

Processi o Thread? – considerazioni sulla scelta tra Processi e Thread – pt.2

Claudia Wilson Gator: Adesso che ci siamo incontrati hai nulla in contrario se non ci vediamo più?

Nel capolavoro **Magnolia** c'è una scena in cui **Claudia** (una bravissima **Melora Walters**) dice a **Jim** (un altrettanto bravo **John C.Reilly**) la frase qui sopra. Ecco, la reazione di **Claudia** è la stessa che hanno molti programmatori dopo i primi esperimenti col **multithreading**, quando si trovano ad affrontare per la prima volta cosucce come **race-condition** o **starvation**. In quel caso la tentazione, forte, è di non usare più i *thread*... *come se usare i processi fosse una passeggiata!* Sì, forse i processi sono un po' più facili da controllare e sincronizzare ma non bisogna sceglierli solo per questo motivo. Bisogna, invece, fare sempre una unica considerazione: nel mio progetto cosa è (tecnicamente) meglio usare? E una volta trovata la risposta (magari aiutandosi con lo specchietto **dello scorso articolo**) agire di conseguenza, senza mai spaventarsi e/o preoccuparsi: risolvere problemi è il nostro lavoro (*ebbene sì: è un lavoro da masochisti*).



...ti avevo detto di usare i processi, ma tu sei un testone...

Allora: dove eravamo rimasti? Sì, ora ricordo, si parlava di processi e *thread*, e l'articolo (che sicuramente conoscete a memoria) si era concluso mostrando un semplicissimo programma *multithread* ed una promessa, questa:

...è un esempio veramente semplice che mi permetterà di mostrarvi, nella seconda parte dell'articolo, una applicazione multiprocess che fa esattamente la stessa cosa, oltretutto con un codice veramente molto simile e con risultati di esecuzione abbastanza interessanti...

Bene, le promesse bisogna mantenerle, quindi è il turno della applicazione-replica in *multiprocess*. Vediamo come è strutturata:

1. Un programma padre che crea i due processi figli (lo chiameremo, ad esempio, *processes.c*). Equivale al *main()* del programma *multithread* dello scorso articolo.
2. Un programma figlio che verrà lanciato due volte dal programma padre (lo chiameremo, ad esempio, *process.c*). Equivale al *threadfunc()* del programma *multithread* dello scorso articolo.
3. Una mini-libreria di comunicazione **IPC** che usa un **memory-mapped file** (la chiameremo, ad esempio, *mmap.c*). Contiene la struttura dati da condividere, che è equivalente alla *shdata* del programma *multithread* dello scorso articolo.

Ed ora bando alle ciance, cominciamo con *processes.c*. Vai col codice!

```
// processes.c - main processo padre
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/wait.h>
```

```

#include "mmap.h" // funzione main()
int main(int argc, char* argv[])
{ // apre il memory mapped file const char *mmname = "mydata";
shdata *data = memMapOpen(mmname); // crea i processi figli
pid_t pid1, pid2; (pid1 = fork()) && (pid2 = fork()); // test
pid processi if (pid1 == 0) { // sono il figlio 1 printf("sono
il figlio 1 (%d): eseguo il nuovo processo\n", getpid()); char
*pathname = "process"; char *newargv[] = { pathname, NULL };
execv(pathname, newargv); exit(EXIT_FAILURE); // exec non
ritorna mai } else if (pid2 == 0) { // sono il figlio 2
printf("sono il figlio 2 (%d): eseguo il nuovo processo\n",
getpid()); char *pathname = "process"; char *newargv[] = {
pathname, NULL }; execv(pathname, newargv);
exit(EXIT_FAILURE); // exec non ritorna mai } else if (pid1 >
0 && pid2 > 0) { // sono il padre printf("sono il padre (%d):
attendo 10 sec per fermare i figli\n", getpid()); sleep(10);
data->stop = true; // attende la terminazione dei processi
figli int status; pid_t wpid; while ((wpid = wait(&status)) >
0) printf("sono il padre (%d): figlio %d terminato (%d)\n",
getpid(), (int)wpid, status); // chiude il memory mapped file
ed esce printf("%s: processi terminati: counter=%ld\n",
argv[0], data->counter); memMapClose(mmname, data, true);
exit(EXIT_SUCCESS); } else { // errore nella fork()
printf("error: %s\n", strerror(errno)); memMapClose(mmname,
data, true); exit(EXIT_FAILURE); }
}

```

Direi che il codice è sufficientemente chiaro, visto che è strutturalmente semplice e **ben commentato**. Qualche dubbio potrebbe sorgere a chi è poco pratico della **fork()**, per cui consiglio, al solito, la lettura del manuale UNIX/Linux nel link, dove viene bene illustrata. Per essere brevi (visto che questo non è un articolo sui segreti della *fork()*) posso solo riassumere che questa **system call** “*sdoppia*” il processo padre creandone uno identico che è il figlio. I due processi proseguono autonomamente dopo la chiamata di sistema e, normalmente, il codice prevede due flussi diversi in base al risultato della *fork()*. Spesso, ma non sempre, il processo figlio esegue un altro programma attraverso la funzione **exec()**: e questo è il nostro caso. Tra l’altro questo è il

caso (si fa per dire) “peggiore”: eseguendo un altro programma la separazione tra i processi è totale, quindi farli parlare tra di loro è un po’ più complicato.

Il programma *multithread* dello scorso articolo aveva due *thread* e, visto che questo nuovo esempio doveva essere identico, *processes.c* crea, come detto sopra al punto 1, due figli: notare il “trucchetto” che ho usato per realizzare questo (segnatevelo, non è del tutto usuale): le due *fork()* vengono chiamate in una espressione AND. Perché? Forse non è evidente a prima vista, ma la *fork()* “sdoppiando” il processo chiamante, deve essere chiamata “in cascata” con una certa cautela, visto che il numero di processi aumenterà con progressione geometrica di ragione 2, quindi è facile ottenere 2, 4, 8... processi, ma per averne tre, come nel nostro caso (un padre e due figli) bisogna operare come mostrato nell’esempio. E aggiungo una curiosità: cosa succede se metto una *fork()* in un loop? Succede che il sistema collassa, perché avete creato una **fork bomb**!

E ora passiamo al codice dei processi figli, *process.c*:

```
// process.c - main processo figlio
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "mmap.h" // funzione main()
int main(int argc, char* argv[])
{ // apre il memory mapped file const char *mmname = "mydata";
  shdata *data = memMapOpen(mmname); // process loop
  printf("processo %d partito\n", getpid()); unsigned long i = 1;
  for (;;) { // lock mutex pthread_mutex_lock(&data->mutex);
    // incrementa i counter data->counter++; i++; // unlock mutex
    pthread_mutex_unlock(&data->mutex); // test stop flag if (data->stop) {
    // il processo esce (e chiude mmname)
    printf("processo %d terminato dal padre (i=%ld counter = %ld)\n",
      getpid(), i, data->counter); memMapClose(mmname, data, false);
    exit(EXIT_SUCCESS); } // sleep processo (uso usleep solo per comodità)
    usleep(1000); } // il processo esce per altro motivo che lo stop flag (e chiude mmname)
```

```
printf("processo %d terminato localmente (i=%ld counter =
%ld)\n", getpid(), i, data->counter); memMapClose(mmname,
data, false); exit(EXIT_SUCCESS);
}
```

Non so se avete notato: è, come promesso, praticamente identico alla funzione *threadfunc()* del programma *multithread* dello scorso articolo, con l'unica differenza (quasi trascurabile) del modo di accesso ai dati condivisi, che qui usa l'IPC fornito dalla mini-libreria *mmap*. Visto che questo codice l'ho, praticamente, già` descritto in **altri articoli**, possiamo passare subito all'ultima parte, la *mmap*: vai col codice!

```
// mmap.h - header mini-libreria IPC con memory mapped file
#include <pthread.h>
#include <stdbool.h> // struttura per i dati condivisi
typedef struct { pthread_mutex_t mutex; // mutex comune ai
processi bool stop; // flag per stop processi unsigned long
counter; // dato comune ai processi
} shdata; // prototipi globali
shdata *memMapOpen(const char *mmname);
int memMapClose(const char *mmname, shdata *ptr, bool remove);
```

questo era l'*header mmap.h*. La struttura dei dati condivisi *shdata* è, come promesso, esattamente identica a quella usata nel programma *multithread* dello scorso articolo. E ora ci manca solo l'implementazione *mmap.c*:

```
// mmap.c - implementazione mini-libreria IPC con memory
mapped file
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/mman.h>
#include "mmap.h" // memMapOpen() - apre un memory mapped file
shdata *memMapOpen( const char *mmname)
{ // apre un memory mapped file (il file "mmname" è creato in
/dev/shm) int fd; if ((fd = shm_open(mmname, O_CREAT | O_RDWR,
```

```

0666)) < 0) { fprintf(stderr, "shm_open error: %s\n",
strerror(errno)); return NULL; } // tronca un memory mapped
file if (ftruncate(fd, sizeof(shdata)) < 0) { fprintf(stderr,
"ftruncate error: %s\n", strerror(errno)); return NULL; } //
mappa un memory mapped file shdata *ptr; if ((ptr = mmap(NULL,
sizeof(shdata), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)) <
0) { fprintf(stderr, "mmap error: %s\n", strerror(errno));
return NULL; } // init mutex in modo "shared memory"
pthread_mutexattr_t attr; pthread_mutexattr_init(&attr);
pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(&ptr->mutex, &attr); // init altri dati
comuni ai processi (e return pointer) ptr->stop = false;
ptr->counter = 0; return ptr;
} // memMapClose() - chiude un memory mapped file
int memMapClose( const char *mmname, shdata *ptr, bool remove)
{ // un-mappa un memory mapped file if (munmap(ptr,
sizeof(shdata)) < 0) { fprintf(stderr, "munmap error: %s\n",
strerror(errno)); return -1; } // cancella un memory mapped
file if (remove && shm_unlink(mmname) < 0) { fprintf(stderr,
"shm_unlink error: %s\n", strerror(errno)); return -1; } //
success return 0;
}

```

Come avrete notato ho scritto una mini-libreria di una semplicità disarmante. Contiene solo due chiamate, una, la *memMapOpen()*, per creare/aprire il *memory-mapped file* (e inizializzare i dati) e un'altra, la *memMapClose()*, per chiuderlo. Non ci sono funzioni di lettura, scrittura, ecc.: una volta aperta la memoria condivisa ci si accede esattamente come se fosse una variabile (*pointer*) del programma. Una cosa veramente semplicissima.

E ora cosa ci manca per compiere la promessa? Ah, sì, avevo detto *...con risultati di esecuzione abbastanza interessanti...* E allora vediamo cosa scrivono sul terminale i nostri due programmi (*threads.c* e *processes.c*) durante l'esecuzione (su *UNIX* o *Linux*, ovviamente, una versione per *Windows* non userebbe *fork()*, ma questa è un'altra storia...):

```

programma multithread "threads"

```

```
-----  
thread 139969076664064 partito  
thread 139969068271360 partito  
thread 139969076664064 terminato dal main (i=9067  
counter=18134))  
thread 139969068271360 terminato dal main (i=9068  
counter=18135))  
./threads: thread terminati: counter=18135 programma  
multiprocess "processes"
```

```
-----  
sono il padre (5780): attendo 10 sec per fermare i figli  
sono il figlio 1 (5781): eseguo il nuovo processo  
sono il figlio 2 (5782): eseguo il nuovo processo  
processo 5781 partito  
processo 5782 partito  
processo 5782 terminato dal padre (i=9051 counter = 18099)  
processo 5781 terminato dal padre (i=9051 counter = 18100)  
sono il padre (5780): figlio 5781 terminato (0)  
sono il padre (5780): figlio 5782 terminato (0)  
./processes: processi terminati: counter=18100
```

Allora: entrambi i programmi operano per 10 secondi (come da codice, e se uno non si fida può eseguirli usando il comando **time**), e risulta che eseguono lo stesso numero di cicli (notare i valori dei contatori). Ossia: la versione *multiprocess* ha, esattamente le stesse prestazioni della versione *multithread*! Ecco perché parlavo di risultati interessanti... Tra l'altro anche usando una *sleep* di 1 us invece di una *sleep* di 1 ms (o, addirittura, senza nessuna *sleep*), si ottengono risultati comparabili, questo nel caso che qualcuno pensasse che il test è falsato da tempi di riposo troppo lunghi.

Ovviamente non sto dicendo che, magicamente, gli *heavy weight process* sono esattamente intercambiabili con i *light weight process*, ma, come evidenziato nel precedente articolo, le differenze prestazionali sono meno evidenti di quello che si crede (il **context-switch** dei *thread* è più leggero ma non così tanto). Concludendo: se vi fidate di me, quando avete dei dubbi seguite lo specchietto (quello in 3 punti) del

precedente articolo, non ve ne pentirete!

Ciao, e al prossimo post!

<https://italiancoders.it/processi-o-thread-considerazioni-sulla-scelta-tra-processi-e-thread-pt-2/>