Sistemi Operativi[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

a.a. 2016/17

Lezione XVI: Concorrenza

Concorrenza

- Concorrenza: *run together* & *compete*
- Un processo non è piú un programma in esecuzione che può essere considerato in isolamento
- Non determinismo: il sistema nel suo complesso ($P_1 + P_2$+Scheduler) rimane deterministico, ma se si ignora lo scheduler le esecuzioni di P_1 e P_2 possono combinarsi in molto modi, con output del tutto differenti
- Sincronizzazione: si usano meccanismi (Peterson, TSL, semafori, monitor, message passing, . . .) per imporre la combinazione voluta di P_1 e P_2

Processi (senza mem. condivisa)


```
int shared[2] = {0 , 0};
/* int clone(int (*fn)(void *),
 *           void *child_stack,
 *           int flags,
 *           void *arg);
 * crea una copia del chiamante (con le caratteristiche
 * specificate da flags) e lo esegue partendo da fn */
if (clone(run, /* il nuovo
                * processo esegue run(shared), vedi quarto
                * parametro */
          malloc(4096)+4096,  /* lo stack del nuovo processo
                               *  (cresce verso il basso) */
          SIGCHLD, // in questo caso la clone \`e analoga alla fork
          shared) < 0){
        perror("Errore nella creazione");exit(1);
}
if (clone(run, malloc(4096)+4096,  SIGCHLD, shared) < 0){
        perror("Errore nella creazione");exit(1);
}
```



```
/* Isolati: ciascuno dei figli esegue 10 volte. */
```

Thread (con mem. condivisa)


```
int shared[2] = {0 , 0};
/* int clone(int (*fn)(void *),
 *           void *child_stack,
 *           int flags,
 *           void *arg);
 * crea una copia del chiamante (con le caratteristiche
 * specificate da flags) e lo esegue partendo da fn */
if (clone(run, /* il nuovo
                * processo esegue run(shared), vedi quarto
                * parametro */
          malloc(4096)+4096,  /* lo stack del nuovo processo
                               *  (cresce verso il basso) */
          CLONE_VM | SIGCHLD, // (virtual) memory condivisa
          shared) < 0){
        perror("Errore nella creazione");exit(1);
}

if (clone(run, malloc(4096)+4096,  CLONE_VM | SIGCHLD, shared)    ){
        perror("Errore nella creazione");exit(1);
}

/* Memoria condivisa: i due figli nell'insieme eseguono 10 o
 * 11 volte; \`e possibile una corsa critica. Il padre
```


Thread (mutua esclusione con Peterson)


```
void enter_section(int process, int* turn, int* interested)
{
        int other = 1 - process;
        interested[process] = 1;
        *turn = process;
        while  (*turn == process && interested[other]){
                printf("Busy waiting di %d\n", process);
        }
}

void leave_section(int process, int* interested)
{
        interested[process] = 0;
}
int run(const int p, void* s)
{
        int* shared = (int*)s; // alias per comodit\`a
        while (enter_section(p, &shared[1], &shared[2]), shared[0] < 10 ) {
                sleep(1);
                printf("Processo figlio (%d). s = %d\n",
                       getpid(), shared[0]);
                if (1(shared[0] < 10)){
```


Performance


```
$ time ./threads-peterson > /tmp/output
real        0m11.091s
user        0m0.000s
sys         0m0.089s
$ grep -c "Busy waiting" /tmp/output
92314477
```


Semafori

*section

Una variabile intera condivisa controllata da system call che interagiscono con lo scheduler:

- **down** decrementa, bloccando il chiamante se il valore corrente è 0; `sem_wait`
- **up** incrementa, rendendo ready altri processi precedentemente bloccati se il valore corrente è maggiore di 0; `sem_post`

Meccanismo di base


```
statement1                    statement2
 sem_init(&ss, 0, 0); // init a 0

    statement1;                  down(&semaforo);
    up(&semaforo);               statement2;
```


Meccanismo di base

1 deve eseguire prima di B, A deve eseguire prima di 2. Come fareste?

```
statement1;                  statementA;
statement2                   statementB
```


Mutua esclusione con semafori


```
void down(sem_t *s){
        if (sem_wait(s) < 0){
                perror("Errore semaforo (down)");
                exit(1);
        }
}

void up(sem_t *s){
        if (sem_post(s) < 0){
                perror("Errore semaforo (up)");
                exit(1);
        }
}
```


Mutua esclusione con semafori


```
int shared = 0;
pthread_t p1, p2;
sem_t ss;

void* run(void* s){
        while (down(&ss),
               shared < 10) {
                sleep(1);
                printf("Processo thread (%p). s = %d\n",
                       pthread_self(), shared);
                if (!(shared < 10)){
                        printf("Corsa critica!!!!\n");
                        abort();
                }
                shared += 1;
                up(&ss);
                pthread_yield();
        }
        up(&ss);
        return NULL;
}
```

POSIX threads

Lo standard POSIX specifica una serie di API per la programmazione concorrente chiamate pthread (su Linux saranno implementate tramite `clone`).

- "multiparadigma": ci concentriamo sul modello a monitor, con mutex e condition variable. (Nota: i monitor sono costrutti specifici nel linguaggio, pthread usa il C, quindi p.es. l'incapsulamento dei dati va curato a mano)

```
pthread_create(thread,attr,start_routine,arg)
pthread_exit (status)
pthread_join (threadid,status)
pthread_mutex_init (mutex,attr)
pthread_mutex_lock (mutex)
pthread_mutex_unlock (mutex)
pthread_cond_init (condition,attr)
pthread_cond_wait (condition,mutex)
pthread_cond_signal (condition)
pthread_cond_broadcast (condition)
```


Il pattern di base

Tralasciando le inizializzazioni dei puntatori mutex e condition:

```
// T1
pthread_mutex_lock(mutex); // Acquisire il lock
while (!predicate) // fintantoch\'e la condizione \`e falsa
   pthread_cond_wait(condition, mutex); // block
pthread_mutex_unlock(mutex); // rilasciare il lock

// T2
// qualche thread rende vero il predicato cos\'{\i}
pthread_mutex_lock(mutex); // Acquisire il lock
predicate = TRUE;
pthread_cond_signal(condition); // e lo segnala
pthread_mutex_unlock(mutex); // rilasciare il lock
```


Perché il mutex?

Il mutex è necessario per sincronizzare il controllo della condizione, altrimenti

```
// T1                                  // T2
pthread_mutex_lock(mutex);             //
while (!predicate)                     //
  //                                   predicate = TRUE;
  //                                   pthread_cond_signal(condition);
  pthread_cond_wait(condition, mutex);
pthread_mutex_unlock(mutex);           //
```


Produttore e consumatore

- Il produttore smette di produrre se il buffer è pieno e deve essere avvisato quando non lo è piú (può ricominciare a produrre)
- Il consumatore smette di consumare se il buffer è vuoto e deve essere avvisato quando non lo è piú (può ricominciare a consumare)
- 2 condition variable: buffer pieno e buffer vuoto (ne servono due perché pieno $\neq \neg$ vuoto)

Produttore e consumatore


```
#define N 10
char* buffer[N];
int count = 0;


pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;

void b_insert(char* o){
  pthread_mutex_lock(&lock);

  while (count == N) pthread_cond_wait(&full, &lock);
  printf("Inserimento in buffer con %d\n", count);
  buffer[count++] = o;
  if (count == 1) pthread_cond_signal(&empty);

  pthread_mutex_unlock(&lock);
}
```



```
/* passaggio per indirizzo per evitare di fare la return fuori dai lock */
```

Produttore e consumatore


```
void b_remove(char** result){
  pthread_mutex_lock(&lock);

  while (count == 0) pthread_cond_wait(&empty, &lock);
  printf("Rimozione in buffer con %d\n", count);
  *result = buffer[--count];
  if (count == N-1) pthread_cond_signal(&full);

  pthread_mutex_unlock(&lock);
}
```


Produttore e consumatore


```
void* producer(void* nonusato){
    printf("Esecuzione del produttore\n");
  while (1){
    char* o = (char*)malloc(sizeof(char));
    printf("Ho prodotto %p\n", o);
    b_insert(o);
  }
}

void* consumer(void* nonusato){
    printf("Esecuzione del consumatore\n");
  while (1){
    char* o;
    b_remove(&o);
    free(o);
    printf("Ho consumato %p\n", o);
  }
}
```


Produttore e consumatore


```
int main(void){
    pthread_t p1, p2;

    pthread_create(&p1, NULL, consumer, NULL);
    pthread_create(&p2, NULL, producer, NULL);

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

  return 0;
}
```