

The Fastpipe – come velocizzare le POSIX IPC in C (e C++)

***Enigmista:** Può esser crudele, poetica o cieca... ma quando è negata, violenza a volte reca.*

***Batman:** Giustizia. La risposta è giustizia.*

Per questo articolo ho scelto di rifarmi al bel **The Batman**, del bravo **Matt Reeves**. **The Batman** è una sorta di *reboot* del mitico Batman di **Nolan**, da cui eredita le atmosfere cupe e la profondità degli intrecci. E anche questo articolo è un po' un *reboot* (e occhio: non è un *remake*, non è la stessa cosa). Ho (ri)preso la parte tecnica (*il codice*) di un **vecchio articolo** e l'ho riscritta per verificare se era possibile aumentare le prestazioni (*spoiler: è possibile, ma con il metodo seguito era, come vedrete, quasi scontato o, se preferite, "lapalissiano"*). l'argomento è, quindi: è possibile velocizzare i meccanismi di POSIX IPC (che già sono veloci di per se, come visto **qui**, **qui** e **qui**)? E magari si può fare anche in modo semplice? Lo vedremo tra poco!



...la mia Fastpipe è più veloce della tua...

Ok, veniamo al dunque: ci sono varie maniere di velocizzare un sistema di scambio dati, ma il primo che viene in mente, il più scontato, deriva da questa semplice espressione:

meno dati = meno tempo

Dopo questa *perla matematica* penso avrete capito perché ho usato più sopra il termine **lapalissiano** (che è una maniera più elegante di dire “e grazie al c...o!”, ma questo è un blog serio, non posso scrivere parolacce, ah ah ah). Quindi tutti sanno che, spesso, quando si inviano dati si cerca di comprimerli, sempre sperando che l'esecuzione del codice di compressione/decompressione non annulli il vantaggio derivante dall'invio/ricezione di “meno dati”. Ma c'è una maniera più semplice di rispettare l'equazione qui sopra? Sì, ed è quella di inviare pacchetti di dati con dimensione variabile, corrispondente alla dimensione reale dei dati, senza sprecare neanche un byte. Quindi, ad esempio, se abbiamo un protocollo di trasmissione di messaggi ASCII (tipo una chat o un file transfer di testi) sarebbe un'ottima idea evitare di inviare

messaggi a lunghezza fissa (e, quindi, di non usare un buffer enorme per trasmettere un semplice "Ciao") no?

Mi sembra evidente che quanto sopra è abbastanza scontato e quasi inutile da verificare... ma, comunque, un bel *benchmark* non fa mai male, tanto per confermare la teoria con la pratica, per cui ho preso il codice del test della **POSIX pipe** visto nel vecchio ciclo di articoli, e l'ho modificato per ottenere due scopi:

1. Dimostrare che la "velocizzazione" è fattibile in maniera abbastanza semplice.
2. Dimostrare che è effettivamente più veloce

Ok, il codice originale ve lo risparmio perché per il *benchmark* non ho modificato praticamente nulla (solo qualche *printf*), quindi potete consultarlo qui. Per cui vi mostro, direttamente la nuova versione, 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 <sys/types.h>
#include <sys/stat.h>
#include "data.h" // funzione main()
int main(int argc, char* argv[])
{ // creo il file fifo (named pipe) if (mkfifo(FIFO_PATH,
DEFFILEMODE) == -1) { // errore di creazione printf("%s: non
posso creare il file fifo (%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 file fifo ed esco printf("%s: processi
terminati\n", argv[0]); remove(FIFO_PATH); exit(EXIT_SUCCESS);
} else { // errore nella fork(): rimuovo il file fifo ed esco
printf("%s: fork error (%s)\n", argv[0], strerror(errno));
remove(FIFO_PATH); exit(EXIT_FAILURE); }
}

```

// writer.c - main processo figlio

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include "data.h"
#include "message.h" // funzione main()
int main(int argc, char *argv[])
{ // apro il file fifo (named pipe) in modo scrittura
printf("processo %d partito (writer)\n", getpid()); int fd; if
((fd = open(FIFO_PATH, O_WRONLY)) == -1) { // errore di
apertura printf("%s: non posso aprire il file fifo (%s)\n",
argv[0], strerror(errno)); exit(EXIT_FAILURE); } // loop di
scrittura messaggi per il reader Message message; Data
*my_data = &message.data; my_data->index = 0; do { // test
index per forzare l'uscita if (my_data->index == N_MESSAGES) {
// il processo chiude il file fifo ed esce per indice
raggiunto printf("processo %d terminato (text=%s
messaggi=%ld)\n", getpid(), my_data->text, my_data->index);
close(fd); 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); } while (fastWrite(fd, &message) != -1); //
il processo chiude il file fifo ed esce per altro motivo
(errore) printf("processo %d terminato con errore (%s)\n",
getpid(), strerror(errno)); close(fd); 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 <fcntl.h>
#include "data.h"
#include "message.h" // funzione main()
int main(int argc, char *argv[])
{ // apro il file fifo (named pipe) in modo lettura
printf("processo %d partito (reader)\n", getpid()); int fd; if
((fd = open(FIFO_PATH, O_RDONLY)) == -1) { // errore di
apertura printf("%s: non posso aprire il fifo (%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 Message message; Data *my_data = &message.data; while
(fastRead(fd, &message) != -1) { // 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 chiude il file fifo ed esce per
indice raggiunto printf("reader: ultimo messaggio ricevuto:
%s\n", my_data->text); printf("processo %d terminato
(messaggi=%ld tempo CPU: %.3f - " "tempo totale:%ld.%ld)\n",
getpid(), my_data->index, t_passed, tv_elapsed.tv_sec,
tv_elapsed.tv_usec / 1000); close(fd); exit(EXIT_SUCCESS); } }
// il processo chiude il file fifo ed esce per altro motivo
(errone) printf("processo %d terminato con errore (%s)\n",
getpid(), strerror(errno)); close(fd); exit(EXIT_FAILURE);
}

```

```

// data.h - header per dati per mini-libreria IPC con pipes
#ifndef DATA_H

```

```

#define DATA_H // path del file fifo (named pipe)
#define FIFO_PATH "myfifo" // 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[16384]; // testo dei dati
} Data; #endif // DATA_H

// message.c - implementazione per read/write per mini-
libreria IPC con pipes
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include "data.h"
#include "message.h" // fastwrite - scrittura con size
ssize_t fastWrite( int fd, Message *buf)
{ // set size reale (somma di tutte le dimensioni dei membri
del // tipo Message (eccetto il membro size)) buf->size =
sizeof(unsigned long) + strlen(buf->data.text); // invio il
messaggio completo: size + real-size (size + somma di // tutte
le dimensioni dei membri del tipo Message (eccetto il membro
size)) return write(fd, buf, SIZE0FP(buf));
} // fastread - lettura con size
ssize_t fastRead( int fd, Message *buf)
{ // legge il size da usare nella successiva read ssize_t
size_rcvd; if ((size_rcvd = read(fd, &buf->size,
sizeof(size_t))) > 0) { // return la read successiva return
read(fd, &buf->data, buf->size); } // ritorna nessun byte
letto o errore return size_rcvd;
}

// message.h - header per read/write per mini-libreria IPC con
pipes
#ifndef MESSAGE_H
#define MESSAGE_H #include "data.h" // struttura Data per i
messaggi
typedef struct { size_t size; // size reale (somma di tutte le
dimensioni dei membri del // tipo Message (eccetto il membro
size)) Data data; // campo dati del messaggio
} Message; // size reale del messaggio
#define SIZE0FP(X) (sizeof(size_t) + sizeof(unsigned long) +

```

```
strlen(X->data.text)) // prototipi globali
ssize_t fastRead(int fd, Message *buf);
ssize_t fastWrite(int fd, Message *buf); #endif // MESSAGE_H
```

Ok, il nuovo codice è **ampiamente commentato** (come sempre), ma comunque è il caso di aggiungere qualche dettaglio. È formato da sei file, quindi due in più di quello di riferimento, per cui la struttura è questa:

1. Il *main* di un processo padre: **processes.c**. Crea ed esegue due processi figli con *fork* + *exec*. I due processi figli si chiameranno *writer* e *reader*.
2. Il *main* del processo *writer*: **writer.c**.
3. Il *main* del processo *reader*: **reader.c**.
4. Un *header file* per *reader* e *writer*: **data.h**.
5. Un nuovo sorgente che serve a gestire la trasmissione di messaggi con lunghezza variabile: **message.c**.
6. Un nuovo *header file* per il file *message.c*: **message.h**.

Come sicuramente avrete notato (e *vabbé*, spero che lo abbiate notato), **processes.c** e **data.h** sono identici a quelli vecchi, mentre **reader.c** e **writer.c** hanno minime differenze rispetto agli originali: inviano e ricevono un dato (definito in **data.h**) incapsulato in un nuovo tipo **Message** (definito in **message.h**), usando le funzioni di *read/write* implementate in **message.c**. Le nuove funzioni di *read/write* sono, come si può notare, abbastanza semplici, e usano internamente le funzioni di *read/write* della *libc*. E qual'è il trucco che fa funzionare tutto questo? Come si vede nel codice, funziona così:

1. Il dato "*Data*" viene incapsulato in un "*Message*" insieme alla lunghezza del dato stesso (che sarebbe il campo "*size*").
2. La funzione **fastWrite()** spedisce il messaggio specificando la lunghezza reale, usando la macro **SIZEOFP()**: spedisce esattamente i *byte* necessari, neanche uno in più.
3. La funzione **fastRead()** riceve il messaggio in due passaggi: legge i primi *byte* (4 o 8 in base al tipo di

architettura in uso) del messaggio per conoscere la lunghezza reale dei dati successivi; dopodiché esegue una nuova lettura usando la la lunghezza reale: riceve esattamente i byte necessari, neanche uno in più.

Visto? Il trucco è semplice e anche la sua implementazione lo è. E i risultati del test quali sono? Erano abbastanza scontati, ma è, comunque, il caso di mostrarli:

```
aldo@Linux $ cd pipes/
aldo@Linux $ ./processes sono il padre (18389): attendo la
terminazione dei figli
sono il figlio 1 (18390): eseguo il nuovo processo
sono il figlio 2 (18391): eseguo il nuovo processo
processo 18390 partito (reader)
processo 18391 partito (writer)
processo 18391 terminato (text=un-messaggio-di-test:2000000
messaggi=2000000)
reader: ultimo messaggio ricevuto: un-messaggio-di-
test:2000000
processo 18390 terminato (messaggi=2000000 tempo CPU: 7.278 -
tempo totale:7.283)
sono il padre (18389): figlio 18390 terminato (0)
sono il padre (18389): figlio 18391 terminato (0)
./processes: processi terminati aldo@Linux $ cd ../fastpipes/
aldo@Linux $ ./processes sono il padre (18401): attendo la
terminazione dei figli
sono il figlio 1 (18402): eseguo il nuovo processo
sono il figlio 2 (18403): eseguo il nuovo processo
processo 18402 partito (reader)
processo 18403 partito (writer)
processo 18403 terminato (text=un-messaggio-di-test:2000000
messaggi=2000000)
sono il padre (18401): figlio 18403 terminato (0)
reader: ultimo messaggio ricevuto: un-messaggio-di-
test:2000000
processo 18402 terminato (messaggi=2000000 tempo CPU: 3.397 -
tempo totale:3.398)
sono il padre (18401): figlio 18402 terminato (0)
./processes: processi terminati
```

Il miglioramento nell'invio di 2000000 messaggi è buono, 3.397 secondi invece di 7.278 secondi, ed era, come detto sopra, scontato, visto che invece di trattare pacchetti (a lunghezza fissa) di 16 KB trattiamo pacchetti di qualche decina di byte; magari ci si poteva aspettare qualcosa di più, ma è evidente che in un *benchmark* di questo tipo anche solo l'avvio di una operazione di *read* o *write* ha il suo peso, e contribuisce al tempo totale a prescindere dalla quantità di dati trattati.

Ok, il risultato è stato raggiunto! Ma ci sono delle **dolenti note**? Ebbene sì, quelle non mancano mai, e sono collegate al fatto che ho scelto, non casualmente, le **POSIX pipe** per effettuare questa semplice dimostrazione. Il fatto è che per la natura stessa di **file FIFO** su cui funzionano le **POSIX pipe** permette che un esempio come quello mostrato vada al primo colpo. Ma, e se volessimo usare lo stesso metodo con dei **POSIX IPC socket**? O con dei **Network socket**? O con delle **POSIX Message Queue**? Ecco, qui il discorso si complica un po'... Vediamo perché:

1. **POSIX IPC socket**: per la natura stessa del protocollo in uso (comparabile con il **TCP/IP**) non viene garantito l'invio "*non spezzettato*" dei pacchetti dati, quindi implementare il meccanismo qui sopra non è impossibile ma non è semplicissimo.
2. **Network socket**: per la natura stessa del protocollo in uso (il **TCP/IP**) è presente lo stesso problema del punto 1, sia usando TCP che UDP.
3. **POSIX Message Queue**: per la natura stessa di queste code il *buffer* di trasferimento è a lunghezza fissa, quindi non è possibile realizzare il trucco descritto.

Riepiloghiamo: a parte il caso 3 (che è impraticabile), se decidete di provare a implementare i punti 1 e 2 non vi stupite se non funzionano al primo colpo usando un semplice approccio come quello che ho usato per le **POSIX pipe**, il codice funzionante (che sicuramente riuscirete a scrivere) sarà sicuramente più complesso, ma i risultati finali saranno

decisamente migliori delle versioni che usano messaggi a lunghezza fissa. *Provare per credere!* Comunque non mi faccio responsabile degli, eventuali, mal di testa che vi verranno per implementare le versioni difficili... e ricordate: “quando il gioco si fa duro i duri cominciano a giocare” (e “non ci sono più le mezze stagioni”, e “si stava meglio quando si stava peggio”, e... non facciamoci mai mancare i luoghi comuni, ah ah ah).

Ciao, e al prossimo post!

<https://italiancoders.it/the-fastpipe-come-velocizzare-le-posix-ipc-in-c-e-c/>