# Sistemi 1

Cosa fa “girare” i programmi? Dal punto di vista hardware è la CPU, che si limita al suo fetch, decode ed execute. La CPU si pesca la prossima istruzione da eseguire dalla RAM, seguendo l’indirizzo presente nell’IR.

All’inizio faremo finta di avere una sola CPU con un solo core di esecuzione.

Data la monotona vita della CPU, per sapere cosa avviene nel mondo esterno sono necessarie le interrupt (e le trap).

Nei sistemi reali, girano parecchi programmi per volta, quindi come distingue la CPU quale programma sta facendo girare? Inoltre chi porta i programmi in RAM? Perché non ci poniamo problemi di memoria noi da utenti quando avviamo un programma?  
Perché di tutto questo se ne occupa il sistema operativo. Infatti un software particolare, il kernel del sistema operativo si occupa di gestire e visualizzare le risorse della CPU:  
per esempio ogni processo “crede” di avere una cpu e una memoria tutta sua. Questa è una facilitazione per il programmatore, ma questo isolamento è anche per questioni di sicurezza (un crash non tirerà giù l’intero sistema).  
Inoltre i programmi non possono accedere direttamente all’hardware, onde evitare che qualunque utente possa modificare i dati di chi vuole. Per questo motivo il sistema operativo implementa i device-drivers.

Ogni processo è identificato da un numero, che si chiama pid, è un numero non negativo. Siccome non sappiamo quanto grandi sono gli interi che rappresentano i pid in ogni sistema, se li vogliamo stampare conviene castarli (quando li si prende) al tipo di intero più grande rappresentabile (=intmax\_t).

La virtualizzazione della CPU e della RAM e la convinzione dei processi di avere le risorse tutte per loro fa sì che le condividano senza rendersene conto (due programmi possono condividere l’indirizzo di alcune variabili nello stack, pur essendo diverse, perché tanto usano la CPU in momenti diversi).

In un certo senso, per ogni processo, il s.o. crea una macchina virtuale:

con le sue CPU, la sua memoria ecc.

alcune istruzioni vengono eseguite direttamente dalla vera CPU, sono le istruzioni degli handler delle system call.

Per virtualizzare la memoria, serve necessariamente dell’hardware capace di realizzare la paginazione, inoltre sono fondamentali le istruzioni privilegiate (e quindi avere almeno 2 stati della CPU).

Il sistema operativo deve essere robusto e affidabile, non dovrebbero esserci dei bug, ma talvolta può capitare che spuntino.

Nei primissimi sistemi si usava il batch processing: allora il sistema operativo era solo una libreria che serviva da isolare dall’hardware il driver, mentre il batch da eseguire sul mainframe era una scelta fatta da un operatore umano. Ciò era problematico dal punto di vista della protezione.   
Successivamente è stato introdotto il metodo per passare da modalità utente a quella privilegiata e i s.o. hanno preso il controllo.

Con l’avvento dei minicomputer è venuta fuori l’idea di sfruttare i tempi dei dispositivi di I/O (che erano più lenti delle CPU, le quali erano/sono parecchio costose) per alternare i programmi eseguiti dalla CPU e renderne l’esecuzione complessiva più efficiente.

Negli anni 60 naque UNIX, il sistema operativo Semplice, ma Potente. Scritto in C (anziché in assembler)\* e i cui sorgenti erano distribuiti praticamente gratis alle università (facendo in modo che si diffondesse) fino al 1984. Nota: la parte più di basso basso livello va comunque scritta in assembler (ad esempio come formattare le tabelle di paginazione), ma consiste di una percentuale minuscola del codice.

A partire da quei sorgenti, diversi distributori hanno sviluppato le proprie versioni: oggi UNIX è un marchio registrato, ci sono però molti sistemi come UNIX (funzionano allo stesso modo ma non c’hanno il nome) che sono gratis e open-source, come Linux.

\*l’assembler era problematico perché dipendeva dalla CPU (ogni CPU aveva il suo “dialetto”), il C invece era molto più multipiattaforma, poiché si compilava facilmente in tutti i tipi di assembler.

Ad oggi, anche se windows ha vinto sul fronte desktop, linux è molto usato per i server ed ha vinto su tutta la linea su mobile (il sistema android ha un kernel linux).

Noi ci affacceremo allo standard POSIX.

Sistema operativo didattico: <https://github.com/zxgio/xv6-SETI>

Nei sistemi UNIX-Like:

La shell è un interprete di comandi, che si può usare sia modalità interattiva, sia tramite degli script.

Il nome shell deriva dal fatto che si tratta di un programma utente che offre un’interfaccia ad alto livello alle funzionalità del kernel (è un guscio attorno al kernel)

Con il termine sistema operativo si può intendere sia il kernel che tutta una serie di programmi (servers) che ci aspettiamo di trovare. In questo corso, tuttavia, siamo più interessati al kernel.

Noi faremo riferimento alla GNU bash, la Bourne-Again Shell, un gioco di parole sul fatto che la prima shell per UNIX si chiamava SH, e il suo autore Stephen Bourne.

Per interagire con una shell bisogna usare un terminale: un dispositivo meccanico dove un operatore digita su una tastiera e l’input viene mandato al sistema, che si occupa di produrre l’output. In UNIX, (quasi) tutto è un file e anche un terminale è visto come un file. Nel mondo moderno non si usano più questi terminali, in Linux ci sono le console virtuali (ttyn), sono terminali virtuali che condividono tastiera e schermo. Si può cambiare quella attiva premendo ALT+Fn.

All’esempio degli ambienti desktop si usa piuttosto un emulatore di terminale, essi sono implementati tramite pseudoterminali (pty), sono una coppia di processi master slave, il master scrive sullo slave che si comporta come un terminale effettivo.

Ogni processo ha tre file descriptor (stdin, stdout, stderr). Lo Stderr è un fd a sé stante perché così non c’è il rischio di non notarlo quando l’output viene rediretto verso un file (infatti non è bufferizzato a differenza dello stdout, dunque ce ne si accorge subito).

C’è un file speciale: /dev/tty. Questo file speciale corrisponde al terminale che si sta utilizzando.

Se si usa il comando tty si può vedere quale terminale in particolare è quello in uso (e si può usare il risultato per mandare info ad altri terminali).

Col comando reptyr -l si può aprire un nuovo terminale.

Con cat si può leggere da stdin e stampare su stdout (oppure specificare file da cui prendere input o output). Tuttavia usando il percorso di altri terminali si può fare in modo che l’input sia preso da un terminale e l’output sia stampato su un altro.

Il carattere ‘|’ si chiama pipe (che si può mettere per dire riesegui il comando una seconda volta mettendo nel secondo file l’output con l’input del primo) e lo si può far seguire da “tee”.

/proc ci permette di osservare i processi tramite il loro PID come directory (con proc/PID si può aprire la directory del processo).   
Con proc/PID/fd ci mostra a cosa puntano i file descriptors.

Ps va a guardare dentro /proc, perché in realtà Linux tutto è visto come un file.