


Figura 5.8 Regione critica condizionale.

no della risorsa e non può essere valutata se, contemporaneamente, qualcuno modifica tale stato. Però, se la C è falsa, il processo deve uscire dalla sezione critica per consentire ad altri processi di invocare altre operazioni su R in modo tale che, modificandone lo stato interno, possano rendere vera la C e, quindi, abilitare il processo che ha invocato la $op_1()$ a proseguire, ovviamente una volta rientrato in sezione critica. Lo schema descritto nella figura 5.8a è però uno schema concettuale, in quanto prevede che il processo che deve sospendersi resti in un ciclo di attesa attiva. In realtà, avendo a disposizione il meccanismo semaforico possiamo implementare la `region` sospendendo il processo su un semaforo da associare alla condizione. Come indicato nella figura 5.8b, supponiamo di aver riservato il semaforo `sem` inizializzato a zero associato alla condizione C in modo tale che il processo, che ha invocato $op_1()$ e ha trovato la condizione C falsa, esca dalla sezione critica e si blocchi su `sem`. Ovviamente dobbiamo supporre che sia disponibile almeno un'ulteriore operazione $op_2()$ che, invocata da un altro processo, modifichi lo stato interno di R in modo tale che C diventi vera. In questo caso, è nell'ambito dell'operazione $op_2()$ che viene eseguita la primitiva `V(sem)` per indicare che la condizione C è adesso vera. Ciò significa che, all'interno dell'operazione $op_2()$, la primitiva `V(sem)` ha il verificarsi della condizione C come sua precondizione logica.

È però necessario che chi esegue la primitiva `P(sem)` trovi il semaforo rosso per avere la garanzia di bloccarsi. Per questo motivo, alla condizione C associamo, oltre al semaforo `sem`, anche un contatore `csem` inizializzato a zero, che viene incrementato prima di eseguire la `P(sem)` e decrementato dopo che il processo viene risvegliato, in modo tale che il suo valore indichi sempre il numero di processi bloccati su `sem`. In questo modo, nell'operazione $op_2()$, la `V(sem)` viene eseguita soltanto se il contatore `csem` è maggiore di zero, cioè se ci sono processi sospesi sul semaforo, evitando quindi che `sem` assuma valori maggiori di zero. In questo modo, `sem` si comporta come un semaforo bloccante, secondo la terminologia introdotta alla fine del paragrafo 5.1. Nella figura 5.9 viene riportato lo uno schema generale di come può essere implementata la `region` corrispondente all'operazione $op_1()$. Nella figura viene anche riportata l'operazione $op_2()$, dove si suppone che il corpo S_2 della funzione modifichi lo stato interno di R, in modo tale da rendere vera la condizione C. Quindi, nello schema delle due funzioni descritto nella figura, si suppone che C sia una precondizione di S_1 e una postcondizione di S_2.

Questo paradigma di uso di un semaforo viene spesso identificato col termine di *semaforo condizione*, per evidenziare il fatto che questo viene utilizzato come semaforo bloccante su cui sospendere ogni processo che trova falsa la condizione concettualmente associata al semaforo stesso.

Come evidenziato nella figura, un processo che invoca op_1 entra in mutua esclusione e valuta la condizione C: se vera, esegue il corpo S_1 della funzione e termina rilasciando la mutua esclusione; se falsa, incrementa il contatore associato al semaforo `sem` e, liberata la mutua esclusione, si blocca su di esso. Quando il processo verrà riattivato da chi ha reso vera la condizione, rientrerà in mutua esclusione, tornerà in testa al ciclo e, trovando vera la C, potrà correttamente completare la funzione. Un processo che esegue op_2, completato il corpo S_2 della funzione e avendo quindi reso vera la condizione C, può svegliare chi eventualmente attende questo evento. Per questo, esso verifica se su `sem` c'è qualche processo bloccato e, in questo caso esegue, `V(sem)` per svegliarne uno, quindi termina l'esecuzione della funzione.

Questo schema di utilizzazione di un semaforo condizione per implementare una `region` viene anche indicato come *schema con attesa circolare*.

Infatti l'operazione che blocca un processo su un semaforo condizione si trova all'interno di un ciclo `while` per cui, quando il processo viene svegliato perché la condizione attesa è diventata vera, deve comunque tornare in testa al ciclo e ritestare la condizione stessa. La necessità di tornare a testare la condizione può sembrare superflua, ma in realtà non lo è perché, non essendo noti i rapporti di velocità fra i pro-

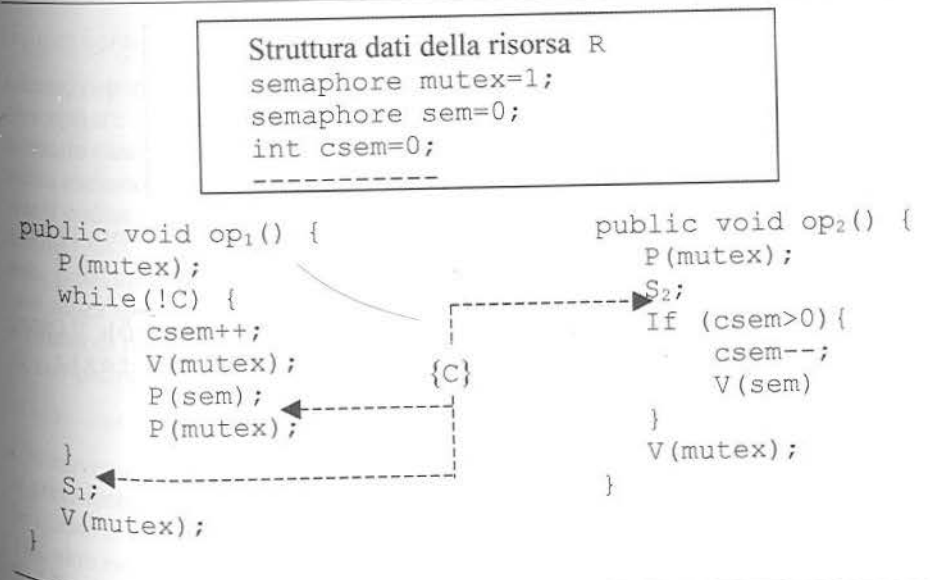


Figura 5.9 Semaforo condizione: schema con attesa circolare.

cessi, possono capitare condizioni di interferenza nel caso di particolari valori di tali rapporti. Per esempio, supponiamo il seguente scenario: il processo P si blocca durante l'esecuzione di op_1 avendo trovato falsa la condizione C. Successivamente il processo Q invoca op_2 e, durante la sua esecuzione, un terzo processo R invoca ancora op_1 bloccandosi su mutex, poiché sulla risorsa sta operando Q. Quando, alla fine di op_2, viene eseguita la V(sem), il processo P viene riattivato, avendo la garanzia che, subito dopo la P(sem) (vedi figura 5.9), la condizione C è vera. Però, dovendo rientrare in mutua esclusione, può darsi che il processo R, anche lui in attesa di entrare in mutua esclusione, entri prima di P e, quindi, esegua op_1 prima di P, completando la funzione poiché la C è vera. Ma, così facendo, R può modificare lo stato interno della risorsa invalidando di nuovo la condizione C. Quando poi il processo P avrà la possibilità di entrare a sua volta in mutua esclusione supponendo che la C sia vera, la troverà in realtà falsa. Questo scenario, per quanto spiacevole, non crea problemi di correttezza poiché il processo P, tornando a testare la condizione e trovandola ancora falsa, si riblocca. L'inconveniente si riduce quindi a un inutile cambio di contesto.

Per evitare comunque il precedente inconveniente, è possibile utilizzare anche un diverso schema con il quale il processo risvegliato può fare a meno di ritestare la condizione. Ciò si ottiene garantendo al processo P la possibilità di essere il primo a rientrare in mutua esclusione dopo che è stato risvegliato. Questo secondo schema, noto col nome di *schema con passaggio di testimone*, viene illustrato nella figura 5.10.

Nella funzione op_1, al posto del while, viene eseguito un if. Inoltre, quando il processo che si sospende trovando la C falsa viene risvegliato, non rientra in mutua esclusione ma continua, avendo la garanzia che potrà terminare la funzione senza pericolo che qualcuno possa "passargli avanti" e invalidargli la condizione per

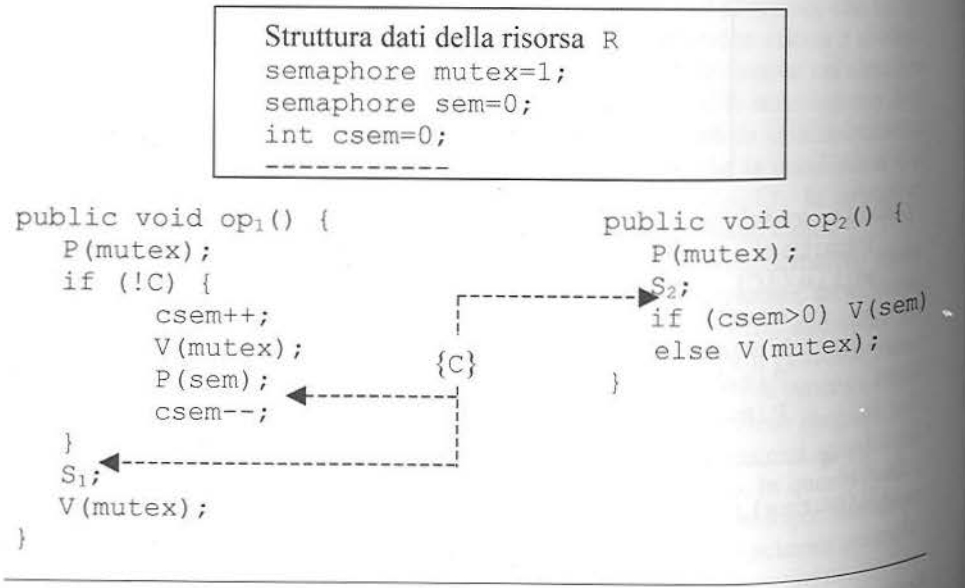


Figura 5.10 Semaforo condizione: schema con passaggio di testimone.

cui è stato svegliato. La garanzia che il processo svegliato possa riprendere l'esecuzione di op_1 in mutua esclusione, pur senza rientrare in sezione critica, è dovuta al fatto che, chi esegue op_2 e sveglia il processo bloccato, termina la funzione senza rilasciare la mutua esclusione (la V(mutex) viene eseguita in op_2 solo se non viene svegliato nessuno). In altri termini, il processo che termina l'esecuzione di op_2 passa al processo svegliato il diritto di operare in mutua esclusione (da cui il nome di schema con passaggio del testimone). In pratica, come si può notare, i semafori mutex e sem vengono adoperati come un semaforo binario composto.

Prima di terminare questo argomento mettiamo in evidenza due caratteristiche dei due schemi generali visti. Il secondo schema è sicuramente più efficiente del primo, però con esso non è possibile svegliare più di un processo alla volta poiché a uno solo è possibile passare il diritto di operare in mutua esclusione. Il primo schema non soffre di questa limitazione. Una seconda osservazione va fatta relativamente alla condizione di sincronizzazione. I due schemi presuppongono che la condizione C, che viene testata all'interno della funzione op_1, sia verificabile anche all'interno della funzione op_2, (dobbiamo essere certi che C sia una postcondizione di S_2). Ciò implica che l'espressione logica che caratterizza la condizione C non deve contenere variabili locali o parametri della funzione op_1 altrimenti non sarebbe verificabile all'interno di op_2.

Vediamo adesso un esempio di applicazione dei semafori condizione.

5.5.1 Gestione di un pool di risorse equivalenti (prima soluzione)

Un caso di competizione leggermente più complesso della semplice mutua esclusione si presenta quando più processi condividono non una sola risorsa ma un pool di risorse equivalenti. Per esempio supponiamo che, una volta definita la classe tipo_risorsa vista nel paragrafo 5.2, vengano dichiarate non una ma N diverse istanze:

```
tipo_risorsa ris[N];
```

Adesso, rispetto al caso di una singola risorsa condivisa, quando ciascun processo deve operare su una risorsa può utilizzare una qualunque delle N risorse disponibili. In questo caso però il problema non può essere risolto semplicemente garantendo la mutua esclusione degli accessi a ciascuna delle N risorse. Infatti, supponiamo di scrivere il codice di un qualunque processo che, a un certo punto, debba eseguire l'operazione $op_i()$ su una delle risorse del pool. Essendo tutte le risorse fra loro equivalenti, possiamo supporre che la risorsa su cui eseguire l'operazione venga scelta a caso senza una particolare strategia. Per esempio, supponiamo che quella scelta sia la risorsa di indice 3. Ciò significa che, nel testo del programma eseguito dal processo, comparirà l'istruzione:

```
ris[3].opi();
```

L'inconveniente che può derivare da questa scelta è che, quando il processo esegue la precedente operazione, ris[3] sia utilizzata da un altro dei processi applicativi. Se questo si verifica, il processo si sospende per mutua esclusione pur essendoci, magari, altre risorse del pool disponibili. Come abbiamo messo in evidenza nel paragrafo 4.2, tale inconveniente deriva dall'aver operato una scelta in fase di scrittura del programma (l'uso della risorsa di indice 3 nel nostro esempio) mentre la conoscenza

di quale risorsa sia disponibile e quale occupata, istante per istante, non può che essere nota in fase di esecuzione del programma. Per risolvere il problema in modo efficiente, conviene quindi operare con un diverso criterio, per esempio, allocando dinamicamente tutte le risorse del pool tramite un gestore che mantenga aggiornato lo stato delle risorse in modo tale da sapere, istante per istante, quale è disponibile e quale occupata. Ciascun processo, quando deve operare su una risorsa, chiede al gestore l'allocazione di una di esse. Il gestore, se ci sono risorse disponibili, ne sceglie una e la alloca in modo dedicato al processo richiedente restituendo allo stesso l'indice della risorsa allocata. Il processo può quindi operare sulla risorsa senza necessità di preoccuparsi della mutua esclusione. Quando la risorsa non serve più, il processo la rilascia al gestore che può quindi allocarla ad altri processi richiedenti. Ovviamente, se un processo richiede al gestore una risorsa quando non ci sono risorse disponibili, il processo si sospende in attesa che si liberi una qualunque risorsa.

Si tratta quindi di modificare la definizione della classe `tipo_risorsa` vista nel precedente paragrafo e di definire il gestore delle sue `N` istanze.

Per quanto riguarda la nuova classe `tipo_risorsa`, dovendo definire un tipo di risorse dedicate, non è più necessario il semaforo di mutua esclusione. Possiamo quindi riscrivere la classe vista nel precedente paragrafo semplicemente eliminando il semaforo `mutex` dalla sua struttura dati e le operazioni `P(mutex)` e `V(mutex)` da ciascuna delle sue funzioni.

Per quanto riguarda il gestore, come già indicato nel precedente capitolo, questo è costituito da una risorsa di tipo astratto che deve offrire ai processi due funzioni per richiedere e rilasciare, rispettivamente, il diritto di operare su una qualunque risorsa del pool. Consideriamo, per esempio, le due funzioni:

```
int richiesta();
void rilascio(int r);
```

la prima, utilizzata per richiedere una risorsa al gestore, restituisce l'indice della risorsa allocata e la seconda è invocata per rilasciare il diritto di operare sulla risorsa il cui indice è passato tramite il parametro `r`.

Indicando con `G` il gestore, possiamo allora specificare nel seguente modo la sincronizzazione negli accessi a `G` da parte dei processi richiedenti:

```
int richiesta(): region G<<
      when(<ci sono risorse disponibili>)
      <scelta di una risorsa disponibile>;
      int i=<indice della risorsa scelta>;
      <registra che la risorsa di indice i non è più disponibile>;
      return i;                                                    (5.17)
      >>

void rilascio(int r): region G<<
      <registra che la risorsa r-esima è di nuovo disponibile>     (5.18)
      >>
```

Le due funzioni costituiscono una classe di sezioni critiche in quanto operano sull'oggetto `G` di tipo astratto, condiviso tra tutti i processi richiedenti. Inoltre, mentre la funzione `rilascio` non richiede ulteriori specifiche di sincronizzazione, la specifica della funzione `richiesta`, tramite la clausola `when`, prevede che il processo richiedente venga sospeso se non ci sono risorse disponibili.

```
class tipo_gestore {
  semaphore  mutex=1; /*semaforo di mutua esclusione*/
  semaphore  sem=0; /*semaforo condizione*/
  int csem=0; /*contatore dei processi sospesi su sem */
  boolean  libera[N]; /*indicatori di risorsa libera*/
  int  disponibili=N; /*contatore delle risorse libere*/
  {for(int i=0;i<N;i++)libera[i]=true;} /*inizializzazione*/
  public int richiesta() {
    int i=0;
    P(mutex);
    if (disponibili==0) {
       csem ++;
       V(mutex);
       P(sem);
       cems--;
    }
    while(!libero[i]) i++;
    libero[i]=false;
    disponibili--;
    V(mutex);
    return i;
  }
  public void rilascio(int r) {
    P(mutex);
    libero[r]=true;
    disponibili++;
    if (csem>0) V(sem);
    else V(mutex);
  }
}
```

{disponibili>0}

Figura 5.11 Gestore di un pool di risorse equivalenti.

Associando quindi un semaforo condizione `sem` alla condizione di sincronizzazione <ci sono risorse disponibili> e utilizzando, per esempio, lo schema con passaggio del testimone, possiamo scrivere il codice della classe `tipo_gestore` di cui l'oggetto `G` rappresenta un'istanza (vedi figura 5.11).

In questo esempio la disponibilità di risorse viene indicata dal contatore `disponibili`, mentre il vettore `libera` denota lo stato (libera o occupata) di ciascuna delle `N` risorse. Indicando con la metavariabile n_{lib} il numero di componenti del vettore `libera` con valore `true`, l'espressione:

$$\texttt{disponibili} == n_{lib}$$

è parte della relazione invariante della classe ed è facile dimostrare come la soluzione riportata nella figura garantisca la validità di tale relazione. Tenendo conto che in questo esempio la condizione di sincronizzazione è rappresentata dall'espressione:

```
disponibili>0
```

e ricordando lo schema generale visto prima nella figura 5.10, si può facilmente verificare come la soluzione riportata sia una fedele implementazione della specifica delle due precedenti `region` (5.17) e (5.18).

Indicata, quindi, con `G` l'istanza della precedente classe utilizzata per allocare dinamicamente le `N` risorse:

```
tipo_gestore G;
```

ogni processo `P` che vuole operare su una risorsa del pool segue uno schema del tipo:

```
process P{
   int ris;
   ------
   ris = G.richiesta ();
   <uso della risorsa di indice ris>
   G.rilascio (ris);
   ------
}
```

5.6 Semafori risorsa

Come è stato detto all'inizio di questo capitolo, le varie tipologie di semafori che vengono presentate in questi paragrafi rappresentano, in realtà, paradigmi diversi di uso di un unico meccanismo, ciascuno dei quali è più adatto a risolvere una particolare classe di problemi. Spesso, però, uno stesso problema si presta a essere risolto anche usando paradigmi diversi. È questo, per esempio, il caso visto prima del gestore di un pool di risorse equivalenti. Infatti, ogni volta che il problema da risolvere prevede un'allocazione di risorse, è possibile fare riferimento a uno schema diverso che utilizza semafori generali, che possono cioè assumere qualunque valore maggiore o uguale a zero. In questi casi, possiamo riservare un unico semaforo `n_ris` inizializzato con un valore uguale al numero di risorse da allocare ed eseguire una `P(n_ris)` in fase di allocazione e una `V(n_ris)` in fase di rilascio. In questo modo, a ogni allocazione, il valore di `n_ris` viene decrementato e, quando non ci sono più risorse disponibili, il suo valore diventa nullo bloccando una ulteriore richiesta fino a quando non viene eseguito un rilascio. È per questo che un semaforo utilizzato in questo modo viene anche indicato come *semaforo risorsa*. Rivediamo quindi il precedente problema utilizzando questo nuovo schema di soluzione.

5.6.1 Gestione di un pool di risorse equivalenti (seconda soluzione)

Nella nuova versione della classe `tipo_gestore` riserviamo quindi il semaforo generale `n_ris` inizializzato a `N`. Inoltre la funzione `richiesta` inizierà mediante una `P(n_ris)` mentre la funzione `rilascio` terminerà mediante una `V(n_ris)`. È ancora necessario il vettore `libera` per tener conto dello stato di allocazione di ciascuna risorsa. In questo modo, se durante l'esecuzione di una `richiesta` il semaforo `n_ris` è verde (cioè se ci sono risorse libere), è necessario scandire il vettore `libera` per trovare una risorsa disponibile da allocare. Poiché tale vettore è condiviso, è anche necessario operarci in mutua esclusione mediante il semaforo binario `mutex` (vedi figura 5.12).

Questa soluzione non rispetta più la specifica delle due funzioni `richesta` e `rilascio` indicata tramite le `region` (5.17) e (5.18). Infatti, ciascuna delle due funzioni non è più costituita da un'unica operazione da eseguire completamente in mutua esclusione, cioè da un'unica `region` e come tale "non divisibile". Prendiamo per esempio la funzione `richiesta`. Come si può notare essa è costituita da due operazioni "non divisibili" in sequenza: la `P(n_ris)`, non divisibile in quanto operazione primitiva e, successivamente, la `region` controllata dal semaforo di mutua esclusione `mutex`. Quindi, un processo `P` che ha invocato la funzione `richiesta` può essere interrotto fra le due operazioni, cioè dopo aver superato la `P(n_ris)` ma prima di eseguire la sezione critica controllata da `mutex`. A quel punto, quindi, un secondo processo `Q` può, a sua volta, iniziare l'esecuzione di `ri-`

```
class tipo_gestore {
  semaphore  mutex=1; /*semaforo di mutua esclusione*/
  semaphore n_ris=N; /*semaforo risorsa*/
  boolean  libera[N]; /*indicatori di risorsa libera*/

  {for(int i=0;i<N;i++)libera[i]=true;} /*inizializzazione*/

  public int richiesta() {
    int i=0;
    P(n_ris);                          { n_lib > 0 }
    P(mutex);
    while(libero[i]==false) i++;
    libero[i]=false;
    V(mutex);
    return i; }

  public void rilascio (int r) {
    P(mutex);
    libero[r]=true;
    V(mutex);
    V(n_ris); }
}
```

Figura 5.12 Gestore di un pool di risorse equivalenti con semafori risorsa.

chiesta o di rilascio. Affinché la precedente soluzione sia corretta dobbiamo accertarci che Q, pur iniziando a operare sul gestore mentre vi sta ancora operando P, trovi la struttura del gestore in stato consistente. Per renderci conto di ciò è necessario dare un più preciso significato alle variabili coinvolte. Indichiamo ancora con la metavariabile n_{lib} il numero di risorse libere in un certo istante (cioè il numero di componenti del vettore libera con valore true) e con val_{n_ris} il valore del semaforo n_ris. È facile rendersi conto che i valori delle due metavariabili coincidono quando nessuno sta operando sul gestore. Durante l'esecuzione di richiesta, però, dopo la P(n_ris) iniziale e prima della sezione critica successiva, un altro processo può iniziare una nuova operazione sul gestore e, in questo caso, troverà:

$n_{lib}==val_{n\_ris}+1$

poiché il semaforo n_ris è stato già decrementato con la P(n_ris) iniziale mentre non è stata ancora allocata una delle risorse libere. Questo stato intermedio non è però uno stato inconsistente. Infatti, fermo restando il significato di n_{lib}, possiamo interpretare la P(n_ris) come la prenotazione di una risorsa che, successivamente, verrà realmente allocata. Quindi, poiché una risorsa viene prima prenotata e poi allocata, la relazione di consistenza tra le due metavariabili val_{n_ris} e n_{lib} diventa in questo caso:

$val_{n\_ris} \leq n_{lib}$

che è valida anche nel precedente stato intermedio.

Se un processo viene interrotto durante l'esecuzione di richiesta dopo la P(n_ris) e prima della sezione critica con cui termina la funzione, lascia il gestore nello stato consistente in cui le risorse prenotabili sono inferiori a quelle disponibili ($val_{n\_ris} < n_{lib}$). Ogni processo che da questo momento operi sul gestore lascerà ancora valida questa relazione. Quindi, anche se vengono eseguite ulteriori richieste queste verranno accettate fino a quando il semaforo n_ris verrà azzerato, il che significa che il processo interrotto avrà la garanzia di riprendere l'esecuzione trovando che almeno una risorsa è ancora disponibile per lui. È cioè garantita la precondizione della sezione critica in cui una risorsa viene allocata che, come indicato nella figura 5.12, è $\{n_{lib}>0\}$ che deve essere vera affinché questa funzioni correttamente.

Per completare l'esempio vediamo come si semplifica il codice del gestore nell'ipotesi in cui questo gestisca una sola risorsa (caso in cui n==1). Il vettore libera diventa una sola variabile di tipo boolean. Inoltre, la funzione richiesta non deve restituire nessun indice in quanto la risorsa allocata è sempre l'unica esistente. Per lo stesso motivo, anche il parametro della funzione rilascio non serve a niente. Per cui il codice si riduce al seguente:

```
class tipo_gestore {
   boolean libera=true;
   semaphore mutex=1;
   semaphore n_ris=1;

public void richiesta() {
     P(n_ris);
     P(mutex);
     libera=false;
     V(mutex);
}

void rilascio () {
     P(mutex);
     libera=true;
     V(mutex);
     V(n_ris);
}
}
```

Si nota subito che la variabile libera non è necessaria. Infatti è una variabile che viene esclusivamente modificata ma il cui valore non viene mai letto. Eliminando quindi tale inutile variabile si può eliminare anche il semaforo mutex, che serve solo per operare in mutua esclusione sulla variabile eliminata. Il codice si riduce quindi ulteriormente, facendo coincidere il gestore col semaforo n_ris inizializzato a uno e in cui la richiesta coincide con la P su questo semaforo e la rilascio con la relativa V.

Abbiamo quindi ritrovato quanto già indicato nel paragrafo 4.2: non c'è nessuna differenza tra gestire una risorsa come condivisa tra un gruppo di processi (operandovi quindi in mutua esclusione) e allocarla dinamicamente agli stessi tramite un gestore. O meglio, non c'è differenza quando non interessa il criterio con cui la risorsa viene allocata ai processi richiedenti. Come vedremo nel seguito, ciò non è più vero se la risorsa deve essere allocata seguendo specifici criteri di allocazione. In questi casi, non sarà possibile allocarla tramite un semplice semaforo di mutua esclusione, ma sarà necessario scrivere opportunamente il codice del gestore al fine di rispettare i criteri di allocazione.

5.6.2 Problema dei produttori/consumatori

Riprendiamo in considerazione la soluzione al problema dello scambio di dati fra processi, visto nel paragrafo 5.4. Tale soluzione prevede una stretta sincronizzazione fra il processo che invia i dati (il *produttore* dei dati) e chi li deve ricevere (il *consumatore*). Infatti, come abbiamo visto, se uno dei due processi è più veloce dell'altro, per esempio il produttore, questo deve adeguarsi alla velocità dell'altro poiché, dopo un invio e prima di poter eseguire quello successivo, deve attendere che il consumatore abbia ricevuto il primo dato. Questa stretta sincronizzazione fra i due processi è conseguenza del fatto che il supporto alla comunicazione viene offerto dalla variabile buffer che può contenere al più un dato alla volta. Se vogliamo disaccoppiare i due processi in modo tale che il produttore possa inviare più dati prima di sincronizzarsi col consumatore è necessario aumentare la capacità del buffer. Per esempio, se n è la capacità del buffer in termini di messaggi, il vincolo descritto nella figura 5.7 si modifica come indicato nella figura 5.13.

Il produttore può cioè inviare fino a n dati successivi prima di sospendersi, se nel frattempo il consumatore non ha ricevuto nessun dato. Al limite, se la capacità del buffer fosse infinita il produttore non si bloccherebbe mai. Inoltre, per garantire che il consumatore riceva i dati nello stesso ordine con cui il produttore li ha inviati,

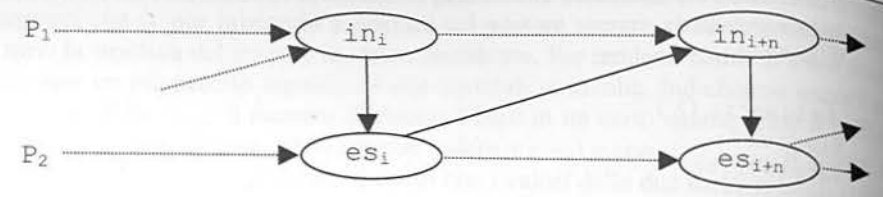


Figura 5.13 Vincoli di precedenza tra invio e ricezione di dati.

il `buffer` deve anche assicurare che l'ordine con cui i dati vengono prelevati sia lo stesso di quello con cui sono stati inseriti. Per questo motivo, il `buffer` non può essere strutturato semplicemente come un vettore di elementi dei tipo `T` (il tipo dei dati scambiati) ma deve essere strutturato come una coda di elementi di tipo `T`. Possiamo quindi generalizzare la soluzione al problema visto nel paragrafo 5.4, riservando in memoria una variabile `buffer` di tipo `coda_di_n_T`. Per semplicità, non riportiamo la definizione di questo tipo che è del tutto analoga a quella già vista nel primo capitolo (vedi figura 1.11): è sufficiente sostituire alla costante 16 il valore `n` e al tipo `boolean` il tipo generico `T`. Inoltre, associamo al `buffer` i due semafori `pn` e `vu` già visti nel paragrafo 5.4, con l'unica variante che, per garantire il vincolo indicato nella figura 5.13, il valore iniziale di `vu` adesso è `n` e non uno. Con queste indicazioni, nella figura 5.14 viene riportata la soluzione generale al problema della comunicazione tra processi.

Per verificare la correttezza della soluzione potremmo ripetere le stesse prove illustrare nel riquadro 5.3, in particolare, con gli stessi criteri già visti potremmo ancora verificare che non possono nascere condizioni di stallo e che non si verificano mai condizioni di overflow o di underflow del `buffer`. L'unica variante rispetto alle verifiche effettuate nel riquadro 5.3, è che adesso nelle prove compare la costante `n` al posto della costante 1. È proprio questa variante che, però, invalida l'altra prova vista nello stesso riquadro, utilizzata per verificare che in quel caso non era necessario riservare un semaforo di mutua esclusione per l'accesso al `buffer` in quanto le due operazioni `inserisci` ed `estrai` non potevano essere mai essere contemporaneamente in esecuzione. Ciò era legato al fatto che i due semafori `pn` e `vu` si comportavano come un semaforo binario composto, cosa che adesso non è ovviamente più vera essendo `n>1` il valore iniziale di `vu`. Per questo motivo è necessario, in generale, introdurre anche un semaforo `mutex` di mutua esclusione per garantire accessi non contemporanei al `buffer` come indicato nella figura 5.14.

In questa soluzione, valida anche nel caso in cui siano presenti più produttori e più consumatori, la struttura dati relativa al `buffer` e ai semafori associati è nota anche col termine di *mailbox* (cassetta postale) per indicare appunto un meccanismo generale per lo scambio di dati fra processi.

Abbiamo riportato questa soluzione in questo paragrafo perché, osservando la struttura delle due funzioni `invio` e `ricezione`, ci possiamo facilmente accorgere che ciascuna di esse ha la stessa struttura della richiesta di una risorsa e l'altra quella di un rilascio e che utilizzano semafori risorse come indicato nella figura 5.12. Infatti `invio` richiede una risorsa di tipo "posto vuoto nel buffer" e il semaforo `vu` rappresenta proprio un semaforo risorsa associato a questo tipo di risorse (di cui all'inizio esistono `n` istanze) mentre `ricezione` corrisponde al rilascio di un posto vuoto. Analogamente, la `ricezione` richiede una risorsa di tipo "posto occupato da un dato nel buffer" e il semaforo `pn` rappresenta proprio un semaforo risorsa associato a questo tipo di risorse (di cui all'inizio esistono zero istanze) mentre l'`invio` corrisponde al rilascio di un posto occupato.

```
                coda_di_n_T  buffer;
                semaphore    pn=0;
                semaphore    vu=n;
                semaphore    mutex=1;

void invio(T dato) {                    T ricezione() {
  P(vu);                                    T dato;
  P(mutex);                                 P(pn);
  buffer.inserisci(dato);                   P(mutex);
  V(mutex);                                 dato=buffer.estrai();
  V(pn);                                    V(mutex);
}                                           V(vu);
                                            return dato;
```

Figura 5.14 Soluzione al problema del produttore/consumatore.

Terminiamo questo esempio con una osservazione valida nel caso in cui la `coda_di_n_T` sia realizzata mediante un vettore circolare di `n` elementi di tipo `T` (in maniera del tutto analoga a quanto visto nella figura 1.11). Rispetto alla soluzione vista nella figura 1.11 non è adesso necessaria la variabile `cont`, che serviva nelle funzioni `inserisci` ed `estrai` semplicemente per verificare le condizioni di overflow e di underflow che, con i due semafori `vu` e `pn` associati alla coda, abbiamo già dimostrato non essere più possibili. Per cui le due funzioni `inserisci` ed `estrai` si riducono a quelle riportate nella figura 5.15. È facile rendersi conto in questo caso che, se siamo in presenza di un solo produttore e di un solo consumatore, il semaforo `mutex` è di nuovo del tutto superfluo. Infatti, senza `mutex` le due operazioni `inserisci` ed `estrai` possono essere eseguite concorrentemente, ma in questo caso non creano inconsistenze. Infatti, se eseguite insieme abbiamo la certezza che l'una inserisce un nuovo dato in un elemento del `buffer` mentre l'altra estrae un dato da un diverso elemento del `buffer`. L'interferenza tra le esecuzioni contemporanee delle due funzioni si avrebbe solo nel caso in cui una inserisse e l'altra estraesse un dato dallo stesso elemento del `buffer`. Ma ciò è possibile solo quando i due interi `primo` e `ultimo` coincidono e questo si verifica soltanto nei due casi di coda piena o di coda vuota. Se però la coda è piena, il semaforo `vu` precludе l'esecuzione di `inserisci` mentre, se la coda è vuota, è il semaforo `pn` che precludе l'esecuzione di `estrai`.

Chiaramente, una diversa implementazione della `coda_di_n_T` implicherebbe di nuovo la necessità del semaforo `mutex`. Per esempio, è lasciata al lettore la verifica di questa necessità nel caso di coda realizzata mediante una lista, tecnica spesso utilizzata per implementare una coda di lunghezza indefinita.

```
class coda_di_n_T {
   T  vettore[n];
    int primo=0; ultimo=0

public void inserisci(T dato) {
 vettore[ultimo]=dato;
 ultimo=(ultimo+1)%n;
}

public T estrai( ) {
  T dato=vettore[primo];
  primo=(primo+1)%n;
   return dato;
}
```

Figura 5.15 Buffer circolare.

5.7 Semafori privati: specifica di strategie di allocazione

In problemi di sincronizzazione complessi, in particolare quando si voglia realizzare una determinata politica di gestione delle risorse, la decisione di consentire a un processo di proseguire l'esecuzione durante l'accesso a una risorsa condivisa dipende, come si è visto, dal verificarsi di una particolare condizione, detta *condizione di sincronizzazione.* Può quindi accadere che più processi siano simultaneamente bloccati durante l'accesso a una risorsa condivisa, ciascuno in attesa che la propria condizione di sincronizzazione sia verificata. In seguito alla modifica dello stato della risorsa da parte di un processo, le condizioni di sincronizzazione di alcuni dei processi bloccati possono essere contemporaneamente verificate. Normalmente in questi casi, in particolare se l'accesso alla risorsa deve essere mutuamente esclusivo, nasce il problema di quale processo riattivare. Nel caso in cui si desideri realizzare una particolare politica di gestione della risorsa, la scelta del processo da riattivare deve avvenire sulla base di algoritmi, specificati risorsa per risorsa, che definiscono una priorità tra i processi.

In tutti gli esempi precedenti, ogni processo per il quale la condizione di sincronizzazione non è verificata viene bloccato facendogli eseguire un'operazione `P` su un particolare semaforo riservato a questo scopo. Se, quando viene eseguita la `V` su quel semaforo, sono presenti più processi sospesi, la scelta di quale processo sospeso riattivare non è quindi esplicitamente programmata, ma effettuata implicitamente tramite l'algoritmo realizzato dalla `V`.

Naturalmente questo algoritmo, dovendo essere sufficientemente generale, in modo da risultare adatto per la maggior parte dei casi, e il più possibile semplice, per rendere efficiente l'esecuzione delle primitive, coincide spesso con l'algoritmo FIFO.

Verranno riportati nel seguito alcuni esempi di definizione di politiche di gestione delle risorse per i quali, viceversa, risulta necessario poter programmare esplicitamente *algoritmi di scelta dei processi da riattivare.*

Esempio 5.1 Su un buffer costituito da n celle di memoria più produttori possono depositare messaggi di dimensione diversa. Il deposito può avvenire ovviamente solo se il numero delle celle libere è superiore o uguale alla dimensione del messaggio; diversamente il processo viene sospeso. Il prelievo avviene estraendo il primo messaggio contenuto nel buffer e liberando le celle di memoria da esso occupate.

Si vuole realizzare la seguente politica di gestione del buffer: tra più produttori ha priorità di accesso al buffer quello che fornisce il messaggio di maggior dimensione. Questa politica comporta che, per tutto il tempo in cui un produttore, il cui messaggio ha dimensione maggiore dello spazio disponibile nel buffer, rimane sospeso, nessun altro produttore può depositare il messaggio anche se le dimensioni di questo possono essere contenute nello spazio libero del buffer. In altre parole, il deposito di un messaggio può avvenire solo se è soddisfatta la seguente condizione di sincronizzazione: c'è sufficiente spazio per memorizzare il messaggio e non ci sono produttori in attesa.

Il prelievo di un messaggio da parte di un consumatore deve prevedere la riattivazione, tra tutti i processi sospesi, di quello il cui messaggio ha la dimensione maggiore, purché esista sufficiente spazio nel buffer. Se lo spazio disponibile non è sufficiente, nessun altro produttore viene attivato.

Esempio 5.2 Un insieme di processi P_1, P_2, ..., P_n utilizza un insieme di risorse comuni R_1, R_2, ..., R_m. Ogni processo può utilizzare una qualunque delle risorse. La condizione di sincronizzazione si riduce quindi a valutare se tra tutte le risorse ne esiste una libera: a ciascun processo è assegnata una priorità per cui, in fase di rilascio di una risorsa, tra tutti i processi in attesa di una risorsa deve essere scelto quello cui corrisponde la massima priorità.

Esempio 5.3 Con riferimento al problema dei lettori/scrittori visto nel paragrafo 5.2.1, realizzare una soluzione che eviti problemi di attesa indefinita (*starvation*) per le due classi di processi.

Una soluzione organica al problema di realizzare una politica di gestione delle risorse si può ottenere introducendo il concetto di *semaforo privato*. Un semaforo si dice privato per un processo quando solo tale processo può eseguire su di esso la primitiva `P`. La primitiva `V` può invece essere eseguita sul semaforo da altri processi. Un semaforo privato viene inizializzato al valore zero.

Vediamo adesso due diversi schemi di utilizzazione dei semafori privati per risolvere problemi di allocazione di risorse permettendo di specificare strategie di allocazione.

Primo schema Sia P_i un processo la cui acquisizione di una risorsa dipende dal verificarsi di una condizione di sincronizzazione. Indicando con `i` il PID del processo e con `priv[i]` un suo semaforo privato, di seguito viene illustrato lo schema di massima del tipo di un gestore che, mediante le operazioni `acquisizione` e `rilascio`, consente di allocare la risorsa secondo una specifica strategia:

```
class tipo_gestore_risorsa{
  <struttura dati del gestore>;
  semaphore mutex=1;
  semaphore priv[n]={0,0,..0};

  public void acquisizione(int i) {
    P(mutex);
```

```
        if (<condizione di sincronizzazione>) {
            <allocazione della risorsa>;
            V(priv[i]);
        }
        else <registrare la sospensione del processo>;
        V(mutex);
        P(priv[i]);
    }

    public void rilascio() {
        int i;
        P(mutex);
        <rilascio della risorsa>;
        if (<esiste almeno un processo sospeso per il quale la condizione di
            sincronizzazione è soddisfatta>) {
                <scelta fra i processi sospesi del processo P_i da riattivare>;
                <allocazione della risorsa a P_i>;
                <registrare che P_i non è più sospeso>;
                V(priv[i]);
        }
        V(mutex);
}
```

Per quanto riguarda l'acquisizione, se dall'analisi delle variabili comuni (stato della risorsa) il processo verifica che la condizione di sincronizzazione è soddisfatta, esso esegue la `V` sul suo semaforo `priv[i]` che diventa 1; diversamente, la primitiva non viene eseguita. In uscita dalla sezione critica l'esecuzione della `P` sul semaforo privato `priv[i]` decide se il processo può proseguire utilizzando la risorsa o se deve sospendersi in attesa che la condizione sia verificata.

Si noti che la verifica della condizione di sincronizzazione deve essere eseguita in modo mutuamente esclusivo da parte di ciascun processo. Tale verifica comporta infatti l'ispezione di variabili comuni a più processi.

Se, in seguito alla modifica delle variabili comuni conseguente al rilascio della risorsa, la condizione di sincronizzazione per la quale alcuni processi stanno attendendo è resa vera, viene eseguita la `V` sul semaforo privato di uno di tali processi, per esempio P_i, scelto sulla base di un determinato algoritmo. Il processo P_i può ora utilizzare la risorsa.

Lo schema presentato è, come si è detto, solamente indicativo delle modalità di impiego dei semafori privati in problemi di acquisizione delle risorse. In seguito verranno presentate e discusse soluzioni per specifici problemi. È possibile tuttavia fin da ora mettere in evidenza due proprietà dello schema presentato:

a) la sospensione del processo, nel caso in cui la condizione di sincronizzazione non sia verificata, non può avvenire all'interno della sezione critica in quanto ciò impedirebbe a un processo che rilascia la risorsa di accedere a sua volta alla sezione critica e di riattivare il processo sospeso. Nella soluzione proposta il processo abbandona la sezione critica prima di sospendersi; la sospensione avviene infatti al di fuori della sezione critica, tramite l'esecuzione della `P` sul semaforo privato del processo.

b) la specifica di un particolare algoritmo di assegnazione della risorsa ai processi, come già si è accennato, non è opportuno che sia realizzata nella `V`: nella soluzione proposta è possibile programmare esplicitamente tale algoritmo, scegliendo in base a esso il processo da attivare ed eseguendo la `V` sul suo semaforo privato.

Si possono inoltre mettere in evidenza le seguenti caratteristiche dello schema:

1. l'operazione `acquisizione` è strutturata come un'istruzione `if-then-else` (allocazione della risorsa e `V` sul semaforo privato nel ramo `then`, indicazione di sospensione del processo nel ramo `else`);
2. l'ultima operazione dell'operazione `acquisizione` è la `P` sul semaforo privato;
3. se in fase di rilascio viene deciso di svegliare un processo sospeso, l'assegnazione della risorsa al processo svegliato deve avvenire all'interno dell'operazione `rilascio`, in quanto, a causa del precedente punto 2, il processo svegliato ha già completato l'operazione `acquisizione`.

Questo schema può presentare a volte degli inconvenienti:

- la `P` sul semaforo privato viene sempre eseguita anche quando il processo richiedente non deve essere bloccato (punto 2);
- il codice relativo all'assegnazione della risorsa viene duplicato nelle operazioni `acquisizione` e `rilascio` (punto 3). Ciò non sempre è possibile, in particolare se l'allocazione prevede che debba essere restituito un valore al processo richiedente tramite un parametro (nella funzione `rilascio` non è infatti possibile vedere i parametri della funzione `acquisizione`).

Secondo schema Un secondo schema può essere ottenuto ricorrendo al concetto di semaforo condizione, solo che, a differenza di quanto visto nel paragrafo 5.5, questo schema prevede di associare, a ogni condizione, un semaforo privato per ogni processo invece di un solo semaforo. Lo schema seguente non soffre dei precedenti inconvenienti:

```
class tipo_gestore_risorsa{
    <struttura dati del gestore>;
    semaphore mutex=1;
    semaphore priv[n]={0,0,..,0};

    public void acquisizione(int i) {
        P(mutex);
        if (! <condizione di sincronizzazione>) {
            <registrare la sospensione del processo>;
            V(mutex);
            P(priv[i]);
            <registrare che il processo non è più sospeso>;
        }
        <allocazione della risorsa>;
        V(mutex);
    }

    public void Rilascio() {
        int i;
```

```
    P(mutex);
    <rilascio della risorsa>;
    if (<esiste almeno un processo sospeso per il quale la condizione di
       sincronizzazione è soddisfatta>) {
          <scelta fra i processi sospesi del processo P_i da riattivare>;
          V(priv[i]);
    }
    else V(mutex);
}
```

Questo schema, pur non soffrendo degli inconvenienti dello schema precedente, è comunque più complesso da usare nei casi in cui, all'atto del rilascio, sia possibile svegliare più di un processo sospeso. Infatti, adottando il criterio del passaggio del testimone visto nel paragrafo 5.5, non è possibile risvegliare più di un processo alla volta. Infatti a uno solo può essere trasferito il diritto di operare in mutua esclusione. In questi casi è necessario complicare la funzione `acquisizione`, in modo tale che sia il solo processo risvegliato, terminando l'`acquisizione`, a verificare se è possibile svegliare un secondo processo e, in caso positivo, a risvegliarlo; questo a sua volta provvederà alla riattivazione di un altro processo e così via.

In conclusione, la scelta fra i due schemi presentati dipende dal particolare tipo di problema di gestione di risorse da risolvere.

5.7.1 Uso dei semafori privati

Viene riportata di seguito la soluzione, mediante l'uso dei semafori privati, ai problemi presentati precedentemente.

Esempio 5.4 Un processo produttore chiama la procedura `acquisizione` per ottenere le celle del buffer sufficienti a consentirgli il deposito del messaggio. Un consumatore, prelevato il messaggio, esegue la procedura `rilascio` per restituire al buffer le celle nelle quali il messaggio prelevato era contenuto.

La soluzione presentata segue il primo dei due schemi illustrati precedentemente.

```
class buffer {
   int richiesta[num_proc]=0; /*richiesta[i]= numero di celle richieste da P_i*/
   int sospesi=0; /*numero dei processi produttori sospesi*/
   int vuote=n; /*numero di celle vuote del buffer*/
   semaphore mutex=1;
   semaphore priv[num_proc]={0,0,..0};

   public void acquisizione(int m, int i) {
          /*m dimensione del messaggio, i nome del processo chiamante */
       P(mutex);
       if (sospesi==0&&vuote>=m) {
          vuote=vuote-m;
          <assegnazione al processo di m celle del buffer>;
          V(priv[i]);
       }
       else{
          sospesi++;
          richiesta[i]=m;
          V(mutex);
          P(priv[i]);
       }
   }

   public void rilascio(int m) {/* m numero di celle rilasciate*/
       int k;
       P(mutex);
       vuote+=m;
       while(sospesi!=0) {
          <individuazione tra i processi sospesi del processo P_k con la massima richiesta>;
          if (richiesta[k]<=vuote) {
             vuote=vuote-richiesta[k];
             <assegnazione a P_k delle celle richieste>;
             richiesta[k]=0;
             sospesi--;
             V(priv[k]);
          }
          else break; /* fine while */
       }
       V(mutex);
   }
```

Questo esempio mostra come con il primo schema sia possibile risvegliare più processi. Infatti, all'atto di un rilascio può essere liberato un numero di celle sufficienti a risvegliare più processi sospesi. Per questo motivo, nella funzione `rilascio` viene utilizzato un ciclo `while` per riattivare tutti i processi che, in base alla loro priorità, possono completare le proprie richieste.

Esempio 5.5 Una soluzione al problema della gestione di un pool di risorse equivalenti condivise da un insieme di processi è già stata presentata nel paragrafo 5.5.1. Tuttavia, in quel caso, non era prevista una politica di risveglio dei processi bloccati basata sulla loro priorità, come invece richiesto dal problema riportato nell'esempio 5.2. Per realizzare tale politica è necessario che, in fase di sospensione, ogni processo lasci traccia della propria identità, in modo che, in fase di risveglio, sia possibile verificare tra tutti i processi sospesi quello a priorità massima.

Si noti che la politica di risveglio basata sulla priorità dei processi viene esplicitamente programmata, indipendentemente dalla politica con cui viene implementata nel nucleo del sistema operativo la primitiva `V`. È evidente infatti che è possibile ottenere lo stesso risultato realizzando a livello di nucleo, per la `V`, un criterio di scelta del processo da risvegliare basato sulla priorità. Spesso però, il criterio per le primitive del nucleo corrisponde alla politica FIFO e, comunque, è sempre buona norma garantire la corretta soluzione, a prescindere dai criteri adottati nel nucleo del sistema operativo, sia perché possono cambiare da sistema a sistema, sia perché non sono necessariamente noti a chi deve implementare il programma applicativo.

La soluzione presentata segue il secondo degli schemi illustrati precedentemente. Infatti, come indicato nel paragrafo 5.5.1, la funzione `richiesta` deve restituire al processo richiedente un parametro, cioè l'indice della risorsa allocata. Come abbiamo visto precedentemente, il primo schema non è adatto in questi casi.

```
class tipo_gestore {
   semaphore mutex=1;
   semaphore priv[num_proc]={0,0,...0} /*semafori privati dei processi*/
   int sospesi=0; /*tiene traccia del numero dei processi sospesi*/
   boolean PS[num_proc]={false,false,....,false}; /*PS[i] assume
         il valore true se il processo Pi è sospeso; false diversamente*/
   int disponibili=num_ris; /*tiene traccia del numero di risorse disponibili*/
   boolean libera[num_ris]={true,true,....,true}; /*libera[j] assume
         il valore true se la risorsa j è libera; false diversamente*/
   public int richiesta(int proc) {
         /* proc indica il nome del processo chiamante*/
      int i=0;
      P(mutex)
      if (disponibili==0) {
         sospesi++;
         PS[proc]=true;
         V(mutex);
         P(priv[proc];
         PS[proc]=false;
         sospesi--;
      }
      while(!libera[i])i++;
      libera[i]=false;
      disponibili--;
      V(mutex);
      return i;
   }
   public void rilascio(int r) {
         /* r indica l'indice della risorsa rilasciata*/
      P(mutex);
      libera[r]=true;
      disponibili++;
      if (sospesi>0) {
         <seleziona il processo Pj a massima priorità tra quelli sospesi
         utilizzando il vettore PS>;
         V(priv[j]);
      }
      else V(mutex);
   }
```

L'esempio ricalca lo stesso schema già visto nel paragrafo 5.5.1 con l'unica variante che, invece di usare un solo semaforo condizione, se ne utilizzano tanti quanti sono i processi richiedenti (semafori privati) al fine di poter implementare l'algoritmo di scelta di quello a più alta priorità nella funzione `rilascio`.

Esempio 5.6 Riprendiamo l'esempio dei lettori/scrittori visto nel paragrafo 5.2.1 la cui soluzione è stata illustrata nella figura 5.3. Come è stato detto, tale soluzione introduce l'eventualità dell'attesa indefinita dei processi scrittori (starvation) a causa di una continua attività dei processi lettori. Per eliminare questa eventualità potremmo introdurre una strategia di allocazione del file comune in base alla quale, in presenza di letture attive, una nuova operazione di lettura non viene accettata dal momento in cui un processo scrittore si blocca. In questo modo abbiamo la garanzia che, alla fine delle letture in corso, un processo scrittore può procedere eliminando il pericolo della starvation. Però questa strategia rischia di creare il problema inverso, cioè di ritardare indefinitamente i processi lettori se le scritture vengono continuamente richieste. Allora introduciamo un ulteriore requisito in base al quale, alla fine di una scrittura, vengono privilegiati eventuali lettori in attesa rispetto agli scrittori. Ricapitolando, vogliamo quindi scrivere il codice di un gestore che alloca il file in base ai seguenti requisiti:

a) devono essere consentite operazioni di lettura contemporanee;
b) le operazioni di scrittura devono essere mutuamente esclusive sia con altre scritture che con operazioni di lettura;
c) una nuova lettura non deve essere accettata se un processo scrittore è in attesa e, alla fine delle letture in corso, deve essere privilegiato un processo scrittore rispetto ai processi lettori;
d) alla fine di una scrittura devono essere privilegiati i processi lettori in attesa (attivandoli tutti se ce ne sono) rispetto a processi scrittori.

Si tratta quindi di scrivere il codice di un gestore che offre quattro funzioni: `richiesta_lettura`, `fine_lettura`, `richiesta_scrittura`, `fine_scrittura`.

In questo esempio il concetto di semaforo privato viene esteso rispetto ai casi precedenti. Infatti, dovendo di volta in volta implementare una strategia di priorità fra gruppi di processi e non fra processi singoli, introduciamo due soli semafori privati: uno riservato ai processi lettori `priv_lett` e uno riservato ai processi scrittori `priv_scrit`. Associati ai due semafori vi sono poi i corrispondenti interi che tengono conto del numero di processi sospesi sui rispettivi semafori `sospesi_lett` e `sospesi_scrit`. Infine sono necessarie le informazioni per tener conto dello stato di allocazione del file: l'intero `lettori_attivi` (nel caso di file allocato ai lettori), che indica il numero di processi lettori che stanno eseguendo l'operazione di lettura, e l'indicatore `boolean scrittore_attivo` che, essendo le scritture mutuamente esclusive, è sufficiente per indicare se il file è allocato a uno scrittore oppure no.

In questo esempio, non dovendo restituire nessuna informazione ai processi richiedenti e potendo inoltre risvegliare più processi a un rilascio (per esempio alla fine di una scrittura dove vanno risvegliati tutti gli eventuali lettori), è sicuramente più indicato il primo dei due schemi visti precedentemente. Come è stato detto, è però possibile usare anche il secondo schema adottando un criterio di risveglio di un processo alla volta. Per cui, lasciando al lettore la soluzione relativa al primo schema, riportiamo di seguito quella realizzata in base al secondo schema, proprio con l'ottica di mostrare come sia possibile risvegliare più processi, pur se uno alla volta, anche col secondo schema.

```
class tipo_gestore_file {
   semaphore mutex=1;
   semaphore  priv_lett=0;
   semaphore  priv_scritt=0;
   int sospesi_lett=0;
```

```
int sospesi_scrit=0;
int lettori_attivi=0;
boolean scrittore_attivo=false;

public void richiesta_scrittura() {
   P(mutex);
   if(lettori_attivi>0 || scrittore_attivo) {
      /*condizione di sincronizzazione non vera per uno scrittore*/
      sospesi_scrit++;
      V(mutex);
      P(priv_scritt); /*sospensione del processo scrittore*/
      sospesi_scrit--;
   }
   scrittore_attivo=true; /*la scrittura viene abilitata*/
   V(mutex);
}

public void fine_scrittura() {
   P(mutex);
   scrittore_attivo=false;
   if(sospesi_lett>0) V(priv_lett); /*se ci sono lettori bloccati
         si privilegiano rispetto agli scrittori e se ne sveglia uno*/
   else if(sospesi_scrit>0) V(priv_scrit); /*altrimenti, se
         ci sono scrittori bloccati se ne sveglia uno*/
         else V(mutex); /*altrimenti si rilascia la mutua esclusione*/
}

public void richiesta_lettura() {
   P(mutex);
   if(sospesi_scrit>0 || scrittore_attivo) {
   /*condizione di sincronizzazione non vera per un lettore*/
      sospesi_lett++;
      V(mutex);
      P(priv_lett); /*sospensione del processo lettore*/
      sospesi_lett--;
   }
   lettori_attivi++; /*la lettura viene abilitata*/
   if(sospesi_lett>0) V(priv_lett); /*se ci sono processi lettori
      bloccati, un altro può essere attivato e questo, a sua volta, terminando
      la richiesta_lettura ne sveglierà un altro e così via per gli altri*/
   else V(mutex); /*quando non ci sono ulteriori lettori bloccati viene
            rilasciata la mutua esclusione*/
}

public void fine_lettura() {
   P(mutex);
   lettori_attivi--;
   if(lettori_attivi==0&&sospesi_scrit>0) /*se non ci sono
      ulteriori lettori attivi si sveglia un eventuale scrittore bloccato*/
      V(priv_scritt);
   else else V(mutex); /*altrimenti si rilascia la mutua esclusione*/
}
```

5.8 Realizzazione dei semafori

Come indicato nel paragrafo 5.1, la realizzazione dei semafori nel nucleo elimina ogni forma di attesa attiva dei processi, conseguenza della definizione della primitiva P. Per motivi di efficienza infatti ogni processo che non può proseguire l'esecuzione, anziché ripetere continuamente l'esame del valore del semaforo, verrà sospeso e il controllo del processore fisico su cui è eseguito assegnato a un altro processo. Il processo sospeso verrà successivamente risvegliato come conseguenza dell'esecuzione da parte di un altro processo di una operazione V sullo stesso semaforo.

Partendo quindi dalla definizione astratta della primitiva P, riportata nello schema a blocchi della figura 5.1, possiamo ottenere lo schema implementativo della primitiva come indicato nella figura 5.16a dove il ciclo di attesa attiva viene sostituito con un cambio di contesto, realizzato utilizzando le funzioni del nucleo (come indicato nel paragrafo 3.6), e dove il simbolo grafico ovale, che racchiude le operazioni, rappresenta il fatto che tali operazioni devono essere eseguite in forma atomica. Come visto nel paragrafo 5.1, se il valore del semaforo è nullo l'atomicità dell'operazione viene interrotta dopo la commutazione di contesto. Nella figura le linee tratteggiate rappresentano il flusso di controllo nel momento in cui il processo che si sospende verrà riattivato.

Avendo eliminato il ciclo di attesa attiva dall'interno della P, il processo sospeso non è in grado, da solo, di verificare quando il semaforo torna verde. Dobbiamo quindi modificare anche lo schema della primitiva V, in modo tale che sia questa a svolgere il compito di svegliare uno dei processi in attesa quando viene eseguita. Per rendere quindi la V compatibile con lo schema implementativo della P, possiamo modificare il suo schema come indicato nella figura 5.16b.

Le due primitive sono strutturate ciascuna come una sola istruzione `if-then-else`. Ciò significa che non tutte le operazioni V implicano un incremento del valore semaforico, così come non tutte le P terminano decrementandolo. Questo accade, in particolare, tutte le volte che nelle primitive viene eseguito il ramo `then`. Ciò potrebbe sembrare una violazione della relazione invariante (5.4), ma in realtà a ogni V che termina eseguendo il ramo `then` corrisponde la terminazione di una P (precedentemente sospesa anch'essa sul ramo `then`) e quindi il decremento e l'incremento del valore semaforico non eseguiti si compensano a vicenda. D'altra parte la strutturazione delle primitive in questo modo è necessaria per evitare corse critiche che potrebbero verificarsi in assenza del ramo `else` (cioè se le operazioni di incremento o decremento fossero sempre eseguite al termine di ogni primitiva). Infatti, per esempio, in tal caso un processo P_i che eseguisse una P su un semaforo con valore zero si bloccherebbe (ramo `then` della P); successivamente un secondo processo P_j, eseguendo la V, sveglierebbe il processo P_i e terminerebbe la primitiva incrementando il contatore semaforico; a questo punto, prima che il processo P_i svegliato possa terminare la primitiva P decrementando il semaforo, un terzo processo P_k potrebbe a sua volta iniziare l'esecuzione della P, trovando il valore del semaforo a uno e, quindi, terminare senza bloccarsi decrementando il semaforo. Successivamente, il processo P_i, terminando la P, decrementerebbe ulteriormente il semaforo portandolo a un valore negativo e, quindi, producendo un comportamento erroneo.

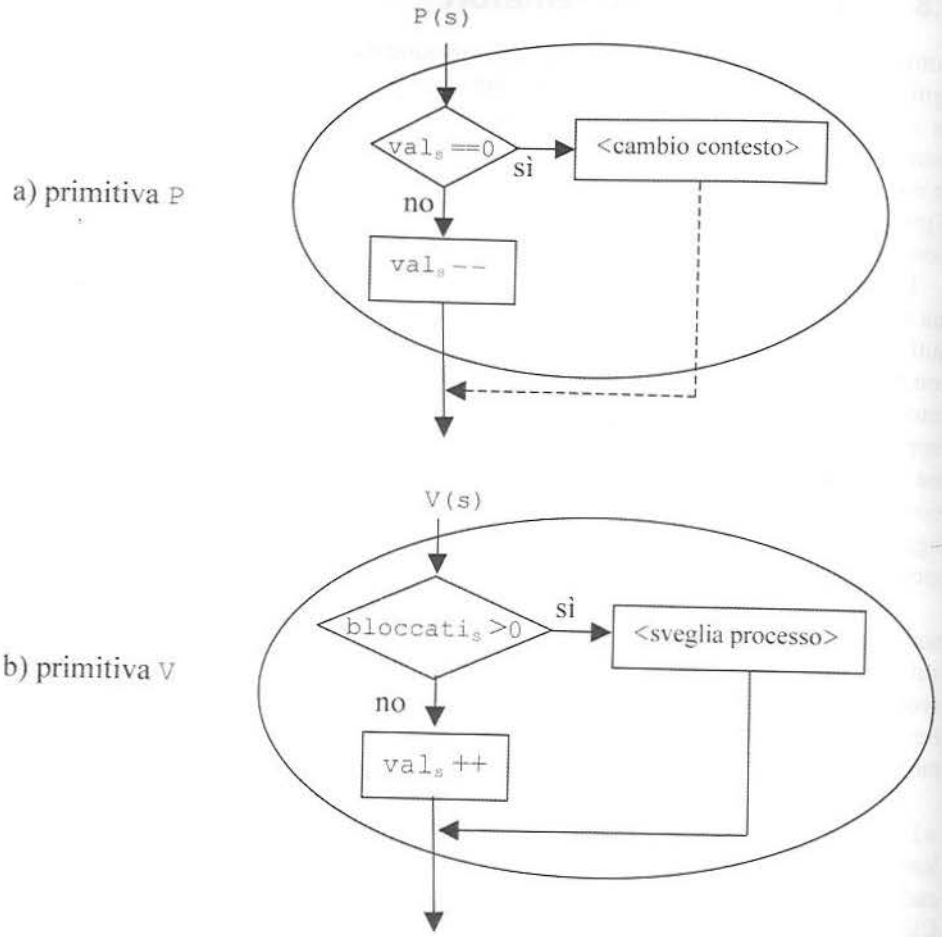


Figura 5.16 Schemi realizzativi delle primitive semaforiche.

Nel riquadro 5.4 viene effettuata la verifica che questo schema implementativo garantisce correttamente la validità dell'invariante semaforico (5.4) e, quindi, che la realizzazione delle due funzioni `P` e `V`, secondo questo schema, soddisfa la loro specifica come indicata nel paragrafo 5.11.

La realizzazione dei semafori comporta alcune aggiunte alle strutture del nucleo definite per fare fronte alle operazioni di creazione e terminazione dei processi e già viste nel paragrafo 3.6. In particolare introdurremo le funzioni `P` e `V` e una struttura dati (`des_semaforo`), per ogni semaforo `s` necessaria per mantenere aggiornate le due quantità che lo caratterizzano: val_s e $bloccati_s$. Con riferimento alle strutture dati già introdotte nel paragrafo 3.6, il tipo `des_semaforo` viene così definito:

Riquadro 5.4

VERIFICA DI CORRETTEZZA DELLA REALIZZAZIONE DELLE OPERAZIONI `P` E `V`

Utilizzando il principio di induzione, si può facilmente dimostrare che lo schema della figura 5.16 garantisce correttamente la validità dell'invariante semaforico (5.4). Infatti, questo è vero dopo l'inizializzazione di un semaforo `s` ($val_s=I_s+nv_s-np_s\geq 0$ essendo inizialmente $nv_s=np_s=0$). Inoltre, se l'invariante è vero prima di una `P` allora resta vero alla fine della stessa. Ciò è ovvio se, quando viene eseguita la `P`, si ha $val_s > 0$. In questo caso, infatti, viene decrementato val_s e quindi l'invariante resta vero alla fine della `P`, poiché entrambi i membri dell'uguaglianza (5.4) vengono decrementati (infatti, alla fine della `P`, la metavariabile np_s risulta ovviamente incrementata rispetto al valore che aveva all'inizio dell'esecuzione della stessa primitiva). Se, viceversa, all'inizio della `P` $val_s==0$, quando la primitiva termina (cioè quando il processo che la esegue è stato riattivato in seguito all'esecuzione di una `V`) il valore del semaforo è rimasto invariato mentre risulta incrementato il valore della metavariabile np_s essendo terminata la `P`. Però anche il valore della metavariabile nv_s risulta incrementato in quanto il processo che ha terminato la `P` è stato svegliato da una `V` che è terminata senza incrementare val_s (vedi figura 5.16). Quindi, anche in questo caso l'invariante resta vero alla fine della `P` poiché restano invariati i due membri dell'uguaglianza (5.4). Con analogo ragionamento possiamo verificare che anche la primitiva `V` mantiene vero l'invariante semaforico.

```
typedef struct{
   int contatore;
   coda_a_livelli coda;
   }des_semaforo;
```

L'introduzione della `P` comporta la presenza, per i processi, di un ulteriore possibile stato *bloccato* su un semaforo, in attesa del completamento dell'esecuzione della `P`. Per tenere traccia dei processi bloccati, ogni descrittore del semaforo conterrà, oltre al valore del semaforo (contenuto nel campo `contatore`), una coda di descrittori di processi bloccati su quel semaforo. Un'opportuna realizzazione delle primitive assicura inoltre che a un processo non venga indefinitamente impedito di proseguire la propria esecuzione. Ciò potrebbe verificarsi quando più processi sono contemporaneamente ritardati come conseguenza dell'operazione `P`. L'esecuzione della corrispondente `V` da parte di un processo consente a uno dei processi in attesa di proseguire. Se la scelta di tale processo avviene seguendo l'ordine di sospensione, la possibilità di un'attesa indefinita è eliminata (gestione FIFO della coda). Per generalità illustriamo comunque il caso in cui la coda del semaforo sia una coda a livelli come illustrato nel paragrafo 3.6.

L'insieme di tutti i descrittori dei semafori viene rappresentato dalla variabile `semafori` così definita:

```
des_semaforo semafori[num_max_sem];
```

dove la costante `num_max_sem` identifica il massimo numero di semafori che il sistema può supportare. All'interno del nucleo ogni semaforo verrà quindi identificato mediante l'indice del suo descrittore nel vettore `semafori`. Per questo motivo, con

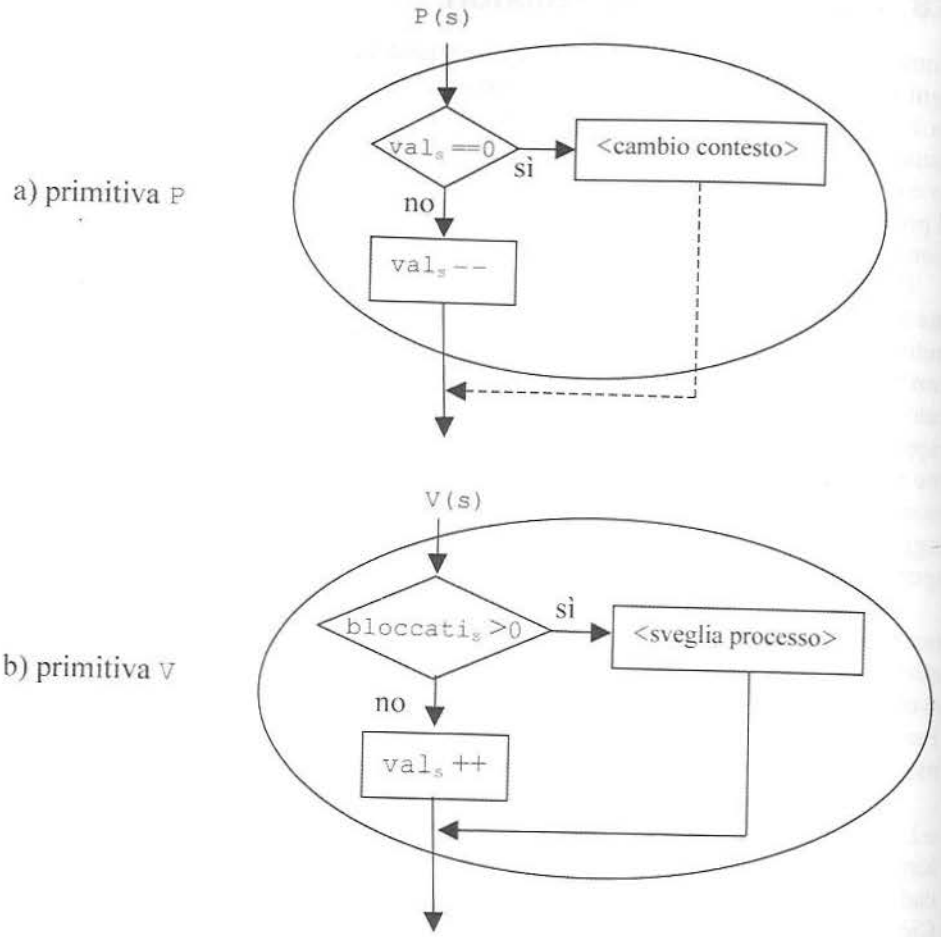


Figura 5.16 Schemi realizzativi delle primitive semaforiche.

Nel riquadro 5.4 viene effettuata la verifica che questo schema implementativo garantisce correttamente la validità dell'invariante semaforico (5.4) e, quindi, che la realizzazione delle due funzioni P e V, secondo questo schema, soddisfa la loro specifica come indicata nel paragrafo 5.11.

La realizzazione dei semafori comporta alcune aggiunte alle strutture del nucleo definite per fare fronte alle operazioni di creazione e terminazione dei processi e già viste nel paragrafo 3.6. In particolare introdurremo le funzioni P e V e una struttura dati (`des_semaforo`), per ogni semaforo `s` necessaria per mantenere aggiornate le due quantità che lo caratterizzano: val_s e $bloccati_s$. Con riferimento alle strutture dati già introdotte nel paragrafo 3.6, il tipo `des_semaforo` viene così definito:

Riquadro 5.4

VERIFICA DI CORRETTEZZA DELLA REALIZZAZIONE DELLE OPERAZIONI P E V

Utilizzando il principio di induzione, si può facilmente dimostrare che lo schema della figura 5.16 garantisce correttamente la validità dell'invariante semaforico (5.4). Infatti, questo è vero dopo l'inizializzazione di un semaforo `s` ($val_s = I_s + nv_s - np_s \geq 0$ essendo inizialmente $nv_s = np_s = 0$). Inoltre, se l'invariante è vero prima di una P allora resta vero alla fine della stessa. Ciò è ovvio se, quando viene eseguita la P, si ha $val_s > 0$. In questo caso, infatti, viene decrementato val_s e quindi l'invariante resta vero alla fine della P, poiché entrambi i membri dell'uguaglianza (5.4) vengono decrementati (infatti, alla fine della P, la metavariabile np_s risulta ovviamente incrementata rispetto al valore che aveva all'inizio dell'esecuzione della stessa primitiva). Se, viceversa, all'inizio della P $val_s == 0$, quando la primitiva termina (cioè quando il processo che la esegue è stato riattivato in seguito all'esecuzione di una V) il valore del semaforo è rimasto invariato mentre risulta incrementato il valore della metavariabile np_s essendo terminata la P. Però anche il valore della metavariabile nv_s risulta incrementato in quanto il processo che ha terminato la P è stato svegliato da una V che è terminata senza incrementare val_s (vedi figura 5.16). Quindi, anche in questo caso l'invariante resta vero alla fine della P poiché restano invariati i due membri dell'uguaglianza (5.4). Con analogo ragionamento possiamo verificare che anche la primitiva V mantiene vero l'invariante semaforico.

```
typedef struct{
   int contatore;
   coda_a_livelli coda;
   }des_semaforo;
```

L'introduzione della P comporta la presenza, per i processi, di un ulteriore possibile stato *bloccato* su un semaforo, in attesa del completamento dell'esecuzione della P. Per tenere traccia dei processi bloccati, ogni descrittore del semaforo conterrà, oltre al valore del semaforo (contenuto nel campo `contatore`), una coda di descrittori di processi bloccati su quel semaforo. Un'opportuna realizzazione delle primitive assicura inoltre che a un processo non venga indefinitamente impedito di proseguire la propria esecuzione. Ciò potrebbe verificarsi quando più processi sono contemporaneamente ritardati come conseguenza dell'operazione P. L'esecuzione della corrispondente V da parte di un processo consente a uno dei processi in attesa di proseguire. Se la scelta di tale processo avviene seguendo l'ordine di sospensione, la possibilità di un'attesa indefinita è eliminata (gestione FIFO della coda). Per generalità illustriamo comunque il caso in cui la coda del semaforo sia una coda a livelli come illustrato nel paragrafo 3.6.

L'insieme di tutti i descrittori dei semafori viene rappresentato dalla variabile `semafori` così definita:

```
des_semaforo semafori[num_max_sem];
```

dove la costante `num_max_sem` identifica il massimo numero di semafori che il sistema può supportare. All'interno del nucleo ogni semaforo verrà quindi identificato mediante l'indice del suo descrittore nel vettore `semafori`. Per questo motivo, con

semaphore denotiamo un alias del sottoinsieme degli interi compresi tra zero e num_max_sem-1 e utilizzati per identificare i semafori:

```
typedef int semaphore;
```

Nel seguito verranno presentate le realizzazioni delle primitive P e V separatamente per l'ambiente monoprocessore e multiprocessore.

5.8.1 Ambiente monoprocessore

In ambiente monoelaboratore, l'atomicità della P e della V viene garantita, come indicato nel paragrafo 3.6 e come illustrato nella figura 3.8, dal fatto che, in quanto primitive di nucleo, le due operazioni semaforiche vengono eseguite a interruzioni disabilitate.

In base allo schema illustrato nella figura 5.16 e ricordando le strutture dati e le funzioni introdotte nel paragrafo 3.6, possiamo quindi dettagliare la funzione primitiva P:

```
void P (semaphore s){
   p_des esec=processo_in_esecuzione;
   int pri=esec -> servizio.priorità;
   if(semafori[s].contatore==0) {
      esec -> stato=<"sospeso sul semaforo s">;
      inserimento (esec.semafori[s].coda[pri]);
      assegnazione_CPU;
   }
   else contatore--;
}
```

Se il valore del semaforo è positivo, questo viene decrementato di una unità e il processo prosegue l'esecuzione; se il valore è nullo, l'esecuzione della P provvede a modificare lo stato del processo che da attivo diventa bloccato e a inserire il suo descrittore nella coda associata al semaforo. L'esecuzione del processo è sospesa e l'unità di elaborazione così liberata viene assegnata a un altro processo. Come già detto, l'accodamento del descrittore avviene rispettando la priorità del processo. In molti casi la gestione avviene secondo una politica FIFO. È da notare che la politica FIFO è di regola preferita per motivi di generalità (e per evitare, come si è detto, situazioni di attesa indefinita). In questo caso sarebbe sufficiente definire la coda del semaforo a un solo livello di priorità.

Se non vi sono processi in attesa nella coda associata al semaforo, l'esecuzione della primitiva V da parte di un processo P_i provvede ad aumentare di una unità il valore del semaforo; P_i procede quindi nella sua esecuzione. Se vi sono processi in attesa nella coda, il più importante di essi (o il primo, a seconda del tipo di gestione) P_k viene prelevato. Utilizzando poi la funzione attiva già vista nel paragrafo 3.6, la sua priorità viene confrontata con quella di P_i. Se la priorità di P_i è maggiore o uguale a quella di P_k, P_k viene inserito nella coda dei processi pronti e P_i prosegue l'esecuzione; viceversa, se la priorità di P_i è minore di quella di P_k, P_i viene inserito nella coda dei processi pronti e P_k viene posto in esecuzione.

```
void V (semaphore s) {
   p_des esec=processo_in_esecuzione;
   p_des p;
   int pri=0;
   while(semafori[s].coda[pri].primo==NULL&&pri<min_priorità)
      pri++;
   if(semafori[s].coda[pri].primo!=NULL) {
   p=prelievo(semafori[s].coda[pri])
   attiva(p);
   }
   else semafori[s].contatore++;
}
```

5.8.2 Ambiente multiprocessore

Facendo riferimento a quanto detto nel paragrafo 3.6.2, si considera l'ipotesi di un sistema multiprocessore in cui i diversi processori hanno accesso a una memoria comune e i diversi processi possono operare su ogni processore. In questo caso, come si è visto, esiste un'unica copia del nucleo nella memoria comune.

Per quanto riguarda le funzioni P e V prima viste per un ambiente monoelaboratore non cambia niente relativamente alla loro funzionalità. La differenza sostanziale fra i due ambienti è quella relativa ai meccanismi necessari per garantire che tutte le funzioni del nucleo siano eseguite come operazioni atomiche. Per garantire l'atomicità delle funzioni del nucleo il criterio più semplice consiste nel rendere esclusivo l'accesso alle sue strutture dati. La presenza di più processori e quindi di più processi contemporaneamente in esecuzione, implica che l'esecuzione delle funzioni di nucleo in modo non interrompibile non è più sufficiente a garantire la loro atomicità. Infatti due funzioni di nucleo possono operare contemporaneamente sulla stessa struttura dati anche se eseguite sui rispettivi processori a interruzioni disabilitate. È quindi necessario prevedere un diverso meccanismo per garantire la mutua esclusione fra le esecuzioni di funzioni del nucleo su processori diversi.

Per risolvere il problema è necessario ipotizzare la disponibilità di alcune istruzioni macchina che, normalmente disponibili nei processori adatti a essere utilizzati in architetture multielaboratore, consentano l'ispezione e la modifica di una locazione in memoria in modo indivisibile, cioè in un solo ciclo di memoria. Per esempio, un'istruzione spesso offerta per questo scopo è l'istruzione test_and_set il cui comportamento può essere così definito:

```
boolean test_and_set(boolean*a) {
   boolean test=*a;
   *a=true;
   return test;
}
```

intendendo con ciò rappresentare la descrizione del microprogramma dell'istruzione test_and_set e non una funzione: come tale, quindi, rappresenta un'operazione atomica. Come si può osservare dalla sua descrizione, questa istruzione ha come operando una locazione di memoria (che si suppone contenere un valore binario: ze-

ro o uno). In un solo ciclo di memoria l'istruzione testa il valore di tale locazione, valore che viene restituito, e, nello stesso ciclo di memoria, pone tale valore a uno.

Avendo a disposizione tale istruzione possiamo realizzare le due seguenti operazioni (`lock` e `unlock`) che implementano i due schemi riportati nella figura 5.17:

```
void lock(boolean*x) {
   while(test_and-set(x));
}
void unlock(boolean*x) {
   *x=false;
}
```

Tali operazioni sono naturalmente atomiche, in quanto le operazioni riporate nei simboli ovali della figura sono due istruzioni macchina: la `test_and-set(x)` nell'operazione `lock` e l'istruzione di azzeramento della locazione `x` nella `unlock`. Ma, come si può osservare confrontando la figura 5.17 con la 5.1, le due primitive corrispondono funzionalmente a una `P` e una `V` su un semaforo binario[3] rappresentato dalla locazione `x`. Se quindi associamo la locazione `x` alle strutture del nucleo inizializzandola al valore `false` e se ogni operazione del nucleo inzia con `lock(x)` e termina con `unlock(x)`, è evidente che tutte le operazioni del nucleo vengono eseguite in mutua esclusione anche se eseguite su processori diversi. In questo caso, se una primitiva di nucleo viene invocata da un processo P_i, che esegue sul processore `i`-esimo mentre è in esecuzione un'altra primitiva di nucleo da parte del processo P_j su un diverso processore, per esempio il `j`-esimo, il processo P_i resta in attesa attiva all'interno della `lock` mantenendo occupato il suo processore. È evidente, però, che tale attesa attiva non crea grossi problemi di efficienza in quanto è limitata al completamento dell'esecuzione della primitiva da parte di P_j sul processore `j`-esimo.

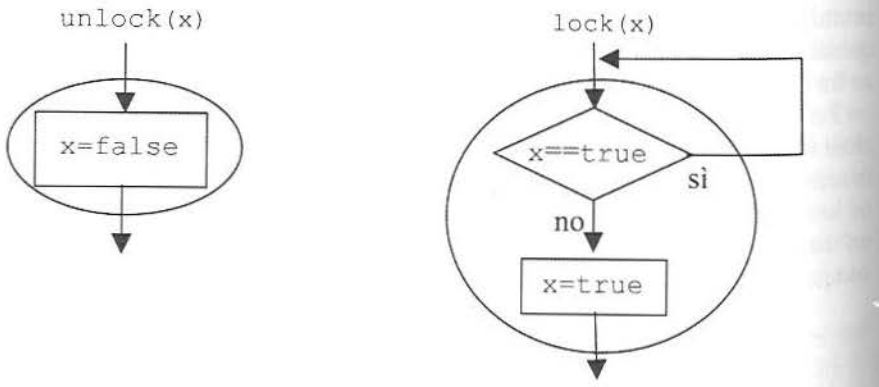


Figura 5.17 Schemi delle primitive `lock` e `unlock`.

[3] Le due operazioni `lock` e `unlock` su una locazione `x` vengono spesso indicate come operazioni di *spin lock* per indicare che la `lock` prevede un'attesa circolare nel caso in cui la locazione `x` contenga il valore `true`; oppure, ancora, come *spin semaphore* per indicare la somiglianza con le operazioni semaforiche.

La soluzione di garantire l'accesso esclusivo al nucleo tramite `lock(x)` e `unlock(x)` ha il pregio della semplicità; tuttavia presenta l'inconveniente di limitare il grado di parallelismo del sistema, escludendo a priori ogni possibilità di esecuzione contemporanea di più funzioni del nucleo, per esempio due `P` su due semafori diversi.

Un maggior grado di concorrenza può essere ottenuto proteggendo sia i singoli semafori, tramite differenti variabili logiche `x` a essi associate e usate dalle `lock` e `unlock`, sia la coda dei processi pronti. In questo caso due operazioni `P` su semafori diversi possono operare in modo contemporaneo se non risultano sospensive. In caso contrario, vengono sequenzializzati solo gli accessi alla coda dei processi pronti.

L'esecuzione di una `V`, in entrambi i casi, può risultare nell'attivazione di un processo con priorità superiore ad almeno uno dei processi in esecuzione. Come è stato mostrato nel paragrafo 3.6.2, è pertanto necessario che il nucleo provveda a revocare l'unità di elaborazione al processo con priorità più bassa e ad assegnarla al processo riattivato.

Un'ultima considerazione riguarda la necessità di continuare a eseguire le funzioni di nucleo a interruzioni disabilitate anche in presenza del meccanismo di `lock` ed `unlock`. Questa necessità deriva dal fatto che, altrimenti, non potremmo garantire che l'attesa attiva su un processore sia limitata nel tempo. Infatti, riprendiamo l'esempio precedente in cui P_i è in attesa attiva sul processore `i`-esimo perché deve attendere che P_j termini l'esecuzione di una funzione di nucleo sul processore `j`-esimo. Se P_j esegue questa funzione di nucleo a interruzioni abilitate può essere interrotto e, sul suo processore, può essere schedulato un terzo processo P_k. In questo caso P_i resta in attesa attiva poiché P_j, che è stato interrotto, non ha la possibilità di completare nel più breve tempo possibile la funzione di nucleo che sta eseguendo.

5.8.3 Una diversa realizzazione delle primitive semaforiche

Spesso viene utilizzata una realizzazione delle primitive semaforiche diversa rispetto a quella illustrata precedentemente (vedi figura 5.16) e riportata per esteso nel paragrafo 5.8.1. In questa diversa implementazione un semaforo viene ancora rappresentato in memoria mediante una struttura esattamente uguale a quella vista precedentemente:

```
typedef struct{
   int contatore;
   coda_a_livelli coda;
}descrittore_semaforo
```

ma dove il significato del campo `contatore` è diverso da prima. Per ogni semaforo `s`, il campo `contatore` invece di contenere il valore del semaforo val_s contiene il seguente valore:

$$\texttt{contatore} == val_s - bloccati_s \tag{5.19}$$

Quindi, tenendo conto delle relazioni (5.3) e (5.4) che definiscono le metavariabili val_s e $bloccati_s$, ne consegue che nel campo `contatore` è sempre presente il seguente valore:

$$\texttt{contatore} == I_s + nv_s - ni_s \tag{5.20}$$

In base a tali relazioni si possono individuare tre diversi stati di un semaforo in ognuno dei quali il valore del `contatore` assume un diverso significato:

- stato in cui un semaforo `s` ha un valore val_s positivo (semaforo verde); in questo caso, per la prima relazione (5.6), $bloccati_s==0$ e quindi, dalla (5.19), risulta che il valore del contatore coincide con val_s;
- stato in cui un semaforo `s` ha un valore val_s nullo (semaforo rosso) e non ci sono processi bloccati sul semaforo; in questo caso, poiché $val_s==bloccati_s==0$, dalla (5.19) risulta ancora che il valore del contatore coincide con val_s;
- stato in cui un semaforo `s` ha un valore nullo (semaforo rosso) e vi sono alcuni processi bloccati su `s`; in questo caso dalla (5.19) risulta che il valore del contatore coincide, a meno del segno, con il numero di processi bloccati sul semaforo ($contatore==-bloccati_s$).

Ricapitolando, con questa implementazione, se il valore del contatore di un semaforo è positivo o nullo, tale valore coincide con il valore del semaforo e inoltre il numero dei processi bloccati su di esso è nullo (coda vuota); se, viceversa, il valore del contatore è negativo, allora il suo valore assoluto coincide con il numero dei processi bloccati nella coda del semaforo (semaforo ovviamente con valore nullo).

La relazione (5.20) indica in che modo le due operazioni `P` e `V` devono operare in questo caso sul contatore del semaforo: il contatore viene inizializzato con il valore I_s, viene incrementato in seguito all'esecuzione di una `V` e viene decrementato quando inizia (e non quando termina) l'esecuzione di una `P` (vedi figura 5.18).

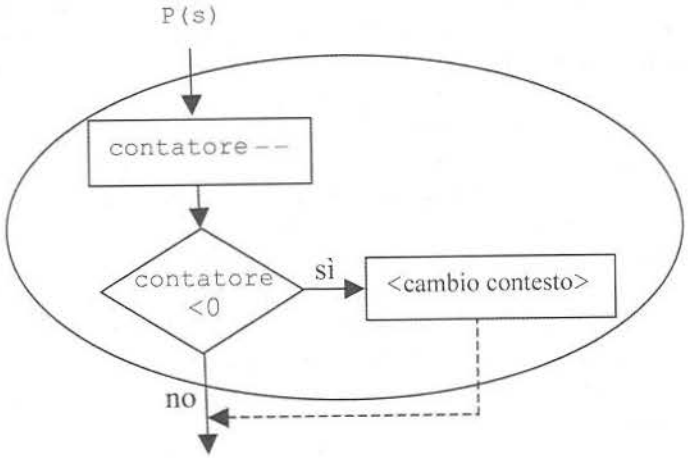
All'interno della primitiva `P`, una volta decrementato il contatore, se ne verifica il valore e se questo è negativo il processo viene bloccato. Infatti ciò significa che, prima del decremento, il contatore del semaforo aveva sicuramente un valore minore o uguale a zero, rappresentando, per quanto detto prima, uno stato in cui il semaforo è rosso. Analogamente nella primitiva `V`, una volta incrementato il contatore, se ne valuta il valore e, qualora questo sia negativo o nullo, viene svegliato un processo. Infatti ciò significa che, prima dell'incremento, il valore del contatore era sicuramente negativo, cioè, per quanto detto prima, vi sono processi bloccati in coda.

In base allo schema illustrato nella figura 5.18 possiamo quindi dettagliare l'implementazione delle due primitive `P` e `V`:

```
void P (semaphore s){
   p_des esec=processo_in_esecuzione;
   int pri;
   semafori[s].contatore--;
   if(semafori[s].contatore<0) {
      pri=esec -> servizio.priorità;
      esec -> stato=<"sospeso sul semaforo s">;
      inserimento(esec.semafori[s].coda[pri]);
      assegnazione_CPU;
   }
}

void V (semaphore s) {
   p_des esec=processo_in_esecuzione;
   p_des p;
   semafori[s].contatore++;
   if(semafori[s].contatore≤0) {
      int pri=0;
      while(semafori[s].coda[pri].primo=0
         pri++;
      p=prelievo(semafori[s].coda[pri])
      attiva(p);
   }
}
```

a) primitiva `P`



b) primitiva `V`

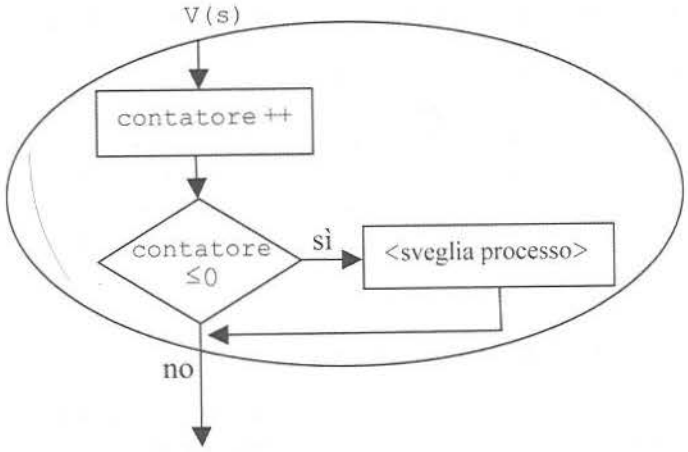


Figura 5.18 Una diversa implementazione delle `P` e `V`.

5.9 Sommario

Il meccanismo di sincronizzazione tra processi (o thread) più noto e utilizzato nei sistemi organizzati secondo il modello a memoria comune è sicuramente quello dei semafori. Scopo di questo capitolo è stato quello di presentare tale meccanismo con l'obiettivo di definirne, in maniera precisa, la semantica, le proprietà, le modalità di uso nel risovere i vari problemi di interazione e sincronizzazione tra processi e la sua implementazione come meccanismo primitivo offerto dal nucleo del sistema operativo.

Il meccanismo semaforico costituisce, infatti, lo strumento primitivo di sincronizzazione più noto fra quelli offerti direttamente dalla macchina concorrente, cioè, come indicato nel paragrafo 2.3, dalla macchina astratta dedicata a fornire il supporto per l'esecuzione di un programma concorrente. Tale strumento viene offerto anche all'interno di librerie standard che consentono di implementare programmi concorrenti utilizzando linguaggi sequenziali come C o il C++ e chiamando le funzioni della libreria sia per la gestione della concorrenza che della sincronizzazione.

Per prima cosa è stata quindi definita la semantica del meccanismo in modo tale da fornire, in maniera precisa, la specifica del comportamento del meccanismo stesso. Tale specifica viene utilizzata, da un lato da chi usa il meccanismo per fornire le soluzioni ai vari problemi di interazione tra processi con lo scopo di verificare la correttezza di tali soluzioni, dall'altro da chi deve implementare il meccanismo, con l'obiettivo di verificare che tale implementazione corrisponda fedelmente alla sua specifica.

Una volta definita la specifica del meccanismo, invece che fornire alcuni esempi di uso dello stesso nella soluzione di classici problemi di sincronizzazione, si è preferito illustrare una serie di diversi paradigmi di uso dei semafori, ciascuno dei quali adatto a risolvere particolari categorie di problemi, illustrando ciascun paradigma mediante alcuni esempi. I singoli pradigmi sono stati identificati con nomi specifici che ne richiamano le proprietà: semafori binari di mutua esclusione, semafori evento, semafori binari composti, semafori condizione, semafori risorsa, semafori privati.

Alla fine, sono state illustrate le tecniche di implementazione del meccanismo semaforico all'interno del nucleo di un sistema operativo, sia per una classica architettura fisica monoelaboratore, sia nel caso più generale, di architetture multielaboratore.

5.10 Note bibliografiche

Il concetto di semaforo è stato introdotto da Dijkstra [30] come strumento per la soluzione di molti problemi di sincronizzazione tipici di un sistema operativo multiprogrammato [47].

Habermann [48] per primo ha fornito una specifica formale del comportamento del meccanismo semaforico. Dijkstra ha anche introdotto il concetto di semaforo privato [30] e Hoare [51] quello di semaforo binario composto (*split binary semaphore*). Andrews [50] ha introdotto la tecnica del passaggio del testimone.

In [51] viene discusso il problema dei lettori/scrittori, ulteriormente approfondito in [52].

Patil [53] e Parnas [54] hanno discusso su potenzialità e limiti del meccanismo semaforico. Altri esempi e paradigmi di uso dei semafori e altri riferimenti bibliografici possono trovarsi su testi generali di programmazione concorrente [41] [17].

Alcuni riferimenti all'implementazione del meccanismo semaforico, soprattutto in ambiente monoelaboratore, possono trovarsi in testi generali di sistemi operativi [39] [40] [26] oltre che in [31].

Per la programmazione concorrente mediante la libreria Pthread si può fare riferimento a [34] [35].

6
Monitor

Come si è visto nel capitolo precedente, è possibile risolvere con i semafori e le primitive `P` e `V` ogni problema di sincronizzazione. Essi rappresentano, tuttavia, strumenti di sincronizzazione a basso livello il cui uso può dare luogo a diversi tipi di errore.

Si consideri, per esempio, il problema della mutua esclusione. Tutti i processi hanno in comune un semaforo `mutex`, inizializzato a 1. Ciascun processo deve eseguire `P(mutex)` prima di entrare nella sezione critica e `V(mutex)` prima di abbandonarla. Se non viene osservata questa sequenza, più processi possono entrare contemporaneamente nella sezione critica, dando così luogo a un comportamento scorretto del sistema.

Si supponga che un processo scambi l'ordine con cui esegue le operazioni `P` e `V`:

```
.......
V(mutex);
<sezione critica>;
S(mutex);
.......
```

In questa situazione vari processi possono eseguire la loro sezione critica contemporaneamente, violando il requisito di mutua esclusione. Si noti che l'errore si manifesta solo per particolari velocità relative dei processi: è necessario infatti che i processi siano simultaneamente attivi entro la loro sezione critica.

Un altro tipo di errore si ha quando il programmatore scrive in un processo:

```
......
P(mutex);
<sezione critica>;
P(mutex);
.......
```

Il processo, una volta entrato nella sezione critica, non è più in grado di uscirne (situazione di stallo o blocco critico). Situazioni di tipo analogo si hanno qualora il programmatore ometta una `P` o una `V` o scambi il nome di un semaforo.

Come già visto nel capitolo 5, le primitive P e V sono utilizzate per risolvere non solo problemi di mutua esclusione ma anche altri problemi di sincronizzazione. Questo rende particolarmente complesso verificare il corretto uso dei semafori e quindi la correttezza della soluzione.

La possibilità di commettere errori è tanto più probabile quanto più complesso è il problema da risolvere; la complessità del problema, come si è visto, obbliga infatti a introdurre un numero crescente di semafori, aumentando in tal modo la difficoltà di un controllo sulla correttezza della soluzione e rendendo di difficile comprensione il testo dei programmi.

Si è posto quindi il problema di introdurre nei linguaggi di programmazione concorrenti strumenti di sincronizzazione a più alto livello, che rendano più semplice, più comprensibile e più facile da verificare la soluzione ai vari problemi di sincronizzazione.

6.1 Definizione del monitor

Come già indicato a proposito del tipo di dato astratto, è opportuno che per motivi di protezione l'accesso a una risorsa comune da parte di processi possa avvenire solo tramite operazioni associate alla risorsa. In altri termini si tende a impedire che i singoli processi abbiano visibilità diretta della rappresentazione interna della risorsa.

Il tipo di dato astratto, come è noto, consente di separare la specifica di una struttura dati dalla sua realizzazione. Volendo estendere tale concetto a un ambito multiprogrammato è necessario, poiché la struttura di dati diventa accessibile a più processi, specificare la sequenza con la quale le operazioni possono accedere a tale struttura. In altre parole è necessario aggiungere alla definizione di tipo di dato astratto una specifica della sincronizzazione tra le esecuzioni delle operazioni.

Nel seguito verrà presentato un metodo per associare alla definizione di tipo di dato astratto le specifiche di sincronizzazione. In questo metodo, che dà luogo al concetto di *monitor*, le specifiche sono contenute nelle operazioni cui è riservato il compito di sospendere i processi che non possono avere accesso alla struttura dati.

6.1.1 Introduzione al concetto di monitor

Il monitor è un costrutto sintattico che associa un insieme di operazioni a una struttura dati (risorsa) comune a più processi, consentendo al compilatore del linguaggio di verificare che esse siano le sole operazioni permesse su quella struttura e assicurando la loro mutua esclusione (cioè un processo alla volta può essere attivo entro il monitor).

Il costrutto `monitor` è sintatticamente del tutto analogo al costrutto `class` ma, mentre quest'ultimo viene utilizzato per definire tipi di risorse dedicate, il `monitor` verrà usato per definire tipi di risorse condivise. Esso è stato introdotto per facilitare la programmazione strutturata di problemi in cui è necessario controllare l'assegnazione di una o più risorse tra più processi concorrenti, secondo determinati algoritmi di gestione.

Come si è detto, la mutua esclusione nell'accesso alla risorsa controllata dal monitor è garantita *implicitamente* assicurando che le operazioni del monitor non possano essere eseguite concorrentemente.

```
monitor tipo_risorsa {
    <dichiarazione delle variabili locali>;
    {<inizializzazione delle variabili locali>};
        public void op1() {
            <corpo della funzione op1>;
    }
    ..........
    public void opn() {
        <corpo della funzione opn>;
    }
}
```

Figura 6.1 Schema di massima del costrutto monitor.

Per la realizzazione di politiche di sincronizzazione nell'accesso alla risorsa da parte dei processi il monitor mette a disposizione un nuovo tipo di strumento rappresentato dalle *variabili condizione*, il cui utilizzo esplicito da parte dei processi risulta, come si vedrà, più semplice di quello dei semafori.

Per la loro potenza espressiva (soprattutto se confrontata con quella dei semafori) i monitor sono stati utilizzati in molti linguaggi concorrenti. Nel seguito verrà presentata la realizzazione del monitor nel linguaggio Java.

Lo schema di massima del costrutto monitor può essere presentato come riportato nella figura 6.1, in analogia a quanto introdotto per la classe nel paragrafo 5.2.

Le variabili locali descrivono lo stato della risorsa controllata dal monitor e le operazioni `public` costituiscono le uniche operazioni utilizzabili dai processi per accedere alle variabili comuni.

Le variabili comuni sono dette anche *permanenti*, in quanto mantengono il loro valore tra successive chiamate alle operazioni del monitor. Esse sono accessibili solo entro il monitor. Il codice per la loro inizializzazione viene eseguito una sola volta prima dell'esecuzione di qualunque operazione.

Le operazioni del monitor possono avere parametri e variabili locali, ciascuno dei quali assume un nuovo valore per ogni chiamata della procedura.

Oltre alle operazioni dichiarate `public` possono essere definite nel monitor altre operazioni non direttamente accessibili dall'esterno e utilizzabili quindi solamente dalle operazioni `public`.

Una particolare istanza del monitor, cioè una struttura dati organizzata come indicato nella dichiarazione dei dati locali al monitor e accessibile solo attraverso le operazioni definite nel monitor, viene creata con una dichiarazione del tipo:

```
tipo_risorsa ris;
```

La chiamata di una generica operazione dell'oggetto `ris` ha quindi la forma:

```
ris.op1();
```

Come si è detto, lo scopo del monitor è quello di controllare l'assegnazione di una risorsa tra processi concorrenti in accordo a determinate politiche di gestione. Questa assegnazione avviene attraverso due livelli di controllo. Il primo garantisce che

un solo processo alla volta può avere accesso alle variabili comuni controllate da un'istanza del monitor. Ciò è ottenuto imponendo, come si è detto, che le operazioni vengano eseguite in modo mutuamente esclusivo. I processi che richiedono l'uso di una operazione dell'istanza, mentre un altro processo è attivo nell'istanza stessa, devono essere ritardati.

La mutua esclusione è garantita dal supporto a tempo di esecuzione del linguaggio concorrente, da opportune librerie o dal sistema operativo, a seconda di come è implementato il monitor. Più avanti verrà esaminata la realizzazione del monitor tramite semafori.

Il secondo livello controlla l'ordine con il quale i processi hanno accesso alla risorsa. Questo tipo di ordinamento può essere ottenuto imponendo che un processo acceda alle variabili comuni dell'istanza del monitor solo se è soddisfatta una condizione logica che assicura l'ordinamento stesso (*condizione di sincronizzazione*). Nel caso ciò non si verifichi, il processo deve essere sospeso e perdere l'accesso esclusivo alle variabili comuni, in modo da consentire a un altro processo di accedere all'istanza del monitor, modificare tali variabili e creare così le condizioni per la sua riattivazione.

La possibilità di uso della risorsa dipende dal valore (vero o falso) di una condizione di sincronizzazione costituita da variabili locali al monitor, relative allo stato della risorsa, e da variabili proprie del processo, passate come parametri alla particolare operazione chiamata.

La sospensione del processo nel caso in cui la condizione non sia verificata, avviene utilizzando variabili di un nuovo tipo, detto tipo `condition`. La dichiarazione di una variabile `cond` di tipo `condition` ha la forma:

```
condition cond;
```

Le operazioni primitive che possono essere eseguite su una variabile di tipo `condition` sono `wait` e `signal`. L'operazione `wait(cond)` provoca la sospensione del processo che la esegue fino che un altro processo esegue `signal(cond)`. L'operazione `signal(cond)` risveglia un processo sospeso sulla variabile condizione `cond`. Se non vi sono processi sospesi sulla variabile condizione `cond` la `signal` non ha effetto.

Le operazioni `wait` e `signal` realizzano una politica FIFO sia per l'accodamento che per il risveglio dei processi (come si vedrà nel seguito è possibile imporre politiche basate sulla priorità dei processi).

È opportuno mettere in evidenza a questo punto le differenze tra le operazioni `wait` e `signal` eseguite su variabili di tipo `condition` e le primitive, già introdotte, `P` e `V` eseguite sui semafori. La `wait` eseguita su una variabile condizione provoca sempre la sospensione del processo invocante. La `P` eseguita su un semaforo è sospensiva per il processo solo se il semaforo ha valore zero. La `signal` eseguita su una variabile condizione la cui coda associata è vuota, non ha alcun effetto mentre, se la coda associata al semaforo è vuota, l'esecuzione della `V` comporta l'incremento del semaforo. La differenza tra il comportamento delle primitive nei due casi deriva dalla differente semantica associata a un semaforo e a una variabile di tipo `condition`. Un semaforo è una struttura dati definita da due componenti, ovvero un numero intero maggiore o uguale a zero, che rappresenta il valore del semaforo, e la coda dei processi eventualmente sospesi sul semaforo. La variabile condizione rappresenta invece solo la coda dei processi eventualmente sospesi.

6.1.2 Semantiche dell'operazione signal

Sia P un processo che si è sospeso su una variabile condizione `cond`, non essendo soddisfatta la condizione di sincronizzazione, e Q il processo che, dopo aver reso vera tale condizione, esegue la `signal` su `cond`. Come conseguenza della `signal` entrambi i processi possono, concettualmente, proseguire la loro esecuzione. Ciò è tuttavia impedito dalla proprietà del monitor che consente a un solo processo alla volta di essere attivo al suo interno. Uno dei due processi deve pertanto essere sospeso in attesa che l'altro completi la sua esecuzione o venga a sua volta sospeso.

Esistono a questo proposito due possibili strategie:

a) La strategia *signal_and_wait* prevede che il processo P risvegliato riprenda immediatamente l'esecuzione e che il processo Q venga sospeso; ciò per evitare che il processo Q, proseguendo, possa modificare la condizione di sincronizzazione rendendola non più vera per il processo P. Il processo Q passa quindi il controllo esclusivo del monitor al processo P (tecnica di passaggio del testimone) e si sospende rientrando nella coda dei processi in attesa di utilizzare il monitor (*entry_queue*). Una volta che P abbia terminato la sua azione (o si sia nuovamente sospeso), Q potrà riacquisire, quando sarà il suo turno (la coda *entry_queue* può contenere altri processi), l'uso esclusivo del monitor e riprendere la sua esecuzione.

Un caso particolare di questa strategia, è quella proposta da Hoare [49] e va sotto il nome di *signal_and_urgent wait*. Tale soluzione prevede che il processo Q abbia la priorità su ogni altro processo che intende entrare nel monitor; ciò si può ottenere sospendendo il processo Q su un apposita coda interna al monitor (*urgent_queue*) e garantendo che il processo P, una volta che abbia terminato la sua esecuzione (o si sia nuovamente sospeso), trasferisca il controllo a Q senza liberare il monitor.

La semantica *signal_and_wait* privilegia quindi il processo segnalato rispetto al segnalante. Ciò implica, come si è detto, che il segnalato, proseguendo la sua esecuzione, è certo di trovare vera la condizione per la quale è stato risvegliato. Quindi, con questa politica lo schema tipico di uso della primitiva `wait(cond)` è all'interno di un'istruzione condizionale del tipo (vedi schema con passaggio del testimone dei semafori condizione, paragrafo 5.5):

```
if (!B) wait(cond);
<accesso alla risorsa>;
```

L'espressione logica `B` rappresenta la condizione di sincronizzazione che deve essere verificata per consentire ai processi l'accesso alla sezione critica. Se `B` non è verificata i processi si sospendono sulla variabile condizione `cond`.

b) La politica *signal_and_continue* privilegia il processo segnalante rispetto al segnalato. Prevede che il processo segnalante Q prosegua la sua esecuzione mantenendo l'accesso esclusivo al monitor, dopo aver risvegliato il processo P. Il processo P viene trasferito dalla coda associata alla variabile condizione alla *en-*

try_queue e potrà rientrare nel monitor una volta che Q lo abbia liberato. Poiché nella *entry_queue* possono esservi altri processi, questi possono precedere l'esecuzione di P e quindi modificare il valore della condizione di sincronizzazione; P deve pertanto testare nuovamente tale condizione prima di proseguire la sua azione all'interno del monitor.

Ciò è vero a maggior ragione se si considera il fatto che il processo segnalante, proseguendo l'esecuzione, potrebbe invalidare la condizione per la quale P è stato risvegliato.

Quindi, con una politica del tipo *signal_and_continue* lo schema tipico di uso della primitiva `wait(cond)` è all'interno di un'istruzione di ciclo:

```
while(!B)wait(cond);
<accesso alla risorsa>;
```

La soluzione proposta può dar luogo a inutili ripetute valutazioni della condizione di sincronizzazione, tuttavia elimina la complessità realizzativa insita nella politica *signal_and_wait* (in particolare, in quella *signal_and_urgent wait)* e può risultare utile quando il processo segnalante abbia priorità sul segnalato.

Il fatto che il processo risvegliato venga inserito nella *entry_queue* e debba quindi riacquisire la mutua esclusione consente di realizzare anche la `signalAll`, che prevede che tutti i processi sospesi sulla variabile `cond` vengano riattivati contemporaneamente. Tali processi vengono tutti inseriti nella *entry_ queue* dalla quale, uno alla volta, potranno rientrare nel monitor.

La maggiore semplicità di tipo realizzativo fa sì che la *signal_and_continue* sia la semantica maggiormente utilizzata (vedi per esempio la realizzazione del monitor nel linguaggio Java e con la libreria `pthreads`).

Un caso particolare della *signal_and_urgent wait* (o della *signal_and_wait*) è rappresentato dalla soluzione proposta da Brinch Hansen e implementata nel Concurrent Pascal [24]. Tale soluzione prevede che la `signal`, se compare all'interno di una funzione, corrisponda a un'istruzione `return` (*signal_and_return*). In altri termini, il processo completa la sua operazione con il risveglio del processo segnalato. Quindi, non si pone più il problema della competizione tra processo segnalante e processo segnalato poiché il segnalante termina la sua funzione. Il segnalante può quindi cedere subito il controllo al segnalato senza rilasciare la mutua esclusione come nel caso della *signal_and_urgent wait,* con l'unica differenza che il segnalante non deve sospendersi su *urgent_queue,* in quanto ha terminato di operare sul monitor.

Prima di concludere questa parte sulle proprietà generali del monitor è importante ricordare alcuni punti che risultano fondamentali per la comprensione del funzionamento del monitor:

1. L'esecuzione della `signal` riattiva un processo sospeso su una variabile condizione. Nel caso ci siano più processi sospesi la politica adottata è quella FIFO, anche se, come vedremo tra poco, è possibile ordinare la coda dei processi sospesi in funzione della loro priorità.
2. La liberazione del monitor, cioè la possibilità che altri processi in attesa all'esterno possano utilizzare una delle operazioni del monitor, può avvenire solo se non ci sono, all'interno del monitor stesso, altri processi in grado di completare l'operazione richiesta (politica *signal_and_urgent wait*).
3. Dovrà essere cura del programmatore (come si è visto nel paragrafo 1.6 per i tipi di dati astratti) assicurare la consistenza della struttura dati del monitor ogni qualvolta questo viene reso disponibile ai processi, cioè in fase di inizializzazione, al completamento di una operazione e al momento della sospensione di un processo.

6.1.3 Ulteriori operazioni sulle variabili condizione

Nella presentazione delle operazioni `wait(cond)` e `signal(cond)` si è supposto fino a ora che l'algoritmo di gestione delle code associate alle variabili condizione fosse di tipo FIFO. L'esecuzione di un'operazione `signal` su una di tali variabili provoca quindi il risveglio del processo in attesa da più tempo. Questo tipo di gestione assicura l'assenza di condizioni di attesa indefinita per i processi sospesi.

Vi sono tuttavia dei casi in cui la politica di risveglio di tipo FIFO può risultare non adeguata, essendo necessario un maggior controllo sulla strategia di gestione della risorsa. Per questo motivo è stata introdotta la primitiva:

```
wait(cond,p);
```

dove `p` rappresenta una espressione intera, il cui valore viene valutato quando viene eseguita la `wait`. I processi sono accodati rispettando il valore (crescente o decrescente) di `p` e vengono risvegliati nello stesso ordine.

Può rivelarsi utile in alcuni casi, prima di eseguire l'operazione `signal` su una variabile condizione, sapere lo stato della coda associata, cioè se essa contiene elementi o è vuota. Si può utilizzare in questo caso l'operazione `empty(cond)`, la cui esecuzione fornisce il valore `false` se esistono processi sospesi nella coda associata alla variabile `cond`, `true` altrimenti.

Come si è accennato precedentemente, la `signal` consente il risveglio di un solo processo in coda alla variabile condizione. In taluni casi, tuttavia, più processi sospesi potrebbero logicamente riprendere l'esecuzione quando una condizione diventa vera (con il vincolo ovvio della mutua esclusione nell'esecuzione delle procedure del monitor). È utile pertanto introdurre una primitiva `signal` di tipo broadcasting:

```
signalAll(cond);
```

che prevede che tutti i processi sospesi sulla variabile `cond` vengano riattivati contemporaneamente. Come già indicato precedenteente, la `signalAll` risulta chiaramente definita soltanto quando viene usata una politica del tipo *signal_and_continue.*

Lo stesso risultato della `signalAll` è ottenibile utilizzando al posto della signal la seguente istruzione:

```
while(!empty(cond)) signal(cond);
```

L'utilizzo di quest'ultima soluzione potrebbe consentire anche l'utilizzo di una politica *signal_and_wait* o *signal_and_urgent wait,* che risulterebbe tuttavia poco efficiente per il numero di sospensioni del processo segnalante e per la necessità, comunque, che ogni processo segnalato vada a testare nuovamente la condizione di sincronizzazione.

```
monitor nome_monitor {
    <dichiarazione delle variabili locali>;
    [<inizializzazione delle variabili locali>];
    condition cond[num_max_proc];
    ......
    public void op1(int k) {
       if(!B) wait(cond[k]);
       ......
    }
    ......
    public void op2(...){
       ......
       signal(cond[i]);
       ......
    }
}
```

Figura 6.2 Variabili condizione monoprocesso.

Per concludere, è opportuno mettere in evidenza come, tramite un uso appropriato delle variabili `condition`, sia possibile raggiungere la stessa potenza espressiva nel linguaggio di quella ottenuta con l'uso dei semafori privati. Si supponga infatti che la politica di gestione della risorsa richieda che debba essere risvegliato, tra tutti i processi sospesi, un ben determinato processo. Le proprietà messe in evidenza fino a ora per le variabili `condition` non consentono di raggiungere questo obiettivo. Infatti i processi sospesi su una variabile `condition` possono essere risvegliati o con una politica FIFO o in base alla loro priorità.

L'obiettivo di risvegliare un ben determinato processo lo si ottiene definendo nel monitor un `array` di variabili `condition`, una per ogni possibile processo che utilizza le operazioni del monitor (variabili condizione *monoprocesso*). Si consideri, per esempio, lo schema di monitor riportato nella figura 6.2.

Si supponga che il generico processo P_i chiami la procedura op_1, passando come parametro il suo indice identificativo `i`. Se la condizione `B` non è soddisfatta il processo si sospenderà sulla variabile `cond[i]` che non potrà essere utilizzata da nessun altro processo per la sospensione. Se un altro processo intende risvegliare il processo P_i deve eseguire `signal(cond[i])`.

6.2 Esempi d'uso

Verranno nel seguito riportati due esempi di utilizzo del costrutto monitor con l'obiettivo di approfondire e meglio chiarire quanto detto precedentemente. In tali esempi verrà utilizzata la signal con la semantica *signal_and_urgent wait*. Nel caso in cui si volesse utilizzare la semantica *signal_and_continue* sarebbe necessario cambiare ogni istruzione `if` contenente una `wait` nella corrispondente istruzione `while`.

Il primo esempio riguarda l'utilizzo del monitor nel problema dei produttori/consumatori, già presentato nel paragrafo 5.6.2 a proposito dell'utilizzo dei semafori. In questo caso i processi potranno utilizzare solo le operazioni `invio` e `ricezione`, rispettivamente per inserire e prelevare messaggi dal `buffer`. La struttura dati che rappresenta il `buffer` appartiene alle variabili locali al monitor e quindi le operazioni `invio` e `ricezione` possono accedere solo in modo mutuamente esclusivo a tali operazioni.

Il secondo esempio riguarda l'utilizzo del monitor per garantire l'accesso esclusivo a una risorsa comune da parte dei processi. Le operazioni `richiesta` e `rilascio` del monitor sono utilizzate solo per garantire l'accesso esclusivo alla risorsa da parte dei processi. La struttura dati gestita dal monitor rappresenta quindi lo stato (libero o occupato) della risorsa e la mutua esclusione tra le operazioni `richiesta` e `rilascio` garantisce che lo stato della risorsa venga esaminato in modo mutuamente esclusivo dai processi. Una volta guadagnato l'accesso alla risorsa i singoli processi provvederanno ad accedere a essa all'esterno del monitor.

Esiste quindi una differenza fondamentale tra le due modalità di uso del monitor. Nel primo esempio le operazioni del monitor devono non solo sincronizzare i processi nell'accesso alla risorsa, ma anche operare su di essa. Nel secondo caso invece, come si è detto, il compito delle operazioni è solo quello di sincronizzare i processi.

L'importanza della differenza segnalata sta nella impossibilità, nel primo caso, di consentire l'accesso contemporaneo di più processi a una risorsa. Proprio nel problema dei produttori/consumatori è possibile e auspicabile, per aumentare il parallelismo nel sistema, che un produttore e un consumatore accedano contemporaneamente al buffer purché in diverse posizioni di esso.

Questo è il motivo per cui, in molti degli esempi che saranno riportati nel seguito (e in molti casi reali), compito delle operazioni del monitor sarà solo quello di realizzare la politica di sincronizzazione tra i processi, lasciando che siano i singoli processi, una volta autorizzati, ad accedere alla risorsa.

Ricordando quanto detto nel paragrafo 5.2, la risorsa potrà essere rappresentata tramite un oggetto astratto e quindi l'accesso da parte dei processi alla risorsa potrà avvenire in modo controllato solo tramite gli operatori associati all'oggetto astratto.

La politica di sincronizzazione realizzata dal monitor potrà quindi, a seconda del tipo di problema trattato, consentire l'accesso alla risorsa a un solo processo o a più contemporaneamente. Il primo esempio, relativo al problema dei produttori/consumatori viene riportato nella figura 6.3.

L'operazione `invio` provvede a inserire il messaggio passato come parametro nel `buffer` o a sospendere il processo nella coda associata alla variabile condizione `non_pieno` se il `buffer` risulta pieno. L'operazione `ricezione` restituisce un messaggio al processo chiamante o sospende il processo qualora il `buffer` sia vuoto. Se l'azione di deposito del messaggio ha successo, l'operazione `invio` provvede, tramite la `signal(non_vuoto)`, a riattivare un eventuale processo consumatore sospeso. Analogamente, se l'azione di prelievo ha successo, l'operazione `ricezione` provvede, tramite la `signal(non_pieno)`, a riattivare un eventuale processo sospeso.

Il secondo esempio relativo all'allocazione di una risorsa viene riportato nella figura 6.4. Come si può facilmente verificare, l'esempio descritto rappresenta la realizzazione di un semaforo binario.

Dall'analisi degli esempi riportati è possibile trarre alcune considerazioni circa le proprietà del monitor. Innanzitutto, la `signal` è stata sempre utilizzata come ulti-

```
monitor buffer_circolare {
   messaggio buffer[N];
   int contatore=0;
   int testa=0;
   int coda=0;
   condition non_pieno;
   condition non_vuoto;
public void invio(messaggio m){
   if (contatore==N) wait(non_pieno);
   buffer[coda]=m;
   coda=(coda+1)%N;
   contatore++;
   signal(non_vuoto);
   }
public messaggio ricezione(){
   messaggio m;
   if (contatore==0) wait(non_vuoto);
   m=buffer[testa];
   testa=(testa+1)%N;
   contatore--;
   signal(non_pieno);
   return m;
}
}
```

Figura 6.3 Scambio di messaggi.

ma operazione nelle procedure. Come si è detto, questa soluzione rende superflua una eventuale sospensione del processo segnalante per garantire che la condizione di sincronizzazione non sia più da esso modificata (politica *signal_and_return*). Il processo segnalante, dopo aver riattivato un processo sospeso su una variabile condizione, prosegue la sua esecuzione ed esce dal monitor senza tuttavia liberarlo, essendoci, al suo interno, un processo pronto ad andare in esecuzione. Qualora non ci siano processi sospesi sulla variabile condizione segnalata, l'esecuzione della `signal` provoca la liberazione del monitor. Inoltre, come si può osservare, l'analisi

```
monitor allocatore {
   boolean occupato=false;
   condition libero;
   public void richiesta(){
      if(occupato)wait(libero);
      occupato=true;
   }
   public void rilascio(){
      occupato=false;
      signal(libero);
   }
}
```

Figura 6.4 Allocazione di una risorsa.

della condizione di sincronizzazione viene fatta all'inizio della procedura chiamata, prima quindi di una qualsiasi modifica delle variabili permanenti del monitor. Poiché tale analisi può comportare la sospensione del processo e quindi la liberazione del monitor, questa avviene mantenendo la consistenza della struttura dati.

Si noti infine che l'aver utilizzato la `signal` come ultima operazione delle procedure ha consentito l'utilizzo di un'istruzione condizionale per l'analisi della condizione di sincronizzazione e l'eventuale sospensione sulla variabile condizione.

6.3 Realizzazione del costrutto monitor

In generale, è compito del supporto di esecuzione del linguaggio concorrente (nucleo) la realizzazione del costrutto monitor, così come di ogni altro strumento di sincronizzazione presente nel linguaggio.

Nel seguito verrà riportata la realizzazione del costrutto monitor in termini di semafori. Questa soluzione è utile nel caso di linguaggi o di librerie software che forniscono lo strumento semaforo e non il monitor, ma soprattutto ci serve per meglio comprendere le proprietà del costrutto monitor e la differenza tra le politiche di segnalazione e riattivazione dei processi.

Ricordiamo che la sincronizzazione tra i processi nell'accesso a risorse comuni avviene attraverso due livelli:

a) mutua esclusione nell'accesso alle variabili che rappresentano lo stato della risorsa (variabili permanenti), che si ottiene imponendo che le operazioni del monitor utilizzate dai processi per accedere alla risorsa debbano essere eseguite in modo mutuamente esclusivo;
b) sospensione e successiva riattivazione dei processi che, utilizzando una operazione del monitor, tentano di accedere alla risorsa in funzione dello stato della risorsa stessa e della natura dei processi. A questo scopo sono utilizzate le variabili condizione e le apposite primitive `wait` e `signal`.

Per quanto riguarda il primo livello, l'accesso esclusivo al monitor, la condizione di mutua esclusione tra le operazioni del monitor può essere semplicemente ottenuta associando a ogni istanza del monitor un semaforo `mutex` inizializzato a uno; la richiesta da parte di un processo di utilizzare una operazione equivale all'esecuzione di una `P(mutex)`.

Per quanto riguarda il rilascio del monitor da parte di un processo che si sospende su una variabile condizione o che abbandona il monitor, così come per la realizzazione delle operazioni sulle variabili condizioni, la soluzione adottata dipende dal tipo di politica adottata: *signal_and_continue, signal_and wait, signal_and_urgent wait* o *signal_and_return*.

Per quanto riguarda il secondo livello, la realizzazione delle operazioni sulle variabili condizione si ottiene associando a ogni variabile condizione `cond` un semaforo condizione (vedi paragrafo 5.5), cioè:

- un semaforo `condsem` inizializzato a zero, sul quale un processo può sospendersi tramite una `wait(condsem)`;
- un contatore `condcount`, inizializzato a zero, per tener conto dei processi sospesi sul semaforo `condsem`.

Nel paragrafo 5.8, relativamente alla realizzazione dei semafori si era supposto, i
generalità, che a ogni semaforo fossero associate più code di processi, una per o
livello di priorità. Nel seguito, in analogia con quanto stabilito per le variabili co
zioni, si farà l'ipotesi che al semaforo sia associata una sola coda gestita FIFO.

Signal_and_continue Si veda la figura 6.5. Si noti che il processo che si
spende sulla variabile condizione tramite `P(condsem)`, quando risvegliato, to
nella *entry_queue* tramite la `P(mutex)`.

```
Prologo di ogni funzione: P(mutex);
Epilogo di ogni funzione: V(mutex);
wait(cond): { condcount++;
              V(mutex);
              P(condsem);
              P(mutex);
            }
signal(cond): if(condcount>0) {
                 condcount--;
                 V(condsem);
              }
```

Figura 6.5 Politica *signal_and_continue*.

Signal_and_wait Si osservi la figura 6.6 Si noti che il processo segnalante s
glia il segnalato tramite la `V(condsem)` e si sospende nella *entry_queue* tram
la `P(mutex)`.

```
Prologo di ogni funzione: P(mutex);
Epilogo di ogni funzione: V(mutex);
wait(cond) { condcount++;
             V(mutex);
             P(condsem);
           }
signal(cond):if(condcount>0){
                condcount--;
                V(condsem);
                P(mutex);
             }
```

Figura 6.6 Politica *signal_and_wait*.

Signal_and_urgent wait La *signal_and_urgent wait* risulta più complessa
spetto alle altre politiche. L'esigenza, infatti, di garantire ai processi segnalant
priorità, nell'acquisizione del monitor, rispetto a tutti gli altri processi sospesi n
entry_queue, una volta che il processo segnalato abbia terminato la sua azio
comporta che tali processi vengano sospesi su una apposita coda associata a un n
vo semaforo `urgent` (inizializzato a zero) tramite una `P(urgent)`; compo

re che, prima di liberare il monitor, si verifichi che nessun processo sia in coda
e semaforo.
ndicando con `urgentcount` un contatore (inizializzato a zero) del numero dei
essi sospesi sul semaforo `urgent`, l'uscita da una operazione del monitor viene
anto così codificata:

```
urgentcount>0)V(urgent);
e V(mutex);
```

vrà pertanto il costrutto riportato nella figura 6.7.

```
ogo di ogni funzione: P(mutex);
ogo di ogni funzione: if(urgentcount>0)V(urgent);
                      else V(mutex);
(cond): { condcount++;
          if(urgentcount>0)V(urgent)else V(mutex);
          P(condsem);
          condcount--;
        }
nal(cond):if(condcount>0) {
             urgentcount++;
             V(condsem);
             P(urgent);
             urgentcount--;
          }
```

ra 6.7 Politica *signal_and_urgent wait*.

oti che il processo segnalante sveglia il segnalato tramite la `V(condsem)` e si
pende nella *urgent_queue* tramite la `P(urgent)`.

nal_and_return Si veda il costrutto riportato nella figura 6.8.

```
ogo di ogni funzione: P(mutex);
ogo di ogni funzione: se la funzione non contiene nessuna signal
llora Epilogo: V(mutex);
e la funzione contiene almeno una signal
llora Epilogo: corrisponde alla signal(cond) (vedi sotto):
(cond): {condcount++;
         V(mutex);
         P(condsem);
         condcount--;
         }
nal(cond): if(condcount>0)V(condsem);
           else V(mutex)
```

ra 6.8 Politica *signal_and_return*.

In tutti i casi esaminati, con riferimento al codice della `wait(cond)`, occorre notare che tra `V(mutex)` e `P(condsem)` il processo potrebbe essere interrotto. Questo può comportare il non rispetto della politica FIFO nella riattivazione dei processi. Come esercizio, si lascia al lettore l'individuazione della corretta soluzione al problema.

6.4 Realizzazione di politiche di gestione delle risorse

Vediamo adesso alcuni esempi di monitor utilizzati per definire politiche di gestione delle risorse.

Allocazione di una risorsa mediante la strategia *Shortest-job-next* Si consideri il caso di più processi che competono per l'uso di una risorsa. Quando la risorsa viene rilasciata, essa viene assegnata, tra tutti i processi sospesi, a quello che la userà per il periodo di tempo inferiore.

Siano:

```
public void richiesta(int tempo);
public void rilascio;
```

le due operazioni del monitor chiamate dai processi, rispettivamente, per chiedere l'accesso alla risorsa per un numero di unità di tempo indicate nel parametro tempo e per liberare la risorsa. Si avrà quindi il costrutto riportato nella figura 6.9.

L'esempio introdotto fa uso della primitiva `wait(cond,p)`, dove `cond` è la variabile condizione `non-occupata` e la priorità `p` è rappresentata in questo caso dal tempo richiesto. I processi sono inseriti nella coda secondo l'ordine crescente di `p` e quindi il primo processo risvegliato è quello che richiede meno tempo.

```
monitor allocatore{
   boolean occupata=false;
   condition non_occupata;
   public void richiesta(int tempo){
      if (occupato) wait(non_occupata, tempo);
      occupata=true;
   }
   public void rilascio(){
      occupato=false;
      signal(non_occupata);
   }
}
```

Figura 6.9 Utilizzo della primitiva `wait` con priorità per i processi sospesi.

Gestione di un disco a teste mobili In un ambiente in cui più processi competono per l'utilizzo di un disco, il Sistema Operativo provvede ad accodare le singole richieste e a servirle secondo diversi criteri di priorità. Rinviando al testo [31] per la descrizione delle diverse tecniche adottate, nel seguito verrà illustrata quella che va sotto il nome di *SCAN* che ha come obiettivo la minimizzazione del numero dei cambiamenti di direzione del braccio del disco. In altri termini, supponendo che il braccio si muova dalla traccia 0 (la più esterna) verso l'interno, verranno servite tutte le richieste che si trovano in quella direzione (le richieste sono servite secondo l'ordine di vicinanza alla richiesta corrente). Quando nella direzione scelta non ci sono più richieste, la direzione del braccio viene invertita e il procedimento è ripetuto. Compito del monitor, che verrà descritto successivamente, è quello di fornire ai processi le procedure `richiesta` e `rilascio`. La procedura `richiesta` ha il compito di accodare, se il disco è occupato, la richiesta del processo in funzione dello stato corrente del disco (direzione del braccio e numero di traccia interessata) in una coda associata a una variabile condizione, in modo da rispettare la politica di risveglio scelta. La procedura `rilascio` ha il compito di servire eventuali altre richieste che si trovino nella direzione scelta o di invertire la direzione del disco.

Per realizzare la politica SCAN occorre distinguere tra le richieste in attesa di essere servite nell'attuale direzione del movimento del braccio da quelle che saranno servite quando il braccio invertirà il movimento. Saranno pertanto necessarie, nel monitor, due variabili condizione sulle quali sospendere le richieste che non possono essere servite in quanto il disco è occupato. La prima comprenderà le richieste che saranno servite quando il disco invertirà la direzione corrente, la seconda comprenderà le richieste che saranno servite fino a quando il braccio del disco mantiene la stessa direzione. Entrambe le code di richieste sono ordinate in funzione della vicinanza alla posizione corrente del disco.

Definiamo ora alcune variabili:

- `occupato` per lo stato del disco (`true`, `false`);
- `posizione` per la posizione corrente del braccio del disco (numero della traccia occupata);
- `dest` per il numero della traccia richiesta;
- `direzione` per la direzione del movimento (`SU`, `GIU`);
- `dir_SU`, `dir_GIU` per le variabili condizione su cui sospendere le richieste.

Nell'ipotesi che il disco sia occupato (`occupato==true`) e che `direzione==SU`, in fase di richiesta si avrà:

- se `dest>posizione`, la richiesta viene sospesa (con l'indicazione delle traccia richiesta) nella coda di `dir_SU`;
- se `dest<posizione`, la richiesta viene sospesa (con l'indicazione del modulo rispetto a `N` della traccia richiesta se `N` è il numero delle traccie) nella coda di `dir_GIU`;
- se `dest==posizione`, la richiesta viene sospesa nella coda `dir_GIU` per evitare possibili condizioni di starvation per le altre richieste.

```
typedef int traccia;
typedef enum{SU,GIU}dir;
monitor movimento_braccio {
   traccia posizione=0;
   boolean occupato=false;
   dir direzione=SU;
   condition dir_SU,dir_GIU;
   public void richiesta(traccia dest) {
      if(occupato==true) {
         if(posizione<dest||
         (posizione==dest && direzione==GIU))
         wait (dir_SU,dest);
      else wait(dir_GIU,(N-dest)); }
   occupato=true;
   posizione=dest;
}
public void rilascio(){
   occupato=false;
   if (direzione==SU) {
      if(!empty(dir_SU))signal(dir_SU);
      else{
         direzione=GIU;
         signal(dir_GIU); }

   else if(!empty(dir_GIU))
            signal(dir_GIU);
         else{direzione=SU;
              signal(dir_SU); }
}
} /* fine monitor */
```

Figura 6.10 Gestione di un disco a teste mobili.

Nell'ipotesi di `direzione==SU`, in fase di rilascio si ha:

- se ci sono richieste nella coda `dir_SU`, viene servita la prima;
- se non ci sono richieste, viene invertita la direzione del braccio e servita la prima richiesta contenuta nella coda `dir_GIU`.

Il tutto è analogo nel caso di `direzione==GIU`. La struttura del monitor è riportata nella figura 6.10.

Problema dei lettori/scrittori Si supponga di voler realizzare la seguente politica di assegnazione della risorsa (vedi il terzo esempio del paragrafo 5.7.1):

1. un nuovo lettore non può acquisire la risorsa se c'è uno scrittore in attesa;
2. tutti i lettori sospesi al termine di una scrittura hanno priorità sul successivo scrittore.

```
nitor lettori_scrittori {
int num_lettori=0;
boolean occupato=false;
condition ok_lettura,ok_scrittura;
public void inizio_lettura {
   if(occupato || !empty (ok_scrittura))
      wait(ok_lettura);
   num_lettori++;
   signal(ok_lettura);
}
public void fine_lettura{
   num_lettori--;
   if(num_lettori=0)signal(ok_scrittura)
}
public void inizio_scrittura {
   if(num_lettori!=0 || occupato=true);
      wait(ok_scrittura);
   occupato=true
}
public void fine_scrittura {
   occupato= false;
   if(!empty(ok_lettura))signal(ok_lettura);
   else signal(ok_scrittura);
}
}
```

Figura 6.11 Problema dei lettori / scrittori.

Se definiamo le seguenti variabili:

- `num_lettori` per il numero di processi lettori attivi sulla risorsa;
- `occupato` è una variabile logica che indica se la risorsa è occupata (`occupato==true`);
- `ok_lettura` e `ok_scrittura` sono due variabili condizione sulle quali si sospendono rispettivamente i processi lettori e i processi scrittori;

la soluzione sarà quella riportata nella figura 6.11.

6.5 Chiamate innestate a procedure del monitor

Come si è visto precedentemente, il rilascio del monitor avviene quando un processo si sospende al suo interno o quando completa l'esecuzione di una operazione. È compito del processo garantire, in tali situazioni, la consistenza della struttura dati del monitor.

In talune applicazioni può essere necessario, mentre un processo `P` sta eseguendo l'operazione `A` di un monitor M_1, che venga chiamata la procedura `A'` di un altro monitor M_2 e che quindi il processo temporaneamente abbandoni il primo monitor.

Ci sono, a questo punto, due possibilità. La prima comporta che M_1 rimanga occupato, essendo `P` l'unico processo che ne ha il controllo esclusivo, anche quando

sta operando su M_2 (di cui pure ha, ovviamente, il controllo esclusivo). L'altra soluzione prevede invece che il controllo del primo monitor venga rilasciato al momento della chiamata dell'operazione del secondo monitor. Entrambe le soluzioni hanno aspetti positivi e negativi.

L'aspetto positivo della prima soluzione è che, non potendo nessun altro processo entrare nel monitor M_1 mentre il processo P sta operando sul secondo monitor, al termine dell'esecuzione della procedura A' di M_2, P ritorna a M_1 con i dati nello stesso stato in cui erano al momento della chiamata di A'.

L'aspetto negativo è legato al fatto che P può essere sospeso mentre sta eseguendo la procedura A' di M_2. In questo caso, il monitor M_2 viene liberato, ma M_1 resta bloccato. I processi che tentano di eseguire operazioni di tale monitor si bloccano all'ingresso esterno con due conseguenze: una perdita di parallelismo e la possibilità di situazioni di blocco critico. Questa seconda situazione si verifica se il risveglio del processo P dipende da un altro processo che deve eseguire anch'esso in modo analogo, lo stato, per esempio, la procedura B di M_1 e la B' di M_2.

Il problema non si presenterebbe se, adottando la seconda soluzione, il monitor M_1 venisse liberato come conseguenza di una richiesta di P di andare a eseguire una procedura di M_2. Questa soluzione prevede, in caso di sospensione di un processo entro un monitor, il rilascio di tutti i monitor implicati nella catena di chiamate di operazioni che hanno portato al monitor finale. Tale soluzione risulta di complessa attuazione per due motivi:

a) occorre garantire la consistenza della struttura dati di un monitor prima di effettuare una chiamata per un'operazione di un altro monitor. L'esecuzione di quest'ultima operazione può infatti, come si è visto, provocare la sospensione del processo, con conseguente liberazione del monitor chiamante.

b) quando un processo viene risvegliato deve riacquistare l'accesso esclusivo di tutti i monitor della catena, prima di riprendere l'esecuzione. Solo se nessuna delle variabili permanenti dei monitor è passata come parametro nelle chiamate di operazione, l'acquisizione del controllo esclusivo dei monitor può avvenire singolarmente al momento del ritorno al punto di chiamata di ogni monitor.

I problemi posti dalle due soluzioni potrebbero essere attenuati avendo la possibilità di eliminare il vincolo della mutua esclusione nell'esecuzione delle operazioni dei monitor. Si è già accennato al problema con riferimento all'esempio del produttore e consumatore che rappresenta un caso in cui le due operazioni di inserimento e di prelievo potrebbero agire in parallelo sul buffer (purché non nella stessa porzione del buffer) senza danni per la consistenza della struttura dati.

Inoltre vi sono casi in cui risulta semplice stabilire la relazione di consistenza prima di una chiamata a un'operazione di un altro monitor e quindi il monitor chiamante può essere liberato.

In definitiva, un costrutto simile al monitor, in cui sia data al programmatore la possibilità di specificare che alcune operazioni possono essere eseguite concorrentemente e in cui, per alcune chiamate, il monitor possa essere liberato, risulterebbe più flessibile da utilizzare, anche se aumenterebbe la responsabilità del programmatore nel garantire il corretto funzionamento del sistema.

Vedremo più avanti, introducendo la realizzazione del concetto di monitor in Java, che alcuni degli obiettivi indicati precedentemente sono raggiungibili.

alizzazione del monitor con Pthread e Java

afi precedenti si è dimostrato come il costrutto monitor, in un ambiente a comune, consenta di risolvere in modo intuitivo ed efficace problemi anessi di realizzazione di politiche di gestione di risorse comuni a più proseguito si mostrerà come sia possibile simulare il comportamento del mozzando dapprima la libreria pthread nell'ambito di un linguaggio sequencessivamente un linguaggio concorrente *object oriented* come Java.
enuna delle due soluzioni, verranno prima richiamati i costrutti messi a dipe per la sincronizzazione dei thread e quindi si esaminerà come tali costrutpossono essere combinati per riprodurre, per quanto è possibile, le proprietà del

hread

itolo 3 sono stati presentati gli strumenti che la libreria pthread mette a dine per la creazione e terminazione dei thread. Nel seguito, come si è detto, o presentati gli strumenti offerti da pthread per la risoluzione dei problemi di *izzazione indiretta* (mutua esclusione nell'accesso a risorse comuni) e *sinazione diretta* (realizzazione di politiche di gestione delle risorse comuni) ad, ovvero i semafori binari (*mutex*) e le variabili condizione (*condition*).

Il valore intero del semaforo `mutex` può essere zero o uno (semaforo bi-
Nel primo caso si dice che il `mutex` è occupato, nel secondo che è libero.
lla libreria pthread il `mutex` è definito del tipo `pthread_mutex_t` che imente rappresenta:

stato del `mutex`;
coda nella quale vengono sospesi i processi in attesa che il `mutex` sia libero.

finizione di un `mutex` M avviene tramite la `pthread_mutex_t` M. Una volinito, mediante l'inizializzazione si attribuisce un valore intero al suo stato (zeino). Ciò avviene tramite la funzione:

```
ead_mutex_init(pthread_mutex_t* M,
onst ptrhead_mutexattr_t *att)
```

M punta al `mutex` da inizializzare e `attr` punta a una struttura che contiene tributi del `mutex`. Se il valore di `attr` è NULL, il `mutex` viene inizializzato a o (che è il valore di default).
Su `mutex` sono possibili solo due operazioni: *lock* e *unlock*, concettualmente valenti alle P e V viste nel capitolo 5 e così definite:

```
read_mutex_lock (pthread_mutex_t * M);
head-mutex-unlock (pthread_mutex_t* M);
```

l caso della lock, se M è occupato (cioè il suo stato è zero), il thread chiamante si spende nella sua coda, altrimenti occupa M (cioè porta lo stato di M a zero).
Il comportamento della unlock è analogo a quello della V. L'effetto di questa priiva, infatti, dipende dallo stato della coda di processi associata a M; se vi sono ocessi in attesa in coda a M, ne viene risvegliato uno, altrimenti M viene liberato.

Il codice racchiuso tra le due primitive viene eseguito in maniera mutuamente esclusiva dai thread:

```
pthread_mutex M;
...
pthread_mutex_lock (&M);
   <sezione critica>;
ptrhead_mutex_unlock (&M);
```

Variabili condizione La variabile condizione nei pthread ha la stessa definizione già vista per il monitor; essa rappresenta una coda nella quale i thread si sospendono a seguito di una `wait` e vengono riattivati tramite una `signal`.

Una variabile condizione `C` viene creata e inizializzata con attributi nel modo seguente:

```
p_thread_cond_t C;
pthread_cond_init(&C, attr);
```

Il parametro `attr` è l'indirizzo della struttura che contiene eventuali attributi specificati per la variabile condizione (se il valore del parametro è `NULL`, gli attributi vengono inizializzati con i valori di default).

Le operazioni che possono essere eseguite sulle variabili condizione sono `wait`, `signal` e `broadcast` (simile a `signalAll`).

Analogamente a quanto visto per il monitor, la sospensione di un thread su una variabile condizione è conseguenza del non verificarsi di una condizione logica. Per esempio nel problema dei produttori/consumatori è necessario che un produttore si sospenda se il buffer dei messaggi è pieno.

Usando i ptrhead questa situazione si può esprimere associando alla condizione di buffer pieno una *condition*, come nell'esempio che segue:

```
/*variabili globali*/
pthread_cond_t C;
boolean bufferpieno=false;
.....
/*codice produttore: */
...
while bufferpieno <sospensione sulla condition C>;
<inserimento messaggio nel buffer>;
```

Va osservato che la verifica della condizione logica è una sezione critica: nell'esempio precedente, infatti, la variabile logica `bufferpieno` è condivisa tra tutti i produttori e consumatori e vi si deve accedere quindi in modo mutuamente esclusivo. Per questo motivo a ogni variabile condizione viene associato un `mutex` il cui ruolo è quello di garantire la mutua esclusione nell'accesso alla sezione critica per la verifica della condizione. L'esempio precedente deve quindi essere modificato introducendo un `mutex` e aggiungendo un prologo e un epilogo rispettivamente prima e dopo la verifica (ed eventuale sospensione del thread) della condizione per garantire la mutua esclusione nell'accesso a `bufferpieno`.

```
variabili globali: */
thread_cond_t C;
rthread_mutex_t M/*per mutua esclusione su condizione*/;
oolean bufferpieno=false;
....
codice produttore: */
thread_mutex_lock (&M);
hile (buffer pieno) <sospensione sulla condition C>;
nserimento messaggio nel buffer>;
thread_mutex_unlock(&M);
```

Tenendo conto di queste considerazioni la libreria pthread prevede implicitamente che, congiuntamente all'uso di una condition, venga sempre introdotto un `mutex`. Questo spiega perché la primitiva di sospensione richiede come parametro aggiuntivo (oltre alla variabile condizione su cui deve operare) una variabile di tipo `mutex`. In particolare la primitiva di sospensione è realizzata dalla funzione:

```
pthread_cond_wait(pthread_cond_t* C, pthread_mutex_t* M);
```

La chiamata di questa primitiva da parte di un thread `t` provoca due effetti: `t` viene sospeso nella coda associata a `C` e il `mutex M` viene liberato. Al successivo risveglio di `t` il thread rioccupa `mutex M` automaticamente.

Il risveglio di un thread sospeso su una variabile condizione può essere ottenuto mediante la funzione:

```
pthread-cond-signal (pthread-cond-t*C);
```

Gli effetti della chiamata sono i seguenti:

- se esistono thread sospesi nella coda associata a `C`, ne viene risvegliato il primo;
- se non vi sono thread sospesi sulla condizione, la `signal` non ha alcun effetto.

La politica realizzata dalla *signal* è del tipo *signal_and_continue*; in altri termini il thread che ha eseguito la *signal* prosegue la sua esecuzione mantenendo il controllo del `mutex` fino a esplicito rilascio.

Per esemplificare l'uso della condition si consideri il caso di una risorsa che può essere utilizzata contemporaneamente da, al massimo, `MAX` thread. Realizziamo una politica di controllo degli accessi mediante variabili condizione. A questo scopo introduciamo la condition `PIENO`, nella quale verranno sospesi i thread che vogliono accedere alla risorsa nel caso di capacità esaurita. Sia `M` il `mutex` associato alla condition `PIENO`. Introduciamo inoltre la variabile intera non negativa `N_in` per rappresentare lo stato della risorsa, cioè il numero di thread che stanno usando la risorsa. Riportiamo il thread risultante nella figura 6.12.

Simulazione del comportamento di un monitor Come è facilmente verificabile, gli strumenti di sincronizzazione offerti dalla libreria pthread consentono di realizzare politiche di assegnazione delle risorse ai thread analoghe a quelle ottenibili con il costrutto monitor (adottando la politica *signal_and_continue*).

```
Define MAX 100
/*variabili globali:*/
int N_in=0;/*numero thread che usano la risorsa*/
p_thread_cond_t PIENO;
p_thread_mutex_t M; /*mutex associate alla cond.PIENO*/
void codice_thread(){
   /*fase di entrata:*/
   p_thread_mutex_lock(&M);
   /*controlla la condizione di accesso:*/
   while(N_in==MAX) p_thread_cond_wait (&PIENO,&M);
   /*aggiorna lo stato della risorsa:*/
   N_in++;
   p_thread_mutex_unlock(&M);
   <uso della risorsa>
   /*fase di uscita:*/
   p_thread_mutex_lock(&M);
   /*aggiorna lo stato della risorsa:*/
   N_in-;
   p_thread_cond_signal(&PIENO);
   p_thread_mutex_unlock(&M);
}
```

Figura 6.12 Struttura di un thread che utilizza una risorsa a capacità limitata.

La mutua esclusione delle operazioni del monitor può essere ottenuta mediante una coppia di lock e unlock su un `mutex` all'inizio e al completamento di ogni operazione. La sospensione sulla variabile condizione si ottiene utilizzando `p_thread_cond_wait` applicata a tale variabile, la riattivazione dei thread sospesi si ottiene tramite la `p_thread_cond_signal`. Come si è detto la politica della *signal* è la *signal_and_continue*. È naturalmente possibile definire più variabili condizione.

La differenza fondamentale tra i due tipi di soluzione è l'assenza, nel caso pthread, di ogni controllo affidabile al compilatore. In particolare è compito del programmatore assicurare la mutua esclusione nell'accesso alle condizioni di sincronizzazione mediante l'utilizzo dei semafori `mutex` e delle lock e unlock.

6.6.2 Java

Java è un esempio di un linguaggio *object oriented*. Ogni tipo di interazione tra i thread avviene quindi tramite oggetti comuni. Nel capitolo 3 sono stati presentati gli strumenti che il linguaggio mette a disposizione per la creazione e la terminazione dei thread. Nel seguito verranno introdotti gli strumenti offerti da Java per la soluzione dei problemi di mutua esclusione e di realizzazione di politiche per l'accesso a oggetti comuni.

Object lock Per risolvere il problema della mutua esclusione in Java, a ogni oggetto, qualunque sia la classe cui appartiene, viene associato dalla Java Virtual Machine (JVM) un meccanismo di mutua esclusione, lock, analogo a un semaforo binario. Tale meccanismo è nascosto all'interno del supporto fornito dalla JVM e non è quindi direttamente visibile al programmatore.

È però possibile denotare alcune sezioni di codice, che operano su un oggetto come sezioni critiche, identificandole con la parola chiave `synchronized`. È poi compito del compilatore garantire che tali sezioni critiche siano eseguite in mutua esclusione inserendo in testa alla sezione critica un prologo, il cui scopo è garantire l'acquisizione del lock associato all'oggetto (se libero, altrimenti il thread che lo esegue viene sospeso), e un epilogo, per rilasciare il lock alla fine della sezione critica.

Per esempio, con riferimento a un oggetto `x`, è possibile definire un blocco di statement come una sezione critica nel seguente modo, noto col termine di blocco sincronizzato (*synchronized block*):

```
synchronized (Object x) <sequenza di statements>;
```

Nella figura 6.13 viene riportato l'esempio del metodo `M`, che più thread possono invocare ma che, al proprio interno, contiene un blocco sincronizzato (<sezione di codice critica>) che viene quindi eseguito in mutua esclusione.

Si noti che l'oggetto `mutexLock`, ottenuto come un'istanza della classe `Object` da cui tutte le classi Java sono derivate, viene usato esclusivamente (come indicato dal suo nome) per sfruttare il suo lock al fine di garantire che la sezione critica di codice all'interno del metodo `M` sia eseguita in mutua esclusione. In particolare, quando un thread invoca `M`, può eseguire la prima parte del metodo (<sezione di codice non critica>) senza nessun vincolo, anche in concorrenza con altri thread che a loro volta abbiano invocato `M`. Quando però un thread tenta di eseguire il blocco sincronizzato può proseguire solo se il lock associato a `mutexLock` è libero, altrimenti il thread viene sospeso dalla JVM in attesa che il lock si liberi.

Se il lock è libero il thread procede e occupa atomicamente il lock disabilitando altri thread a entrare a loro volta nella sezione critica. Quando il thread termina l'esecuzione del blocco sincronizzato, se non ci sono altri thread in attesa, il lock viene reso libero altrimenti la JVM sceglie uno dei thread in attesa abilitandolo a occupare nuovamente il lock e a eseguire, a sua volta, la sezione critica.

L'istruzione `synchronized(mutexLock)` serve perché un thread che esegue il metodo `M` acquisisca in modo esclusivo il lock associato all'oggetto `mutexLock` ed esegua quindi <sezione di codice critica> in modo esclusivo. Al completamento del blocco sincronizzato il lock viene rilasciato.

```
Object mutexLock=new Object;

public void M() {
   <sezione di codice non critica>;
   synchronized(mutexLock) {
      <sezione di codice critica>;
   }
   <sezione di codice non critica>;
}
```

Figura 6.13 Esempio di un blocco sincronizzato.

```
public class intVar{
private int i=0;
public void synchronized incrementa () {
   i++;}
public void synchronized decrementa () {
   i--;}
}
```

Figura 6.14 Metodi sincronizzati.

Il fatto che a ogni oggetto sia implicitamente associato un lock implica che a esso anche associato un insieme di thread (eventualmente vuoto) che, avendo tentato eseguire un blocco sincronizzato controllato dal lock di tale oggetto e avendolo tr vato occupato, sono stati sospesi in attesa che il lock venga liberato. Questo insie di thread viene anche indicato come *entry set*.

In Java esiste anche la possibilità di definire la mutua esclusione tra metodi del stessa classe. In questo caso è sufficiente che tali metodi siano caratterizzati dalla parola *sinchronized*.

Quando uno di tali metodi viene invocato per operare su un oggetto della classe l'esecuzione del metodo viene garantita in mutua esclusione sfruttando il *lock* de l'oggetto.

Si consideri l'esempio riportato nella figura 6.14.

La classe `intVar` definisce variabili intere sulle quali più thread possono eseguir le operazioni di `incrementa` e `decrementa` in maniera mutuamente esclusiva.

Riassumendo, la realizzazione della mutua esclusione in Java avviene nel s guente modo:

- ad ogni oggetto viene associato dalla JVM un lock (analogo a un semaforo bin rio) non visibile direttamente al programmatore;
- per accedere a un metodo *synchronized*, o a un blocco *synchronized*, un thre deve prima acquisire il lock dell'oggetto;
- il *lock* è automaticamente rilasciato quando il thread esce dal metodo *synchro zed*, o dal blocco *synchronized*, (o se viene interrotto da una eccezione);
- un thread che non riesce ad acquisire il lock rimane sospeso in una coda detta *try set* fino alla liberazione del lock. Più precisamente esso rimane nello st *runnable*. Appena il lock è rilasciato, lo scheduler può potenzialmente mett in esecuzione il thread;
- un metodo sincronizzato può chiamarne un altro sullo stesso oggetto senza bl carsi, al fine di evitare condizioni di blocco critico;
- due diversi metodi, uno sincronizzato e uno non sincronizzato, possono ess eseguiti concorrentemente sullo stesso oggetto.

Wait e notify Per realizzare le politiche di accesso alla risorsa controllata da u classe vengono utilizzate `wait`, `notify` e `notifyAll`, molto simili a wa `signal` e `signalAll` introdotte nel caso del monitor.

Java `wait`, `notify` e `notifyAll` sono metodi della classe `Object`. In analo a quanto visto per il monitor, anche nel caso di Java, `wait` e `notify` possono ere eseguite da un thread solo all'interno di un metodo sincronizzato o di un bloc sincronizzato (possesso del lock dell'oggetto).

La JVM associa implicitamente a ogni oggetto, oltre al meccanismo dei lock vi precedentemente, anche una coda di thread, inizialmente vuota, nota col nome di *it set*. I thread entrano ed escono da questa coda utilizzando i due metodi `wait` e `tify`.

Più precisamente:

l'esecuzione del metodo `wait()` risulta nelle seguenti azioni:
- il thread rilascia il lock dell'oggetto;
- lo stato del thread diventa *blocked*;
- il thread viene inserito nel *wait set* dell'oggetto;

l'esecuzione del metodo `notify()` risulta nelle seguenti azioni:
- se *wait set* relativo all'oggetto è vuoto, non viene eseguita alcuna azione; in caso contrario la JVM sceglie un thread `t`, estraendolo dal *wait set*, e lo inserisce nell'entry set in modo tale che, riattivando la sua esecuzione, questo possa riacquisire il lock e riprendere l'esecuzione dall'istruzione successiva alla `wait` con cui si era sospeso;
- `t`, dovendo riacquisire il lock per riprendere l'esecuzione, deve comunque attendere che il thread che ha invocato la `notify` rilasci a sua volta il lock che detiene (azione che avviene alla fine del blocco o metodo sincronizzato in cui si trova la `notify`). Inoltre, poiché la `notify` inserisce `t` nell'entry set, per motivi di competizione può accadere che all'atto del rilascio del lock questo venga acquisito da un thread t' prima che `t` riesca a sua volta ad acquisirlo.

politica adottata è quindi quella, già vista, nota come *signal_and_continue*: il ocesso segnalante non rilascia il lock e prosegue la sua esecuzione fino al comple mento del blocco o del metodo sincronizzato, al cui interno è stata eseguita la `no tify`.

La scelta del thread da mettere in esecuzione, mediante acquisizione del lock del ggetto, tra quelli sospesi su entry set, dipende dalla politica realizzata dalla JVM.

Per il metodo `wait` esiste la possibilità di specificare un tempo massimo da pas e nel wait set prima di essere risvegliato automaticamente.

La chiamata del metodo `notifyAll()` comporta l'estrazione di tutti i thread wait set e il loro inserimento in entry set.

In generale i thread risvegliati non ricevono alcuna informazione sul perché sono ti riattivati e quindi una volta entrati nel blocco, o nel metodo syncronized, prima eseguire la sezione critica dovranno ritestare la condizione di sincronizzazione o del `while`).

Con riferimento all'innestamento delle procedure del monitor, se un metodo o occo contiene la chiamata a un metodo di un altro oggetto, la politica adottata da va è quella di non rilasciare i lock del primo oggetto. Come visto, questa politica ulta più semplice da realizzare, ma può provocare condizioni di deadlock.

Vedremo nel seguito l'utilizzo degli strumenti *object-lock*, `wait`, `notify` e `tifyAll` per la risoluzione di alcuni problemi di sincronizzazione tra thread.

Problema produttori/consumatori Consideriamo dapprima il caso pi
plice di un solo produttore e di un solo consumatore che si scambiano mess
traverso un'area condivisa `buffer` in grado di contenere un solo messaggio
poniamo che il tipo di messaggi sia, per semplicità, di tipo intero. Nella figu
viene presentata la classe `Mailbox`, cui dovrà appartenere l'oggetto `buff`
utilizzare come area condivisa tra i due thread.

La variabile intera `contenuto` rappresenta l'area di memoria condivisa
l'indicatore booleano `pieno` serve per registrare lo stato di tale area contene
messaggio depositato dal produttore e non ancora prelevato dal consumatore (
`true`), oppure (valore `false`) quando ancora nessun messaggio è stato depo
o, se già depositato, è stato anche prelevato.

I metodi `preleva()` e `deposita()` sono *synchronized* in quanto op
su un oggetto condiviso.

La `wait` e la `notify` servono per la sospensione e la riattivazione dei proc

La struttura del programma avrebbe consentito anche l'uso di una istruzion
al posto del `while` per verificare la condizione di sincronizzazione.

Diversa è la situazione in cui un oggetto `buffer` della precedente class
condiviso tra molti processi produttori e consumatori. In questo caso diventa ne
sario, come vedremo, l'uso del `while` e del `notify-all`.

Infatti, in questi caso, nell'ipotesi che il `buffer` sia vuoto può accadere che
no contemporaneamente bloccati alcuni thread consumatori che, avendo invocat
metodo `preleva` e non essendoci niente da prelevare, si sono sospesi entrando
wait set di `buffer`.

```
public class Mailbox{
private int contenuto;
   private boolean pieno=false;
   public synchronized int preleva()
               throws InterruptedExceptions{
      try{
         while(pieno==false){ wait();}
         pieno=false;
         }
      finally { notify(); }
      return contenuto;
   }
   public synchronized void deposita (int valore)
               throws InterruptedExceptions{
      try{
         while (pieno==true){ wait();}
         contenuto= valore;
         pieno=true;
         }
      finally { notify(); }
   }
}
```

Figura 6.15 Mailbox unitaria con singolo produttore e singolo consumatore.

niamo anche che alcuni produttori siano sospesi nell'entry set in attesa riac-
il lock ed eseguire il metodo `deposita`.
ppena il primo tra questi acquisisce il lock, esegue `deposita` riempiendo il
er e con la `notify` sveglia uno dei consumatori prelevandolo dal wait set e
ndolo nel entryset. Quando tale produttore termina l'esecuzione di deposito ri-
il lock e quindi uno dei thread presenti nell'entry set viene abilitato a prose-
Se per caso viene scelto ancora un produttore, questi troverà il buffer pieno e
si sospenderà. Da questo momento in poi abbiamo nel wait set sia thread con-
tori sospesi perché hanno trovato il buffer vuoto, sia produttori che hanno tro-
il buffer pieno. Quando un consumatore riuscirà ad acquisire il lock potrà prele-
il messaggio contenuto nel buffer ed eseguire la `notify` per risvegliare un
uttore. Però, come si è visto, la `notify` sceglie in maniera casuale uno dei
d presenti nel wait set e allora può accadere che, invece di un produttore, venga
o un consumatore che, evidentemente, non può essere in grado di proseguire es-
o vuoto il `buffer`.

```
ic class Mailbox{
rivate int[] contenuto;
rivate int contatore,testa,coda;
ublic Mailbox() {
 contenuto=new int[N];
 contatore=0;
 testa=0;
 coda=0;
}

ublic synchronized int preleva()throws
            InterruptedExceptions{
 int elemento;
 try{
    while (contatore==0) {wait();}
    elemento=contenuto[testa];
    testa=(testa+1)%N;
    --contatore;
    }
 finally{ notifyAll(); }
 return elemento;
}

public synchronized void deposita (int valore)
            throws InterruptedExceptions {
 try{
    while (contatore==N) {wait();}
    contenuto[coda] = valore;
    coda=(coda+1)%N;
    ++contatore;
    }
 finally{ notifyAll(); }
}
```

gura 6.16 Buffer circolare.

In questo caso, per garantire la correttezza della soluzione, è quindi necessario so
tuire la `notify` con la `notifyAll` che risvegli tutti i thread presenti nel wait
Questi, uno alla volta, riacquisteranno il lock, ma necessariamente dovranno riva
tare la condizione per verificare se possono proseguire o se debbono sospender
nuovo. È questo il motivo per cui è necessario inserire la `wait` all'interno di un
clo nel quale il thread rimanga fino a quando non trovi la condizione falsa.

Nella figura 6.16 l'esempio viene generalizzato supponendo che l'area condi
sia in grado di contenere non uno soltanto, ma fino a N messaggi contemporaneam
te. In questo caso la variabile `contenuto` diventa un `array` di N elementi i q
vengono gestiti con tecnica FIFO mediante gli indici `testa` e `coda`. Inoltre vi
aggiunta la variabile `contatore` che è destinata a contenere il numero di elem
pieni dell'`array`. La condizione buffer vuoto coincide, in questo caso, con `cont
tore==0` e, analogamente, la condizione buffer pieno con `contatore==N`.

Variabili condizione Nella soluzione al problema della mailbox mediante buf
circolare ottenuta con il costrutto monitor, si era fatto uso delle *variabili condizio*
per la sospensione dei thread per i quali la condizione logica non era verificata.
esattamente erano state introdotte due variabili condizione `non_pieno` e non
`vuoto` su cui sospendere, rispettivamente, consumatori e produttori.

In generale, le variabili condizione, introdotte anche nei pthread, semplificano
modo significativo la soluzione di problemi di sincronizzazione consentendo di s
parare i processi sospesi all'interno del monitor in funzione del tipo di condizio
logica attesa e facilitando quindi la realizzazione di politiche di risveglio.

Si noti che, in entrambi i casi, l'analisi della condizione di sincronizzazione
parte dei thread deve avvenire in mutua esclusione. Ciò è ottenuto, nel caso del m
nitor, associando un semaforo di mutua esclusione all'oggetto condiviso e garant
do, a tempo di compilazione, che utilizzando tale semaforo le sue operazioni di t
`public` operino in mutua esclusione.

Nel caso dei pthread, invece, l'analisi in mutua esclusione delle condizioni
sincronizzazione è lasciata a carico del programmatore che deve introdurre un sem
foro `mutex` di mutua esclusione e operare su di esso con le operazioni lock e
lock.

Le soluzioni precedentemente presentate utilizzando il linguaggio Java fa
uso di una sola variabile condizione *definita implicitamente* sulla quale operan
operazioni `wait`, `notify` e `notifyAll` e realizzata tramite la coda wait se
maniera analoga al caso del monitor, è associato a ogni oggetto condiviso un lo
la garanzia che l'analisi delle condizioni di sincronizzazione avvenga in mu
esclusione è assicurata a tempo di compilazione dai metodi *synchronized*.

Nelle versioni più recenti di Java (*Java2 PlatformStandard Ed. 5.0*) esist
possibilità di dichiarare e utilizzare in modo esplicito più variabili condizioni. Ci
ottenibile tramite l'uso di apposite *interfacce*[1] del tipo:

[1] Nelle interfacce Condition e Lock, così come definite in Java.util.concurrent.locks [64], s
presenti altri metodi che qui non vengono riportati per semplicità.

```
blic class Mailbox{
private int[] contenuto;
private int contatore,testa,coda;
private Lock lock=new ReentrantLock();
private Condition non_pieno=lock.newCondition();
private Condition non_vuoto=lock.newCondition();
public Mailbox() {
  contenuto=new int[N];
  contatore=0;
  testa=0;
  coda=0;
}
public int preleva ()throws InterruptedException{
  int elemento;
  lock.lock();
  try{
     while(contatore==0) non_vuoto.await();
     elemento=contenuto[testa];
     testa=(testa+1) %N;
     --contatore;
     non_pieno.signal();
     }
  finally{lock.unlock();}
  return element;
}
public void deposita(int valore)throws
          InterruptedException {
  lock.lock();
  try{
     while(contatore==N)non_pieno.wait();
     contenuto[coda]=valore;
     coda=(coda+1) %N;
     ++contatore;
     non_vuoto.signal();
     }
  finally {
  lock.unlock();}
}
```

ura 6.17 Buffer circolare con variabili condizione.

```
lic interface Condition{
//Public instance methods
void await()throws InterruptedException;
void signal();
void signalAll();
}
```

e i metodi `await`, `signal`, e `signalAll` sono del tutto equivalenti ai metodi
it, `notify` e `notifyAll`. Un'altra interfaccia è la seguente:

```
public interface Lock{
   //Public instance methods
   void lock();
   void unlock();
   Condition newCondition();
}
```

Per creare un oggetto `Condition` (che implementa la `Condition` inte
si opera nel seguente modo:

```
Lock lockvar=new Reentrantlock;[2]
Condition condvar= lockvar.newCondition();
```

La variabile condizione così creata risulta collegata al lock proprio dell
`lockvar`; in altri termini le operazioni `await`, `signal` e `signalAll` fa
ferimento a tale lock.

È possibile naturalmente dichiarare più variabili condizione entro lo ste
getto della classe `Lock` raggiungendo così l'obiettivo che tutte facciano rife
allo stesso lock.

Come nel caso dei pthread, rimane a carico del programmatore l'utilizzo d
todi `lock` e `unlock` dell'oggetto `lockvar` per garantire la mutua esclusion
l'analisi della condizione di sincronizzazione.

Utilizzando due variabili condizione, `non_pieno` e `non_vuoto`, la soli
al problema del buffer circolare viene riportata nella figura 6.17.

6.7 Sommario

Viene innanzitutto sottolineato come l'utilizzo dei semafori, per la soluzione de
blemi di comunicazione e sincronizzazione tra processi, possa presentare una se
inconvenienti legati alle difficoltà insite in un loro uso corretto, soprattutto per p
mi di una certa complessità, e alla impossibilità di evidenziare errori nella loro
zazione a tempo di compilazione. Proprio per superare questi problemi viene int
to il costrutto monitor, soffermandosi dapprima sulla sua definizione sintattica
mantica. Vengono successivamente introdotte le variabili condizioni e le oper
`wait` e `signal` che operano su di esse per realizzare politiche di sincronizzazi

Particolare attenzione è posta nella definizione delle diverse semantiche del
mitiva `signal` che hanno un notevole impatto sulle modalità di realizzazion
monitor in termini di semafori. In particolare vengono discusse le sema
signal_and_continue, *signal_and_wait*, *signal_and_urgent wait* e *signa
d_return*. Per ciascuna di esse viene presentata la realizzazione delle pri
`wait` e `signal` in termini di semafori e delle istruzioni necessarie per acc
(prologo) e per rilasciare (epilogo) il monitor.

[2] `Reentrantlock` è una classe che implementa l'interfaccia `Lock`, con l'aggiunta di m
per determinare quale thread è in possesso del lock, per determinare quanti thread sono in
del lock e sono sospesi su una condizione associata e per testare se un thread specificato è i
sa di acquisire un lock. Il termine "reentrant" deriva dalla possibilità per un thread che po
il lock di chiamare un altro metodo con lock senza bloccarsi.

ere in evidenza la capacità espressiva del monitor, vengono presentati alcuni
sia nel caso in cui il monitor sia utilizzato come gestore della risorsa, sia nel
e esso venga utilizzato semplicemente come allocatore della risorsa. Con ri-
to a quest'ultimo caso, viene presentata la soluzione di alcuni classici proble-
alizzazione di politiche di gestione delle risorse, alcuni dei quali già trattati
mafori.

ne viene affrontato il problema della simulazione del comportamento del mo-
ilizzando le primitive di sincronizzazione proprie della libreria pthread e del
gio Java.

ntrambi i casi vengono discusse le proprietà di tali primitive e presentati alcu-
pi di un loro utilizzo.

Note bibliografiche

alisi critica dello strumento semaforo per la soluzione di problemi di sincro-
ione complessi la si può trovare nei classici lavori di Hoare [12] [46] e di
Hansen [24].

costrutto monitor è stato introdotto per la prima volta da Brinch Hansen [24] e
lizzato da Hoare [49]. Hoare utilizza, nella realizzazione del monitor, la politi-
nal_and_urgent wait.

i linguaggi concorrenti sviluppati in ambiente a memoria comune va ricorda-
oncurrent Pascal per il suo valore didattico e per essere stato il primo a incor-
i costrutti monitor, processi e classi. Un'esauriente trattazione del Concurrent
si trova in Brinch Hansen [22] [55]. Sempre a Brinch Hansen è dovuta la trat-
ne del linguaggio Edison [33], da utilizzarsi in ambienti distribuiti per applica-
in tempo reale.

aggior sviluppo industriale ha avuto il linguaggio Modula proposto da N.
[56] [57], soprattutto nelle versioni Modula-2 [58] e Modula-3, che fanno uso
eccanismo delle *coroutine*.

ltrettanta importanza, per le sue applicazioni, ha avuto il linguaggio Mesa svi-
to presso la Xerox PARC [59], usato in particolare come linguaggio per lo svi-
di sistemi. In [60] è riportato una interessante esperienza di uso del monitor e
rocessi con Mesa.

ono stati sviluppati molti altri linguaggi basati sull'utilizzo diretto del monitor;
uesti va ricordato Concurrent Euclid [61] utilizzato per la realizzazione di un nu-
UNIX-compatibile. Infine va ricordato Pascal-plus [62] che possiede molte ca-
istiche comuni al Concurrent Pascal e al Modula ed è stato progettato per la pro-
mazione modulare di applicazioni concorrenti, tipicamente sistemi operativi.

er quanto riguarda l'utilizzo della libreria pthread si può fare riferimento alla
rizione della libreria LinuxThreads [63]. Per quanto riguarda un buon tutorial su
a si può fare riferimento a [37]. In [38] si trova una trattazione della concorrenza
ava. In [64] si può trovare la definizione delle variabili condizione e dei lock.

7

Modello a scambio di messaggi

Un modello architetturale di macchina concorrente del tutto diverso da quello a cui abbiamo fatto riferimento nei precedenti capitoli è quello noto col nome di *modello a scambio di messaggi*. In questo tipo di architetture, note anche come *architetture a memoria distribuita*, non è prevista nessuna memoria comune ai vari processori, ciascuno dei quali può accedere esclusivamente alla propria memoria locale (vedi figura 2.12). Come indicato nel paragrafo 2.4, in questi casi si usa spesso parlare di *programmazione distribuita*, invece che di programmazione concorrente, proprio per mettere in evidenza che l'elaborazione complessiva è composta da un insieme di attività sequenziali che vengono svolte concorrentemente da processori distribuiti e collegati tramite una rete di comunicazione. Al solito, il modello si riferisce alla macchina virtuale, indipendentemente dalla reale architetttura della macchina fisica che fornisce il supporto alla macchina virtuale stessa. Obiettivo di questo capitolo è quello di mettere in evidenza le principali caratteristiche di questo modello architetturale al fine di caratterizzare i meccanismi per la specifica delle interazioni tra processi, tanto quelli primitivi offerti dalla macchina astratta quanto quelli linguistici che vengono offerti al programmatore dai linguaggi che seguono questo modello.

7.1 Aspetti caratterizzanti il modello

Come vedremo, i vincoli architetturali imposti dal modello hanno notevoli implicazioni sulle modalità di gestione delle risorse, sui meccanismi utilizzati per garantire le interazioni tra processi e, in ultima analisi, sui paradigmi adottati per programmare tali interazioni.

Il fatto che non esista alcuna memoria comune ai vari processori virtuali e, quindi, che ogni processo possa operare esclusivamente sulle risorse allocate nella propria memoria locale, impone di rivedere le tecniche di allocazione delle risorse rispetto a quelle descritte relativamente al modello a memoria comune e sinteticamente riportate nella figura 4.2. In particolare, ogni processo può accedere esclusivamente alla propria memoria locale e quindi non esiste nessuna possibilità di condivi-

sione di risorse. Ogni risorsa è privata in quanto allocata nella memoria locale di un processo, che è quindi l'unico a poterci operare. Non ha senso, perciò, parlare di risorse comuni e quindi di allocazione dinamica di risorse o di risorse condivise.

Questa considerazione comporta, come conseguenza, l'impossibilità, per processi diversi, di accedere a risorse comuni, eliminando quindi ogni problema di mutua esclusione negli accessi a una risorsa in quanto accessibile sempre a un unico processo. Ciò potrebbe indurre a ritenere che una delle forme di interazione introdotte nel secondo capitolo, la competizione, non sia più possibile in questo tipo di architetture. In realtà, come vedremo, le cose non stanno così. La competizione è stata infatti introdotta come una necessità legata alla limitatezza del numero di risorse. Abbiamo fatto, come esempio, il caso di un'applicazione dove più processi hanno la necessità di stampare dei dati durante le loro esecuzioni. Sarebbe assurdo che, per risolvere questo problema, fossero necessarie tante stampanti quanti sono i processi. Problemi di questo tipo si presentano però anche su architetture distribuite e anche in questo caso, è necessario prevedere che una risorsa, la stampante nel nostro esempio, possa fornire il proprio servizio a più processi anche se esiste il vincolo architetturale che un solo processo può operare direttamente sulla risorsa. La soluzione, in casi come questo, è quella di programmare il solo processo che può accedere alla risorsa in modo tale che questo operi sulla stessa per conto di altri. Tutte le volte che ciò si verifica, parleremo di processi clienti, riferendoci a quelli che hanno la necessità di usufruire della risorsa, e di processo servitore (*server*), quello che opera direttamente sulla risorsa per conto dei clienti. I processi clienti, utilizzando il meccanismo di comunicazione offerto dalla rete di interconnessione, inviano richieste di servizio al processo server il quale ha il compito di riceverle e, adottando opportune strategie di gestione, di servire tali richieste operando sulla risorsa. Se il servizio prevede la restituzione di risultati, il processo server, sempre mediante la rete di comunicazione, li invia al processo cliente di cui ha servito la richiesta. La competizione è quindi ancora presente anche se diverso è il criterio con cui viene espletata. In pratica è compito del server garantire la corretta competizione degli accessi alla risorsa effettuati a favore dei processi clienti. Come anticipato nel capitolo 4, in questo caso è il processo server che svolge il compito di gestore della risorsa.

Cambia quindi il paradigma di programmazione utilizzato per consentire a un processo di usufruire dei servizi di una risorsa *remota*, cioè di una risorsa che non è allocata nella propria memoria locale. Non potendo agirci direttamente invocando una delle operazioni previste dal tipo della risorsa, il processo dovrà interagire col server della risorsa stessa utilizzando l'unico meccanismo disponibile per le interazioni tra processi, cioè il meccanismo di comunicazione. È questo il meccanismo di base utilizzato dai processi per qualunque tipo di interazione. Esso fornisce infatti il supporto per lo scambio di dati necessario per qualunque tipo di cooperazione. Ma, come abbiamo detto, viene utilizzato anche come supporto per la competizione. In questo caso le informazioni che vengono scambiate tramite il meccanismo sono le richieste di servizio tra un cliente e il server e gli eventuali risultati tra il server e il cliente.

7.2 Canali di comunicazione

Il meccanismo che maggiormente caratterizza le modalità con cui programmare le interazioni tra processi, in un'applicazione concorrente, nel modello a memoria comune è sicuramente il meccanismo di sincronizzazione. Il suo uso è necessario per coordinare gli accessi a risorse comuni al fine di garantire una corretta competizione sulle stesse e, nel caso in cui queste vengano utilizzate come supporto allo scambio di informazioni, per garantire la corretta cooperazione. Per questo motivo, nei precedenti capitoli è stata posta particolare enfasi alla definizione sia del meccanismo primitivo di sincronizzazione offerto dalla macchina concorrente, come per esempio il meccanismo semaforico, sia di quello offerto da linguaggi di alto livello, come il costrutto monitor, illustrando in particolare, in entrambi i casi, l'interfaccia di programmazione utilizzata per risolvere qualunque problema di interazione tra processi. Per lo stesso motivo, porremo la stessa enfasi nel definire l'interfaccia dei processi verso la rete di comunicazione che, nel modello a scambio di messaggi, rappresenta l'unico strumento disponibile per la programmazione di interazioni tra i processi di un'applicazione distribuita.

Il concetto fondamentale che entra in gioco in un contesto distribuito è quello di canale, inteso genericamente come collegamento logico mediante il quale due processi comunicano. È compito del nucleo del sistema operativo fornire l'astrazione *canale* come meccanismo primitivo per lo scambio di informazioni. Tale meccanismo rappresenta l'astrazione della rete di comunicazione che collega i vari processori della macchina fisica, nel caso di architetture fisicamente distribuite, o un'astrazione realizzata tramite la memoria comune, se l'architettura fisica è multiprocessore o monoprocessore. Sarà inoltre compito di un linguaggio di programmazione, che segue il modello a scambio di messaggi, offrire gli strumenti linguistici di alto livello per consentire al programmatore di specificare i canali di comunicazione e di utilizzarli per programmare le varie interazioni tra i processi dell'applicazione.

Negli ultimi anni il numero dei meccanismi di comunicazione, sia di quelli linguistici che di quelli primitivi, è andato progressivamente crescendo anche in relazione allo sviluppo di ambienti di programmazione per applicazioni in sistemi distribuiti. Le principali differenze tra tali meccanismi riguardano alcuni parametri che caratterizzano il concetto di canale e ne determinano la semantica. Tra questi sono particolarmente importanti:

a) la tipologia del canale, intesa come direzione del flusso dei dati che un canale può trasferire;
b) la designazione del canale e dei processi sorgente e destinatario di ogni comunicazione;
c) il tipo di sincronizzazione fra i processi comunicanti.

Con riferimento al parametro a), si possono avere due diverse tipologie di canali: quelli *monodirezionali*, che prevedono il flusso di dati in una sola direzione, dal processo mittente al ricevente, e quelli che, viceversa, consentono un flusso di dati *bidirezionale*. In quest'ultimo caso un processo può utilizzare un canale sia per inviare che per ricevere informazioni. Come vedremo, questo tipo di canali è particolarmente adatto per collegare un processo cliente a un processo server. L'interazione

tra questi due processi richiede, infatti, sia il trasferimento dei dati relativi alla richiesta di servizio dal cliente al server sia, in generale, il trasferimento dei risultati dal server al cliente.

Rispetto alla designazione del canale e dei processi, che possono utilizzarlo per inviare e/o ricevere dati (parametro b)), si possono avere tre diversi casi. Il caso più semplice è quello di un canale utilizzato da due soli processi, in particolare, se il canale è monodirezionale, da un solo processo mittente e da un solo ricevente. In questo caso si parla di comunicazione *simmetrica* e *link* è il termine più spesso utilizzato per denotare questo tipo di canali. Si parla invece di comunicazione *asimmetrica* nel caso in cui il canale abbia tanti processi mittenti e un solo ricevente e *port* è il termine utilizzato in questi casi per denotare il canale. Mentre una comunicazione simmetrica è tipica della connessione tra due processi tipo produttore/consumatore, in cui cioè uno dei due processi fornisce all'altro i dati che quest'ultimo dovrà elaborare (vedi figura 7.1), la comunicazione asimmetrica è più adatta a una relazione tra processi del tipo cliente/servitore. È naturale infatti che un server sia predisposto ad accogliere su uno stesso canale richieste di servizio provenienti da più clienti diversi (vedi figura 7.2).

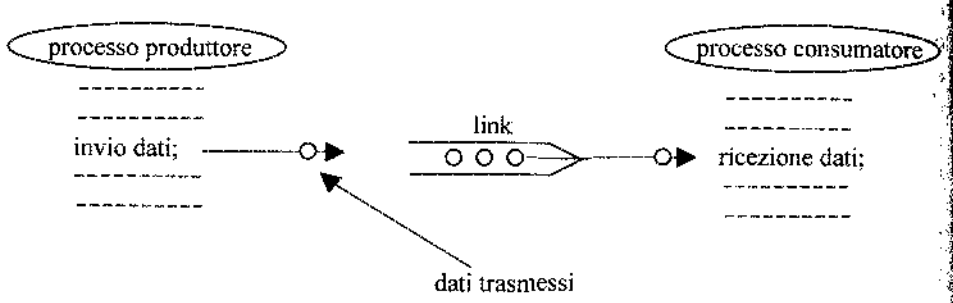


Figura 7.1 Comunicazione simmetrica produttore/consumatore.

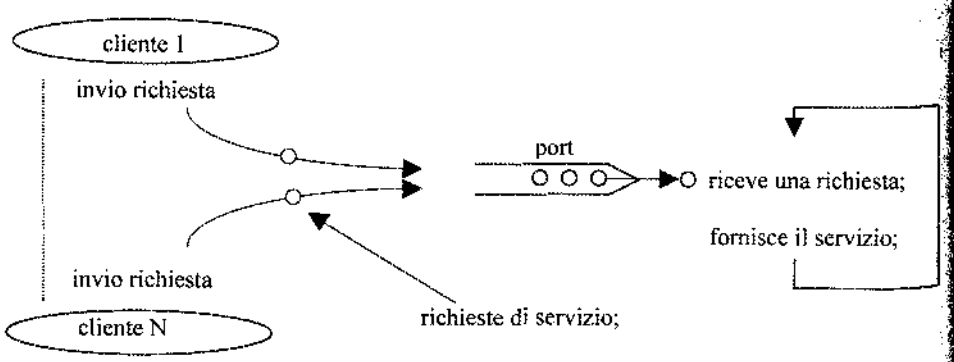


Figura 7.2 Comunicazione asimmetrica cliente/servitore.

Il caso più generale è, infine, quello di un canale su cui più processi mittenti possono inviare dati e più processi riceventi possono riceverli. Questo tipo di canale, che indicheremo come canale *da molti a molti*, è noto anche col termine di *mailbox*. Nei precedenti capitoli abbiamo visto molti esempi di possibili implementazioni di questo concetto in un ambiente che segue il modello a memoria comune. L'utilità di un canale di questo tipo si manifesta tutte le volte che un messaggio può essere ricevuto da uno qualunque fra un insieme di processi riceventi e corrisponde al caso in cui, in sistemi a memoria comune, siano disponibili più risorse equivalenti. Comunque, nei sistemi che seguono il modello a scambio di messaggi, raramente questo tipo di canale viene offerto come strumento primitivo di comunicazione, sia a livello di macchina concorrente sia a livello di linguaggio. Vedremo infatti nei successivi capitoli come implementare una mailbox partendo da canali simmetrici (link) o asimmetrici (port).

Infine, per quanto riguarda il parametro c), cioè il tipo di sincronizzazione che l'uso di un canale induce fra i processi che tramite questo comunicano, abbiamo ancora tre diverse tipologie, tutte legate alle diverse semantiche dell'operazione utilizzata per inviare messaggi. Mentre, infatti, l'operazione di ricezione di un messaggio costituisce sempre un punto di sincronizzazione, la stessa cosa non è necessariamente vera per l'operazione di invio. Che l'operazione di ricezione di un messaggio implichi una sincronizzazione deriva dall'ovvia conseguenza che un messaggio per essere ricevuto deve essere stato già inviato. Quindi il processo ricevente deve bloccarsi se all'atto della ricezione sul canale non sono presenti messaggi già inviati, sincronizzandosi così con l'invio di un messaggio. Viceversa, relativamente all'operazione di invio, abbiamo tre diverse possibilità, note come:

a) comunicazione asincrona;
b) comunicazione sincrona;
c) comunicazione con sincronizzazione estesa (detta anche *a rendez-vous*).

Il tipo a) prevede che il processo mittente continui la sua esecuzione immediatamente dopo che il messaggio è stato inviato. Il processo mittente è quindi del tutto asincrono col ricevente, proseguendo nella sua esecuzione dopo l'invio indipendentemente da "se e quando" il messaggio sarà ricevuto. Questo tipo di comportamento dell'operazione di invio rappresenta sicuramente quello più semplice ed efficiente poiché, non essendo un punto di sincronizzazione, non implica mai una sospensione del processo mittente e quindi tende a limitare il numero di commutazioni di contesto. Per contro, è anche l'operazione di livello più basso rispetto alle altre alternative che, come avremo modo di vedere, comporta soluzioni più complesse ad alcuni problemi di interazione tra processi. Ciò in particolare si verifica tutte le volte che è necessario imporre una sincronizzazione esplicita tra i processi comunicanti, sincronizzazione che, non essendo associata all'operazione di invio, dovrà essere esplicitamente programmata tramite uno scambio di messaggi col solo scopo di scambiare segnali di sincronizzazione.

Una caratteristica peculiare di questa alternativa è quella di fare riferimento a canali che abbiano a loro disposizione una memoria tampone (*buffer*) nella quale possano trovare spazio i messaggi già inviati sul canale ma non ancora ricevuti. Inoltre, poiché un processo mittente non si blocca mai durante l'invio, è necessario che la

capacità di tale buffer sia concettualmente illimitata. Non è infatti previsto, come nel caso di un buffer limitato, che il processo mittente si blocchi quando il buffer è pieno. In pratica, non essendo possibile la realizzazione di una memoria infinita, ciò significa semplicemente che l'operazione di invio può anche terminare in modo anomalo, generando un'eccezione quando la capacità del buffer viene superata.

La comunicazione sincrona, tipo b), prevede invece che l'operazione di invio costituisca un punto di sincronizzazione. In pratica, il processo mittente si sospende in attesa che il messaggio inviato sia stato ricevuto. Poiché, come abbiamo visto precedentemente, l'operazione di ricezione costituisce sempre un punto di sincronizzazione, in questo tipo di comunicazione il primo, fra due processi comunicanti, che esegue o l'invio o la ricezione si sospende in attesa che l'altro sia pronto a eseguire l'operazione duale, rispettivamente, di ricezione o di invio. Come si può intuire, in questo caso non esiste concettualmente la necessità di prevedere un buffer associato al canale in quanto un messaggio può essere inviato soltanto quando il ricevente è pronto a riceverlo. Rispetto alla comunicazione asincrona è adesso più semplice programmare interazioni che prevedano la necessità di imporre dei vincoli di sincronizzazione tra i processi in quanto già impliciti nell'operazione di invio. Per altro, questo meccanismo è in generale meno efficiente del precedente poiché tende a limitare il parallelismo del programma e quindi ad aumentare il numero di commutazioni di contesto. Come vedremo, utilizzando questo meccanismo, se vogliamo disaccoppiare due processi comunicanti è necessario interporre tra di loro un terzo processo che gestisca esplicitamente una risorsa di bufferizzazione. Infine possiamo concludere che tale meccanismo è sicuramente meno primitivo del precedente nel senso che, disponendo di un meccanismo asincrono, è molto facile implementare il meccanismo sincrono mediante un semplice protocollo che preveda che il processo mittente, dopo un invio, si ponga in attesa di ricevere un messaggio (*acknowledge*) che denoti l'avvenuta ricezione da parte del processo ricevente (vedi figura 7.3 dove il messaggio *ak* viene utilizzato per questo scopo).

L'ultima alternativa, tipo c), relativa a comunicazioni con sincronizzazione estesa, rappresenta il meccanismo di livello più alto fra i tre esaminati. Tale meccanismo è stato proposto per semplificare la programmazione di interazioni di tipo cliente/servitore, in particolare in congiunzione con l'uso di canali bidirezionali, e prevede

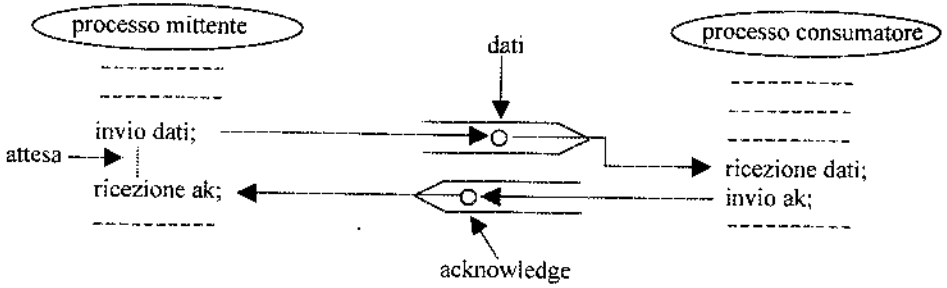


Figura 7.3 Implementazione di un meccanismo sincrono mediante comunicazioni asincrone.

che il processo mittente (il cliente di un servizio) si sincronizzi, come nel caso precedente, rimanendo in attesa che l'altro processo, il server, riceva la richiesta di servizio. In questo caso però il processo mittente prolunga la sua sospensione fino a quando il servizio richiesto non è stato completamente eseguito da parte del server e gli eventuali risultati non sono stati restituiti al mittente. Da qui il significato dell'aggettivo "estesa" con cui viene denotato il tipo di sincronizzazione di questo meccanismo. Questo schema, come indicato nella figura 7.4, prevede che il processo mittente, una volta inviata la richiesta di servizio, resti in attesa dei risultati.

Questo meccanismo è noto anche col nome di *chiamata di operazione remota*. Esiste, infatti, un'analogia semantica col meccanismo relativo alle chiamate di funzioni. Come nel caso di una chiamata di funzione, il programma chiamante continua solo dopo che l'esecuzione della funzione è terminata. La differenza sostanziale risiede nel fatto che la funzione (il servizio) viene eseguita remotamente da un processo diverso dal chiamante. A questa analogia semantica, come vedremo, corrisponde spesso anche un'analogia sintattica che consente di utilizzare questo meccanismo seguendo un paradigma di programmazione analogo a quello visto nel modello a memoria comune. Come già indicato, al fine di associare il messaggio contenente i risultati con il corrispondente messaggio di richiesta si ricorre spesso, in questo caso, a canali bidirezionali (vedi figura 7.4).

Come nel precedente caso, anche adesso l'invio di una richiesta, corrispondente a un chiamata remota, costituisce un punto di sincronizzazione ma, adesso, la riduzione del parallelismo rispetto a una comunicazione asincrona è ancora maggiore poiché maggiore è l'intervallo temporale durante il quale il processo cliente rimane sospeso. Essendo questo meccanismo particolarmente adatto a risolvere problemi di tipo cliente/servitore, le conseguenze di questa perdita di parallelismo si manifestano soprattutto nel risolvere problemi che comportano semplici scambi di dati.

Dalla figura 7.4 si può anche desumere come sia semplice simulare questo tipo di meccanismo mediante i più semplici meccanismi di comunicazione sincrona o asincrona.

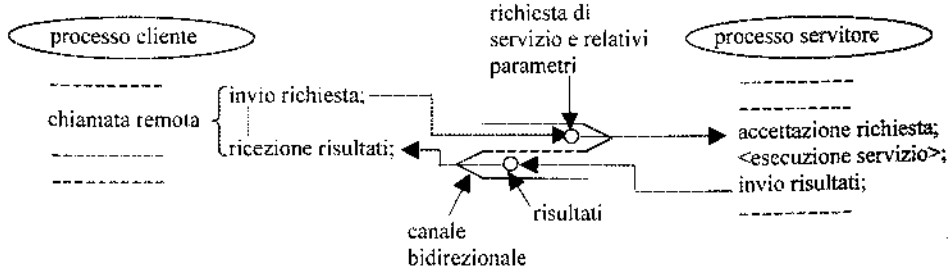


Figura 7.4 Chiamata di operazione remota.

7.3 Primitive di comunicazione

Come risulta da quanto esposto nel precedente paragrafo, molte sono le possibili varianti del meccanismo di comunicazione utilizzato dai processi per interagire nell'ambito di un'applicazione distribuita. Nei prossimi tre capitoli verranno esaminati i tre casi più spesso utilizzati che corrispondono ai tre meccanismi di comunicazione asincrona, sincrona e di chiamata di operazione remota. Nei primi due casi vengono utilizzati canali monodirezionali, nel terzo, come già visto, canali bidirezionali.

Definiamo adesso le primitive linguistiche a cui faremo riferimento nel seguito per ciascuno dei precedenti tre casi, introducendo anche le notazioni sintattiche che verranno utilizzate negli esempi illustrati nei successivi capitoli. In particolare, fra le molte notazioni che sono state proposte, faremo riferimento a delle notazioni che siano particolarmente semplici da usare.

La notazione sintattica che utilizzeremo per denotare le primitive asincrone e sincrone è la stessa per i due casi anche se, ovviamente, le primitive di invio di un messaggio sono semanticamente diverse fra loro. Come abbiamo già detto, i canali in questi due casi sono sempre monodirezionali ma possono essere simmetrici o asimmetrici. Noi faremo riferimento, come esempio, al caso di canali asimmetrici (ovvero sullo stesso canale possono inviare messaggi più processi mittenti ma un solo ricevente è abilitato a ricevere) che, come già visto, verranno indicati col termine di `port`. Per definire un canale simmetrico, in questo caso sarà sufficiente, come caso particolare, limitare a uno il numero dei processi mittenti.

Tenendo conto che parliamo di primitive linguistiche, nel dichiarare un canale definiremo anche il tipo di messaggi che, tramite questo, possono essere scambiati in modo tale da abilitare il controllo statico dei tipi da parte del compilatore. Adotteremo quindi la seguente notazione per dichiarare un canale:

```
port <tipo> <identificatore>;
```

dove <tipo> identifica il tipo dei messaggi che tramite il canale possono essere trasmessi e <identificatore> denota il nome con cui il canale verrà identificato. Per esempio:

```
port int ch1;
```

dichiara che il canale `ch1` viene utilizzato per trasferire messaggi di tipo `integer`.

Avendo ogni canale un solo ricevente, adotteremo il criterio che questo venga dichiarato locale al processo che rappresenta l'unico ricevente dei messaggi inviati sul canale. Inoltre, per abilitare i processi mittenti a inviare messaggi sul canale, faremo l'ipotesi che il suo identificatore sia pubblico, cioè visibile a tutti i processi mittenti i quali potranno identificarlo utilizzando la *dot notation*:

```
<nome del processo>.<identificatore del canale>
```

Per inviare messaggi un processo utilizza la funzione primitiva

```
send (<valore>) to <porta>;
```

dove <porta> identifica il canale su cui inviare il messaggio e <valore> è un'espressione dello stesso tipo di <porta> e il cui valore rappresenta il contenuto del messaggio inviato. Per esempio, supponendo che la precedente dichiarazione del canale `ch1` sia locale al processo `ric`, il proceeso `mit` eseguendo l'operazione

```
send (a+10) to ric.ch1;
```

invia il valore 15 al processo `ric` tramite il canale `ch1` se, quando l'operazione viene eseguita, la variabile intera a contiene il valore 5.

Analogamente un processo, per ricevere un messaggio da un proprio canale, utilizza la funzione primitiva

```
receive (<variabile>) from <porta>;
```

dove <porta> identifica ancora il canale, locale al processo ricevente, dal quale ricevere il messaggio e <variabile> è l'identificatore di una variabile, dello stesso tipo di <porta>, a cui assegnare il valore del messaggio ricevuto.

Nel caso di canali simmetrici ogni processo, quando riceve un messaggio, può facilmente desumere chi sia il relativo mittente. Nel nostro caso, però, avendo ipotizzato canali asimmetrici, è necessario che il ricevente riesca a risalire a chi sia, fra vari mittenti, lo specifico processo che ha inviato il messaggio ricevuto. Per questo motivo, identificando con `process` un tipo predefinito offerto dal linguaggio e per il quale l'insieme dei valori coincide con i nomi di tutti i processi dell'applicazione faremo l'ipotesi che la funzione primitiva `receive` restituisca un valore di tipo `process` che corrisponde al nome del processo che ha inviato il messaggio ricevuto tramite la primitiva stessa. Per esempio, supposto che `mes` sia una variabile di tipo `integer` e `proc` una variabile di tipo `process`, l'esecuzione da parte del processo `ric` dell'operazione

```
proc=receive(mes) from ch1;
```

consente di ricevere un messaggio dal canale `ch1` assegnando a `mes` il valore ricevuto e a `proc` il nome del processo mittente.

Come abbiamo già messo in evidenza, la primitiva `send` non costituisce un punto di sincronizzazione nel caso di comunicazione asincrona mentre lo costituisce nel caso di comunicazione sincrona. Viceversa, la primitiva `receive` è sempre un punto di sincronizzazione, indipendentemente dal tipo di comunicazione, nel senso che il processo ricevente si blocca se tenta di eseguire la `receive` quando non sono messaggi pronti per essere ricevuti. Questo fatto, in certi casi, può costituire un problema. Per chiarire questo aspetto facciamo riferimento a un esempio. Supponiamo che su una risorsa astratta `R` sia possibile eseguire due operazioni `A` e `B` e che ciascuno degli `m` processi $P_1, ..., P_m$ abbia la necessità di operare su `R` tramite una delle due precedenti operazioni. Come è noto, in un ambiente a memoria comune questo problema è facilmente risolubile programmando `R` come un monitor con le funzioni `A(..)` e `B(..)`, che rappresentano i due metodi di accesso a `R` e dove ognuno dei processi, quando desidera operare su `R`, invoca la funzione desiderata passando i dovuti parametri. La competizione negli accessi a `R` viene automaticamente risolta mediante il meccanismo di mutua esclusione proprio dei monitor. Come abbiamo già detto, in un ambiente a scambio di messaggi `R` non può essere condivisa, ma lo stesso problema può essere facilmente risolto dichiarando `R` come l'istanza di una classe all'interno del solo processo `S`, il server di `R`, e obbligando ciascuno degli `m` proces

clienti a richiedere a S di eseguire su R, per suo conto, l'operazione desiderata, A o B. È quindi sufficiente, in questo semplice esempio, programmare il processo server S come un processo ciclico che ripete per sempre il seguente ciclo:

```
process S {
    ...........
    while(true) {
        <riceve una richiesta di servizio da un cliente>;
        <esegue su R la funzione A(..) o B(..) richiesta>;
        <restituisce al cliente i risultati>;
    }
}
```

e dove S, all'inizio del ciclo, si sospende se non ci sono richieste in arrivo, per essere riattivato non appena arriva la prima richiesta. La mutua esclusione tra richieste relative a clienti diversi, in questo caso, è implicita nel fatto che le operazioni su R sono eseguite da un processo sequenziale, il server, che le esegue quindi, per definizione, una alla volta. Cerchiamo adesso di capire come strutturare la comunicazione tra un qualunque cliente e il server. Le informazioni che un cliente deve passare al server sono fondamentalmente due:

a) il tipo di servizio richiesto (A o B);
b) i parametri con cui il server dovrà invocare la funzione richiesta per operare su R.

Per consentire ai clienti l'invio di queste richieste di servizio al processo server possiamo seguire due diverse alternative. La prima consiste nel prevedere un solo canale locale al server e tramite il quale questo riceve tutte le richieste di servizio, oppure dichiarare nel server due diversi canali sui quali indirizzare rispettivamente le richieste di servizio di tipo A e di tipo B. Per ciascuna delle due alternative cerchiamo di capire il tipo di messaggi che il cliente deve inviare al server.

Nel primo caso, avendo a disposizione un unico canale, il messaggio dovrà specificare sia il tipo del servizio richiesto (e per questo è sufficiente un valore binario che indichi uno dei due servizi A o B) sia i valori da passare come parametri alla funzione che deve essere eseguita. In generale, i tipi dei parametri per le due funzioni possono essere diversi. Per esempio, supponiamo che per eseguire A sia richiesto un parametro di tipo `integer` mentre per eseguire B sia richiesto un `real`. In questo caso il messaggio che il cliente deve inviare diventa una struttura con due campi: il primo campo che denota tipo di servizio richiesto (A o B) mentre il secondo campo contiene il valore del parametro, da passare al server, che sarà un `integer` nel caso di servizio A o un `real` nel caso di servizio B. Avendo però un unico canale per inviare qualunque richiesta, è ovvio che il tipo del messaggio deve contemplare una qualunque delle due precedenti varianti. Siamo perciò costretti a definire il tipo del canale, e quindi anche il tipo del messaggio, come una struttura con varianti di cui il tipo del servizio, A o B, rappresenta il discriminante. Ciò determina una complicazione sintattica e di conseguenza, come vedremo in un esempio nel prossimo capitolo, una maggiore complessità del programma del server, complicazione che diventa ancora più significativa con l'aumentare del numero di tipi di servizi offerti dal server.

La seconda alternativa, che prevede di riservare due diversi canali locali al server da cui ricevere separatamente le varie richieste di servizio, l'uno per i servizi di tipo A e l'altro per quelli di tipo B, è sicuramente molto più semplice. Infatti il primo canale sarà di tipo `integer` (il tipo del parametro richiesto per questo servizio) e il secondo di tipo `real`. Con questa seconda alternativa nasce però un altro problema: come si programma la prima istruzione del ciclo del server, indicata nel precedente schema di S come <riceve una richiesta di servizio da un cliente>? In altri termini, da quale canale il server decide di ricevere inizialmente? Se non ci sono particolari esigenze di priorità possiamo supporre di iniziare con una `receive` da uno qualunque dei due canali, per esempio dal canale relativo ai servizi di tipo A. In questo caso però potrebbe accadere che non ci siano messaggi da ricevere e allora il server si bloccherebbe pur avendo magari richieste disponibili sull'altro canale. Ovviamente avremmo potuto avere un problema simmetrico se la scelta fosse stata quella di iniziare con la `receive` sull'altro canale. Potremmo allora pensare all'opportunità di disporre, oltre che della `receive` bloccante, come abbiamo previsto, anche di una primitiva che si limiti a verificare se su un canale ci sono messaggi in arrivo, senza bloccare il processo se non ce ne sono. Purtroppo il problema non sarebbe risolto poiché nel caso in cui non fossero disponibili né messaggi su un canale né sull'altro il processo dovrebbe allora bloccarsi senza dover restare in attesa attiva continuando a testare i canali. Il problema quindi si ripresenterebbe: su quale canale si deve bloccare il server eseguendovi la `receive`? Se si blocca sul canale di tipo A, potrebbero arrivare per prime richieste sull'altro canale e viceversa. Il meccanismo di ricezione ideale per risolvere questo problema alla radice dovrebbe viceversa consentire al processo server di verificare la disponibilità di messaggi sui due canali, abilitando la ricezione di un messaggio da uno qualunque dei canali contenenti messaggi, se ce ne sono, o bloccando il processo in attesa che arrivi un messaggio, qualunque sia il canale su cui arriva, quando nessun canale contiene messaggi.

Molti linguaggi di alto livello hanno adottato un meccanismo di questo tipo utilizzando la notazione dei comandi con guardia introdotta da Dijkstra [68]. Un comando con guardia ha la seguente forma sintattica

<guardia> → <istruzione>;

dove, nel nostro caso, la <guardia> è costituita da una primitiva `receive` mentre <istruzione> è una qualunque istruzione del linguaggio. Una guardia può essere valutata per verificare se la ricezione è possibile senza ritardi. In particolare, la valutazione di una guardia restituisce uno dei due seguenti possibili valori: *guardia valida* se sul canale indicato nella `receive` ci sono messaggi disponibili (e cioè se la `receive` è eseguibile senza ritardi) o *guardia ritardata* se sul canale non ci sono messaggi e quindi se la `receive` non può essere eseguita. I comandi con guardia sono utilizzati all'interno di un'istruzione alternativa concettualmente simile all'istruzione `switch` presente in molti linguaggi di programmazione. L'istruzione alternativa ha la seguente sintassi, dove n rappresenta il numero dei rami dell'istruzione:

```
if
    [] <guardia_1> → <istruzione_1>;
    .
    .
    .
    [] <guardia_n> → <istruzione_n>;
fi
```

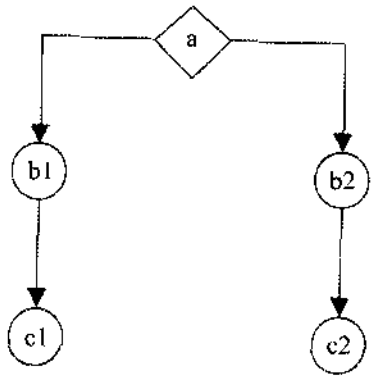


Figura 7.5 Comando con guardia senza condizione di sincronizzazione.

Eseguire questa istruzione significa svolgere le azioni schematizzate nella figura 7.5 e riportate di seguito:

a) vengono valutate le guardie di tutti i rami;

- **b1)** se una o più guardie sono *valide* viene scelto, in maniera non deterministica, uno dei rami con guardia *valida* e la relativa guardia viene eseguita (viene cioè eseguita la `receive` che coincide con la guardia scelta);
- **c1)** viene quindi eseguita l'istruzione relativa al ramo scelto e con ciò termina l'esecuzione dell'intera istruzione alternativa.

- **b2)** se tutte le guardie sono *ritardate* il processo in esecuzione si sospende in attesa che arrivi un messaggio che abilita la transizione di una guardia da *ritardata* a *valida*;
- **c2)** a quel punto il processo torna attivo, esegue la guardia che è diventata valida e quindi l'istruzione a essa associata e con ciò termina l'esecuzione dell'intera istruzione alternativa.

Oltre all'istruzione alternativa i comandi con guardia vengono utilizzati anche all'interno di istruzioni ripetitive

```
do
   [] <guardia_1> → <istruzione_1>;
   :
   [] <guardia_n> → <istruzione_n>;
od
```

la cui unica differenza rispetto al caso alternativo è dovuta al fatto che le azioni precedenti si ripetono per sempre.

Avendo a disposizione i comandi con guardia e supposto di indicare con `T` il tipo astratto di cui `R` è un'istanza, il precedente esempio può essere facilmente programmato nel seguente modo

```
ocess S {
 port int servizioA;
 port real servizioB;
 T R;
 int x;
 real y;
 do
 [] receive (x) from servizioA; ->
    {R.A(x);
    <eventuale restituzione dei risultati al cliente>;
    }
 [] receive (y) from servizioB; ->
    {R.B(y);
    <eventuale restituzione dei risultati al cliente>;
    }
 od
}
```

L'esempio precedente mette in evidenza due aspetti dei comandi con guardia che i caratterizzano la semplicità d'uso. Il primo riguarda il non determinismo implici nella semantica di queste istruzioni. Con riferimento al precedente esempio, ciò s gnifica che, quando il server esegue il comando ripetitivo e le due guardie sono e trambe valide, la scelta del ramo da eseguire è del tutto indifferente. Infatti ciò sign fica che sono presenti richieste di eseguire sia la funzione `A` sia la funzione `B`. M in questo caso, la corretta competizione tra processi richiede soltanto che le due op razioni non vengano eseguite concorrentemente per cui è corretto sia il caso in cui viene eseguita prima di `B` sia il viceversa, come specificato nel paragrafo 2.3 (ve figura 2.9). Il programma perciò sarebbe erroneo qualora funzionasse soltanto nell potesi in cui il compilatore, nel tradurre tale programma, implementasse una ben d terminata scelta. Questo livello di non determinismo è del tutto naturale in un sist ma concorrente. Il secondo aspetto positivo di queste istruzioni è che consentono programmatore di esprimere il fatto che un processo si sospende in attesa di u qualunque fra un insieme di eventi, nella fattispecie in attesa che arrivi uno qualu que fra un insieme di messaggi su canali diversi. Questo era proprio quanto richies per risolvere il problema precedente evitando le attese attive. Chiaramente, essend comandi con guardia un meccanismo linguistico di alto livello, la loro semplicità uso dipende da una maggiore complessità di implementazione da parte del compi tore e del supporto a tempo di esecuzione del linguaggio.

Prima di completare l'argomento sui comandi con guardia, vediamo la loro v sione più generale che mette in evidenza tutta la loro potenza espressiva. Per ques facendo ancora riferimento al precedente esempio, supponiamo che la risorsa `R` cui possono essere eseguite le sole operazioni `A` e `B` sia tale per cui, per ognuna de due operazioni, sia necessario specificare una condizione di sincronizzazione. I esempio, supponiamo che l'operazione `A` sia eseguibile soltanto quando lo stato terno di `R` soddisfa la condizione `condA` e l'operazione `B` solo quando è v `condB`. Ciò significa che, per esempio, `condA` è parte integrante della precondiz ne di `A`. Se `R` fosse una coda e `A` l'operazione di estrazione, la condizione `con` corrisponderebbe allo stato interno di *coda con almeno un elemento*. In ambient memoria comune abbiamo visto come, mediante semafori o variabili `conditi`

sia possibile bloccare un processo che intende eseguire un'operazione su una risorsa quando la precondizione dell'operazione non è verificata. Utilizzando i comandi con guardia, le condizioni di sincronizzazione possono essere specificate complicando il concetto di guardia che, nei casi più generali, non è costituita da una semplice primitiva `receive` ma da una coppia

(<espressione di tipo boolean>) ; <primitva `receive`>

dove l'espressione rappresenta una condizione di sincronizzazione. In questo caso, la valutazione di una guardia può dare luogo a tre diversi valori: una guardia è *fallita* se il valore dell'espressione boolean è `false`, è *valida* se il valore dell'espressione boolean è `true` e la `receive` può essere eseguita senza ritardo ed è *ritardata* se il valore dell'espressione boolean è `true` ma la `receive` non può essere eseguita.

Adesso, eseguire un'istruzione alternativa significa svolgere le azioni schematizzate nella figura 7.6 e qui di seguito riportate:

a) vengono valutate le guardie di tutti i rami;

b1) se tutte le guardie sono *fallite* è un errore di programmazione.

b2) se una o più guardie sono *valide* viene scelto, in maniera non deterministica, uno dei rami con guardia *valida* e la relativa guardia viene eseguita (viene cioè eseguita la `receive` che coincide con la guardia scelta);

c2) viene quindi eseguita l'istruzione relativa al ramo scelto e con ciò termina l'esecuzione dell'intera istruzione alternativa.

b3) se tutte le guardie non *fallite* sono *ritardate* il processo in esecuzione si sospende in attesa che arrivi un messaggio che abilita la transizione di una guardia da *ritardata* a *valida*;

c3) a quel punto il processo torna attivo, esegue la guardia che è diventata valida e, quindi, l'istruzione a essa associata e, con ciò, termina l'esecuzione dell'intera istruzione alternativa.

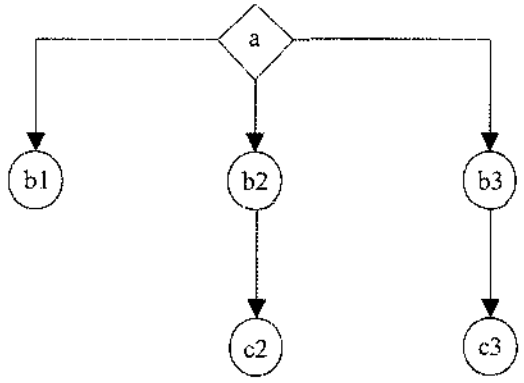


Figura 7.6 Comando con guardia con condizione di sincronizzazione.

La stessa cosa vale per l'istruzione ripetitiva con l'unica variante che l'eventualità in cui tutte le guardie sono fallite coincide, in questo caso, con la terminazione dell'istruzione ripetitiva. Non è detto, infine, che le due componenti di una guardia siano sempre presenti: se manca l'espressione `boolean` significa che non esiste condizione di sincronizzazione, ovvero è come se fosse presente un'espressione con valore `true`; se manca la primitiva `receive` significa che il valore della guardia non potrà mai essere *ritardata*.

Possiamo adesso completare l'esempio di prima con la specifica delle due condizioni di sincronizzazione:

```
process S {
   port int servizioA;
   port real servizioB;
   T R;
   int x;
   real y;
   do
   [] (condA) ; receive (x) from servizioA; ->
      {R.A(x);
         <eventuale restituzione dei risultati al cliente> ;
      }
   [] (condB) ; receive (y) from servizioB; ->
      {R.B(y);
         <eventuale restituzione dei risultati al cliente> ;
      }
   od
```

Possiamo concludere questa introduzione alle primitive di comunicazione, con riferimento ai due meccanismi asincrono e sincrono, con le seguenti considerazioni. Gli strumenti linguistici più adatti a risolvere problemi di interazione tra processi del tipo cliente/servitore sono sicuramente i comandi con guardia e a essi faremo riferimento nei successivi capitoli. In assenza di questi strumenti sarà necessario definire un solo canale per il processo server, definendo il tipo di messaggi come una struttura con varianti. Per generalità e al fine di verificare il diverso livello di complessità illustreremo alcuni esempi adottando questo criterio.

Mentre per i casi di comunicazione asincrona e sincrona faremo quindi riferimento alle due primitive `send` e `receive`, nel terzo caso che prenderemo in considerazione nei successivi capitoli, quello relativo alla sincronizzazione estesa (chiamata di operazioni remote), la notazione a cui faremo riferimento coincide con quella nota delle chiamate di funzione. Questa notazione è particolarmente adatta a programmare interazioni tipo cliente/servitore. In pratica un processo cliente viene programmato adottando lo stesso paradigma di programmazione usato nei sistemi che seguono il modello a memoria comune, cioè utilizzando una chiamata di funzione tutte le volte che è richiesta un'operazione su una risorsa, come se la risorsa fosse un monitor direttamente visibile al processo cliente. In realtà, a questa semplice notazione sintattica corrisponde una coppia di operazioni di scambio di messaggi (come indicato nella figura 7.4) per trasferire una richiesta di servizio dal processo cliente al server che dovrà eseguire l'operazione chiamata e, successivamente, dal server al cliente per restituire i risultati del servizio stesso. Tale corrispondenza viene attuat

dal compilatore utilizzando il supporto a tempo di esecuzione del linguaggio. Come vedremo, a fronte di questo meccanismo sintattico utilizzato per programmare il processo cliente, verranno introdotte anche opportune notazioni per programmare il processo server che è destinato a ricevere le richieste di servizio.

7.4 Sommario

Scopo di questo capitolo è stato quello di introdurre e illustrare le principali caratteristiche di un modello architetturale di macchina concorrente del tutto diverso rispetto a quello a cui è stato fatto riferimento nei precedenti tre capitoli. Questo modello architettutrale, noto come modello a scambio di messaggi, sarà quello a cui faremo riferimento nei successivi tre capitoli. Al fine di illustare le principali caratteristiche del modello e di definire anche gli strumenti linguistici che verranno utilizzati nella parte restante del testo, è stato definito il concetto di *canale* come principale strumento da utilizzare per consentire le interazioni tra processi in architetture organizzate secondo il modello a scambio di messaggi.

Del canale sono state illustrate le varie tipologie in base ai seguenti parametri: la direzione del flusso dei dati che un canale può trasferire (canali mono e bidirezionali), la designazione del canale e dei processi sorgente e destinatario di ogni comunicazione (canali simmetrici e asimmetrici), il tipo di sincronizzazione fra i processi comunicanti (canali asincroni, sincroni e con sincronizzazione estesa).
Infine sono state introdotte le varie tipologie di primitive di comunicazione che verranno utilizzate, di volta in volta, nei successivi tre capitoli del testo, ivi incluse le istruzioni relative ai comandi con guardie.

7.5 Note bibliografiche

Il modello a scambio di messaggi è stato introdotto come modello organizzativo di un sistema operativo alla fine degli anni sessanta prima ancora, cioè, che iniziasse lo sviluppo dei sistemi distribuiti. Il primo esempio di questo tipo è dovuto a Brinch Hansen [65] che ha adottato il medello a scambio di messaggi per organizzare e progettare il sistema operativo RC-4000. Successivamente molti sono stati i sistemi operativi progettati e realizzati secondo questo modello (basti pensare la meccanismo delle socket che molte versioni di UNIX forniscono per abilitare processi a inviare e ricevere messaggi [66]).

Le principali notazioni usate per la comunicazione e la sincronizzazione dei processi in ambienti a scambio di messaggi sono confrontate nel lavoro di Andrews e Schneider [67].

I comandi con guardie utilizzati nei linguaggi che seguono il modello a scambio di messaggi sono stati introdotti da Dijkstra [68] e ulteriormente illustrati da Bernstein [69].

8

Primitive di comunicazione asincrone

Nel precedente capitolo abbiamo introdotto la notazione sintattica che utilizzerem in questo capitolo per illustrare come le primitive di comunicazione asincrone possa no essere utilizzate per risolvere vari problemi di interazione tra processi in un am biente, virtualmente o fisicamente, distribuito. Inizieremo illustrando come, dispo nendo di un linguaggio che fornisce tali primitive, si possano risolvere gli stessi pro blemi visti nel modello a memoria comune, utilizzando però un diverso paradigm di programmmazione. In particolare, metteremo in evidenza come sia possibile ind viduare uno schema di corrispondenza tra i meccanismi utilizzati per risolvere u problema nel modello a memoria comune e quelli utilizzati per risolvere lo stess problema in un ambiente a scambio di messaggi.

Alla fine del capitolo verranno poi illustrati alcuni esempi di possibili implemen tazioni di queste primitive. In particolare vedremo una loro possibile realizzazion come primitive del nucleo di un sistema operativo, offerte quindi come meccanism primitivo direttamente dalla macchina concorrente, distinguendo i due casi di nucle per architetture fisiche a memoria condivisa (monoprocessori o multiprocessori) per architetture fisicamente distribuite.

8.1 Processi servitori

Come primo esempio di uso delle primitive asincrone vedremo come sia possibi simulare, tramite un processo servitore, il concetto di risorsa condivisa proprio del schema a memoria comune, nel quale questo concetto è stato implementato media te il costrutto monitor. Ciò consentirà di definire uno schema generale di corrispon denza fra i due diversi paradigmi con cui uno stesso problema viene risolto nei du diversi ambienti. Per definire questo schema di corrispondenza partiremo dal cas più semplice di risorsa condivisa che mette a disposizione di un insieme di proces clienti una sola operazione, su cui è previsto il solo vincolo della mutua esclusion Passeremo poi al caso più generale in cui sulla risorsa siano eseguibili più operazi ni, ovviamente ancora in mutua esclusione, ma senza particolari vincoli di sincroni

zazione diretta, cioè senza che siano specificate particolari condizioni di sincronizzazione. Infine, tratteremo il caso più generale che prevede non solo più operazioni sulla risorsa, ma anche la specifica di particolari vincoli di sincronizzazione.

Iniziamo, quindi, col caso più semplice di una risorsa condivisa su cui operare con una sola operazione. Per esempio, utilizzando il costrutto monitor e indicando con `ris` la risorsa in questione e con `tipo_ris` il suo tipo, potremmo fare riferimento, in generale, al seguente codice:

```
monitor tipo_ris {
   tipo_var var;
   {<eventuale istruzione di inizializzazione>;}
   public tipo_out fun(tipo_in x) {
      <corpo della funzione fun>;
   }
}

      .........
tipo_ris ris;
      .........
```

dove, `var` rappresenta la struttura dati locale al monitor, `tipo_var` è il tipo di tale struttura, `fun` è la funzione che rappresenta l'unica operazione offerta dal monitor; tale funzione, in generale, può richiedere un parametro `x`, il cui tipo è indicato con `tipo_in`, e restituire un valore di tipo `tipo_out` (come caso particolare, tale valore può ovviamente essere `void`). Nel codice è riportato, per generalità, anche l'eventuale istruzione che deve essere eseguita per inizializzare la struttura dati `var` di ogni istanza del monitor.

Un qualunque processo che debba operare sulla risorsa si limita, in questo caso, a invocare la funzione `fun` passandogli il parametro richiesto:

```
process cliente {
   tipo_in a;
   tipo_out b;
      .........
   b=ris.fun(a);
      .........
}
```

Volendo risolvere lo stesso problema in un ambiente a scambio di messaggi dovremmo riservare la risorsa `ris` locale a un solo processo, il `server`, e obbligare tutti i processi clienti a richiedere esplicitamente al `server` di eseguire l'operazione `fun` a loro favore. Nella figura 8.1b) è riportato il codice relativo a una possibile realizzazione del processo `server` mentre nella figura 8.1a) compare il codice di un generico processo `cliente`. Al processo `server` potranno arrivare esclusivamente richieste relative all'unico servizio corrispondente all'esecuzione della sola operazione `fun`. Ciò significa che dovremo dichiarare un solo canale, locale al processo, su cui ricevere messaggi di tipo `tipo_in`, cioè messaggi contenenti il valore del parametro necessario per eseguire l'operazione `fun`. Eseguita l'operazione, il `server` dovrà inoltre restituire il risultato, di tipo `tipo_out`, al cliente che ha richiesto il servizio. Ogni processo cliente dovrà quindi avere un canale di tipo `tipo_out` su cui ricevere il risultato dell'operazione richiesta.

a) processo `cliente`

```
process cliente {
   port tipo_out risposta;
   tipo_in a;
   tipo_out b;
   process p;
      ........
      ........
   send(a)to server.input;
   p=receive(b)from risposta;
      ........
}
```

b) processo `server`

```
tipo_out fun(tipo_in x);
process server {
   port tipo_in input;
   tipo_var var;
   process p;
   tipo_in x;
   tipo_out y;
   {<eventuale inizializzazione>;}
   while(true){
      p=receive(x)from input;
      y=fun(x);
      send(y)to p.risposta;
   }
}
```

Figura 8.1 Processo `server` con un unico servizio senza condizioni di sincronizzazione.

Locale al `server` compare quindi la dichiarazione del canale `input` da cui ricevere le richieste di servizio e quella della variabile `var` corrispondente alla struttura dati del precedente monitor. La variabile `x` è destinata a contenere il valore del parametro ricevuto tramite una richiesta di servizio e `y` è la variabile a cui assegnare il valore restituito dalla funzione `fun`. Il codice del `server` è un ciclo senza fine che inizia con la ricezione di una richiesta di servizio, ricezione bloccante se non ci sono richieste pendenti. Una volta ricevuta una richiesta, vengono eseguite le istruzioni relative alla funzione `fun`. Tali istruzioni, operando anche sulla variabile `x`, assegnano a `y` il risultato del servizio (come specificato nella definizione della funzione `fun`). Quindi, il valore di `y` viene restituito al processo `cliente`, il cui nome è stato assegnato alla variabile `p` dalla primitiva `receive`. Tale valore viene restituito tramite il canale `risposta` che ogni cliente deve dichiarare per questo scopo.

Locale al processo `cliente` compare la dichiarazione del canale `risposta`. Quando il `cliente` ha bisogno di eseguire l'operazione `fun` sulla risorsa, invia la richiesta al `server`, contenente il valore `a` del parametro, e quindi si sospende in attesa di ricevere il risultato dal canale `risposta`, risultato che viene poi assegnato alla variabile `b`.

Nello schema precedente abbiamo fatto riferimento al caso in cui la funzione `fun` abbia un solo parametro e restituisca un valore diverso da `void`. Chiaramente lo schema non cambia se la funzione `fun` richiede più parametri. In questo caso, il tipo del messaggio dovrebbe essere una struttura i cui campi corrispondono ai singoli parametri della funzione. Nel caso particolare in cui la funzione non abbia parametri, il messaggio non contiene nessuna informazione e la ricezione di una richiesta di servizio corrisponde semplicemente alla ricezione di un messaggio il cui contenuto informativo è nullo. Per questo motivo, faremo l'ipotesi che il linguaggio offra un tipo predefinito, che indicheremo con `signal`, il cui unico valore è `void`. Infine, nel caso particolare in cui la funzione `fun` non restituisca nessun valore (`void`), il `server` non deve inviare nessun risultato al `cliente`, il quale quindi, una volta inviata la richiesta di servizio, non si pone in attesa del risultato, ma prosegue con

correntemente col server. In questo caso, la corrispondenza col modello a memoria comune viene meno in quanto il cliente esegue concorrentemente con l'esecuzione della funzione fun. Se volessimo mantenere la stessa corrispondenza dovremmo obbligare il cliente ad attendere la ricezione di un messaggio, che potrebbe essere semplicemente di tipo signal, da parte del server.

Passiamo adesso al secondo caso, quello in cui sulla risorsa condivisa sia possibile operare con due diverse operazioni, fun1 e fun2. Con riferimento ancora al modello a memoria comune, dovremmo specificare il tipo della risorsa mediante il seguente monitor:

```
monitor tipo_ris {
   tipo_var var;
   [<eventuale istruzione di inizializzazione>;]
   public tipo_out1 fun1(tipo_in1 x1) {
      <corpo della funzione fun1>;}
   public tipo_out2 fun2(tipo_in2 x2) {
      <corpo della funzione fun2>;}
}
```

In questo caso, avendo due funzioni, abbiamo indicato con tipo_in1 e tipo_in2, rispettivamente, i tipi dei parametri delle due funzioni e, analogamente, con tipo_out1 e tipo_out2 i tipi dei valori restituiti.

In base a quanto visto nel paragrafo 7.3, per mostrare come risolvere il problema in ambiente a scambio di messaggi, inizieremo supponendo di non disporre dei comandi con guardie e quindi utilizzando un unico canale per far pervenire al server tutte le richieste di servizio; successivamente, mostreremo la stessa soluzione ma utilizzando i comandi con guardie.

Nel primo caso, definiamo preventivamente l'unico tipo di messaggi (che denoteremo come in_mes) che il server può ricevere. Questo tipo deve indicare due cose: l'operazione richiesta dal cliente e il valore del parametro richiesto dall'operazione stessa. Essendo, in generale, diversi i tipi dei parametri, dovremo ricorrere a una struttura con varianti. La struttura in_mes ha quindi due campi: il primo campo (servizio) specifica il tipo di operazione, che nel nostro esempio è di tipo enumerazione, con i due valori corrispondenti alle due possibili operazioni, e che costituisce anche il discriminante fra le due varianti della struttura; il secondo campo (parametri) è l'unione tra le due possibili varianti corrispondenti ai due tipi dei parametri. Avremo perciò:

```
typedef struct {
   enum {fun1,fun2} servizio;
   union {
      tipo_in1 x1;
      tipo_in2 x2;
   } parametri;
} in_mes;
```

Definito il tipo dei messaggi inviati al server, possiamo adesso riscrivere il codice di questo processo (vedi figura 8.2). Il canale input del processo è adesso di tipo in_mes e dello stesso tipo è la variabile locale richiesta, utilizzata per ricevere un messaggio all'inizio del ciclo senza fine del server.

```
tipo_out1 fun1(tipo_in1 x1);
tipo_out2 fun2(tipo_in2 x2);
process server {
   port in_mes input;
   tipo_var var;
   process p;
   in_mes richiesta;
   tipo_out1 y1; tipo_out2 y2;
   [<eventuale istruzione di inizializzazione>;]
   while(true) {
      p=receive(richiesta) from input;
      switch(richiesta.servizio) {
         case fun1: {
            y1=fun1(richiesta.parametri.x1);
            send(y1) to p.risposta1;
            break;
         }
         case fun2: {
            y2=fun2(richiesta.parametri.x2);
            send(y2) to proc.risposta2;
            break;
         }
      }
   }
}
```

Figura 8.2 Processo server con più servizi senza comandi con guardia.

Una volta ricevuto un messaggio, il processo esegue, tramite uno switch, un ram che corrisponde al tipo di servizio richiesto. Se, per esempio, è stata richiesta l'op razione fun1, si eseguono le istruzioni relative alla funzione fun1, operando s valore del parametro x1 ricevuto nel campo parametri del messaggio e, infin viene restituito al cliente il risultato y1, tramite il canale risposta1 che il clien stesso avrà riservato per questo scopo. Un'analoga sequenza di operazioni viene es guita nel caso di servizio di tipo fun2.

Se, viceversa, sono disponibili comandi con guardia, lo schema del server semplifica poiché possiamo definire due diversi canali (input1 e input2), l'u di tipo tipo_in1 e l'altro di tipo_in2, utilizzandoli per ricevere richieste d due diversi tipi all'interno di un'istruzione ripetitiva (vedi figura 8.3), come indica nel paragrafo 7.3.

Lo schema del processo cliente ovviamente non cambia nei due casi e cor sponde a quello già visto nel caso di risorsa con una sola operazione, salvo riserva uno o due canali di tipo diverso da cui ricevere i risultati a seconda del servizio chiesto al server.

Vediamo adesso il caso più generale di una risorsa condivisa che offre le d operazioni specificando, stavolta, anche delle condizioni di sincronizzazione. P esempio supponiamo che la condizione logica cond1 sia una precondizione per l'

secuzione di `fun1` e, analogamente, `cond2` lo sia per l'esecuzione della `fun2`. In questo caso, nel monitor la funzione `fun1` inizia con l'usuale istruzione:

`if(!cond1) c1.wait;` oppure `while(!cond1) c1.wait;`

dove `c1` è una variabile `condition` e altrettanto vale per l'altra funzione.

Chiaramente, se `cond1` è una precondizione di `fun1`, ciò significa che un processo si blocca sulla variabile condition `c1` se invoca questa funzione quando la condizione non è vera. Il processo potrà proseguire solo dopo che la condizione attesa è diventata vera e ciò potrà avvenire solo dopo che un altro processo, avendo eseguito un'altra funzione, nel nostro caso `fun2`, avrà cambiato lo stato interno della risorsa rendendo vera `cond1`. È quindi naturale che, alla fine della funzione `fun2`, verificata la validità della condizione `cond1`, venga eseguita una `signal` su `c1`. La stessa cosa vale ovviamente per l'altra condizione `cond2`. Possiamo quindi ipotizzare, come visto anche nei vari esempi riportati nel capitolo 6 e facendo riferimento a una semantica della `signal` tipo *signal_and_urgent*, che lo schema delle due funzioni del monitor sia il seguente:

```
tipo_out1 op1(tipo_in1 x1){
     .........
   if (! cond1) wait(c1);
     .........
   signal(c2);
     .........
}
```

```
tipo_out2 op2(tipo_in2 x2){
     .........
   if (! cond2) wait(c2);
     .........
   signal(c1);
     .........
}
```

Rivediamo quindi come cambiano gli schemi relativi al processo `server` nei due casi senza e con comandi con guardia.

```
tipo_out1 fun1(tipo_in1 x1);
tipo_out2 fun2(tipo_in2 x2);
process server {
   port tipo_in1  input1;
   port tipo_in2  input2;
   tipo_var var;
   process p;
   tipo_in1 x1; tipo_in2 x2;
   tipo_out1 y1; tipo_ou2t y2;
   {<eventuale istruzione di inizializzazione>;}
   do
      [] p=receive(x1) from input1; ->
         y1=fun1(x1);
         send(y1) to proc.rispostal;
      [] p=receive(x2) from input2; ->
         y2=fun1(x2);
         send(y2) to proc.risposta2;
   od;
}
```

Figura 8.3 Processo `server` con più servizi e con comandi con guardia.

Riprendiamo per primo lo schema visto precedentemente, senza usare comandi co guardia, limitandoci al codice eseguito dal processo e senza riportare i dettagli relat vi alle dichiarazioni. In particolare, il corpo del processo continua a essere costitui dall'istruzione `receive` seguita da uno `switch` con i due rami, uno per ciascun dei due casi relativi ai due tipi di servizi richiesti. Ciascuno dei due rami però adess è molto più complesso, poiché deve prevedere che la richiesta di servizio ricevu non sia eseguibile qualora non sia verificata la corrispondente precondizione logic Quindi ogni ramo dello `switch` è costituito da un'istruzione alternativa (`if-el se`) per tener conto dei due casi: "richiesta di servizio eseguibile poiché la precond zione logica è vera" e "richiesta non eseguibile poiché la precondizione è falsa". quest'ultimo caso è necessario memorizzare in una coda locale al processo `serve` la richiesta di servizio già ricevuta, ma che non è stato possibile servire, al fine prenderla in considerazione in un secondo momento. Se viceversa, la richiesta p essere servita, alla fine del servizio è necessario verificare se, avendo modificato stato della risorsa, non sia diventata vera la condizione attesa da una delle richies già ricevute ma non ancora servite. In quest'ultimo caso, una delle richieste pende può essere servita.

Vediamo, viceversa, quanto sia semplice l'estensione dello schema con coman con guardia (vedi figura 8.4). Infatti, è adesso sufficiente inserire nella guardia ciascuno dei due rami del `server` anche la componente logica rappresentata da corrispondente condizione.

Concludiamo questo paragrafo ricapitolando, nella tabella 8.1, lo schema di co rispondenza tra il concetto di risorsa condivisa (`monitor`), proprio del modello memoria comune, e quello di un processo `server`, tipico del modello a scambio messaggi: viene fatta l'ipotesi di avere a disposizione comandi con guardia.

```
tipo_out1 fun1(tipo_in1 x1);
tipo_out2 fun2(tipo_in2 x2);
process server {
   port tipo_in1  input1;
   port tipo_in2  input2;
   tipo_var var;
   process p;
   tipo_in1 x1; tipo_in2 x2;
   tipo_out1 y1; tipo_ou2t y2;
   {<eventuale istruzione di inizializzazione>;}
   do
      [](cond1); p=receive(x1) from input1; ->
         y1=fun1(x1);
         send(y1) to proc.rispostal;
      [](cond2); p=receive(x2) from input2; ->
         y2=fun1(x2);
         send(y2) to proc.risposta2;
   od;
}
```

Figura 8.4 Processo `server` con la specifica di condizioni di sincronizzazione.

Modello a memoria comune: `monitor`	**Modello a scambio di messaggi:** `server`
risorsa condivisa: istanza di un monitor	risorsa condivisa: struttura dati locale a un processo server
identificatore di funzione di accesso al monitor	porta del processo server
tipo dei parametri della funzione	tipo della porta
tipo del valore restituito dalla funzione	tipo di una porta da cui il processo cliente riceve il risultato
per ogni funzione del monitor	un ramo (comando con guardia) dell'istruzione ripetitiva che costituisce il corpo del server
condizione di sincronizzazione di una funzione	espressione logica componente la guardia del ramo corrispondente alla funzione
chiamata della funzione	invio della richiesta sulla corrispondente porta del server seguito da attesa dei risultati sulla propria porta
esecuzione in mutua esclusione fra le chiamate alle funzioni del monitor	scelta di uno dei rami con guardia valida del comando ripetitivo del server
corpo della funzione	istruzione del ramo corrispondente alla funzione
relazione invariante del monitor	relazione invariante dell'istruzione ripetitiva del server

Tabella 8.1 Corrispondenza tra `monitor` (modello a memoria comune) e processo `server` (modello a scambio di messaggi).

La tabella di corrispondenza vuole mettere in evidenza esclusivamente il fatto che la tipologia di problemi, che nei due ambienti devono essere risolti, è la stessa anche se diversi, ma logicamente equivalenti, sono i paradigmi di soluzione degli stessi problemi. A questa equivalenza logica, ovviamente, non corrisponde un'analoga equivalenza di prestazioni. Questa, infatti, dipende strettamente dall'efficienza con cui i diversi meccanismi sono implementati sulle corrispondenti macchine virtuali. Inoltre è necessario tener conto che, nel modello a scambio di messaggi, al concetto di monitor, cioè di oggetto passivo, corrisponde un processo, ovvero un'entità attiva che in alcuni casi, come visto anche precedentemente, può generare schemi non del tutto equivalenti a quelli tipici del modello a memoria comune.

8.2 Esempi di processi servitori

Per chiarire ulteriormente quanto riportato in modo generico nel precedente paragrafo, cerchiamo adesso di applicare quei criteri di corrispondenza al fine di risolvere alcuni problemi già esaminati nei precedenti capitoli relativi al modello a memoria comune. In particolare, cercheremo di risolvere un problema di allocazione dinamica di risorse specificando il codice di un processo server che corrisponda al gestore di un pool di risorse equivalenti. Come secondo esempio illustreremo come sia possibile implementare, tramite un processo server, una mailbox, cioè un canale da molti a molti.

8.2.1 Gestione di un pool di risorse equivalenti

Il primo problema è quello tipico di un processo cliente che, avendo a disposizione un pool di risorse equivalenti, richiede a un gestore l'uso esclusivo di una fra quelle disponibili, rilasciando al gestore la risorsa quando questa non è più necessaria.

Per schematizzare questo problema in un ambiente a scambio di messaggi, dove al solito ogni risorsa è propria di un processo, possiamo ipotizzare che alle `n` risorse equivalenti corrispondano `n` processi P_1, ..., P_n, ciascuno dei quali è l'unico a poter operare sulla corrispondente risorsa in risposta a precise richieste da parte dei processi clienti. La necessità di un ulteriore processo `server` nasce dall'esigenza che un cliente ha di scegliere, fra gli `n` processi, uno fra quelli disponibili, se ce ne sono, evitando di inviare richieste di servizio a chi sta già operando per altri clienti. Con riferimento alla figura 8.5, possiamo quindi illustrare il protocollo che ogni processo cliente deve eseguire per operare su una delle risorse del pool.

Per prima cosa il cliente invia al `server` un messaggio per chiedere l'indice di una risorsa disponibile e quindi si pone in attesa di ricevere dal `server` questa informazione. Successivamente può operare direttamente col processo il cui indice è stato restituito dal server. Quando poi la risorsa non è più necessaria il cliente invia al `server` un messaggio per indicare il rilascio della risorsa stessa.

Iniziamo mostrando il codice del `server`, nell'ipotesi di avere a disposizione comandi con guardia. In particolare, con riferimento agli esempi già visti nei capitoli relativi al modello a memoria comune, applichiamo le regole precedentemente illustrate:

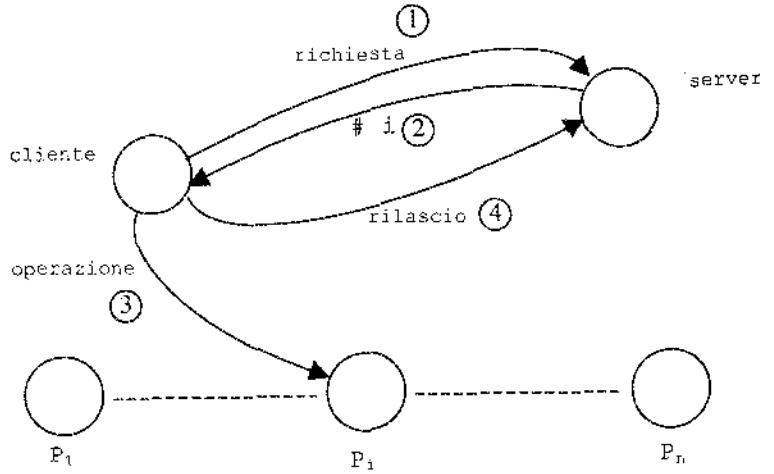


Figura 8.5 Gestore di un pool di risorse equivalenti

```
process server{
   port signal richiesta;
   port int rilascio;
   int disponibili=N;
   boolean libera[N];
   process p;
   signal s;
   int r;
   for (int i=0;i<N;i++)libera[i]=true; //inizializzazione
   do
   [] (disponibili>0);p=receive(s)from richiesta; ->
      int i=0;
      while (!libera[i]) i++;
      libera[i] = false;
      disponibili--;
      send(i) to p.risorsa;
   [] p=receive(r)from rilascio; ->
      disponibili++;
      libera[r]=true;
   od;
}
```

Il `server` ha due porte. La prima (`richiesta`), attraverso la quale riceve la richiesta di una risorsa, è di tipo `signal` in quanto la `richiesta` non ha parametri. L'altra (`rilascio`) è di tipo intero, poiché il processo che rilascia una risorsa indica l'indice della risorsa rilasciata. Il codice del `server`, dopo l'istruzione di inizializzazione, è costituito da una sola istruzione ripetitiva con due rami. Il primo, caratterizzato dalla ricezione di una richiesta, è condizionato dalla disponibilità di risorse e, quando viene eseguito, svolge esattamente lo stesso algoritmo che la funzione `richiesta` svolgerebbe nel caso di un monitor `gestore`. Adesso, però, invece di restituire l'indice della risorsa allocata tramite l'istruzione `return`, questo valore viene restituito al processo cliente mediante una `send`. A questo scopo, il processo cliente dovrà riservare un'apposita porta, indicata nel testo con l'identificatore `risorsa`. L'altro ramo, che corrisponde a un rilascio, non è condizionato come non lo sarebbe la funzione `rilascio` di un monitor `gestore`. Di seguito viene riportato anche il codice di un generico processo `cliente`.

```
process cliente{
   port int risorsa;
   signal s;
   int r;
   process p;
      .........
   send(s)to server.richiesta; //richiesta della risorsa
   p=receive(r)from risorsa; //attesa della risposta
      <uso della risorsa r-esima>
   send(r)to server.rilascio; //rilascio della risorsa
      .........
}
```

La soluzione dello stesso problema, senza utilizzare comandi con guardia, sarebbe risultata molto più complessa. Lasciamo al lettore il compito di descrivere tale soluzione utilizzando i criteri generali prima esposti.

Prima di terminare questo esempio, riportiamo di seguito il codice del `server` nel caso particolare in cui il pool di risorse coincida con una risorsa soltanto. In questo caso anche il rilascio non prevede nessun parametro e quindi anche la porta `rilascio` del `server` è di tipo `signal`. Analogamente, la risposta da restituire al `cliente` è un semplice `signal` per indicare che l'allocazione è stata completata. Infine, nel `server`, al posto del vettore `libera`, sarà sufficiente una sola variabile `boolean` rendendo del tutto inutile l'intero `disponibili`:

```
process server{
   port signal richiesta;
   port signal rilascio;
   boolean libera=true;
   process p;
   signal s;
   do
   [] (libera); p=receive(s) from richiesta; ->
      libera=false;
      send(s)to p.risorsa;
   [] p:=receive(s) from rilascio; ->
      libera=true;
   od;
}
```

Il codice del `server`, come è già stato indicato nei precedenti capitoli, in questo caso particolare ha la stessa funzionalità di un semaforo binario. Da questo semplice esempio possiamo risalire al codice di un processo servitore che simula un semaforo generale:

```
process semaphore{
   port signal P;
   port signal V;
   int valore=I;
   process proc;
   signal s;
   do
   [] (valore>0);p=receive(s)from P; ->
      valore--;
      send(s)to proc.risposta;
   [] p=receive(s) from V; ->
      valore++;
   od;
}
```

dove la costante `I` rappresenta il valore iniziale del semaforo; con `risposta` abbiamo indicato la porta di tipo `signal` su cui il processo `cliente` resta in attesa che sia completata l'operazione `P`, di cui ha chiesto l'esecuzione al `server` tramite un messaggio inviato sull'omonima porta.

8.2.2 Problema dei produttori/consumatori e realizzazione di una mailbox

Come secondo esempio riprendiamo il classico problema dei produttori/consumatori che è stato introdotto nei sistemi a memoria comune e che è stato risolto sia con i semafori che con i monitor. In quei casi, il problema è stato risolto programmando una risorsa condivisa con l'obiettivo di fornire il buffer per memorizzare i messaggi già inviati ma non ancora ricevuti. Anche per questo problema potremmo riprendere la soluzione vista con i monitor e, utilizzando i criteri di corrispondenza prima esaminati, scrivere il codice di un processo server che fornisce le stesse funzionalità del monitor. In questo caso però, essendo il problema un esempio di comunicazione tra processi, possiamo semplificare la soluzione in quanto le primitive `send` e `receive` forniscono già il supporto alla comunicazione. In particolare, come è stato indicato nel paragrafo 7.2, le primitive asincrone eliminano la necessità di programmare esplicitamente un buffer in quanto i canali di comunicazione contengono già, al loro interno, un buffer di dimensione concettualmente illimitata.

In base a queste considerazioni, se limitassimo l'esempio a uno o più processi produttori e a un solo consumatore (vedi figura 8.6), la realizzazione di un processo server sarebbe del tutto superflua in quanto la soluzione sarebbe offerta direttamente dalla porta a cui un produttore invia i messaggi e da cui il consumatore li preleva.

Pur non essendo necessario programmare esplicitamente un buffer dei messaggi, un processo server è comunque necessario se estendiamo la soluzione al caso in cui siano presenti più consumatori o nel caso in cui si voglia limitare il numero dei messaggi già inviati ma non ancora ricevuti, per esempio fissando tale limite a un valore `N` predefinito.

In presenza di più consumatori, infatti, nasce un problema analogo a quello già visto nel caso di un pool di risorse equivalenti. In questo caso, quando il produttore deve inviare un messaggio a quale consumatore lo invia? Se ne viene scelto uno a caso, può capitare che il consumatore prescelto in quel momento non sia disponibile a ricevere messaggi mentre altri potrebbero esserlo. Per risolvere efficientemente il problema, conviene quindi interporre tra il produttore e i consumatori un server, a cui il produttore invia i messaggi, e lasciare al server il compito di scegliere il consumatore a cui indirizzare il messaggio, in base alle disponibilità a ricevere messaggi che i singoli consumatori possono specificare allo stesso server. Nella figura 8.7 viene descritto lo schema di collegamento fra i processi. Il server, il cui nome è indicato con `smistatore`, ha due porte: la prima (`dati`) attraverso la quale riceve messaggi dal `produttore` (e che, per quanto già detto, fornisce anche il buffer dei messaggi), la seconda (`pronto`) a cui ciascun `consumatore`, quando è pronto a

Figura 8.6 Singolo produttore/singolo consumatore.

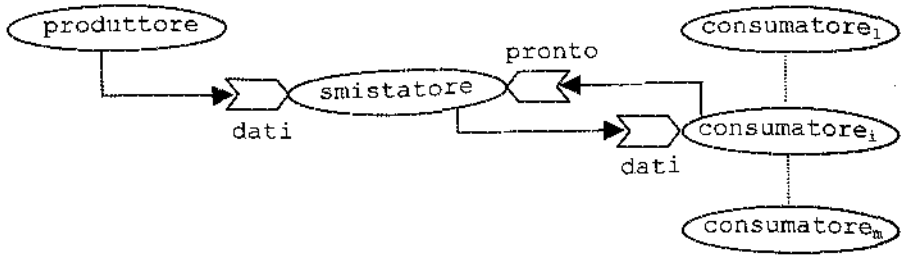


Figura 8.7 Produttore con più consumatori.

ricevere messaggi, invia un segnale per indicare la propria disponibilità. Il server è costituito da un ciclo senza fine al cui interno attende di ricevere un segnale da un qualunque `consumatore`, quindi attende un messaggio dal `produttore` e lo invia al `consumatore` che si è dichiarato pronto.

Nel programma, abbiamo indicato con `T` il generico tipo dei messaggi inviati dal produttore e quindi anche il tipo della porta `dati`. Ogni consumatore dovrà inoltre dichiarare una propria porta, che abbiamo ancora indicato con l'identificatore `dati` per ricevere i messaggi dal server. Il produttore quando vuole inviare un messaggio si limita a inviarlo sulla porta `dati` del server `smistatore`. Ogni consumatore quando desidera ricevere un messaggio dal produttore, invia un segnale sulla porta `pronto` del server e quindi si mette in attesa di ricevere il messaggio sulla propria porta `dati`.

Il seguente programma, relativo al server `smistatore`, rimarrebbe inalterato anche qualora fossero presenti più processi produttori:

```
process smistatore{
   port T dati;
   port signal pronto;
   T messaggio;
   process prod, cons;
   signal s;
   while (true) {
      cons=receive(s)from pronto;
      prod=receive(messaggio)from dati;
      send(messaggio)to cons.dati;
   }
}
```

L'altro caso in cui è necessario un server è quello in cui, anche in presenza di un solo produttore e di un solo consumatore, imponiamo il vincolo che siano al più `N` messaggi già inviati ma non ancora ricevuti. Questo vincolo equivale a imporre un limite superiore alla dimensione del buffer dei messaggi. Per garantire ciò è necessario che il produttore non possa inviare un nuovo messaggio se i precedenti `N` non sono stati ancora ricevuti. Quindi il produttore, prima di inviare un messaggio, deve chiedere (al server) se la precedente condizione è vera, rimanendo in attesa in caso contrario. Questo criterio vale anche in presenza di più produttori e di più consum

tori. Nella figura 8.8 è riportato lo schema del protocollo di comunicazione fra produttori, consumatori e server che, in questo caso, implementa le funzionalità di una mailbox, cioè di un canale da molti a molti e di dimensioni finite (N). In particolare, ogni produttore, quando è pronto a inviare un messaggio, invia un segnale al server tramite la porta `pronto_prod`, quindi si pone in attesa di un segnale dal server tramite la sua porta `ok_to_send`. Alla ricezione di questo segnale il produttore può infine inviare il messaggio sulla porta `dati` del server. Il server, per conoscere il numero di messaggi presenti nel buffer della porta `dati`, gestisce un `contatore`; quando il valore di tale `contatore` è minore di N, il server può ricevere il segnale dal produttore e quindi rispondergli mediante il segnale sulla porta `ok_to_send`, incrementando il `contatore` per indicare che da ora in poi ci sarà un messaggio in più nel buffer. Ogni consumatore, a sua volta, quando è pronto a ricevere un messaggio invia un segnale al server tramite la porta `pronto_cons` e quindi si pone in attesa di ricevere il messaggio tramite la propria porta `dati`. Quando il `contatore` ha un valore maggiore di zero, il server può ricevere il segnale dal consumatore e, una volta decrementato il `contatore`, ricevere un messaggio presente nella porta `dati` inviandolo successivamente alla porta `dati` del consumatore stesso.

In base alle precedenti considerazioni, riportiamo di seguito il codice del processo mailbox:

```
process mailbox{
   port T dati;
   port signal pronto_prod, pronto_cons;
   T messaggio;
   process prod,cons;
   signal s;
   int contatore=0;
   do
   [] (contatore<N);prod=receive(s)from pronto_prod; ->
      contatore ++;
      send(s)to prod.ok_to_send;
   [] (contatore>0);cons:=receive(s)from pronto_cons; ->
      prod=receive(messaggio)from dati;
      contatore-;
      send(messaggio)to cons.dati;
   od;
}
```

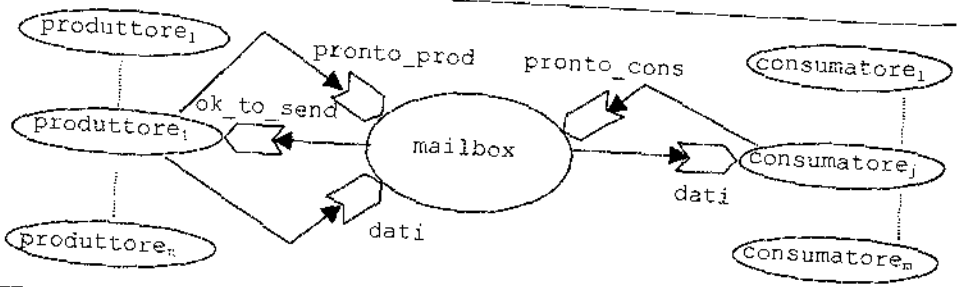


Figura 8.8 Mailbox di dimensioni finite.

Di seguito vengono riportati anche i codici relativi al generico produttore e al generico consumatore:

```
process produttorei{
   port signal ok_to_send;
   T messaggio;
   process p;
   signal s;
      .........
   <produci il messaggio>;
   send(s)to mailbox.pronto_prod;
   p=receive(s)from ok_to_send;
   send(messaggio)to mailbox.dati;
      .........
}

process consumatorej{
   port T dati;
   T messaggio;
   process p;
   signal s;
   .........
send(s)to mailbox.pronto_cons;
p=receive(messaggio)from dati;
<consuma il messaggio>;
   .........
}
```

8.3 Specifica di strategie di priorità

Come abbiamo visto nei precedenti paragrafi, l'uso dei comandi con guardia semplifica drasticamente la scrittura di processi servitori quando, in particolare, questi offrono più servizi ai processi clienti. Questa semplicità di uso deriva sostanzialmente da due caratteristiche:

1. la possibilità di specificare una condizione di sincronizzazione in una guardia, abilitando quindi il server a ricevere una richiesta di servizio soltanto quando è in grado di servirla. Senza questa possibilità il servitore potrebbe ricevere una richiesta anche quando non è in grado di poterla servire.
2. il non determinismo associato alla scelta del ramo da eseguire fra tutti quelli che hanno la guardia valida.

Ci sono casi, però, in cui queste caratteristiche complicano, invece di semplificare, la soluzione ai problemi di sincronizzazione. Ciò si verifica tutte le volte che, nello scrivere il codice di un processo servitore, è necessario implementare una particolare strategia di gestione delle richieste di servizio. In questi casi, infatti, la scelta non deterministica fra varie alternative è in contrasto con qualunque criterio di scelta prioritaria. Per questo motivo, è stata proposta una variante dei comandi con guardia in cui la scelta del ramo da eseguire, fra quelli con guardia valida, non viene effettuata in maniera non deterministica ma in base all'ordine con cui i vari rami, dell'istruzione ripetitiva o alternativa, sono scritti nel testo. In pratica viene scelto il primo ramo

con guardia valida scandendo i rami dal primo all'ultimo (dall'alto verso il basso). Se è presente questa alternativa, per implementare una strategia di gestione prioritaria delle richieste di servizio è necessario scrivere i vari rami che compongono l'istruzione ripetitiva di un processo servitore in modo tale che l'ordine con cui i rami vengono scritti rispetti il criterio di priorità. Però non è sempre possibile, o comunque semplice, tradurre il criterio di priorità da implementare nell'ordine con cui scrivere i rami dell'istruzione ripetitiva. Vediamo quindi un criterio generale che consente di implementare una strategia di gestione delle richieste di servizio anche in assenza della precedente variante dei comandi con guardia.

Per semplicità, faremo riferimento a un esempio particolare, anche se il criterio descritto è del tutto generale e applicabile, quindi, a un qualunque altro caso di gestione su base prioritaria. Riprendiamo, in particolare, il caso del processo `server` che gestisce un pool di risorse equivalenti, visto nel precedente paragrafo, ma con l'aggiunta del seguente criterio: supponiamo che i processi P_0, P_1, ..., P_{M-1} siano gli `M` processi clienti e che il `server`, nel gestire le loro richieste, debba privilegiare le richieste di P_0 rispetto a quelle di P_1 e queste rispetto a quelle di P_2 e così via. In pratica, ciò significa che quando un qualunque processo P_i invia al `server` una richiesta, questa deve essere ovviamente servita se ci sono risorse disponibili mentre, in caso contrario, P_i deve rimanere sospeso. Quando poi una risorsa viene rilasciata, il `server` deve essere in grado di conoscere se ci sono processi clienti sospesi e, in questo caso, quali sono. Infatti il `server` deve scegliere fra i processi sospesi quello a cui assegnare la risorsa, che deve essere quello con la priorità più alta (nel nostro esempio quello con indice più basso).

Con riferimento alla soluzione vista nel precedente paragrafo, resta inalterato lo schema di ciascun processo cliente che, una volta inviato al `server` un segnale sulla porta `richiesta`, resta in attesa di ricevere l'indice della risorsa che gli è stata allocata tramite propria porta `risorsa`. Quindi, tutti i processi clienti, che in un certo istante sono sospesi, sono quelli che, inviato il segnale di richiesta, sono bloccati nell'esecuzione della primitiva `receive` sulla propria porta `risorsa`. Purtroppo non è invece più valido lo schema seguito per il processo `server`. Infatti, avendo specificato nel primo ramo dell'istruzione ripetitiva la condizione di sincronizzazione `(disponibili>0)`, il `server` non può ricevere richieste quando non è in grado di servirle e quindi non potrà mai essere in grado di conoscere chi fra i propri clienti è sospeso. Per garantire al `server` questa conoscenza è necessario che lo stesso riceva le richieste di servizio anche quando non è in grado di servirle memorizzando in una struttura dati locale il nome del cliente la cui richiesta non è stata servita. In questo modo, al successivo rilascio è possibile sapere quali clienti sono sospesi e fra questi scegliere quello con priorità più alta. Possiamo quindi utilizzare ancora i comandi con guardia senza però specificare la condizione di sincronizzazione:

```
process server{
   port signal richiesta;
   port int rilascio;
   int disponibili=N;
   boolean libera[N];
   process p ;
   signal s;
   int r;
   int sospesi=0;boolean bloccato[M];
   process client[M];
      //inizializzazione
      for(int i=0;i<N;i++) libera[i]=true;
      for(int j=0;j<M;j++) bloccato[j]=false;
      client[0]="P0";.....client[M-1]="PM-1";}
   do
   [] p=receive(s)from richiesta; ->
         if(disponibili>0){
            int i=0;
            while(!libera[i])i++;
            libera[i]=false;
            disponibili--;
            send(i)to p.risorsa;}
         else{ int j=0;
               sospesi++;
               while(client[j]!=p) j++;
               bloccato[j]=true;}
   [] p:=receive (r) from rilascio: ->
         if(sospesi==0){
            disponibili++;
            libera[r]=true;}
         else{
            int i=0;
            while(!bloccato[i]) i++;
            sospesi--;
            bloccato[i]=false;
            send(r)to client[i].risorsa;}
   od;
}
```

Rispetto all'esempio del `server` visto nel precedente paragrafo, abbiamo aggiunt nella struttura dati l'intero `sospesi`, che serve per indicare il numero di proces clienti in attesa di ricevere risposta alla loro richiesta, e il vettore `bloccato`, ch serve per indicare chi, fra gli `M` clienti, è sospeso e chi non lo è. Nella fase di inizia lizzazione, vengono inseriti nel vettore `client` i nomi dei processi clienti in bas all'ordine di priorità con cui vengono gestiti. Nel corpo del `server` abbiamo elim nato la condizione di sincronizzazione dal primo ramo dell'istruzione ripetitiva: i questo modo il `server` può ricevere qualunque richiesta. Una volta ricevuta, per viene eseguito il corpo del ramo, che è costituito da un `if-else`: se la richiesta esaudibile, viene ricercata la prima risorsa disponibile il cui indice viene restituito cliente, altrimenti il `server` registra che il cliente, il cui nome è stato restituito da la primitiva `receive`, rimane sospeso. Quindi anche il secondo ramo, relativo all ricezione di un rilascio, è costituito da un `if-else`. In particolare, se al rilasci non ci sono processi sospesi, la risorsa rilasciata viene registrata come di nuovo d sponibile. In caso contrario, viene ricercato, fra i clienti sospesi, quello a priorità pi alta a cui viene inviato l'indice della risorsa rilasciata abilitandolo quindi a riprend re la sua esecuzione.

Come risulta dall'esempio, quando è necessario specificare una particolare politica di allocazione, la convenienza a usare comandi con guardia diminuisce in quanto la struttura del server è molto simile a quella che avremmo ottenuto senza i comandi con guardia.

8.4 Realizzazione delle primitive asincrone

Nei precedenti paragrafi abbiamo supposto di disporre, a livello di linguaggio di programmazione, delle due primitive di comunicazione e del costrutto `port` per definire i canali utilizzati dalle stesse primitive. Vediamo adesso come implementare tali meccanismi linguistici utilizzando gli strumenti di comunicazione offerti direttamente dalla macchina concorrente.

Come indicato nel paragrafo 7.2, le primitive asincrone sono sicuramente quelle più semplici e di "basso livello" tra le tre tipologie di meccanismi di comunicazione che vedremo. In effetti, nella maggior parte dei sistemi operativi che sono stati realizzati adottando il modello a scambio di messaggi il meccanismo primitivo offerto dal nucleo è proprio un meccanismo di tipo asincrono. In questi casi, la traduzione del meccanismo linguistico da parte del compilatore in termini del meccanismo primitivo offerto dalla macchina concorrente è molto semplice in quanto i meccanismi linguistici trovano un diretto supporto da parte dei meccanismi primitivi del sistema. In questo paragrafo ci limiteremo, quindi, a illustrare come il meccanismo di comunicazione asincrona possa essere realizzato a livello del nucleo del sistema. Per questo, distingueremo tre casi relativi alle tre diverse tipologie di architetture fisiche su cui il nucleo viene realizzato: architetture monoelaboratore, architetture multielaboratore e architetture fisicamente distribuite.

8.4.1 Architetture monoelaboratore e multielaboratore

Relativamente alle architetture monoelaboratore, per quanto riguarda le strutture dati e le funzioni del nucleo che sono a supporto della concorrenza, faremo riferimento a quelle già introdotte nel paragrafo 3.6.

Supponiamo, quindi, di voler realizzare nel nucleo un meccanismo di comunicazione asincrono. Procederemo in modo analogo a quanto visto nel paragrafo 5.8.1 dove, nel definire la realizzazione del meccanismo semaforico, abbiamo esteso le strutture dati e le funzioni, precedentemente introdotte nel paragrafo 3.6, con le strutture dati per rappresentare i semafori e le funzioni primitive che su esse operano. Adesso, dovendo realizzare il supporto per una macchina concorrente strutturata secondo il modello a scambio di messaggi, estenderemo le strutture dati e le funzioni, introdotte nel paragrafo 3.6, con le strutture dati necessarie per fornire il supporto ai canali di comunicazione e le funzioni primitive che su essi operano.

Per specificare il meccanismo di comunicazione primitivo da realizzare facciamo le seguenti ipotesi semplificative:

a) tutti i messaggi che vengono scambiati fra i processi sono di un unico tipo `T` predefinito a livello di nucleo, per esempio una stringa di byte di dimensione fissa. Sarà poi compito del compilatore tradurre un messaggio di un qualunque tipo, definito tramite il linguaggio di alto livello, in uno o più messaggi del tipo primitivo `T`;

b) manteniamo l'ipotesi che tutti i canali siano da molti a uno (`port`) e quindi associati al processo ricevente;

c) essendo le primitive asincrone, per quanto visto precedentemente, ogni porta deve contenere un buffer (coda di messaggi) di lunghezza indefinita, in modo che non possa mai essere pieno.

Per garantire quanto previsto dal precedente punto c) realizzeremo la coda di messaggi associata a una porta mediante una lista. Definiamo, perciò, la struttura `messaggio` in modo tale che questa rappresenti il generico elemento di una lista messaggi:

```
typedef struct {
   T informazione;
   PID mittente;
   messaggio * successivo;
} messaggio;
```

dove il campo `informazione` rappresenta il contenuto informativo del messaggio (del tipo predefinito tipo `T`), il campo `mittente` è destinato a contenere il PID (nome) del processo che invia il messaggio (il tipo `PID` è stato definito nel paragrafo 3.6), il campo `successivo` è di tipo puntatore a messaggio e serve per concatenare un messaggio, una volta inviato, nella coda associata alla porta. Possiamo poi implementare la struttura `coda-di-messaggi` che realizza la coda come una lista di elementi di tipo `messaggio`:

```
typedef struct {
   messaggio * primo;
   messaggio * ultimo;
   } coda_di_messaggi;
```

e le funzioni per inserire un elemento nella lista (`inserisci`), per togliere un elemento da una lista non vuota (`estrai`) e la funzione booleana `coda_vuota` che restituisce `true` se la lista non contiene elementi:

```
void inserisci(messaggio * m,coda_di_messaggi c)
   if(c.primo==null)c.primo=m;
   else c.ultimo -> successivo=m;
   c.ultimo=m;
   m -> successivo=null;
}
messaggio * estrai (coda_di_messaggi c) {
   messaggio * pun;
   pun=c.primo;
   c.primo=c.primo -> successivo;
   if(c.primo==null) c.ultimo=null;
   return pun;
}
boolean coda_vuota(coda_di_messaggi c) {
   if (c.primo==null) return true;
   return false;
}
```

Possiamo adesso definire il descrittore di una porta (des_porta), cioè la struttura che la rappresenta, contenente due campi: una coda_di_messaggi e un puntatore al descrittore di un'altra porta che può essere utilizzato per concatenare i descrittori delle porte all'interno di una lista:

```
typedef struct {
   coda_di_messaggi coda;
   p_porta puntatore;
} des_porta;
```

dove, quindi, il tipo puntatore p_porta può essere definito come:

```
typedef des_porta * p_porta
```

Poiché, come indicato nel precedente punto b), ogni porta è associata a un processo (l'unico che da essa può ricevere), modifichiamo leggermente la struttura dati descrittore_processo, già vista nel paragrafo 3.6:

```
typedef struct {
   p_porta porte_processo[M];
   PID nome;
   modalità_di_servizio servizio;
   tipo_contesto contesto;
   tipo_stato stato;
   PID padre;
   int N_figli;
   des_figlio prole[max_figli];
   p_des successivo;
} des_processo;
```

Per evidenziare quali sono le porte associate al processo, abbiamo aggiunto il nuovo campo porte_processo[M], un vettore di puntatori ai descrittori di tali porte. La costante M rappresenta il massimo numero di porte che un processo può avere. Con queste indicazioni, dovremo modificare anche la primitiva fork relativa alla creazione di un processo in quanto, per esempio, per creare un processo a cui siano associate due porte, è necessario, oltre a quanto già visto nel paragrafo 3.6, anche ricercare due descrittori di porte disponibili e inserire i puntatori a tali descrittori nel campo porte_processo del descrittore del processo che viene creato. Per conoscere quali siano i descrittori delle porte disponibili può essere gestita una lista porte_libere, del tutto analoga alla lista descrittori_liberi dei descrittori di processo già vista nel paragrafo 3.6. In tal modo, all'atto di una fork, verrebbero ricercate le porte disponibili, da assegnare al processo, che sarebbero successivamente indicate come disponibili all'atto della terminazione del processo, stesso. Per semplicità evitiamo di riportare il codice delle primitive fork e quit così modificate.

Nel descrittore di un processo possiamo infine caratterizzare meglio il tipo stato_processo per tener conto che, con le primitive asincrone, l'unica causa di sospensione di un processo è relativa al tentativo di ricevere un messaggio da una porta vuota. Per questo motivo potremmo definire tipo_stato ancora come una enumerazione di valori che consentano di evidenziare se un processo è in stato attivo (*pronto* o *esecuzione*) oppure sospeso e, in questo caso, su quale porta attende di ricevere messaggi. Come vedremo, per implementare un comando con guardie dob biamo però prevedere anche la possibilità che un processo sia sospeso su un insiem di porte, in attesa che arrivi un messaggio su una qualunque di queste; questo acc de, per esempio, quando tutte le guardie non fallite sono guardie ritardate. Infatti, i questo caso, il processo si sospende in attesa che un messaggio arrivi su una qualu que delle porte su cui sono state specificate le funzioni di ricezione presenti nel guardie ritardate. Un modo molto semplice per rappresentare il tipo stato_pro cesso potrebbe essere tramite un vettore di M bit. In questo caso potremmo, p esempio, indicare che un processo è in stato attivo quando tutti i bit del camp stato hanno il valore zero. Analogamente, un processo sospeso in attesa di u messaggio dalla porta di indice 3 (indice nel vettore porte_processo) verrebb indicato azzerando tutti i bit del campo stato meno quello di indice 3, che dovrel be avere il valore uno. Infine, se avessero valore uno i bit del campo stato di ind ce 2 e 5, ciò significherebbe che il processo è sospeso in attesa che arrivi un messa gio, o dalla sua porta di indice 3 o da quella di indice 5.

Utilizzando quindi il campo stato del descrittore di un processo così defini possiamo specificare le seguenti funzioni per testare lo stato di un processo o per r gistrare una commutazione di tale stato del processo in esecuzione:

```
boolean bloccato_su(p_des p, int ip) {
   <testa il campo stato nel descrittore del processo di cui p è il puntatore e restituisce il valo
   true se il processo risulta bloccato in attesa di ricevere messaggi dalla porta il cui indice n
   campo porte_processo è ip>;
}
void blocca_su(int ip) {
   <modifica il campo stato del descrittore del processo_in_esecuzione per indica
   che lo stesso si blocca in attesa di messaggi dalla porta il cui indice nel campo porte_pro
   cesso è ip>;
}
```

Prima di definire la funzioni primitive offerte dal nucleo della macchina concorrent definiamo due semplici funzioni di libreria da utilizzare, in un ipotetico linguagg di basso livello, per abilitare processi a inviare o ricevere messaggi e realizzate tr mite opportune chiamate alle primitive del nucleo. Saranno queste le funzioni che compilatore utilizzerà per tradurre le send e receive di alto livello definite n paragrafi precedenti.

Per chiarire meglio le cose, vediamo come nei vari livelli vengono identificat processi e le porte. Come abbiamo visto nei precedenti paragrafi, un processo in programma scritto in un linguaggio di alto livello viene identificato mediante il su nome (un identificatore simbolico), che rappresenta un valore del tipo proces predefinito dal linguaggio. Viceversa, a livello di funzioni di libreria, un proces viene identificato mediante il suo PID e, infine, a livello di nucleo, mediante il pu tatore al suo descrittore (un valore del tipo p_des). Analogamente, una porta ad to livello viene denotata mediate il suo identificatore simbolico definito tramite il c strutto port; a livello di libreria viene identificata mediante un intero, che rappr senta l'indice nel vettore delle porte locali al processo ricevente (porte_proces so). Infine, a livello di nucleo, ogni porta viene identificata mediante il puntatore suo descrittore (un valore del tipo p_porta). Con queste indicazioni, possian

specificare le due funzioni di libreria da utilizzare, rispettivamente, per inviare o ricevere un messaggio del tipo predefinito T:

```
void send (T inf, PID proc, int ip);
void receive (T &inf, PID &proc, int ip);
```

La prima ha tre parametri: `inf` è l'informazione di tipo `T` da inviare, `proc` il `PID` del processo a cui inviarla, `ip` l'indice della porta locale a `proc` e a cui indirizzare il messaggio. Anche la seconda ha tre parametri: `inf` è il riferimento a una variabile di tipo `T`, alla quale assegnare l'informazione ricevuta; `proc` il riferimento a una variabile di tipo `PID`, alla quale assegnare il nome del mittente; `ip` l'indice della porta da cui ricevere.

Come si può notare, la traduzione di una `send` di alto livello nella corrispondente `send` di libreria è del tutto ovvia, a parte la codifica del tipo del messaggio in termini del tipo predefinito `T`; altrettanto vale per la `receive`.

Per realizzare le due funzioni di libreria definiamo adesso tre primitive di nucleo:

- la primitiva `testa_porta`, per testare se su una porta sono presenti messaggi, bloccando, in caso contrario, il processo che la esegue;
- la primitiva `estrai_da_porta`, per estrarre un messaggio dalla coda associata a una porta (coda non vuota), restituendone il puntatore;
- la primitiva `inserisci_porta`, per inserire un messaggio nella coda di messaggi associata a una porta.

Vediamo adesso la prima delle tre primitive:

```
void testa_porta (int ip){
   p_des esec=processo_in_esecuzione;
   p_porta pr=esec->porte_processo[ip];
   if(coda_vuota(pr->coda)) {
      blocca_su(ip);
      assegnazione_CPU;
   }
}
```

Alla primitiva viene passato l'indice `ip` della porta da testare. Per prima cosa si ricava il puntatore `esec` al descrittore del processo in esecuzione e, tramite questo, il puntatore `pr` al descrittore della porta da testare. Se la coda associata alla porta è vuota, lo stato del processo in esecuzione viene modificato, per indicare che lo stesso si blocca in attesa di messaggi sulla porta di indice `ip`. Quindi, eseguendo `assegnazione_CPU`, viene schedulato un altro processo in modo tale che, quando la primitiva termina, si abbia una commutazione di contesto.

Di seguito viene riportato il codice della seconda primitiva:

```
messaggio *estrai_da_porta (int ip) {
   messaggio *m;
   p_des esec=processo_in_esecuzione;
   p_porta pr=esec->porte_processo[ip];
   m=estrai(pr->coda);
   return m;
}
```

Anche in questo caso si ricava il puntatore `esec` al descrittore del processo in esecuzione e, tramite questo, il puntatore `pr` al descrittore della porta da cui estrarre il messaggio, il cui puntatore viene poi restituito dalla primitiva.

Riportiamo, infine, il codice dell'ultima primitiva:

```
void inserisci_porta (messaggio *m, PID proc, int ip){
   p_des destinatario=descrittore(proc);
   p_porta pr=destinatario->porte_processo[ip];
   inserisci(m, pr->coda);
   if (bloccato_su(destinatario, ip)) attiva(destinatario);
}
```

Il parametro `m` denota il puntatore al messaggio da inserire nella coda della porta di indice `ip` del processo, il cui `PID` è passato tramite il parametro `proc`. Per prima cosa, mediante la funzione `descrittore` vista nel paragrafo 3.6, noto il `PID` del processo `destinatario`, si ricava il puntatore al suo descrittore e, tramite questo noto l'indice `ip` della porta nella quale inserire il messaggio, si ricava il puntatore `pr` al descrittore della porta stessa. Quindi, mediante la funzione `inserisci` vista prima, il messaggio viene inserito nella coda associata alla porta. Infine, se il processo `destinatario` è nello stato sospeso in attesa di messaggi sulla porta di indice `ip`, il processo può essere riattivato mediante la funzione `attiva` vista nel paragrafo 3.6.

Tramite queste tre primitive di nucleo è adesso molto semplice dettagliare il corpo delle due funzioni di libreria prima definite:

```
void send (T inf,PID proc,int ip){
   messaggio *m=new messaggio;
   m->informazione=inf;
   m->mittente=PIE();
   inserisci_porta(m,proc,ip);
}

void receive (T &inf, PID &proc, int ip){
   messaggio *m;
   testa_porta(ip);
   m=estra_da_porta(ip);
   proc=m->mittente;
   inf=m->informazione;
}
```

La funzione `send` alloca in memoria un nuovo messaggio e ne riempie i campi `informazione` e `mittente` (quest'ultimo tramite la primitiva `PIE()`, vista nel paragrafo 3.6, che restituisce il `PID` del processo in esecuzione). Quindi, mediante la primitiva `inserisci_porta`, il messaggio così compilato viene inserito nella porta di indice `ip` del processo destinatario `proc`.

La funzione `receive` testa la porta da cui ricevere e, se la porta è vuota, blocca il processo. In caso contrario (e quando il processo verrà riattivato dopo una `send`) viene estratto un messaggio dalla porta e i valori relativi all'informazione e al nome del mittente, contenuti nel messaggio, vengono assegnati alle variabili che il ricevente ha passato come parametri.

Come si può notare, la funzione `receive` non è atomica, in quanto la sua esecuzione comporta quella in sequenza di due diverse primitive di nucleo. Come tale, fra la terminazione della prima e l'inizio della seconda, la funzione può essere interrotta e, a causa di ciò, può andare in esecuzione un altro processo. Questo evento, in particolare, avviene sicuramente quando la prima delle due primitive (la `testa_porta`) comporta una commutazione di contesto. Questa perdita di atomicità, come è noto, può comportare delle interferenze. In questo caso però la garanzia che queste non si possono verificare è data dal fatto che, su una porta, può operare con la `receive` un solo processo. Se questo non fosse vero potrebbero veramente sorgere problemi, come per esempio nel caso in cui un processo, che si sospende poiché ha trovato la porta vuota, viene riattivato in seguito a una `send` ma, prima di essere schedulato e poter estrarre il messaggio dalla porta, viene scavalcato da un secondo processo ricevente che preleva il messaggio prima di lui.

Per implementare un comando con guardia è utile disporre anche di una terza funzione di libreria, che consenta a un processo di ricevere un messaggio non da una specifica porta ma da una qualunque fra un insieme di porte. Per esempio, possiamo definire la seguente funzione:

```
int receive_any(T &inf,PID &proc,int ip[],int n);
```

la cui specifica è la seguente: i primi due parametri sono gli stessi della funzione `receive` ma, invece di passare come terzo parametro l'indice di una porta da cui ricevere un messaggio, si passa un vettore di indici di porte, vettore la cui dimensione è `n` (quarto parametro). La funzione verifica lo stato di tutte le porte, gli indici delle quali sono passati tramite il vettore `ip`. Se una o più porte contengono messaggi, ne viene scelta una qualunque (per esempio la prima che si trova) e viene quindi estratto il messaggio da tale porta, restituendo al processo ricevente l'informazione `inf` e il nome del processo mittente `proc`; la funzione termina restituendo l'indice della porta scelta da cui è stato ricevuto il messaggio. Se tutte le porte sono vuote la funzione sospende il processo tramite una commutazione di contesto. Quando, successivamente, un messaggio viene inviato a una delle precedenti porte, la funzione riprende la sua esecuzione estraendo il messaggio da tale porta e restituendo il suo indice. Utilizzando questa funzione è molto semplice implemetare un comando con guardie. In pratica il compilatore di un linguaggio ad alto livello, valutate tutte le condizioni di sincronizzazione presenti nelle guardie, invoca la funzione `receive_any` passando, come terzo parametro, il vettore contenente gli indici delle porte su cui devono essere eseguite le `receive` presenti nei rami con guardie non fallite. L'indice della porta restituito dalla funzione `receive_any` può essere poi utilizzato per selezionare il ramo il cui corpo deve essere eseguito dopo la ricezione del messaggio.

Per implementare la funzione `receive_any` è necessario prevedere anche una quarta primitiva di nucleo che, come la `testa_porta` già vista, può essere utilizzata per testare non una, ma tutte le porte di un insieme:

```
int testa_porte(int ip[],int n){
   p_porta pr,int ris=-1;int indice_porta;
   p_des esec=processo_in_esecuzione;
   for (int i=0;i<n;i++){
      indice_porta=ip[i];
         pr=esec->porte_processo[indice_porta];
         if(coda_vuota(pr->coda))blocca_su(indice_porta);
         else{
            ris=indice_porta;
            esec->stato=<processo attivo>;
            break;
         }
   if(ris==-1)assegnazione_CPU;
   return ris;
}
```

Alla primitiva viene passato il vettore di `n` indici di porte, ciascuna delle quali vie testata per verificare se contiene messaggi. Per ogni porta si ricava, dal puntatore descrittore del processo in esecuzione, tramite il campo `porte_processo`, n l'indice della porta, il puntatore al descrittore della porta stessa e quindi si testa se sua coda è vuota. Nel caso in cui sia vuota, si indica che il processo si sospenderà attesa di messaggi da tale porta e si procede con il test della porta successiva. Alla ne del ciclo, se tutte le porte sono state trovate vuote, lo stato del processo risult "processo sospeso su tutte le porte controllate", cioè è in attesa di un messaggio una qualunque di esse. Quindi, poiché la variabile locale `ris` inizializzata al val `-1` non è stata modificata, la primitiva termina eseguendo una `assegnazione_CPU` (e cioè una commutazione di contesto) e restituisce il valore `-1`. Se pe durante il ciclo `for`, una porta viene trovata non vuota, si interrompe l'esecuzic del ciclo, si assegna a `ris` l'indice della porta trovata (che verrà poi restituito da primitiva) e si resetta il campo stato del processo come "stato attivo"; la funzic termina quindi restituendo l'indice della porta trovata, senza eseguire nessuna co mutazione di contesto. Mediante questa primitiva possiamo, infine, descrivere il c po della funzione `receive_any`:

```
int receive_any(T &inf,PID &proc,int ip[],int n){
   messaggio * mes; int indice_porta;
   do
      indice_porta=testa_porte(ip,n);
   while(indice_porta==-1);
   mes=estrai_da_porta(indice_porta);
   proc=mes->mittente;
   inf=mes->informazione;
   return indice_porta;
}
```

La funzione si limita a chiamare la precedente primitiva all'interno di un ciclo `while`. Se la primitiva restituisce un valore diverso da `-1` (senza quindi bloccare processo), la funzione esce subito dal ciclo e prosegue estraendo un messaggio da porta, il cui indice è stato restituito dalla primitiva, e quindi assegnando ai parame `inf` e `proc` i valori, rispettivamente, del campo informativo e del nome del mitten contenuti nel messaggio ricevuto. Se la primitiva blocca il processo, essa restituisc valore `-1`. In questo caso, quando il processo verrà riattivato (e ciò, per quanto vi precedentemente, avviene alla fine di una `send` su una delle porte indicate come

rametri) la funzione esegue di nuovo il ciclo ma, stavolta, con la certezza di uscirne subito, poiché la porta su cui il messaggio è arrivato è ovviamente non vuota.

La realizzazione del meccanismo di comunicazione asincrona, vista fino a ora per un'architettura monoelaboratore, resta valida anche per architetture multielaboratore, fatti salvi i meccanismi per garantire l'atomicità delle primitive del nucleo, già visti nel capitolo 5, e quelli necessari per garantire gli accessi esclusivi alle strutture dati del nucleo, da parte di processi che girano contemporaneamente su processori diversi.

8.4.2 Architetture distribuite

In questo paragrafo, col termine *architettura distribuita* faremo riferimento a un insieme di architetture fisiche caratterizzate dalla presenza di più calcolatori interconnessi tra loro al fine di costituire un unico sistema complessivo e dove, a differenza delle architetture multielaboratore, non esiste nessuna memoria condivisa. Per esempio, una rete locale di calcolatori connessi tramite una rete Ethernet rientra in questa categoria. Le architetture distribuite costituiscono il supporto fisico per la realizzazione di quei sistemi di elaborazione noti col termine di *sistemi distribuiti*. Pur non esistendo una chiara definizione che caratterizzi in maniera precisa la struttura, l'organizzazione interna e le proprietà di un sistema distribuito, possiamo genericamente caratterizzare un sistema di questo tipo come "un insieme di calcolatori (*nodi*) che a un qualunque utente possa apparire come un unico e coerente sistema di elaborazione". Per un più approfondito esame di questo tipo di sistemi, che esula dagli scopi del testo, si rimanda a [70].

Indipendentemente, quindi, dalla struttura fisica del sistema, ci preme mettere in evidenza, di nuovo, la struttura astratta, così come viene percepita dall'utente e che, come anche per gli altri tipi di architetture fisiche, è costituita, oltre che dai componenti hardware, anche dai componenti software offerti dal sistema operativo. Da questo punto di vista, in letteratura si tende a distinguere fra due diverse tipologie di sistemi operativi: i sistemi operativi distribuiti (DOS – *Distributed Operating Systems*) e i sistemi operativi di rete (NOS – *Network Operating Systems*). I primi sono caratterizzati da un insieme di nodi tra loro omogenei e tutti dotati dello stesso sistema operativo, in particolare dello stesso nucleo. Sono quindi sistemi strettamente connessi e per i quali le funzionalità del nucleo sono concettualmente le stesse che abbiamo già visto relativamente al nucleo di un sistema mono o multielaboratore, anche se ripetute per ciascuno dei nodi della rete. Scopo del sistema è quindi quello di gestire tutte le risorse nascondendo all'utente la distribuzione delle stesse sulla rete. Nel caso dei NOS, viceversa, siamo spesso in presenza di nodi eterogenei e, al limite, anche con sistemi operativi diversi e autonomi, nodo per nodo. Sono quindi sistemi lascamente connessi dove però ogni nodo della rete è in grado di offrire servizi a clienti remoti presenti su altri nodi della stessa. Per questo motivo, ogni sistema operativo possiede appositi meccanismi per fornire servizi di rete come, per esempio, il meccanismo delle socket [66]. In questo caso, la trasparenza della distribuzione delle risorse non viene ottenuta tramite il nucleo del sistema operativo, ma per mezzo di uno strato di software (*middleware*), che viene interposto su ogni nodo tra il sistema operativo e le applicazioni che girano sul nodo stesso (vedi

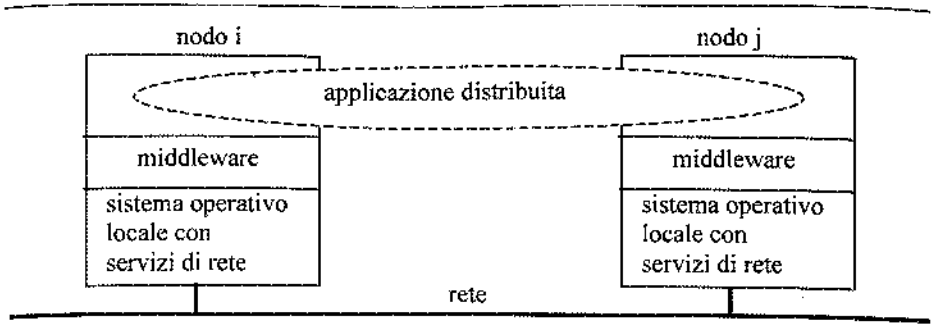


Figura 8.9 Network Operating System.

figura 8.9). Un esempio di funzionalità offerte dal middleware sarà illustrato nel capitolo 10 quando verrà esaminato il meccanismo delle chiamate di procedure remote (RPC – *Remote Procedure Call*).

Poiché lo scopo di questo paragrafo è quello di illustrare una possibile implementazione delle primitive di comunicazione asincrona come meccanismo offerto direttamente dal nucleo del sistema, faremo riferimento al caso di un sistema operativo distribuito (DOS). Quindi, faremo l'ipotesi di una rete di nodi dove ogni nodo rappresenta un calcolatore sul quale gira la stessa copia del sistema operativo e, in particolare, la stessa copia del nucleo del sistema. Come vedremo, il nucleo deve offrire le stesse primitive che abbiamo esaminato nel precedente paragrafo, oltre ad alcune ulteriori funzioni per la gestione delle interfacce di rete.

Fra le componenti che è necessario aggiungere al nucleo, visto nel precedente paragrafo, vi sono sicuramente le funzioni dedicate alla gestione delle interfacce di rete. Senza entrare in eccessivi dettagli hardware, che esulano dai nostri scopi, possiamo schematizzare in maniera molto sintetica un'interfaccia di rete come indicato nella figura 8.10.

In pratica ogni interfaccia è composta da due parti distinte, indicate nella figura come *canale di trasmissione* e *canale di ricezione*, utilizzate rispettivamente per trasmettere o ricevere pacchetti. La struttura del pacchetto dipende ovviamente da particolare tipo di rete fisica. In pratica, i due canali sono assimilabili a due qualunque dispositivi d'ingresso/uscita, che quindi generano un'interruzione quando termina l'operazione per cui sono stati attivati. Così, il canale di trasmissione, una volta attivato per trasmettere un pacchetto sulla rete, genera un'interruzione alla fine della trasmissione per indicare che è disponibile alla successiva trasmissione. Analogamente, il canale di ricezione lancia un'interruzione alla fine della ricezione di un pacchetto indirizzato al nodo su cui risiede l'interfaccia.

Faremo inoltre l'ipotesi che su ciascuno dei due canali sia disponibile un registro buffer della stessa dimensione di un pacchetto. Nel registro buffer del canale di trasmissione deve essere preventivamente trasferito il pacchetto da trasmettere prima di attivare il canale stesso. Analogamente, il canale di ricezione inserisce nel proprio registro buffer le informazioni che vengono ricevute fino al termine della ricezione dell'intero pacchetto. Il canale in ricezione è sempre attivo, pronto a ricevere un pacchetto indirizzato al nodo su cui si trova il canale, segnalando, come detto prima

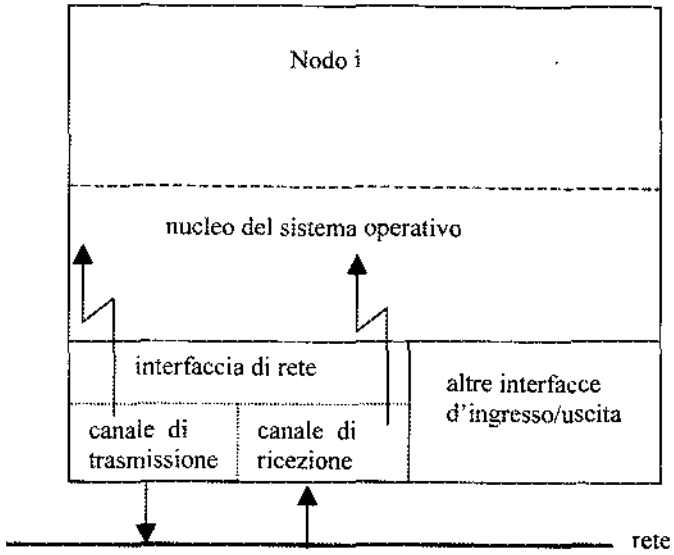


Figura 8.10 Interfaccia di rete.

il completamento della ricezione tramite un'interruzione. Sarà compito del gestore di queste interruzioni liberare il registro buffer del canale riattivandolo per una successiva ricezione. Viceversa, sul canale di trasmissione può verificarsi una competizione fra più processi residenti sul nodo che vogliono trasmettere contemporaneamente un loro pacchetto. Per risolvere tale competizione possiamo ipotizzare di definire nel nucleo, associata al canale di trasmissione, una coda di pacchetti pronti per essere inviati. In questo caso, indicando con `packet` il tipo di una struttura contenente tutti i campi previsti dal pacchetto, faremo l'ipotesi di aver definito il tipo astratto `coda_di_pacchetti`, implementato in uno qualunque dei vari modi visti nei capitoli precedenti. Quindi, senza riportare il codice di questo tipo astratto, supporremo di averne stanziato l'oggetto `packet_queue` su cui operare mediante le funzioni:

- `void inserisci(packet p)`, per inserire il pacchetto `p` nella coda `packet_queue`;
- `packet rimuovi()`, per restituire un pacchetto estratto da `packet_queue`;
- `boolean vuota()`, per testare se la coda `packet_queue` è vuota.

Possiamo quindi definire la funzione primitiva `invia_pacchetto` invocata per trasmettere un pacchetto sulla rete:

```
void invia_pacchetto(packet p) {
   if (<canale di trasmissione occupato>)
      packet_queue.inserisci(p);
   else {
      <inserimento di p nel registro buffer del canale>;
      <attivazione trasmissione>; }
}
```

La funzione, testando il registro di stato del canale, verifica se lo stesso è occupato trasmettere un precedente pacchetto o se, viceversa, è disponibile. Nel primo cas la funzione termina inserendo il pacchetto da trasmettere nella coda associata al c nale. Nell'altro caso, il pacchetto viene inserito nel registro buffer del canale e la tr smissione viene attivata. Alla fine di una trasmissione, il canale lancia un'interruzi ne che viene gestita dalla seguente funzione:

```
void tx_interrupt_handler() {
   packet p;
   salvataggio_stato();
   if(!packet_queue.vuota()){
      p=packet_queue.estrai();
      <inserimento di p nel registro buffer del canale>;
      <attivazione trasmissione>; }
   ripristino_stato();
}
```

Come tutte le funzioni di risposta alle interruzioni, per prima cosa salva lo stato d processo interrotto, quindi verifica lo stato della coda associata al canale. Se ques non è vuota, estrae il primo pacchetto in essa contenuto e riattiva la trasmissione, a trimenti il canale resta *idle*. Termina comunque riattivando il processo interrotto m diante la funzione `ripristino_stato`.

Vediamo quindi come implementare le due funzioni di libreria viste nel prec dente paragrafo, `void send (T inf,PID proc,int ip)` e `void receiv (T &inf,PID &proc,int ip)`, per consentire a un processo di inviare o riceve re messaggi sia da processi residenti sullo stesso nodo della rete o, in maniera d tutto trasparente, da processi residenti su nodi diversi. Per prima cosa è necessari adottare un criterio che abiliti il nucleo di ogni nodo a identificare il nodo della re su cui risiede un processo. Per esempio possiamo ridefinire il tipo `PID` di un proce so non come un semplice intero ma come una coppia di interi:

```
typedef struct {
   int indice_nodo;
   int PID_locale;
} PID;
```

Il primo intero denota il nodo su cui risiede il processo mentre il secondo denota PID locale al nodo. In questo modo ogni processo viene ancora identificato in ma niera univoca, all'interno del nodo su cui risiede, mediante un intero (il suo PID l cale), esattamente come abbiamo visto nel precedente paragrafo. Il campo `indi ce_nodo` consente invece di identificare il nodo su cui il processo risiede: quest significa che ogni nodo della rete viene identificato mediante un diverso intero. Sup poniamo, per esempio, che il nucleo di ogni nodo contenga la costante intera `no me_nodo` che lo identifica in modo univoco all'interno della rete.

Rivediamo quindi le due funzioni di comunicazione iniziando con la `send`. Que sta funzione può essere facilmente realizzata mediante un unica istruzione `if-else`

```
void send (T inf,PID proc,int ip){
   if (proc.indice_nodo==nome_nodo)
      local_send(inf,proc,ip);
   else remote_send(inf,proc,ip);
```

Se l'indice del nodo su cui risiede il processo destinatario del messaggio coincide col nodo su cui risiede il mittente, siamo in presenza di una comunicazione locale e quindi viene invocata la funzione `local_send`. Nell'altro caso il messaggio deve essere inviato a un processo remoto e quindi viene invocata la funzione `remote_send`. Si tratta quindi di realizzare queste due ulteriori funzioni. La funzione `local_send` coincide esattamente con la `send` implementata nel precedente paragrafo. Infatti, coinvolge due processi (mittente e ricevente) entrambi residenti sullo stesso nodo e quindi effettua le stesse operazioni, in particolare invoca la stessa primitiva (`inserisci_porta`) vista nel caso di architetture non distribuite. Viceversa, l'altra funzione, `remote_send`, deve confezionare un pacchetto contenente tutte le informazioni relative ai suoi parametri, pacchetto da inviare tramite la funzione `invia_pacchetto` al nodo su cui risiede il processo destinatario:

```
void remote_send (T inf, PID proc, int ip){
   packet p;
   PID mit=PIE();
   indice_nodo_destinatario=proc.indice_nodo;
   <vengono riempiti i vari campi del pacchetto p: in particolare, viene inserito, nel campo relativo al nodo a cui inviare il pacchetto, il valore indice_nodo_destinatario. Inoltre, nel campo del pacchetto destinato a contenere le informazioni da inviare, vengono inseriti il nome mit del processo che invia e i tre parametri della funzione inf, proc, e ip>;
   invia_pacchetto(p);
}
```

Il pacchetto, così confezionato e inviato sulla rete, viene ricevuto dal canale di ricezione del nodo destinatario. Una volta terminata la ricezione e quindi col pacchetto presente nel registro proprio buffer, il canale di ricezione genera un'interruzione che viene gestita dalla seguente funzione, che ha il compito di completare l'inserimento del messaggio (`inf`) inviato dal processo mittente nella porta di indice `ip` del processo ricevente `proc`:

```
void rx_interrupt_handler() {
   packet p;
   PID mit; T inf; PID proc; int ip;
   salvataggio_stato();
   <assegnamento a p del pacchetto ricevuto presente nel buffer del canale>;
   <attivazione ricezione>;
   <estrazione dal campo del pacchetto p, contenente le informazioni ricevute, del nome mit del mittente e dei tre parametri della funzione send inf, proc, e ip>;
   messaggio * m=new messaggio;
   m->informazione=inf;
   m->mittente=mit;
   inserisci_porta(m, proc, ip);
   ripristino_stato();
}
```

La routine di risposta alle interruzioni, una volta salvato lo stato del processo interrotto, per prima cosa assegna alla variabile `p` il contenuto del pacchetto ricevuto, presente nel registro buffer del canale, in modo tale da poter riattivare subito il canale per la ricezione successiva; quindi, dal campo del pacchetto contenente le informazioni ricevute, estrae i seguenti quattro dati: nome `mit` del processo mittente, informazione `inf` inviata dal mittente, nome `proc` del processo a cui inviare il messaggio e indice `ip` della sua porta, attraverso la quale il messaggio dovrà essere ricevuto. La funzione termina poi esattamente come una normale `send` locale, in modo tale che il processo ricevente possa eseguire la `receive` indipendentemente dal nodo su cui risiede il processo mittente. Al termine della gestione dell'interruzione viene, infine, ripristinato lo stato del processo interrotto.

Con queste modifiche alle funzioni del nucleo, la funzione `receive` resta esattamente la stessa che abbiamo visto nel precedente paragrafo.

8.5 Sommario

In questo capitolo è stato presentato il primo dei tre meccanismi di comunicazione che vengono presi in considerazione nel testo con riferimento al modello a scambio di messaggi e, in particolare, il più *primitivo* fra i tre, che prevede che l'operazione di invio sia asincrona. Per prima cosa è stato mostrato come, con tale meccanismo, sia possibile scrivere processi servitori che simulano le stesse funzionalità che, nel modello a memoria comune, erano proprie di una risorsa condivisa. Infatti il concetto di gestore di una risorsa, o di un pool di risorse equivalenti, che nel modello a memoria comune coincideva con una risorsa condivisa, adesso coincide con quello di processo servitore. È stato messo in evidenza come, adottando un diverso paradigma di programmazione, sia possibile risolvere gli stessi problemi di allocazione di risorse visti nell'altro modello, arrivando anche a una tabella di corrispondenza tra le soluzioni previste nell'ambito dei due modelli: questa corrispondenza non deve però essere intesa anche come equivalenza in termini di efficienza.

Per chiarire questi concetti sono state illustrate le soluzioni ad alcuni problemi di allocazione di risorse precedentemente visti anche nel modello a memoria comune, ivi incluse le tecniche per la specifica di strategie di priorità.

Alla fine, sono state mostrate possibili implementazioni delle funzioni di comunicazione, tramite le primitive, nel nucleo di un sistema operativo adatto sia per architetture monoelaboratore o multielaboratore sia per architetture distribuite.

8.6 Note bibliografiche

Il lavoro di Brinch Hansen [65] descrive la struttura del nucleo di uno dei primi sistemi operativi che hanno adottato il meccanismo di comunicazione asincrona, il sistema RC-4000.

Il problema del confronto fra i due modelli a memoria comune e a scambio di messaggi viene ampiamente discusso nel lavoro di Lauer e Needham [43]. Tale lavoro risulta di particolare interesse al fine di comprendere le differenze di stile e le diverse organizzazioni dei programmi che si ottengono utilizzando i meccanismi propri dei due modelli. Un confronto fra i due modelli è pure contenuto nel lavori di Bryant e Dennis [42] e di Wegner e Smolka [71].

Per approfondire il tema dei sistemi distribuiti può essere preso in considerazione il testo di Tanenbaum [70] mentre in [66] viene descritto il meccanismo delle socket.

9

Primitive di comunicazione sincrone

Come è stato mostrato nel capitolo precedente, le primitive di comunicazione asincrone rappresentano il più semplice meccanismo che può essere utilizzato in un ambiente di programmazione che segue il modello a scambio di messaggi per programmare le interazioni tra processi. La caratteristica peculiare di quel meccanismo è costituita dal completo asincronismo tra il processo che invia un messaggio e il processo destinato a riceverlo. Questo aspetto, come è stato messo in evidenza nei due precedenti capitoli, ha due diverse implicazioni: una relativa al modo in cui il meccanismo viene utilizzato per risolvere problemi di interazione tra processi, l'altra relativa all'implementazione dello stesso meccanismo di comunicazione. La prima delle due implicazioni riguarda la necessità di programmare esplicitamente opportuni protocolli di comunicazione tutte le volte che sia necessario garantire particolari forme di sincronizzazione fra i processi comunicanti (vedi per esempio, con riferimento alla figura 8.8, la realizzazione di un processo server che implementa una mailbox di dimensioni finite). L'altra implicazione, di tipo realizzativo, è relativa alla necessità di associare a ogni canale di comunicazione un buffer di dimensioni illimitate per memorizzare i messaggi già inviati ma non ancora ricevuti. Per questo motivo, come illustrato nel paragrafo 8.4, a ogni porta è stata associata una coda di messaggi realizzata mediante una lista concatenata. Il compito di questo capitolo è quello di introdurre il secondo meccanismo di comunicazione, quello sincrono, in cui il processo mittente, invocando l'operazione di invio, attende, prima di proseguire, che il destinatario abbia ricevuto il messaggio.

L'obiettivo è quello di mettere in evidenza le differenze rispetto al precedente meccanismo asincrono, sia per quanto riguarda i criteri d'uso per risolvere problemi di interazione tra processi, sia relativamente all'implementazione. Per questo motivo, cercheremo di ripercorrere gli stessi esempi visti nel precedente capitolo, in modo tale da puntualizzare più precisamente le differenze tra i due diversi meccanismi di comunicazione.

Come messo in evidenza nel paragrafo 7.3, la notazione sintattica che utilizzeremo per denotare le primitive sincrone è la stessa usata per le primitiva asincrone an-

che se, ovviamente, le primitive d'invio di un messaggio sono semanticamente diverse fra loro:

```
send (<valore>) to <porta>;
receive (<variabile>) from <porta>;
```

In particolare, come esempio, continueremo a utilizzare il concetto di *porta* intesa come canale asimmetrico, cioè con la possibilità che, sullo stesso canale, possano inviare messaggi più processi mittenti ma un solo ricevente sia abilitato a ricevere.

9.1 Confronto fra le primitive sincrone e le primitive asincrone

La differenza fondamentale fra le primitive sincrone e quelle asincrone risiede nella semantica della primitiva `send` che, come è noto, prevede che il processo mittente si sincronizzi con l'esecuzione della primitiva `receive` da parte del destinatario, in modo tale che il trasferimento dell'informazione avvenga quando entrambi i processi sono pronti a comunicare. Ciò ha due conseguenze:

a) la prima riguarda il grado di concorrenza tra le esecuzioni di due processi comunicanti. Tale concorrenza è sicuramente maggiore nel caso di primitive asincrone con le quali un processo mittente, eseguendo una `send`, invia il messaggio sulla porta e prosegue in concorrenza col processo ricevente, anche se quest'ultimo non è pronto a ricevere il messaggio. Viceversa, nel caso di primitive sincrone, il primo dei due processi comunicanti che arriva al punto in cui desidera inviare (se è il mittente) o ricevere (se è il destinatario) deve attendere che anche l'altro processo sia pronto a comunicare; le esecuzioni dei due processi potranno riprendere concorrentemente tra loro solo dopo che l'informazione è stata trasferita dal mittente al destinatario.

b) la seconda conseguenza riguarda la completa assenza di buffer nei canali di comunicazione. Infatti, non esiste più la necessità di memorizzare messaggi già inviati ma non ancora ricevuti.

Per comprendere meglio quest'ultima considerazione, supponiamo di prendere in esame il caso di due processi (`produttore` e `consumatore`) che si scambiano messaggi di tipo `T` tramite un canale simmetrico (la porta `dati`), nel quale cioè il solo processo `produttore` può inviare messaggi (vedi figura 9.1).

Quando il `produttore` esegue la `send` per inviare il messaggio (contenuto nella variabile `sorgente`) è del tutto inutile che tale messaggio sia memorizzato temporaneamente all'interno della porta `dati` in attesa di essere ricevuto.

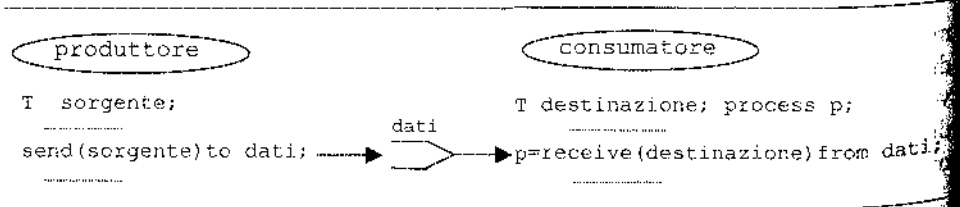


Figura 9.1 Singolo produttore e singolo consumatore.

Infatti il `produttore`, prima di proseguire ed eventualmente inviare un nuovo messaggio, deve comunque attendere che il `consumatore` sia disponibile a ricevere e a quel punto, almeno dal punto di vista teorico, il contenuto del messaggio può essere trasferito direttamente dalla variabile `sorgente` alla variabile `destinazione`, in cui il `consumatore` desidera riceverlo. Leggermente diverso si presenta il caso in cui la comunicazione sia asimmetrica (alla porta `dati` possono inviare messaggi più processi mittenti). Anche in questo caso ogni processo `produttore`, eseguendo la `send`, deve attendere che il `consumatore` sia pronto a ricevere il suo messaggio. La differenza rispetto a un canale simmetrico è che, in un certo istante, più processi produttori possono aver invocato la `send` e quindi, quando il `consumatore` esegue la `receive`, questo deve entrare in *rendez-vous* con un solo `produttore`, quello che ha invocato la `send` per primo, mentre gli altri restano in attesa di una successiva `receive`. Quindi, di nuovo, il costrutto `port` identifica un canale privo di buffer. È però necessario che venga mantenuta l'informazione relativa all'ordine con cui i vari processi produttori hanno invocato la `send`, in modo tale che, all'atto dell'esecuzione della `receive`, il `consumatore` possa discriminare con quale `produttore` entrare in comunicazione. Riprenderemo questi aspetti nell'ultimo paragrafo, dove descriveremo le tecniche di implementazione delle primitive sincrone.

Possiamo quindi ricapitolare brevemente i pro e i contro delle primitive sincrone rispetto alle primitive asincrone. Un primo vantaggio, relativo al precedente punto b), riguarda una più semplice implementazione delle primitive sincrone: non è più necessaria infatti una coda di messaggi di lunghezza illimitata per ogni porta, con evidente risparmio di memoria. Un secondo vantaggio, riguarda la maggiore semplicità con cui vengono risolte alcune particolari interazioni tra processi, in particolare quelle che prevedono la necessità di sincronizzare le velocità dei processi stessi. In questi casi, infatti, la sincronizzazione implicita nella primitiva `send` garantisce un minor numero di messaggi scambiati tra i processi. Metteremo in evidenza questi aspetti nelle soluzioni di alcuni problemi nei successivi paragrafi. D'altro canto, il principale svantaggio delle primitive sincrone è legato alla riduzione della concorrenza fra un processo mittente e il relativo ricevente che, come conseguenza, produce anche un maggior numero di commutazioni di contesto a livello del nucleo del sistema operativo.

Esiste infine un'ultima differenza tra i due tipi di primitive di comunicazione che riguarda, in particolare, i comandi con guardia. Questo tipo di istruzioni è stato introdotto nel paragrafo 7.3 per fornire al programmatore uno strumento col quale controllare il non determinismo implicito in ogni programma concorrente. Per esempio, nello scrivere il codice di un processo servitore che utilizza una risorsa R a favore di alcuni processi clienti, si hanno due sorgenti di non determinismo: una *locale* al processo servitore, l'altra *globale*. Utilizzando i comandi con guardia, il controllo dei due tipi di non determinismo viene effettuato dal programmatore mediante le due componenti di ogni guardia: l'espressione logica (detta anche *componente booleana* della guardia) e la primitiva `receive` (detta anche *componente di input* della guardia). Per esempio, se indichiamo con op_1, ..., op_n le n operazioni che possono essere eseguite su R, la causa locale di non determinismo riguarda il fatto che in un certo istante, il processo servitore può eseguire su R una qualunque delle pre

cedenti operazioni con alcune eccezioni. Per esempio, può accadere che l'operazione op_i sia eseguibile solo se la risorsa R si trova in uno stato interno in cui è soddisfatta una particolare condizione logica C_i (*condizione di sincronizzazione*). In altri termini, quando la condizione C_i non è soddisfatta il servitore può eseguire una qualunque operazione su R purché diversa da op_i. Quindi, se al servitore sono arrivate richieste per eseguire varie operazioni (per esempio op_i, op_j, op_k ecc.) può essere scelta, in modo non deterministico, una qualunque di queste ma non op_i. Come è stato visto nei precedenti capitoli, questo livello di controllo del non determinismo viene ottenuto specificando l'espressione logica C_i come parte booleana della guardia relativa al ramo nel cui corpo viene eseguita l'operazione op_i. Infatti quando C_i non è vera la guardia che la contiene non è valida e quindi l'operazione op_i non può essere eseguita.

Il non determinismo globale corrisponde al fatto che al servitore, in un certo istante, possono essere arrivate richieste da vari processi clienti e, in assenza di particolari strategie di gestione e fatto salvo quanto detto precedentemente, la scelta di quale richiesta accettare per prima è del tutto indifferente. Anche il controllo di questo livello di non determinismo viene effettuato mediante le guardie e, in particolare, con le loro componenti di input. Ogni guardia valida corrisponde alla possibilità di ricevere una richiesta per l'esecuzione di una particolare operazione la cui condizione di sincronizzazione è vera. La scelta del particolare ramo fra quelli con guardie valide è del tutto indifferente e, quindi, può essere effettuata in modo non deterministico. Il fatto poi che la primitiva `receive` sia bloccante quando non ci sono messaggi sulla porta su cui deve essere eseguita, consente al processo servitore di bloccarsi quando tutte le guardie non fallite sono ritardate, cioè quando non ci sono richieste in arrivo per operazioni eseguibili.

In un programma concorrente, adatto per un'architettura che segue il modello a scambio di messaggi, esistono però anche altre cause di non determinismo globale. Per esempio, possono capitare situazioni in cui un processo mittente debba inviare un messaggio non a uno specifico ricevente, bensì a uno qualunque fra un insieme di possibili processi riceventi. In questi casi può capitare che alcuni fra i riceventi siano pronti a eseguire la `receive` mentre altri no. Se si verificano tali condizioni, è del tutto indifferente la scelta di quale sia il particolare processo, fra coloro che sono in attesa sulla `receive`, a cui il mittente invia il proprio messaggio. Se poi nessuno fra i riceventi fosse ancora disponibile a ricevere il messaggio, allora il processo mittente dovrebbe sospendersi in attesa che uno, o più d'uno, fra i riceventi invochi la `receive`. A ben vedere, questa forma di non determinismo globale è perfettamente simmetrica a quella vista precedentemente nel caso del processo servitore, che poteva ricevere una richiesta da uno qualunque fra un insieme di mittenti diversi. È quindi del tutto plausibile risolvere il problema in maniera simmetrica, cioè mediante un comando con guardia, solo che, al posto della componente di input della guardia (una `receive`), stavolta dobbiamo inserire una componente di output (una `send`). In questo caso la guardia assume la seguente forma sintattica:

(<espressione di tipo boolean>) ; <primitiva `send`>

Anche adesso, la valutazione di una guardia può dare luogo a tre diversi valori: una guardia è *fallita* se il valore dell'espressione booleana è `false`, è *valida* se il valore dell'espressione booleana è `true` e la `send` può essere eseguita senza ritardo e, i fine, è *ritardata* se il valore dell'espressione booleana è `true` ma la `send` non p essere eseguita perché il processo ricevente non è pronto a ricevere.

Con questa modifica delle guardie, il precedente problema può essere facilmer risolto mediante un unico comando nelle cui guardie non è necessario specificare componenti booleane. Per esempio, se il processo `mittente` deve inviare il m saggio `mes` a uno qualunque fra gli n processi $proc_1$, $proc_2$, ..., $proc_n$, in ciasc no dei quali è presente la porta `pr` da cui ricevere `mes`, il `mittente` può sempli mente eseguire il seguente comando:

```
if
   [] send(mes) to proc1.pr -> <istruzione_1>;
   .
   .
   .
   [] send(mes) to procn.pr -> <istruzione_n>;
fi
```

In questo caso, se tutti i processi riceventi fossero in attesa sulla rispettiva `rece ve`, tutte le guardie sarebbero valide e quindi la scelta del potrebbe essere fatta maniera completamente non deterministica. Se fossero in attesa solo alcuni dei ri venti, soltanto le guardie contenenti le `send` a essi indirizzate sarebbero valide e nuovo, la scelta del corrispondente ramo sarebbe del tutto ininfluente. Se, infi nessun ricevente fosse pronto sulla `receive` allora tutte le guardie sarebbero ri date e il processo mittente dovrebbe sospendersi.

Questa possibilità di avere nelle guardie la `send`, oltre che la `receive`, è caratteristica propria delle primitive sincrone. Con le primitive asincrone infatti avrebbe avuto nessun significato, in quanto una guardia contenente una `send` avrebbe mai potuto essere ritardata. Nel seguito, riprenderemo alcuni esempi già sti nel precedente capitolo illustrando come questa possibilità, offerta dalle primi sincrone, possa semplificare le relative soluzioni.

Quest'ultima differenza tra i due tipi di primitive di comunicazione è stata me in evidenza soprattutto per meglio caratterizzare le primitive sincrone rispetto a qu asincrone. Da un punto di vista pratico però, difficilmente la possibilità di utilizzar `send` all'interno di una guardia viene offerta da un linguaggio di programmazio ciò per una serie di difficoltà di tipo realizzativo che adesso cercheremo di illustrar

Come prima considerazione possiamo asserire che, se la `send` fosse l'unica mitiva utilizzabile all'interno delle guardie, la realizzazione dei comandi con gua sarebbe del tutto analoga a quella dei comandi che utilizzano la sola `receiv` problemi nascono quando, nello stesso programma, sono presenti sia comandi tenenti guardie di input che comandi contenenti guardie di output. Supponiamo esempio, che due processi P e Q stiano eseguendo, ciascuno, un comando con g dia come indicato nella figura 9.2 dove, per semplicità, non vengono utilizzat componenti booleane delle guardie.

L'esecuzione dei comandi presenti nei due processi P e Q implica, per prima sa, la valutazione delle relative guardie. Con riferimento alla figura 9.2, supponi che il processo R, non riportato nella figura, sia in attesa, avendo invocato la ceive sulla sua porta `portar`.

a)

```
process P {
   .......
   if
      [] send(ma) to Q.portaq1 -> <istruzione_a>;
      [] send(mb) to R.portar -> <istruzione_b>;
   fi
   .......
}
```

b)

```
process Q {
   .......
   if
   [] receive(va) from portaq1 -> <istruzione_c>;
   [] receive(vb) from portaq2 -> <istruzione_d>;
   fi
   .......
}
```

Figura 9.2 Guardie di input (a) e di output (b).

Allora, la guardia relativa al secondo ramo del comando di P è valida perché R è pronto a ricevere il messaggio mb. Per valutare la prima guarda di P è però necessario verificare lo stato in cui si trova il processo Q; poiché anch'esso sta eseguendo un comando alternativo, dobbiamo valutare anche le sue guardie. Supponiamo ora che un quarto processo, anch'esso non riportato nella figura, abbia invocato una send per inviare un messaggio sulla porta portaq2 di Q: allora anche la seconda guardia di Q è valida. Infine, la prima guardia di Q è valida solo se P è pronto a inviare un messaggio sulla sua porta portaq1, ma ciò vero nel caso in cui venga scelto il primo ramo del comando di P. Questo ramo può essere scelto se è valida la sua guardia, per la quale valgono le stesse considerazioni viste per la prima guardia di Q. In conclusione possiamo asserire che tutte le guardie, sia di P sia di Q, sono potenzialmente valide. Però, poiché la scelta dei rami da eseguire nei due comandi avviene in maniera non deterministica e del tutto indipendente l'una dall'altra, può accadere che il processo P scelga il suo secondo ramo mentre Q sceglie il primo. In questo caso, pur avendo valutato come valida la prima guardia di Q, il processo resta sospeso poiché P ha deciso di eseguire un ramo alternativo a quello che avrebbe consentito la comunicazione tra P e Q.

Questa difficoltà di valutazione delle guardie può condurre, in certi casi, anche a condizioni di stallo come nel caso illustrato nella figura 9.3 dove sono presenti quattro processi: P e Q, entrambi pronti a ricevere messaggi o da R o da W, e R e W, entrambi pronti a inviare messaggi o a P o a Q. In base alle considerazioni viste precedentemente, tutte le guardie dei quattro processi possono essere valutate come valide e quindi ogni processo può, autonomamente, scegliere uno dei due rami del proprio comando con guardie. La condizione di stallo si verifica se, per esempio, P decide di scegliere il primo ramo, che prevede di ricevere un messaggio da R, e contempo-

```
process P {
   .....
   if
⇨  [] receive(va) from p1
         -> <..>;
    [] receive(vb) from p2
         -> <..>;
   fi
   .....
}
```

```
process Q {
   .....
   if
    [] receive(vc) from q1
         -> <..>;
⇨  [] receive(vd) from q2
         -> <..>;
   fi
   .....
}
```

```
process R {
   .....
   if
    [] send(ma) to P.p1
         -> <..>;
⇨  [] send(mc) to Q.q1
         -> <..>;
   fi
   .....
}
```

```
process W {
   .....
   if
⇨  [] send(mb) to P.p2
         -> <..>;
    [] send(md) to Q.q2
         -> <..>;
   fi
   .....
}
```

Figura 9.3 Possibile condizione di stallo.

raneamente R sceglie il suo secondo ramo, che prevede di inviare un messaggio a Supponiamo inoltre che Q scelga il suo secondo ramo, nel quale è previsto di riceve un messaggio da W, mentre W sceglie il suo primo ramo, che prevede di inviare messaggio a P. Con queste ipotesi, si verifica la classica situazione di attesa circola P in attesa da R, R in attesa da Q, Q in attesa da W e W in attesa da P. Nella figura 9.3, l'interno dei comandi con guardia dei quattro processi, la precedente condizione stallo è rappresentata indicando con una freccia grigia il ramo scelto da ogni proces

Queste difficoltà nascono quando, come è stato precedentemente detto, sono p senti sia comandi con guardie di output che comandi con guardie di input, o com di con entrambe le tipologie di guardia contemporaneamente. Per evitare queste c ficoltà faremo riferimento, nel seguito, a una sola tipologia di guardie e, in partico re, alle sole guardie di input. Infatti, la possibilità di avere la receive in una gu dia rappresenta lo strumento più importante per controllare il non determinismo g bale durante la realizzazione di un processo server che, come è stato illustrato an nel precedente capitolo, rappresenta il concetto equivalente a quello di risorsa con visa del modello a memoria comune.

9.2 Processi servitori

Nel paragrafo 8.1 abbiamo illustrato come, utilizzando le primitive di comunicaz ne asincrone, sia possibile simulare, tramite un processo servitore, il concetto di sorsa condivisa proprio dello schema a memoria comune. Nella tabella 8.1 è st

inoltre sintetizzato uno schema di corrispondenza fra i paradigmi e gli strumenti di programmazione che, utilizzati nei due diversi ambienti, consentono di risolvere lo stesso tipo di problemi. Se volessimo ripetere le stesse considerazioni, utilizzando però le primitive sincrone, otterremmo gli stessi risultati: eviteremo perciò di ripetere quanto già visto nel precedente capitolo, limitandoci a rimarcare che i risultati esposti nella tabella 8.1 restano del tutto validi anche se riferiti alle primitive di comunicazione sincrone. Infatti, in base a quanto visto nel precedente paragrafo, le varianti fra una soluzione che utilizza le primitive asincrone e quella che utilizza le primitive sincrone, riguarda soprattutto l'efficienza di esecuzione legata al diverso grado di concorrenza e quindi al diverso numero di commutazioni di contesto fra processi che avvengono a livello del nucleo del sistema operativo. Viceversa, normalmente non esiste nessuna variazione per quanto riguarda il testo dei programmi (questo almeno se ci limitiamo, nel caso di comandi con guardia, alle sole guardie di input). Le uniche varianti si possono verificare in alcuni casi nei quali la sincronizzazione implicita nella primitiva `send` consente di eliminare qualche scambio di segnali fra i processi comunicanti. Vedremo, nei successivi paragrafi, alcuni esempi di questi casi particolari.

9.2.1 Gestione di un pool di risorse equivalenti

Il problema della gestione di un insieme di risorse equivalenti è lo stesso che abbiamo già presentato nel paragrafo 8.2.1 a proposito delle primitive asincrone. Come abbiamo visto in quel caso, la necessità di realizzare un processo `server` nasce dall'esigenza che un cliente ha di scegliere fra gli n processi P_1, ..., P_n, ciascuno dei quali è l'unico a poter operare sulla corrispondente risorsa, uno fra quelli disponibili, se ce ne sono, evitando di inviare richieste di servizio a chi sta già operando per altri clienti (vedi figura 8.5). Questo problema rappresenta un primo esempio in cui la possibilità di utilizzare guardie di output nei comandi con guardie eliminerebbe del tutto la necessità di realizzare il processo `server`. Sarebbe, infatti, sufficiente che ogni cliente utilizzasse semplicemente un comando alternativo composto da n rami, come visto nel paragrafo 9.1. L'`i`-esimo ramo del comando dovrebbe essere caratterizzato da una guardia contenente la `send` con cui il cliente invia la richiesta di servizio al corrispondente processo P_i.

Se però non prendiamo in considerazione la possibilità di utilizzare guardie di output, sorge ancora la necessità di realizzare il processo `server` esattamente come nel caso delle primitive asincrone. In altri termini, è di nuovo necessario realizzare lo stesso protocollo di comunicazione fra un qualunque processo cliente, il processo `server` e i processi P_1, ..., P_n che gestiscono le risorse, così come illustrato nella figura 8.5. In base alla specifica di tale protocollo, descritta più in dettaglio nel paragrafo 8.2.1, il codice del `server` corrisponde esattamente a quello già visto nel precedente capitolo e riportato nella figura 8.6. Quindi, a parte la diversa semantica delle primitive, la struttura sintattica della soluzione resta inalterata. In particolare, quando un cliente ha bisogno di una risorsa, invia un messaggio di richiesta al `server` e resta in attesa di conoscere l'indice della risorsa assegnata. Quando poi la risorsa viene rilasciata, non è necessario nessun messaggio di risposta da parte del `server` al cliente.

Se adesso esaminiamo però, la soluzione allo stesso problema nell'ipotesi in cui `server` gestisca una sola risorsa, il codice del `server` si semplifica rispetto quello visto con le primitive asincrone, riportato nella figura 8.7. Come si può nota anche in questo caso, all'atto di una richiesta, il cliente resta in attesa di un messa gio da parte del `server`, messaggio che stavolta non porta nessun contenuto inf mativo in quanto la risorsa da utilizzare è nota a priori essendocene una sola. Il co tenuto del messaggio di risposta dal `server` al cliente è un semplice segnale, n cessario al cliente per sapere quando il `server` ha accolto la propria richiesta, modo tale da poter iniziare a utilizzare la risorsa. Come è facile capire, in questo c so particolare, la comunicazione tra `server` e cliente può essere semplificata s pendo che la `send` è sincrona. Infatti, quando il cliente invia la propria richiesta, sta in attesa che il `server` la riceva ed è del tutto inutile l'ulteriore messaggio c con le primitive asincrone doveva essere spedito dal `server` al cliente. Lo scher del `server` diventa quindi il seguente:

```
process server{
   port signal richiesta;
   port signal rilascio;
   boolean libera=true;
   process p ;
   signal s;
   do
   [] libera==true;p=receive(s) from richiesta; ->
      libera=false;
   [] p=receive(s) from rilascio; ->
      libera=true;
   od;
}
```

Ricordando, infine, che in questo caso particolare il `server` corrisponde a un s maforo binario, possiamo concludere che anche il codice di un processo `serv` che fornisce le funzionalità di un semaforo generale si semplifica rispetto a quel visto con le primitive asincrone, per lo stesso motivo indicato precedentemente. D resto, essendo un semaforo un puro meccanismo di sincronizzazione, è ovvio che sua realizzazione mediante la `send` sincronizzata sia più semplice di quella vis con le primitive asincrone. Lo schema del semaforo diventa adesso:

```
process semaphore{
   port signal P;
   port signal V;
   int valore=1;
   process proc;
   signal s;
   do
   []valore>0;p=receive(s) from P; ->
      valore--;
   [] p=receive(s) from V; ->
      valore++;
   od;
}
```

9.2.2 Problema dei produttori/consumatori e realizzazione di una mailbox

La realizzazione di una mailbox (canale da molti a molti) di dimensioni limitate (pari a N) è stata illustrata nel capitolo 8.2.2, con riferimento alle primitive asincrone, mediante un processo server, a cui è stato affidato il compito di garantire che il numero di messaggi già inviati, ma non ancora ricevuti, non superi mai il valore prestabilito N e, al tempo stesso, il compito di trasferire i messaggi già inviati ai processi consumatori, quando gli stessi sono pronti a riceverli. Come abbiamo visto nel precedente capitolo, con le primitive asincrone la necessità di programmare un server non ci sarebbe stata se fossero stati presenti un solo processo produttore e un solo consumatore e se non fossero posti limiti alla dimensione del buffer dei messaggi; ciò, come sappiamo, in quanto la capacità di memorizzare i messaggi già inviati ma non ancora ricevuti è fornita direttamente dal buffer dello stesso canale asincrono.

Nel caso di primitive sincrone, il problema di gestire una coda di messaggi è molto più generale e ha senso anche nell'ipotesi di un solo processo produttore e di un solo consumatore, in quanto un canale sincrono non dispone di nessun buffer al proprio interno. Vediamo quindi come realizzare una mailbox con le primitive sincrone partendo subito dal caso generale di avere più produttori e più consumatori e di voler ancora definire pari a N la dimensione del buffer, ricordando però che tale soluzione è valida anche nel caso di un solo produttore e di un solo consumatore e qualunque sia il valore di N.

Come schematicamente illustrato nella figura 8.10, con le primitive asincrone avevamo introdotto le due porte di tipo `signal`, `pronto_prod` e `pronto_cons`, utilizzate dal processo `mailbox` per sincronizzarsi, rispettivamente, con i processi produttori e con i consumatori. Quando un produttore (o un consumatore) era pronto a inviare (o a ricevere) un messaggio, inviava un segnale sulla corrispondente porta del processo `mailbox`. Inviato il segnale di pronto, ogni consumatore restava poi in attesa dei dati eseguendo una `receive`, mentre il protocollo eseguito da ogni produttore era più complesso in quanto, una volta segnalata la propria disponibilità a inviare dati, doveva attendere un segnale di conferma da parte del processo `mailbox` attraverso la porta `ok_to_send`. Tale segnale di sincronizzazione era necessario per evitare l'invio dei dati nel caso di buffer pieno. Nel risolvere il problema con le primitive sincrone quest'ultimo messaggio di controllo può essere eliminato, in quanto stavolta la sincronizzazione è implicita nella `send`. I protocolli eseguiti dai produttori e dai consumatori diventano quindi del tutto simmetrici, come illustrato di seguito (vedi anche figura 9.4).

Nel descrivere la soluzione, per semplicità non viene dettagliata la realizzazione del tipo astratto `coda_messaggi`, locale al processo `mailbox`, di cui viene dichiarata l'istanza `coda`. In particolare, supporremo che su oggetti del tipo `coda_messaggi` si possa operare con le seguenti funzioni:

- `void inserimento(T mes);` //per inserire il messaggio `mes` nella coda
- `T estrazione();` //che restituisce il primo elemento della coda
- `boolean piena();` // che restituisce `true` se la coda è piena
- `boolena vuota();` // che restituisce `true` se la coda è vuota

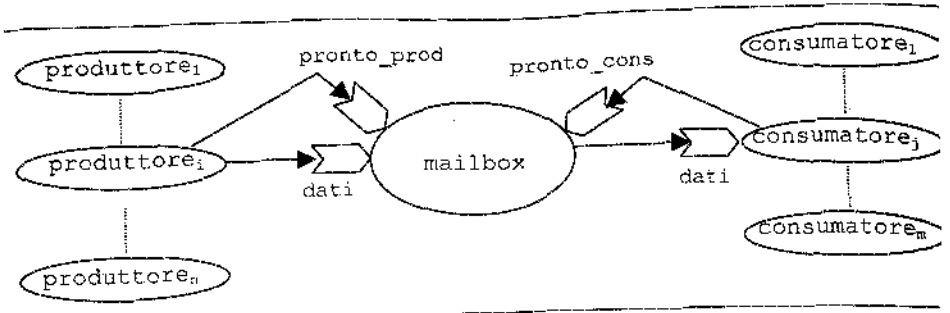


Figura 9.4 Mailbox di dimensioni finite (protocolli simmetrici).

Con tali indicazioni il codice del processo mailbox è il seguente:

```
process mailbox{
   port T dati;
   port signal pronto_prod, pronto_cons;
   T messaggio:
   process p:
   signal s:
   coda_messaggi coda:
      <inizializzazione>:
   do
   [] (!coda.piena());p=receive(s)from pronto_prod; ->
      p=receive(messaggio)from dati;
      coda.inserimento(messaggio);
   [](!coda.vuota());p=receive(s)from pronto_cons; ->
      messaggio=coda.estrazione;
      send(messaggio)to p.dati;
   od;
}
```

L'istruzione di inizializzazione serve per inizializzare l'oggetto `coda` nello st "coda vuota". Il codice del processo `mailbox` è molto semplice. È in pratica co tuito da un comando ripetitivo con due rami. Il primo è condizionato dal fatto ch coda non sia piena e dal fatto che un produttore sia pronto a inviare un messagg quando ciò è vero, il messaggio può essere ricevuto e inserito nella `coda`. In que caso, essendo la `send` sincronizzata, non è necessario restituire un segnale n porta `ok_to_send` del mittente, come nel caso delle primitive asincrone. L'a ramo è condizionato dal fatto che la `coda` non sia vuota e dal fatto che un consu tore sia pronto a ricevere un dato: quando ciò è vero, un dato viene estratto dalla da e, successivamente, inviato al consumatore. Di seguito viene riportato anch codice relativo al generico produttore e quello relativo al generico consumatore.

```
process produttorei{
   T messaggio;
   signal s;
      ..........
   <produci il messaggio>;
   send(s)to mailbox.pronto_prod;
```

```
    send(messaggio)to mailbox.dati;
       .....
 }
 process consumatore_j{
    port T dati;
    T messaggio;
    process p;
    signal s;
       .....
    send(s)to mailbox.pronto_cons;
    p=receive(messaggio)from dati;
    <consuma il messaggio>;
       .....
 }
```

Esaminando più in dettaglio il codice eseguito dal generico produttore ci si può facilmente accorgere di un'ulteriore semplificazione. Il fatto che un produttore debba inviare al processo mailbox prima un segnale, per indicare che è pronto, e successivamente i dati, sincronizzandosi quindi due volte di seguito, costituisce una inutile perdita di tempo. Possiamo eliminare l'invio del segnale sulla porta `pronto_prod`, che serve soltanto a dichiarare la disponibilità del processo a inviare un messaggio. Tale disponibilità, in virtù della sincronizzazione implicita nella `send`, può essere specificata inviando subito il messaggio nella porta `dati`, sapendo che comunque il processo `mailbox` lo riceverà soltanto quando la `coda` è non piena. Da queste considerazioni si ricava, quindi, la versione finale del processo `mailbox` e, di conseguenza, anche quella del generico produttore:

```
process mailbox{
   port T dati;
   port signal pronto_cons;
   T messaggio;
   process p;
   signal s;
   coda_messaggi coda;
<inizializzazione>;
   do
      [] (! coda.piena()); p=receive(messaggio)from dati; ->
         coda.inserimento(messaggio);
      [] (! coda.vuota()); p=receive(s)from pronto_cons; ->
         messaggio=coda.estrazione;
         send(messaggio)to p.dati;
   od;
}
process produttore_i{
   T messaggio;
   process p;
   signal s;
   .....
   <produci il messaggio>;
   send(messaggio)to mailbox.dati;
   .....
}
```

La possibilità di utilizzare anche la `send` in guardia comporterebbe un'analoga ulriore semplificazione, in questo caso nel protocollo fra `mailbox` e generico consmatore, eliminando anche la necessità dell'altra porta `pronto_cons`. Per esempse fosse presente un solo `consumatore` e indicando con `primo()` la funzioche restituisce il primo elemento in `coda` senza toglierlo dalla coda stessa, `mailbox` si potrebbe ridurre a:

```
process mailbox{
   port T dati;
   T messaggio;
   process p;
   coda_messaggi coda;
<inizializzazione>;
   do
   [](! coda.piena()); p=receive(messaggio)from dati;
      -> coda.inserimento(messaggio);
   [](!coda.vuota()); send(coda.primo())to consumatore.dati;
      -> messaggio=coda.estrazione;
   od
}
```

Per il processo consumatore avremmo invece:

```
process consumatore_j{
   port T dati;
   T messaggio;
   process p;
   .....
   p=receive(messaggio)from dati;
   <consuma il messaggio>;
   .....
}
```

Nel caso generale di più consumatori, nel processo `mailbox` il secondo ramo comando ripetitivo dovrebbe essere ripetuto per ogni consumatore.

Prima di terminare questo paragrafo, vogliamo mostrare un'ultima soluzionproblema della mailbox, soluzione che ha senso esclusivamente con le primitive scrone e che realizza la `mailbox`, invece che con un solo processo servitore, conarray di N processi servitori estremamente semplici e tutti uguali tra loro. Tale sozione simula, in software, il comportamento di un registro in traslazione. Ogni mento del registro viene simulato mediante un processo ciclico che si limita a rivere, in un proprio buffer locale, un messaggio tramite la sola porta di cui dispo Successivamente, invia il messaggio ricevuto alla porta del processo che lo se nell'array e torna in testa al ciclo (vedi figura 9.5).

Di seguito viene riportato il codice del generico processo appartenente all'acche costituisce la mailbox. L'unico processo dell'array che è leggermente diversl'ultimo, quello di indice `N-1`. La differenza risiede nel fatto che il messaggio rvuto dal processo che lo precede nell'array non viene inviato al processo successche in questo caso non esiste, ma direttamente al destinatario, il processo consmatore.

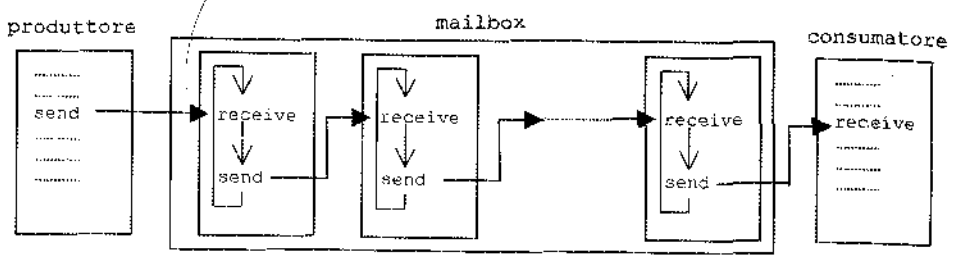


Figura 9.5 Mailbox concorrente.

```
process m_i { //(0=i=N-2)
   port T dati;
   T buffer;
   process p;
   while(true){
      p=receive(buffer)from dati;
      send(buffer)to m_i+1.dati;}
}
```

Come si può notare, all'inizio tutti i processi dell'array sono sospesi sulla `receive` in testa al proprio ciclo. Quando il `produttore` invia il primo messaggio, questo viene ricevuto e memorizzato nel proprio buffer da m_1 che, successivamente, lo invia a m_2 e così via. In questo modo il messaggio, dopo N passaggi fra i processi dell'array, finisce nel buffer di m_{N-1}, il quale si sospende sulla `send` in attesa che il `conumatore` sia pronto a riceverlo, mentre tutti gli altri processi della mailbox sono tornati in attesa sulla `receive` in testa al loro ciclo. Questa soluzione è molto semplice ed elegante. Presuppone però un supporto hardware a elevato parallelismo. In assenza di un elevato numero di processori fisici su cui allocare i processi dell'array, l'efficienza di questa soluzione risulterebbe molto penalizzata. Per esempio, nel caso di macchine fisiche monoelaboratore, ogni messaggio inviato dal produttore implicherebbe N commutazioni di contesto fra i processi dell'array prima di arrivare a destinazione.

9.3 Specifica di strategie di priorità

Per quanto riguarda le tecniche di specifica di strategie di priorità, le considerazioni illustrate nel paragrafo 8.3 restano del tutto inalterate anche per le primitive sincrone. In particolare, anche in questo caso, una tecnica che consente di risolvere ogni problema di specifica di priorità consiste nell'accettare, da parte del processo `server`, ogni richiesta di servizio anche quando questa non è immediatamente esaudibile, salvo registrare questo fatto, e nel servire le richieste pendenti, quando sarà possibile, in base all'ordine di priorità che si vuole implementare.

Per esempio, se volessimo riscrivere la soluzione relativa alla realizzazione di un processo `server` che alloca N risorse equivalenti agli M processi P_0, P_1, ..., P_{M-1}, in modo tale da privilegiare le richieste di P_0 rispetto a quelle di P_1 e queste rispe a quelle di P_2 e così via, otterremmo una soluzione sintatticamente identica a que riportata nella figura 8.11.

Per concludere questo argomento, riportiamo di seguito la stessa soluzione i caso particolare di una sola risorsa (N=1), soluzione di nuovo identica a quella c otterremmo utilizzando primitive asincrone:

```
process server{
   port signal richiesta;
   port int rilascio;
   boolean libera;
   process p ;
   signal s;
   int sospesi=0;boolean bloccato[M];
   process client[M];
   //inizializzazione
   libera=true;
   for(int j=0;j<M;j++)bloccato[j]=false;
   client[0]="P0";.....client[M-1]="PM-1";}
   do
   [] p=receive(s)from richiesta; ->
      if(libera){
         libera=false;
         send(s)to p.risorsa;}
      else{
         sospesi++;
         int j=0;while(client[j]!=p)j++;
         bloccato[j]=true;}
   [] p=receive(s)from rilascio;
      if(sospesi==0)libera=true;
      else{
         int i=0;
         while(!bloccato[i])i++;
         sospesi--;
         bloccato[i]=false;
         send(s)to client[i].risorsa;}
   od;
}
```

Senza entrare nei dettagli di questa soluzione, in quanto già descritti nel paragr 8.3, è comunque opportuno sottolineare una sua specifica caratteristica. Nel parag fo 9.2.1 abbiamo visto che la soluzione del `server` di una sola risorsa con le pri tive sincrone consentiva di risparmiare un messaggio nei confronti dell'analoga luzione con primitive asincrone. Ciò in quanto non era necessario, in fase di rich sta, che il `server` restituisse un segnale di accettazione al processo richieden Adesso lo stesso problema, risolto però con la specifica di una strategia di prior non consente di ottenere lo stesso vantaggio. Infatti, il fatto di dover comunque cettare richieste anche quando non sono immediatamente esaudibili rende sem necessaria la risposta al cliente una volta che la richiesta sia stata completamente stita, ciò affinché il cliente possa sapere quando la risorsa è a sua disposizione.

9.4 Realizzazione del meccanismo di comunicazione sincrono

Nel paragrafo 8.4, nel descrivere una possibile realizzazione delle primitive asincrone, abbiamo scelto di illustrare come realizzare il meccanismo di comunicazione asincrona direttamente a livello del nucleo del sistema operativo e ciò proprio in virtù del fatto che il meccanismo di comunicazione asincrono rappresenta sicuramente quello di "più basso livello" tra le tre tipologie di meccanismi di comunicazione che prendiamo in considerazione. Dovendo adesso descrivere la realizzazione delle primitive sincrone illustreremo, viceversa, tre diverse tecniche: la prima, relativa alla simulazione di un meccanismo di comunicazione sincrono all'interno di un sistema organizzato secondo il modello a memoria comune; la seconda, relativa alla realizzazione delle funzioni di comunicazione sincrone supponendo di avere a disposizione le primitive di nucleo asincrone viste nel precedente capitolo; infine, descriveremo una tecnica per realizzare le stesse primitive sincrone come primitive offerte dal nucleo di un sistema operativo.

9.4.1 Simulazione di un meccanismo di comunicazione sincrono mediante semafori

Scopo di questo paragrafo è quello di mostrare come, pur avendo a disposizione un sistema organizzato secondo il modello a memoria comune, sia possibile simulare sullo stesso un meccanismo di comunicazione sincrono, per esempio mediante l'uso dei semafori. Nel precedente capitolo non è stato illustrato un analogo esempio relativamente al meccanismo di comunicazione asincrono in quanto la simulazione di questo meccanismo corrisponderebbe esattamente alla soluzione del classico problema produttore/consumatore già visto nel capitolo 5 (vedi figura 5.14), nell'ipotesi in cui il buffer fosse realizzato mediante un lista concatenata di messaggi e quindi di dimensioni illimitate.

Iniziamo con il caso più semplice relativo alla simulazione di un canale sincrono e simmetrico, cioè nell'ipotesi in cui su ogni porta invii messaggi un solo mittente e, ovviamente, riceva un solo ricevente. In questo caso, la sincronizzazione tra i due processi comunicanti coincide esattamente con la soluzione del problema del *rendez-vous* tra due processi, che utilizzava due *semafori evento*, vista alla fine del paragrafo 5.3 e relativa allo scambio di due segnali temporali. Tenendo conto che un canale sincrono, come già detto precedentemente, non ha necessità di fornire un buffer, la simulazione delle primitive sincrone potrebbe quindi coincidere, per quanto riguarda la sola sincronizzazione tra i processi comunicanti, con quella illustrata nella figura 5.6. D'altra parte, pur non essendo concettualmente necessario un buffer nella realizzazione di un canale sincrono, la sua disponibilità può risultare utile, da un punto di vista pratico, per altre considerazioni. In assenza del buffer, quando i due processi comunicanti sono pronti, l'uno a inviare e l'altro a ricevere, il dato da trasferire può essere direttamente prelevato dalla locazione *sorgente* specificata dal processo mittente e inserito nella locazione indicata come *destinazione* dal processo ricevente. Come è noto dalla teoria dei sistemi operativi, però, ogni processo dispone di una propria memoria virtuale e gli indirizzi della sorgente e della destinazione, indicati dai due processi, sono due indirizzi virtuali del tutto scorrelati l'uno risp all'altro, appartenendo a due spazi degli indirizzi indipendenti. In questo caso, a a disposizione una locazione nel canale, da utilizzare come buffer, può essere m utile in quanto l'indirizzo di tale buffer sarebbe condiviso tra gli spazi degli indir dei due processi.

Con queste ipotesi, e supponendo che l'operazione di ricezione, oltre a restitu il valore del dato ricevuto, restituisca anche il nome (il `PID`) del processo mitte il messaggio che i due processi si scambiano deve contenere i seguenti due camp campo `informazione` del tipo `T`, che rappresenta il tipo del contenuto inform vo scambiato tra i due processi, e il campo `mittente` di tipo `PID`, che identific nome del processo mittente. Quindi con il tipo `messaggio` possiamo denotare struttura che contiene i due precedenti campi:

```
typedef struct {
   T informazione;
   PID mittente;
} messaggio;
```

Utilizzando la precedente definizione di `messaggio`, possiamo dichiarare la cl porta che definisce sia la struttura dati di un canale sincrono e simmetrico si due operazioni di `invio` e `ricezione` che, rispetto a quelle già viste nel para fo 5.6.2, hanno adesso un comportamento semanticamente corrispondente al fun namento sincrono:

```
class porta {
   messaggio buffer;
   semaphore M=0;
   semaphore R=0;
   public void invio(T dato){
      messaggio mes;
      mes.informazione=dato;
      mes.mittente=PIE();
      buffer=mes;
      V(R);
      P(M);
   }
   public void ricezione(T &dato,PID &mit){
      messaggio mes;
      P(R);
      mes=buffer;
      V(M);
      dato=mes.informazione;
      mit=mes.mittente;
   }
}
```

La struttura dati di una porta ha tre campi: un `buffer` di tipo `messaggio` e semafori evento `M` e `R`, associati l'uno al processo mittente e l'altro al processo vente ed entrambi inizializzati a zero. Quando il mittente invoca `invio` passand il `dato` da inviare, il messaggio da spedire viene preliminarmente confeziona una variabile locale `mes` di tipo `messaggio`, inserendovi il `dato` e il `PID` del

tente (che corrisponde a quello del processo in esecuzione). A questo punto, il buffer della porta è sicuramente vuoto. Infatti, un messaggio eventualmente inviato con una precedente invio è sicuramente già stato ricevuto, in base alla sincronizzazione implicita in tale operazione. Quindi il messaggio mes può essere subito inserito nel buffer della porta. Successivamente, la funzione termina eseguendo una V sul semaforo R, su cui può essere in attesa il ricevente, e poi sospende il processo mittente mediante una P sul semaforo M, in attesa che il ricevente abbia estratto il messaggio dal buffer. La funzione ricezione, viceversa, inizia con una P sul semaforo R per sospendere, eventualmente, il ricevente se questo fosse il primo ad arrivare al *rendez-vous*; quindi estrae il messaggio dal buffer e termina con una V sul semaforo M per riattivare il mittente, essendo completato il trasferimento del dato. Prima di terminare, la funzione restituisce al ricevente l'informazione ricevuta, tramite il parametro dato, e il nome del mittente, tramite il parametro mit.

Possiamo adesso descrivere la soluzione generale nel caso di canale asimmetrico (cioè con N mittenti e un solo ricevente). La soluzione, in pratica, è una semplice generalizzazione della precedente: il buffer è ora costituito da un array di N messaggi, gestito in modo circolare, come una coda FIFO, tramite i puntatori primo e ultimo; inoltre, invece di un solo semaforo M, è adesso presente un array di N semafori, uno per ogni mittente. La disponibilità di N posizioni nel buffer garantisce che ciascuno degli N mittenti ha la certezza di trovare un elemento del buffer disponibile quando invoca la invio. Infatti, anche in questo caso, a causa della sincronizzazione implicita nella funzione invio, ogni mittente non può inviare un nuovo dato prima che quello eventualmente inviato in precedenza non sia stato ricevuto. Inoltre, al fine di evitare interferenze tra le esecuzioni concorrenti della funzione invio, è necessario un semaforo mutex di mutua esclusione:

```
class porta {
   messaggio buffer[N];
   int primo=0;
   int ultimo=0;
   semaphore M[N]={0,0,...,0};
   semaphore R=0;
   semaphore mutex=1;
   public void invio(T dato){
      int i;
      messaggio mes;
      mes.informazione=dato;
      mes.mittente=PID();
      P(mutex);
      i=ultimo;
      ultimo=(ultimo+1)%N;
      buffer[i]=mes;
      V(mutex);
      V(R);
      P(M[i]);
   }
   public void ricezione(T &dato,PID &mit){
      int i;
      messaggio mes;
      P(R);
      i=primo;
      primo=(primo+1)%N;
      mes=buffer[i];
      V(M[i]);
      dato=mes.informazione;
      mit=mes.mittente;
   }
}
```

9.4.2 Realizzazione di un meccanismo di comunicazione sincrono mediante primitive asincrone

Vediamo adesso come realizzare due funzioni di libreria da utilizzare, rispettivamente, per inviare o ricevere un messaggio di un tipo predefinito T in modo sincrono, supponendo di avere a disposizione le funzioni asincrone già viste nel paragrafo 8.4.1.

Le due funzioni che vogliamo realizzare sono sintatticamente identiche a quelle asincrone già viste nel precedente capitolo, anche se sono semanticamente diverse:

```
void send (T inf,PID proc,int ip);
void receive (T &inf,PID &proc,int ip);
```

La prima ha tre parametri: inf è l'informazione di tipo T da inviare, proc il PID del processo a cui inviarla, ip l'indice della porta, locale a proc, a cui indirizzare il messaggio. Anche la seconda ha tre parametri: inf è il riferimento a una variabile di tipo T alla quale assegnare l'informazione ricevuta, proc il riferimento a una variabile di tipo PID alla quale assegnare il nome del mittente, ip l'indice della porta da cui ricevere. La semantica è quella tipica delle primitive sincronizzate, ovvero sia la send sia la receive costituiscono un punto di sincronizzazione per i processi che le invocano; tali processi, una volta invocate le rispettive funzioni, possono proseguire solo dopo che l'informazione inf è stata trasferita dal mittente al destinatario.

Supponiamo quindi di avere a disposizione le funzioni di libreria viste nel paragrafo 8.4.1. Per evitare confusione, essendo queste sintatticamente uguali a quelle sincrone che vogliamo realizzare, le distingueremo rispetto a queste ultime denotandole con gli identificatori a_send (*asynchronous send*) e a_receive (*asynchronous receive*).

Disponendo delle funzioni di comunicazione asincrone, la realizzazione di quelle sincrone è estremamente banale, come è stato schematicamente illustrato nella figura 7.3. In particolare, ogni processo che intenda eseguire la funzione send deve dichiarare una sua porta locale di tipo signal (il cui indice viene di seguito indicato con ak) dalla quale ricevere, dal processo a cui invia un messaggio, il segnale di *acknowledge* che indichi che il messaggio è stato ricevuto:

```
void send (T inf,PID proc,int ip){
   signal s;
   a_send(inf,proc,ip);
   a_receive(s,proc,ak);
}
```

```
void receive (T &inf, PID &proc, int ip){
   signal s;
   a_receive(inf,proc,ip);
   a_send(s,proc,ak);
}
```

Il mittente invia il messaggio e si pone in attesa del segnale di *acknowledge*; il ricevente si pone in attesa del messaggio e, una volta ricevuto, restituisce al mittente il segnale di *acknowledge*.

9.4.3 Realizzazione delle primitive sincrone nel nucleo del sistema operativo

Vediamo, infine, come realizzare le primitive di comunicazione sincrone direttamente all'interno del nucleo del sistema operativo iniziando, al solito, con un'architettura monoelaboratore. Per semplificare la presentazione, possiamo fare riferimento a quanto visto paragrafo 8.4.1 cercando di utilizzare lo stesso schema, ma con le opportune modifiche che derivano dalla natura sincrona delle primitive che vogliamo realizzare. Ipotizziamo anche di indicare ancora con N il massimo numero di processi mittenti che possono inviare messaggi su una porta. Se, come abbiamo già indicato nella simulazione delle porte con i semafori, associamo a ogni porta un buffer, questo, a differenza di quanto visto con le primitive asincrone, non sarà costituito da una coda di dimensioni illimitate ma bensì da una coda dimensione pari a N in modo tale che ogni mittente non la trovi mai piena.
Possiamo quindi definire il tipo `messaggio` e il tipo `coda_di_N_messaggi` con i quali definire poi il descrittore di una porta (`des_porta`) e il tipo puntatore a un descrittore di porta (`p_porta`):

```
typedef struct {
   T informazione;
   PID mittente;
} messaggio;
typedef struct {
   messaggio buffer[N];
   int primo,ultimo,cont;
} coda_di_N_messaggi;
typedef struct {
   coda_di_N_messaggi coda;
   p_porta successivo;
} des_porta;
typedef des_porta *p_porta
```

Associate al tipo `coda_di_N_messaggi` possiamo poi definire le funzioni per inserire un messaggio in una coda, estrarre un messaggio da una coda e per testare se una coda è vuota. Per semplicità, non riportiamo il codice di tali funzioni, del resto già viste in molte altre occasioni:

```
void inserisci(messaggio m,coda_di_N_messaggi c){
//inserisce il messaggio m nella coda di messaggi c
.............
}
messaggio estrai(coda_di_N_messaggi c) {
//estrae dalla coda di messaggi c un messaggio e lo restituisce
.............
}
boolean coda_vuota(coda_di_N_messaggi c) {
//testa la coda di messaggi c per verificare se il suo buffer è vuoto
.............
}
```

Ricordiamo anche la struttura di un descrittore di processo già esaminata nel precedente capitolo:

```
typedef struct {
   p_porta porte_processo[M];
   PID nome;
   modalità_di_servizio un doppio spazio? servizio;
   tipo_contestocontesto;
   tipo_stato stato;
   PID padre;
   int N_figli;
   des_figlio prole[max_figli];
   p_des successivo;
} des
```

L'unica variante rispetto al caso delle primitive asincrone riguarda il fatto che, ades
so, un processo può sospendersi non solo sulla `receive` ma anche sulla `sen`
Quindi, in questo caso, `tipo_stato` deve ancora consentire di identificare se
stato del processo è <attivo> oppure <sospeso sulla `receive`>, e in questo caso
quale porta o insieme di porte, ma anche se è <sospeso sulla `send`>.

Esattamente come già visto nel precedente capitolo con rispetto alle primiti
asincrone, utilizzando il campo `stato` del descrittore di un processo, possiamo d
finire le seguenti funzioni, da utilizzare per testare lo stato del processo o per re
strare una commutazione di tale stato del processo in esecuzione:

```
boolean bloccato_su(p_des p,int ip) {
   <testa il campo stato nel descrittore del processo di cui p è il puntatore e restituisce il valo
   true se il processo risulta bloccato in attesa di ricevere messaggi dalla porta il cui indice
   campo porte_processo è ip>;
}
void blocca_su(int ip) {
   <modifica il campo stato del descrittore del processo_in_esecuzione per indic
   che lo stesso si blocca in attesa di messaggi dalla porta il cui indice nel campo porte_p
   cesso è ip>;
}
```

L'obiettivo resta ancora quello di definire le seguenti due funzioni di libreria che
compilatore utilizzerà per tradurre le `send` e `receive` di alto livello definite
paragrafi precedenti:

```
void send (T inf,PID proc,int ip);
void receive (T &inf,PID &proc,int ip);
```

Ricordiamo ancora che a livello di funzioni di libreria un processo viene identifica

mediante il suo PID, mentre a livello di nucleo viene identificato mediante il puntatore al suo descrittore (un valore del tipo p_des). Analogamente, una porta a livello di libreria viene identificata mediante un intero che rappresenta l'indice nel vettore delle porte locali al processo ricevente (porte_processo), mentre a livello di nucleo ogni porta viene identificata mediante il puntatore al suo descrittore (un valore del tipo p_porta).

Come nel caso delle primitive asincrone, anche adesso definiamo alcune primitive di nucleo necessarie per implementare le precedenti funzioni di libreria. In particolare le due primitive testa_porta e inserisci_porta, del tutto identiche a quelle viste nel precedente capitolo:

```
void testa_porta(int ip){
// testa la porta di indice ip del processo in esecuzione, bloccandolo se vuota
   p_des esec=processo_in_esecuzione;
   p_porta pr=esec->porte_processo[ip];
   if(coda_vuota(pr->coda)) {
      blocca_su(ip);
      assegnazione_CPU;
   }
}
void inserisci_porta(messaggio mes,PID proc,int ip){
//inserisce il messaggio mes nella porta di indice ip del processo proc e, se
   questo è in attesa sulla porta, lo attiva
   p_des destinatario=descrittore(proc);
   p_porta pr=destinatario->porte_processo[ip];
   inserisci(m, pr->coda);
   if(bloccato_su(destinatario,ip))attiva(destinatario);
}
```

Definiamo, inoltre, la primitiva estrai_da_porta in modo leggermente diverso da quello visto nel precedente capitolo:

```
messaggio estrai_da_porta(int ip) {
// estrae dalla porta di indice ip (porta sicuramente non vuota) del processo in esecuzione
   un messaggio, lo restituisce e attiva il mittente del messaggio ricevuto
   messaggio mes;p_des mit;
   p_des esec=processo_in_esecuzione;
   p_porta pr=esec->porte_processo[ip];
   mes=estrai(pr->coda);
   mit=descrittore(mes.mittente);
   attiva(mit);
   return mes;
}
```

Infatti, rispetto all'analoga primitiva prevista nel precedente capitolo, contiene anche l'operazione mediante la quale, alla fine della primitiva, viene attivato il mittente del messaggio che viene estratto dalla porta: tale processo, con le primitive sincrone, una volta inviato un messaggio resta in attesa che lo stesso venga ricevuto. Per consentire a un processo mittente di rimanere sospeso in attesa che il suo messaggio venga ricevuto definiamo, infine, la seguente ulteriore primitiva:

```
void attendi_ricezione() {
   p_des esec=processo_in_esecuzione;
   esec->stato=<bloccato sulla send>;
   assegnazione_CPU;
}
```

Avendo a disposizione le precedenti quattro primitive è ora possibile definire le du funzioni di libreria che caratterizzano il meccanismo di comunicazione sincrono:

```
void send (T inf,PID proc,int ip){
   messaggio mes;
   mes.informazione = inf;
   mes.mittente=PIE();
   inserisci_porta(mes, proc, ip);
   attendi_ricezione();
}
void receive (T &inf,PID &proc,int ip){
   messaggio mes;
   testa_porta(ip);
   mes=estrai_da_porta(ip);
   proc=mes.mittente;
   inf=mes.informazione;
}
```

Per la realizzazione dei comandi con guardia si può definire una funzione di libre receive_any, analoga a quella vista nel precedente capitolo.

Le stesse considerazioni viste nel precedente capitolo restano valide anche quanto riguarda la realizzazione delle primitive sincrone in ambiente multielabora re o in ambiente distribuito.

9.5 Sommario

In questo capitolo è stato introdotto il meccanismo per la comunicazione sincro In particolare, sono state messe in evidenza le principali differenze fra questo m canismo e quello asincrono illustrato nel precedente capitolo. Tali differenze rigu dano sia l'implementazione del meccanismo sia l'efficienza delle soluzioni ai problemi di interazione tra processi.

Per illustrare più in dettaglio quest'ultimo aspetto, sono stati rivisti alcuni esen già sviluppati nel precedente capitolo, con lo scopo di mettere meglio in evidenz possibili soluzioni che possono essere realizzate con i due diversi meccanismi di municazione.

Relativamente alla realizzazione del meccanismo, sono state illustrate tre dive tecniche. Innanzi tutto, è stato mostrato come sia possibile simulare un meccansi di comunicazione sincrona in un sistema organizzato secondo il modello a mem comune e, in particolare, utilizzando il meccanismo semaforico.

Successivamente è stato illustrato come un meccanismo di comunicazione crono possa essere facilmente realizzato mediante un meccanismo primitivo di asincrono.

Infine, è stata illustrata anche la soluzione che prevede di realizzare lo stesso meccanismo sincrono come meccanismo primitivo offerto direttamente dal nucleo del sistema operativo.

9.6 Note bibliografiche

I meccanismi di comunicazione sincrona sono stati introdotti per la prima volta da Hoare nel lavoro in cui presenta i Communicating Sequential Processes (CSP) [72]. In un successivo libro [73], Hoare descrive la semantica dei CSP introducendo anche la problematica delle guardie di output. Il linguaggio più conosciuto fra quelli che hanno adottato la comunicazione sincrona è sicuramente OCCAM [74]. Questo linguaggio è stato sviluppato alla INMOS direttamentre per le architetture basate sui transputer. Una buona rassegna sull'evoluzione di OCCAM (OCCAM2) e sui transputer si trova nel lavoro di Burns [75].

1(

Chiamate di procedura remot e rendez-vou

Come visto nel paragrafo 7.2, per semplificare le interazioni di tipo cliente/servit è stato proposto un meccanismo di comunicazione e sincronizzazione che prev che il processo mittente (il cliente di un servizio) si sincronizzi con il processo ser tore, sospendendosi in attesa che il servizio richiesto sia stato completamente eseg to e che gli siano arrivati gli eventuali risultati. Da qui il significato dell'aggetti "estesa" con cui viene denotato il tipo di sincronizzazione di questo meccanismo.

Questo meccanismo è noto, generalmente, con il nome di *chiamata di operaz ne remota*. Esiste, infatti, un'analogia semantica col meccanismo relativo alle ch mate di funzioni. Come nel caso di una chiamata di funzione, il programma ch mante continua solo dopo che l'esecuzione della funzione è terminata. La differer sostanziale risiede nel fatto che la funzione (il servizio) viene eseguita, in realtà, motamente da un processo diverso dal chiamante.

Nel seguito, per tenere conto delle diverse modalità di esecuzione del servizio chiesto, introdurremo due distinte notazioni, la chiamata di procedura remota RI (*Remote Procedure Call*) e il *rendez-vous*.

Nel caso della chiamata di procedura remota, per ogni operazione che un proc so cliente può richiedere viene dichiarata, lato server, una procedura e per ogni chiesta di operazione viene creato un nuovo processo servitore con il compito di e guire la procedura corrispondente.

La seconda notazione, introdotta nel linguaggio ADA [76], prevede invece c l'operazione richiesta sia specificata come un insieme di istruzioni che può compa re in un punto qualunque del processo servitore. Il processo servitore utilizza un struzione di input (*accept*) che lo sospende in attesa di una richiesta dell'operazio All'arrivo della richiesta il processo esegue il relativo insieme di istruzioni e i ris tati ottenuti vengono inviati al processo chiamante. Il termine rendez-vous è stato trodotto dai progettisti di ADA per indicare che i due processi rimangono sincron zati per tutto il tempo necessario a svolgere il servizio.

Come si può notare la RPC rappresenta solamente un meccanismo di comuni zione tra processi; la possibilità che più operazioni siano eseguite concorrentemer

comporta che si debba provvedere separatamente a una sincronizzazione tra i processi servitori, per esempio nell'accesso a variabili comuni.

Il rendez-vous combina invece comunicazione con sincronizzazione. Esiste, infatti, un solo processo servitore al cui interno sono definite le istruzioni che consentono di realizzare il servizio richiesto. Tale processo si sincronizza con il processo cliente quando esegue l'operazione di accept.

Nel seguito verranno introdotte le caratteristiche fondamentali delle due notazioni e verrà illustrata la loro applicazione in due linguaggi utilizzati per la programmazione distribuita: Java per RPC e ADA per rendez-vous.

10.1 Chiamata di procedura remota

Le procedure chiamabili da un processo cliente possono essere esportate da uno specifico modulo di programmazione o, più in particolare, da un processo. La proposta dei DP (*Distributed Processes*), dovuta a Brinch Hansen [77], che costituisce il primo esempio di un linguaggio per applicazioni in tempo reale basato sul meccanismo della chiamata di procedura remota, prevedeva che le procedure comuni fossero dichiarate all'interno dei processi. Ogni processo poteva accedere alle sue variabili locali e chiamare procedure comuni definite entro altri processi.

Nel seguito faremo l'ipotesi più generale che esista un unico componente di programmazione, il *modulo*, che contiene le procedure corrispondenti alle operazioni chiamabili dai processi clienti (*procedure entry*). Si supporrà inoltre che il modulo possa contenere anche processi locali. Tali processi, che non vanno confusi con i processi servitori creati per eseguire le operazioni chiamate, possono accedere solo alle variabili locali al modulo e alle procedure locali; la comunicazione con altri processi definiti entro altri moduli avviene chiamando procedure da questi esportate.

I singoli moduli operano in spazi di indirizzamento diversi e possono quindi essere allocati su nodi distinti di una rete.

Si ha:

```
module <nome_del_modulo>
   <dichiarazione delle procedure entry>;
   body
   (
      <dichiarazione delle variabili locali>;
      <inizializzazione delle variabili locali>;
      entry op1(<parametri formali>)(
         <corpo della procedura op1>;)
      .........
      entry opn(<parametri formali>)(
         <corpo della procedura opn>;)
      <dichiarazione di procedure locali>;
      <dichiarazione di processi locali>;
   )
```

Le procedure `entry` sono le procedure esportate dal modulo e che possono essere chiamate da un processo situato su un diverso nodo (*procedure remote*). Le variabili locali rappresentano lo stato della risorsa sulla quale operano le procedure.

La chiamata della procedura remota op_i da parte di un processo cliente avviene tramite la notazione:

```
call <nome_del_modulo>.opi(<parametri formali>);
```

La realizzazione della chiamata di procedura remota appartenente a un modulo comporta che essa sia eseguita da un processo servitore situato sullo stesso nodo. Come si è detto, per ogni richiesta viene creato un processo servitore il cui compito è quello di eseguire la procedura corrispondente. Il meccanismo a supporto della notazione provvederà a fare arrivare la richiesta, sotto forma di nome della procedura e valori dei parametri di ingresso, e a inviare al processo cliente i parametri di uscita, al completamento dell'azione del processo servitore.

Come si è detto, la presenza di più processi servitori, in esecuzione contemporaneamente all'interno di un modulo per soddisfare le richieste di determinati servizi, richiede che si provveda alla loro sincronizzazione nell'accesso alle risorse del modulo

```
module allarme
   entry richiesta_di_sveglia(int intervallo_di_attesa,int id);
   body (
      int time;
      semaphore mutex=1;
      semaphore priv[N]=[0,0,....,0];
      coda_ordinata coda_risveglio;
      // il tipo coda_ordinata rappresenta una coda di elementi ciascuno dei quali contien
         due interi: l'intero sveglia e l'intero id (che identifica un processo cliente). La cod
         è ordinata per valori non decrescenti di sveglia.
      entry richiesta_di_sveglia(int intervallo_di_attesa,int id) (
         int sveglia=time+intervallo_di_attesa;
         P(mutex);
         <inserimento di sveglia e id in un elemento di coda_risveglio>;
         V(mutex);
         P(priv[id]);
      )
      process clock(
         int cliente_da_svegliare;
         <avvia il clock>;
         while(true) (
         <attende per l'interruzione, quindi riavvia il clock>;
         time++;
         P(mutex);
         while (time>=<dell'intero sveglia relativo al primo elemento di
               coda_risveglio>) (
            <rimozione del primo elemento di coda_risveglio e assegnamento del valore
               id contenuto in tale elemento a cliente_da_svegliare>;
            V priv[cliente_da_svegliare];
         )
         V(mutex);
      )
   )
)
```

Figura 10.1 Esempio di RPC: risveglio programmato di processi.

Ciò può essere ottenuto, trattandosi di un modello a memoria comune, con una qualsiasi delle soluzioni viste precedentemente (semafori, monitor).

Per illustrare l'utilizzo della chiamata di procedura remota si consideri il caso di un `modulo allarme`, riportato nella figura 10.1, che ha il compito di risvegliare un insieme di processi clienti che richiedono questo servizio dopo un tempo da loro prefissato.

Ogni processo cliente richiede il servizio di sveglia chiamando la procedura `richiesta_di_sveglia` e passando, come parametri, l'intervallo di tempo per il quale vuole attendere (`intervallo_di_attesa`) e il suo identificatore (`id`).

Ogni richiesta viene servita da un processo servitore, per cui `id` identifica sia il processo cliente che il processo servitore a esso dedicato.

Il processo servitore esegue la procedura `richiesta_di_sveglia` e provvede ad accodare la richiesta del cliente rispettando l'ordine di risveglio. Si noti che l'operazione di accodamento deve avvenire in mutua esclusione con riferimento ad altre richieste servite da altri processi servitori e all'azione del processo `clock`, anch'esso dichiarato nel modulo.

Il processo servitore si sospende quindi su un semaforo privato in attesa di essere risvegliato dal processo `clock`. Poiché cliente e servitore sono sincronizzati per tutta la durata del servizio anche il processo cliente risulta bloccato.

Il processo `clock` tiene aggiornato il tempo corrente e, in mutua esclusione, va a esaminare la coda delle richieste per verificare se un intervallo di attesa si è completato. In caso affermativo esegue una `V` sul relativo semaforo privato, risvegliando il processo servitore e quindi il corrispondente processo cliente.

L'esempio visto mette in evidenza come sia necessario provvedere alla sincronizzazione tra i processi che operano, lato servitore, sulle variabili del modulo.

10.2 Rendez-vous

Le modalità di richiesta di un servizio da parte di un processo cliente è la stessa come nel caso della RPC. In questo caso, tuttavia, l'operazione corrispondente è eseguita da un processo servitore esistente.

Il processo servitore fa uso di un'apposita istruzione per attendere le richieste di servizio e quindi per eseguire il servizio stesso. Nel seguito si farà riferimento alla soluzione adottata dal linguaggio ADA che prevede che l'istruzione d'ingresso sia del tipo:

```
accept <nome_operazione> (in <parametri_ingresso>, out <parametri_uscita>)
   {S1; ... Sn;}
```

dove $S_1; \ldots S_n;$ rappresentano le azioni eseguite dal processo servitore come risposta a una richiesta di esecuzione del servizio <nome_operazione> da parte del processo cliente; `in` sta a indicare i parametri d'ingresso e `out` quelli di uscita.

Se non sono presenti richieste di servizio, l'esecuzione di `accept` provoca la sospensione del processo servitore. All'arrivo della richiesta, il processo servitore esegue $S_1; \ldots S_n;$ utilizzando i parametri d'ingresso; durante l'esecuzione i due processi rimangono sincronizzati. Al termine, i risultati vengono forniti al processo cliente sotto forma di parametri di uscita e i due processi proseguono indipendentemente la loro esecuzione.

Lo stesso servizio può essere richiesto da più processi clienti prima che il processo servitore esegua il corrispondente `accept`. In tal caso le chiamate vengono inserite in una coda associata alla particolare operazione richiesta gestita, generalmente in modo FIFO.

A una stessa operazione possono essere associati più `accept`; pertanto, a una richiesta, possono corrispondere azioni diverse in funzione del punto di elaborazione del processo servitore. Ciò mette in evidenza una netta distinzione con la RPC, che si basa sulla identificazione di una particolare operazione con una determinata procedura.

Le figure seguenti mostrano le possibili sequenze di eventi durante un rendez-vous. Se il ricevente esegue `accept` prima della chiamata da parte del mittente (caso di figura 10.2), esso viene sospeso. Al momento dell'esecuzione della chiamata i parametri d'ingresso sono copiati dal mittente al ricevente, il mittente viene sospeso e il ricevente esegue il codice associato ad `accept`. Quando tale codice è completato i parametri di uscita vengono inviati dal ricevente al mittente che riprende l'esecuzione dal punto in cui ha fatto la chiamata.

Se la richiesta è effettuata prima che sia stato eseguito `accept` (vedi figura 10.3), il chiamante è sospeso come nel caso precedente. Quando il ricevente esegue `accept` e vi sono più richieste di esecuzione in attesa, viene scelta quella che è in attesa da più tempo; viene quindi eseguito il codice associato all'istruzione `accept` e tutto procede come nel caso precedente.

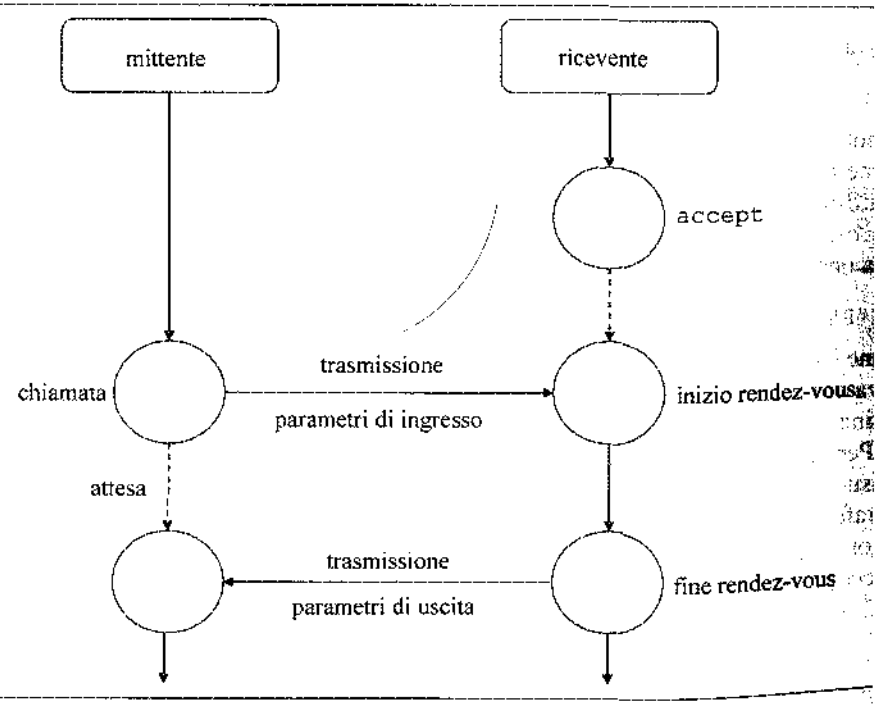


Figura 10.2 Possibili sequenze di eventi in una chiamata di procedura remota.

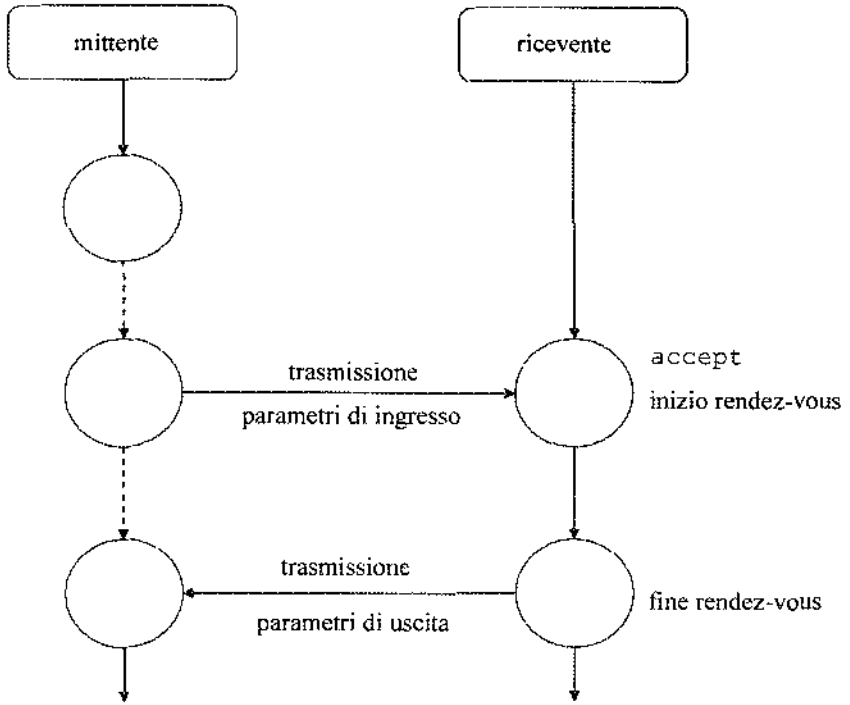


Figura 10.3 Possibili sequenze di eventi in una chiamata di procedura remota.

Si noti che il meccanismo di rendez-vous consente di realizzare, in modo semplice, forme di interazione del tipo *comunicazione sincrona* già visto nel paragrafo 7.2. È sufficiente infatti che la chiamata abbia un solo parametro di ingresso e l'istruzione `accept` abbia un corpo costituito dal solo assegnamento del valore del parametro a una variabile, sia cioè del tipo:

```
accept <nome_operazione> (in x:T) {B=x;}
```

Infine l'accoppiamento tra una chiamata priva di parametri e una istruzione `accept` priva di corpo rappresenta la trasmissione e il relativo riconoscimento di un segnale di sincronizzazione tra due processi.

Per illustrare il meccanismo di rendez-vous si consideri l'esempio del produttore/consumatore. Nell'esempio verranno utilizzati i *comandi con guardia* (visti nel paragrafo 7.3) per consentire a un processo servitore di attendere o di selezionare, tra n possibili, una particolare richiesta. In questo caso, la guardia di input è costituita, invece che da una `receive`, dall'istruzione `accept`. Per il processo `buffer`, si ha il codice riportato nella figura 10.4.

```
module buffer_limitato
   process buffer{
      entry inserimento(in messaggio dato);
      entry preleva(out messaggio dato);
      messaggio buff[N];
      int testa=0;int coda=0;
      int contatore=0;
   do
      [] (contatore <N);accept inserimento(in messaggio dato)
         {buff[coda]=dato;} ->
            contatore++;
            coda=(coda+1)%N;
      [] (contatore >0);accept preleva(out messaggio dato)
         {dato=buff[testa];} ->
            contatore--;
            testa=(testa+1)%N;
   od;
}
```

Figura 10.4 Gestione di un buffer di dimensioni limitate.

Per i processi produttori e consumatori si ha invece il costrutto riportato nella fig 10.5.

Si noti come la sincronizzazione tra processo chiamante e processo chiamato limitata alle sole istruzioni collegate ad `accept`.

Nel processo chiamante l'individuazione dell'insieme di istruzioni e del proce che deve eseguirle avviene tramite la notazione `nome_processo.nome_operazione`, dove `nome_operazione` è una delle operazioni `entry` dichiarate processo servitore e, al tempo stesso, l'identificatore associato a un'istruzione a `cept`.

Si noti, infine, che il meccanismo illustrato può risultare, per alcune applicazi del modello cliente/servitore, di difficile uso e di minore potenza espressiva. Ciò

```
process produttore_i{          /*(1<=i<=n)*/
   messaggio dati;
   for(;;){
      <produci dati>;
      call buffer.inserimento(dati);
   }
}
process consumatore_j{          /* (1<=j<=m) */
   messaggio dati;
   for(;;){
      call buffer.preleva(dati);
      <consuma dati>;
   }
}
```

Figura 10.5 Processi produttori e consumatori.

viene in particolare quando la decisione se servire o no una richiesta dipende, oltre che dallo stato della risorsa, anche dai parametri della richiesta stessa. In questo caso infatti, la guardia logica che condiziona l'esecuzione dell'azione richiesta deve essere espressa anche in termini dei parametri d'ingresso. È pertanto necessaria una doppia interazione tra processo cliente e processo servitore, la prima per trasmettere i parametri della richiesta e la seconda per richiedere il servizio.

Nell'ipotesi di un numero limitato di differenti richieste si può ottenere una semplice soluzione al problema associando a ogni richiesta una differente operazione di servizio (*vettore di operazioni di servizio*) [76].

Si consideri per esempio il caso del `process allarme` il cui compito sia di inviare una segnalazione di sveglia a un insieme di processi che richiedono questo servizio, dopo un tempo da essi stabilito. I servizi offerti da questo processo sono `richiesta_di_sveglia`, per trasmettere a `process allarme` l'informazione relativa all'intervallo di tempo che deve passare prima del suo risveglio, e `svegliami`, per chiedere di essere risvegliato al tempo definito.

Il processo `allarme` serve anche, a sua volta, un processo orologio che periodicamente invia richieste `tick` per tenere traccia del tempo. L'ordine con cui il processo `allarme` risponde alle richieste esterne dipende solo dal parametro (`intervallo_di_attesa`) trasferito con la richiesta. Si ha quindi il codice riportato nella figura 10.6.

```
module risveglio_processi
   process allarme{
      entry tick;
      entry richiesta_di_sveglia(in int attesa);
      entry svegliami[first..last];
      typedef struct {
         int risveglio;
         int intervallo;} dati_di_risveglio;
   in tempo;
   dati_di_risveglio tempo_di_sveglia[N];
   int intervallo_di_attesa;
   do
      []accept tick; -> tempo++;
      []accept richiesta_di_sveglia(in int attesa)
         {intervallo_di_attesa=attesa} ->
         <inserimento di risveglio=tempo+intervallo_di_attesa e di intervallo=attesa in un elemento del vettore tempo_di_sveglia in modo da mantenere tale vettore ordinato secondo valori non decrescenti di risveglio>;
      [](tempo==tempo_di_sveglia[1].risveglio);
         accept svegliami[tempo_di_sveglia[1].intervallo];
            -> <riordinamento del vettore tempo_di_sveglia>;
   od;
}
```

Figura 10.6 Risveglio programmato di un insieme di processi.

Un processo che intende sospendersi per un tempo `T` esegue le seguenti chiamate processo allarme:

```
allarme.richiesta_di_sveglia(T);
allarme.svegliami[T];
```

Si noti che esistono più servizi `svegliami`, uno per ogni possibile valore di `T`. dichiarazione `svegliami[first..last]` definisce un vettore di entry selez nabili attraverso un indice di valore compreso tra `first` e `last`. Ciò consente processi clienti di sospendersi sulla coda associata al particolare valore di `T`, tram la chiamata `allarme.svegliami[T]` (dove `T` serve per identificare un parti lare servizio del vettore), e consente al processo `allarme` di risvegliare, tramite l secuzione di `accept svegliami[tempo_di_sveglia[1].intervall` proprio il processo per il quale è valida la condizione espressa dalla guardia logic

Si noti che possono esistere più processi per i quali è valida la condizi `(tempo==tempo_di_sveglia[i].risveglio)` (con `i>1`). Il loro risveg è garantito dal comportamento ciclico del processo `allarme` (nell'ipotesi ch tempi di aggiornamento del clock siano superiori rispetto ai tempi di esecuzione servizio).

10.3 Linguaggio ADA

ADA [76] fu sviluppato per conto del Dipartimento della Difesa degli Stati Uniti me standard per le applicazioni di tipo militare, comprese le applicazioni in ten reale. Per il periodo in cui fu sviluppato, il linguaggio presentava caratteristiche t temente innovative sia nella parte sequenziale che in quella concorrente.

Per quanto riguarda l'aspetto della gestione della concorrenza, ADA costitui un esempio di linguaggio che adotta, come strumento di interazione tra i proce quello del rendez-vous introdotto nel precedente paragrafo. In una più recente v sione di ADA [78] sono stati introdotti altri strumenti quali i `protected type`, mili al monitor, e l'istruzione `requeue` per dare al programmatore maggior c trollo sulla sincronizzazione e sullo scheduling. Nel seguito ci si soffermerà s mente sugli aspetti relativi alla gestione dell'interazione tra i processi, rinviando la parte sequenziale a [76] [78]

I processi, chiamati *task*, possono essere creati, attivati e terminati dinamicam te. Due sono i meccanismi di creazione e attivazione dei task. Il primo segue le *s pe rule* del linguaggio: tutti i task dichiarati localmente a un blocco vengono cr quando sono elaborate le corrispondenti dichiarazioni e, quindi, quando il contr raggiunge il blocco in questione. I task così creati vengono quindi inizializzati (a vati) e la loro esecuzione avviene in parallelo a quella del blocco.

Oltre alla dichiarazione di task esiste la possibilità di dichiarare *task type*. Di task type possono poi essere successivamente dichiarate varie istanze che condivi no lo stesso corpo.

Anche la terminazione dei task segue le scope rule del linguaggio: il compl mento del blocco in cui i task sono dichiarati è condizionato dalla terminazion tutti i task. Esistono condizioni particolari di terminazione per le quali si rimanda la bibliografia già citata.

In ADA la comunicazione tra task è di tipo asimmetrico e sincrono a rendez-vous. Il rendez-vous tra due task viene stabilito quando entrambi esprimono la volontà di eseguire una stessa operazione. Tale operazione presenta l'aspetto di una procedura e prende il nome di *entry*. Una entry, definita in un task P e resa visibile all'esterno di P, può essere chiamata da un altro task Q secondo il normale meccanismo di chiamata a una procedura.

Lo schema base di un task prevede una parte di specifica che definisce le operazioni entry e una parte body che contiene la realizzazione di tali operazioni.

Per la parte di specifica si ha

```
task Name is
   <dichiarazione delle entry>;
end;
```

Le `entry` definiscono le operazioni servite dai task e hanno la forma:

```
entry identifier (parametri formali);
```

Per la parte body si ha:

```
task body Name is
   <dichiarazioni locali>;
begin
   <istruzioni del task>;
end Name;
```

10.3.1 Rendez-vous

Una entry, definita in un task P e resa visibile all'esterno di P, può essere chiamata da un altro task Q (contenuto nello scope di P) secondo il normale meccanismo di chiamata a una procedura:

```
call P.entryname (<parametri effettivi>);
```

La comunicazione tra P e Q ha luogo quando P esprime la volontà di eseguire la particolare entry invocata da Q, mediante l'istruzione:

```
accept entryname(in parametri-ingresso, out-parametri uscita);
   do S1;...,Sn; end;
```

dove $S_1;\ldots,S_n;$ rappresentano le azioni eseguite da P quando la comunicazione ha luogo; durante la loro esecuzione il processo Q rimane sincronizzato col processo P.

L'esecuzione di `accept entryname` da parte di P sospende il task fino a quando non vi è una chiamata di `entryname`. In quel momento i parametri effettivi sono copiati nei parametri di ingresso e viene eseguita la lista di istruzioni; al loro completamento, i risultati sono copiati nei parametri di uscita. A questo punto cessa la sincronizzazione tra P e Q ed entrambi continuano la loro esecuzione.

Una stessa entry può essere invocata da più processi prima che il task che la definisce esegua la corrispondente istruzione `accept`. In tal caso le chiamate vengono inserite in una coda associata a tale entry gestita in modo FIFO.

A una stessa entry possono essere associate più `accept`. Pertanto a esse posso in generale, corrispondere azioni diverse in funzione del punto di elaborazione task che la definisce. L'istruzione:

```
select
      accept entry1 do...end;
   or accept entry2 do...end;
   ....
   or accept entryn do...end;
end select;
```

rappresenta la forma più semplice di comando con guardia in ADA e viene usata indicare che un task può comunicare con un qualsiasi altro task che abbia chiam una delle entry elencata nell'istruzione `accept`. Se nessuna di esse viene chiam il task si pone in attesa; se una sola è stata chiamata, essa viene immediatamente cettata; se più entry sono già state chiamate, ne viene scelta una in modo non de ministico.

Per permettere a un task di selezionare (o impedire) un insieme di comunicazi sulla base del suo stato interno, ADA consente di premettere guardie logiche, pre dute dalla parola chiave `when`, innanzi a ogni alternativa di una istruzione `sele` (le alternative prive di guardie sono implicitamente precedute da un'espressione gica di valore vero). Si ha cioè:

```
select
      when cond1→accept entry1 ....
   or when cond2→accept entry2 ....
   ...
   or when condn→accept entryn ....
end select;
```

L'esecuzione dell'istruzione select avviene in questo caso nel seguente modo:

a) vengono valutate le condizioni. Le alternative per le quali la condizione è vengono dette aperte;
b) viene scelta arbitrariamente una delle alternative aperte per cui è immediatam possibile il corrispondente rendez-vous; se nessun rendez-vous è possibile, s tende che vi sia tale possibilità.

Ovviamente il caso in cui tutte le guardie siano di valore falso rappresenta un er di programmazione che genera un'eccezione (eccezione *program error*).

Una comunicazione tra processi in ADA, una volta stabilita attraverso i mecc smi visti precedentemente, non può venire sospesa. Ciò, se da un lato consent mantenere per il task il normale significato di processo sequenziale, influenza d altro lato la potenza espressiva del linguaggio relativamente a particolari categ di problemi quali, per esempio, algoritmi di assegnazione delle risorse.

In particolare, con i soli meccanismi di comunicazione e sincronizzazione fin esaminati, risultano di difficile definizione politiche dipendenti:

a) dal tipo della richiesta;
b) da parametri associati al task, come già discusso nel paragrafo 10.2 relativ meccanismo generale di rendez-vous.

Per consentire l'espressione di tali politiche, ADA utilizza il concetto di famiglie di entry già illustrato precedentemente:

```
entry entryname(first..last)(in..out..);
```

Tramite questo costrutto è possibile suddividere una coda di attesa in un insieme di code selezionabili individualmente.

Il linguaggio ADA presenta meccanismi di comunicazione e sincronizzazione mediante i quali è possibile realizzare sia il concetto classico di mutua esclusione dei task su risorse comuni, sia segnalamenti espliciti tra task. Entrambi questi obiettivi sono infatti raggiunti tramite un unico meccanismo base: l'accoppiamento tra chiamata a una entry e l'istruzione `accept`. Tale meccanismo consente la realizzazione di forme di rendez-vous sia stretto (nel caso in cui una entry abbia solo parametri di ingresso e il corpo dell'istruzione `accept` si limiti a trasferire il valore dei parametri in variabili locali al task) sia esteso, mediante le quali si ottiene la sequenzializzazione delle comunicazioni con uno stesso task e quindi la mutua esclusione dei task chiamanti nell'uso della risorsa astratta rappresentata dal task chiamato.

L'accoppiamento tra una chiamata a una entry priva di parametri e un'istruzione `accept` priva di corpo rappresenta la trasmissione e il relativo riconoscimento di un segnale di sincronizzazione tra due task.

Il costrutto select consente inoltre di eseguire delle chiamate condizionali di una entry (*conditional entry call*). La sequenza di istruzioni:

```
select
   <chiamata di una entry>;
   else <comandi>;
end select;
```

è equivalente alla chiamata di una `entry`, se il processo in cui essa è definita è sospeso su una `accept` per la `entry` stessa; nel caso opposto vengono eseguiti i comandi che seguono la clausola `else`. Ciò consente a un processo di non sospendersi in attesa dell'accettazione della chiamata delle `entry` e può quindi risultare utile per le applicazioni in tempo reale.

Un'altra possibilità di uso della `select` è quello di esprimere chiamate cui deve essere assicurato un servizio entro un tempo prefissato. In caso contrario la chiamata viene annullata e il task chiamante prosegue l'esecuzione.

Per esempio la sequenza di istruzioni:

```
select
   <chiamata di una entry>;
or
   delay <intervallo di tempo>;
end select;
```

specifica che il task chiamante attende, per l'esecuzione del comando, al più un tempo pari a un intervallo fissato.

Per concludere, occorre precisare che ADA ammette la possibilità che più task condividano variabili globali; ciò accade in virtù del fatto che la descrizione delle unità del programma da eseguire in parallelo (definizione dei task) è incapsulata nella struttura a blocchi del linguaggio: quindi il programma in ADA può essere visto come il corpo di un task immaginario che racchiude le definizioni innestate di al task. Ciò introduce una gerarchia statica tra i task che possono utilizzare le variab globali, secondo le regole di visibilità degli identificatori del linguaggio.

Tuttavia ADA, nella sua versione originale, non presenta alcun meccanismo p realizzare accessi diretti a variabili globali, oltre ai meccanismi di comunicazione sincronizzazione visti precedentemente.

10.3.2 Protected type

Nella versione successiva [78], la parte concorrente del linguaggio è stata arricch di nuovi costrutti, tra cui i più significativi sono i `protected type` e la primiti `requeue`.

Un `protected type` incapsula una o più strutture dati condivise tra i task ne sincronizza l'accesso. Le istanze di un `protected type` sono chiamate *pr tected object* e sono molto simili al costrutto monitor. Lo schema generale della p te di specifica di un `protected type` è il seguente:

```
protected type Name is
   <dichiarazioni di funzioni, procedure o entry>;
private
   <dichiarazione di variabili>;
end Name;
```

Lo schema per la parte `body` ha la forma:

```
protected body Name is
   <corpo delle funzioni, procedure o entry>;
end Name;
```

Come si vede un `protected type` può esportare funzioni, procedure o `entr` Le funzioni consentono solo un accesso in lettura alle variabili dichiarate `privat` mentre le procedure e le `entry` consentono un accesso in lettura e scrittura di ti esclusivo, come nel caso del monitor. La differenza tra procedure ed `entry` è cl in queste ultime, è possibile specificare che l'accesso alle variabili `private` è co sentito solo se è verificata una condizione di sincronizzazione. A differenza del m nitor non vengono tuttavia utilizzate le variabili condizione per sospendere i ta nell'ipotesi che la condizione di sincronizzazione non sia verificata. Viene utilizz invece una condizione a più alto livello che fa uso della clausola `when B`. Se la co dizione di sincronizzazione non è verificata, la chiamata viene accodata e il ta chiamante viene bloccato fino a che la chiamata non è stata completata e il p `tected type` liberato; diversamente, la chiamata viene accettata ed eseguita modo protetto. Al termine della sua esecuzione, vengono esaminate le condizioni sincronizzazione di tutte le `entry`; tutte le chiamate accodate, a condizioni diven vere, vengono eseguite una alla volta in modo esclusivo (cioè senza rilasciare il v colo di mutua esclusione). Al termine, il `protected type` viene liberato.

Si consideri l'esempio del produttore/consumatore riportato nella figura 10.7.

Se `Get` è chiamata quando `Nof_Items` è zero, la chiamata viene accoda Quando un altro task chiama `Put`, `Nof_Items` viene incrementato. Al termine `Put`, la condizione di `Get` è diventata vera e quindi la chiamata accodata viene presa sbloccando il corrispondente task.

```
protected type Integer_Bounded_Buffer is
   entry Put (I:in Integer);
   entry Get (I:out Integer);
private
   Buffer: array (1..10) of Integer;
   First, Last : Natural:=1;
   Nof_Items: Natural:=0;
end Integer_Bounded_Buffer;
protected body Integer_Bounded_ Buffer is
   entry Put (I:in Integer)
         when Nof_Items < Buffer
begin
      Buffer(Last):=I;
      Last:=Last mod Buffer;
      Nof_Items:=Nof_Items+1;
   end Put;
   entry Get (I:out Integer)
      when Nof_Items>0 is
   begin
      I:= Buffer(First);
      First:=First mod Buffer;
      Nof_Items:= Nof_Items-1;
   end Get;
end Integer_Bounded_Buffer;
```

Figura 10.7 Gestione di un buffer di dimensioni limitate.

```
protected type Punto_di_sincronizzazione is
   procedure Pronto;
private
   entry esecuzione_task;
   count: Integer:=0;
   tutti-pronti: Boolean:=False;
end Punto_di_sincronizzazione;
protected body Punto_di_sincronizzazione is
   entry Pronto is begin
      count:=count+1;
      if count<N then
         requeue esecuzione_task;
      else
         count:=count-1; tutti_pronti=True;
      end if;
   end;
   entry esecuzione_task when tutti_pronti is begin
      count:=count-1;
      if count=0 then tutti_pronti:=False;
      end if;
   end;
end Punto_di_sincronizzazione;
```

Figura 10.8 Utilizzo della primitive `requeue`.

Come esempio d'uso della primitiva `requeue` (vedi figura 10.8), si consideri il ca so di n processi la cui esecuzione è vincolata dal verificarsi della condizione che tut ti i processi siano pronti per eseguire. Ogni processo utilizza, per segnalare la pro pria disponibilità, la procedura `Pronto` del `protected type Punto_di_sin cronizzazione`. La procedura viene completata solo quando tutti i processi han no dichiarato la propria disponibilità. Diversamente, le chiamate alla procedura ven gono accodate alla entry `esecuzione_task` mediante la primitiva `requeue`. A completamento della procedura `Pronto` (arrivo dell'ennesimo processo) la condi zione di sincronizzazione della `entry esecuzione_task`, `tutti_pronti` di venta vera e tutte le chiamate accodate possono essere eseguite, sbloccando i relativ processi.

10.4 Java: Remote Method Invocation (RMI)

Nel capitolo 6 è stato affrontato il problema della sincronizzazione, tra più threa operanti sulla stessa macchina virtuale Java, nell'accesso a oggetti comuni. In quest paragrafo verrà affrontato il problema della realizzazione in Java del meccanism RPC in programmi distribuiti.

Il meccanismo RPC prevede che l'interazione tra client e server avvenga tramit chiamate di procedure, la cui specifica (interfaccia) è nota a livello client e la cui rea lizzazione è demandata al server. Poiché le operazioni su oggetti in Java sono chia mati *metodi*, l'equivalente della RPC in Java prende il nome di RMI (*Remote Me thod Invocation*).

Il RMI consente a un client di chiedere l'esecuzione di un metodo appartenente un *oggetto remoto*. Col termine remoto si definisce un oggetto i cui metodi possono essere chiamati da una macchina virtuale Java diversa da quella su cui l'oggetto re moto risiede. Le due macchine virtuali possono appartenere a macchine fisiche di ferenti (questo sarà il caso che verrà preso in considerazione nel seguito). Come vedrà, ciò avviene utilizzando una sintassi identica a quella per gli oggetti locali tr mite il supporto di due package di libreria `java.rmi` e `java.rmi.server`.

10.4.1 Componenti di un'applicazione

In un'applicazione che utilizza RMI sono presenti tre componenti: l'*interfaccia r mota*, che specifica i metodi che possono essere chiamati da remoto, una *classe se ver*, che implementa i metodi definiti nell'interfaccia, e uno o più *client*, che chiam no i metodi remoti.

Nel seguito verranno illustrate le proprietà dei tre componenti prendendo con esempio il caso, molto semplice, di un oggetto remoto i cui metodi hanno il compi di operare su una variabile intera incrementandone il valore di un'unità (`Incre ment`), azzerandone il valore (`Reset`) e fornendone il valore corrente (`GetValue`

Interfaccia remota L'interfaccia remota deve:

a) essere pubblica;
b) estendere l'interfaccia `java.rmi.Remote`, che consente di classificare l'inte faccia come *interfaccia remota*, ovvero i suoi metodi possono essere chiam

```
//definizione interfaccia RemoteICounter
import java.rmi.*
public interface RemoteICounter extends Remote{
public void Increment() throw RemoteException;
   public void Reset() throw Remote exception;
   public int Getvalue() throw RemoteException;
}
```

Figura 10.9 Definizione dell'interfaccia remota.

da una macchina virtuale non locale: solo i metodi specificati in un'interfaccia remota possono essere chiamati da remoto. Come si vedrà, ogni oggetto remoto deve implementare un'interfaccia remota.

c) ogni metodo deve dichiarare `java.rmi.RemoteException`. Un client che utilizza un metodo remoto deve poter essere avvertito del verificarsi di qualche errore durante la comunicazione. A questo scopo, in ogni metodo dell'interfaccia è presente la clausola `throws RemoteException`, dove `Remote Exception` è una classe (`package java.rmi`) per la gestione di eccezioni che possono avvenire durante l'esecuzione di un metodo remoto.

Si ha quindi il costrutto riportato nella figura 10.9.

Classe server La classe cui appartiene l'oggetto remoto deve:

a) estendere la classe `java.rmi.server.UnicastRemotObject`. Tale classe fornisce le funzionalità di base necessarie per tutti gli oggetti remoti. In particolare, il suo costruttore esporta l'oggetto per renderlo disponibile a ricevere chiamate remote. L'esportazione dell'oggetto permette all'oggetto server remoto di attendere le richieste di connessione da parte del client.

b) implementare l'interfaccia remota;

c) definire esplicitamente il costruttore, che deve dichiarare `RemoteException` nella clausola `throws`, dal momento che `UnicastRemoteObject` può lanciare delle *remote exceptions*;

d) creare un oggetto della classe server;

e) registrare il suo nome in modo che sia accessibile da parte del client.

Si ha quindi il costrutto riportato nella figura 10.10.

Per quanto riguarda la registrazione del nome, si deve notare che, poiché l'oggetto remoto risiede su una macchina diversa da quella sulla quale opera il client, è necessario che l'oggetto sia identificabile dal client in modo unico.

A questo scopo, il server deve provvedere a registrare l'oggetto remoto nel *registro dei nomi remoto*, contenuto sulla stessa macchina del server, stabilendo una corrispondenza tra il nome locale dell'oggetto e un nome globale che i client possono utilizzare per fare riferimento a esso.

Per convenzione, si adotta come nome globale una stringa del tipo:

```
rmi://hostname:port/pathname;
```

```
//Definizione della classe RemoteICounterserver
import java.rmi.*;
import java.rmi.server.*
class RemoteICounterserver extends UnicastRemote Object
        implements RemoteICounter{
      private int value=0;
      public void Increment(){value++;}
      public void Reset(){value=0;}
      public int Getvalue(){return value;}
      //costruttore per inserire la clausola throws
      public RemoteICounterserver() throws RemoteException{
         super();
      }
      public static void main (String[] args) {
         try {
      //creazione di un oggetto di RemoteICounterserver
      RemoteICounterserver Counter=
         new RemoteICounterserver();
      //registrazione del nome
      String name=rmi://hostname/mycounter;
      Naming.rebind(name,Counter);
      System.out.println(name+''is running'')
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Figura 10.10 Definizione della classe server.

dove `hostname` identifica l'indirizzo internet della macchina sulla quale è in cuzione l'oggetto remoto, `port` il numero di porta (valore di default 1099) s quale è in attesa un programma (`rmiregistry`) che ha il compito di gestire i gistro dei nomi remoto, `pathname` è il nome usato dal client per fare riferim all'oggetto remoto.

Nell'esempio considerato la stringa ha la forma:

```
rmi://hostname/mycounter;
```

Il main del server, dopo aver creato l'oggetto remoto `Counter`, provvede a r strare un riferimento a esso (normalmente la sua `stub`, vedi paragrafo success nel registro dei nomi remoto, stabilendo una corrispondenza tra tale riferimento nome globale `rmi://hostname/mycounter`, utilizzando il metodo `rebi` della classe `java.rmi.Naming`.

[1] Il metodo `rebind`, a differenza di `bind`, sostituisce un eventuale nome di oggetto remot gistrato in precedenza. Questa caratteristica diventa utile quando diventa disponibile una n versione di un oggetto server remoto.

Il programma `rmiregistry` viene messo in esecuzione in background sulla macchina server tramite il comando `rmiregistry port&`. Si noti infine che il metodo `main` dell'esempio stampa una linea per indicare che è in esecuzione.

Stub e skeleton L'oggetto remoto `Counter`, istanziato dalla classe `RemoteICounterserver`, va in esecuzione sul lato server della nostra applicazione. D'altra parte il client chiama i metodi dell'oggetto remoto come se fossero locali (nella stessa JVM). Per risolvere questo problema il sistema RMI Java mette a disposizione un comando, `rmic`, eseguito dopo la compilazione delle varie classi, che genera, per ogni oggetto remoto, due oggetti chiamati *stub* e *skeleton*, cui è affidato il compito di gestire l'interazione tra client e server.

Lo *stub* implementa l'interfaccia remota dell'oggetto, contiene cioè i nomi dei metodi dell'oggetto remoto e l'indicazione dei parametri formali. Quando il client chiama un metodo dell'oggetto remoto in realtà chiama un metodo dello *stub* relativo all'oggetto remoto, i cui compiti sono la preparazione di un messaggio contenente il nome del metodo chiamato e i parametri a esso relativi (*marshalling*) e l'invio dello stesso al server. Sul server il messaggio viene ricevuto dallo *skeleton*, i cui compiti sono chiamare il metodo richiesto dopo aver ricostruito i parametri (*unmarshalling*), aspettare fino al completamento dell'operazione e quindi inviare il risultato allo stub. Lo stub dovrà poi trasferire i risultati al programma client.

Per riassumere, quando il client chiama un metodo dello stub, questo provvede a:

- stabilire una connessione con la JVM remota contenente il corrispondente oggetto remoto;
- preparare e trasmettere i parametri relativi al metodo chiamato (*marshalling*);
- bloccarsi in attesa dei risultati dell'esecuzione del metodo remoto (corrispondentemente anche il client rimane bloccato);
- ricostruire i risultati o le eccezioni (*unmarshalling*) all'arrivo del messaggio da parte dello skeleton dell'oggetto remoto;
- trasferire il controllo al client.

Analogamente, quando lo skeleton riceve una chiamata di metodo, esso provvede a:

- ricostruire (*unmarshalling*) i parametri per il metodo remoto;
- chiamare il metodo richiesto dell'oggetto remoto;
- preparare e trasmettere (*marshalling*) il messaggio di ritorno allo stub chiamante.

È importante sottolineare che tutte le operazioni necessarie per la comunicazione di rete tra il client e il server sono delegate da stub e skeleton all'apposito supporto fornito dal sistema java RMI e risultano pertanto trasparenti a chi scrive i programmi del client e del server[2] (vedi figura 10.11).

[2] Nella versione Java 2 SDK, Standard Edition, v. 1.2, è stato introdotto un nuovo protocollo stub che elimina la necessità dello skeleton.

```
//definizione della classe Client
import java.rmi.*;
class Client {
   public static void main(String[]) args) {
   try {
   //installazione dello standard RMI security manager
      System.setSecurityManager(new.RMISecurityManager());
   //riferimento all'oggetto remoto
      RemoteICounter cont=(RemoteIcounter)Naming.lookup
      (''rmi://hostname/mycounter'');
   //operazioni sull'oggetto remoto
   int value;
   cont.Increment () ;
   ...
   cont.Reset() ;
   value=cont.Getvalue() ;
   }
   Catch (Exception e) {
   System.err.println(e);
   }
}
```

Figura 10.11 Definizione della classe client.

Class client La classe `client` ha come scopo fondamentale la chiamata dei n todi dell'oggetto remoto. Queste chiamate vengono fatte, come si è detto, utilizzando lo stub dell'oggetto remoto che risiede, assieme allo skeleton, sulla macchina s ver.

Utilizzando il metodo `lookup` della classe `Naming`, passando come parame la stringa contenente il nome del server che ospita l'oggetto remoto e il nome glol le con cui l'oggetto è stato registrato, si ottiene un riferimento all'oggetto che, di to, è lo stub dell'oggetto stesso e il cui tipo quindi è quello dell'interfaccia dell'o getto remoto.

Poiché lo stub dell'oggetto remoto (e tutte le altre classi che fanno parte de RMI API) vengono scaricate dinamicamente dalla rete, si pone il problema della curezza. Il client utilizza a questo scopo lo standard RMI `Security Manage` che va installato prima di tentare una connessione con un oggetto remoto.

Si ha pertanto lo schema riportato in figura 10.11.

10.4.2 Compilare ed eseguire il server e il client

Una volta create le classi corrispondenti all'interfaccia remota (`RemoteICou ter`), al server remoto (`RemoteICounterserver`), al client (`Client`), i pa di esecuzione del programma sono i seguenti:

1. compilazione delle classi con `javac`;
2. compilazione della classe `RemoteICounterserver`, utilizzando il comp tore e `rmic`, al fine di produrre una `classe stub`. Come si è detto, un ogg

di una classe stub consente al client di invocare i metodi remoti dell'oggetto server. L'oggetto stub riceve tutte le chiamate di metodo remote e le passa al sistema Java RMI, che esegue le operazioni di networking che permettono al client di collegarsi al server e che interagiscono con l'oggetto server remoto.

3. messa in esecuzione sul server del comando `rmiregistry` (se non si indica il numero di porta, questa è per default la 1099) in modo che il registro dei nomi remoto sia legato alla porta 1099;
4. attivazione del server mediante il comando `java RemoteICounterserver`. Il metodo `main` della classe `RemoteICounterserver` crea l'oggetto `Counter`. L'esecuzione del costruttore `RemoteICounterserver` chiama il costruttore della superclasse `UnicastRemoteObject` che esporta l'oggetto remoto. Il main lega l'oggetto remoto al registro dei nomi remoti, usando il metodo `rebind` della classe `Naming`. Quando `RemoteICounterserver` è pronto a ricevere chiamate remote, verrà stampato un messaggio sul server per indicare che è in esecuzione.
5. attivazione del client su un nodo della rete mediante il comando `java Client`.

Come si è visto, dovranno essere inoltre importate dinamicamente, durante l'esecuzione del programma, le classi `java.rmi.*` e `Java.rmi.server.*`.

Si noti che nell'esempio riportato si è supposto per semplicità che l'oggetto remoto non riceva parametri. Il solo metodo `getvalue` restituisce un valore intero.

Nel caso la struttura dati, passata tra client e server, fosse stata più complessa, si sarebbe posto il problema della sua traduzione in una sequenza di byte inserita in un file da spedire su una connessione di rete. Questa operazione e quella corrispondente di ricostruzione della struttura dell'oggetto, dalla sequenza di byte, una volta arrivato a destinazione vanno sotto il nome, rispettivamente, di *serializzazione* e *deserializzazione*.

Per risolvere automaticamente questi problemi, Java richiede che una classe, i cui oggetti devono essere serializzati, implementi l'interfaccia *Serializable* [64].

La presenza di più client renderebbe evidentemente più complessa la soluzione, soprattutto nel caso che l'oggetto remoto dovesse essere usato in mutua esclusione. Si lascia ai lettori, come esercizio, la soluzione del problema.

10.5 Sommario

Il capitolo introduce due nuovi meccanismi di comunicazione tra processi, utilizzati prevalentemente in ambiente distribuito: la *chiamata di procedura remota* (RPC) e il *rendez-vous*.

Viene dapprima precisata la differenza tra i due meccanismi. Entrambi vengono utilizzati da un processo client per chiedere, lato server, l'esecuzione di un determinato servizio e, in entrambi i casi, il client rimane in attesa del completamento del servizio e dell'arrivo dei risultati. La RPC rappresenta, tuttavia, solo un meccanismo di comunicazione tra client e server; all'arrivo della richiesta viene creato, lato server, un nuovo thread che ha il compito di realizzare il servizio. Poiché più client possono richiedere lo stesso servizio contemporaneamente, oppure servizi che condividono le stesse variabili globali, è necessario provvedere esplicitamente alla sin
nizzazione dei thread.

Con il rendez-vous invece il servizio è realizzato da un singolo processo se
che rimane in attesa delle richieste dei client e provvede a soddisfarle una alla v
ed a inviare i risultati ai processi client.

Successivamente, vengono presi in considerazione due linguaggi, ADA e Jav
viene discussa la soluzione in essi contenuta relativamente ai due meccanismi
particolare, con riferimento al linguaggio ADA viene discusso il meccanismo
dez-vous e ne vengono illustrate le proprietà tramite alcuni esempi. Vengono
sentati anche due costrutti, *protected type* e *requeue*, introdotti nella seconda
sione di ADA con l'obiettivo di inserire nel linguaggio alcuni concetti tipici dei
guaggi object oriented (il *protected type* è simile al monitor) e di dare al progr
matore più controllo sulla sincronizzazione e sullo scheduling.

Le proprietà del secondo meccanismo, la chiamata di procedura remota, veng
discusse facendo riferimento alla sua realizzazione nel linguaggio Java dove pre
il nome di RMI (*Remote Method Invocation*). Anche in questo tali proprietà
gono discusse facendo riferimento a un esempio concreto.

In entrambi i casi viene data per scontata la conoscenza di base dei due lingua

10.6 Note bibliografiche

Una trattazione organica delle proprietà della chiamata di procedura remota si
trovare in [79] e, per quanto riguarda la sua implementazione nel kernel di un s
ma operativo, in [80]. La prima proposta di un linguaggio basato sul meccani
della chiamata di procedura remota è dovuta Brinch Hansen che realizzò nel 19
linguaggio DP (Distributed Processes) [77]. Il concetto di distributed processe
proposto per applicazioni in tempo reale controllate da un sistema costituito da
processori privo di memoria comune. Secondo la proposta DP, un programm
tempo reale è visto come un insieme di processi distribuiti che iniziano la loro
cuzione contemporaneamente. Un processo può accedere solo alle sue variabili l
li: non vi sono variabili comuni. Esso può tuttavia chiamare procedure comuni
nite entro altri processi. Una soluzione simile ai DP è rappresentata da MML (*M
ple Micro Language) [81].

RPC è stata introdotta in molti altri linguaggi tra cui Emerald, Argus, Cedar.

Il meccanismo rendez-vous è stato introdotto dai progettisti del linguaggio A
[76] e, indipendentemente, da Andrews nel linguaggio SRL [82] [83].

Il linguaggio ADA, sviluppato da Honeywell Bull, fu scelto nel 1979 a seg
di una gara internazionale indetta dal Department of Defense (DoD) degli Stati U
per fare fronte ai costi crescenti di sviluppo e manutenzione del software. Una
cessiva versione di ADA fu prodotta nel 95. Il meccanismo di sincronizzazion
ADA si trova ampiamente descritto in [78]. Esiste inoltre un sito che contiene in
mazioni aggiornate su ADA [84].

Concurrent C è un altro linguaggio basato su rendez-vous [85] [86]. Esten
linguaggio C con i processi e il rendez- vous e realizza un accept statement più
tente di quello di ADA. Infatti, similmente a quanto proposto da SR, consente c

trodurre, nelle condizioni di sincronizzazione dei parametri e nello statement accept espressioni per lo scheduling dei processi.

Infine il linguaggio Concurrent C++ combina Concurrent C con C++ [87].

Il RMI è stato introdotto in Java nella versione 1.1. Una precisa definizione delle proprietà di RMI si può trovare nel sito [64]. Esempi di uso di RMI si possono trovare in [88] e [89].

Esercizi di riepilog

In questa appendice sono riportati alcuni esercizi sull'utilizzo degli strumenti di s cronizzazione presentati nel testo e relativi alla realizzazione di politiche di gesti delle risorse sia in ambiente a memoria comune che in ambiente a scambio di m saggi.

Gli esercizi proposti sono stati usati dagli autori nelle prove di esame dei corsi loro tenuti presso le Facoltà d'Ingegneria di Bologna e di Pisa. Le soluzioni sono portate sul sito **www.ateneonline.it/ancilotti** associato a questo volume.

Gli esercizi sono stati raggruppati in base ai meccanismi di interazione tra p cessi che devono essere utilizzati nella soluzione dei problemi stessi (semafori, n nitor, primitive di comunicazione asincrone e/o sincrone, rendez-vous). In alc esercizi viene esplicitamente richiesta la conoscenza di più meccanismi contemp neamente, in particolare quando, nota la soluzione mediante un particolare mecc smo, viene richiesta la simulazione mediante meccanismi diversi. Molti degli ese zi possono essere risolti, comunque, utilizzando anche meccanismi diversi da q proposti. In questi casi, è utile risolvere lo stesso esercizio con i diversi meccani al fine di confrontare le diverse soluzioni ottenute.

Sono infine riportate altre due tipologie di esercizi: una relativa all'uso dell breria pthread, l'altra relativa all'uso del linguaggio Java.

E1 Semafori

Esercizio E1.1 In un sistema organizzato secondo il modello a memoria com N diversi processi clienti competono per l'uso di 10 risorse equivalenti allocate d micamente mediante un gestore G. Ogni processo può richiedere (e rilasciare) u due risorse contemporaneamente mediante le seguenti funzioni del gestore:

```
void richiesta1(int &x,int t,int c);
void richiesta2(int &x,int &y,int t,int c);
void rilascio1(int x);
void rilascio2(int x, int y);
```

Nelle due funzioni di richiesta il parametro t denota il tempo (espresso in una

che unità di misura) necessario al processo richiedente per utilizzare le risorse; tramite i parametri x e y, invece, vengono restituiti gli indici delle risorse allocate; il parametro c, infine, denota l'indice (da 0 a N − 1) del processo cliente che richiede le risorse. Analogamente, nelle funzioni di rilascio, i parametri indicano gli indici delle risorse rilasciate. Il gestore alloca le risorse in base al criterio di privilegiare chi impegna per il minor tempo complessivo le risorse stesse, dove il tempo complessivo viene valutato come il prodotto fra il numero delle risorse richieste per il tempo di impegno previsto. In questo senso, una richiesta deve essere esaudita solo se è disponibile un numero di risorse libere sufficienti e se non è sospeso nessun processo con priorità maggiore (dove la priorità è valutata come indicato sopra); in caso contrario il processo richiedente viene sospeso. Viceversa, nessun processo richiedente deve essere sospeso quando la condizione per esaudire la sua richiesta è verificata. Nel caso di richieste con lo stesso valore del tempo complessivo non viene specificato nessun criterio di priorità fra le due richieste.

Realizzare il gestore utilizzando il meccanismo dei semafori.

Esercizio E1.2 Il sistema operativo RC-4000 è stato uno dei primi sistemi organizzati secondo il modello a scambio di messaggi. Nel nucleo di tale sistema sono realizzate quattro primitive per la comunicazione tra processi. I messaggi scambiati sono tutti di uno stesso tipo predefinito T e, per ogni messaggio ricevuto, il processo ricevente è tenuto a restituire al mittente un messaggio di risposta, anch'esso dello stesso tipo predefinito T (ciò per consentire la realizzazione di schemi cliente/servitore). Ogni processo è dotato di una coda (FIFO) d'ingresso nella quale sono collocati tutti i messaggi a lui inviati. Come supporto all'invio dei messaggi e delle risposte il nucleo gestisce un pool di B buffer.

Indicando con la costante intera N_proc il numero di processi del sistema, ogni processo viene identificato mediante un'intero compreso fra 0 e N_proc-1.

Il sistema offre ai processi quattro primitive per l'invio e la ricezione dei messaggi:

- `int send_message(int mit, int ric, T messaggio)`, utilizzata per inviare un messaggio. La primitiva sceglie dal pool dei buffer un buffer libero e restituisce al chiamante l'indice di tale buffer; inserisce in esso l'intero `mit` che identifica il processo mittente e il `messaggio`; quindi inserisce il buffer così riempito nella coda del processo ricevente, identificato mediante l'intero `ric`, attivandolo se questo è bloccato in attesa di messaggi.
- `int wait_message(int &mit, T &messaggio)`, utilizzata per ricevere un messaggio. Se la coda d'ingresso del processo chiamante è vuota, il processo viene sospeso fino a quando un buffer contenente un messaggio viene inserito nella sua coda. Quando uno o più buffer sono disponibili, il primo viene tolto dalla coda e l'indice del buffer estratto viene restituito dalla primitiva. Dal buffer vengono estratti il nome del mittente, che viene restituito tramite il parametro `mit`, e il messaggio, restituito tramite il parametro `messaggio`.
- `void send_answer(T risposta, int indice_buffer)`, utilizzata per inviare una risposta. La `risposta`, passata come parametro, viene inserita nel buffer di indice `indice_buffer` (lo stesso tramite il quale il processo ha precedentemente ricevuto il messaggio a cui risponde). Il buffer con la risposta viene quindi consegnato al processo che ha precedentemente inviato il messaggio se tale processo è in attesa di questa risposta, viene attivato.
- `T wait_answer(int indice_buffer)`, utilizzata per ricevere una risposta. La primitiva blocca il processo chiamante fino all'arrivo della risposta inviata nel buffer di indice `indice_buffer` (tramite il quale lo stesso processo ha precedentemente inviato il messaggio di cui attende la risposta). Quando la risposta è disponibile, viene estratta dal buffer. Il buffer, ormai vuoto, viene quindi rilasciato al pool di buffer disponibili e la risposta restituita al processo chiamante.

Utilizzando i semafori, simulare il precedente meccanismo primitivo. Se necessario si può ipotizzare disponibile la primitiva di sistema `PIE()` (vedi paragrafo 1.3) che, quando invocata, restituisce l'indice (compreso tra 0 e N_proc-1) del processo che è in esecuzione.

Esercizio E1.3 In un sistema che utilizza i semafori come strumento di sincronizzazione si vuole realizzare un meccanismo di comunicazione tra processi che consenta a un processo mittente M di inviare messaggi a tre processi riceventi R1, R2, R3. Il meccanismo è costituito da una mailbox implementata mediante un buffer circolare di 10 posizioni. Ogni messaggio inviato mediante la funzione `invio` dal processo M deve essere ricevuto da tutti i processi riceventi (ciascuno di essi chiamando una propria funzione di ricezione: `ricevi1`, `ricevi2` e `ricevi3` rispettivamente). Ogni ricevente deve ricevere i messaggi nello stesso ordine con cui sono stati inviati, però riceventi diversi possono ricevere lo stesso messaggio in tempi diversi a seconda delle loro velocità. Perciò, un ricevente più veloce, in un certo istante può aver ricevuto più messaggi rispetto a uno più lento.

a) Implementare il meccanismo di comunicazione con le relative funzioni indicando genericamente con T il tipo dei messaggi scambiati.

b) Ripetere la soluzione supponendo che fra i tre processi riceventi venga imposta la seguente priorità negli accessi alla mailbox: R1 massima priorità, R2 priorità intermedia, R3 minima priorità.

Esercizio E1.4 In un sistema organizzato secondo il modello a memoria condivisa n diversi processi mittenti (M_1, M_2, ..., M_n) inviano messaggi a un solo processo ricevente R. Ogni messaggio che un qualunque mittente invia a R è costituito da una stringa di caratteri che termina con "\0". Messaggi diversi possono avere lunghezza diversa (per esempio possiamo assumere che la lunghezza di ogni messaggio sia compresa tra un minimo di 5 e un massimo di 100 caratteri).

I processi comunicano tramite una `mailbox` realizzata tramite un buffer circolare di caratteri di lunghezza pari a 10.

Utilizzando il meccanismo dei semafori, implementare la `mailbox` con le relative funzioni `invio` e `ricezione` in modo tale da garantire la corretta ricezione dei messaggi inviati a R da parte di qualunque mittente e indipendentemente dalla dimensione dei messaggi inviati.

Esercizio E1.5 Una risorsa R è allocata dinamicamente a un insieme di processi mediante un gestore G. Sulla risorsa R ciascun processo può eseguire l'operazione OpA, oppure OpB o ancora OpC. Il gestore mette a disposizione dei richiedenti

operazioni `Richiesta_A`, `Rilascio_A`, `Richiesta_B`, `Rilascio_B`, `Richiesta_C` e `Rilascio_C`, usate, rispettivamente, per richiedere e rilasciare l'uso della risorsa a fine di operarci mediante le operazioni `OpA`, `OpB` o `OpC`. Sono consentite esecuzioni contemporanee della stessa operazione, ma operazioni diverse devono essere eseguite in mutua esclusione.

a) Usando il meccanismo dei semafori, realizzare il gestore `R`, ipotizzando che le richieste relative all'operazione `OpA` siano da privilegiare su quelle relative all'operazione `OpB` e queste su quelle relative all'operazione `OpC`, senza preoccuparsi della starvation.
b) Riscrivere il gestore eliminando la starvation senza implementare nessuna regola di priorità.

Esercizio E1.6 All'interno di un'applicazione vari processi (ciclici) cooperano, secondo i criteri specificati nel seguito, accedendo a una risorsa condivisa `R`: alcuni fra questi processi, arrivati a un certo punto della loro esecuzione, per procedere devono controllare che si sia verificato un certo evento (che indicheremo come *eventoA*). In caso positivo procedono senza bloccarsi altrimenti attendono che l'evento si verifichi. Questo controllo viene effettuato accedendo alla risorsa `R` tramite la funzione `testa_A`. Altri processi sono preposti a segnalare l'occorrenza dell'evento invocando la funzione `segnala_A`. Gli eventi segnalati sono però, per così dire, "consumabili". In particolare, se un processo, invocando `testa_A`, verifica che *eventoA* si è già verificato, procede senza bloccarsi ma resetta l'evento, cioè da quel momento in poi, se un altro processo invoca `testa_A`, questo si deve bloccare in attesa che, tramite `segnala_A`, l'evento sia di nuovo segnalato. Se all'atto dell'invocazione di `testa_A` l'evento non si è ancora verificato, o se verificato è già stato consumato, allora il processo richiedente si blocca. Viceversa, quando viene invocata `segnala_A`, se non ci sono processi in attesa dell'evento, allora questa segnalazione resta disponibile per il primo processo che invochi `testa_A`; se viceversa uno o più processi sono in attesa dell'evento, *tutti* vengono riattivati e l'evento viene consumato. Infine, se all'atto dell'invocazione di `segnala_A` è ancora presente un precedente evento non consumato, il processo segnalante si deve bloccare in attesa che il precedente evento venga consumato.

a) Utilizzando il meccanismo dei semafori, implementare `R` con le due funzioni `testa_A` e `segnala_A`.
b) Supponendo che i processi che segnalano l'evento siano P_0, ..., P_{N-1}, e che ciascun processo, nell'invocare la funzione `segnala_A`, passi il proprio indice (intero tra `0` e `N-1`) come parametro, modificare la precedente soluzione imponendo una priorità tra questi processi, quando devono essere risvegliati dopo un blocco, in modo tale che P_0 abbia priorità su P_1, P_1 su P_2, ecc.

Esercizio E1.7 Un gestore di risorse equivalenti alloca dinamicamente a `N` processi clienti 9 istanze di uno stesso tipo di risorse (`R1`, `R2`, ..., `R9`).

Gli `N` processi clienti (P_0, ..., P_{N-1}) vengono serviti dal gestore privilegiando i processi di indice più basso.

Un processo che chiede una risorsa si blocca se all'atto della richiesta non ci sono risorse disponibili e resta in attesa che una risorsa venga rilasciata. Però il processo rimane bloccato, al massimo, per `t` quanti di tempo successivi al blocco (dove `t` è un intero passato come parametro in fase di chiamata della funzione `richiesta`). Quando un processo viene svegliato per lo scadere del tempo massimo da lui indicato, la funzione `richiesta` termina senza allocare niente al richiedente.

Utilizzando il meccanismo dei semafori, scrivere il codice del `gestore` che offre le tre seguenti funzioni membro:

- `int richiesta(int t,int p)` dove `t` denota il time out espresso come numero di quanti di tempo, mentre `p` denota l'indice del processo richiedente. La funzione restituisce l'indice della risorsa allocata (`1`, ..., `9`) oppure `0` se il processo termina la funzione dopo un time out senza aver allocato niente.
- `void rilascio(int r)` dove `r` denota l'indice della risorsa rilasciata.
- `void tick()` funzione chiamata dal driver delle interruzioni dell'orologio in tempo reale a ogni interruzione di clock.

Esercizio E1.8 In un sistema organizzato secondo il modello a memoria comune si vuole realizzare un meccanismo di comunicazione tra processi che simula una `mailbox` a cui `M` diversi processi mittenti inviano messaggi di un tipo `T` predefinito e da cui prelevano messaggi `R` diversi processi riceventi.

Per simulare tale meccanismo si definisce il tipo `busta` di cui vengono usate `N` istanze (costituenti un pool di risorse equivalenti). Un gestore `G` alloca le buste appartenenti al pool ai processi mittenti i quali, per inviare un messaggio, eseguono il seguente algoritmo:

invio di un messaggio →
1. richiesta al gestore `G` di una `busta` vuota;
2. inserimento nella `busta` del `messaggio`;
3. inserimento della `busta` piena nella `mailbox`.

Analogamente, ogni processo ricevente per ricevere un messaggio esegue il seguente algoritmo:

ricezione di un messaggio →
1. estrazione di una `busta` piena dalla `mailbox`;
2. estrazione del `messaggio` dalla `busta`;
3. rilascio della `busta` vuota al gestore `G`.

a) Realizzare il precedente meccanismo utilizzando i semafori e garantendo che la funzione di `ricezione` sia bloccante quando nella `mailbox` non ci sono `buste` piene e che la funzione di `invio` sia bloccante quando il gestore non ha `buste` vuote disponibili. Indicare, in particolare, come viene definito il tipo `busta` e scrivere il codice del `gestore` e della `mailbox` (per garantire la ricezione FIFO dei messaggi, organizzare la `mailbox` come una lista concatenata di buste piene). Le due funzioni di accesso alla `mailbox` sono le seguenti:
   - `void invio(T messaggio)` per l'invio del messaggio passato come parametro e
   - `T ricezione()` per ricevere un messaggio (messaggio che viene restituito dalla funzione).

   Il gestore alloca le buste vuote ai processi mittenti adottando una politica prioritaria in base a un parametro `priorità`, che ciascun processo indica nel momento in cui chiede una `busta` vuota al `gestore` passandole un parametro in-

tero che può assumere i seguenti valori: 0 (priorità massima), 1 (priorità intermedia) oppure 2 (priorità minima). Per richieste con uguale priorità non viene specificata nessuna particolare politica. Le due funzioni di accesso al gestore sono quindi:

- `int richiesta(int priorità)`
- `void rilascio(int b)`

La funzione `richiesta` restituisce l'indice (compreso tra 0 e N-1) della busta allocata; alla funzione `rilascio` viene passato come parametro l'indice b della busta vuota rilasciata.

b) Modificare la precedente soluzione utilizzando i monitor al posto dei semafori e ipotizzando di adottare una analoga politica di priorità fra i processi riceventi nell'estrarre le buste dalla mailbox.

Esercizio E1.9 Nel meccanismo dei monitor, fra le varie semantiche dell'operazione `signal` su variabili `condition` è stata definita anche la sematica *signal_and_continue*. Tale semantica consente di fornire, insieme all'operazione `signal` che sveglia al più un processo, anche l'operazione `signalAll` (o `signal_broadcast`) che risveglia tutti i processi in attesa sulla variabile `condition`.

Si chiede di descrivere l'implementazione del monitor che adotta tale semantica mediante il meccanismo semafori. In particolare, indicare chiaramente le strutture dati che sono necessarie per tale implementazione, il prologo e l'epilogo di ogni funzione di monitor e le operazioni `wait`, `signal` e `signalAll`.

Spiegare chiaramente e in maniera succinta perché con altre semantiche (con riferimento in particolare alla semantica *signal_and_urgent*) la `signalAll` non viene fornita e come può essere risolto il problema, in questi casi, se vogliamo sveglire tutti i processi in attesa su una `condition`.

Esercizio E1.10 Con riferimento a un linguaggio che adotta il meccanismo del rendez-vous, indicare come tale meccanismo può essere simulato mediante il meccanismo primitivo dei semafori.

a) In particolare, individuare una soluzione facendo riferimento alle seguenti porzioni di codice:

1. in un processo `server` viene dichiarata la entry `E` che viene accettata quando il processo è disponibile a fornire il corrispondente servizio:

```
process Server {
   entry E(in T1 x,out T2 y);
   ............
   T1 a;
   ............
   ............
   accept E(in T1 x,out T2 y) {
      a:=x;
      y:=f(a);
   }
   ............
}
```

2. vi sono `N` processi clienti (`P_1`, `P_2`, ..., `P_N`). In ogni processo cliente viene invocata la entry `E` per ottenere il corrispondente servizio da parte del `server`:

```
process P_i {     //(1≤i<N)
   T1 a; T2 b;
   ............
   call E(a,b);
   ............
}
```

b) Per ognuno dei seguenti casi particolari indicare se è possibile ottimizzare il codice prodotto, e in caso positivo in che modo:

1. sapendo che vi è un solo cliente (N=1);
2. se la entry `E` ha un solo parametro `x` di modo `in`;
3. se la entry `E` non ha parametri e il corpo dell'accept è lo statement nullo.

E2 Monitor

Esercizio E2.1 Il dipartimento di applicazioni multimediali di una società informatica è composto da `N` workstation e utilizza uno storage system centralizzato per l'immagazzinamento dei file multimediali e per il backup dei dischi delle workstation. Tre tipi di processi possono accedere concorrentemente dalle workstation allo storage system: processi `M`, che *leggono* e *scrivono* file multimediali, processi `B` che effettuano il backup e processi `R` che ripristinano i dati di backup. La politica di accesso impone che processi di tipo `M` non possano accedere contemporaneamente a processi di tipo `B` e che, dalla stessa workstation, non possano accedere contemporaneamente processi `R` e `B`.

Implementare una soluzione usando il costrutto monitor per modellare lo storage system e i processi `M`, `R` e `B`, e si descriva la sincronizzazione tra i processi. Nella soluzione dare priorità all'utilizzo delle risorse. Discutere se la soluzione proposta può presentare starvation, e in caso positivo per quali processi, e proporre modifiche e/o aggiunte per evitare starvation.

Nota: in questa soluzione si suppone che ci siano un solo processo `B` e un solo processo `R` da ogni workstation.

Esercizio E2.2 Un monitor offre ai processi di un'applicazione concorrente la sola funzione:

```
int scambia(int x)
```

Tale funzione viene utilizzata da due processi per scambiarsi un valore intero. In pratica, il valore che viene passato come parametro dal primo processo che invoca la funzione viene restituito al secondo processo e viceversa. Ovviamente il monitor deve essere riusabile nel senso che, una volta scambiati i valori relativi ai parametri della prima coppia di processi che hanno invocato la funzione, la stessa cosa deve avvenire nei confronti della seconda coppia e così via.

a) Realizzare il precedente monitor utilizzando la semantica *signal_and_urgent* relativa all'operazione `signal`.

b) Verificare se la soluzione resta invariata nel caso in cui si dovesse utilizzare la sematica *signal_and_continue*. Se la soluzione non cambia spiegare perché, altrimenti riscrivere una diversa soluzione.

Esercizio E2.3 In un negozio sportivo lavorano `C` commessi e un supervisore. Il negozio è frequentato da clienti normali e clienti che chiedono la sostituzione di articoli acquistati precedentemente. Il negozio ha una capacità massima di `CAP` clienti. Ogni cliente normale è servito da un commesso, mentre per servire un cliente che chiede una sostituzione è necessaria la presenza di un commesso e del supervisore. I clienti normali (e solo quelli normali), terminato l'acquisto, vanno alla cassa a pagare, mettendosi in coda nel caso in cui sia già occupata.

Implementare una soluzione, usando il costrutto monitor per modellare il negozio e i processi per modellare i clienti, e descrivere la sincronizzazione tra i processi. Nella soluzione massimizzare l'utilizzo delle risorse. Discutere se la soluzione proposta può presentare starvation, e in caso positivo per quali processi, e proporre modifiche e/o aggiunte per evitare starvation.

Esercizio E2.4 In un centro per il prelievo del sangue lavora un medico che ha a disposizione `L` lettini. Le persone che effettuano il prelievo si dividono in due categorie: donatori e pazienti. Ogni persona può iniziare il prelievo solo quando è disponibile il medico e c'è almeno un lettino vuoto, altrimenti aspetta. Dopo che il medico ha iniziato il prelievo, la persona aspetta che il medico finisca il prelievo. Terminato il prelievo, dopo essersi ripresa, la persona libera il lettino. Nella soluzione si tenga presente che i donatori hanno la precedenza sui pazienti.

Implementare una soluzione, usando il costrutto monitor per modellare il centro prelievi e i processi per modellare il medico e le persone, e descrivere la sincronizzazione tra i processi. Nella soluzione massimizzare l'utilizzo delle risorse. Discutere se la soluzione proposta può presentare starvation, e in caso positivo per quali processi, e proporre modifiche e/o aggiunte per evitare starvation.

Esercizio E2.5 Sia dato un sistema automatizzato per produrre ciambelle costituito da tre unità: un'impastatrice, un nastro trasportatore e un forno. L'impastatrice ha una capacità massima pari a `C_MAX` che rappresenta la quantità di ingredienti necessaria per produrre una ciambella. Il nastro e il forno possono trasportare o cuocere rispettivamente una sola ciambella alla volta.

Un processo specifico di tipo `gestore_ingredienti` regola l'immissione graduale degli ingredienti nella macchina impastatrice; al fine di facilitare l'impasto il processo introduce un'unità di ingredienti per volta fino a raggiungere `C_MAX`. La reimmissione è consentita solo al termine dello svuotamento del contenuto dell'impastatrice.

Quando la capacità massima dell'impastatrice è raggiunta, il sistema affida a un altro processo di tipo `gestore_nastro` lo svuotamento dell'impastatrice e il deposito dell'intera pasta sul nastro trasportatore. Lo stesso processo deve trasportare la pasta in forma di ciambella fino al forno, deve introdurla nel forno e, a cottura terminata, svuotare il forno.

Di tanto in tanto forno e nastro trasportatore devono essere puliti. Un processo specifico di tipo `pulizia` è incaricato di pulire le due risorse in un certo ordine (può pulire prima il forno e poi il nastro al termine della pulizia del forno o viceversa). Tuttavia, la pulizia di una risorsa (forno o nastro) può avvenire solo se la risorsa è libera, ossia se non è stata già acquisita dal processo di tipo `gestore_nastro`.

a) Individuare e commentare i possibili casi di deadlock che si possono verificare (si noti che forno e nastro sono risorse condivise tra il processo di tipo `pulizia` e il processo di tipo `gestore_nastro`).
b) Utilizzando il costrutto monitor, realizzare una politica di gestione del sistema per la produzione di ciambelle che eviti le situazioni di deadlock evidenziate. In particolare, tra le possibili soluzioni scegliere preferibilmente quella che ottimizza l'uso delle risorse forno e nastro da parte dei processi di tipo `pulizia` e `gestore_nastro`.

Esercizio E2.6 Un cinema prevede spettatori `Abbonati` e `Non_Abbonati`. L'ingresso e l'uscita del cinema sono resi disponibili da due corridoi stretti tali da non consentire il passaggio contemporaneo di spettatori in direzioni opposte: l'utilizzo di ogni corridoio è pertanto a senso unico alternato. Il cinema ha inoltre una capacità massima `CAP` equivalente al numero di posti a sedere disponibili nella sala. Inoltre, per motivi di sicurezza, anche per ogni corridoio è stabilita una capacità massima `CC` che esprime il numero massimo di utenti che possono transitare contemporaneamente per il corridoio.

Gli `Abbonati` accedono al cinema attraverso uno dei due corridoi; i `Non_Abbonati` entrano nel cinema percorrendo l'altro corridoio (ingressi separati per `Abbonati` e `Non_Abbonati`). Per ciò che riguarda l'uscita, invece, ogni corridoio è utilizzabile da entrambe le categorie di spettatori.

Utilizzando il costrutto monitor realizzare una politica di gestione del cinema che tenga conto dei seguenti vincoli:

- gli `Abbonati` devono essere favoriti sia nell'ingresso che nell'uscita dal cinema;
- per l'acquisizione di ciascun corridoio è necessario dare la priorità agli spettatori in uscita.

Esercizio E2.7 Alcuni processi *mittenti* inviano periodicamente messaggi di un tipo `mes` a due processi riceventi (`R1` e `R2`). I messaggi vengono inviati tramite una `mailbox` da realizzare per mezzo di un buffer circolare di `N` posizioni. Tutti i messaggi inviati devono essere ricevuti da entrambe i processi riceventi e ciascun processo deve ricevere tutti i messaggi in ordine FIFO (quindi una posizione della `mailbox`, una volta riempita con un messaggio, potrà essere resa libera solo dopo che tale messaggio sia stato ricevuto dai due processi `R1` e `R2`). Chiaramente, un generico messaggio potrà essere ricevuto prima da `R1` e poi da `R2` mentre un altro messaggio potrà essere ricevuto in ordine inverso a seconda dei rapporti di velocità tra i processi.

a) Utilizzando i monitor, realizzare la precedente `mailbox` con le tre funzioni `invio`, `ricezione1` e `ricezione2` invocate, rispettivamente, da ciascun mittente e da `R1` e `R2`.
b) Ripetere la soluzione con il vincolo che ogni messaggio debba sempre essere ricevuto prima da `R1` e poi da `R2`.

Esercizio E2.8 Un ponte consente il transito di autoveicoli tra le rive di un canale navigabile. Il canale è percorso da imbarcazioni di grosse dimensioni. Affinché sia possibile il passaggio delle imbarcazioni, il transito dei veicoli sul ponte deve necessariamente essere interrotto per permettere l'apertura del ponte stesso. In particolare, il ponte è percorribile da due tipi di veicoli: `camion` e `automobili`.

La larghezza del ponte è tale da impedire il passaggio contemporaneo di camion in direzioni opposte; in tutti gli altri casi il ponte può essere utilizzato a doppio senso di circolazione. Il ponte ha inoltre una capacità massima `C_max` che rappresenta il massimo peso sopportabile dal ponte. Il canale è transitabile da imbarcazioni in entrambi i versi di percorrenza contemporaneamente.

Utilizzando il costrutto monitor realizzare una politica di gestione del ponte che tenga conto dei seguenti vincoli:

- il transito di un'imbarcazione deve essere prioritario rispetto al transito di autoveicoli sul ponte: la presenza di un'imbarcazione in attesa di transitare deve necessariamente bloccare l'accesso di autoveicoli sul ponte, per permetterne l'apertura;
- l'accesso al ponte da parte di autoveicoli deve essere regolato in modo tale da favorire i camion rispetto alle automobili.

Esercizio E2.9 Si consideri un libretto di risparmio condiviso tra più intestatari abilitati sia in prelievo sia in deposito. Prelievo e deposito non possono avvenire simultaneamente. Utilizzando il costrutto monitor realizzare una politica di gestione del libretto considerando i seguenti vincoli:

- il saldo del libretto di risparmio non deve mai essere negativo;
- le operazioni di deposito hanno sempre la precedenza rispetto a quelle di prelievo;
- il prelievo di somme più piccole ha la precedenza sul prelievo di somme più consistenti al fine di mantenere il libretto maggiormente in attivo.

Esercizio E2.10 Si consideri un garage per auto e moto, organizzato su `N` livelli e con capacità massima pari a `N*MAX`. Ogni livello contiene posteggi numerati progressivamante da `1` a `MAX`. All'arrivo i conducenti richiedono il posteggio dei veicoli che devono essere parcheggiati nel primo spazio libero a partire dal primo livello. Successivamente i conducenti ottengono il ritiro del veicolo su presentazione del numero di posteggio occupato. Si consideri che il garage ha un solo punto di accesso in ingresso/uscita a senso unico alternato.

Utilizzando il costrutto monitor realizzare una politica di gestione del garage che tenga conto dei seguenti vincoli:

- le auto hanno la precedenza in ingresso sulle moto;
- i veicoli in uscita hanno sempre la precedenza su quelli in entrata.

Esercizio E2.11 Utilizzando i monitor, si vuole realizzare un meccanismo di comunicazione tra processi (`IPCM`) tramite il quale un numero imprecisato di processi mittenti invia messaggi a un solo processo ricevente. Ogni messaggio è costituito da stringa di caratteri. I vari messaggi hanno dimensioni variabili tra `1` e `80` incluso il carattere "`/0`" di fine stringa. Le funzioni che il monitor deve fornire per inviare e ricevere i messaggi sono le seguenti: `public void send (char mes[])` e `public void receive(char mes[])`. Supponiamo di avere a disposizione il monitor `buffer_circolare` che implementa una mailbox di 5 caratteri, con le funzioni `invio` e `ricezione` per inviare e ricevere rispettivamente un carattere (vedi la figura 6.3, dove `N=5` e il tipo `messaggio` coincide in questo caso con `char`).

a) Di seguito vengono presentate tre soluzioni, ciascuna delle quali presenta dei problemi. Indicare, per ciascuna di esse, perché non funziona.

- Prima soluzione errata

```
class IPCM {
   buffer_circolare mailbox;
   public void send (char mes[]){
      int i=0;
      mailbox.invio(mes[i]);
      while(mes[i]!='/0')
         {i++; mailbox.invio(mes[i]);}
   }
   public void receive(char mes[]){
      int i=0;
      mes[i]=mailbox.ricezione();
      while(mes[i]!='/0')
         {i++; mes[i]=mailbox.ricezione();}
   }
}
```

- Seconda soluzione errata

```
monitor IPCM {
   buffer_circolare mailbox;
   public void send (char mes[]){
      <stesso codice della prima soluzione> }
   public void receive(char mes[]){
      <stesso codice della prima soluzione> }
}
```

- Terza soluzione errata: non viene utilizzati il monitor `buffer_circolare` e viene realizzato il seguente monitor.

```
monitor IPCM {
   char buffer[5];
   int testa=0; coda=0; cont=0;
   condition non_pieno, non_vuoto;
   public void send (char mes[]){
      int i=0;
      do
         if (cont==5) wait(non_pieno);
         buffer[coda]=mes[i];
         coda = (coda+1)%N5;
         cont++;
         signal(non_vuoto);
      while(mes[i]!='/0');
   }
```

```
    public void receive(char mes[])
       int i=0;
       do
          if (cont==0) wait(non_pieno);
          mes[i]=buffer[testa];
          testa = (testa+1)%5;
          cont---;
          signal(non_pieno);
       while(mes[i]!='/0');
    }
}
```

b) Descrivere una possibile soluzione corretta.

Esercizio E2.12 In un aeroporto un nastro trasportatore collega la zona di check-in con un terminal di imbarco/sbarco. Il nastro trasportatore viene utilizzato per il trasporto di bagagli da e verso il terminal. In particolare, il nastro trasportatore serve i voli sia nazionali sia internazionali afferenti al terminal (in partenza o in arrivo) e ha un capacità limitata di peso C_MAX, oltre la quale non può consentire il caricamento di ulteriori bagagli.

Ogni bagaglio è quindi caratterizzato da un peso. I bagagli possono essere di due tipi:

- staff, ossia le valigie del personale delle compagnie aeree;
- ordinario, ossia i bagagli dei normali viaggiatori.

In generale i bagagli in partenza devono avere sempre la priorità su quelli in arrivo. Inoltre per i colli in partenza devono essere favoriti (nell'acquisizione del nastro) i bagagli staff rispetto a quelli ordinari. Diversamente, nell'acquisizione in arrivo non viene prevista alcuna politica di priorità: i bagagli acquisiranno il nastro in modo dipendente solo dall'ordine cronologico con cui si presentano all'imbocco del nastro.

Utilizzando il costrutto monitor realizzare una politica di gestione del nastro trasportatore.

E3 Primitive di comunicazione asincrone e/o sincrone

Esercizio E3.1 Supponendo di avere a disposizione le primitive di comunicazione asincrone (e asimmetriche), realizzare un processo server che offra le funzionalità di una malibox limitata (destinata cioè a contenere non più di 10 messaggi, tutti dello stesso tipo generico T). I clienti del processo `mailbox` sono costituiti da un numero imprecisato di processi mittenti (i produttori dei messaggi) e da cinque processi riceventi (i cinque processi consumatori dei messaggi noti come C_0, C_1, C_2, C_3 e C_4). Partendo dalla nota realizzazione della mailbox che utilizza le primitive asincrone (vedi il paragrafo 8.2.2), fornire la soluzione nel caso in cui si voglia privilegiare, tra i cinque consumatori, il processo di indice 0 rispetto a quello di indice 1, questo rispetto a quello di indice 2 e così via. Ciò significa che, quando un consumatore è disponibile a ricevere un messaggio, deve poter riceverlo subito, se almeno un messaggio è presente nella mailbox; in caso contrario il ricevente deve sospendersi e, quando un messaggio verrà inviato nella mailbox, lo stesso messaggio dovrà essere indirizzato al ricevente più importante, secondo la precedente specifica, nell'ipotesi in cui siano in attesa più riceventi.

Esercizio E3.2 Un sistema, organizzato secondo il modello a scambio di messaggi, fornisce come meccanismo di comunicazione tra processi le due primitive `send` e `receive` sincrone (e asimmetriche). In tale sistema N processi clienti P_0, P_1, ... P_{N-1} possono utilizzare il servizio offerto dal processo Ra e quello offerto dal processo Rb richiedendone (e successivamente rilasciando) il loro uso esclusivo a un server S. I processi possono richiedere (e rilasciare) l'uso di uno qualunque dei servizi offerti o da Ra o da Rb, oppure richiedere entrambi i servizi contemporaneamente.

Realizzare il codice del server S e indicare i protocolli utilizzati dai clienti per richiedere e rilasciare al server uno qualunque dei servizi o entrambi i servizi:

a) nell'ipotesi che il server non applichi nessuna regola di priorità;
b) nell'ipotesi che il server privilegi richieste singole rispetto alle richieste doppie.

Esercizio E3.3 In un sistema che utilizza il meccanismo del rendez-vous, sono presenti tre processi P1, P2 e P3 che interagiscono tra loro secondo quanto riportato nei seguenti spezzoni di programma:

```
process P1 {
   int b;
   ......
   P3.E1(b);
   ......
}

process P2 {
   int a;
   ......
   P3.E2(a);
   ......
}

process P3 {
   entry E1(int out y);
   entry E2(int in x);
   .........
   accept E1(int out y) {
      accept E2(int in x) {
         y=f(x);
      }
   }
.........
}
```

Utilizzando le primitive di comunicazione asincrone, riscrivere il codice dei tre processi in modo tale che, utilizzando le primitive `send` e `receive`, si comportino nella stessa identica maniera dei tre processi illustrati precedentemente.

Esercizio E3.4 Supponiamo di avere un monitor M che offre ai processi le seguenti due funzioni:

```
public void fun1(int i, int &ris) {
   if (!B1) wait(c1,i);
   S1;
   signal(c2);
   }

   void fun2(int &ris) {
      if (!B2)wait(c2);
      S2;
      signal(c1);
   }
```

Lo statement S1 (corpo della funzione fun1) termina assegnando un valore al parametro ris e garantisce la validità della postcondizione B2. Analogamente, lo statement S2 (corpo della funzione fun2) termina assegnando un valore al parametro ris e garantisce la validità della postcondizione B1.

La funzione fun1 può essere chiamata esclusivamente dal processo P1, il quale passa come primo parametro il valore 1, oppure dal processo P2, il quale passa il valore 2, oppure ancora dal processo P3, il quale passa il valore 3.

a) Utilizzando le primitive di comunicazioni sincronizzate, scrivere il codice di un processo server S che simula il comportamento del precedente monitor. Indicare anche cosa corrisponde in questa simulazione alla chiamata della funzione M.fun1 (i, r) da parte di uno dei tre processi Pi (i=1,2,3) e cosa corrisponde alla chiamata M.fun2(r) da parte di un qualunque altro processo.
b) Ripetere la soluzione utilizzando il meccanismo del rendez-vous.

Esercizio E3.5

a) Utilizzando le primitive di comunicazione asincrone, realizzare un processo server che offra a N processi clienti il servizio di coordinamento negli accessi a una risorsa condivisa secondo lo schema dei lettori/scrittori. Garantire l'assenza di starvation (per esempio imponendo gli stessi vincoli visti nell'esempio 5.6). Indicare quali sono le interazioni tra un cliente lettore, o scrittore, e il server.
b) Ripetere la soluzione utilizzando il meccanismo del rendez-vous.

Esercizio E3.6 In un sistema organizzato secondo il modello a scambio di messaggi si desidera realizzare un meccanismo di attese programmate che consenta a ognuno degli N processi applicativi (P_0, P_1, ..., P_{N-1}) di poter chiedere a un apposito processo servitore (il processo timer) di rimanere in attesa per un certo numero x di quanti di tempo consecutivi. Si vuole realizzare il meccanismo utilizzando le primitive di comunicazione asincrone e asimmetriche. Il processo timer offre servizio a due categorie di processi clienti. La prima categoria è costituita dagli N processi applicativi, a cui offre il servizio di attesa programmata (cioè, se t è il numero di quanti di tempo durante i quali un processo ha richiesto di rimanere sospeso, timer riattiva tale processo dopo che sono trascorse t interruzioni dell'orologio in tempo reale). La seconda categoria di clienti è costituita dal solo processo relativo al driver delle interruzioni dell'orologio in tempo reale. Per questo motivo il processo timer è dotato due porte: la prima (indicata come porta servizio) tramite la quale riceve le richieste di attesa da parte di ciascuno degli N processi applicativi; la seconda (indicata come porta tempo) utilizzata per ricevere messaggi da parte del driver dell'orologio in tempo reale, messaggi necessari a timer per registrare il passare del tempo scandito dalle interruzioni dell'orologio in tempo reale.

a) Scrivere il codice del processo servitore timer, indicando anche quali azioni deve eseguire ciascuno degli N processi applicativi per richiedere un'attesa di x quanti di tempo specificando il numero e il tipo di messaggi scambiati tra il generico processo applicativo e timer. Indicare, analogamente, quali messaggi sono scambiati tra il driver delle interruzioni in tempo reale e timer.
b) Indicare come cambia la soluzione nell'ipotesi di utilizzare primitive di comunicazione sincrone.
c) Fornire la soluzione utilizzando il meccanismo del rendez-vous.

E4 Rendez-vous

Esercizio E4.1 Utilizzando il meccanismo del rendez-vous, realizzare un processo server che alloca a un insieme di processi clienti due risorse equivalenti. Ciascun cliente può richiedere (e successivamente rilasciare) una qualunque delle due risorse o le due risorse contemporaneamente. Realizzare il server in modo tale che, all'atto di un rilascio, le richieste di due risorse siano privilegiate rispetto alle richieste singole. Specificare anche quali operazioni deve eseguire un cliente rispettivamente per chiedere e rilasciare sia un singola risorsa che le due risorse insieme.

Esercizio E4.2 Utilizzando come strumento di interazione tra processi il meccanismo del rendez-vous, si vuole realizzare un meccanismo di comunicazione tra processi che consenta a cinque processi mittenti P_0, P_1, ..., P_4 di inviare messaggi (di un tipo predefinito T) a cinque processi riceventi P_5, P_6, ..., P_9 tramite un buffer circolare di N posizioni. Il buffer circolare viene gestito da un processo server che offre apposite entry ai due tipi di processi clienti. Si chiede di implementare il processo server adottando la strategia di privilegiare le richieste dei processi di indice più basso. È richiesta anche la specifica delle operazioni che ciascun processo, mittente o ricevente, deve eseguire per richiedere al server il servizio necessario, rispettivamente, per inviare o ricevere un messaggio.

Esercizio E4.3 Utilizzando il meccanismo del rendez-vous, scrivere il codice di un processo server S che fornisce, a un insieme di processi clienti, i servizi relativi all'allocazione di tre risorse R0, R1 e R2. I processi clienti possono richiedere al server l'allocazione di una qualunque risorsa o di due risorse contemporaneamente fra quelle disponibili. Quando le risorse non sono più necessarie ogni cliente chiede al server di rilasciare la o le risorse precedentemente richieste.

a) Indicare quali sono le entry che il server deve mettere a disposizione dei clienti.
b) Scrivere il codice relativo al processo server nell'ipotesi che lo stesso dia priorità alle richieste singole rispetto alle richieste di due risorse.
c) Indicare quali operazioni deve eseguire un processo cliente per richiedere e rilasciare, rispettivamente, una o due risorse.

Esercizio E4.4 In un sistema che utilizza il meccanismo delle chiamate di dure remote, realizzare un processo `server` che alloca a `N` processi clienti (P_0, ..., P_{N-1}) `M` risorse equivalenti. Ogni processo, quando necessita di una risorsa de al `server` quale risorsa usare e quando non gli serve più rilascia al `server` risorsa allocata. Il `server` alloca le risorse ai processi richiedenti implemen una politica prioritaria in base alla quale privilegia i processi di indice più bass nei confronti di `P1`, `P1` nei confronti di `P2`, ecc.).

Indicare quali `entry` il `server` deve mettere a disposizione dei processi denti, scrivere il codice del processo `server` e indicare quali operazioni deve guire ogni cliente per richiedere e per rilasciare una risorsa.

E5 Libreria Pthread

Esercizio E5.1 Si consideri un castello di interesse storico. L'accesso al caste consentito a due tipi di visitatori: adulti o bambini. La visita al castello non può venire in modo libero, ma deve essere sempre guidata dal proprietario del caste (Essendo il proprietario unico, è implicito che in ogni istante ci può essere, al una e una sola visita in atto). La visita assume caratteristiche diverse a seconda l'insieme di visitatori da accompagnare sia costituito da adulti oppure da bambi Per questo motivo, il gruppo di persone che partecipa a ogni visita è un insie omogeneo di visitatori (cioè o tutti adulti, oppure tutti bambini). Per ottimizzare tilizzo del castello ogni visita può iniziare soltanto quando il numero `P` dei parte panti ha raggiunto un valore prestabilito `PMAX`. Quando il gruppo dei partecipanti completo, la visita può avere inizio attraverso l'attivazione di un opportuno proces proprietario che rappresenta la guida di ogni visita. Al termine della visita, il pro so proprietario provvede a far uscire i partecipanti e, successivamente, si pone in tesa di un nuovo gruppo di visitatori per la visita successiva. Alla fine di ogni vis il tipo dei partecipanti della visita successiva viene stabilito in base al numero e al po di visitatori in attesa; in particolare, detti `AS` e `BS`, rispettivamente, il numero adulti in attesa e il numero di bambini in attesa:

- se `AS>BS` allora la visita sarà per adulti;
- se `BS>AS` allora la visita sarà per bambini;
- se `BS=AS` verrà data la precedenza ai bambini: la visita sarà per bambini.

Definire una politica di gestione del castello e la si realizzi utilizzando la libr pthread.

Esercizio E5.2 Una società di noleggio di automobili offre ai propri clienti tre di automobili: piccole, medie, grandi. Ogni tipo di auto è disponibile in numero mitato (`Npiccole`, `Nmedie`, `Ngrandi`). I clienti del noleggio possono essere due tipi: convenzionati e non convenzionati. Ogni cliente richiede una macchina un particolare tipo (piccola, media o grande); il noleggio, sulla base della propria di sponibilità corrente:

- assegnerà a chi ha richiesto una macchina piccola, preferenzialmente una ma china piccola (se è disponibile), oppure una media (solo se vi è immediata disp nibilità di medie);
- assegnerà a chi ha richiesto una media, preferenzialmente una media (se è disponibile), oppure una grande (solo se vi è immediata disponibilità di grandi);
- a chi richiede una macchina grande, deve essere comunque assegnata una grande.

Il cliente, dopo aver ritirato la macchina presso la sede del noleggio, la usa per un intervallo di tempo arbitrario e, successivamente, procede alla restituzione della macchina. Le regole del noleggio prevedono che ritiro e restituzione delle auto avvengano sempre presso la stessa sede: non è prevista la restituzione in una sede diversa da quella del ritiro.

Per la gestione del noleggio, vale inoltre il seguente vincolo: nel ritiro delle auto, i clienti convenzionati devono essere prioritari rispetto ai non convenzionati.

Definire una politica di gestione del noleggio e implementarla utilizzando la libreria pthread.

Esercizio E5.3 Uno stadio ospita un torneo internazionale di basket al quale partecipano squadre italiane e straniere. Si prevede una continua affluenza di tifosi in ingresso e in uscita dallo stadio, dato il continuo susseguirsi di partite. I tifosi si suddividono in *adulti* e *adulti con bambini*.

Lo stadio è accessibile attraverso due corridoi (`Cancello Nord`, `Cancello Sud`) utilizzabili sia per l'ingresso sia per l'uscita. Per motivi di sicurezza si è deciso di gestire ogni corridoio in modo tale che i tifosi di squadre italiane non possano incrociarsi (nello stesso corridoio) con tifosi di squadre straniere: quindi ogni corridoio può funzionare a doppio senso di percorrenza soltanto nel caso in cui gli utenti che lo attraversano siano tutti tifosi dello stesso tipo (cioè tutti italiani, oppure tutti stranieri).

Lo stadio ha una capienza massima `MAX_C` (che esprime il numero massimo di tifosi consentito all'interno dello stadio) oltre la quale non è permesso l'accesso di ulteriori persone.

Realizzare una politica di gestione dello stadio, utilizzando la libreria pthread, considerando i seguenti vincoli:

- i tifosi *adulti con bambini* hanno la precedenza nell'accesso e nell'uscita;
- i gruppi di tifosi stranieri, per ragioni di ospitalità, sono favoriti nell'accesso e nell'uscita dallo stadio;
- tifosi che desiderano uscire dallo stadio hanno la precedenza sui tifosi che intendono entrare, nel tentativo di evitare situazioni di saturazione.

E6 Java

Esercizio E6.1 Il canale di Suez mette in collegamento il Mar Mediterraneo con il Mar Rosso e può essere utilizzato da imbarcazioni per il trasporto di passeggeri, da navi per il trasporto di merci e da petroliere. Il canale è sufficientemente largo da permettere il transito contemporaneo di qualunque tipo di nave nei due versi di percorrenza, ma si supponga che, per ragioni di sicurezza, il transito delle petroliere sia ammesso solo quando non ci sono imbarcazioni di tipo diverso all'interno del canale.

Realizzare una politica di gestione del canale, utilizzando il linguaggio Java, che tenga conto dei seguenti vincoli:

- nell'accesso al canale, le imbarcazioni per il trasporto dei passeggeri hanno sempre la precedenza sulle altre navi;
- le petroliere hanno la precedenza sulle navi per il trasporto delle merci.

Esercizio E6.2 Cinque filosofi si ritrovano tutti i giorni, per pranzare insieme, intorno a una tavola rotonda. Al centro della tavola c'è sempre un grande piatto di spaghetti. Ogni filosofo ha bisogno di due forchette per mangiare.

Purtroppo i filosofi sono poveri e dispongono di sole cinque forchette (tante quante sono i filosofi). Ogni forchetta è disposta tra due filosofi e ogni filosofo può utilizzare solo la forchetta alla propria destra e solo quella alla propria sinistra.

Ogni filosofo non può acquisire una forchetta già acquisita da un altro filosofo, esso adiacente, ma deve attendere che la forchetta sia rilasciata.

Realizzare una politica di gestione delle forchette, utilizzando il linguaggio Java, in modo da evitare una situazione di deadlock. La situazione da evitare è quindi quella in cui ogni filosofo ha una forchetta alla propria destra o alla propria sinistra.

Inoltre, realizzare una politica che prevenga la starvation dei filosofi.

Esercizio E6.3 Si consideri un piccolo giardino botanico che possa essere visitato da turisti. Il giardino è accessibile da due ingressi, situati uno a ovest e uno a est dello stesso parco. Da ciascun ingresso, i visitatori entrano uno alla volta.

All'interno del parco è presente un contatore che viene incrementato di uno ogni volta che una persona entra nel giardino.

a) Realizzare un meccanismo di sincronizzazione delle entrate al parco, che non prenda in considerazione le uscite, in modo che il contatore rilevi, in ogni istante, il numero esatto di persone all'interno del giardino.

b) Ipotizzare che il giardino botanico possa contenere al massimo 40 persone. Realizzare, quindi, una politica di sincronizzazione del parco che consideri sia le entrate sia le uscite dei visitatori. La politica di sincronizzazione deve permettere alle persone di entrare, da uno dei due ingressi, solo quando il parco non è pieno (ci sono cioè meno di 40 persone all'interno) e di uscire solo se nel parco c'è almeno una persona.
Implementare la politica di sincronizzazione privilegiando le entrate al parco piuttosto che le uscite. Inoltre, ipotizzare che i visitatori possano uscire soltanto dal cancello situato a est.

b) Realizzare una politica di sincronizzazione delle entrate e delle uscite del parco, in modo da garantire un flusso equo delle persone in ingresso e in uscita.
Implementare le persone che desiderano entrare dai due ingressi e uscire dall'ingresso est, tramite thread Java.

Esercizio E6.4 Si consideri un circolo di golf. Il circolo può noleggiare solamente un numero limitato di palline da golf. In caso di necessità, i giocatori possono noleggiare le palline da golf durante il gioco e al termine del gioco devono restituirle. In particolare, i giocatori esperti noleggiano una o al massimo due palline da golf (tipicamente non perdono le palline durante il gioco). I giocatori meno esperti, invece, noleggiano più palline da golf. In ogni caso, devono ricomprare le palline che eventualmente perdono in modo da restituire il numero esatto di palline originariamente noleggiato.

Realizzare, utilizzando il linguaggio Java, una politica di allocazione delle palline da golf.

Suggerimenti: per simulare il golf club, si crei un thread per ogni giocatore che arriva al club. Ogni giocatore è caratterizzato da un nome (per esempio una lettera assegnata consecutivamente in ordine alfabetico) e dal numero di palline che il giocatore richiede. A ogni giocatore quando entra nel golf club viene assegnato un nome, prende le palline a noleggio, gioca; poi al termine del gioco riconsegna il numero di palline noleggiate.

Realizzare un gestore per l'allocazione delle palline capace di controllare il noleggio delle palline. Inizializzare il numero massimo di palline da noleggiare alla costante `NMAX`.

Maurelio Boari

Programmazione concorrente e distribuita

Questo libro è il frutto di una lunga esperienza di ricerca e insegnamento svolta dagli Autori nel campo della Programmazione concorrente e dei Sistemi operativi. Per gli argomenti trattati, il libro è particolarmente adatto come testo per un corso approfondito di Sistemi operativi, così come è normalmente previsto nell'ambito dei corsi sia in Ingegneria informatica sia in Scienze dell'informazione, oltre che come testo per un corso di Programmazione concorrente e distribuita. Il presente volume si pone come logica continuazione di *Sistemi operativi*, pubblicato dagli stessi Autori e concepito per un corso base.

Vengono trattate le principali problematiche introdotte dalla concorrenza sia in sistemi non distribuiti (sistemi monoelaboratore o multielaboratore a memoria condivisa) sia in sistemi sviluppati su architetture distribuite costituite da macchine interconnesse tramite una rete di comunicazione.

Come esemplificazione dei concetti introdotti, viene fatto riferimento a specifici linguaggi (per esempio Java) o a librerie standard (come la libreria Pthread).

Il libro è corredato da numerosi esercizi di riepilogo relativi agli strumenti di sincronizzazione presentati e alla realizzazione di politiche di gestione delle risorse, sia in ambiente a memoria comune sia in ambiente distribuito.

All'indirizzo web www.ateneonline.it/ancilotti sono disponibili, per gli studenti, le soluzioni commentate degli esercizi di riepilogo e, per i docenti, i lucidi in formato PowerPoint.

Paolo Ancilotti è professore ordinario di Sistemi operativi presso la Scuola Superiore di Studi Universitari Sant'Anna di Pisa, di cui è direttore.
Maurelio Boari è professore ordinario di Calcolatori elettronici presso l'Università degli Studi di Bologna.

€ 30,00 (i.i.)

ISBN 88-386-6358-0

▶ www.mcgraw-hill.it

▶ www.ateneonline.it