Esercizi Monitor

Esercizio 1

Un **conto corrente** bancario è condiviso da diversi utenti (processi). Ciascuno può depositare o prelevare soldi. Il saldo non deve mai diventare negativo. I prelievi di somme più piccole hanno la precedenza sui prelievi di somme più alte.

a) Scrivere un **monitor** che risolva questo problema; la semantica della **signal** sia `signal_and_urgent_wait`.

b) Modificare il monitor precedente imponendo che i prelievi debbano essere eseguiti secondo la politica FIFO. Si supponga che le code associate alle variabili `condition` siano gestite con politica FIFO.

Esercizio 1a - Soluzione

Si suppone che esista un tipo **priorityqueue** che realizzi una coda a priorità fissa secondo un parametro intero p (p minimo, priorità massima);

su questo tipo di coda sono definite le procedure:

- *insert(coda,valore)*
- *remove(coda)*

e la funzione

- *first(coda)* (restituisce il valore del primo elemento della coda senza rimuoverlo)

Esercizio 1a - Soluzione

```
monitor conto_corrente1  {
 int saldo = valore iniziale
 int n_sosp = 0
 priorityqueue coda // inizialmente vuota//
 condition sospesi;
 public void preleva (int cifra) {
  if (cifra > saldo)  {
        n_sosp ++;
       insert(coda,cifra);
       wait(sospesi, cifra);
       n_sosp -- -- ;
   };
   saldo = saldo – cifra;
   if (n_sosp > 0).AND.(first(coda) ≤ saldo) {
       remove(coda);
       signal(sospesi);
   };
 }
```

```
  public void deposita
(int cifra) {
    saldo = saldo + cifra;
    if (n_sosp > 0).AND.
(first(coda) ≤ saldo) {
       remove(coda);
       signal(sospesi);
     };
 }
}
```

Esercizio 1a – Un’altra soluzione

```
monitor conto_corrente1bis  {
 int saldo = valore iniziale
 condition sospesi;

 public void preleva (int cifra)
{
 while (cifra > saldo)
wait(sospesi, cifra);
saldo = saldo – cifra;
signal(sospesi);
}
```

```
   public void
deposita (int cifra) {
     saldo = saldo +
cifra;
     signal(sospesi);
     }
}
```

Esercizio 1b - Soluzione

Si suppone che esista un tipo **fifoqueue** che realizzi una coda a priorità fissa secondo un parametro intero p (p minimo, priorità massima);

su questo tipo di coda sono definite le procedure:

- *insert(coda,valore)*
- *remove(coda)*

e la funzione

- *first(coda)* (restituisce il valore del primo elemento della coda senza rimuoverlo)

Esercizio 1b - Soluzione

```
monitor conto_corrente2  {
 int saldo = valore iniziale
 int n_sosp = 0
 fifoqueue coda // inizialmente vuota//
 condition sospesi;
 public void preleva (int cifra) {
  if (cifra > saldo) .OR. (n_sosp > 0)  {
        n_sosp ++;
       insert(coda,cifra);
       wait(sospesi,);
       n_sosp -- -- ;
   };
   saldo = saldo – cifra;
   if (n_sosp > 0).AND.(first(coda) ≤ saldo)
{
       remove(coda);
       signal(sospesi);
   };
 }
```

```
  public void deposita
(int cifra) {
    saldo = saldo + cifra;
   if (n_sosp > 0).AND.
(first(coda) ≤ saldo) {
       remove(coda);
       signal(sospesi);
     };
 }
}
```

Esercizio 2

Si consideri il codice seguente che modella il comportamento di un sistema costituito da **un pasticcere e da n clienti**, utilizza per la sincronizzazione il costrutto semaforico.

a) Si modelli questo sistema con le **stesse caratteristiche di sincronizzazione** usando il costrutto monitor con semantica `signal_and_urgent_wait`.

b) Si provi poi la validità del seguente invariante di monitor:

$$0 \leq biscotti \leq m$$

Esercizio 2

```
semaphore mutex = 1
semaphore pieno = 0
semaphore vuoto = 0
int biscotti =  0

process Client_i{
:
 P(mutex)
 if (biscotti = = 0) {
 V(vuoto);
 P(pieno);
 };
 biscotti--;
 V(mutex);
 <mangia biscotto>;
.
```

```
process Pasticcere {
    :
 while(true)  {
 P(vuoto);
 <prepara biscotti e riempi il vassoio>;
 biscotti = m;
 V(pieno);
 }
}
```

Esercizio 3

Si consideri il monitor seguente che rappresenta una soluzione al **problema dei cinque filosofi**.

La semantica della **signal** è la `signal_and_urgent_wait`.

Sia `E` la variabile di stato che conta quanti filosofi stanno mangiando, cioè hanno eseguito la procedura `takeForks` e non hanno ancora invocato la procedura `releaseForks`.

Si dimostri che il monitor preserva il seguente invariante:

$$\sum_{i=0}^{4} \texttt{fork[i]} + 2 \times \texttt{E} = 10.$$

Esercizio 3

```
monitor ForkMonitor {
 int array[0..4] fork //initially [2, . . . , 2]//
 condition array[0..4] OKtoEat
 public void takeForks(integer i) {
   if (fork[i] < 2) wait(OKtoEat[i]);
   fork[(i+1)%5] --;
   fork[(i−1)%5] --;
  }
 public void releaseForks(integer i) {
  fork[i+1] ++;
  fork[i−1] ++;
  if (fork[(i+1)%5] = = 2) signal(OKtoEat[(i+1)%5]);
  if (fork[(i−1)%5] = = 2) signal(OKtoEat[(i−1)%5]);
 }
```

Esercizio 4

Si consideri il sistema seguente che rappresenta una soluzione al problema delle molecole dell'acqua usando il costrutto monitor con semantica `signal_and_urgent_wait`.

Siano `nH` e `nO` le variabili di stato che contano rispettivamente il numero di caratteri "H" e "O" stampati. Si dimostri che il monitor preserva i seguenti invarianti:

$$count = nH - 2*nO$$

$$0 \leq count \leq 2$$

Esercizio 4

```
monitor H2O {
int count = 0;
condition Oxy;
condition Hydro;

 public void StampaH {
  if (count == 2) wait(Hydro);
  <stampa”H”>;
  count++;
  if (count ==2) signal(Oxy);
 }

 public void StampaO {
  if (count < 2) wait(Oxy);
  <stampa”O”>;
  count= 0;
  signal(Hydro);
 }
}
```

```
H2O acqua
process Idrogeno {
 while (True) {
 <costruisci atomo idrogeno>;
 Acqua.StampaH
 }
}
```

```
process Ossigeno {
 while (True) {
  <costruisci atomo ossigeno>;
  Acqua.StampaO
  }
}
```