

Sistemi Operativi (Laboratorio)

Lorenzo Martignoni

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano, Italia
lorenzo@security.dico.unimi.it

a.a. 2008/09

Lezione V: Concorrenza

Fork

fork.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int i, x;
    srand(getpid());
    x = fork();

    if (x < 0){
        perror("Errore nella fork:"); exit(1);
    } else {
        if (x != 0){
            printf("Processo padre (%d)\n", getpid());
            for (i = 0; i < 10; i++)
                printf("Padre\n"); sleep(rand() % 3);
        } else {
            printf("Processo figlio (%d)\n", getpid());
            for (i = 0; i < 15; i++)
                printf("Figlio\n"); sleep(rand() % 3);
        }
    }

    printf("Process terminato (%d)\n", getpid());
    return 0;
}
```

Exec

exec.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int i;
    int x = fork();

    if (x < 0){
        perror("Errore nella fork:"); exit(1);
    } else {
        if (x != 0){
            fprintf(stderr, "Processo padre (%d)\n", getpid());
        } else {
            fprintf(stderr, "Processo figlio (%d): %s", getpid(), argv[1]);
            for (i = 2; i < argc; i++)
                fprintf(stderr, "%s ", argv[i]);
            fprintf(stderr, "\n");
            execv(argv[1], argv + 1);
            perror("Errore nella exec:");
        }
    }

    fprintf(stderr, "Process padre terminato\n");
    return 0;
}
```

Wait

wait.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int i, j;
    int x = fork();

    if (x < 0){
        perror("Errore nella fork."); exit(1);
    } else {
        if (x != 0){
            fprintf(stderr, "Processo padre (%d)\n", getpid());
        } else {
            fprintf(stderr, "Processo figlio (%d): %s ", getpid(), argv[1]);
            for (i = 2; i < argc; i++)
                fprintf(stderr, "%s ", argv[i]);
            fprintf(stderr, "\n");
            execv(argv[1], argv + 1);
            perror("Errore nella exec.");
        }
    }

    j = wait(&i);
    fprintf(stderr, "Process padre & figlio terminati (%d/0 %d/%d)\n", getpid(), j, i);
    return 0;
}
```

Processi eseguiti in parallelo

parallelo.sh

```
function p1(){
    for i in $(seq 1 10); do
        echo $i
    done
}

function p2(){
    for i in $(seq 11 20); do
        echo $i
    done
}

p1 & p2 &
wait
```

Quale sarà l'output?

Processi (senza memoria condivisa) I

threads-isolated.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sched.h>

int run(void* s) {
    int* shared = (int*)s; /* alias per comodita' */
    while (shared[0] < 10) {
        sleep(1);
        printf("Processo figlio (%d). s = %d\n", getpid(), shared[0]);
        if (!(shared[0] < 10)) {
            printf("Corsa critica!!!!\n"); abort();
        }
        shared[0] += 1;
    }
    return 0;
}
```

Processi (senza memoria condivisa) II

threads-isolated.c

```
int main(void){
    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, /* 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);
    }
}
```

Processi (senza memoria condivisa) III

threads-isolated.c

```
/* Isolati: ciascuno dei figli esegue 10 volte. Per il padre
 * shared[0] e' sempre 0 */

while(1) {
    sleep(1);
    printf("Processo padre. s = %d\n", shared[0]);
}
return 0;
}
```

Thread (memoria condivisa) I

threads-shared.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sched.h>

int run(void* s) {
    int* shared = (int*)s; /* alias per comodita' */
    while (shared[0] < 10) {
        sleep(1);
        printf("Processo figlio (%d). s = %d\n", getpid(), shared[0]);
        if (!(shared[0] < 10)) {
            printf("Corsa critica!!!\n"); abort();
        }
        shared[0] += 1;
    }
    return 0;
}
```

Thread (memoria condivisa) II

threads-shared.c

```
int main(void){
    int shared[2] = {0, 0};

    if (clone(run, malloc(4096)+4096, CLONE_VM | SIGCHLD, shared) < 0) {
        perror("Errore nella creazione"); exit(1);
    }

    if (clone(run, malloc(4096)+4096, CLONE_VM | SIGCHLD, shared) < 0){
        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 condivide shared[0] con i figli */

    while(1) {
        sleep(1);
        printf("Processo padre. s = %d\n", shared[0]);
    }
    return 0;
}
```

Algoritmo di Peterson

```
flag[0] = 0
flag[1] = 0
turn = 0
```

```
P0:
    flag[0] = 1
    turn = 1
    while( flag[1] && turn == 1 )
        ; // do nothing
        // critical section
    ...
    // end of critical section
    flag[0] = 0
```

```
P1:
    flag[1] = 1
    turn = 0
    while( flag[0] && turn == 0 )
        ; // do nothing
        // critical section
    ...
    // end of critical section
    flag[1] = 0
```

Thread (mutua esclusione con Peterson) I

threads-peterson.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sched.h>

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;
}
```

Thread (mutua esclusione con Peterson) II

threads-peterson.c

```
int run(const int p, void* s) {
    int* shared = (int*) s;
    while (enter_section(p, &shared[1], &shared[2]), shared[0] < 10) {
        sleep(1);
        printf("Processo figlio (%d). s = %d\n", getpid(), shared[0]);
        if (!(shared[0] < 10)) {
            printf("Corsa critica!!!!\n"); abort();
        }
        shared[0] += 1;
        leave_section(p, &shared[2]);
    }
    return 0;
}

int run0(void* s) { return run(0, s); }
int run1(void* s) { return run(1, s); }
```

Thread (mutua esclusione con Peterson) III

threads-peterson.c

```
int main() {
    int shared[4] = {0, 0, 0, 0};

    if (clone(run, malloc(4096)+4096, CLONE_VM | SIGCHLD, shared) < 0) {
        perror("Errore nella creazione"); exit(1);
    }

    if (clone(run, malloc(4096)+4096, CLONE_VM | SIGCHLD, shared) < 0){
        perror("Errore nella creazione"); exit(1);
    }

    while(shared[0] < 10) {
        sleep(1);
        printf("Processo padre. s = %d %d %d %d\n", shared[0], shared[1],
            shared[2], shared[3]);
    }
    return 0;
}
```

Thread (mutua esclusione con TSL) I

threads-tsl.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sched.h>

void enter_section(int *s) {
    asm __volatile__ (" \
    spin: \
    lock bts $0x0, %0; \
    jc spin; : "=m" (*s));
}

void leave_section(int *s) {
    *s = 0;
}
```

Thread (mutua esclusione con TSL) II

threads-tsl.c

```
int run(const int p, void* s) {
    int* shared = (int*)s; /* alias per comodita' */
    while (enter_section(&shared[1]), shared[0] < 10) {
        sleep(1);
        printf("Processo figlio (%d). s = %d\n",
            getpid(), shared[0]);
        if (!(shared[0] < 10)) {
            printf("Corsa critica!!!\n"); abort();
        }
        shared[0] += 1;
        leave_section(&shared[1]);
        sched_yield();
    }
    return 0;
}

int run0(void* s){ return run(0, s); }
int run1(void* s){ return run(1, s); }
```

Thread (mutua esclusione con semaforo) I

threads-sem.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sched.h>
#include <semaphore.h>

sem_t S;

void enter_section() {
    if (sem_wait(&S) < 0)
        abort();
}

void leave_section() {
    if (sem_post(&S) < 0)
        abort();
}
```

Thread (mutua esclusione con semaforo) II

threads-sem.c

```
int run(const int p, void* s) {
    int* shared = (int*)s;
    while (enter_section(), *shared < 10) {
        printf("Processo figlio (%d). s = %d\n", getpid(), *shared);
        sleep(1);
        if (!( *shared < 10 )) {
            printf("Corsa critica!!!\n"); abort();
        }
        *shared += 1;
        leave_section();
        sched_yield();
    }
    return 0;
}
```

Thread (mutua esclusione con semaforo) III

threads-sem.c

```
int main(void){
    int shared = 0;

    if (sem_init(&S, 0, 1))
        abort();

    if (clone(run, malloc(4096)+4096, CLONE_VM | SIGCHLD, shared) < 0)
        abort();

    if (clone(run, malloc(4096)+4096, CLONE_VM | SIGCHLD, shared) < 0)
        abort();

    while(shared < 10) {
        sleep(1); printf("Processo padre. s = %d\n", shared);
    }

    sem_destroy(&S);
    return 0;
}
```

Concorrenza in Java

- ▶ 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.

Classi attive

Basic.java

```
class ClasseAttiva extends Thread {
    public void run(){
        while (true) System.out.println(this.getName());
    }
}

public class Basic {
    public static final void main(final String[] args) {
        ClasseAttiva o = new ClasseAttiva();
        o.start();
        while (true) System.out.println(" Main thread");
    }
}
```

Sincronizzazione

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

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

- ▶ 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.

Wait

- ▶ `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**

```
synchronized(lock){
    while (condizione(sharedX)) {
        try {
            lock.wait(); // rilascia lock
        } catch (InterruptedException e){
            e.printStackTrace(System.err);
        }
    }
    // quando la "condizione" e' falsa, proseguo (tenendo il lock)
    ...
}
```

NotifyAll

- ▶ `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()`

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

Altri modelli di concorrenza

- ▶ 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.*`

Semafori

- ▶ `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()`

Monitor

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

```
Lock l = new ReentrantLock();
Condition cond = lock.newCondition();
l.lock();
try {
    cond.await();
    cond.signal();
} finally {
    l.unlock();
}
```