Oltre von Neumann

> “Storiella”

Come abbiamo già visto lo scorso anno, i moderni sistemi operativi sono **multitasking**, cioé in grado di lanciare ed eseguire più compiti in contemporanea, alternando la loro esecuzione nella CPU; sono però anche **multithread**, cioé in grado di gestire più linee di esecuzione all'interno di un unico processo. Vedremo ora come entrambi gli aspetti assumono sempre più importanza in un discorso di efficienza generale del sistema, in particolare nel caso di sistemi multicore.

I processi

Come già sappiamo, i processi sono programmi in esecuzione: essi vengono creati, solitamente, da un'operazione dell'utente (tramite CLI o GUI). Ora vedremo con maggiore dettaglio cosa accade nel caso di creazione di un processo, nel caso Windows e nel caso Linux/Unix.

Creazione un un processo (modello Linux/Unix)

Il modello dei processi di UNIX è fortemente gerarchico: ogni processo è sempre creato da un altro, chiamato processo genitore[1]: questa impostazione comporta effetti importanti sulla gestione dell'intero sistema.

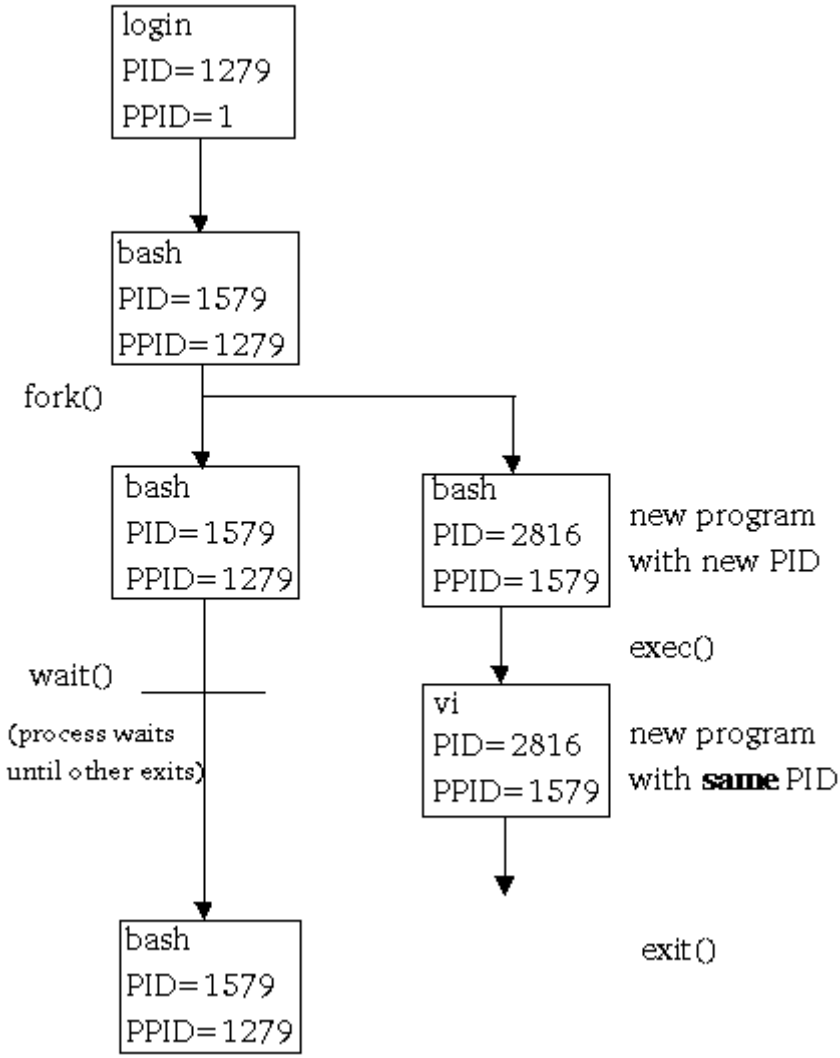
La creazione di un processo avviene in due fasi distinte, richiamando due funzioni di sistema, **fork()** ed **exec()**. Per capire come funziona, vediamo cosa accade in un caso concreto: un utente al terminale che vuole eseguire un programma (per esempio gedit). Ecco cosa accade

1. **L'utente scrive “gedit”.**
   La shell capisce che l'utente vuole lanciare un nuovo programma.

1 Fa eccezione il processo “init”, generato dal kernel al momento del boot.

2. **Fork**
   Il sistema genera una copia del processo in esecuzione, copiando nel nuovo processo il codice in esecuzione, i diritti, i descrittori dei file aperti. I due processi si distinguono solo per il proprio pid e quello del genitore.
3. **Exec**
   Mentre il processo genitore viene solitamente posto in attesa, il processo figlio carica il nuovo programma e lo sostituisce al proprio codice.

IMMAGINE DA RIFARE



Esaminiamo ora la cosa con un dettaglio maggiore: Il codice fa riferimento a due chiamate di sistema :

- `pid_t fork()` Duplica il codice e restituisce il pid (process id) del figlio.

- `int execl(const char *path, const char *arg, …)` Sostituisce al codice in esecuzione quello presente nel file path, e passa al codice un numero variabile di argomenti. Il path deve essere sempre un percorso ASSOLUTO; il primo argomento deve essere il nome del programmi, mentre il numero di parametri che seguono è del tutto arbitrario, ma deve terminare con NULL. Dato che sostituisce completamente il codice, questa funzione non tornerà mai indetro nulla – tranne nel caso in cui si verifichi un errore.

Un problema interessante, perfino filosofico, è il seguente: se fork() duplica ogni aspetto del processo, come è possibile sapere se quello che è attualmente in esecuzione è il padre o il figlio? Potrebbero andare in conflitto, e anche il sistema operativo avrebbe difficoltà a distinguerli.

Video: https://www.youtube.com/watch?v=BXLppLDIbcI

Didascalia:Ecco cosa succede quando due soggetti identici interagiscono (Film: Il sesto giorno)

Per fortuna, una piccola differenza permette di stabilire chi è chi, in modo analogo a ciò che accade nel film.una piccola differenza c'è, in modo analogo a quanto accade ad Arnold Schwarzenegger ne "Il sesto giorno": nel caso del padre, il valore di ritorno della funzione fork() è il pid del figlio; nel caso del figlio conterrà invece uno 0. Controllando quindi il valore di ritorno il programma può prendere due strade diverse.

Esempio:

Un esempio elementare di uso del costrutto fork-exec:

```
#include <stddef.h>
  #include <stdlib.h>
  #include <unistd.h>
  #include <sys/types.h>
  #include <sys/wait.h>
  int main () {     {
```

```
       int status;
       pid_t pid;

       pid = fork ();
       if (pid == 0)
         {
           /* Questo è il figlio, che lancia gedit */
           execl("/usr/bin/gedit", "gedit", NULL);
           _exit (EXIT_FAILURE);
         }
       else if (pid < 0)
         /* Fork fallito.  */
         status = -1;
       else
         /* Questo è il genitore, che aspetta che il
figlio termini */
         if (waitpid(pid, &status, 0) != pid)
           status = -1;
       return status;
     }
```

`waitpid` è una funzione che mette in attesa il programma in attesa di un evento da parte del figlio, e ne riceve il valore di uscita (quello fornito come valore di ritorno dal main).

Delle funzioni *`exec`* e *`wait`* esistono moltissime varianti, che non esponiamo per motivi di spazio. Basta scrivere `man exec` o `man wait` per ottenere ulteriori dettagli

Demoni, Zombie e Orfani.

Il modo Unix è ricco di terminologie colorate, che ha contribuito a dare al sistema un alone un po' mistico. Ad ogni modo, si tratta di casi interessanti.

Quando un processo genera un figlio,molto spesso deve consegnare il valore di ritorno del main al padre: per questo motivo, il PCB rimane in memoria – anche per parecchio tempo – finché il genitore non si risveglia e legge il dato. Il fatto che il processo sia "morto" ma sia ancora presente nella tabella dei processi ha generato il nome azzeccato di **processo zombie.**

Esempio:

se nel programma fork.exec appena usiamo il seguente codice:

```
pid = fork ();
if(pid==0)
    {
        exit(0);
    }
    else
    {
        while(1)
        {
        printf("Sono il genitore di uno zombie!\n");
        sleep(2);
        }
    }
```

Dato che il figlio termina immediatamente e il genitore non riceve mai i dati di ritorno del programma, otteniamo un processo zombie duraturo.

I **processi orfani** sono un po' il contrario degli zombie. In linea di massima, se un processo termina, in cascata fa terminare anche tutti i propri figli. In certi casi, però si può fare in modo che il processo orfano (in quanto ha perso il genitore)

sia “adottato” dal un altro, solitamente il processo di login dell'utente o il processo `init`. In quest'ultimo caso il processo resta attivo sino allo spegnimento del sistema.

```
* create child process */
 p=fork();

 if(p==0) {
    /* fork() returns Zero to child */
    sleep(10);
 }
 printf("The child process pid is %d parent pid %d\n", getpid(), getppid());
/*parent/child waits for 20 secs and exits*/
 sleep(20);
 printf("\nProcess %d is done its Parent pid %d...\n", getpid(), getppid());

 return 0;
}
```

```
nohup command-name &
```

La chiamata di sistema in questione è `nohup`.

I demoni invece sono processi che solitamente lavorano in background, attivandosi solo quando necessario: sono i programmi che forniscono i servizi più vari, spesso essenziali per il funzionamento del sistema. Tra i più noti troviamo smbd, dbus, named, e via dicendo.

> **Demone nel computer?**
> Il termine “demone” per un certo tipo di processi UNIX prende il nome del diavoletto di Maxwell, una creatura immaginaria che organizzava le molecole senza farsi vedere, e dalla terminologia greca, che indica una creatura a metà tra l'umano e il divino (in questo caso, forse è un intermediario tra i processi utenti e il kernel). In inglese, si può scrivere sia DEMON (pronunciato *dìmon*) o DAEMON (pronunciato *dèimon*).

Creazione di processi (Windows)

Il modello di creazione dei processi in Windows è radicalmente diverso: ogni processo è un'unità a se stante, senza relazioni con il processo che lo ha creato.

CONTROLLARE CODICE

```
#include <Windows.h>
#include <tchar.h>

#include <iostream>
using namespace std;

int main ()
{
    STARTUPINFO si = {};
    si.cb = sizeof si;

    PROCESS_INFORMATION pi = {};
    const TCHAR* target = _T("c:\\WINDOWS\\system32\\calc.exe");

    if ( !CreateProcess(target, 0, 0, FALSE, 0, 0, 0, 0, &si, &pi) )
    {
        cerr << "CreateProcess failed (" << GetLastError() << ").\n";
    }
    else
    {
        cout << "Waiting on process for 5 seconds.." << endl;
        WaitForSingleObject(pi.hProcess, 5 * 1000);
        /*
        if ( TerminateProcess(pi.hProcess, 0) ) // Evil
            cout << "Process terminated!";
        */
        if ( PostThreadMessage(pi.dwThreadId, WM_QUIT, 0, 0) ) // Good
            cout << "Request to terminate process has been sent!";

        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
    }

    cin.sync();
    cin.ignore();

    return 0;
}
```

Controllo di un processo (sospensione, risveglio, uccisione)

Caso Unix

Caso Windows

NOTA: Cos'è POSIX?

Comunicazione tra processi(IPC)

Sappiamo già che i processi sono entità del tutto indipendenti all'interno del sistema operativo, il che è ottimo per permettere ai processi di lavorare indipendentemente. Ma è molto meno utile se vogliamo che i processi collaborino e si scambino informazioni per cooperare. Ci sono tipicamente due metodi per farlo

- **Memoria condivisa**
  I processi interessati allo scambio di informazioni individuano una zona di memoria comune ai due processi. La memoria condivisa farà quindi parte di entrambi i processi ed è quindi possibile avere variabili condivise tra i processi coinvolti. Le difficoltà di questo approccio sono come individuare e condividere lo spazio, nonché problemi di sincronizzazione (vedi oltre)
- **Scambio di messaggi**
  In questo caso un agente esterno, parte del sistema operativa, si occupa di trasferire dati da un processo ad un altro. Il processo si limita a utilizzare una funzione o system call, solitamente chiamata send(), e il programma ricevente una funzione solitamente chiamata receive. In linea di massima questo sistema è preferibile per semplicità e flessibilità(), anche se meno immediato rispetto al precedente. Alcuni esempi d'uso di questo sistema