Tutta colpa della fork – come usare la fork(2) in multithreading in C (e C++)

albergatore: *Lui mi sembra una persona in gamba, intelligente, per bene... dice sempre buongiorno e buonasera... sempre buongiorno e buonasera... però... boh..*
Romeo: *Come boh?*
albergatore: *No per carità, mica per dirne male, per l'amor di Dio... uno che dice sempre buongiorno e buonasera... però... boh.*

Il dialogo surreale qui sopra è tratto dal bel **Tutta colpa del Paradiso** del compianto **Francesco Nuti**. Un dialogo che rappresenta l'incertezza nel giudicare persone e cose, e che ci aiuta a introdurre il tema del giorno: la **fork(2)** che è una *system call* veramente classica, preziosa e indispensabile dei sistemi **POSIX** (e di cui abbiamo parlato **qui** e non solo) si può sempre usare "*come se niente fosse*" o bisogna usarla con cautela? Uhmm… vediamolo!



…la fork? Si, funziona bene, però… boh…

Allora, tanto per tagliare subito la testa al toro possiamo dire che la **fork(2)** in un *"programma normale"* è affidabile al 100%, e ci mancherebbe solo: è una *system call* che esiste da sempre su **UNIX** (e, dopo, anche su **Linux** e in tutta la famiglia **POSIX**) ed è alla base della scrittura delle applicazioni **multiprocess**.

Ma cosa si intende per *"programma normale"* ? Ecco, direi che in questo caso si intende una applicazione **singlethread**: la **fork(2)** risale a una delle primissime versioni di **UNIX**: la **V1** (detta anche *UNIX First Edition*, 1971) ed è stata poi standardizzata nel primo standard **POSIX 1003.1-1988**. Internamente la **fork(2)** ha avuto varie evoluzioni: su **Linux**, ad esempio, è implementata internamente tramite una chiamata alla *system call* **clone(2)**, ma alla fin fine rimane sempre e comunque la cara, vecchia e affidabile **fork(2)**. Il **multithreading** è apparso su **UNIX** ben più tardi, ed è stato standardizzato, poi, con **POSIX 1003.1c-1995**, e i (presunti) problemi della **fork(2)**, come vedremo più avanti, sono cominciati lì…

E come si usa la **fork(2)**? Come detto sopra ne abbiamo già parlato, comunque vi propongo, di seguito, uno degli esempi più classici di *fork* + *exec* + *wait*. Vai col codice!

```
// testfork.c - test della fork(2)
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h> // funzione main()
int main(int argc, char* argv[])
{ // eseguo la fork() pid_t pid = fork(); if (pid == 0) { // sono il figlio: eseguo il comando "ls ./testfork" char *pathname = "/usr/bin/ls"; char *newargv[] = { "ls", "./testfork", NULL }; execv(pathname, newargv); exit(EXIT_FAILURE); // exec non ritorna mai } else if (pid > 0) { // sono il padre: attendo l'uscita del figlio int status;
```

```
waitpid(pid, &status, 0); exit(EXIT_SUCCESS); } else { // errore fork() printf("error: }
}
```

E non credo che ci sia nulla da aggiungere su questo codice chiaro *(spero)* e **ben commentato** (*o meglio: rileggetevi gli articoli citati sopra, grazie*).

Ma veniamo al nocciolo della questione: come siamo messi col **multithreading** mischiato col **multiprocessing**? Ecco, prima di impazzire cercando articoli e riferimenti sull'argomento (spinoso, devo dire) e/o scrivere test complicati, è meglio dare un occhiatina al manuale, che è sempre la fonte primaria e più completa di informazioni. La pagina ufficiale del **The Linux man-pages project**, **System Calls Manual – fork(2)** dice:

> After a fork() in a multithreaded program, the child can safely call only async-signal-safe functions (see signal-safety(7)) until such time as it calls execve(2).

e, per completare il quadro, vediamo anche cosa dice il **POSIX Programmer's Manual – FORK(3P)**:

> A process shall be created with a single thread. If a multi-threaded process calls fork(), the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, to avoid errors, the child process may only execute async-signal-safe operations until such time as one of the exec functions is called.

Ok, sembra che qualche problemino c'è…

In pratica cosa succede? Succede che se un programma *multithread* chiama la **fork(2)**, si crea, come previsto, un processo figlio che è una copia esatta del processo padre (un *clone:* non per nulla su **Linux**, come visto sopra, la **fork(2)** usa internamente la *system call* **clone(2)**), ma il nuovo processo è *singlethread*, ed è una copia del *thread* in cui è stata invocata la **fork(2)**. Infatti, sempre nel manuale di **Linux** troviamo:

The child process is created with a single thread—the one that called fork(). The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of pthread_atfork(3) may be helpful for dealing with problems that this can cause.

E quindi, alla fine della fiera, il problema principale risiede nelle *race-conditions* dovute a eventuali *mutex* (o altri tipi di *lock)* che risiedono, contemporaneamente, nei processi padre e figlio, il che crea la possibilità che si verifichino strani blocchi. E, se vogliamo completare il discorso, bisogna aggiungere che anche la gestione dei **segnali** diretti al nostro processo padre diventa problematica, visto che dopo avere eseguito **fork(2)** abbiamo in circolazione anche un figlio-clone.

Ma come si risolve tutto questo? Beh, negli estratti dei manuali appena presentati si raccomanda di usare nel processo figlio, prima di una eventuale **exec(3)**, solo funzioni della famiglia **async-signal-safe**; queste sembrano tante ma… non fatevi ingannare, in realtà sono pochissime! Pensate che, tra le tante cose assenti, c'è tutto *stdio*, inclusa la **printf(3)**! E non si può neanche manipolare la memoria con **malloc(3)** e **free(3)**! E non si può neanche registrare un problema con **syslog(3)** visto che usa dei *mutex* (*un grazie al collega Xavier P. per avermelo fatto notar*e). È un bel problema… In parallelo alla raccomandazione precedente si può, poi, usare **pthread_atfork(3)**, ma con molta cautela, perché è possibile dimenticarsi qualche dettaglio visto che non è una soluzione molto semplice da realizzare.

E quali sono i sintomi tipici di *"qualcosa è andato male nella fork + exec"* ? Direi che l'evento più probabile è che la **exec(3)** non si esegua perché il *child process* si è bloccato a causa di un *lock* ereditato dal padre: in questo caso, usando semplicemente il comando **ps**, si noterà che abbiamo due processi con lo stesso nome, padre e figlio, con il figlio che

non è stato *(ahimè)* sostituito da un altro programma tramite la **exec(3)**.

Credo che a questo punto sia il caso di mostrare una piccola guida riassuntiva di come procedere quando non si può fare a meno di usare la **fork(2)** in un programma *multithread* (applicherò, semplicemente, le avvertenze dei manuali). E quindi: la **fork(2)** si usa senza paura anche in *multitreading* (io l'ho fatto molte volte) ma seguendo il seguente schema numerato in ordine di importanza:

1. **Se possibile** usate sempre la sequenza *fork* + *exec* + *wait*, ed eseguite **exec(2)** immediatamente dopo la **fork(2)** senza mettere praticamente nulla in mezzo (esattamente come nell'esempio mostrato più sopra): la **exec(2)** cancella tutti gli (eventuali) *lock* in comune tra padre e figlio e il problema è risolto alla radice. Notare che **POSIX** mette addirittura a disposizione una funzione, la **posix_spawn(3)**, che esegue *fork* + *exec* in un passaggio solo, ed è quindi intrinsecamente sicura, ma non è semplicissima da usare (bene).
2. **Se proprio non potete** eseguire immediatamente **exec(2)**, prima di eseguirla dovete avere l'accortezza di usare solo funzioni di tipo **async-signal-safe**, e ricordate: non potete usare neanche la *"innocua"* **printf(3)**, per cui dovrete arrangiarvi con la **write(2)**.
3. **Se proprio non dovete** eseguire **exec(2)**, riducete al minimo il codice del processo figlio (usando solo funzioni di tipo **async-signal-safe**) e uscite quanto prima usando **_exit(2)** (e non **exit(3)**!). Non chiamate altre funzioni del programma, a meno che non siate sicuri al 10
4. **Se proprio siete** in una situazione *"speciale"* (non compresa nei tre punti precedenti) usate, con molta attenzione, **pthread_atfork(3)**, che è l'ultima risorsa disponibile.

E per oggi può bastare. Spero di aver contribuito a sfatare

alcuni (falsi) miti sulla problematicità della **fork(2)**, una *system call* storica e indispensabile, e che funziona benissimo… *ma bisogna saperla usare*. Immagino che molti di voi saranno già in pieno assetto pre-festivo, e invasi dallo Spirito Natalizio (beh, io si). Quindi vi lascerò in pace per un po': Buon Natale e Buon Anno a tutti!

Ciao, e al prossimo post!

https://italiancoders.it/tutta-colpa-della-fork-come-usare-la-fork2-in-multithreading-in-c-e-c/