Sistemi Operativi[1]

Mattia Monga

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

a.a. 2019/20

[1] 🅭🅯🅭 2008–19 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
Concorrenza
Semafori
Sincronizzazione con monitor pthreads
Software factory
Make
Debugger
Low level programming
Programmare sistemi operativi
diff & patch
Versioning

Lezione XVI: Concorrenza

Sistemi Operativi
Bruschi Monga
Concorrenza
Semafori
Sincronizzazione con monitor pthreads
Software factory
Make
Debugger
Low level programming
Programmare sistemi operativi
diff & patch
Versioning

Concorrenza

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

Sistemi Operativi
Bruschi Monga
Concorrenza
Semafori
Sincronizzazione con monitor pthreads
Software factory
Make
Debugger
Low level programming
Programmare sistemi operativi
diff & patch
Versioning

Processi (senza 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!) */
                  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 \textbf{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 comodit\`a
        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]);
}
```


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;
```


Performance

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


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`

Meccanismo di base


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

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


Meccanismo di base

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

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


Mutua esclusione con semafori


```
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);
        }
}
```


Mutua esclusione con semafori


```
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;
}
```

POSIX threads

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)
```


Il pattern di base

Tralasciando le inizializzazioni dei puntatori mutex e condition:

```
// T1
pthread_mutex_lock(mutex); // Acquisire il lock
while (!predicate) // fintantoch\'e la condizione \`e 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
```


Perché il mutex?

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

```
// T1                                  // T2
pthread_mutex_lock(mutex);             //
while (!predicate)                     //
  //                                   predicate = TRUE;
  //                                   pthread_cond_signal(condition);
  pthread_cond_wait(condition, mutex);//
pthread_mutex_unlock(mutex);           //
```


Produttore e consumatore

- Il produttore smette di produrre se il buffer è pieno e deve essere avvisato quando non lo è piú (può ricominciare a produrre)
- Il consumatore smette di consumare se il buffer è vuoto e deve essere avvisato quando non lo è piú (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);
}
```



```
/* 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);
}
```


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);
  }
}
```


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;
}
```

Incrementer e watcher

- 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 e sono equivalenti perché una volta superata la soglia, il predicato "maggiore della soglia" rimane vero.

Un piccolo esperimento


```
#define LIMIT 1000000000
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);
  // ....
}
```


Lezione XIX: The UNIX software factory

UNIX software factory

- 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,...)

Edit/Compile

- 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`).

Esercizio

- 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

Make

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
```


Esercizio

- 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

Debugger: Concetti fondamentali

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.

Stato

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 tramite system call `brk` (API `malloc`).

Uso del debugger

- `break ...` (un simbolo o un indirizzo `*0x...`)
- `run ...` (eventualmente con argv)
- `print ...` (`x`)
- `next` (`nexti`)
- `step` (`stepi`)
- `backtrace`

Simboli

La *symbol table* serve al *linker* per associare nomi simbolici e indirizzi prodotti dal compilatore:

- contenuta in tutti gli oggetti, generalmente viene lasciata anche negli eseguibili (ma può essere scartata con `strip`)
- una versione piú ricca viene detta "simboli di debug" (vari formati, p.es. DWARF)
- le tabelle dei simboli possono essere consultate con `nm`

Edit/Compile

- 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`).

Perché capire i dettagli delle fasi?

Per costruire sistemi operativi a volte serve alterare il flusso tradizionale

```
gcc -O -nostdinc -I. -c bootmain.c
gcc -nostdinc -I. -c bootasm.S
ld -m    elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootasm
objdump -S bootblock.o > bootblock.asm
objcopy -S -O binary -j .text bootblock.o bootblock
$ nm kernel | grep _start
8010b50c D _binary_entryother_start
8010b4e0 D _binary_initcode_start
0010000c T _start
```


Assembly in C

In alcuni casi è comodo mischiare l'assembly al C (meno laborioso di organizzare il collegamento)

```
__asm__("nop");

__asm__("movl %eax, %ebx");
__asm__("xorl %ebx, %edx");
__asm__("movl $0, _booga");

__asm__("pushl %eax\n\t"
     "movl $0, %eax\n\t"
     "popl %eax");
```

Attenzione! Il compilatore C non "vede" l'effetto delle istruzioni assembly.

Assembly in C (cont.)

Si possono fare anche cose piú complicate, ma la sintassi è poco "amichevole"

```
__asm__("cld\n\t"
     "rep\n\t"
     "stosl"
     : /* no output registers */
     : "c" (count), "a" (fill_value), "D" (dest)
     : "%ecx", "%edi" );
```

La sintassi è

```
__asm__( "statements" : output_registers : input_registers : clobbered_registers);
```

http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html

diffutils

Con `cmp` è possibile controllare se due file sono identici.
Per i file di testo organizzato il righe esistono strumenti piú sofisticati:

- `diff` elenca le modifiche necessarie per trasformare un file in un altro (`diff3` si aiuta con un "antenato" comune, fondamentale per facilitare il *merge*)
- `diff` (e in maniera piú evoluta `diff3`) cerca di identificare le righe che *non sono cambiate*: le modifiche sono organizzate per hunk
- `patch` riapplica gli hunk di modifica al file originale (o versioni *leggermente* modificate dei medesimi)

Revision, version, configuration management

Dagli anni '80 sono stati proposti molti strumenti per trattare in modo efficiente:

- le successive revisioni di un file
- le versioni di un prodotto software
- le configurazioni che permettono di ottenere una specifica versione del prodotto

SCCS, RCS, CVS, SVN, `git`...
Si basano tutti sulla conservazione della "storia" dello sviluppo in un *repository*: per lavorare occorre fare *checkout* di un *artifact*, e poi chiedere il *commit* delle modifiche.

Versioning

L'idea può essere incorporata a vari livelli: Emacs può "salvare" automaticamente le versioni precedenti dei file (generalmente una sola `*~`, altrimenti `*~1~`...), oppure addirittura nel *file system*.
Git invece ricrea un suo "file system": blob e tree, ref.

- multi-phase commit: *working directory*, *stage* e *local repository*
- distribuito senza necessariamente server centralizzati: `pull` e `push`
- in un commit è conservato l'insieme delle modifiche (come 'diff') fatte ad un insieme (*change-set*) di file: perciò è associato a un *tree*
- una branch è semplicemente una *reference* mobile a una linea di sviluppo.