Licorice System pt.2 – come scrivere una system(3) con cattura dello stdout in C

Alana: *Lo sapevo! Sapevo che era quello che stavi pensando. Pensi sempre alle cose, pensatore! Tu, pensatore! Tu pensi cose!*

Nell'ultimo articolo avevo annunciato un seguito incentrato sui metodi di programmazione real-time, ma ho deciso di rimandarlo, perché ho avuto la necessità di aggiungere (*per motivi miei esterni al blog*) una nuova funzionalità a una versione migliorata della **system(3)** che avevo proposto **in un altro articolo**, e mi è sembrata una buona idea fare una seconda puntata con un po' di dettagli (spero interessanti). Il fatto è che bisognerebbe sempre essere come il "*pensatore*" della frase citata qui sopra (tratta dal bel **Licorice Pizza** del Maestro **Paul Thomas Anderson**). *Pensare, pensare, pensare… quello si che è un gran lavoro, e i programmatori ne sanno qualcosa, no?*



…ho pensato ripetutamente, ma non ricordo cosa…

Riepilogando, vi ricordo che questo articolo è, di fatto, la

terza parte di **System? No, grazie!**, anche se il titolo originale si era già perso nella seconda parte (*e, già che ci sono, vi ricordo che di articoli della famiglia "No Grazie!" ne ho scritti altri, e vi invito a leggerli o rileggerli,* ***qui****,* ***qui****,* ***qui*** *e* ***qui***).

Nella prima parte avevo descritto i mille problemi della **system(3)**, una vera funzione anti-pattern da non usare mai, e nella seconda parte avevo proposto una funzione, la **toutSystem()**, che correggeva vari problemi, tra cui il più grave:

"Quando si chiama la system(3) il programma principale viene sospeso fino al termine del comando invocato, e non c'è nessuna maniera di controllare efficacemente quello che sta succedendo."

e quanto sopra implica anche che, oltre alla mancanza di controllo, potremmo avere la nostra applicazione (o un thread dell'applicazione) completamente bloccata da una **system(3)** che non è ritornata! Con la **toutSystems()** questo problema sparisce, perché si introduce un fondamentale *timeout* oltre il quale il nostro comando esterno viene bloccato restituendo un adeguato tracciamento dell'errore. *Molto bene, no?*

Poi, però, ho pensato che alla **toutSystem()** manca qualcosa: Ok, non si blocca e ci permette di conoscere l'esito, buono o cattivo, della nostra esecuzione (attraverso il semplice codice di ritorno) ma… e se avessimo anche bisogno di ricevere dei dati dal comando esterno eseguito? Magari scritti, come è logico aspettarsi, nello *standard output (stdout per gli amici)*? Si può fare? Ma certo! vai col codice!

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <time.h>
```

```
#include <fcntl.h>
#include <sys/wait.h> // prototipi locali
static int toutSystemStdout(const char* command, unsigned int timeout_ms, char *dest, size_t n);
static void mySleep(unsigned int milliseconds); #define TOUT_SLEEP 500 // intervallo di sleep per il loop busy wait del timeout // funzione main()
int main(int argc, char *argv[])
{ // test con un programma che scrive sullo stdout e esce printf("main: eseguo toutSystemStdout(\"./test\", 5000, buf, sizeof(buf))\n"); char buf[256]; // verificare se questo può andare in overflow if (toutSystemStdout("./test", 5000, buf, sizeof(buf)) != -1) printf("%s: toutSystemStdout: cmd output: %s\n", __func__, buf); else printf("%s: toutSystemStdout error\n", __func__); return 0;
} // toutSystem() - una system(3) con timeout e catturo dello stdout del comando
static int toutSystemStdout( const char *command, // il comando shell da eseguire (e.g.: cp -v file1 file2) unsigned int timeout_ms, // timeout in ms: 0 significa senza timeout char *dest, // buffer destinazione per lo standard output del comando size_t n) // size del buffer
{ char errmsg_buf[256]; // creo una pipe in modo nonblocking int pipefd[2]; // pipefd[0] = lato input della pipe; pipefd[1] = lato output della pipe if (pipe2(pipefd, O_NONBLOCK) == -1) { // errore pipe printf("%s: errore pipe: %s\n", __func__, strerror_r(errno, errmsg_buf, sizeof(errmsg_buf))); return -1; } // fork + exec + wait pid_t pid = fork(); if (pid == 0) { // figlio // // chiudo pipefd[0] che è il lato input/read della pipe close(pipefd[0]); // uso dup2() invece di close()+dup() per evitare eventuali race-condition dup2(pipefd[1], STDOUT_FILENO); // per il stdout // chiudo pipefd[1] che è il lato output/write della pipe che già non serve close(pipefd[1]); // eseguo il comando come lo esegue la system(3) execl("/bin/sh", "sh", "-c", command, (char *) NULL); // questo viene eseguito solo se fallisce exec (exec non ritorna mai) printf("%s: figlio: processo %d: errore exec: %s\n", __func__, getpid(), strerror_r(errno, errmsg_buf, sizeof(errmsg_buf))); exit(EXIT_FAILURE); } else if (pid > 0) { // padre // // chiudo pipefd[1] che è il lato output/write della pipe close(pipefd[1]); // attesa uscita del figlio
```

```
printf("%s: padre: processo %d: attesa uscita del figlio\n", __func__, getpid()); int rc_wait; int status; if (timeout_ms > 0) { // check timeout // busy wait con timeout int cnt_wait = 0; while (read(pipefd[0], dest, n) != 0) { mySleep(TOUT_SLEEP); if (++cnt_wait > timeout_ms / TOUT_SLEEP) { // figlio non uscito prima del timeout: return errore printf("%s: padre: processo %d: waitpid timeout scaduto\n", __func__, getpid()); return -1; } } rc_wait = waitpid(pid, &status, WNOHANG); } else { // wait senza timeout while (read(pipefd[0], dest, n) != 0) mySleep(TOUT_SLEEP); rc_wait = waitpid(pid, &status, 0); } // attesa terminata: analizzo il risultato if (rc_wait != pid) { // errore waitpid printf("%s: padre: processo %d: errore waitpid (%s)\n", __func__, getpid(), strerror_r(errno, errmsg_buf, sizeof(errmsg_buf))); return -1; } else { // processo terminato: return risultato int result = -1; if (WIFEXITED(status)) { // questo è l'unico risultato accettato come successo result = 0; printf("%s: padre: processo %d: pid %d uscito (status=%d)\n", __func__, getpid(), pid, WEXITSTATUS(status)); } else if (WIFSIGNALED(status)) printf("%s: padre: processo %d: pid %d ucciso dal segnale %d\n", __func__, getpid(), pid, WTERMSIG(status)); else if (WIFSTOPPED(status)) printf("%s: padre: processo %d: pid %d fermato dal segnale %d\n", __func__, getpid(), pid, WSTOPSIG(status)); else printf("%s: padre: processo %d: pid %d con stato sconosciuto (status=%d)\n", __func__, getpid(), pid, status); return result; } } else { // errore fork printf("%s: processo %d: errore fork: %s\n", __func__, getpid(), strerror_r(errno, errmsg_buf, sizeof(errmsg_buf))); return -1; }
} // mySleep() - wrapper per nanosleep()
static void mySleep(unsigned int milliseconds)
{ struct timespec ts; ts.tv_sec = milliseconds / 1000; ts.tv_nsec = (milliseconds % 1000) * 1000000; nanosleep(&ts, NULL);
}
```

Come avrete notato dal codice e, come al solito, dai **commenti prolissi**, la nuova funzione che ho originalissimamente chiamato **toutSystemStdout()** mantiene la struttura della versione precedente però con un bel po’ di cambi, perché questo di intercettare nel processo padre lo *standard output*

del processo figlio non è esattamente un gioco da ragazzi. Il trucco consiste, fondamentalmente, nell'usare una *pipe* per mettere in comunicazione i due processi, ma bisogna farlo con un certo stile, se no non funziona nulla. Riassumo i punti più importanti commentati nel codice:

- Si aggiunge un *buffer* (con la relativa lunghezza) al prototipo della funzione, per salvare i dati scritti nello *stdout* dal comando esterno eseguito.
- Si crea una *pipe* (con **pipe2(2)**) di cui verrà usato solo il canale di comunicazione da processo figlio a processo padre. La *pipe* deve essere *nonblocking* per rispettare la natura della nostra *pseudo-system* con *timeout*.
- I descrittori del canale di scrittura della *pipe* deve essere duplicato sullo *standard output*, e bisogna usare **dup2(2)** invece della sequenza **close(2**)+**dup(2)**: questo perché **dup2(2)** esegue internamente la sequenza in maniera atomica (evitando problemi di *race-condition*), *e voi sapete già quanto ci teniamo alla robustezza della programmazione in ambito multitasking/multithreading, no?*
- Il ciclo di *busy-wait* per la gestione del *timeout* si fa, ora, sulla lettura (con **read(2)**) dello *stdout* invece che sul **waitpid(2)**: ovviamente se non si ha bisogno di leggere lo *stdout* del comando esterno è consigliabile usare la normale **toutSystem()**: *non esistono funzioni universali, per ogni caso d'uso bisogna usare sempre la funzione più adatta. Ricordatelo!*

Ah, prima che mi dimentichi: la funzione qui sopra è un esempio didattico: nella versione di produzione bisognerebbe mettere qualche chiusura e qualche controllo in più: ad esempio ne manca uno (solo possibile, credo) sull'*overflow* del buffer destinazione dei dati scritti. *Ma questo compito ve lo lascio a voi, sbizzarritevi!*

Per provare questa nuova funzione ho scritto un piccolissimo programma, **test.c**, da usare come comando esterno: scrive in

loop cinque volte *“ciao”* e poi esce dopo aver eseguito una **sleep(3)** di 4 secondi: la *sleep* serve per misurare se funziona ancora il *timeout* (che continua ad essere la parte fondamentale della funzione): chiamando la **toutSystemStdout()** con un *timeout* di 5 secondi si può giocare con la **sleep(3)** di **test.c** per verificare il buon funzionamento della nuova funzione (a parte verificare che riesca veramente a catturare i dati dello *stdout* del comando esterno). Il codice di **test.c** è questo:

```
#include <stdio.h>
#include <unistd.h> // funzione main()
int main(int argc, char* argv[])
{ for (int i = 0; i < 5; i++) fprintf(stdout, "ciao %d ", i);
sleep(4); return 0;
}
```

Se eseguiamo il nostro programma di prova della **toutSystemStdout()** con *timeout=5sec* e **test.c** compilato con *sleep=4* otteniamo:

```
aldo@Linux $ ./toutSystemStdout
main: eseguo toutSystemStdout("./test", 5000, buf, sizeof(buf))
toutSystemStdout: padre: processo 19866: attesa uscita del figlio
toutSystemStdout: padre: processo 19866: pid 19867 uscito (status=0)
main: toutSystemStdout: cmd output: ciao 0 ciao 1 ciao 2 ciao 3 ciao 4
```

mentre, se eseguiamo ancora con *timeout=5sec* e **test.c** compilato con *sleep=6sec* otteniamo:

```
aldo@Linux $ ./toutSystemStdout
main: eseguo toutSystemStdout("./test", 5000, buf, sizeof(buf))
toutSystemStdout: padre: processo 19880: attesa uscita del figlio
toutSystemStdout: padre: processo 19880: waitpid timeout scaduto
```

main: toutSystemStdout error

Come volevasi dimostrare: nel primo caso (quello buono) ottengo i dati scritti dal comando test sullo *stdout*, mentre nel secondo caso ottengo (come sperato) un errore di *timeout* senza dati. *Provare per credere!*

E anche per oggi può bastare. nel prossimo articolo torneremo sulla programmazione *real-time… o magari no, chissà che non mi tocchi rimandarlo ancora una volta, non si s*a mai!

Ciao, e al prossimo post!

https://italiancoders.it/licorice-system-pt-2-come-scrivere-una-system3-con-cattura-dello-stdout-in-c/