Le system call nel sistema Unix

Antonio Rocchia

April 18, 2022

Contents

1 Le system call di Unix

Il kernel di Linux espone un API di funzioni che un utente o un processo può chiamare per richiedere che un servizio venga svolto dal sistema operativo a livello kernel.

Ogni linguaggio di programmazione fa uso di queste system call in tantissimi modi. Ad esempio per aprire e creare file, leggere da un file, inizializzare un processo e così via.

Nei linguaggi di programmazione moderni le system call sono esposte al programmatore tramite una serie di funzioni(wrapper) che ne semplificano l'uso ed impediscono che vengano commesse violazioni dell'API.

2 Lista delle system call Unix

- Controllo dei processi/thread:
  - Creazione processi
  - Terminazione processi
  - Sospensione processi
  - Attesa terminazione
  - Sostituzione di un processo
- Gestione del file system
- Gestione dei dispositivi
- Comunicazione

3 Wrapper delle system call nel linguaggio C

3.1 fork()

fork - Crea un processo figlio ,Includere fork

```
#include <unistd.h>

int fork();
```

Descrizione: `fork()` genera un processo figlio duplicando il processo che chiama questa funzione. Il nuovo processo è chiamato *figlio*, il processo chiamante è chiamato *padre*

Il processo figlio ed il padre *continuano l'esecuzione in spazi di memoria diversi.* Al momento della fork() *il figlio ottiene una copia dello spazio di memoria del padre* Il padre ed il figlio sono identici eccetto per queste differenze:

- Il figlio ha il suo PID unico e diverso dal padre
- il figlio non eredita allarmi (*alarm()*) dal padre
- Il segnale di terminazione del figlio è sempre *SIGCHLD*
- Il figlio eredita le copie dei descrittori(*file descriptor*) aperti dal padre
- Il filgio eredita una copia dello stream di direttori aperti dal padre (*opendir()*)

Valore di ritorno: In caso di successo, il processo padre riceve il PID del processo figlio appena generato, e il figlio riceve 0. In caso di fallimento, il padre riceve *-1*, non viene generato nessun processo figlio e *errno* viene impostata correttamente.

Esempi: Di seguito vengono riportati alcuni esempi di codice che mostrano gli utilizzi tipici di fork(). Si cerca dove possibile di spiegare al meglio cio che succede durante l'esecuzione

,Esempio minimale di chiamata a fork()

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

int main(void) {
   //  Inizializzo e controllo il valore di ritorno di fork()
   int pid = 0;
   pid = fork();
```

Il processo figlio ottiene una copia dello spazio di memoria del padre, in questo caso la variabile pid inizializzata a 0. Il processo padre ed il processo figlio proseguono l'esecuzione allo stesso momento (in parallelo), ma con una differenza, *fork restituisce il PID del processo figlio generato al padre e restituisce 0 al figlio.* Questo valore viene assegnato alla variabile pid. Ora possiamo scrivere del codice che permetta al processo di capire se è il padre o il figlio. ,

```
   if(pid==0) { // Sono il figlio
       // Codice del figlio
       printf("Sono il padre");
       return 0;
   } else if(pid > 0) { // Sono il padre
       // Codice del padre
       printf("Sono il padre");
   }
   return 0;
}
```

Il processo figlio esegue lo stesso codice del processo padre. Pertanto nella maggior parte dei casi è necessario terminare l'esecuzione del figlio prima di uscire dal blocco *if(pid==0)*

3.2 getpid(),getppid()

getpid, getppid - restituisce l'identificatore del processo(PID) ,Includere getpid()

```c
#include <unistd.h>

int getpid();
int getppid();
```

Descrizione: *getpid()* restituisce al processo che chiama la funzione il sui identificatore di processo *(PID)*.

getppid() restituisce al processo che chiama la funzione l'identificativo di processo *(PID)* del suo processo padre.

Controllo degli errori Questa system call non fallisce mai. Non è necessario alcun controllo.

Esempi: Di seguito alcuni esempi di codice che mostrano l'utilizo di getpid() e getppid().

getpid() in azione

```c
#include <unistd.h>

int main(void) {
    int pid = 0;

    pid = fork();
    if(pid==0) {
        // processo figlio
        printf("La variabile pid = %d\n", pid);
        printf("Il mio pid: %d\n", getpid());
        printf("Il pid del mio processo padre: %d\n",
                        getppid());
        return 0;
    } else if(pid < 0) {
        perror("Errore durante la fork");
        return -1;
    } else { // if(pid>0)
        // processo padre
        printf("La variabile pid = %d\n", pid);
        printf("Il mio pid: %d\n", getpid());
        printf("Il pid del mio processo padre: %d\n",
                        getppid());
        return 0;
    }
}
```

3.3 exit()

exit - Causa la terminazione *volontaria* di un processo.

Un processo può terminare involontariamente, se si tentano azioni illegali o se il processo viene interrotto tramite *segnale* (*kill()*), o volontariamente, se viene eseguita l'ultima istruzione o si esegue la funzione exit();