Semafori classici

Semafori

- I semafori sono primitive, implementate attraverso dei contatori, fornite dal sistema operativo per permettere la sincronizzazione tra processi e/o thread.
- Per queste primitive è garantita l'atomicità. Quindi, ogni modifica o check del valore di un semaforo può essere effettuata senza sollevare race conditions.

Race condition

- Più processi accedono concorrentemente agli stessi dati, e il risultato dipende dall'ordine di interleaving dei processi.
  - Frequenti nei sistemi operativi multitasking, sia per dati in user space sia per strutture in kernel.
  - Estremamente pericolose: portano al malfunzionamento dei processi coo-peranti, o anche (nel caso delle strutture in kernel space) dell'intero sistema
  - difficili da individuare e riprodurre: dipendono da informazioni astratte dai processi (decisioni dello scheduler, carico del sistema, utilizzo della memoria, numero di processori, . . .)

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**
- **Differenza sostanziale: un mutex deve sempre essere sbloccato dal thread che lo ha bloccato, mentre per un semaforo l'operazione `post` può non essere eseguita dal thread che ha eseguito la chiamata `wait`.**

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) (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)`

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>
#include <errno.h>
```

errno

- Quasi tutte le funzioni delle librerie del C sono in grado di individuare e riportare condizioni di errore, ed è una norma fondamentale di buona programmazione controllare sempre che le funzioni chiamate si siano concluse correttamente.

- In genere le funzioni di libreria usano un valore speciale per indicare che c'è stato un errore. Di solito questo valore è `-1` o un puntatore `NULL` o la costante `EOF` (a seconda della funzione); ma questo valore segnala solo che c'è stato un errore, non il tipo di errore.

- 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 (1 di 2)

- **Per verificare la presenza di uno stato di errore si usa la funzione `ferror()` che restituisce un valore diverso da zero se questo stato esiste effettivamente:**

```
int ferror (FILE *flusso_di_file);
```

- **Per interpretare l'errore annotato nella variabile `errno` e visualizzare direttamente un messaggio attraverso lo standard error, si può usare la funzione `perror()`**

```
void perror (const char *s);
```

- **La funzione `perror()` mostra un messaggio in modo autonomo, aggiungendo davanti la stringa che può essere fornita come primo argomento**

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
  - se 1 il semaforo è condiviso tra processi
  - se 0 il semaforo è privato del processo
  - attualmente l'implementazione supporta solamente `pshared = 0`
- `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: lettori e scrittori (1 di 5)


```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define LUN 20
#define CICLI 1
#define DELAY 100000
struct {
  char scritta[LUN+1];
  /* Variabili per la gestione del buffer */
  int primo, ultimo;
  /* Variabili semaforiche */
  sem_t mutex, piene, vuote;
} shared;
void *scrittore1(void *);
void *scrittore2(void *);
void *lettore(void *);
```

Continua ⇨

Esempio 7: lettori e scrittori (2 di 5)


```
int main(void) {
  pthread_t s1TID, s2TID, lTID;
  int res, i;
  shared.primo = shared.ultimo = 0;
  sem_init(&shared.mutex, 0, 1);
  sem_init(&shared.piene, 0, 0);
  sem_init(&shared.vuote, 0, LUN);
  pthread_create(&lTID, NULL, lettore, NULL);
  pthread_create(&s1TID, NULL, scrittore1, NULL);
  pthread_create(&s2TID, NULL, scrittore2, NULL);
  pthread_join(s1TID, NULL);
  pthread_join(s2TID, NULL);
  pthread_join(lTID, NULL);

 printf("E' finito l'esperimento ....\n");
}
```

Continua ⇨

Esempio 7: lettori e scrittori (3 di 5)


```
void *scrittore1(void *in) {
  int i, j, k;
  for (i=0; i<CICLI; i++) {
    for(k=0; k<LUN; k++) {
      sem_wait(&shared.vuote); /* Controllo che il buffer non sia pieno */
      sem_wait(&shared.mutex);        /* Acquisisco la mutua esclusione */
      shared.scritta[shared.ultimo] = '-';       /* Operazioni sui dati */
      shared.ultimo = (shared.ultimo+1)%(LUN);
      sem_post(&shared.mutex);                        /* Libero il mutex */
      sem_post(&shared.piene);    /* Segnalo l'aggiunta di un carattere */
      for(j=0; j<DELAY; j++);              /* ... perdo un po' di tempo */
    }
  }
  return NULL;
}
```

Continua ⇨

Esempio 7: lettori e scrittori (4 di 5)

```
void *scrittore2(void *in) {
  int i, j, k;
  for (i=0; i<CICLI; i++) {
    for(k=0; k<LUN; k++) {
      sem_wait(&shared.vuote); /* Controllo che il buffer non sia pieno */
      sem_wait(&shared.mutex);        /* Acquisisco la mutua esclusione */
      shared.scritta[shared.ultimo] = '+';       /* Operazioni sui dati */
      shared.ultimo = (shared.ultimo+1)%(LUN);
      sem_post(&shared.mutex);                    /* Libero il mutex */
      sem_post(&shared.piene);    /* Segnalo l'aggiunta di un carattere */
      for(j=0; j<DELAY; j++);              /* ... perdo un po' di tempo */
    }
  }
  return NULL;
}
```

Continua ⇨

Esempio 7: lettori e scrittori (5 di 5)

```
void *lettore(void *in) {
  int i, j, k; char local[LUN+1]; local[LUN] = 0;      /* Buffer locale */
  for (i=0; i<2*CICLI; i++) {
    for(k=0; k<LUN; k++) {
      sem_wait(&shared.piene); /* Controllo che il buffer non sia vuoto */
      sem_wait(&shared.mutex);        /* Acquisisco la mutua esclusione */
      local[k] = shared.scritta[shared.primo];   /* Operazioni sui dati */
      shared.primo = (shared.primo+1)%(LUN);
      sem_post(&shared.mutex);                        /* Libero il mutex */
      sem_post(&shared.vuote);     /* Segnalo che ho letto un carattere */
      for(j=0; j<DELAY; j++);              /* ... perdo un pò di tempo */
    }
    printf("Stringa = %s \n", local);
  }
  return NULL;
}
```