



Sistemi Operativi¹

Mattia Monga

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

a.a. 2012/13

¹ © 2011–13 M. Monga. Creative Commons Attribuzione-Condividi allo stesso modo 3.0 Italia License. <http://creativecommons.org/licenses/by-sa/3.0/it/>. Immagini tratte da [?] e da Wikipedia.



Lezione XIX: Concorrenza

Concorrenza



- **Concorrenza:** *run together & compete*
- Un processo non è più un programma in esecuzione che può essere considerato in isolamento
- Non determinismo: il sistema nel suo complesso ($P_1 + P_2 + \text{Scheduler}$) rimane deterministico, ma se si ignora lo scheduler le esecuzioni di P_1 e P_2 possono combinarsi in molti 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)



```

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

```

Thread (con mem. condivisa)



```
1  int shared[2] = {0, 0};
2  /* int clone(int (*fn)(void *),
3   * void *child_stack,
4   * int flags,
5   * void *arg);
6   * crea una copia del chiamante (con le caratteristiche
7   * specificate da flags) e lo esegue partendo da fn */
8  if (clone(run, /* il nuovo
9   * processo esegue run(shared), vedi quarto
10  * parametro */
11  malloc(4096)+4096, /* lo stack del nuovo processo
12  * (cresce verso il basso!) */
13  CLONE_VM | SIGCHLD, // (virtual) memory condivisa
14  shared) < 0){
15  perror("Errore nella creazione");exit(1);
16  }
17
18  if (clone(run, malloc(4096)+4096, CLONE_VM | SIGCHLD, shared) < 0){
19  perror("Errore nella creazione");exit(1);
20  }
21
22  /* Memoria condivisa: i due figli nell'insieme eseguono 10 o
23  11 volte: è possibile una corsa critica. Il padre
```

Sistemi
Operativi

Bruschi
Monga

Concorrenza

Semafori
Sincronizzazione
con monitor
pthreads

362

Performance



```
1 $ time ./threads-peterson > /tmp/output
2 real 0m11.091s
3 user 0m0.000s
4 sys 0m0.089s
5 $ grep -c "Busy waiting" /tmp/output
6 92314477
```

Sistemi
Operativi

Bruschi
Monga

Concorrenza

Semafori
Sincronizzazione
con monitor
pthreads

364

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 }
16
17 int run(const int p, void* s)
18 {
19     int* shared = (int*)s; // alias per comodità
20     while (enter_section(p, &shared[1], &shared[2]), shared[0] < 10){
21         sleep(1);
22         printf("Processo figlio (%d). s = %d\n",
23             getpid(), shared[0]);
24     }
25 }
```

Sistemi
Operativi

Bruschi
Monga

Concorrenza

Semafori
Sincronizzazione
con monitor
pthreads

Thread (mutua esclusione con TSL)



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

Sistemi
Operativi

Bruschi
Monga

Concorrenza

Semafori
Sincronizzazione
con monitor
pthreads

365



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`

366



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)
```

368



```
1 void down(sem_t *s){
2     if (sem_wait(s) < 0){
3         perror("Errore semaforo (down)");
4         exit(1);
5     }
6 }

1     perror("Errore semaforo (up)");
2     exit(1);
3 }
4 }
5
6 int shared = 0;
7 pthread_t p1, p2;

1     perror("Errore semaforo");
2     exit(1);
3 }
4
5 pthread_create(&p1, NULL, run, NULL);
6 pthread_create(&p2, NULL, run, NULL);
7
8 pthread_join(p1, NULL);
```

367



Tralasciando le inizializzazioni dei puntatori mutex e condition:

```
1 // T1
2 pthread_mutex_lock(mutex); // Acquisire il lock
3 while (!predicate) // fintantoché la condizione è 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 così
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
```

369

Perché il mutex?



Sistemi Operativi

Bruschi Monga

Concorrenza
Semafori
Sincronizzazione con monitor
pthreads

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

```
1 // T1                1 // T2
2 pthread_mutex_lock(mutex);  2 //
3 while (!predicate)      3 //
4 //                    4 predicate = TRUE;
5 //                    5 pthread_cond_signal(condition);
6 pthread_cond_wait(condition, mutex);  6 //
7 pthread_mutex_unlock(mutex);  7 //
```

370

Incrementer e watcher



Sistemi Operativi

Bruschi Monga

Concorrenza
Semafori
Sincronizzazione con monitor
pthreads

- 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.

371

Produttore e consumatore



Sistemi Operativi

Bruschi Monga

Concorrenza
Semafori
Sincronizzazione con monitor
pthreads

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

372