



Sistemi Operativi¹

Mattia Monga

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

a.a. 2009/10

¹ © 2010 M. Monga. Creative Commons Attribuzione-Condividi allo stesso modo 2.5 Italia License.
<http://creativecommons.org/licenses/by-sa/2.5/it/>. Immagini tratte da [?] e da Wikipedia.



Lezione XI: Concorrenza e strumenti di sviluppo

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)



DICo

```
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 comodita' */
20     while (enter_section(p, &shared[1], &shared[2]),
21            shared[0] < 10) {
22         sleep(1);
23         printf("Processo figlio (%d). s = %d\n",
```

225

Sistemi Operativi

Bruschi Monga

Concorrenza
Concorrenza in Java

Concorrenza in Java



DICo

- In Java è possibile definire **oggetti attivi**, ossia con un *thread of control* parallelo a quello del main
- Il modo più diretto è derivare una classe da `java.lang.Thread`
- Il thread of control deve essere poi specificato ridefinendo il metodo **public void run()**, che di default non fa nulla.
- il thread of control di un oggetto attivo può essere attivato con il metodo `start()` che ha l'effetto di rendere il thread pronto per essere, prima o poi, schedato
- I thread condividono la memoria secondo le normali regole di visibilità di Java.

227

Sistemi Operativi

Bruschi Monga

Concorrenza
Concorrenza in Java

Thread (mutua esclusione con TSL)



DICo

```
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    }
23 }
24
25 return 0;
```

226

Sistemi Operativi

Bruschi Monga

Concorrenza
Concorrenza in Java

Classi attive



DICo

```
1 class ClasseAttiva extends Thread{
2     public void run(){
3         while (true) {
4             try {
5                 Thread.sleep(100);
6             }
7             catch (InterruptedException e){
8                 System.err.println(e);
9             }
10            System.out.println(this.getName());
11        }
12    }
13 }
14
15 public class Basic {
16     public static final void main(final String[] args) {
17         ClasseAttiva o1 = new ClasseAttiva();
18         ClasseAttiva o2 = new ClasseAttiva();
19         o1.start();
20         o2.start();
21         while (true){
```

228

Sistemi Operativi

Bruschi Monga

Concorrenza
Concorrenza in Java



- La mutua esclusione e la sincronizzazione possono essere ottenute tramite *blocchi synchronized*

```
1 private int sharedX; // condiviso fra piu' thread
2 private Object lock = new Object() // condiviso fra piu' thread
3 synchronized(lock){
4     // usa sharedX
5 }
```

- L'esecuzione delle istruzioni del blocco può iniziare solo se il lock lock non è posseduto da un altro thread.
- Il lock lock viene quindi acquisito dal thread in esecuzione e rilasciato al raggiungimento del termine del blocco.
- È possibile definire metodi **synchronized** che implicitamente usano **this** come lock.

229



- `lock.wait()` può essere eseguita solo se si possiede il lock
 - Ha l'effetto di portare il thread in stato di blocked e di **rilasciare il lock**

```
1 synchronized(lock){
2     while (condizione(sharedX)){
3         try{
4             lock.wait(); // rilascia lock
5         } catch (InterruptedException e){
6             e.printStackTrace(System.err);
7         }
8         // quando la "condizione" e' falsa, proseguo (tenendo il lock)
9     }
10 }
```

230



- `lock.notifyAll()` può essere eseguita solo se si possiede il lock
 - Ha l'effetto di portare il thread in stato di ready tutti i thread in stato di blocked in seguito a una `lock.wait()`

```
1 synchronized(lock){
2     set(sharedX);
3     lock.notifyAll(); // notifica tutti i thread in wait su "lock"
4                     // che lo stato della
5                     // memoria condivisa controllato da "lock"
6                     // e' cambiato
7 }
```

231



- Le primitive di Java possono essere direttamente usate per programmare mutua esclusione e sincronizzazioni anche complesse
- Nelle ultime versioni sono presenti anche librerie che simulano altri modelli di concorrenza
 - Semafori con `java.util.concurrent.Semaphore`
 - Monitor con `java.util.concurrent.locks.*`

232



- `s = new Semaphore(1);` crea un semaforo binario
- `s = new Semaphore(n);` crea un semaforo generalizzato
- `s.acquire()` è la `down()`
- `s.release()` è la `up()`



Questa libreria è molto simile alle primitive di Java pure, ma i lock sono oggetti espliciti a cui si associano delle *condition variable*

```
1 Lock l = new ReentrantLock();
2 Condition cond = lock.newCondition();
3 l.lock();
4 try {
5     cond.await();
6     cond.signal();
7 } finally {
8     l.unlock();
9 }
```