Indice

Appunti di sistemi operativi

I processi

Il sistema operativo mette a disposizione dell'utente (un programma in esecuzione) una macchina virtuale che astrae la macchina fisica creando un ambiente in cui ogni programma si vede come l'unico in esecuzione senza interferenze esterne.

Questa parallelizzazione viene realizzata attraverso i processi, ossia degli esecutori completi che eseguono i vari programmi. Ogni processo può essere visto come una macchina virtuale a disposizione del programma. I processi vengono parallelizzati dal sistema operativo. Un sistema operativo con questa capacità è detto multiprogrammato o multitasking.

I processi non sono esclusivamente usati dagli utenti, essi infatti possiedono due modalità di esecuzione:

- supervisore: riservata al sistema operativo
- utente: riservata all'utente

Anche il sistema operativo internamente usa dei processi. Questi processi hanno accesso diretto al processore in quanto sono eseguiti in modalità supervisore.

Caratteri generali di un processo

Ogni processo, tranne il primo, viene creato da un altro processo, in gergo è figlio di un processo padre. Ogni processo è identificato da un Process ID (PID) univoco.

La memoria di ogni processo è divisa in diverse parti dette segmenti:

- il segmento di testo (codice)
- il segmento dati: contiene tutti i dati sia statici che dinamici; quelli dinamici si dividono in allocati sullo stack o sulla heap
- il segmento di sistema: contiene i dati non gestiti dal programma ma dal sistema operativo, come ad esempio la tabella dei file aperti

Il sistema operativo fornisce a servizio delle applicazioni dei servizi di sistema per la manipolazione dei processi.

Chiamate di sistema per la gestione dei processi (POSIX)

1. Creazione di un processo figlio: `pid_t fork(void)`

   Crea un processo figlio identico al processo padre (all'istante di `fork()`). L'unico valore diverso tra i due è il PID (Process ID). La chiamate restituisce al processo il padre il PID del figlio e al processo figlio 0.

   Se la creazione del processo fallisce, viene restituito -1.

2. Terminazione di un processo: `void exit(int)`

   Termina il processo e restituisce un codice al processo padre.

3. Conoscere il proprio PID: `pid_t getpid(void)`

4. Aspetta fino alla terminazione del figlio: `pid_t wait(int*)`

   L'esecuzione del padre viene sospesa finché non termina il figlio il cui PID è il `pid_t` restituito. Il valore di ritorno moltiplicato per 256 viene salvato nel puntatore passato come argomento. Se i figli sono più di uno, la `wait()` aspetta

un figlio qualunque. Se devo aspettare un figlio in particolare bisogna usare `pid_t waitpid(pid_t, int*, int)` (il terzo parametro si chiama `options` e lo considereremo maggiore di 0).

Se un processo figlio termina quando il processo padre non è ancora arrivato alla `wait` esso terminerà ma non "morirà" in quanto esso deve restituire il codice di uscita al padre. Un processo in questa situazione viene detto zombie. Quando il padre chiamerà la `wait` l'esecuzione non si fermerà e il processo zombie verrà terminato definitivamente.

5. Cambiare il codice del programma: `int execl(char*, char*, ...)`

   Sostituisce il segmento dati e il segmento codice del processo con quello di un altro programma. La prima stringa di parametri è il percorso del programma da mandare in esecuzione, seguono N stringhe che specificano gli argomenti da passare a questo nuovo programma. Il primo argomento deve essere il nome del programma, mentre l'ennesimo deve essere `NULL`.

Proviamo a scrivere un semplice programma che usa queste chiamate di sistema:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char **argv) {
  pid_t pid, pidr;
  int ret_status = 0, stato_exit;

  // N.B.: l'ordine di esecuzione tra padre e figlio non è definito
  pid = fork();

  if (pid == 0) {
    printf("Sono il figlio con PID %d\n", getpid());
    ret_status = 1;
    exit(ret_status);
  } else {
    printf("Sono il padre\n");
    pidr = wait(&stato_exit);
    exit(ret_status);
  }
}
```

I thread

Abbiamo due esigenze nel nostro modello di parallelizzazione:

- Creare programmi concorrenti senza cooperazione tra di loro
- Creare attività che devono condividere dati e sincronizzarsi tra di loro

Il primo è realizzato tramite i processi, il secondo invece grazie ai thread.

Un thread non è altro che un flusso di controllo, ossia una sequenza di istruzioni. Più thread possono essere attivi contemporaneamente. A differenza dei processi, i thread non possono esistere da soli, ma devono essere sempre contenuti in un processo. Ogni processo ha almeno 1 thread.

Una rappresentazione schematica dei thread e dei processi può essere:

```
             Processo
+---------------------------------+
| codice, dati, file e sistema    |
+----------+----------+----------+
| Thread 1 | Thread 2 | Thread 3 |
| Registri | Registri | Registri |
| Stack    | Stack    | Stack    |
| ...      | ...      | ...      |
+----------+----------+----------+
```

I thread, quindi, condividono memoria, codice e risorse tra di loro. Il loro vantaggio è, oltre la condivisione della memoria, l'assenza della necessità di duplicazione delle strutture necessarie per la virtualizzazione delle risorse necessarie per generare un nuovo processo. La creazione e la distruzione di thread è, quindi, molto più economica dell'equivalente con i processi.

Ad ogni thread è associato uno stato:

- Esecuzione: in esecuzione sulla CPU (1 thread per CPU, noi ne assumeremo 1)
- Attesa: bloccato da operazioni di attesa (IO, `wait()` o altri eventi esterni)
- Pronto: in attesa di essere eseguito sulla CPU

Ad ogni thread è anche associato un Thread ID univoco all'interno del processo. La terminazione del processo implica la terminazione di tutti i suoi thread, indipendentemente dal loro stato. Sarà compito del programmatore far sì che un processo termini dopo che tutti i suoi thread hanno terminato l'esecuzione.

L'ordine di esecuzione dei thread non è definito.

Utilizzeremo le API per il threading POSIX in C. La libreria di gestione che useremo è la NPTL (Native POSIX Thread Library) fornita da Linux. Nel modello di esecuzione che useremo i thread sono tutti visibili al kernel del sistema operativo e ne gestisce il tempo di esecuzione.

Differenza tra concorrenza e parallelismo

Diamo un po' di definizioni:

- Sequenziali - Due attività si dicono sequenziali se è possibile stabilire che una attività è sempre svolta dopo un'altra. Lo indicheremo con $A < B \vee B < A$
- Concorrenti - Due attività sono concorrenti se non sono sono sequenziali.
- Parallele - Due attività di dicono parallele se non è possibile stabilire un ordine di esecuzione tra le istruzioni delle due.

Se il numero di CPU è 1, la concorrenza e il parallelismo vengono a coincidere.

Chiamate di sistema per la gestione dei thread (POSIX)

1. Creazione: `int pthead_create(pthread_t*, pthread_attr_t*, void* (*)(void*), void*)`

   Viene creato un thread con thread id puntato dall'argomento passato. Questo thread eseguirà la funzione passata come argomento e riceverà come argomenti l'array passato. La struttura `pthread_attr_t` contiene degli attributi del thread; se come puntatore a questa struttura viene passato NULL vengono usati gli attributi standard. La funzione ritorna 0 se tutto va a buon fine e un codice di errore altrimenti.

2. Attesa: `int pthread_join(pthread_t*, int*)`

   Il thread che la invoca si pone in attesa della terminazione di un altro thread con thread id passato in argomento. Il valore di ritorno del thread che si sta attendendo viene salvato nell'interno passato. La funzione ritorna 0 se va a buon fine e un codice di errore altrimenti.

3. Terminazione: `void pthread_exit(int)`

   Termina l'esecuzione del thread passando un codice al thread che si è messo in attesa tramite `pthread_join()`.

```c
#include <stdio.h>
#include <pthread.h>

void *tf1(void *tid) {
  int conta = 0;
  conta++;
  printf("Sono thread n %d; conta = %d\n", (int)tid, conta);
  return NULL;
}

int main(void) {
  pthread_t tid1, tid2;

  pthread_create(&tid1, NULL, &tf1, (void*) 1);
  pthread_create(&tid2, NULL, &tf1, (void*) 2);

  pthread_join(tid1, NULL);
```

```
  pthread_join(tid2, NULL);

  return 0;
}
```

Sincronizzazione di thread

La sincronizzazione dei thread serve a coordinare l'accesso alle risorse condivise, coordinare l'allocazione delle risorse o consentire una esecuzione deterministica.

I thread possono sincronizzarsi o su base temporale tramite message-passing o tramite altri metodi non su base temporale. Per creare il secondo tipo di sincronizzazione è necessario garantire serializzazione e la mutua esclusione.

I mutex Analizziamo un caso di conflitto su variabili globali: lo schema produttore-consumatore. Risulteranno in un conflitto tutte le operazioni del tipo `x = f(x)`. Per risolvere questo tipo di conflitti basta definire una serie di operazioni atomiche indivisibili. Le uniche operazioni veramente atomiche, però, sono quelle del linguaggio macchina e non quelle di un linguaggio di alto livello come il C. Sarà quindi necessario definire dei metodi per rendere delle operazioni di alto livello atomiche: le sezioni critiche.

Le sezioni critiche vengono create tramite dei costrutti messi a disposizione dalla libreria di threading: i mutex. Un mutex si comportano come un lucchetto: esso può essere attivato da un thread (prende possesso del mutex) e blocca l'esecuzione della sezione critica finché il thread che lo aveva bloccato non lo sblocca (lo rilascia). Tutti i thread che incontrano un mutex bloccato verranno messi in attesa.

Ecco le funzioni di libreria per la gestione dei mutex:

1. Creazione: `int pthread_mutex_init(pthread_mutex_t*, pthread_mutexattr_t*)`
2. Bloccaggio del mutex: `int pthread_mutex_lock(pthread_mutex_t*)`
3. Sbloccaggio del mutex: `int pthread_mutex_unlock(pthread_mutex_t*)`

Realizzazione di un mutex Come possiamo implementare un mutex? Usiamo un caso speciale di un costrutto di sincronizzazione più generico: il semaforo binario. Consideriamo le seguente implementazione assai semplificata.

```
typedef int mutex;

void mutex_init(mutex *m) {
  *m = 0;
}

void mutex_lock(mutex *m) {
  while (*m == 1)
    ;
  *m = 1;
}

void mutex_unlock(mutex *m) {
  *m = 0;
}
```

Rimane ancora il problema della non atomicità del ciclo in `mutex_lock()`. Proviamo a usare 2 variabili. Per semplificare l'implementazione consideriamo solo 2 thread.

```
typedef struct {
  int blocca1; // thread 1 vuole entrare nella sezione critica
  int blocca2; // thread 2 vuole entrare nella sezione critica
} mutex;

void mutex_init(mutex *m) {
  m->blocca1 = 0;
  m->blocca2 = 0;
}
```

```
void mutex_lock(mutex *m) {
  if (is_thead_id(1)) {
    m->blocca1 = 1; // 1
    while (m->blocca2 == 1) // 2
      ;
  }
  if (is_thread_id(2)) {
    m->blocca2 = 1; // 3
    while (m->blocca1 == 1) // 4
      ;
  }
}

void mutex_unlock(mutex *m) {
  if (is_thread_id(1))
    m->blocca1 = 0;
  if (is_thread_id(2))
    m->blocca2 = 0;
}
```

Questa implementazione garantisce la mutua esclusione, ma crea un altro problema: nessun thread riesce ad accedere alla sezione critica. Infatti se la funzione `mutex_lock()` nel primo thread viene interrotta durante l'istruzione 1 e il secondo thread riprende eseguendo 3 e viene anch'esso interrotto cadremo in stallo (deadlock). Proviamo con 3.

```
typedef struct {
  int blocca1;  // thread 1 vuole entrare nella sezione critica
  int blocca2;  // thread 2 vuole entrare nella sezione critica
  int favorito; // garantisce che un thread possa sempre progredire
} mutex;

void mutex_init(mutex *m) {
  m->blocca1 = 0;
  m->blocca2 = 0;
  m->favorito = 1;
}

void mutex_lock(mutex *m) {
  if (is_thread_id(1)) {
    m->blocca1 = 1;
    m->favorito = 2;
    while (m->blocca1 == 1 & m->favorito == 2)
      ;
  }
  if (is_thread_id(2)) {
    m->blocca2 = 1;
    m->favorito = 1;
    while (m->blocca2 == 1 & m->favorito == 1)
      ;
  }
}

void mutex_unlock(mutex *m) {
  if (is_thread_id(1))
    m->blocca1 = 0;
  if (is_thread_id(2))
    m->blocca2 = 0;
}
```

Con l'utilizzo di 3 variabili siamo riusciti a rendere impossibile il deadlock.

I semafori I semafori generalizzano il sistema di bloccaggio introdotto dai mutex. Un semaforo non è altro che una variabile intera sulla quale si può operare nel modo seguente:

- inizializzata con un valore intero
- incrementata di 1
- decrementata di 1

Per standard POSIX un semaforo non può assumere valori negativi.

Nella libreria `pthread` (header `<semaphore.h>`) esiste il tipo `sem_t` e sono dichiarate le seguenti funzioni:

- `int sem_init(sem_t *, int, unsigned int)`

  Inizializza il semaforo a un valore intero senza segno. Possono essere passati dei flag (secondo intero), noi lo considereremo sempre 0.

- `int sem_wait(sem_t*)`

  Decrementa il semaforo; se esso è già 0 essa blocca il thread finché un altro thread non incrementa il semaforo.

- `int sem_post(sem_t*)`

  Incrementa il semaforo

- `int sem_getvalue(sem_t*, int*)`

  Restituisce il valore corrente del semaforo.

Il valore del semaforo rappresenta, quindi, il numero di thread che possono concorrentemente accedere a una risorsa. L'uso di un semaforo più semplice di un semaforo è quello di segnalare ad un altro thread che è avvenuto un evento. Possiamo così risolvere il problema della serializzazione di eventi.

Utilizzi dei semafori

- Rendezvous:

```
sem_t a_ok, b_ok;

void *tf1(void *a) {
  a1();

  sem_wait(&b_ok);
  sem_post(&a_ok);

  a2();
  return NULL;
}

void *tf2(void *a) {
  b1();

  sem_post(&b_ok);
  sem_wait(&a_ok);

  b2();
  return NULL;
}

int main(void) {
  ....
  sem_init(&a_ok, 0, 0);
  sem_init(&b_ok, 0, 0);
  ...
}
```

- Implementazione di un mutex:

```
sem_t mutex;
int i = 0;

void *tf(void*a) {
  sem_wait(&mutex)
  i += 1;
  sem_post(&mutex);
  return NULL;
}

int main(void) {
  ...
  sem_init(&mutex, 0, 1);
  ...
}
```

Se inizializzo il semaforo a `n >= 1` possiamo garantire l'accesso concorrente a `n` thread.

- Barriera - tutti i thread si sincronizzano in un punto

```
#define T ...

sem_t barriera;
mutex_t mutex;
int count; // quanti thread sono arrivati alla barriera

void *tf(void *a) {
  ...
  pthread_mutex_lock(&mutex)
  count++;
  pthread_mutex_unlock(&mutex);
  if (count == T)
    sem_post(&barriera);
  else {
    sem_wait(&barriera);
    sem_post(&barriera);
  }
  ...
}

int main(void) {
  ...
  sem_init(&barriera, 0, 0);
  ...
}
```

- Problema dei 5 filosofi

```
sem_t forchette[5];

void mangia(void) { ... }

void *filosofo(void *arg) {
  int index = (int) arg;
  if (index >= 0 && index <=3) {
    sem_wait(&forchette[index]);
    sem_wait(&forchette[index + 1]);
    mangia();
    sem_post(&forchette[index]);
    sem_post(&forchette[index + 1]);
  } else {
    sem_wait(&forchette[0]);
```

```
    sem_wait(&forchette[index]);
    mangia();
    sem_post(&forchette[0]);
    sem_post(&forchette[index]);
  }
  return NULL;
}

int main(void) {
  ...
  for (int i = 0; i < 5; i++)
    sem_init(&forchette[i], 0, 1);
  ...
}
```