Esercizi
Sistemi Operativi

1 Il produttore-consumatore

Si consideri l'esempio del produttore-consumatore con i semafori:

```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                 /* TRUE is the constant 1 */
          item = produce_item();    /* generate something to put in buffer */
          down(&empty);             /* decrement empty count */
          down(&mutex);             /* enter critical region */
          insert_item(item);        /* put new item in buffer */
          up(&mutex);               /* leave critical region */
          up(&full);                /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                 /* infinite loop */
          down(&full);              /* decrement full count */
          down(&mutex);             /* enter critical region */
          item = remove_item();     /* take item from buffer */
          up(&mutex);               /* leave critical region */
          up(&empty);               /* increment count of empty slots */
          consume_item(item);       /* do something with the item */
     }
}
```

Supponiamo che nella funzione produttore le istruzioni `up(&pieni)` e `up(&mutex)` vengano invertite. La soluzione continua ad essere corretta o si possono generare situazioni di stallo? In quest'ultimo caso, spiegare perchè.

Rispondere alla stessa domanda nel caso le istruzioni ad essere invertite siano `down(&vuoti)` e `down(&mutex)`, sempre nella funzione produttore.

2 Richiesta Risorse

Si considerino tre processi P1, P2 e P3. Il processo P1 usa le risorse Ra ed Rb, P2 usa Ra ed Rc, P3 usa Rb ed Rc. Si supponga che il frammento di programma che P1 esegue per richiedere l'accesso alle risorse sia il seguente:

```
request(Ra)
request(Rb)
...... utilizzo delle risorse
release(Rb)
release(Ra)
```

mentre P2 esegue il programma:

```
request(Rc)
request(Ra)
...... utilizzo delle risorse
release(Ra)
release(Rc)
```

Scrivere l'analogo frammento di programma per P3, facendo in modo che non sia possibile per i 3 processi andare in deadlock.

3 Ordinamento Temporale

Si consideri il seguente programma

```
<???>

void p1(void)
{
   .... sezione 1 ....
   <???>
   .... sezione 2 ....
}

void p2(void)
{
   ..... sezione 3 ....
  <???>
   .... sezione 4 ...
}
```

Le funzioni p1 e p2 corrispondono a due processi distinti. Si vuole fare in modo che p2 possa passare ad eseguire la sezione 4 solo DOPO che p1 ha terminato l'esecuzione della sezione 1 (ad esempio perchè la sezione 4 usa dei file che vengono scritti dalla sezione 1). Come si possono riempire gli spazi <???> con delle istruzioni che garantiscono quest'ordine nell'esecuzione? (usare i semafori)

4 Semafori e Mutex

Sia dato il seguente programma, dove p1 e p2 sono due funzioni in esecuzione su processi diversi:

```
semaforo s=1;

void p1(void)
{
   down(&s);
   ...sezione 1...
   down(&s);
   ...sezione 2...
}
```

```
void p2(void)
{
   up(&s);
   ..sezione 3....
}
```

e una sua variante che utilizza i mutex

```
mutex s;

void p1(void)
{
   wait(&s);
   ...sezione 1...
   wait(&s);
   ...sezione 2...
}

void p2(void)
{
   signal(&s);
   ..sezione 3...
}
```

I due programi sono equivalenti? (cioè, si comportano allo stesso modo in tutte le condizioni?)

Considera ora il programma

```
semaforo s=1;

void p1(void)
{
   down(&s);
   ...sezione 1...
   up(&s);
}

void p2(void)
{
   down(&s);
   ..sezione 2...
   up(&s)
}
```

e la sua variante con i mutex

```
mutex s;

void p1(void)
{
   wait(&s);
```

```
   ...sezione 1...
   signal(&s);
}

void p2(void)
{
   wait(&s);
   ..sezione 2...
   signal(&s)
}
```

Questi si comportano allo stesso modo?

5 Attraversamento Pedonale

L'incrocio tra un strada di grande traffico e una strada pedonale è regolato da un vigile che, senza preoccuparsi di evitare l'attesa indefinita, applica la seguente politica:

- i pedoni sono autorizzati ad attraversare solo quando non vi sono veicoli che stiano attraversando l'incrocio o che abbiano già ottenuto il consenso per attraversare;
- i veicoli sono autorizzati ad attraversare solo quando non vi sono pedoni che stiano attraversando l'incrocio o che abbiano già ottenuto il consenso per attraversare.

I veicoli e i pedoni sono processi, numeri da 0 a MaxVeicoli-1 / MaxPedoni-1 rispettivamente. Quando un veicolo vuole attraversare chiama la procedura `RichiestaAttraversamentoVeicolo` indicando come parametro il numero del proprio veicolo. Alla fine dell'attraversamento chiama la procedura `FineAttraversamentoVeicolo`. La stessa cosa succede per i pedoni, con le procedure `RichiestaAttraversamentoPedone` e `FineAttraversamentoPedone`.

Riempire gli spazi indicati con <???> in modo da implementare la politica appena descritta.

```
#define MaxPedoni 50
#define MaxVeicoli 30
#define true 1
#define false 0

semaforo veicoli[MaxVeicoli];  /* tutti i semafori in questi array */
semaforo pedoni[MaxPedoni];    /* sono inizializzati al valore 0 */

semaforo mutex= <????> ;

bool VeicoliInAttesa[MaxVeicoli];   /* array automaticamente inizializzati */
bool PedoniInAttesa[MaxPedoni];     /* con il valore false */

int VeicoliCheAttraversano=0;
int PedoniCheAttraversano=0;
```

```
void RichiestaAttraversamentoVeicolo(int i)
{
down(&mutex);
if (PedoniCheAttraversano==0) {
   VeicoliCheAttraversano=VeicoliCheAttraversano+1;
   up(&veicoli[i]);
} else
   VeicoliInAttesa[i]=true;
up(&mutex);
<???????????>
}

void FineAttraversamentoVeicolo(int i)
{
int j;

down(&mutex);
VeicoliCheAttraversano=VeicoliCheAttraversano-1;
if (VeicoliCheAttraversano==0) {
   j=0;
   while ( <?????> ) {
      if (PedoniInAttesa[j]) {
          PedoniInAttesa[j]=false;
          up(&pedoni[j]);
      }
      j=j+1;
   }
up(&mutex);
}

void RichiestaAttraversamentoPedone(int i)
{
down(&mutex);
if (VeicoliCheAttraversano==0) {
   PedoniCheAttraversano=PedoniCheAttraversano+1;
   <????????????>
} else
   PedoniInAttesa[i]=true;
up(&mutex);
down(&pedoni[i]);
}


void FineAttraversamentoPedone(int i)
{
int j;

down(&mutex);
PedoniCheAttraversano=PedoniCheAttraversano-1;
```

```
if (PedoniCheAttraversano==0) {
   j=0;
   while ( <?????> ) {
      if (VeicoliInAttesa[j]) {
          VeicoliInAttesa[j]=false;
          up(&veicoli[j]);
      }
      j=j+1;
   }
up(&mutex);
}
```

6 Soluzioni

6.1 Produttore-Consumatore

Invertire le due istruzioni `up` non ha nessuna conseguenza sulla validità del programma. Al contrario, invertire le due `down`, può portare a un deadlock. Infatti, supponiamo che il buffer sia completamente pieno e che nessun dei due processi sia nella propria sezione critica. Supponiamo che producer inizi una iterazione del ciclo while, eseguendo l'istruzione `item=produce_item()` e successivamente, in sequenza, `down(&mutex)` e `down(&empty)` (visto che li abbiamo invertiti). Siccome nessuno dei due processi è nella propria zona critica, la prima down ha successo e il semaforo mutex passa al valore 0. Tuttavia, poichè il buffer è pieno (ovvero `empty`=0), la seconda `down` sospende il processo producer. D'altro canto, quando il processo `consumer` tenta di entrare nella zona critica, esegue in sequenza `down(&full)` e `down(&mutex)`. Il primo ha successo perchè il buffer è pieno, quindi `full`=N maggiore di zero, ma il secondo down sospende il processo producer perchè `mutex` è stato posto a zero dal producer. Quindi entrambi i processi si sospendono e si ha una situazione di stallo.

Si può riportare la spiegazione di cui sopra con una tabella degli accessi alle zone critiche:

evento	**full**	**empty**	**mutex**
	N	0	1
producer entra	N	0_{producer}	0
consumer entra	$N-1$	0_{producer}	0_{consumer}

6.2 Richiesta Risorse

Se si numerano le risorse Rc, Ra ed Rb con i valori 1, 2 e 3 rispettivamente, si nota che le risorse con numero inferiore vengono richieste prima di quelle con numero superiore. In P1, ad esempio, la risorsa Ra (valore 2) viene richiesta prima di Rb (valore 3). Basta continuare a rispettare questa convenziona anche per P3, e quindi richiedere Rc (valore 1) prima di Rb (valore 3):

```
request(Rc)
request(Rb)
...... utilizzo delle risorse
release(Rb)
```

```
release(Rc)
```

6.3 Ordinamento Temporale

```
semaforo s=0;

void p1(void)
{
   .... sezione 1 ....
   up(&s);
   .... sezione 2 ....
}

void p2(void)
{
   ..... sezione 3 ....
  down(&s);
   .... sezione 4 ...
}
```

6.4 Semafori e Mutex

I primi due programmi presentati non sono equivalenti. Mentre nella soluzione con i semafori la sezione viene sicuramente eseguita, nella versione con i mutex questo non è assicurato. Infatti, inizialmente il mutex assume il valore vero (o 1). Se l'istruzione `signal(&s)` viene eseguita prima delle due `down(&s)`, essa non avrà alcun effetto in quanto i mutex possono assumere solo valore falso (0) e vero (1). Ne segue che il processo p1 si blocca, successivamente, alla seconda `down(&s)`.

I due sucessivi programmi presentati sono invece equivalenti. Infatti `up` e `down` sono disposti in maniera tale che s non ha mai valori superiori a 1. Sotto questa condizione, mutex e semafori sono equivalenti.

6.5 Attraversamento Pedonale

```
#define MaxPedoni 50
#define MaxVeicoli 30
#define true 1
#define false 0

semaforo veicoli[MaxVeicoli];  /* tutti i semafori in questi array */
semaforo pedoni[MaxPedoni];    /* sono inizializzati al valore 0 */

semaforo mutex= 1 ;

bool VeicoliInAttesa[MaxVeicoli];   /* array automaticamente inizializzati */
bool PedoniInAttesa[MaxPedoni];     /* con il valore false */

int VeicoliCheAttraversano=0;
```

```
int PedoniCheAttraversano=0;

void RichiestaAttraversamentoVeicolo(int i)
{
down(&mutex);
if (PedoniCheAttraversano==0) {
   VeicoliCheAttraversano=VeicoliCheAttraversano+1;
   up(&veicoli[i]);
} else
   VeicoliInAttesa[i]=true;
up(&mutex);
down(&veicoli[i]);
}

void FineAttraversamentoVeicolo(int i)
{
int j;

down(&mutex);
VeicoliCheAttraversano=VeicoliCheAttraversano-1;
if (VeicoliCheAttraversano==0) {
   j=0;
   while ( j<MaxPedoni ) {
      if (PedoniInAttesa[j]) {
          PedoniInAttesa[j]=false;
          up(&pedoni[j]);
      }
      j=j+1;
   }
up(&mutex);
}

void RichiestaAttraversamentoPedone(int i)
{
down(&mutex);
if (VeicoliCheAttraversano==0) {
   PedoniCheAttraversano=PedoniCheAttraversano+1;
   up(&pedoni[i]);
} else
   PedoniInAttesa[i]=true;
up(&mutex);
down(&pedoni[i]);
}


void FineAttraversamentoPedone(int i)
{
int j;

down(&mutex);
```

```
PedoniCheAttraversano=PedoniCheAttraversano-1;
if (PedoniCheAttraversano==0) {
   j=0;
   while ( j<MaxVeicoli ) {
      if (VeicoliInAttesa[j]) {
          VeicoliInAttesa[j]=false;
          up(&veicoli[j]);
      }
      j=j+1;
   }
up(&mutex);
}
```