Processi e Thread

Thu, 20 Oct

Processi

L'obiettivo di un Sistema Operativo è quello di eseguire più **programmi** applicativi **contemporaneamente**.

Solitamente, sono molti di più i **programmi** da eseguire rispetto ai **processori** (*core*) fisici disponibili.

Per ovviare a questa limitazione **hardware**, il S.O. "crea" una macchina **astratta** che può eseguire virtualmente un numero illimitato di programmi, mettendo a disposizione un concetto astratto: il **processo**.

Programma vs Processo

Prima di proseguire, è necessario evidenziare la distinzione tra **programma** e **processo**.

- Un **programma** è un'entità *scarica;*
  È un insieme di **istruzioni** macchina che possono essere **eseguite**.
- Un **processo** è un'entità *attiva;*
  È un **costrutto** logico che **esegue** un programma.

Multiprogrammazione e Multitasking

Per rendere possibile l'esecuzione **contemporanea** di più processi rispetto al numero di processori disponibili, ogni **processo** in esecuzione ha l'illusione di evere una CPU "*tutta sua*".

A questo fine, vengono implementate teniche dette di **multiprogrammazone** e **multitasking** (*time-sharing*).

Multiprogrammazione

Per rendere possibile la **simultanea** esistenza di tanti processi, il S.O. mantiene in zone di **memoria** separate tra loro tutti i dati relativi ai **processi**, e quando la **CPU** si libera, essa viene **assegnata** ad uno dei processi in **attesa** presenti in memoria.

Multitasking (*Time-Sharing*)

La CPU è effettivamente in grado di eseguire tanti processi quanti sono i suoi **processori** (*core*), per cui, per poter eseguire **più** processi di quanti sono i processori fisici, è necessario **alternare** la CPU tra i vari processi, assegnando il processore ai vari processi in base a dei **criteri**.

Uno di questi criteri è basato sul **tempo di utilizzo**: ogni processo può usare la CPU per un **limitato** periodo di tempo (ad esempio, $1ms$), e quando finisce il suo tempo, il processo viene interroto e la CPU assegnata ad uno degli altri processi in **attesa**.

Operazioni sui Processi

La creazione dei processi avviene in modo **ricorsivo**. Questo significa che per creare un processo, è necessario avere un altro processo padre che lo generi.

Durante la fase di **boot** del Sistema Operativo, si creano I processi detti "**primordiali**" e dopo di loro, a *cascata*, si creano tutti gli altri processi.

Esistono **istruzioni** ed **API** apposite per gestire la creaztione (e la cancellazione) dei processi.

Processi in un sistema POSIX

In ambiente POSIX, ogni processo è identificato da un codice, detto **PID** (*Process Identifier*), che lo identifica univocamente. Il **primo** processo avrà PID uguale a 1.

Come detto in precedenza, i processi formano una struttura **gerarchida** ad **albero**, dove ogni processo, tranne il primo, ha un processo **padre**.

È dunque importante definire la **relazione** tra processo **padre** e processo **figlio**, e definire una politica di **condivisione** delle risorse tra essi. Un processo figlio viene inizialmente creato come **copia**, **clone**, del processo padre che lo richiama.

In POSIX, non esiste la **terminazione a cascata** dei processi: dunque, eliminando un processo padre, i suoi **figli non** vengono eliminati. Per questo motivo possono crearsi, durante l'esecuzione, processi detti **Orfani**, ovvero processi figli senza più un processo padre.

Esiste anche il concetto di processo **Zombie**, ovvero processi figli **terminati** il cui **valore di ritorno** non è stato acquisito dal processo padre.

API POSIX per Gestire i Processi

- `fork()`
  Crea processo figlio duplicato del processo padre; questo nuovo processo viene eseguito in concorrenza al padre, e inizia l'esecuzione dall'istruzione successiva alla `fork()`. Ha come valore di ritorno il PID del processo figlio.
- `exec()`
  Sostituisce il programma correntemente in esecuzione da un processo che ha chiamato la `exec()`, con un altro programma.
- `wait()`
  Funzione chiamata dal processo padre per attendere la chiusura del processo figlio. Ha come valore di ritorno il PID del processo figlio terminato.
- `exit()`
  Termina il processo corrente, esso rilascia le sue risorse in suo e viene e eliminato dal Sistema Operativo.
- `abort()`
  Funzione chiamata dal processo padre per terminare forzatamente il processo figlio.