Soluzioni ai problemi di Mutua Esclusione
Primitive di sincronizzazione

Soluzioni basate su primitive di sincronizzazione

Le primitive di sincronizzazione piú comuni sono:

- Lock (mutex) - realizzati in software, con primitive linguistiche, con system call. In java:
  - keyword *synchronized*
  - classe Lock
- Mutex rientranti (un thread puó acquisire lo stesso mutex piú volte) - realizzati in software, con primitive linguistiche, con system call.
- Monitor - realizzati con primitive linguistiche
- Semafori - realizzati con system call, monitor
- Scambio di messaggi - realizzati con system call, in software

1) NB: le primitive sono usate sia per risolvere la MUTUA ESCLUSIONE che per risolvere la SINCRONIZZAZIONE tra thread o processi

2) NB: le primitive sono EQUIVALENTI (da ognuna é possibile ottenere le altre)

Una NonSoluzione.

Pensiamo a delle semplici primitive SLEEP() e WAKEUP() che rispettivamente sospendono e risvegliano il thread o processo.

```
void Prod() {
  while(true){
    el=produci();
    while(n==N) sleep();   //se il buffer e' pieno, aspetta
    A[n]=el; n++;
    if(n==1) wakeup(Cons); //1 elemento: sveglia il consumatore
  }
}
void Cons() {
  while(true){
    while(n==0) sleep();   //se il buffer e' vuoto, aspetta
    el=A[n]; n--;
    if(n==N-1) wakeup(Prod); //1 posto libero: sveglia il produttore
  }
}
```

Problema: un context switch prima di sleep() provoca la perdita del risveglio

Soluzioni basate sui Lock

Il Lock é un oggetto software che puo' essere Disponibile/Nondisponibile.
Quando un thread chiede di acquisire il Lock

- se il Lock é Disponibile, diventa Nondisponibile e si continua l'esecuzione
- se il Lock é Nondisponibile, il thread aspetta fino a che il lock diventa disponibile

Esempio (pseudocodice): supponiamo che i mutex siano gestiti con le API, lock() e unlock().

```
void incrementa()
{
    lock(&mutex);  count = count + 1;   unlock(&mutex);
}
int get_count()
{
    int  c;
    lock(&mutex);  c = count;  unlock(&mutex);
    return (c);
}
```

Soluzioni basate sui lock rientranti

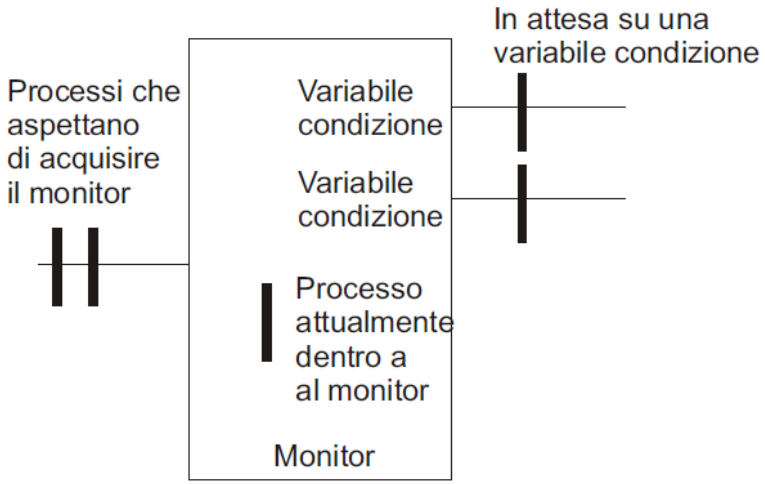
- I lock Rientranti (Reentrant mutex) possono acquisire lo stesso lock piú volte.
- I mutex impliciti di Java sono rientranti (reentrant).
- Bisogna sbloccare il Lock alla fine dell'utilizzo! Esempio in Java:

```
Lock m = ...;
m.lock();

try {
    // accesso alla risorse protetta da questo lock
} finally {
   m.unlock();
}
```

MONITOR

- Simile alla definizione di classe, che associa a delle variabili condivise le procedure che le utilizzano.
- Garantisce che i metodi del Monitor siano usati in mutua esclusione.
- Sincronizzazione tra i metodi realizzata mediante VARIABILI CONDIZIONE e operatori *wait, signal* che si appoggiano su di esse
- *var_cond.wait* sospende il thread e rilascia la mutua esclusione per consentire ad altri di modificare le variabili
- *var_cond.signal* risveglia processi sospesi su var_cond.



Schema del Produttore/Consumatore con i MONITOR

```
monitor prodcons
  condition pieno, vuoto;
  entry Prod() {
    while(true){
      el=produci();
      while(n==N) pieno.wait;   //se il buffer e' pieno, aspetta
      A[n]=el; n++;
      if(n==1) vuoto.signal; //1 elemento: sveglia il consumatore
    }
  }
  entry Cons() {
    while(true){
      while(n==0) vuoto.wait;   //se il buffer e' vuoto, aspetta
      el=A[n]; n--;
      if(n==N-1) pieno.signal;//1 posto libero: sveglia il produttore
    }
  }
end monitor;
```

IL SEMAFORO di Dijkstra

Dijkstra introduce due operazioni NON INTERROMPIBILI, (che Dijkstra chiama P e V) che possiamo chiamare **down() e up()**, che operano su una variabile intera non negativa *s*.
Funzione delle operazioni:

```
down(s)
{
    while (s<=0); /* loop di attesa: BUSY WAITING */
    s--;
}
up(s)
{
    s++;
}
```

Ma: le operazioni DEVONO ESEGUIRE ATOMICAMENTE!

Il semaforo di Djikstra

Protezione di una sezione critica:

```
Processo1(){
   while(1){
      down(s);
      Sezione_Critica1();
      up(s);
      Sezione_NON_Critica1();
   }
}
```

```
Processo2(){
   while(1){
      down(s) ;
      Sezione_Critica2();
      up(s);
      Sezione_NON_Critica1();
   }
 }
```

Le **SPINLOCK** consumano tempo di calcolo (**busy waiting**).

Vantaggi: portabili, semplici, adatti per attese brevi.

IL SEMAFORO di Dijkstra

- Il semaforo di Dijkstra aspetta in busy waiting
- Viene chiamato semaforo Spin-Lock
- Ci sono altri semafori in cui l'attesa é fatta spostando il processo in stato di Wait
- Vengono chiamati semafori Wait-Lock
- I semafori possono essere binari (s=0,1) o contatori (s¿0)

Esempio nella sincronizzazione:

```
s1=0
s2=0
```

```
Thread1
funzione1()
up(s1)
```

```
Thread2
down(s1)
funzione2()
up(s2)
```

```
Thread3
down(s2)
funzione3()
```

I SEMAFORI SPIN LOCK USANDO TSL e SWAP

In pratica:

```
Usando TSL
(s condivisa)

down()
{
    while (TSL(&s));
}

up()
{
    s=0;
}
```

oppure:

```
Usando Swap
(a condivisa, b locale)

down()
{    b=1;
     while(b==1) Swap(a,b); //spinlock
}

up()
{
    a=0;
}
```

ATTENZIONE: in questi casi non sono le down, up ad essere atomiche, ma le TSL o Swap!

I SEMAFORI WAIT LOCK

Per attese lunghe (*wait locks*) **si introduce una coda d'attesa su s**:

```
down(s)
{
   if(s<=0)
       aggiungi il descrittore del processo in coda su s;
   else s--;
 }

up(s)
{
  if(c'e' un descrittore in attesa su s)
      esegui il processo;
  else
      s++;
 }
```

Le wait lock sono piú lente ma adatte ad attese lunghe.

I SEMAFORI WAIT LOCK

- Le down() e up() dei semafori wait-lock sono implementati nel kernel (chiamate di sistema)
- In Linux sono gestiti con le chiamate semget(), semctl(), semop()
- semctl() inizializza o preleva proprietá del semaforo, semop() effettua down() o up() (parametro)
- Nei pthread sono gestiti con le chiamate sem_init(), sem_wait(), sem_post(). sem_wait() effettua la down(), sem_post la up()

Soluzioni basate sullo scambio di messaggi

Assumiamo che esistano due primitive:

- send(dest, &msg) - non bloccante, invia il messaggio msg a dest (pid, tid, file, indirizzo di rete...)
- recv(src, &msg) - bloccante, riceve msg da src (stesso pid, tid, file, indirizzo di rete...)

Assumendo che due processi (per esempio produttore/consumatore) siano in mutuaesclusione:

```
void produttore(void)
{
     while (TRUE) {
          item=produci();
          ...
          send( consumatore, &item );
     }
}
```

```
void consumatore(void)
{
      while (TRUE) {
           recv(producer, &item);
           ...
            consuma(item);
     }
}
```

Nota: Il meccanismo puó essere modellato con la teoria delle code!!