Sincronizzazione fra processi

- Mutex
- Semafori

Mutex

- Tutte le variabili e le risorse condivise vanno protette mediante una qualche forma di sincronizzazione.
  - Senza sincronizzazione, applicazioni concorrenti possono dare luogo a comportamenti non prevedibili.
- Un *mutex* (*mut*ual *ex*clusion) è un oggetto che permette a processi o thread concorrenti di sincronizzare l'accesso a dati condivisi.
- Ogni volta che un processo o thread ha bisogno di accedere ai dati condivisi, acquisisce il mutex (***mutex_lock()***).
- Quando l'operazione è terminata, il mutex viene rilasciato (***mutex_unlock()***), permettendo ad un altro processo o thread di acquisirlo per eseguire le sue operazioni.

Acquisizione e rilascio dei mutex

- Un mutex ha due stati: bloccato e non bloccato:
  - Quando un mutex è bloccato (0) da un thread, gli altri thread che tentano di bloccarlo restano in attesa;
  - Quando il thread che blocca rilascia il mutex (1), uno dei thread in attesa lo acquisisce.
- Un processo o thread per bloccare un mutex usa ***mutex_lock()***
  - La funzione ritorna quando il mutex è stato bloccato dal processo o thread chiamante;
  - Il mutex resta bloccato fino a quando non è sbloccato dal processo o thread chiamante.
- Per sbloccare un mutex si usa ***mutex_unlock()***
  - Se vi sono più processi in attesa di acquisire il mutex, la politica di scheduling dei processi o thread stabilisce chi lo acquisisce.

Mutua esclusione con i mutex

- Variabili condivise:
  - **mutex = 1;**
- Processo P_i:

```
do {
    mutex_lock ( mutex );
      sezione critica
    mutex_unlock ( mutex );
      sezione non critica
} while (1);
```

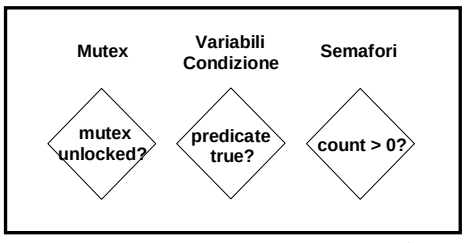

Serializzazione con i mutex

- Problema: Eseguire l'istruzione S_1 di P_1 prima che S_2 è stato eseguita da P_2
- Si impiega un mutex *flag* inizializzato a 0 (bloccato)
- Codice:

P_1	P_2
S_1; ***mutex_unlock*(*flag*)** ;	***mutex_lock*(*flag*)**; ***S_2***;

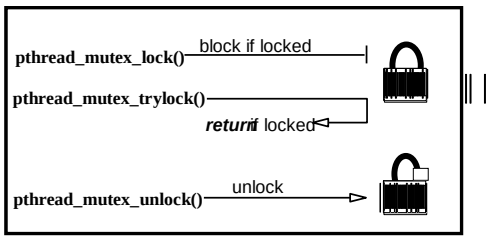
Sincronizzazione dei thread

- Posix1.c mette a disposizione due primitive per la sincronizzazione dei thread in processi multipli: **mutex** e **variabili condizione**.
- POSIX.1b permette di sincronizzare thread in processi multipli con **i semafori**.
- Altri meccanismi di sincronizzazione includono gli *spinlocks*, *lock* in lettura/scrittura e le *barriere*.



Operazioni sui mutex

```
#include<pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```



Esempio

```
#include <pthread.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>

extern void  fatal_error(int err_num, char *func);
#define check_error(return_val, msg) {              \
       if (return_val != 0)  fatal_error(return_val, msg); \
}

/* Creazione di due thread  che:
   - dopo un intervallo di tempo casuale,
   - aggiornano metà elementi di un array condiviso
     con il proprio pid.
   Si impone un lock in modo da avere
   - l'accesso esclusivo all'array
   - l'aggiornamento dell'indice dell'array */

/* Variabili condivise */
int  condivisa[] = {0,0,0,0,0,0,0,0,0,0};
int ncondivisa = 0;
```

```
/* Creazione di due thread  che:
   - dopo un intervallo di tempo casuale,
   - aggiornano metà elementi di un array condiviso
     con il proprio pid.
   Si impone un lock in modo da avere
   - l'accesso esclusivo all'array
   - l'aggiornamento dell'indice dell'array */

/* Variabili condivise */
int  condivisa[] = {0,0,0,0,0,0,0,0,0,0};
int ncondivisa = 0;

/* il lock per regolare l'accesso alla memoria condivisa */
pthread_mutex_t Mutex = PTHREAD_MUTEX_INITIALIZER;

/* aggiornatore casuale */
void *aggiorna ( int dim);
int main()
{
  pthread_t tid1,tid2;
  int retcode, k, dim;

  dim=sizeof(condivisa)/sizeof(int);

  retcode=pthread_create(&tid1,NULL,(void *(*)())aggiorna,(void*) dim);
  check_error(retcode, "create failed");

  retcode=pthread_create(&tid2,NULL,(void *(*)())aggiorna,(void*) dim);
```

```
/* aggiornatore casuale */
void *aggiorna ( int dim);
int main()
{
  pthread_t tid1,tid2;
  int retcode, k, dim;

  dim=sizeof(condivisa)/sizeof(int);

  retcode=pthread_create(&tid1,NULL,(void * (*)())aggiorna,(void *) dim);
  check_error(retcode, "create failed");

  retcode=pthread_create(&tid2,NULL,(void * (*)())aggiorna,(void *) dim);
  check_error(retcode, "create failed");

  /* attende la terminazione di entrambi i thread */
  retcode = pthread_join(tid1,NULL);
  check_error(retcode, "join failed");

  retcode = pthread_join(tid2,NULL);
  check_error(retcode, "join failed");

  for(k=0; k<dim; k++)
      printf("condivisa[%d]=%d\n", k,condivisa[k]);
  exit(0);
}
```

```
  check_error(retcode, "join failed");

  for(k=0; k<dim; k++)
      printf("condivisa[%d]=%d\n", k,condivisa[k]);
  exit(0);
}

void * aggiorna(int dim)
{
   int i;
   int sl;

   srand(getpid());
   for(i=0; i<dim/2 ; i++) {
    sl = 1 + (int) (5.0 * rand()/(RAND_MAX+1.0));
    sleep(sl);

    pthread_mutex_lock( &Mutex );
    condivisa[ncondivisa]=getpid();
    ncondivisa++;
    pthread_mutex_unlock( &Mutex );
  }
  return( (void *) NULL);
}

/* Print error information, exit with -1 status. */
void
fatal_error(int err_num, char *function)
{
```

```
   srand(getpid());
   for(i=0; i<dim/2 ; i++) {
    sl = 1 + (int) (5.0 * rand()/(RAND_MAX+1.0));
    sleep(sl);

    pthread_mutex_lock( &Mutex );
    condivisa[ncondivisa]=getpid();
    ncondivisa++;
    pthread_mutex_unlock( &Mutex );
  }
  return( (void *) NULL);
}

/* Print error information, exit with -1 status. */
void
fatal_error(int err_num, char *function)
{
       char    *err_string;

       err_string = strerror(err_num);
       fprintf(stderr, "%s error: %s\n", function, err_string);
       exit(-1);
}
```

Semafori

- I mutex forniscono un meccanismo per sincronizzare l'accesso ad una singola risorsa condivisa.
- Nei casi in cui è necessario coordinare l'accesso a più istanze della stessa risorsa, si utilizzano i *semafori*.
- I semafori sono simili ai mutex, in quanto forniscono una sincronizzazione per l'accesso alle risorse condivise.
- Come per i mutex, esistono due operazioni principali di acquisizione e rilascio di un semaforo.
- Il concetto è stato introdotto da W. Dijkstra nel 1965.

Semafori

- Un singolo semaforo è utilizzato per regolare l'accesso a risorse condivise multiple.
- Il semaforo è inizializzato ad un valore pari al numero di istanze della risorsa disponibili;
  - Se le risorse non sono disponibili (il valore è = 0) il processo o thread resta in attesa fino a quando una istanza si rende disponibile (il valore è > 0); il processo acquisisce quindi il semaforo, (il valore è decrementato);
  - Quando il processo o thread ha finito, rilascia il semaforo (il valore è incrementato).
  - Se più processi o thread sono in attesa di una risorsa, la politica di scheduling decide chi acquisisce la risorsa.

Esempio

- Supponiamo di avere un'applicazione, costituita da 10 thread, che scarica file su computer remoti tramite i 4 modem di cui è dotato il sistema su cui è in esecuzione.
- Un semaforo è inizializzato con il valore dei modem disponibili.
  - Ciascun thread che vuole scaricare un file deve prima acquisire il semaforo (***sem_wait()***):
  - Se non vi sono modem disponibili, il thread si blocca finché il semaforo è disponibile; quando il semaforo è stato acquisito, il thread può usare uno dei modem.
  - Quando il thread ha finito di utilizzare il modem, rilascia il semaforo (***sem_signal()***), il che permette ad un altro thread di acquisire il semaforo e utilizzare un modem.

Semafori

- L'acquisizione ed il rilascio avvengono mediante le funzioni ***sem_wait()*** e ***sem_signal()***.
- La definizione in un codice fittizio è la seguente:

```
sem_wait ( S ) {
        while ( S<= 0 )
            ; // no–op
        S– –;
}
```

```
sem_signal ( S ) {
        S++;
}
```

- Le modifiche al valore del semaforo e la sua verifica devono essere eseguite in maniera atomica.

Uso dei semafori

- I semafori sono uno strumento utile per sincronizzare l'accesso a risorse condivise.
- In genere un'operazione su un semaforo è un'operazione più costosa di una su un mutex.
  - I semafori, come le altre strutture condivise, sono memorizzate nel kernel→ il loro uso richiede chiamate di sistema;
  - Le operazioni sui mutex nei thread possono essere gestite dalla libreria in spazio utente
- Quando la capacità di contare dei semafori non serve, conviene utilizzare i mutex.
- Nel caso in cui il valore del semaforo può essere solo 0 o 1, si parla di *semafori binari*.

Attesa attiva

- Le soluzioni al problema della mutua esclusione, così come la definizione di semaforo, prevedono che si debba continuamente verificare un ciclo stretto, fino a che l'ingresso non è consentito.
- Questo comportamento porta ad uno spreco di tempo di CPU, e un semaforo sì fatto viene detto *spinlock*, perché "gira" (*spin*) finché rimane bloccato (*lock*).
  - Il vantaggio di uno spinlock è che non si deve compiere alcun cambio di contesto mentre un processo attende per un accesso; utili per attese con tempo minore del cambio di contesto
  - Lo svantaggio è che si può avere *un'inversione delle priorità*
    - Due processi H (alta priorità) e L (bassa priorità) sono in esecuzione; L è nella sezione critica e H è in attesa attiva; L non viene mai scelto quando H è in esecuzione → **stallo**!!!

Implementazione dei semafori

- Per evitare l'attesa attiva si può ricorrere ad una definizione alternativa di semaforo.
- Si definisce un semaforo come un record:

```
typedef struct {
    int valore;
    struct processo *L;
} semaforo;
```

- Si assume che siano disponibili due operazioni (syscall):
  - ***block()*** sospende il processo che la invoca;
  - ***wakeup(P)*** riprende l'esecuzione di un processo bloccato ***P***.

Implementazione dei semafori

- Le operazioni sui semafori possono essere definite come…

```
void sem_wait( semaforo S ):
        S.valore– –;
        if ( S.valore < 0 ) {
            aggiungi il processo P a S.L;
            block();
        }
```

|S.valore| è il numero di processi in coda.

```
void sem_signal( semaforo S ):
        S.valore++;
        if ( S.valore <= 0 ) {
            rimuovi il processo P da S.L;
            wakeup( P );
        }
```

In un ambiente con un unico processore è possibile disabilitare le interruzioni all'inizio delle operazioni ***sem_wait*** e ***sem_signal***.

Deadlock e starvation

- ***Stallo (****Deadlock)*: due o più processi sono in attesa indefinita di un evento che può essere generato solo da uno dei due processi in attesa.
- Siano *S* e *Q* due semafori inizializzati a 1:

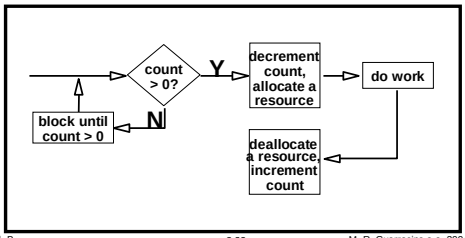
P_0	P_1
sem_wait(S);	***sem_wait(Q);***
sem_wait(Q);	***sem_wait(S);***
⋮	⋮
sem_signal(S);	***sem_signal(Q);***
sem_signal(Q);	***sem_signal(S);***

Se dopo ***sem_wait(S)*** di P_0 viene eseguita ***sem_wait(Q)*** di P_1 si ha un **deadlock**.

- ***Attesa indefinita*** *(Starvation)* —; un processo attende indefinitamente ad un semaforo, e non può essere rimosso dalla coda del semaforo su cui è sospeso.

Semafori POSIX

- I semafori sono stati introdotti in POSIX1.b come meccanismo per sincronizzare i processi e possono essere *named* e *unnamed*
- Sui semafori *named* si usano le operazioni ***sem_open***, ***sem_close*** e ***sem_unlink***, e sono utili per sincronizzare più processi.
- I semafori *unnamed* possiedono funzioni simili ai mutex e per utilizzarli in applicazioni multiprocesso è necessario allocarli nella memoria condivisa.

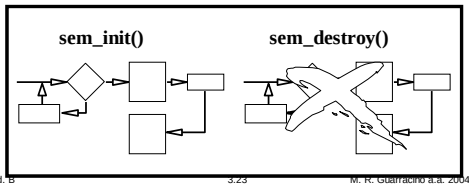


Inizializzazione dei semafori

- Per inizializzare un semaforo *unnamed*, si usa:

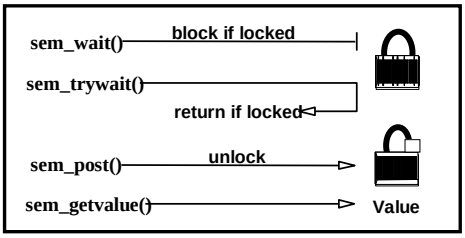
```
int sem_init(
    sem_t           *sem
    int             pshared
    unsigned int    value
)
```

- Inizializza il semaforo ***sem*** con il valore ***value***
- Un valore zero per ***pshared*** indica che il semaforo è usato solo in thread del processo chiamante; è non zero altrimenti



Operazioni sui semafori

- Per attendere indefinitamente l'acquisizione di un semaforo, si usa **sem_wait()**
- Per tentare di acquisirlo, ma continuare se non è disponibile, si usa **sem_trywait()**
- Per rilasciare un semaforo, si usa **sem_post()**
- Per controllarne il valore, si usa **sem_getvalue()**



Esempio produttori - consumatori

```
typedef struct {
    char buf[BSIZE];
    sem_t occupied;
    sem_t empty;
    int nextin;
    int nextout;
    sem_t pmut;
    sem_t cmut;
} buffer_t;

buffer_t buffer;

sem_init(&buffer.occupied, 0, 0);
sem_init(&buffer.empty,    0, BSIZE);
sem_init(&buffer.pmut,     0, 1);
sem_init(&buffer.cmut,     0, 1);
buffer.nextin = buffer.nextout = 0;
```


Esempio produttori - consumatori

```
void producer(buffer_t *b,
              char item) {
    sem_wait(&b->empty);
    sem_wait(&b->pmut);

    b->buf[b->nextin] = item;
    b->nextin++;
    b->nextin %= BSIZE;

    sem_post(&b->pmut);
    sem_post(&b->occupied);
}
```

```
char consumer(buffer_t *b) {
    char item;

    sem_wait(&b->occupied);
    sem_wait(&b->cmut);

    item = b->buf[b->nextout];
    b->nextout++;
    b->nextout %= BSIZE;

    sem_post(&b->cmut);
    sem_post(&b->empty);

    return(item);
}
```


Approfondimenti

- **Seqlocks and Futexes**
  - ☞ http://www.linux-mag.com/2004-06/compile_01.html