



Sistemi Operativi¹

Mattia Monga

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

a.a. 2015/16

¹© 2008–16 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.



Lezione XV: Memoria di massa 2



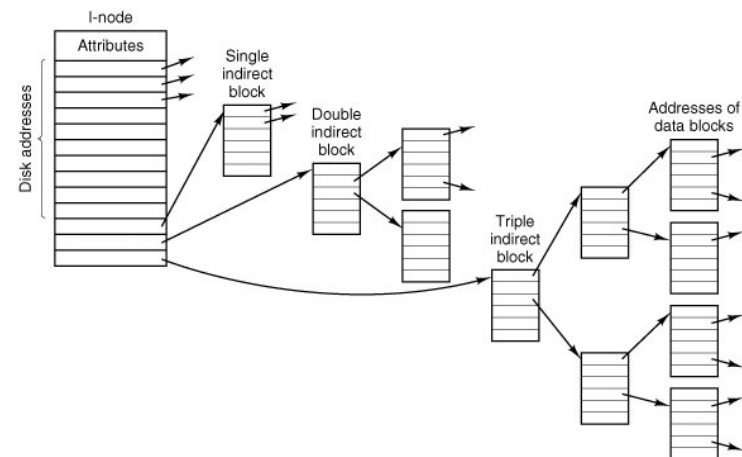
Creare e usare un fs

Una partizione può essere usata per ospitare un *file system*.

- Un file system va *creato* (`mkfs`)
- Un file system va *montato* (`mount`)
- Corrispondentemente va *smontato* (`umount`)
- Ogni file è caratterizzato da un *i-node* e conosciuto tramite uno o più link o nomi (`ln`)



i-node



Un esempio di i-node



Sistemi Operativi

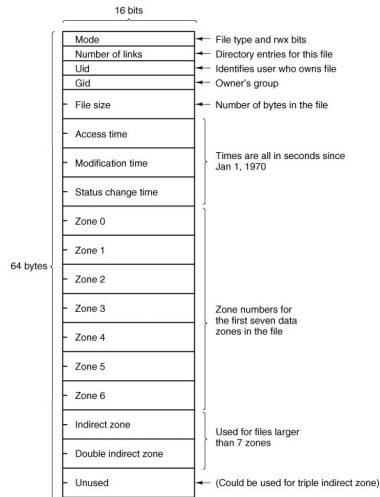
Bruschi Monga Re

Dischi virtuali

Esercizio

Concorrenza Semafori

Sincronizzazione con monitor pthreads



Con blocchi da 1KB, zone da 32 bit (default con EXT-2, che però ha 12 zone dirette):

- Qual è la dimensione massima di un file con un solo blocco di overhead?
- Qual è la dimensione massima di un file?
- Quanti blocchi di overhead sono necessari per un file da 100MB di dati?

280

Aggiungere un disco in QEmu



Sistemi Operativi

Bruschi Monga Re

Dischi virtuali

Esercizio

Concorrenza Semafori

Sincronizzazione con monitor pthreads

Fuori dalla macchina virtuale `qemu-img` create `disco.img` 100M
Poi può essere usato aggiungendo `-hdb disco.img`
In generale (anche al di là delle macchine virtuali) un file può facilmente essere usato come "disco": i *loop device* servono proprio per utilizzare un file (che è uno stream di caratteri) come *device a blocchi*

- 1 `dd if=/dev/zero of=prova.img seek=10M bs=1 count=0`
- 2 `/sbin/mkfs.ext2 prova.img`
- 3 `sudo mount -o loop prova.img /mnt`
- 4 `echo ciao > /mnt/pippo`
- 5 `sudo umount prova.img`

281

Esercizio



Sistemi Operativi

Bruschi Monga Re

Dischi virtuali

Esercizio

Concorrenza Semafori

Sincronizzazione con monitor pthreads

- 1 Creare un disco virtuale
- 2 Partizionare il disco
- 3 Creare il file system
- 4 Montare il file system

282

Lezione XVI: Concorrenza



Sistemi Operativi

Bruschi Monga Re

Dischi virtuali

Esercizio

Concorrenza Semafori

Sincronizzazione con monitor pthreads

283



- 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

284

Thread (con mem. condivisa)



```

1  int shared[2] = {0, 0};
2  /* int clone(int (*fn)(void *),
3   * void *child_stack,
4   * int flags,
5   * void *arg);
6   * crea una copia del chiamante (con le caratteristiche
7   * specificate da flags) e lo esegue partendo da fn */
8  if (clone(run, /* il nuovo
9           * processo esegue run(shared), vedi quarto
10          * parametro */
11         malloc(4096)+4096, /* lo stack del nuovo processo
12          * (cresce verso il basso!) */
13         CLONE_VM | SIGCHLD, // (virtual) memory condivisa
14         shared) < 0){
15     perror("Errore nella creazione");exit(1);
16 }
17
18 if (clone(run, malloc(4096)+4096, CLONE_VM | SIGCHLD, shared) < 0){
19     perror("Errore nella creazione");exit(1);
20 }
21
    
```

286

Processi (senza mem. condivisa)



```

1  int shared[2] = {0, 0};
2  /* int clone(int (*fn)(void *),
3   * void *child_stack,
4   * int flags,
5   * void *arg);
6   * crea una copia del chiamante (con le caratteristiche
7   * specificate da flags) e lo esegue partendo da fn */
8  if (clone(run, /* il nuovo
9           * processo esegue run(shared), vedi quarto
10          * parametro */
11         malloc(4096)+4096, /* lo stack del nuovo processo
12          * (cresce verso il basso!) */
13         SIGCHLD, // in questo caso la clone è analoga alla fork
14         shared) < 0){
15     perror("Errore nella creazione");exit(1);
16 }
17 if (clone(run, malloc(4096)+4096, SIGCHLD, shared) < 0){
18     perror("Errore nella creazione");exit(1);
19 }
20
21 /* Isolati: ciascuno dei figli esegue 10 volte. */
22 // Per il padre shared[0] è sempre 0 */
    
```

285

Thread (mutua esclusione con Peterson)



```

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 comodità
20     while (enter_section(p, &shared[1], &shared[2]), shared[0] < 10){
21         sleep(1);
22         printf("Processo figlio (%d) con %d\n", p, shared[0]);
23     }
24 }
    
```



```

1 $ time ./threads-peterson > /tmp/output
2 real 0m11.091s
3 user 0m0.000s
4 sys 0m0.089s
5 $ grep -c "Busy waiting" /tmp/output
6 92314477

```

288



```

1
2 void enter_section(int *s); /* in enter.asm */
3 void leave_section(int *s){ *s = 0; }
4
5 int run(const int p, void* s){
6     int* shared = (int*)s; // alias per comodità
7     while (enter_section(&shared[1]), shared[0] < 10) {
8         sleep(1);
9         printf("Processo figlio (%d). s = %d\n",
10             getpid(), shared[0]);
11         fflush(stdout);
12         if (!(shared[0] < 10)){
13             printf("Corsa critica!!!!\n");
14             abort();
15         }
16         shared[0] += 1;
17         leave_section(&shared[1]);
18         sched_yield();
19         usleep(50);
20     }
21     leave_section(&shared[1]); // il test nel while è dopo enter_section
22     return 0;

```

289



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

290



```

statement1                                statement2
1 sem_init(&ss, 0, 0); // init a 0
1 statement1;                               1 down(&semaforo);
2 up(&semaforo);                             2 statement2;

```

291



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

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



```
1 void down(sem_t *s){
2     if (sem_wait(s) < 0){
3         perror("Errore semaforo (down)");
4         exit(1);
5     }
6 }
```

```
1 void up(sem_t *s){
2     if (sem_post(s) < 0){
3         perror("Errore semaforo (up)");
4         exit(1);
5     }
6 }
```



```
1 void down(sem_t *s){
2     if (sem_wait(s) < 0){
3         perror("Errore semaforo (down)");
4         exit(1);
5     }
6 }
```

```
1 void up(sem_t *s){
2     if (sem_post(s) < 0){
3         perror("Errore semaforo (up)");
4         exit(1);
5     }
6 }
```



```
1 int shared = 0;
2 pthread_t p1, p2;
3 sem_t ss;
4
5 void* run(void* s){
6     while (down(&ss),
7            shared < 10) {
8         sleep(1);
9         printf("Processo thread (%p). s = %d\n",
10              pthread_self(), shared);
11         if (!(shared < 10)){
12             printf("Corsa critica!!!\n");
13             abort();
14         }
15         shared += 1;
16         up(&ss);
17         pthread_yield();
18     }
19     up(&ss);
20     return NULL;
21 }
```



Gli esempi visti finora usano clone per creare i thread, che però è una system call specifica di Linux. 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)

```

1 pthread_create(thread,attr,start_routine,arg)
2 pthread_exit (status)
3 pthread_join (threadid,status)
4 pthread_mutex_init (mutex,attr)
5 pthread_mutex_lock (mutex)
6 pthread_mutex_unlock (mutex)
7 pthread_cond_init (condition,attr)
8 pthread_cond_wait (condition,mutex)
9 pthread_cond_signal (condition)
10 pthread_cond_broadcast (condition)
    
```



Tralasciando le inizializzazioni dei puntatori mutex e condition:

```

1 // T1
2 pthread_mutex_lock(mutex); // Acquisire il lock
3 while (!predicate) // fintantoché la condizione è falsa
4     pthread_cond_wait(condition, mutex); // block
5 pthread_mutex_unlock(mutex); // rilasciare il lock
6
7 // T2
8 // qualche thread rende vero il predicato così
9 pthread_mutex_lock(mutex); // Acquisire il lock
10 predicate = TRUE;
11 pthread_cond_signal(condition); // e lo segnala
12 pthread_mutex_unlock(mutex); // rilasciare il lock
    
```



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

```

1 // T1
2 pthread_mutex_lock(mutex);
3 while (!predicate)
4     //
5     //
6     pthread_cond_wait(condition, mutex);
7 pthread_mutex_unlock(mutex);

1 // T2
2 //
3 //
4 predicate = TRUE;
5 pthread_cond_signal(condition);
6 //
7 //
    
```



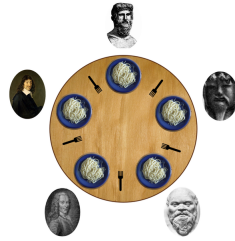
- Il produttore smette di produrre se il buffer è pieno e deve essere avvisato quando non lo è più (può ricominciare a produrre)
- Il consumatore smette di consumare se il buffer è vuoto e deve essere avvisato quando non lo è più (può ricominciare a consumare)
- 2 condition variable: buffer pieno e buffer vuoto (ne servono due perché pieno \neq \neg vuoto)



Sistemi Operativi

Bruschi Monga Re

- Ogni filosofo o mangia o pensa
- Mutua esclusione: per mangiare servono due forchette (forse sono chopsticks. . .)
- No starvation: Nessuno filosofo deve morire di fame (*to starve*)



Dischi virtuali
Esercizio

Concorrenza
Semafori

Sincronizzazione
con monitor
pthreads



Sistemi Operativi

Bruschi Monga Re

Dischi virtuali
Esercizio

Concorrenza
Semafori

Sincronizzazione
con monitor
pthreads