1 Lezione del 18-03-25

1.1 Time-sharing

Abbiamo accennato al funzionamento dei calcolatori in modalità *batch*, dove più programmi vengono eseguiti in sequenza, uno dopo l'altro.

Un paradigma sicuramente più piacevole per l'utente, e più diffuso al giorno d'oggi, è quello del **time-sharing**, dove il processore dà l'illusione agli utenti di portare avanti più attività contemporaneamente, mentre il tempo della CPU è in verita diviso in frammenti temporali ridotti dove si dedica a ogni attività singolarmente.

Il meccanismo stesso della protezione che abbiamo introdotto alla lezione precedente serve appunto a difendere i programmi l'uno dall'altro in caso di esecuzione "parallela" (da non confondere col *multithreading*). Infatti, anche se è un concetto nato nei *mainframe* a uso pubblico, la protezione si è subito diffusa anche nelle macchine personali degli utenti, in modo da difendere non più programmi di diversi utenti ma più programmi dello *stesso* utente, magari soggetti a bug che potrebbero corrompere lo stato di altri programmi o dell'intero sistema.

Oggi il meccanismo di protezione si trova in tutti i calcolatori moderni, dai telefoni cellulari ai supercomputer, ed è risparmiato solo nel caso dei microcontrollori più semplici.

La domanda che ci poniamo adesso è quindi quella di *come* realizzare un sistema capace di dare quest'illusione dell'esecuzione "parallela" di più programmi, che avevamo introdotto all'inizio del corso come **multiprogrammazione**.

1.1.1 Processo

Chiamiamo **processo** un programma in esecuzione. Ciò che vorremo eseguire in parellelo sono, più propriamente, non programmi ma *processi*.

Intendiamo quindi un processo non come il codice che definisce un programma, ma come il programma stesso una volta che viene messo in esecuzione nel calcolatore, quindi tutti gli stati di elaborazione (disposti nel tempo) del calcolatore nell'esecuzione di tale programma.

Il modo in cui andremo a definire il paradigma della multiprogrammazione è assumendo un processo come un insieme di operazioni **atomiche**, che possono essere interrotte al loro termine o prima del loro inizio, e che bastano insieme all'istruzione successiva del codice a determinare lo stato successivo di esecuzione del processo.

1.1.2 Contesto

Un altro concetto chiave nella multiprogrammazione sarà il **contesto** di un processo. Avevamo parlato di contesto in termini di protezine: adesso diamo un significato leggermente diverso. Ogni processo si aspetterà infatti di trovarsi nel *suo* contesto personale: le operazioni intaccheranno i suoi registri, che si aspetta essere l'unico a modificare, ecc... Il sistema operativo dovrà quindi essere in grado di fornire a ogni processo il suo contesto specifico.

Vediamo che questa idea si può tradurre già lato software. Il **cambio di contesto** può essere infatti effettuato, prendendo l'esempio dei soli registri, mantenendo una struttura data che contiene un'entrata per ogni registro. Al momento del cambio basterà copiare l'insieme dei registri corrispondenti al contesto di un certo processo nei registri veri e propri del processore.

Un discorso analogo sarà quella della memoria: ogni processo si aspetterà che al suo contesto corrisponda una sua copia della memoria. Possiamo mantenere un'altra struttura dati, simile a quella posta per i registri, che si occupa di mantenere informazioni riguardo alle regioni di memoria corrispondenti ad ogni contesto, e caricare quindi queste in una sezione dedicata su e della memoria stessa, o per semplicità su e dall'hard disk (così erano i primi sistemi time-sharing). Vedremo più nel dettaglio questo aspetto quando introdurremo la *memoria virtuale*.

Facciamo un ultima nota sulla *comunicazione* fra processi: nel caso più semplice, ogni processo non è al corrente dell'esistenza degli altri processi, e gestisce la sua *memoria privata*. Il sistema che studieremo dispone invece anche di una *memoria condivisa*, che permette ai processi di condividere informazioni fra di loro.

1.1.3 Kernel

Il programma che si occupa di effettuare queste operazioni di cambio di contesto si chiama **kernel** o *nucleo*. E' sempre in esecuzione in modo sistema e gestisce i contesti e le risorse assegnate ad ogni processo.

Immaginiamo quindi il kernel come un intermediario fra **processi** e **hardware**. Notiamo che questo non significa che kernel e processi sono *contemporaneamente* in esecuzione: questo è impossibile, in quanto la CPU è una sola. Kernel e processi sono infatti in esecuzione singolarmente, l'uno alla volta, e l'unico modo in cui si restituisce il controllo al kernel da un processo è attraverso i 3 tipi di interruzioni:

- Interruzioni esterne (dai dispositivi);
- Eccezioni (errori e altri malfunzionamenti, non necessariamente dati da errori di programmazione);
- Interruzioni interne (sollevate dall'istruzione `INT`).