Mutex

Cosa sono i semafori?

- I semafori sono primitive fornite dal sistema operativo per permettere la sincronizzazione tra processi e/o thread.

Operazioni sui semafori

- In generale sono TRE le operazioni che vengono eseguite da un processo su un semaforo:
  - Create: creazione di un semaforo.
  - Wait: attesa su di un semaforo dove si verifica il valore del semaforo

```
while(sem_value <=0)
   ;// wait;
```
ad esempio blocca il thread

```
sem_value--;
```

  - Post: incremento del semaforo.

```
sem_value++;
```

Cosa sono i mutex? (1 di 2)

- Una variabile mutex è una variabile che serve per la *protezione* delle sezioni critiche:
  - variabili condivise modificate da più thread

- Il mutex è un semaforo **binario,** cioè il valore può essere 0 (*occupato*) oppure 1 (*libero*)

Cosa sono i mutex? (2 di 2)

- Pensiamo ai mutex come a delle serrature:
  - il primo thread che ha accesso alla coda dei lavori lascia fuori gli altri thread fino a che non ha portato a termine il suo compito.

- Viene inserito un mutex nelle sezioni di codice nelle quali vengono condivisi i dati.

Garantire la Mutua Esclusione (1 di 2)

- Due thread devono decrementare il valore di una variabile globale `data` se questa è maggiore di zero
  - `data = 1`

```
THREAD1              THREAD2
if(data>0)             if(data>0)
   data --;               data --;
```

Garantire la Mutua Esclusione (2 di 2)

- A seconda del tempo di esecuzione dei due thread, la variabile `data` assume valori diversi.

```
Data     THREAD1              THREAD2
1        if(data>0)
1           data --;
0                         if(data>0)
0                            data --;
```

0 = valore finale di data

```
1        if(data>0)
1                         if(data>0)
1           data --;
0                            data --;
```

-1 = valore finale di data

Procedure nell' uso dei mutex

- Creare e inizializzare una variabile mutex
- Più thread tentano di accedere alla risorsa invocando l'operazione di `lock`
- Un solo thread riesce ad acquisire il mutex mentre gli altri si bloccano
- Il thread che ha acquisito il mutex manipola la risorsa
- Lo stesso thread la rilascia invocando la `unlock`
- Un altro thread acquisisce il mutex e così via
- Distruzione della variabile mutex

Creazione mutex

- Per creare un mutex è necessario usare una variabile di tipo `pthread_mutex_t` contenuta nella libreria pthread
  - `pthread_mutex_t` è una struttura che contiene:
    - ⇨ Nome del mutex
    - ⇨ Proprietario
    - ⇨ Contatore
    - ⇨ Struttura associata al mutex
    - ⇨ La *coda* dei processi *sospesi* in attesa che mutex sia libero.
    - ⇨ … e simili

Inizializzazione mutex

- statica
  - contestuale alla dichiarazione
- dinamica
  - attraverso
    - `pthread_mutex_t mutex;`
    - `pthread_mutex_init (&mutex, NULL);`

Inizializzazione statica

- Per il tipo di dato `pthread_mutex_t`, è definita la macro di inizializzazione `PTHREAD_MUTEX_INITIALIZER`

- Il mutex è un tipo definito "ad hoc" per gestire la mutua esclusione quindi il valore iniziale può essergli assegnato anche in modo statico mediante questa macro.

```
/* Variabili globali */

pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
```

Inizializzazione dinamica

```
pthread_mutex_t mutex;

int pthread_mutex_init( pthread_mutex_t *mutex, const
pthread_mutexattr_t *mattr )
```

- `pthread_mutex_t *mutex`
  - puntatore al mutex da inizializzare
- `pthread_mutexattr_t *mattr`
  - attributi del mutex da inizializzare
  - se `NULL` usa valori default
- Valore di ritorno
  - sempre il valore `0`

Interfacce

- Su mutex sono possibili solo due operazioni: **locking** e **unlocking (**equivalenti a *wait* e *signal* sui semafori)

Interfaccia: `Lock`

- Ogni thread, prima di accedere ai dati condivisi, deve effettuare la `lock` su una stessa variabile mutex.

- Blocca l'accesso da parte di altri thread.

- Se più thread eseguono l'operazione di `lock` su una stessa variabile mutex, solo uno dei thread termina la `lock` e prosegue l'esecuzione, gli altri rimangono bloccati nella `lock`. In tal modo, il processo che continua l'esecuzione può accedere ai dati (protetti mediante la mutex).

Operazioni: `lock` e `trylock`

- `lock`
  - bloccante (standard)
- `trylock`
  - non bloccante (utile per evitare deadlock)
  - è come la `lock()` ma se si accorge che la mutex è già in possesso di un altro thread (e quindi si rimarrebbe bloccati) restituisce immediatamente il controllo al chiamante con risultato `EBUSY`

lock

```
int pthread_mutex_lock( pthread_mutex_t *mutex )
```

- `pthread_mutex_t *mutex`
  - puntatore al mutex da bloccare
- Valore di ritorno
  - `0` in caso di successo
  - diverso da `0` altrimenti

trylock

```
int pthread_mutex_trylock( pthread_mutex_t *mutex )
```

- `pthread_mutex_t *mutex`
  - puntatore al mutex da bloccare
- Valore di ritorno
  - `0` in caso di successo e si ottenga la proprietà della mutex
  - `EBUSY` se il mutex è occupato

Interfaccia: `Unlock`

- Libera la variabile mutex.
- Un altro thread che ha precedentemente eseguito la `lock` su un mutex potrà allora terminare la `lock` ed accedere a sua volta ai dati.

unlock

```
int pthread_mutex_unlock( pthread_mutex_t *mutex )
```

- `pthread_mutex_t *mutex`
  - puntatore al mutex da sbloccare
- Valore di ritorno
  - `0` in caso di successo

destroy

```
int pthread_mutex_destroy( pthread_mutex_t *mutex )
```

- Elimina il mutex
- `pthread_mutex_t *mutex`
  - puntatore al mutex da distruggere
- Valore di ritorno
  - `0` in caso di successo
  - `EBUSY` se il mutex è occupato

Esempio 4: uso dei mutex (1 di 2)

```
#include <pthread.h>
int a=1, b=1;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
void* thread1(void *arg) {
   pthread_mutex_lock(&m);
   printf("Primo thread (parametro: %d)\n", *(int*)arg);
   a++; b++;
   pthread_mutex_unlock(&m);
}
void* thread2(void *arg) {
   pthread_mutex_lock(&m);
   printf("Secondo thread (parametro: %d)\n", *(int*)arg);
   b=b*2; a=a*2;
   pthread_mutex_unlock(&m);
}
```

Continua ⇨

Esempio 4: uso dei mutex (2 di 2)

```
main() {
   pthread_t threadid1, threadid2;
   int i = 1, j=2;
   pthread_create(&threadid1, NULL, thread1, (void *)&i);
   pthread_create(&threadid2, NULL, thread2, (void *)&j);
   pthread_join(threadid1, NULL);
   pthread_join(threadid2, NULL);
   printf("Valori finali: a=%d b=%d\n", a, b);
}
```

Semafori classici

Semafori classici (generali) (1 di 2)

- Semafori il cui valore può essere impostato dal programmatore
  - utilizzati per casi più generali di sincronizzazione
  - esempio: produttore consumatore
- Interfaccia
  - operazione `wait`
  - operazione `post` (`signal`)

Mutex vs Semaforo (1 di 2)

- Il mutex è un tipo definito "ad hoc" per gestire la mutua esclusione quindi il valore iniziale può essergli assegnato anche in modo statico mediante la macro `PTHREAD_MUTEX_INITIALIZER`.
- Al contrario, un semaforo come il `sem_t` deve essere di volta in volta inizializzato dal programmatore *col valore desiderato.*

Mutex vs Semaforo (2 di 2)

- Un semaforo può essere impiegato come un mutex

inizializzo un mutex;

```
pthread_mutex_lock(&mutex);
```

sezione critica

```
pthread_mutex_unlock(&mutex);
```

inizializzo un semaforo (1);

```
sem_wait(&sem);
```

sezione critica

```
sem_post(&sem);
```

Semafori classici (generali) (2 di 2)

- Semafori classici e standard POSIX
  - non presenti nella prima versione dello standard
  - introdotti insieme come estensione real-time con lo standard IEEE POSIX 1003.1b (1993)
- Utilizzo
  - associati al tipo `sem_t`
  - includere l'header

```
#include <semaphore.h>
```

errno

- Per riportare il tipo di errore il sistema usa la variabile globale `errno` definita nell'header `errno.h`
- Il valore di `errno` viene sempre impostato a zero all'avvio di un programma.
- La procedura da seguire è sempre quella di controllare `errno` immediatamente dopo aver verificato il fallimento della funzione attraverso il suo codice di ritorno.

Stato di errore (2 di 2)

- L'esempio seguente mostra un programma completo e molto semplice, in cui si crea un errore, tentando di scrivere un messaggio attraverso lo standard input. Se effettivamente si rileva un errore associato a quel flusso di file, attraverso la funzione `ferror()`, allora si passa alla sua interpretazione con la funzione `strerror()`

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
int main (void){
    char *cp;
    fprintf (stdin, “Hello world!\n");
    if (ferror (stdin)){
        cp = strerror (errno);
        fprintf (stderr, "Attenzione: %s\n", cp);
      }
    return 0;
}
```

Esempio `errno` con i semafori

```
…
   ret = sem_init(sem, pshared, value);
   if (ret == -1){
       printf("sem_init: thread %d,
                                %s: failed: %s\n",
                                pthread_self(),
                                msg, strerror(errno));
       exit(1);
     }
…
```

Creazione semaforo

- `sem_t`: tipo di dato associato al semaforo

```
sem_t sem;
```

Inizializzazione (1 di 2)


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

- I semafori richiedono un'inizializzazione *esplicita* da parte del programmatore
- `sem_init` serve per inizializzare il valore del contatore del semaforo specificato come primo parametro

Inizializzazione (2 di 2)

- `sem_t *sem`
  - puntatore al semaforo da inizializzare, cioè l'indirizzo dell'oggetto semaforo sul quale operare
- `int pshared`
  - flag che specifica se il semaforo è condiviso fra più processi
- `unsigned int *value`
  - valore iniziale da assegnare al semaforo
- Valore di ritorno
  - `0` in caso di successo,
  - `-1` altrimenti con la variabile `errno` settata a `EINVAL` se il semaforo supera il valore `SEM_VALUE_MAX`

Interfaccia `wait` (1 di 2)

- Consideriamo il semaforo come un intero, sul cui valore la funzione `wait` esegue un test
- Se il valore del semaforo è minore o uguale a zero (*semaforo rosso*), **la `wait` si blocca**, forzando un cambio di contesto a favore di un altro dei processi pronti che vivono nel sistema
- Se il test ha successo cioè se il semaforo presenta un valore maggiore od uguale ad `1` (*semaforo verde*), la `wait` decrementa tale valore e ritorna al chiamante, che può quindi procedere nella sua elaborazione.

```
void wait (semaforo s) {
   s.count--;
   if (s.count <= 0)
    <cambio di contesto>;
}
```

Interfaccia `wait` (2 di 2)

- Due varianti
  - `wait`: bloccante (standard)
  - `trywait`: non bloccante (utile per evitare deadlock)

wait

```
int sem_wait( sem_t *sem )
```

- `sem_t *sem`
  - puntatore al semaforo da decrementare
- Valore di ritorno
  - sempre `0`

trywait

```
int sem_trywait( sem_t *sem )
```

- `sem_t *sem`
  - puntatore al semaforo da decrementare
- Valore di ritorno
  - `0` in caso di successo
  - `-1` se il semaforo ha valore `0`
    - ⇨ setta la variabile `errno` a `EAGAIN`

Interfaccia `signal`

- L'operazione di `signal` incrementa il contatore del semaforo
- Se a seguito di tale azione il contatore risultasse ancora minore od uguale a zero, significherebbe che altri processi hanno iniziato la `wait` ma hanno trovato il semaforo rosso
- la `signal` sveglia quindi uno di questi; pertanto **esiste una coda di processi bloccati per ciascun semaforo**.

```
void signal (semaforo s) {
   s.count++;
   if (s.count <= 0)
    <sveglia processo>;
}
```

sem_post

```
int sem_post( sem_t *sem )
```

- `sem_t *sem`
  - puntatore al semaforo da incrementare
- Valore di ritorno
  - `0` in caso di successo
  - `-1` altrimenti con la variabile `errno` settata in base al tipo di errore
    - ⇨ `sem_post` restituisce `EINVAL` se il semaforo supera il valore `SEM_VALUE_MAX` dopo l'incremento

sem_destroy

```
int sem_destroy( sem_t *sem )
```

- `sem_t *sem`
  - puntatore al semaforo da distruggere
- Valore di ritorno
  - `0` in caso di successo
  - `-1` altrimenti con la variabile `errno` settata in base al tipo di errore
    - ⇨ `sem_destroy` restituisce `EBUSY` se almeno un thread è bloccato sul semaforo

sem_getvalue

- Serve per poter leggere il valore attuale del contatore del semaforo

```
int sem_getvalue( sem_t *sem, int *sval )
```

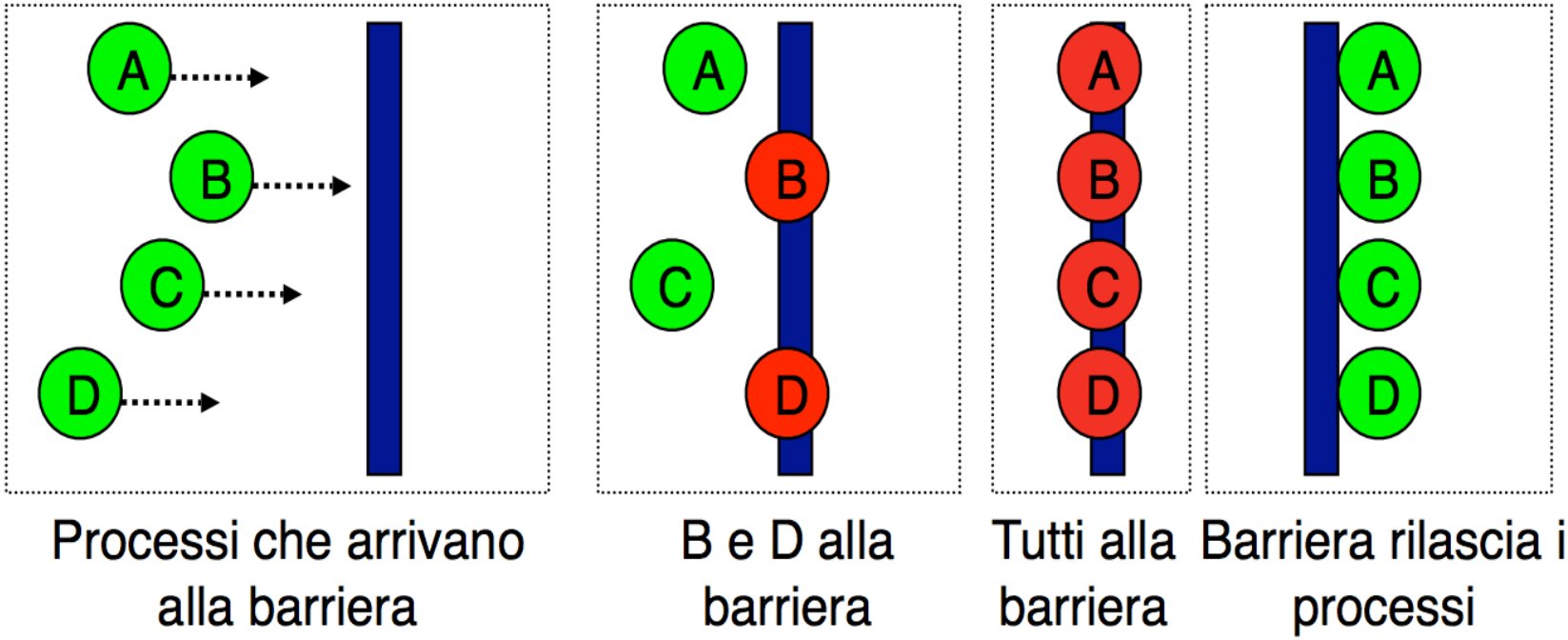
- `sem_t *sem`
  - puntatore del semaforo di cui leggere il valore
- `int *sval`
  - valore del semaforo
- Valore di ritorno
  - sempre `0`

Esempio 7: produttori e consumatore

- Supponiamo che ci siano DUE thread produttori e un thread consumatore:
  - I TRE thread devono sincronizzarsi tra di loro per evitare incongruenze nelle strutture dati.

- Fornire una soluzione al problema, usando i semafori di tipo "mutex" e i semafori generici.
  - Supporre che esista un buffer "limitato".

Esempio: Sincronizzazione con barriera



Prima soluzione

Thread # 1:
...
signal(ready1);
wait(ready2);
...

Thread #2:
...
signal(ready2);
wait(ready1);
...

Variabili:
Semaphore ready 1 = 0;
Semaphore ready2 = 0;

Seconda soluzione

Thread # 1: (driver)
...
signal(busy);
wait(done);
...

Thread #2: (controller)
...
wait(busy);
signal(done);
...

Variabili:
Semaphore busy = 0;
Semaphore done = 0;

Variabili condition

Condition vs Semafori

- Le variabili condition sono molto diverse dai semafori di sincronizzazione, anche se semanticamente fanno la stessa cosa

- Le primitive delle condition si preoccupano di rilasciare ed acquisire la **mutua esclusione** prima di bloccarsi e dopo essere state sbloccate
- I semafori generali, invece, prescindono dalla presenza di altri meccanismi

Cosa sono le variabili condition?

- **Strumento di sincronizzazione**: consente la sospensione dei thread in attesa che sia soddisfatta una condizione logica.

- Una condition variable, quindi, è utilizzata per sospendere l'esecuzione di un thread in attesa che si verifichi un certo evento.

- Ad ogni condition viene associata una **coda** per la sospensione dei thread.

- La variabile condizione non ha uno *stato*, rappresenta solo una *coda di thread*.

Variabili condition

- Attraverso le variabili condition è possibile implementare condizioni più complesse che i thread devono soddisfare per essere eseguiti.
- Linux garantisce che i threads bloccati su una condizione vengano sbloccati quando essa cambia.

Mutua esclusione

- Una variabile condizione non fornisce la mutua esclusione.
- C'è bisogno di un MUTEX per poter sincronizzare l'accesso ai dati.

Sincronizzazione

- Una variabile condition è sempre associata ad un mutex
  - un thread ottiene il mutex e testa il predicato
  - se il predicato è verificato allora il thread esegue le sue operazioni e rilascia il mutex
  - se il predicato non è verificato, in modo atomico
    - ⇨ il mutex viene rilasciato (implicitamente)
    - ⇨ il thread si blocca sulla variabile condition
  - un thread bloccato *riacquisisce* il mutex nel momento in cui viene svegliato da un altro thread

Creazione condition

- Oggetti di sincronizzazione su cui un processo si può bloccare in attesa
  - associate ad una condizione logica arbitraria
  - generalizzazione dei semafori
  - nuovo tipo `pthread_cond_t`
  - attributi variabili condizione di tipo `pthread_condattr_t`

Inizializzazione statica

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- Per il tipo di dato `pthread_cond_t`, è definita la macro di inizializzazione
  `PTHREAD_COND_INITIALIZER`

Inizializzazione dinamica

```
int pthread_cond_init( pthread_cond_t *cond,
                       pthread_condattr_t *cond_attr )
```

- `pthread_cond_t *cond`
  - puntatore ad un'istanza di condition che rappresenta la condizione di sincronizzazione
- `pthread_condattr_t *cond_attr`
  - punta a una struttura che contiene gli attributi della condizione
  - se NULL usa valori di default

Distruzione variabili condition

```
int pthread_cond_destroy( pthread_cond_t *cond )
```

- Dealloca tutte le risorse allocate per gestire la variabile condizione specificata
- Non devono esistere thread in attesa della condizione

- `pthread_cond_t *cond`
  - puntatore ad un'istanza di condition da distruggere
- Valore di ritorno
  - `0` in caso di successo oppure un codice d'errore $\neq 0$

Interfacce

- Operazioni fondamentali:
  - `wait` *(sospensione)*
  - `signal` *(risveglio)*

Interfaccia `wait`

- La `wait` serve per sincronizzarsi con una certa condizione all'interno di un blocco di dati condivisi e protetti da un `mutex`

- La presenza del `mutex` fra i parametri garantisce che, al momento del bloccaggio, esso venga liberato, eliminando a monte possibili errori di programmazione che potrebbero condurre a condizioni di deadlock.

- Se la `wait` ritorna in modo regolare, è garantito che la mutua esclusione, sul semaforo `mutex` passatole, è stata nuovamente acquisita.

wait

```
int pthread_cond_wait( pthread_cond_t *cond,
pthread_mutex_t *mutex )
```

- `pthread_cond_t *cond`
  - puntatore ad un'istanza di condition che rappresenta la condizione di sincronizzazione
  - puntatore all'oggetto condizione su cui bloccarsi
- `pthread_mutex_t *mutex`
  - l'indirizzo di un semaforo di mutua esclusione necessario alla corretta consistenza dei dati
- Valore di ritorno
  - sempre `0`

Interfaccia `signal`

- Il mutex deve essere rilasciato esplicitamente, altrimenti si potrebbe produrre una condizione di deadlock.

Lo sblocco dei processi nella coda di wait della condizione è NON DETERMINISTICO. Quindi non è dato sapere chi verrà svegliato dalla signal().

- Due varianti
  - Standard: sblocca un solo thread bloccato
  - Broadcast: sblocca tutti i thread bloccati

signal

```
int pthread_cond_signal ( pthread_cond_t *cond)
```

- Se esistono thread sospesi nella coda associata a `cond`, viene risvegliato il primo.
- Se non vi sono thread sospesi sulla condizione, la `signal` non ha effetto.

- `pthread_cond_t *cond`
  - puntatore all'oggetto condizione
- Valore di ritorno
  - sempre `0`

broadcast

```
int pthread_cond_broadcast ( pthread_cond_t *cond )
```

- `pthread_cond_t *cond`
  - puntatore all'oggetto condizione
- Valore di ritorno
  - sempre `0`

Valutazione condizione

- Perchè la condizione va testata usando un ciclo WHILE invece che un semplice IF?

```
pthread_mutex_lock(&mutex);
while(!condition_to_hold)
   ptread_cond_wait(&cond, &mutex);
computation();
pthread_mutex_unlock(&mutex);
```

Prod / Cons ERRATO

```
void*  prod(Object o) {
   pthread_mutex_lock(&mutex);
   if(buf.size()==MAX_SIZE) {
      ptread_cond_wait(&cond, &mutex); // called if the buffer is full
   }
   buf.add(o);
   ptread_cond_signal(&cond); // called in case there are any getters or putters waiting
}

void* cons get() {
   // Y
   pthread_mutex_lock(&mutex);
   if (buf.size()==0) {
      ptread_cond_wait(&cond, &mutex); // called if the buffer is empty
      // X
   }
   Object o = buf.remove(0);
   ptread_cond_signal(&cond);  // called if there are any getters or putters waiting
   return o;
}
```

Stato della coda

- Non è prevista una funzione per verificare lo stato della coda associata a una condizione.

Signal VS. Broadcast

- Siamo sicuri che nell' esempio di prima vada fatta una signal e non una broadcast?

Signal VS. Broadcast

- Supponiamo che ci siano 6 processi: 3 produttori (P1, P2, P3) e 3 consumatori (C1, C2, C3).
- Cosa potrebbe succedere con il codice scritto su?

Grazie per l'attenzione.