Politiche di assegnazione risorse tramite semafori privati

I semafori ***privati*** sono tali solo per come vengono usati: il meccanismo messo a disposizione dalle funzioni è lo stesso, senza alcun controllo su quale processo/thread usa i semafori e come

- Un semaforo privato s_priv_P «di» un processo P (o di una classe di processi) è inizializzato a 0
- Solo il processo P (o un processo della classe associata al semaforo) esegue down(&s_priv_P); la esegue quando deve attendere che diventi vera una condizione (booleana) di sincronizzazione
- qualsiasi processo (P incluso) può eseguire up(&s_priv_P) se serve svegliare P, o serve non farlo sospendere se fa down, perché è vera la condizione di sincronizzazione

Dining Philosophers: soluzione tramite semafori privati

```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N–1)%N          /* number of i's left neighbor */
#define RIGHT            (i+1)%N            /* number of i's right neighbor */
#define THINKING         0                  /* philosopher is thinking */
#define HUNGRY           1                  /* philosopher is trying to get forks */
#define EATING           2                  /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
      while (TRUE) {                        /* repeat forever */
            think( );                       /* philosopher is thinking */
            take_forks(i);                  /* acquire two forks or block */
            eat( );                         /* yum-yum, spaghetti */
            put_forks(i);                   /* put both forks back on table */
      }
}
```

Dining Philosophers: soluzione tramite semafori privati

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                     /* enter critical region */
      state[i] = HUNGRY;                /* record fact that philosopher i is hungry */
      test(i);                          /* try to acquire 2 forks */
      up(&mutex);                       /* exit critical region */
      down(&s[i]);                      /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                     /* enter critical region */
      state[i] = THINKING;              /* philosopher has finished eating */
      test(LEFT);                       /* see if left neighbor can now eat */
      test(RIGHT);                      /* see if right neighbor can now eat */
      up(&mutex);                       /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

"ovviamente" down(&s[i]) è dopo up(&mutex)

per non fermarsi quando fa down, in take_forks;
o per svegliare il vicino, in put_forks;

Dining Philosophers: soluzione tramite semafori privati

La soluzione può essere elaborata per garantire l'assenza di *starvation*: per esempio non permettendo ad un filosofo di mangiare più di *k* volte di fila se un suo vicino è affamato:

- ogni volta che il filosofo *i* preleva la forchetta che serve anche a un vicino affamato si incrementa un contatore
- quando il contatore arriva a *k* il filosofo si sospende

Il problema dei lettori e scrittori

Supponiamo di avere una struttura dati che viene utilizzata da molti utenti contemporaneamente (come un database, anche se qui non parliamo di DBMS in cui ovviamente questi problemi esistono)

Le operazioni che gli utenti possono richiedere sono di due tipi: consultazione (lettura) o modifica (scrittura).

Per assicurare un utilizzo efficiente del sistema vogliamo che le consultazioni possano procedere in parallelo.

Tuttavia per assicurare la consistenza della struttura dati e delle informazioni ottenute durante la consultazione, occorre garantire che ogni modifica avvenga in mutua esclusione con qualsiasi altra operazione (lettura o scrittura)

Il problema dei lettori e scrittori: soluzione con semafori

Utilizziamo:

- una variabile condivisa **rc** (inizialmente =0) che indica quanti lettori stanno usando il database
- due semafori:
  - **db** inizializzato a 1 per accedere al database in «mutua» esclusione (in questo caso: 1 scrittore *oppure* N lettori)
  - **mutex** inizializzato a 1 associato alla variabile **rc**

```
void writer(void)
{
        while (TRUE) {                  /* repeat forever */
                think_up_data( );       /* noncritical region */
inizio scrittura down(&db);             /* get exclusive access */
                write_data_base( );     /* update the data */
fine scrittura  up(&db);                /* release exclusive access */
        }
}
```

Il problema dei lettori e scrittori: soluzione con semafori

```
void reader(void)
{
    while (TRUE) {                          /* repeat forever */
        down(&mutex);                       /* get exclusive access to rc */
        rc = rc + 1;                        /* one reader more now */
        if (rc == 1) down(&db);             /* if this is the first reader ... */
        up(&mutex);                         /* release exclusive access to rc */
        read_data_base( );                  /* access the data */
        down(&mutex);                       /* get exclusive access to rc */
        rc = rc – 1;                        /* one reader fewer now */
        if (rc == 0) up(&db);               /* if this is the last reader ... */
        up(&mutex);                         /* release exclusive access to rc */
        use_data_read( );                   /* noncritical region */
    }
}
```

inizio lettura

fine lettura

Se il database è occupato da uno scrittore, il primo lettore viene sospeso su down(&db), i successivi su down(&mutex)

dopo che il primo lettore ha superato l'istruzione down(&db) (trova verde; o si sospende e viene svegliato dallo scrittore) gli altri riusciranno ad entrare (rc>1, non fanno down(&db))

Quando un lettore conclude la lettura, aggiorna rc, se é l'ultimo lettore rilascia la mutua esclusione sul DB (up(&db)).

Il problema dei lettori e scrittori: soluzione con semafori

Questa soluzione non garantisce l'assenza di starvation, es. per uno scrittore in attesa se c'è un flusso continuo di lettori

Anche in questo caso si può realizzare una politica più sofisticata utilizzando semafori privati diversi per i lettori e per gli scrittori

Per evitare la starvation:

- se un lettore arriva quando ci sono degli scrittori in attesa, si blocca (per non superarli)
- quando un lettore termina di usare il DB e non vi sono altri lettori che operano sul DB, sveglia uno scrittore in attesa (se c'è)
- quando uno scrittore termina di usare il DB, se vi sono lettori in attesa li sveglia tutti, se no sveglia uno scrittore

Usiamo due semafori privati sem_priv_lettori e sem_priv_scrittori, entrambi inizializzati a 0, che servono ai lettori o agli scrittori per sospendersi quando non possono accedere al DB

Il problema dei lettori e scrittori: soluzione senza starvation

```
iniziolettura()
 {down(&mutex);
    if ( (scrittori_attivi==0) && (scrittori_bloccati==0)
                  { lettori_attivi++;
              up(&sem_priv_lettori);}
    else
              lettori_bloccati++;
    up(&mutex);
    down(&sem_priv_lettori);
 }
finelettura()
{down(&mutex);
  lettori_attivi--;
  if ((lettori_attivi==0) && (scrittori_bloccati>0))
         { scrittori_bloccati--;
      scrittori_attivi++;
      up(&sem_priv_scrittori);}
  up(&mutex);
}
```

Per non fermarsi sulla successiva down

se un lettore arriva quando ci sono degli scrittori (attivi o) in attesa, si blocca (per non superarli se in attesa)

quando un lettore termina di usare il DB e non vi sono altri lettori attivi, sveglia uno scrittore in attesa (se c'è)

Il problema dei lettori e scrittori: soluzione senza starvation

```
inizioscrittura()
 { down(&mutex);
   if ( (scrittori_attivi==0) &&
        (lettori_attivi==0) )
              { scrittori_attivi++;
            up(&sem_priv_scrittori);}
   else
            scrittori_bloccati++;
   up(&mutex);
   down(&sem_priv_scrittori);
 }
```

Per non fermarsi sulla successiva down

```
finescrittura()
{down(&mutex);
  scrittori_attivi--;
  if (lettori_bloccati > 0)
      while (lettori_bloccati>0)
         { lettori_bloccati--;
           lettori_attivi++;
           up(&sem_priv_lettori); }
  else if (scrittori_bloccati>0)
       { scrittori_bloccati--;
         scrittori_attivi++;
         up(&sem_priv_scrittori);}
  up(&mutex); }
```

quando uno scrittore termina di usare il DB, se vi sono lettori in attesa li sveglia tutti, se no sveglia uno scrittore

Uso di semafori privati per il problema generale di assegnazione risorse

Si consideri un insieme di processi P1,..,Pn che devono poter acquisire ed utilizzare un certo numero di risorse da un pool di *k* risorse equivalenti. Prima di poter utilizzare una risorsa il processo Pi deve acquisirla. Dopo aver usato le risorse acquisite le dovrà rilasciare

```
Pi
...
acquisisci_risorsa();
<usa risorsa>
rilascia_risorsa();
...
```

La soluzione potrebbe semplicemente essere:

- un semaforo *ris* inizializzato a *k;*
- acquisisci_risorsa() { *down(&ris)* ;}
- rilascia_risorsa() {up(&ris*);* }

Uso di semafori privati per il problema generale di assegnazione risorse

Se due processi Pi e Pj sono in attesa di una risorsa (bloccati su down(&ris)), e un terzo processo Ph esegue up(&ris), la scelta di quale dei due processi viene svegliato è nell'implementazione di up()

Può andare bene, ma se si vuole applicare una *politica* diversa, occorre realizzarla esplicitamente. Lo si può fare con questo *meccanismo*:

- il processo che deve attendere una risorsa viene sospeso su un proprio semaforo privato s_priv_i
- si ricorda in una variabile sospeso_i che Pi è sospeso
- al rilascio di una risorsa, si sceglie (con la *politica*) quale risvegliare tra tutti i processi sospesi (noti, grazie alle variabili sospeso_i) e lo si risveglia con up(&sem_priv_i)

Se la politica si basa su una suddivisione dei processi in classi, si usa un semaforo privato per ciascuna classe (uno per ciascun processo è un caso particolare con classi di 1 solo processo)

Una soluzione

```
mutex inizializzato a 1;
priv[K] inizializzati a 0; /* K è il numero di classi di processi */

acquisisci(int i)  /* i = identificatore della classe di processi, o del processo*/
 { down(&mutex);
    if (condizione di sincronizzazione soddisfatta)
          {<annota risorsa allocata>
           up(&priv[i]); }
    else
        <annota processo di classe i bloccato>
    up(&mutex);
    down(&priv[i]);
 }

rilascia()
 {int j;
  down(&mutex);
  <annota risorsa rilasciata>
  while (condizione di sincronizzazione vera per qualche  processo )
      {  j = scegli_classe_processo_da_attivare;
         <annota risorsa allocata>;
         < annota processo non più sospeso >;
        up(&priv[j]);}
  up(&mutex);
 }
```

down(&priv[i]) è sospensiva
solo se prima non si è passati da up(&priv[i])

Semafori privati e cinque filosofi

La soluzione del problema dei 5 filosofi basata su semafori privati è un caso particolare

```
Filosofo(i)
while (TRUE)
     { think();
       take_forks(i);
       eat();
       put_forks(i);
     }
```

```
void test(int i)
{    if (state[i] == HUNGRY
          && state[LEFT]!=EATING
          && state[RIGHT]!=EATING)
        { state[i]=EATING;
          up(&s[i]) } }
```

LEFT=i+1%N; RIGHT=i-1%N

```
void take_forks(int i)
{ down(&mutex);
   state[i]=HUNGRY;
   test(i)
   up(&mutex);
   down(&s[i]);
}
```

```
void put_forks(int i)
{ down(&mutex);
   state[i]=THINKING;
   test(LEFT);
   test(RIGHT);
up(&mutex);
}
```

Semafori privati e cinque filosofi

```
void take_forks(int i)
{ down(&mutex);
   state[i]=HUNGRY;
   if (state[i] == HUNGRY
        && state[LEFT]!=EATING
        && state[RIGHT]!=EATING)
      { state[i]=EATING;
        up(&s[i]) }
   else
        state[i]=HUNGRY;
   up(&mutex);
   down(&s[i]);
}
```

test(i)

```
acquisisci(int i)
{ down(&mutex);

  if (condizione di
      sincronizzazione
      soddisfatta)
   {<annota risorsa allocata>
     up(priv[i]); }
   else
       <annota processo i bloccato>
   up(mutex);
   down(priv[i]);
}
```

Semafori privati e cinque filosofi

```
void put_forks(int i)
{ down(&mutex);
   state[i]=THINKING;
   if (state[LEFT] == HUNGRY
        && state[(LEFT-1)%N]!=EATING
        && state[(LEFT+1)%N]!=EATING)
      { state[LEFT]=EATING;
        up(&s[LEFT]) }
   if (state[RIGHT] == HUNGRY
        && state[(RIGHT-1)%N]!=EATING
        && state[(RIGHT+1)%N]!=EATING)
      { state[RIGHT]=EATING;
        up(&s[RIGHT]) }
up(&mutex);
}
```

test(LEFT) test(RIGHT)

```
Rilascia(int i)
 {down(mutex);
  <annota risorsa rilasciata>
  while (condizione di
       sincronizzazione vera per
        qualche  processo )
  {  j =scegli_proc_da_attivare;
     <annota risorsa allocata e
       processo non più sospeso >
    up(priv[j]);}

up(mutex);
 }
```