

# Realizzazione di Politiche di Gestione delle Risorse: i Semafori Privati

## Condizione di sincronizzazione

- Qualora si voglia realizzare una determinata politica di gestione delle risorse, la decisione se ad un dato processo sia consentito proseguire la esecuzione dipende dal verificarsi di una condizione detta ***condizione di sincronizzazione***.
- La condizione è espressa in termini di variabili che rappresentano lo stato della risorsa e di variabili locali ai singoli processi.
- Più processi possono essere bloccati durante l'accesso ad 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 processi bloccati possono essere ***contemporaneamente verificate***.

***Problema:*** quale processo mettere in esecuzione (accesso alla risorsa mutuamente esclusivo)?

➔ Definizione di una *politica per il risveglio dei processi bloccati*.

- Nei casi precedenti la condizione di sincronizzazione era particolarmente semplificata (vedi mutua esclusione) e la scelta di quale processo riattivare veniva effettuata tramite l'algoritmo implementato nella *signal*.
- Normalmente questo algoritmo, dovendo essere sufficientemente generale ed il più possibile efficiente, coincide con quello *FIFO*.

# Condizione di sincronizzazione

**Esempio 1:** Su un buffer da N celle di memoria più produttori possono depositare messaggi di *dimensione diversa*.

**Politica di gestione:** tra più produttori ha *priorità di accesso* quello che fornisce il messaggio di *dimensione maggiore*.

➔ La politica di gestione comporta che finché un produttore, il cui messaggio ha dimensioni *maggiori* dello spazio disponibile nel buffer, rimane sospeso *nessun altro produttore* può depositare un messaggio anche se la sua dimensione potrebbe essere contenuta nello spazio libero del buffer.

**Condizione di sincronizzazione :** il deposito può avvenire se 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 prevede la riattivazione tra i produttori sospesi, di quello il cui messaggio ha la *dimensione maggiore*, semprechè esista sufficiente spazio nel buffer.
- Se lo spazio disponibile non è sufficiente *nessun produttore viene riattivato*.

**Esempio 2:** Un insieme di processi utilizza un insieme di risorse comuni  $R_1, R_2, \dots, R_n$ .

- Ogni processo può utilizzare una qualunque delle risorse. La **condizione di sincronizzazione** si riduce quindi a valutare se esiste una risorsa libera.
- A ciascun processo è assegnata *una priorità*.
- In fase di riattivazione dei processi sospesi viene scelto quello cui corrisponde *la massima priorità*

**Esempio 3:** Con riferimento al problema dei *lettori scrittori*, si intende realizzare una politica di gestione che eviti condizioni di *attesa indefinita* per entrambe le classi di processi.

# Semaforo Privato

**Def:** Un semaforo si dice ***privato*** per un processo quando solo tale processo può eseguire sul semaforo la primitiva **wait**.

- La primitiva **signal** sul semaforo può essere invece eseguita **anche da altri processi**.
- Il semaforo privato viene inizializzato con il valore **zero**

# Acquisizione e Rilascio di una risorsa

- E` possibile implementare particolari politiche di gestione di risorse facendo uso di semafori privati:
  - il processo che acquisisce la risorsa puo` (se la condizione di sincronizzazione non e` soddisfatta) eventualmente sospendersi sul suo semaforo privato
  - chi rilascia la risorsa, risveglierà uno tra i processi sospesi (in base alla politica scelta) mediante una signal sul semaforo privato del processo prescelto.



# Procedure di acquisizione e di rilascio

```
semaphore mutex;
```

```
semaphore priv[Nproc]=...; /*      array di semafori  
                                inizializzati a 0;*/
```

```
mutex.value=1;
```

```
void acquisizione (int i)
```

```
{  wait(&mutex);
```

```
    if (< condizione di sincronizzazione>)
```

```
    {          <allocazione della risorsa>  
              signal(&priv[i]);
```

```
    }
```

```
    else
```

```
        <indicazione di sospensione del processo>
```

```
    signal(&mutex);
```

```
    wait(&priv[i]);
```

```
}
```

```
void rilascio ()
{
    int i;
    wait(&mutex);
    <rilascio della risorsa>;
    if (<esiste almeno un processo sospeso per il
        quale la cond. di sincronizzazione e`
        soddisfatta>)
    {
        <scelta del processo Pi da risvegliare>;
        <allocazione della risorsa a Pi>;
        <indicazione che Pi non e` piu` sospeso>;
        signal(&priv[i]);
    }
    signal(&mutex);
}
```

# Proprietà della soluzione

- a) La sospensione del processo, nel caso in cui la condizione di sincronizzazione non sia soddisfatta, *non può avvenire entro la sezione critica* in quanto ciò impedirebbe ad un processo che rilascia la risorsa di accedere a sua volta alla sezione critica e di riattivare il processo sospeso. La sospensione avviene *al di fuori della sezione critica*.
- b) La specifica del particolare algoritmo di assegnazione della risorsa non è opportuno che sia realizzata *nella primitiva signal*.
- ➔ Nella soluzione proposta è possibile programmare esplicitamente tale algoritmo scegliendo in base ad esso il processo da attivare ed *eseguendo la signal sul suo semaforo privato*.

Lo schema presentato può, in certi casi, presentare degli inconvenienti.

- a) l'operazione wait sul semaforo privato viene *sempre eseguita* anche quando il processo richiedente non deve essere bloccato.
- b) Il codice relativo all'assegnazione della risorsa viene *duplicato* nelle procedure acquisizione e rilascio

Si può definire uno schema che non ha questi inconvenienti.

```
semaphore mutex;
```

```
semaphore priv[Nproc]=...; /* array di  
                               semafori privati  
                               inizializzati a 0;
```

```
mutex.value=1;
```

```
for(int i=0;i<Nproc; i++) priv[i].value=0;
```

```
void acquisizione (int i)
```

```
{  wait(&mutex);
```

```
    if (! <condizione di sincronizzazione>)
```

```
    {      <indicazione di sospensione del processo>
```

```
        signal(&mutex);
```

```
        wait(&priv[i]);
```

```
        <indicazione processo non più sospeso>;
```

```
    }
```

```
    <allocazione della risorsa>
```

```
    signal(&mutex);
```

```
}
```

```
void rilascio ()
{
    int i;
    wait(&mutex);
    <rilascio della risorsa>;
    if (<esiste almeno un processo sospeso per il
        quale la cond. di sincronizzazione e`
        soddisfatta>)
    {
        <scelta del processo Pi da risvegliare>;
        signal(&priv[i]);
    }
    else signal(&mutex);
}
```

## Commenti:

- A differenza della soluzione precedente, in questo caso risulta più complesso realizzare la riattivazione di più processi per i quali risulti vera contemporaneamente la condizione di sincronizzazione.
- Lo schema prevede infatti che il processo che rilascia la risorsa attivi *al più un processo sospeso*, il quale dovrà a sua volta provvedere alla riattivazione di eventuali altri processi.

# Soluzione Esempio 1

Su un buffer da N celle di memoria più produttori possono depositare messaggi di *dimensione diversa*.

**Politica di gestione:** tra più produttori ha *priorità di accesso* quello che fornisce il messaggio di *dimensione maggiore*

## Soluzione:

```
int richiesta[Nproc]={0,0,...0};
int sospesi=0; int vuote=N;
semaphore mutex, priv[Nproc];
message Buffer[N];
mutex.value=1;
for(int i=0;i<Nproc; i++) priv[i].value=0;
```



```
void acquisizione(int i, int m)
{ wait (&mutex);
  if ((sospesi==0)&&(vuote>=m))
  {   vuote= vuote-m;
      <assegnazione al processo di m celle di buffer>;
      signal(&priv[i]);
  }
  else
  {   sospesi++;
      richiesta[i]= m;
  }
  signal(&mutex);
  wait (&priv[i]);
}
```

```

void rilascio (int m) /* m: num. celle rilasciate */
{
    int k;
    wait(&mutex);
    vuote= vuote+m;
    while (sospesi!=0)
    {
        <individuazione del processo Pk
        con la massima richiesta>;
        if (richiesta[k]<=vuote)
        {
            vuote=vuote-richiesta[k];
            <assegnazione a Pk delle celle richieste>;
            richiesta[k]=0;
            sospesi--;
            signal(&priv[k]);
        }
        else break; /* uscita dal ciclo */
    }
    signal(&mutex);
}

```

# Soluzione Esempio 2

Un insieme di processi utilizza un insieme di risorse comuni  $R_1, R_2, \dots, R_n$ ; a ciascun processo è assegnata *una priorità*.

**Politica di gestione:** In fase di riattivazione dei processi sospesi viene scelto quello cui corrisponde *la massima priorità*

**Soluzione:** *introduzione delle seguenti variabili:*

- ***PS[i]***: variabile logica che assume il valore vero se il processo  $P_i$  è sospeso; il valore falso diversamente.
- ***risorse[j]***: variabile logica che assume il valore falso se la risorsa  $j$ -esima è occupata; il valore vero diversamente.
- ***disponibile***: il numero delle risorse non occupate
- ***sospesi***: il numero dei processi sospesi
- ***mutex***: un semaforo di mutua esclusione
- ***priv(i)***: il semaforo privato del processo  $P_i$

```
int risorse[Nris]={1,1,1..1};
int PS[Nproc]={0,0,..0};
int sospesi=0;
int disponibile=Nris;
semaphore mutex;
semaphore priv[Nproc]=...; /* array di
                             semafori privati
                             inizializzati a 0;

mutex.value=1;
for(int i=0;i<Nproc; i++) priv[i].value=0;
```

```
int Richiesta (int i)
/* ritorna il numero di risorsa allocata al
processo richiedente Pi */
{
    int k=0;
    wait(&mutex);
    if (!disponibile)
    {
        sospesi++;
        PS[i]=1;
        signal(&mutex);
        wait(&priv[i]);
        PS[i]=0;
        sospesi--;
    }
    while(!risorse[k]) k++;
    disponibile--;
    risorse[k]=0;
    signal(&mutex);
    return k;
}
```

```
void Rilascio (int x) /*x: numero risorsa */
{ int j;
  wait(&mutex);
  disponibile++;
  risorse[x]=1; /*la risorsa viene liberata*/
  if (sospesi>0)
  { <seleziona il processo Pj a massima priorità tra
    quelli sospesi utilizzando il vettore PS>
    signal(&priv[j]);
  }
  else signal(&mutex);
}
```

# Considerazioni sulle soluzioni presentate

- Ogni processo che nella fase di acquisizione della risorsa trova la condizione di sincronizzazione non soddisfatta, **deve lasciare traccia in modo esplicito** della sua sospensione entro la sezione critica.
- Il processo che libera la risorsa deve infatti eseguire la primitiva `signal(priv[i])` **solo se** esistono processi sospesi. In tutte le soluzioni e' stata introdotta un'apposita variabile per indicare il numero dei processi **sospesi**.
- La fase di assegnazione di una risorsa ad un processo è **separata** dalla fase di uso della risorsa stessa.
- Occorre quindi **lasciare traccia in modo esplicito** entro la sezione critica della **assegnazione** e quindi della **non disponibilità** della risorsa, decrementando la variabile *disponibili* e assegnando a `risorse[k]` il valore falso.