Copyright © 2007 – The McGraw-Hill Companies srl

Esercizio E1.1

Impostazione

Il problema da risolvere prevede la realizzazione di un gestore di un pool di risorse equivalenti. L'allocazioone delle risorse deve avvenire in base a un criterio di priorità. È quindi opportuno fare riferimento ad uno schema di soluzione che prevede l'uso di semafori privati (vedi paragrafo 5.7). Inoltre, poiché le funzioni di richiesta prevedono la restituzione di valori tramite il parametro `x` o i parametri `x` e `y` rispettivamente, fra i due schemi di uso dei semafori privati è sicuramente più adatto il secondo schema che utilizza la tecnica del *passaggio del testimone* anche se, come è stato mostrato, questo schema può risultare più complesso nel caso in cui, all'atto di un rilascio si debbano svegliare più processi. Questa eventualità può verificarsi soltanto quando vengono rilasciate due risorse. Ad esempio, se prima del rilascio non sono disponibili risorse e il processo bloccato a più alta priorità richiede una sola risorsa ed esiste un secondo processo bloccato a priorità immediatamente inferiore che richiede, a sua volta, una sola risorsa, allora con le due risorse rilasciate possono essere svegliati entrambi i processi, uno dopo l'altro. Analogamente ciò accade se è disponibile una sola risorsa mentre il processo bloccato a più alta priorità ne richiede due e quello a priorità immediatamente inferiore ne richiede una sola. Poiché il secondo schema dei semafori privati utilizza la tecnica del *passaggio del testimone*, all'atto di un rilascio può essere svegliato un solo processo. Ciò significa, come visto nel paragrafo 5.7, che è necessario complicare le funzioni `rilascio` in modo tale che sia il solo processo risvegliato, terminando il `rilascio`, a verificare se è possibile svegliare un secondo processo e, in caso positivo, a risvegliarlo.

```
class tipo_gestore{
   semaphore  mutex=1; /*semaforo di mutua esclusione*/
   semaphore priv[N]={0,0,…0}; /*semafori privati dei processi clienti*/
   int sospesi=0; /*tiene traccia del numero dei processi sospesi*/
   int tempo[N]={0,0,…0}; /*se Pi è attivo allora tempo[i]==0 mentre se
                  Pi è sospeso tempo[i] ha un valore != 0 che indica il tempo complessivo relativo
                  all'impegno delle risorse richieste*/
   int quante[N]={0,0,…0}; /* se Pi è attivo allora quante[i]==0 mentre
                  se Pi è sospeso quante[i] ha un valore != 0 che indica il numero
                  di risorse richieste*/
   int disponibili=10; /*tiene traccia del numero di risorse disponibili*/
   boolean libera[10]={true, true,…, true}; /*libera[j] assume il
                        valore true se la risorsa j è libera; false diversamente*/

/*alcune relazioni di consistenza della struttura dati del gestore sono le seguenti:
   {il valore dalla variabile intera sospesi corrisponde al numero di elementi dei vettori tempo e quante
   che hanno valori diversi da zero. Tale valore coincide col numero di processi clienti sospesi}
   {il valore della variabile intera disponibili coincide con il numero di elementi del vettore libera che
   hanno valore true. Tale intero coincide col numero di risorse disponibili}
   È facile verificare che tali relazioni sono invarianti del tipo astratto tipo_gestore. Infatti sono
   ovviamente vere all'inizio ed è facile verificare che, se sono vere prima dell'inizio di una funzione, allora
   tornano vere alla fine della sua esecuzione.*/

   int minimo() {
      /*funzione locale (non pubblica) che, se chiamata quando ci sono processi sospesi,
         trova il (o uno dei) valori minimi del vettore tempo  e ne restituisce l'indice*/
     int min=0; int indice_min;
     for(int i=0; i<N; i++)
       if(tempo[i]!=0)   /*cioè se il processo Pi è sospeso*/
          if(min==0||tempo[i]<min) /*cioè se Pi è il primo processo sospeso
                        che trovo oppure se il tempo complessivo richiesto da Pi
```

Copyright © 2007 – The McGraw-Hill Companies srl

```
                        è minore del valore minimo finora trovato*/
          {min=tempo[i]; indice_min=i;} /*aggiorno il nuovo valore
                        minimo e registro in indice_min l'indice del vettore
                        tempo in cui tale valore si trova*/
  return indice_min;
}

boolean da_svegliare(int &chi){
   /*funzione booleana locale (non pubblica) che, se ci sono processi sospesi,
      verifica se quello a più alta priorità può essere svegliato. In caso positivo restituisce il valore true e,
      tramite il parametro chi, l'indice di tale processo
      In caso contrario restitiuisce il valore false*/
    if(sospesi>0){
       int imin;  int ris;
       imin=minimo(); /*imin è l'indice del processo sospeso a più alta priorità*/
       ris=quante[imin]; /*ris è il numero di risorse richieste dal processo*/
       if(disponibili>=ris) /*il processo può essere svegliato/*
          {chi=imin; return true;}
    return false /*altrimenti no*/
}

public void richiesta1(int &x , int t,  int c){
    int tcomp=t;  /*tempo complessivo richiesto dal processo richiedente*/
    int ind_min; /*indice dell'eventuale processo sospeso a priorità più alta */
    int tmin; /*tempo complessivo richiesto da tale processo*/
    int proc;
    P(mutex); /*si inizia entrando in mutua esclusione*/
    if(sospesi>0) {ind_min=minimo();  tmin=tempo[ind_min];}
    if(disponibili==0||sospesi>0&&tmin<tcomp){
       /*se non ci sono risorse disponibili, oppure se ci sono processi sospesi e quello
          a più alta priorità richiede un tempo complessivo inferiore a quello del processo richiedente,
          allora quest'ultimo va bloccato/*
        tempo[c]=tcomp;
        quante[c]=1;
        sospesi++;
        V(mutex);
        P(priv[c]);
        sospesi--;
        quante[c]=0;
        tempo[c]=0;
    }
       /*si arriva a questo punto se per il processo richiedente non sono verificate le
          condizioni per la sua sospensione oppure, se dopo essere stato sospeso è stato riattivato con la
          tecnica del passaggioo del testimone/*
    int i=0; /*tramite l'indice i si scandisce il vettore libera per trovare la
                          prima risorsa disponibile*/
       /*la precondizione dello statement successivo è che almeno una risorsa sia
          disponibile {disponibili>0}  per garantire che il ciclo while abbia termine. Ciò è
          ovviamente vero se la condizione del precedente statement if è stata trovata falsa. Se viceversa il
          processo è stato prima sospeso e poi riattivato, dovremo assicurarci che all'interno delle funzioni
          rilascio, quando un processo viene riattivato, la precedente condizione sia vera/*
    while(!libera[i]) i++;
       /*trovata la prima risorsa libera, quella di indice i, si registra che non è più
```

McGraw-Hill
Tutti i diritti riservati

Copyright © 2007 – The McGraw-Hill Companies srl

```
        libera, diminuendo anche la variabile disponibili, e se ne restituisce l'indice al processo
        richiedente tramite il parametro x/*
    libera[i]=false;
    disponibili --;
    x=i;
      /*prima di terminare è necessario verificare se esiste un possibile processo
        sospeso che, a questo punto, possa essere riattivato. Ciò, come visto, può accadere se il processo
        in esecuzione è stato riattivato da un rilascio2 /*
    if(da_svegliare(proc)) V(priv[proc]);
    else V(mutex);
      /*se un tale processo esiste, lo si riattiva senza rilasciare la mutua esclusione
        (tecnica del passaggio del testimone) altrimenti si termina rilasciando la mutua esclusione senza
        svegliare nessuno/*
}

public void richiesta2(int &x , int &y, int t,  int c){
  /*la funzione richiesta2 ha la stessa struttura della richiesta1 con la sola
    differenza che le risorse richieste sono ora due e quindi il tempo complessivo richesto è ore 2*t */
    int tcomp=2*t;
    int ind_min;
    int tmin;
    int proc;
    P(mutex);
    if(sospesi>0) {ind_min=minimo();  tmin=tempo[ind_min];}
    if(disponibili<2||sospesi>0&&tmin<tcomp){
       tempo[c]=tcomp;
       quante[c]=2;
       sospesi++;
       V(mutex);
       P(priv[c]);
       sospesi--;
       quante[c]=0;
       tempo[c]=0;
    }
    int i=0;
    while(!libera[i]) i++;
    libera[i]=false;
    x=i;
    i++;
    while(!libera[i]) i++;
    libera[i]=false;
    y=i;
    disponibili=disponibili-2;
    if(da_svegliare(proc)) V(priv[proc]);
    else V(mutex);
}

public void rilascio1(int x){
    int proc;
    P(mutex);
      /* una volta entrati in mutua esclusione, viene indicato che la risorsa di indice x
          è di nuovo libera /*
    libera[x]=true;
    disponibili++;
      /*prima di terminare è necessario verificare se, nel caso in cui esistano
```

McGraw-Hill
Tutti i diritti riservati

Copyright © 2007 – The McGraw-Hill Companies srl

processi sospesi, sia possibile riattivare, utilizzando la risorsa rilasciata, quello a più alta priorità /*

```
    if(da_svegliare(proc)) V(priv[proc]);
    else V(mutex);
```

/*se tale processo esiste, lo si riattiva senza rilasciare la mutua esclusione (tecnica *del passaggio del testimone*) altrimenti si termina rilasciando la mutua esclusione senza svegliare nessuno. Da notare che se il processo di indice `proc` viene riattivato è perché la funzione `da_svegliare` restituisce il valore `true`. E ciò implica, per come è programmata tale funzione, che esistono tante risorse libere quante ne sono richieste dal processo da riattivare. Ciò garantisce la verità delle proceondizioni degli statement `while` presenti nelle funzioni `richiesta`/*

```
 }

 public void rilascio2(int x, int y){
```

/*la funzione `rilascio2` ha la stessa struttura della `rilascio1` con la sola differenza che le risorse rilasciate sono ora due */

```
    int proc;
    P(mutex);
    libera[x]=true;
    libera[y]=true;
    disponibili=disponibili+2;
    if(da_svegliare(proc)) V(priv[proc]);
    else V(mutex);
 }

}
```

McGraw-Hill
Tutti i diritti riservati