Sistemi Operativi[1]

Mattia Monga

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

a.a. 2010/11

Lezione XX: Concorrenza

Processi (senza mem. condivisa)

```
1   int shared[2] = {0 , 0};
2
3   /* int clone(int (*fn)(void *),
4    * void *child_stack,
5    * int flags,
6    * void *arg);
7    * crea una copia del chiamante (con le caratteristiche
8    * specificate da flags) e lo esegue partendo da fn */
9   if (clone(run, /* il nuovo
10                 * processo esegue run(shared), vedi quarto
11                 * parametro */
12            malloc(4096)+4096, /* lo stack del nuovo processo
13                                * (cresce verso il basso!) */
14            SIGCHLD, /* in questo caso la clone e' analoga alla fork */
15            shared) < 0){
16          perror("Errore nella creazione");
17          exit(1);
18  }
19
20  if (clone(run, malloc(4096)+4096, SIGCHLD, shared) < 0){
21          perror("Errore nella creazione");
```


Thread (con mem. condivisa)

```
1   int shared[2] = {0 , 0};
2
3   /* int clone(int (*fn)(void *),
4    * void *child_stack,
5    * int flags,
6    * void *arg);
7    * crea una copia del chiamante (con le caratteristiche
8    * specificate da flags) e lo esegue partendo da fn */
9   if (clone(run, /* il nuovo
10                 * processo esegue run(shared), vedi quarto
11                 * parametro */
12            malloc(4096)+4096, /* lo stack del nuovo processo
13                                * (cresce verso il basso!) */
14            CLONE_VM | SIGCHLD, /* la (virtual) memory e' condivisa */
15            shared) < 0){
16          perror("Errore nella creazione");
17          exit(1);
18  }
19
20  if (clone(run, malloc(4096)+4096, CLONE_VM | SIGCHLD, shared) < 0){
21          perror("Errore nella creazione");
```

Thread (mutua esclusione con Peterson)


```
1
2  void enter_section(int process, int* turn, int* interested)
3  {
4          int other = 1 - process;
5          interested[process] = 1;
6          *turn = process;
7          while (*turn == process && interested[other]){
8                  printf("Busy waiting di %d\n", process);
9          }
10 }
11
12 void leave_section(int process, int* interested)
13 {
14         interested[process] = 0;
15 }
```

```
1  int run(const int p, void* s)
2  {
3          int* shared = (int*)s; /* alias per comodita' */
4          while (enter_section(p, &shared[1], &shared[2]),
5                  shared[0] < 10) {
6                   sleep(1);
7                   printf("Processo figlio (%d). s = %d\n",
```


Thread (mutua esclusione con TSL)


```
1  void enter_section(int *s); /* in enter.asm */
2
3  void leave_section(int *s){
4          *s = 0;
5  }
6
7  int run(const int p, void* s)
8  {
9          int* shared = (int*)s; /* alias per comodita' */
10         while (enter_section(&shared[1]),
11                 shared[0] < 10) {
12                  sleep(100);
13                  printf("Processo figlio (%d). s = %d\n",
14                          getpid(), shared[0]);
15                  if (!(shared[0] < 10)){
16                          printf("Corsa critica!!!!\n");
17                          abort();
18                  }
19                  shared[0] += 1;
20                  leave_section(&shared[1]);
21                  sched_yield();
22         }
```


POSIX threads

Gli esempi visti finora usano clone per creare i thread, che però è una system call specifica di Linux. 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)

```
1  pthread_create(thread,attr,start_routine,arg)
2  pthread_exit (status)
3  pthread_join (threadid,status)
4  pthread_mutex_init (mutex,attr)
5  pthread_mutex_lock (mutex)
6  pthread_mutex_unlock (mutex)
7  pthread_cond_init (condition,attr)
8  pthread_cond_wait (condition,mutex)
9  pthread_cond_signal (condition)
10 pthread_cond_broadcast (condition)
```


Il pattern di base

Tralasciando le inizializzazioni dei puntatori mutex e condition:

```
1    // T1
2    pthread_mutex_lock(mutex); // Acquisire il lock
3    while (!predicate) // fintantoch\'e la codizione \'e falsa
4        pthread_cond_wait(condition, mutex); // block
5    pthread_mutex_unlock(mutex); // rilasciare il lock
6
7    // T2
8    // qualche thread rende vero il predicato cos\'{\i}
9    pthread_mutex_lock(mutex); // Acquisire il lock
10   predicate = TRUE;
11   pthread_cond_signal(condition); // e lo segnala
12   pthread_mutex_unlock(mutex); // rilasciare il lock
```

Perché il mutex?

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

```
1 // T1
2 pthread_mutex_lock(mutex);
3 while (!predicate)
4   //
5   //
6   pthread_cond_wait(condition, mutex);
7 pthread_mutex_unlock(mutex);
```

```
1   // T2
2   //
3   //
4   predicate = TRUE;
5   pthread_cond_signal(condition);
6   //
7   //
```


Incrementer e watcher

- Due "incrementatori" aumentano un contatore condiviso
- Un "guardiano" aspetta che il contatore raggiunga un certo valore
- 1 condition variable: permette di attendere che il contatore *superi* una certa soglia (12)
  - In questo caso **if** e **while** sono equivalenti perché una volta superata la soglia, il predicato "maggiore della soglia" rimane vero.

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 $\neq$ /vuoto)