Creazione di processi

Per capire al meglio come funzionano i processi costruiamo quella che è una shell minimal. Cosa deve fare una shell:

- Attendere che l'utente digiti un comando.
- Avviare un processo per eseguire un comando.
- Attendere che il processo sia terminato.

Per fare questo dobbiamo utilizzare tre istruzioni principali:

- **fork**: Duplica il processo corrente, restituisce il PID del figlio nel chiamante(genitore) e restituisce 0 nel nuovo processo.

```
pid_t fork(); //pid_t tipo di dato per rappresentare il PID
//Restituisce valore negativo in caso di errore
```

- **wait**: Attende che i processi figli cambino stato, questo verrà scritto in **wstatus**, può causare un exit o un segnale.

```
pid_t wait(int *wstatus);
```

Programma in linea generale che usa le funzioni insieme:

```
void main(void)
{
        int pid, child_status;
        if(fork() == 0){
                do_something_in_child();
        } else {
                wait(&child_status); //wait for child
        }
}
```

- **execv**: Ci permette di caricare un nuovo binario nel percorso corrente, cioè **il programma in esecuzione viene rimpiazzato** da un nuovo programma, mantenendo lo stesso **PID** (identificatore del processo), ma con un nuovo codice in esecuzione.

```
int execv(const char *path, char *constargv[ ]);
//Constarg contiene gli argomenti del programma
```

```
//L'ultimo argomento è NULL
//Ne esistono diverse varianti
```

Tramite questi comandi possiamo costruire una shell estremamente minimale:

```
while (1)
{
        char cmd[256], *args[256];
        int status;
        pid_t pid;
        read_command(cmd, args); /* reads command and arguments from command
line */

        pid = fork();

        if (pid == 0) {
                execv(cmd, args);
                exit(1);
        }
        else{
                wait(&status);
        }
}
```

Adesso ci tocca trovare un modo per gestire i segnali, altrimenti non potremmo mai essere in grado di svolgere alcune funzioni come l'interruzione di un programma.
Per far ci che il segnale inviato sia catturato dobbiamo installare un gestore di segnali (**Signal Handler**).
Di seguito alcune funzioni utili che si trovano all'interno delle librerie standard C per implementarlo:

- **sighandler_t signal**: Consente di impostare un gestore di segnali per un segnale specifico

```
sighandler_t signal(int signum, sighandler_t handler)
//signum è il numero di segnale da gestire
//handler è la funzione da eseguire quando viene ricevuto signum, può essere
un puntatore ad una funzione di tipo sighandelr_t.
```

Esempio di utilizzo:

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

```c
#include <unistd.h>

void handler(int signum) {
    printf("Signal %d received\n", signum);
}

int main() {
    signal(SIGINT, handler);  // Gestore per il segnale SIGINT (Ctrl+C)
    while (1) {
        pause();  // Pausa in attesa di un segnale
    }
    return 0;
}
```

- **unsigned int alarm(unsigned int seconds)**: Consegna SIGALRM in un numero di secondi specificato.

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

void handler(int signum) {
    printf("SIGALRM received after 3 seconds\n");
}

int main() {
    signal(SIGALRM, handler);  // Impostiamo il gestore per SIGALRM
    alarm(3);  // Il segnale SIGALRM verrà inviato dopo 3 secondi
    pause();   // Pausa in attesa del segnale
    return 0;
}
```

- **int kill(pid_t pid, int sig)**: Invia il segnale sig al processo pid.

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
void alarm_handler(int signal){
        printf(“In signal handler: caught signal %d!\n",signal);
        exit(0);
}

int main(int argc, char **argv){
```

```
        signal(SIGALRM, alarm_handler);
        alarm(1); // alarm will send signal after 1 sec

        while (1) {
                printf("I am running!\n");
        }
        return 0;
}
```

Comunicazione tre processi attraverso pipe

Per permette a processi di comunicare fra di loro usiamo le seguenti chiamate:

```
int open(const char *pathname, int flags);
```

La chiamata di sistema `open` in Unix e nei sistemi operativi simili è utilizzata per aprire un file o un dispositivo. Si tratta di una funzione di basso livello fornita dal kernel che permette di accedere a file, dispositivi o altre risorse specificate nel pathname.

```
int close(int fd)
```

La chiamata close ci permette di chiudere il descrittore di file specificato in fd.

```
int pipe (int pipefd[2])
```

Crea una pipe con due fd per le sue estremità.

```
int dup(int oldfd)
```

Crea una copia del descrittore di file oldfd utilizzando il descrittore di file inutilizzato con il numero più basso per la copia, servono a chiamate di sistema che duplica i descrittori dei file. Li usiamo perchè permettono una gestione più avanzata dell'input/output.

Ricorda sempre di chiudere i file per i seguenti motivi:

1. **Evitare Blocchi** Chiudi la fine di lettura della pipe per impedire al processo di scrittura di rimanere bloccato.
2. **Ricezione EOF** `sort` aspetta un EOF per terminare la lettura. Chiudi la fine di scrittura per inviare un EOF a `sort`.
3. **Evitare Letture Accidentali** Nel processo `cat`, chiudi la fine di lettura per evitare letture inaspettate dalla pipe.

4. **Reindirizzamento di `STDOUT` in `cat`** Dopo la duplicazione del file descriptor, chiudi il descriptor originale per garantire che `cat` scriva solo nella pipe.
5. **Reindirizzamento di `STDIN` in `sort`** Dopo la duplicazione del file descriptor, chiudi il descriptor originale per assicurarti che `sort` legga solo dalla pipe.
6. **Descriptor nel Processo Padre** Dopo il `fork()`, il processo padre dovrebbe chiudere entrambe le estremità della pipe.