

Il buono, il brutto, l'IPC – considerazioni sulle prestazioni della POSIX IPC – pt.3

Il Biondo: Vedi, il mondo si divide in due categorie: chi ha la pistola carica, e chi scava. Tu scavi.

E siamo arrivati (*finalmente! uff...*) all'**ultima parte** della nostra personale **Trilogia del dollaro**... ebbene sì, lo ammetto: la **Trilogia del IPC**, non diventerà una pietra miliare come i mitici tre film del grandissimo **Sergio Leone**, ma comunque io ce l'ho messa tutta, e spero che a qualcuno tutta questa sbrodolata torni utile. E, se non siete d'accordo, questa volta vi manderò a casa **il cattivo** a convincervi!



...il mondo si divide in due categorie: chi sa usare l'IPC e chi no...

Allora: chi ha letto le prime due parti della serie (**qui** e **qui**), oltre a concorrere per il premio "il lettore più

paziente dell'anno" può leggere in scioltezza quanto segue. Per gli altri raccomando una veloce lettura previa (almeno per capire de che se sta a parla') e, per punizione, lettura "in ginocchio sui ceci" (un po' di cultura popolare non guasta mai... anzi: promemoria per un futuro articolo: "Considerazioni sull'uso dei detti popolari nella divulgazione informatica").

E dunque: abbiamo già fornito tre (gagliardi) esempi d'uso di **POSIX IPC: FIFO (Named Pipe), Message Queue e UNIX domain socket (IPC socket)**. Oggi tocca alla quarta e ultima, la **Shared Memory**, che ho lasciato in fondo perché è un po' diversa dalle altre, non essendo un tipico mezzo di scambio di messaggi ma, come dice il nome, un sistema di condivisione della memoria. Rivediamo un attimo la definizione:

Shared Memory: *la comunicazione tra due o più processi viene raggiunta attraverso un pezzo di memoria condiviso tra tutti i processi. La memoria condivisa deve essere protetta dagli accessi simultanei usando meccanismi di sincronizzazione.*

E infatti, **in un altro articolo**, avevamo già visto un esempio d'uso nella sua forma tipica. Beh, ho deciso di inserire ugualmente in questa specie di "*Olimpiadi dell'IPC*" anche la **Shared Memory** forzandola a scambiare messaggi. Visto che questo uso è una forzatura non potremo aspettarci né grandi prestazioni né un codice particolarmente lineare e senza ridondanze, ma sarà, comunque, un interessante esercizio di programmazione.

E cominciamo con la parte "normale" di questo ciclo di articoli, ossia l'*header* **data.h**, il padre **processes.c** e i due figli **writer.c** e **reader.c**... vai col codice!

```
#ifndef DATA_H
#define DATA_H // nome del memory mapped file
#define MMAP_NAME "mymmap" // numero di messaggi da scambiare per il benchmark
#define N_MESSAGES 2000000 // struttura Data per i messaggi
typedef struct { unsigned long index; // indice dei dati char text[1024]; // testo dei dati
```

```
} Data; #endif /* DATA_H */
```

```
// 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"
```

```
#include "data.h" // funzione main()
```

```
int main(int argc, char* argv[])
```

```
{ // creo il memory mapped file shdata *data; if ((data =  
memMapOpen(MMAP_NAME, sizeof(Data), true)) == NULL) { //  
errore di creazione printf("%s: non posso creare il memory  
mapped file (%s)\n", argv[0], strerror(errno));  
exit(EXIT_FAILURE); } // 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 =  
"reader"; 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 = "writer"; 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 la terminazione dei figli\n", getpid()); int status;  
pid_t wpid; while ((wpid = wait(&status)) > 0) printf("sono il  
padre (%d): figlio %d terminato (%d)\n", getpid(), (int)wpid,  
status); // rimuovo il memory mapped file ed esco printf("%s:  
processi terminati\n", argv[0]); memMapClose(MMAP_NAME, data);  
exit(EXIT_SUCCESS); } else { // errore nella fork(): rimuovo  
il memory mapped file ed esco printf("%s: fork error (%s)\n",  
argv[0], strerror(errno)); memMapClose(MMAP_NAME, data);  
exit(EXIT_FAILURE); }  
}
```

```
// writer.c - main processo figlio
```

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

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

```

#include <string.h>
#include <errno.h>
#include <unistd.h>
#include "mmap.h"
#include "data.h" // funzione main()
int main(int argc, char* argv[])
{ // apro il memory mapped file printf("processo %d
partito\n", getpid()); shdata *data; if ((data =
memMapOpen(MMAP_NAME, sizeof(Data), false)) == NULL) { //
errore di apertura printf("%s: non posso aprire il memory
mapped file (%s)\n", argv[0], strerror(errno));
exit(EXIT_FAILURE); } // loop di scrittura messaggi per il
reader Data my_data; my_data.index = 0; for (;;) { // test
index per forzare l'uscita if (my_data.index == N_MESSAGES) {
// il processo esce per indice raggiunto printf("processo %d
terminato (text=%s index=%ld)\n", getpid(), my_data.text,
my_data.index); exit(EXIT_SUCCESS); } // compongo il messaggio
e lo invio my_data.index++; snprintf(my_data.text,
sizeof(my_data.text), "un-messaggio-di-test:%ld",
my_data.index); memMapWrite(data, &my_data, sizeof(Data)); }
// il processo esce per altro motivo (errore: non gestito in
questa versione) printf("processo %d terminato con errore
(%s)\n", getpid(), strerror(errno)); exit(EXIT_FAILURE);
}

```

// reader.c - main processo figlio

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#include "mmap.h"
#include "data.h" // funzione main()
int main(int argc, char* argv[])
{ // apro il memory mapped file printf("processo %d
partito\n", getpid()); shdata *data; if ((data =
memMapOpen(MMAP_NAME, sizeof(Data), false)) == NULL) { //
errore di apertura printf("%s: non posso aprire il memory
mapped file (%s)\n", argv[0], strerror(errno));
exit(EXIT_FAILURE); } // set clock e time per calcolare il

```

```

tempo di CPU e il tempo di sistema clock_t t_start = clock();
struct timeval tv_start; gettimeofday(&tv_start, NULL); //
loop di lettura messaggi dal writer Data my_data; for (;;) {
// leggo un messaggio memMapRead(data, &my_data,
sizeof(Data)); // test index per forzare l'uscita if
(my_data.index == N_MESSAGES) { // get clock e time per
calcolare il tempo di CPU e il tempo di sistema clock_t t_end
= clock(); double t_passed = ((double)(t_end - t_start)) /
CLOCKS_PER_SEC; struct timeval tv_end, tv_elapsed;
gettimeofday(&tv_end, NULL); timersub(&tv_end, &tv_start,
&tv_elapsed); // il processo esce per indice raggiunto
printf("reader: ultimo messaggio ricevuto: %s\n",
my_data.text); printf("processo %d terminato " "(index=%ld CPU
time elapsed: %.3f s - total time elapsed:%ld.%ld s)\n",
getpid(), my_data.index, t_passed, tv_elapsed.tv_sec,
tv_elapsed.tv_usec / 1000); exit(EXIT_SUCCESS); } } // il
processo esce per altro motivo (errore: non gestito in questa
versione) printf("processo %d terminato con errore (%s)\n",
getpid(), strerror(errno)); exit(EXIT_FAILURE);
}

```

Ok, fino ad adesso nessuna sorpresa, il codice è mooolto simile a quello degli altri esempi visti finora, tranne che le funzioni di *read/write* sono usate (come avranno notato i lettori più attenti) senza testare il valore di ritorno. E adesso, per rompere la monotonia (*e prima che qualcuno si addormenti*) vediamo la “forzatura” di questo codice, ovvero la **mini-libreria mmap** che ho scritto per usare la memoria condivisa per leggere e scrivere messaggi: *puro masochismo però è un codice interessante (spero)*. Vediamolo, sono due file, **mmap.h** e **mmap.c**:

```

// 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 pthread_cond_t cond; // condition variable comune ai
processi bool data_ready; // flag per dati disponibili
(true=ready) size_t len; // lunghezza campo data char data[1];
// dati da condividere
} shdata; // prototipi globali

```

```

shdata *memMapOpen(const char *mmname, size_t len, bool
create);
void memMapClose(const char *mmname, shdata *ptr);
void memMapRead(shdata *ptr, void *buf, size_t count);
void memMapWrite(shdata *ptr, const void *buf, size_t count);

// mmap.c - implementazione mini-libreria IPC con memory
mapped file
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include "mmap.h" // memMapOpen() - apre un memory mapped file
shdata *memMapOpen( const char *mmname, size_t len, bool
create)
{ shdata *ptr; // test se modo create o modo open di un mmfile
già creato if (create) { // apre un memory mapped file (il
file "mmname" è creato in /dev/shm) int fd; if ((fd =
shm_open(mmname, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR)) == -1)
return NULL; // esce con errore // tronca un memory mapped
file if (ftruncate(fd, sizeof(shdata) + len) == -1) return
NULL; // esce con errore // mappa un memory mapped file if
((ptr = mmap(NULL, sizeof(shdata) + len, PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0)) == MAP_FAILED) return NULL; //
esce con errore // chiude la shared memory: questo non
compromette il map eseguito close(fd); // 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);
pthread_mutexattr_destroy(&attr); // init condition variable
in modo "shared memory" pthread_condattr_t attrcond;
pthread_condattr_init(&attrcond);
pthread_condattr_setpshared(&attrcond,
PTHREAD_PROCESS_SHARED); pthread_cond_init(&ptr->cond,
&attrcond); pthread_condattr_destroy(&attrcond); // init altri
dati ptr->data_ready = false; ptr->len = len; } else { // apre
un memory mapped file (il file "shmname" è creato in /dev/shm)
int fd; if ((fd = shm_open(mmname, O_RDWR, S_IRUSR | S_IWUSR))
== -1) return NULL; // esce con errore // mappa un memory
mapped file if ((ptr = mmap(NULL, sizeof(shdata) + len,

```

```

PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)) == MAP_FAILED)
return NULL; // esce con errore // chiude la shared memory:
questo non compromette il map eseguito close(fd); } // ritorna
il pointer return ptr;
} // memMapClose() - chiude un memory mapped file
void memMapClose( const char *mmname, shdata *ptr)
{ // rilascio tutte le risorse acquisite
pthread_mutex_destroy(&ptr->mutex);
pthread_cond_destroy(&ptr->cond); munmap(ptr, sizeof(shdata) +
ptr->len); shm_unlink(mmname);
} // memMapRead() - legge dati dal mapped-file
void memMapRead( shdata *ptr, void *buf, size_t count)
{ // lock mutex pthread_mutex_lock(&ptr->mutex); // aspetta la
condizione while (!ptr->data_ready)
pthread_cond_wait(&ptr->cond, &ptr->mutex); // legge i dati
dal mapped-file e segnala la condizione memcpy(buf, ptr->data,
count); ptr->data_ready = false;
pthread_cond_signal(&ptr->cond); // unlock mutex
pthread_mutex_unlock(&ptr->mutex);
} // memMapWrite() - scrive dati nel mapped-file
void memMapWrite( shdata *ptr, const void *buf, size_t count)
{ // lock mutex pthread_mutex_lock(&ptr->mutex); // aspetta la
condizione while (ptr->data_ready)
pthread_cond_wait(&ptr->cond, &ptr->mutex); // scrive i dati
sul mapped-file e segnala la condizione memcpy(ptr->data, buf,
count); ptr->data_ready = true;
pthread_cond_signal(&ptr->cond); // unlock mutex
pthread_mutex_unlock(&ptr->mutex);
}

```

Ok, no? È un codice abbastanza semplificato che, per essere messo in produzione, necessiterebbe di un po' più di controllo degli errori... ma è solo un esercizio, chi userebbe un sistema così per mandare messaggi quando ci sono le funzioni ad-hoc della famiglia **Message Queues**? Comunque, è un esempio funzionante (*compilare ed eseguire per credere, eh!*). Ovviamente, visto il tipo di funzionamento della **Shared Memory** (vedi la definizione qua sopra) nella struttura base *shdata* sono previsti un *mutex* e una *condition variable* per gestire la sincronizzazione degli accessi (meccanismo che è, invece,

intrinseco con le altre **POSIX IPC** viste negli articoli precedenti).

Ci manca solo da descrivere una cosa: il trucco del “*char data[1]*” (in **mmap.h**) usato per rendere generici i dati da condividere. Questo campo è (non a caso) l’ultimo della struttura dati “*shdata*” che descrive il *mapped-file*, e funziona così: quando si crea il file si passa, alla **memMapOpen()**, il size dei dati da scambiare (usando un *sizeof*): quindi nel nostro caso la dimensione passata è *sizeof(Data)*. Dentro la **memMapOpen()** il *mapped-file* viene mappato (usando la *system call* **mmap()**) con la dimensione totale richiesta, che è composta da una parte base fissa (la struttura *shdata*) e dalla parte che potremmo definire “*variabile*” (la struttura *Data*). Il risultato finale è un *mapped-file* impostato per scambiare dati nella sua parte variabile “*char data[1]*”, che di base è lunga “*un char*” ma che, in realtà, è lunga “*sizeof(Data) char*” una volta mappato il file. *Un trucchetto da niente.*

E abbiamo anche i risultati!

```
sono il padre (14836): attendo la terminazione dei figli
sono il figlio 1 (14837): eseguo il nuovo processo
sono il figlio 2 (14838): eseguo il nuovo processo
processo 14837 partito
processo 14838 partito
processo 14838 terminato (text=un-messaggio-di-test:2000000
index=2000000)
reader: ultimo messaggio ricevuto: un-messaggio-di-
test:2000000
processo 14837 terminato (index=2000000 CPU time elapsed:
4.077 s - total time elapsed:4.657 s)
sono il padre (14836): figlio 14837 terminato (0)
sono il padre (14836): figlio 14838 terminato (0)
./processes: processi terminati
```

Evidentemente i risultati non sono stupefacenti, ma neanche disastrosi: un messaggio ogni 2 us, e siamo quasi al livello delle **IPC socket**. Quindi: l’uso improprio della **Shared Memory**

viene bocciato a causa dei risultati non proprio eccellenti e, soprattutto, per la inutile complicazione del codice. Però vi assicuro che, se non la usiamo per leggere/scrivere messaggi, ma la usiamo *“come si deve”* (**tipo l'esempio dell'articolo citato sopra**) accedendo ad essa con un *pointer*, consente di scrivere codice elegante ed efficiente, che può risultare utile in molti progetti.

Teoricamente l'articolo e il ciclo dovrebbero terminare qui, ma quell'irascibile di **mio cuggino** mi ha ricordato (*con male parole, al solito*) che avevo promesso di fare un confronto finale con i thread:

mio cuggino: *E tu vorresti perdere una occasione per sfatare la leggenda urbana che i thread sono più veloci dei processi? Le promesse si mantengono! Non ti leggo più!*

io: *va bene, ma visto che un confronto di codice e prestazioni coi thread l'avevo già fatto in quell'altro articolo, ti va bene se qui lo sfumo un po'?*

E allora sfumerò e sarò brevissimo: senza stare a ripetere codice già visto, provate a immaginare (e magari provate a scriverlo, è un utile e semplice esercizio) un confronto con la **Message Queue**, che è velocissima e ben si presta all'uso sia *multiprocess* che *multithread* (è anche *thread-safe!*). Scrivete un semplice processo padre che crea la coda esattamente **come nell'esempio già visto**, e che invece di generare due processi figli, crea due *thread*. I due *thread* eseguono due funzioni che devono essere (praticamente) identiche ai *main()* dell'esempio con la **Message Queue** (**si può fare**, ve lo assicuro). Aggiungete un po' di pepe e sale, compilate ed eseguite. I risultati dorrebbero essere questi:

```
thread 140026602206976 partito
thread 140026593814272 partito
thread 8884 terminato (text=un-messaggio-di-test:2000000
index=2000000)
reader: ultimo messaggio ricevuto: un-messaggio-di-
test:2000000
thread 8884 terminato (index=2000000 CPU time elapsed: 3.848 s
```

```
- total time elapsed: 1.931 s)
./threads: thread terminati
```

Ma guarda un po'... le prestazioni **sono identiche alla versione multiprocess!** E non fatevi ingannare dal CPU time che è il doppio del total time: non è un errore, è che su una macchina *multi-core* come quella che ho usato per i test (*un i7 4 core/8 thread*) i due *thread* vengono eseguiti su due *thread* diversi della CPU e il tempo calcolato nel codice è la somma dei due tempi *"reali"*. E quindi è vero: come già visto anteriormente, i processi (almeno su Linux) hanno una velocità paragonabile a quella dei thread, e la scelta *multithread/multiprocess* si deve fare secondo altri criteri (**ricordate le regole che avevo descritto in quell'altro articolo?**). *Meditate gente, meditate...*

Ciao, e al prossimo post!

<https://italiancoders.it/il-buono-il-brutto-lipc-considerazioni-sulle-prestazioni-della-posix-ipc-pt-3/>