



Sistemi Operativi

Bruschi
Monga Re

Concorrenza

Semafori

Sincronizzazione
con monitor
pthreads

Software
factory

Make

Sistemi Operativi¹

Mattia Monga

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

a.a. 2018/19

¹ © 2008–18 M. Monga. Creative Commons Attribuzione — Condividi allo stesso modo 4.0 Internazionale. <http://creativecommons.org/licenses/by-sa/4.0/deed.it>. Immagini tratte da [2] e da Wikipedia.



Sistemi Operativi

Bruschi
Monga Re

Concorrenza

Semafori

Sincronizzazione
con monitor
pthreads

Software
factory

Make

Lezione XV: Concorrenza

Concorrenza



Sistemi Operativi

Bruschi
Monga Re

Concorrenza

Semafori

Sincronizzazione
con monitor
pthreads

Software
factory

Make

- 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)



Sistemi Operativi

Bruschi
Monga Re

Concorrenza

Semafori

Sincronizzazione
con monitor
pthreads

Software
factory

Make

```
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) */
          malloc(4096)+4096, /* lo stack del nuovo processo
                               * (cresce verso il basso) */
          SIGCHLD, // in questo caso la clone è 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);
}

/* Isolati: ciascuno dei figli esegue 10 volte. */
/* Per il padre shared[0] è \testbf{sempre} 0 */

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

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

Thread (con mem. condivisa)



```
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) */
         malloc(4096)+4096, /* lo stack del nuovo processo
         * (cresce verso il basso) */
         CLONE_VM | SIGCHLD, // (virtual) memory condivisa
         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: è possibile una corsa critica. Il padre
 * condivide shared[0] con i figli */
while(shared[0] < 10) {
    sleep(1);
    printf("Processo padre. s = %d\n", shared[0]);
}
```

294

Sistemi Operativi

Bruschi Monga Re

Concorrenza

Semafori

Sincronizzazione con monitor pthreads

Software factory

Make

Thread (mutua esclusione con Peterson)



```
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;
}

int run(const int p, void* s)
{
    int* shared = (int*)s; // alias per comodità
    // Comma operator: https://en.wikipedia.org/wiki/Comma_operator
    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]);
    }
    leave_section(p, &shared[2]); // il test nel while è dopo enter_section

    return 0;
}
```

295

Sistemi Operativi

Bruschi Monga Re

Concorrenza

Semafori

Sincronizzazione con monitor pthreads

Software factory

Make

Performance



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

296

Sistemi Operativi

Bruschi Monga Re

Concorrenza

Semafori

Sincronizzazione con monitor pthreads

Software factory

Make

Semafori



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

297

Sistemi Operativi

Bruschi Monga Re

Concorrenza

Semafori

Sincronizzazione con monitor pthreads

Software factory

Make



```
statement1;           statement2;
    sem_init(&ss, 0, 0); // init a 0
```

```
statement1;           down(&semaforo);
up(&semaforo);       statement2;
```



1 deve eseguire prima di B, A deve eseguire prima di 2. Come fareste?

```
statement1;           statementA;
statement2            statementB
```



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

```
void up(sem_t *s){
    if (sem_post(s) < 0){
        perror("Errore semaforo (up)");
        exit(1);
    }
}
```



```
int shared = 0;
pthread_t p1, p2;
sem_t ss;

void* run(void* s){
    while (down(&ss),
           shared < 10) {
        sleep(1);
        printf("Processo thread (%p). s = %d\n",
               pthread_self(), shared);
        if (!(shared < 10)){
            printf("Corsa critica!!!\n");
            abort();
        }
        shared += 1;
        up(&ss);
        pthread_yield();
    }
    up(&ss);
    return NULL;
}
```



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)

```
pthread_create(thread,attr,start_routine,arg)
pthread_exit (status)
pthread_join (threadid,status)
pthread_mutex_init (mutex,attr)
pthread_mutex_lock (mutex)
pthread_mutex_unlock (mutex)
pthread_cond_init (condition,attr)
pthread_cond_wait (condition,mutex)
pthread_cond_signal (condition)
pthread_cond_broadcast (condition)
```



Tralasciando le inizializzazioni dei puntatori mutex e condition:

```
// T1
pthread_mutex_lock(mutex); // Acquisire il lock
while (!predicate) // fintantoch'è la condizione `è falsa
    pthread_cond_wait(condition, mutex); // block
pthread_mutex_unlock(mutex); // rilasciare il lock

// T2
// qualche thread rende vero il predicato cos'{'i}
pthread_mutex_lock(mutex); // Acquisire il lock
predicate = TRUE;
pthread_cond_broadcast(condition); // e lo segnala
pthread_mutex_unlock(mutex); // rilasciare il lock
```



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

```
// T1
pthread_mutex_lock(mutex);
while (!predicate)
    //
    //
    pthread_cond_wait(condition, mutex);
pthread_mutex_unlock(mutex);

// T2
//
//
predicate = TRUE;
pthread_cond_signal(condition);
//
```



- 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)

Produttore e consumatore



```
#define N 10
char* buffer[N];
int count = 0;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;

void b_insert(char* o){
    pthread_mutex_lock(&lock);

    while (count == N) pthread_cond_wait(&full, &lock);
    printf("Inserimento in buffer con %d\n", count);
    buffer[count++] = o;
    if (count == 1) pthread_cond_signal(&empty);

    pthread_mutex_unlock(&lock);
}
```

Sistemi Operativi

Bruschi Monga Re

Concorrenza

Semafori

Sincronizzazione con monitor pthreads

Software factory

Make

306

/ passaggio per indirizzo per evitare di fare la return fuori dai lock */*

Produttore e consumatore



```
void b_remove(char** result){
    pthread_mutex_lock(&lock);

    while (count == 0) pthread_cond_wait(&empty, &lock);
    printf("Rimozione in buffer con %d\n", count);
    *result = buffer[--count];
    if (count == N-1) pthread_cond_signal(&full);

    pthread_mutex_unlock(&lock);
}
```

Sistemi Operativi

Bruschi Monga Re

Concorrenza

Semafori

Sincronizzazione con monitor pthreads

Software factory

Make

307

Produttore e consumatore



```
void* producer(void* nonusato){
    printf("Esecuzione del produttore\n");
    while (1){
        char* o = (char*)malloc(sizeof(char));
        printf("Ho prodotto %p\n", o);
        b_insert(o);
    }
}

void* consumer(void* nonusato){
    printf("Esecuzione del consumatore\n");
    while (1){
        char* o;
        b_remove(&o);
        free(o);
        printf("Ho consumato %p\n", o);
    }
}
```

Sistemi Operativi

Bruschi Monga Re

Concorrenza

Semafori

Sincronizzazione con monitor pthreads

Software factory

Make

308

Produttore e consumatore



```
int main(void){
    pthread_t p1, p2;

    pthread_create(&p1, NULL, consumer, NULL);
    pthread_create(&p2, NULL, producer, NULL);

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    return 0;
}
```

Sistemi Operativi

Bruschi Monga Re

Concorrenza

Semafori

Sincronizzazione con monitor pthreads

Software factory

Make

309

Un piccolo esperimento



Sistemi Operativi

Bruschi Monga Re

Concorrenza

Semafori

Sincronizzazione con monitor pthreads

Software factory

Make

```
#define LIMIT 100000000
int printf(const char*, ...);
```

```
int main(){
    long i;
    long sum = 0;
    for (i=0; i<LIMIT; i++){
        sum += i;
    }
    printf("Sequential: %ld\n", sum);
    return 0;
}
```

Distribuire questa somma su N (p.es. 4) thread. Suggerimento:

```
void* run(void* param){
    int i = *((int*)(param));
    long start = (LIMIT / N) * i;
    long end = start + (LIMIT / N);
    // ....
}
```

310

UNIX software factory



Sistemi Operativi

Bruschi Monga Re

Concorrenza

Semafori

Sincronizzazione con monitor pthreads

Software factory

Make

- UNIX nasce come sistema *per i programmatori* (l'unica tipologia di utente all'inizio degli anni '70...)
- progettato insieme ad un linguaggio di programmazione (C)
- la 'filosofia di UNIX' (piccoli programmi che fanno molto bene una sola cosa su file) si adatta perfettamente al paradigma di sviluppo edit-compile-debug
- tool all'avanguardia nell'elaborazione di *file di testo* (per lo piú organizzati per "righe") e per la scrittura dei programmi di elaborazione stessi (lex, yacc,...)

311

Edit/Compile



Sistemi Operativi

Bruschi Monga Re

Concorrenza

Semafori

Sincronizzazione con monitor pthreads

Software factory

Make

- Editor: ed, vi, emacs manipolano arbitrariamente i byte di un file, generalmente interpretandoli come caratteri stampabili (testo)
- Compilatore: cc (gcc)
 - 1 cc sorgente (.c) \rightsquigarrow assembly (.s)
 - 2 as assembly \rightsquigarrow oggetto (.o)
 - 3 (ar archivia diversi oggetti in una *libreria* (.a)
 - 4 ld *oggetti e librerie* \rightsquigarrow eseguibile (a.out) (il formato storico è COFF, oggi ELF)

Si noti che a sua volta anche la compilazione vera e propria è fatta da due passi (pre-processore cpp e compilazione cc1).

312

Esercizio



Sistemi Operativi

Bruschi Monga Re

Concorrenza

Semafori

Sincronizzazione con monitor pthreads

Software factory

Make

- Scrivere in assembly (nasm) una funzione *somma* che restituisce (in eax secondo la convenzione del C) la somma di due interi (passati sullo stack, secondo la convenzione del C)
- Scrivere un programma C che usa la funzione *somma*
- Collegare i due programmi in un unico eseguibile

313



Stuart Feldman, 1977 at Bell Labs.

Permette di specificare dipendenze fra processi di generazione.

Dipendenze: se cambia (secondo la data dell'ultima modifica) un prerequisito, allora il processo di generazione deve essere ripetuto.

```
helloworld.o: helloworld.c
    cc -c -o helloworld helloworld.c
```

```
helloworld: helloworld.o
    cc -o $@ $<
```

```
.PHONY: clean
```

```
clean:
    rm helloworld.o helloworld
```

314



- Scrivere in assembly (nasm) una funzione **somma** che restituisce (in `eax` secondo la convenzione del C) la somma di due interi (passati sullo stack, secondo la convenzione del C)
- Scrivere un programma C che usa la funzione **somma**
- Collegare i due programmi in un unico eseguibile
- Codificare il procedimento in un Makefile

315