



Sistemi Operativi¹

Mattia Monga

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

a.a. 2008/09

¹ © 2009 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)

```

29  int shared[2] = {0, 0};
30
31  /* int clone(int (*fn)(void *),
32   * void *child_stack,
33   * int flags,
34   * void *arg);
35   * crea una copia del chiamante (con le caratteristiche
36   * specificate da flags) e lo esegue partendo da fn */
37  if (clone(run, /* il nuovo
38           * processo esegue run(shared), vedi quarto
39           * parametro */
40         malloc(4096)+4096, /* lo stack del nuovo processo
41           * (cresce verso il basso!) */
42         SIGCHLD, /* in questo caso la clone e' analoga alla fork */
43         shared) < 0){
44     perror("Errore nella creazione");
45     exit(1);
46 }
47
48 if (clone(run, malloc(4096)+4096, SIGCHLD, shared) < 0){
49     perror("Errore nella creazione");

```



Thread (con mem. condivisa)

```

29  int shared[2] = {0, 0};
30
31  /* int clone(int (*fn)(void *),
32   * void *child_stack,
33   * int flags,
34   * void *arg);
35   * crea una copia del chiamante (con le caratteristiche
36   * specificate da flags) e lo esegue partendo da fn */
37  if (clone(run, /* il nuovo
38           * processo esegue run(shared), vedi quarto
39           * parametro */
40         malloc(4096)+4096, /* lo stack del nuovo processo
41           * (cresce verso il basso!) */
42         CLONE_VM | SIGCHLD, /* la (virtual) memory e' condivisa */
43         shared) < 0){
44     perror("Errore nella creazione");
45     exit(1);
46 }
47
48 if (clone(run, malloc(4096)+4096, CLONE_VM | SIGCHLD, shared) < 0){
49     perror("Errore nella creazione");

```

Thread (mutua esclusione con Peterson)



DICo

```
10
11 void enter_section(int process, int* turn, int* interested)
12 {
13     int other = 1 - process;
14     interested[process] = 1;
15     *turn = process;
16     while (*turn == process && interested[other]){
17         printf("Busy waiting di %d\n", process);
18     }
19 }
20
21 void leave_section(int process, int* interested)
22 {
23     interested[process] = 0;
24 }
25
26 int run(const int p, void* s)
27 {
28     int* shared = (int*)s; /* alias per comodita' */
29     while (enter_section(p, &shared[1], &shared[2]),
30           shared[0] < 10) {
31         sleep(1);
32         printf("Processo figlio (%d). s = %d\n",
```

220

Sistemi Operativi

Bruschi
Martignoni
Monga

Concorrenza

Concorrenza in Java

MINIX

Concorrenza in Java



DICo

- In Java è possibile definire *oggetti attivi*, ossia con un *thread of control* parallelo a quello del main
- Il modo piú 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, schedulato
- I thread condividono la memoria secondo le normali regole di visibilità di Java.

222

Sistemi Operativi

Bruschi
Martignoni
Monga

Concorrenza

Concorrenza in Java

MINIX

Thread (mutua esclusione con TSL)



DICo

```
11 void enter_section(int *s); /* in enter.asm */
12
13 void leave_section(int *s){
14     *s = 0;
15 }
16
17 int run(const int p, void* s)
18 {
19     int* shared = (int*)s; /* alias per comodita' */
20     while (enter_section(&shared[1]),
21           shared[0] < 10) {
22         sleep(rand() % 3);
23         printf("Processo figlio (%d). s = %d\n",
24               getpid(), shared[0]);
25         if (!(shared[0] < 10)){
26             printf("Corsa critica!!!!\n");
27             abort();
28         }
29         shared[0] += 1;
30         leave_section(&shared[1]);
31     }
32     return 0;
33 }
```

221

Sistemi Operativi

Bruschi
Martignoni
Monga

Concorrenza

Concorrenza in Java

MINIX

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){
```

223

Sistemi Operativi

Bruschi
Martignoni
Monga

Concorrenza

Concorrenza in Java

MINIX



- 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.

224



- `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 }
```

225



- `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 }
```

226



- 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.*`

227

Semafori



DICo

Sistemi Operativi

Bruschi
Martignoni
Monga

Concorrenza
Concorrenza in
Java

MINIX

- `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()`

228

Monitor



DICo

Sistemi Operativi

Bruschi
Martignoni
Monga

Concorrenza
Concorrenza in
Java

MINIX

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 }
```

229

Make



DICo

Sistemi Operativi

Bruschi
Martignoni
Monga

Concorrenza
Concorrenza in
Java

MINIX

Stuart Feldman, 1977 at Bell Labs.

Permette di specificare **dipendenze** fra processi di generazione.

Dipendenze: se cambia questo file, allora il processo di generazione deve essere ripetuto.

```
1 helloworld: helloworld.o
2     cc -o $@ $<
3
4 helloworld.o: helloworld.c
5     cc -c -o $@ $<
6
7 .PHONY: clean
8 clean:
9     rm helloworld.o helloworld
```

230

Concetti fondamentali



DICo

Sistemi Operativi

Bruschi
Martignoni
Monga

Concorrenza
Concorrenza in
Java

MINIX

Breakpoint

Un punto del programma in cui l'esecuzione deve essere bloccata, tipicamente per esaminare lo stato in quell'istante.

Stepping

Eseguire il programma *passo a passo*. La granularità del passo può arrivare fino all'istruzione macchina.

231



Lo stato del programma può essere analizzato come:

- **forma simbolica**: secondo i simboli definiti nel linguaggio di alto livello e conservati come *simboli di debugging*
- **memoria virtuale**: stream di byte suddiviso in segmenti
 - Text: contiene le istruzioni (spesso read only)
 - Initialized Data Segment: variabili globali inizializzate
 - Uninitialized Data Segment (bss): variabili globali non inizializzate
 - Stack: collezione di *stack frame* per le chiamate di procedura. Cresce verso il basso.
 - Heap: Strutture dati create dinamicamente. Cresce verso l'alto.



- `break ...`
- `run ...`
- `print ...`
- `next`
- `step`
- `backtrace`