Indice

Appunti di sistemi operativi

I processi

Il sistema operativo mette a disposizione dell'utente (un programma in esecuzione) una macchina virtuale che astrae la macchina fisica creando un ambiente in cui ogni programma si vede come l'unico in esecuzione senza interferenze esterne.

Questa parallelizzazione viene realizzata attraverso i processi, ossia degli esecutori completi che eseguono i vari programmi. Ogni processo può essere visto come una macchina virtuale a disposizione del programma. I processi vengono parallelizzati dal sistema operativo. Un sistema operativo con questa capacità è detto multiprogrammato o multitasking.

I processi non sono esclusivamente usati dagli utenti, essi infatti possiedono due modalità di esecuzione:

- supervisore: riservata al sistema operativo
- utente: riservata all'utente

Anche il sistema operativo internamente usa dei processi. Questi processi hanno accesso diretto al processore in quanto sono eseguiti in modalità supervisore.

Caratteri generali di un processo

Ogni processo, tranne il primo, viene creato da un altro processo, in gergo è figlio di un processo padre. Ogni processo è identificato da un Process ID (PID) univoco.

La memoria di ogni processo è divisa in diverse parti dette segmenti:

- il segmento di testo (codice)
- il segmento dati: contiene tutti i dati sia statici che dinamici; quelli dinamici si dividono in allocati sullo stack o sulla heap
- il segmento di sistema: contiene i dati non gestiti dal programma ma dal sistema operativo, come ad esempio la tabella dei file aperti

Il sistema operativo fornisce a servizio delle applicazioni dei servizi di sistema per la manipolazione dei processi.

Chiamate di sistema per la gestione dei processi

1. Creazione di un processo figlio: `pid_t fork(void)`

   Crea un processo figlio identico al processo padre (all'istante di `fork()`). L'unico valore diverso tra i due è il PID (Process ID). La chiamate restituisce al processo il padre il PID del figlio e al processo figlio 0.

   Se la creazione del processo fallisce, viene restituito -1.

2. Terminazione di un processo: `void exit(int)`

   Termina il processo e restituisce un codice al processo padre.

3. Conoscere il proprio PID: `pid_t getpid(void)`

4. Aspetta fino alla terminazione del figlio: `pid_t wait(int*)`

   L'esecuzione del padre viene sospesa finché non termina il figlio il cui PID è il `pid_t` restituito. Il valore di ritorno moltiplicato per 256 viene salvato nel puntatore passato come argomento. Se i figli sono più di uno, la `wait()` aspetta un figlio qualunque. Se devo aspettare un figlio in particolare bisogna usare `pid_t waitpid(pid_t, int*, int)` (il terzo parametro si chiama `options` e lo considereremo maggiore di 0).

   Se un processo figlio termina quando il processo padre non è ancora arrivato alla `wait` esso terminerà ma non "morirà" in quanto esso deve restituire il codice di uscita al padre. Un processo in questa situazione viene detto zombie. Quando il padre chiamerà la `wait` l'esecuzione non si fermerà e il processo zombie verrà terminato definitivamente.

5. Cambiare il codice del programma: `int execl(char*, char*, ...)`

   Sostituisce il segmento dati e il segmento codice del processo con quello di un altro programma. La prima stringa di parametri è il percorso del programma da mandare in esecuzione, seguono N stringhe che specificano gli argomenti da passare a questo nuovo programma. Il primo argomento deve essere il nome del programma, mentre l'ennesimo deve essere `NULL`.

Proviamo a scrivere un semplice programma che usa queste chiamate di sistema:

```c
#include <stdio.h>
#include <sys/types.h>

int main(int argc, char **argv) {
  pid_t pid, pidr;
  int ret_status = 0, stato_exit;

  // N.B.: l'ordine di esecuzione tra padre e figlio non è definito
  pid = fork();

  if (pid == 0) {
    printf("Sono il figlio con PID %d\n", getpid());
    ret_status = 1;
    exit(ret_status);
  } else {
    printf("Sono il padre\n");
    pidr = wait(&stato_exit);
    exit(ret_status);
  }
}
```