INFORMATICA - *Crash course* sui sistemi operativi

(prof. Capolupo)

Sistemi operativi: cosa sono e a cosa servono

Definizione. Un Sistema Operativo (S.O.) è un programma in grado di gestire le risorse (memoria, CPU, ecc.) di un dispositivo (PC, smartphone, tablet, server, …). Gestire le risorse significa, in altri termini, sfruttare le risorse hardware al fine di esporre o fornire un insieme di servizi all'esterno. Il SO gestisce, per esempio: la memoria, i dispositivi di I/O, la rete… al fine di renderle disponibili per l'operatività dell'utente.

Lo scopo del sistema operativo è quello di **condividere e gestire le risorse**, *in primis*, ma anche quello di mettere a disposizione le cosiddette **macchine virtuali**. Grazie al S.O. posso proteggermi da errori e minacce informatiche, oltre a poter gestire situazioni di conflitto (per es. due utenti che vogliono accedere alla stessa area di memoria, ad esempio).

Vedremo pertanto come funziona il mondo dei sistemi operativi e come gestisce, in generale questo genere di situazioni.

Definizione: le macchine virtuali

L'uso dei sistemi operativi è legato al mondo delle macchine virtuali, che sono una forma di astrazione utilizzata per garantire un accesso regolamentato e intelligenze alle risorse. Il concetto di astrazione è tipico del mondo dell'informatica e consiste nell'individuare gli elementi generali o essenziali di un sistema al fine di facilitarne l'uso.

Le macchine virtuali sono una potente astrazione che evita all'utente di doversi programmare in proprio periferiche e dispositivi come tastiere, sensori, mouse, schermi e così via. In generale distingueremo tra un approccio **ad alto livello**, tipico dell'astrazione, in cui definiremo opportune *handle* (maniglie) per gestire il sistema, ed uno **a basso livello**, che invece interessa i progettisti e chi lavora a stretto contatto con l'hardware, fino ad arrivare al livello fisico fatto di codifica binaria. Mediante il processo di virtualizzazione posso sia gestire le risorse del sistema che emulare il comportamento di una macchina, senza avere necessariamente bisogno di disporne fisicamente.

Esempio. Con la virtualizzazione del servizio offerto da Microsoft Azure, ad esempio, posso avere a disposizione un database SQL molto capiente a cui potrò accedere mediante internet,

senza vedere fisicamente alcuna macchina.

Con una macchina virtuale è possibile allocare (mettere da parte, riservare) aree di memoria per usi specifici; d'altro canto permettono all'utente di accedere ai file di sistema (cosiddetto *file system*).

Grazie alle macchine virtuali posso, inoltre, usare più programmi contemporaneamente dentro un sistema operativo che viene appositamente **emulato**: es. scrivere un *doc* Word, inviarlo per posta, aprire *google.it* su un *browser*. Non solo, quindi, ho la possibilità di gestire più programmi, ma posso aprire dentro una finestra di un sistema Windows un'istanza virtuale di Linux che definirà il proprio ambiente di lavoro, con i propri rispettivi programmi.

Nota. In generale, un file dentro ad un S.O. può essere 1) un programma eseguibile oppure 2) un dato informativo (documento, immagine, …). Distinguiamo inoltre tra programmi di sistema (il cui funzionamento è riservato al S.O.) e programmi utente utilizzabili dall'operatore umano.

Terminale di comando

La linea o terminale di comando (terminale) *è un programma testuale che consente all'utente di operare sul computer, a livello di file system. Ogni comando sarà importito mediante una singola linea di testo seguita dal tasto invio, nel formato classico:*

comando parametro1 parametro2 … parametroN

Introduzione al file system

Il *file system* è una struttura dati del S.O. adibita a memorizzare contenuto e posizione di ogni file all'interno del sistema operativo. In un file system siamo soliti distinguere un livello cosiddetto *root* (radice) dal quale si dipanano vari sotto-livelli, che possono essere file (di dati o di programmi) oppure *directory* (altro nome per le cartelle o "contenitori" di file).

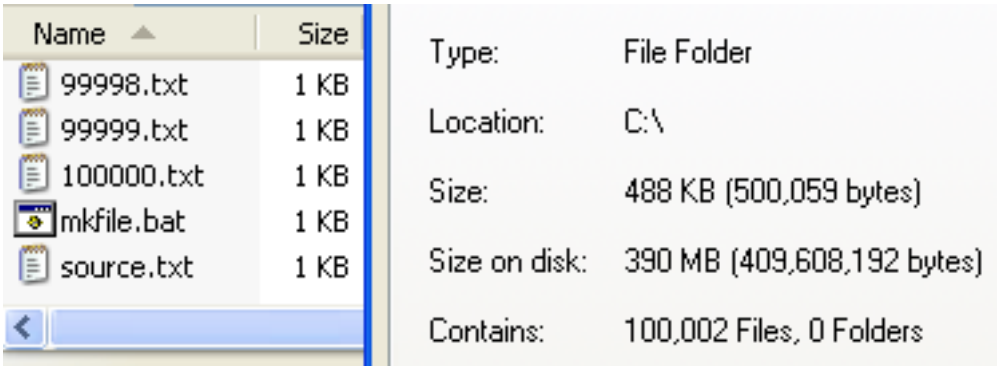
Livello root del file system

In un S.O. Linux esiste un livello *root* (rappresentato dal simbolo \) e varie *directory* funzionali ad uno scopo specifico: troviamo ad esempio la cartella *bin* che contiene solo i file

binari o “eseguibili”, *etc* che contiene le informazioni gestite dal sistema, *mnt* che elenca tutte le periferiche connesse, *boot* che gestisce le risorse da utilizzare in fase di avvio del sistema e così via. Su sistemi Mac OS questa struttura è utilizzata in forma analoga, mentre su quelli Windows la gerarchia del *file system* è organizzata sulla base di criteri differenti (troviamo un drive rappresentato da una lettera e un *path* (“cammino”) del tipo *C:\Programmi\prova\file.ext*).

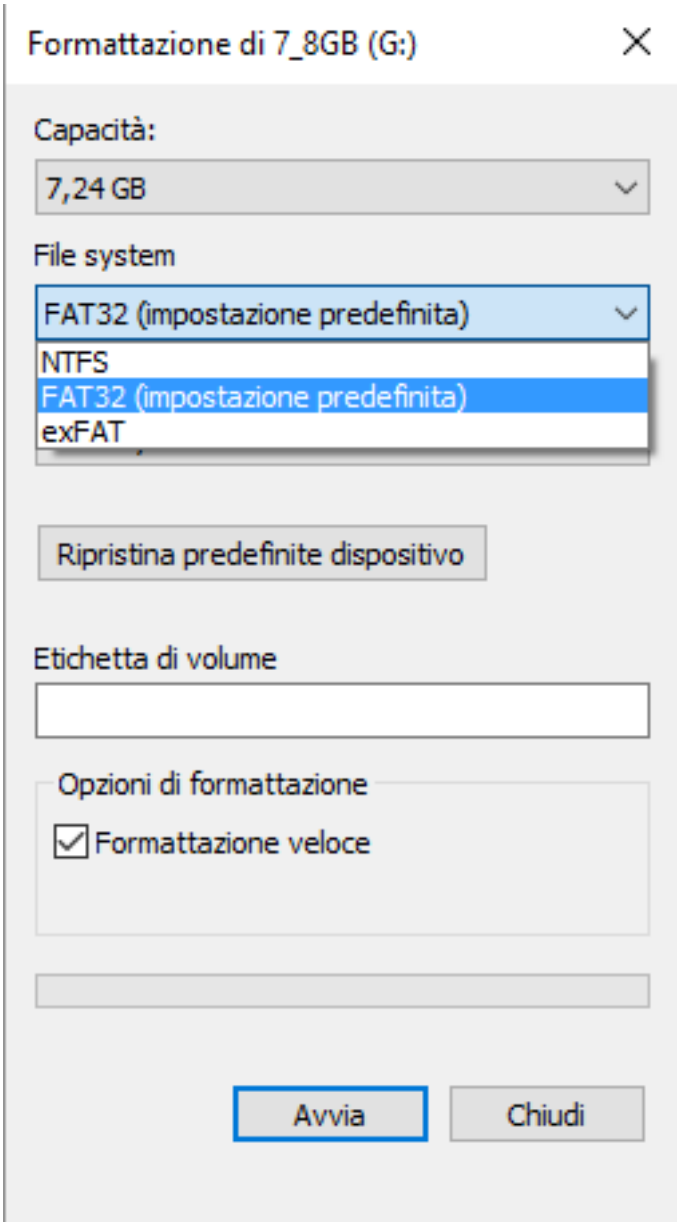
Di seguito possiamo vedere una tipica gerarchia del file system su un sistema Linux.

Quello della figura successiva, invece, è il tipico File manager (Gestione risorse) che rappresenta il livello di astrazione del file system Windows.



Cenni ai tipi di File System

Esistono due tipi di file system (FS), a livello generale, ed ognuno di essi supporta file con caratteristiche diverse: i FS **per memorie di massa** (localizzati su un solo dispositivo di massa) e quelli **per memorie distribuite** (distribuiti su più dispositivi connessi tra loro in rete).I file system per memorie di massa sono tipici di hard disk drive (HDD), memorie USB rimovibili, CD ROM e così via, mentre quelli per memorie distribuite possono “pescare” i file anche da supporti diversi, facendo uso della rete come tramite. Se i FS per memorie di massa si basano sulle astrazioni di file e directory, quelli distribuiti sfruttano anche il concetto di nodi o cluster.



Esempi pratici di *file system* del primo tipo sono ad esempio: NTFS (Windows), AFS (Mac OS), FAT/FAT32 (Windows), mentre un esempio del secondo tipo è GFS (Google File System). In base alle specifiche del file system in uso sarà possibile definire dove salvare i file, come recuperarli, la dimensione massima di un dato che possiamo salvare e le modalità con cui il S.O. più recuperarle.

Esempi. *Con un file system di tipo FAT32, tipico dei sistemi Windows più datati e di alcuni dispositivi USB rimovibili, possiamo gestire singoli file di dimensione massima di 4GB, con capienza complessiva fino a 2TB, con il vantaggio di disporre di alta compatibilità, dato che possiamo usare un dispositivo di questo tipo su qualsiasi S.O.*

Con un file system NTFS, al contrario, potrò gestire singoli file molto più grandi, dai 16 TB fino ai 256 TB, disponendo di una capienza complessiva molto più grande. Le specifiche dettagliate sono disponibili nella documentazione di questi prodotti e, in genere, non sono uniformi: lo stesso tipo di file system può, in genere, supportare N caratteristiche.

AFS e ext2 sono tipici dei sistemi Apple e Linux, e funzionano solo su quei S.O.

Dal programma al processo

Ogni S.O. (windows, Mac OS, Linux, …) è caratterizzato da una propria politica di gestione delle risorse, ovvero definisce il numero di attività che può eseguire nell'unità di tempo, cercando di ottimizzarla e di effettuare più attività possibili. Le risorse, ovviamente, sono sempre limitate (molto spesso per motivi di costi), e non dovranno essere né sotto-utilizzate né tantomeno saturate. L'idea generale da perseguire quando viene progettato un S.O. è che **si possano sfruttare le risorse il meno che si può, al fine di ottenere il massimo delle prestazioni.** Questo è possibile mediante alcune opportune politiche di gestione.

Definiamo "processo" un programma (utente o di sistema) nel suo periodo di attività. Un programma in esecuzione viene pertanto chiamato processo; ogni processo è - a sua volta -composto da almeno un thread *("filo"). Il task manager di Windows, ad esempio, è il gestore dei processi a livello utente (si usa CTRL ALT CANC per visualizzarlo), mentre in Linux usando il comando* top *da terminale potrò vedere sia i processi utente che quelli di sistema attivi in quel momento.*

In informatica l'esecuzione di un processo comporta, di base:

- L'occupazione temporanea di un'area di memoria RAM, a seconda delle politiche di allocazione del S.O.;
- L'uso di cicli di CPU, nel momento in cui il processo abbia bisogno di effettuare calcoli o elaborare dei dati

Dal "punto di vista" del sistema operativo, a questo punto, è desiderabile che i processi siano molto numerosi e possano essere eseguiti in contemporanea, senza attese inutili che minerebbero l'operatività e ne rallenterebbero l'uso

Definizione. A tal proposito si definisce THROUGHPUT (produttività) il numero di processi eseguiti nell'unità di tempo.

Di seguito ho elencato le differenze tra i tre principali sistemi operativi in termini di diffusione, licenza d'uso, programmi disponibili, prezzo e quote di mercato.

	Windows	**Mac OS**	**Linux**
Diffusione	ALTA	MEDIA	MEDIO BASSA
Licenza dei software disponibili	Proprietaria (in prevalenza)	Proprietaria (in prevalenza)	Open source (sempre)
Programmi	Ampia diffusione	Molto diffusa in certi settori (ad esempio grafica e montaggio video)	Diffusione specifica su dispositivi customizzati, soprattutto
Quote di mercato (2019)	91%	7%	2%
Prezzo	Medio-Alto	Alto	Basso (gran parte dei software open source sono anche gratuiti)

Altri sistemi operativi sono MS-DOS (funziona con un terminale di comando, anche prima dell'uso di Windows, della Microsoft), Windows Phone, Android (basato su kernel Linux), iOS (per iPad e iPhone).

Esercizio: esecuzione monotask vs multitask

Per imparare a quantificare le prestazioni di un sistema può essere agevole ricorrere ad un esempio. Siano A, B, C 3 programmi in esecuzione (processi) nel S.O., e sia *t* il tempo misurato (ad esempio in millisecondi)

Per inciso. Ogni computer funziona al "battito" della frequenza di un core della CPU, caratterizzata da un ciclo di funzionamento suddiviso in più fasi ripetitive. Consideriamo per semplicità un ciclo di CPU (il "motore" della macchina) che sia composto da due sole

fasi: una fase di *fetch* (prelievo della prossima istruzione da eseguire, ammesso di averla caricata in memoria) ed una di *execute* (esecuzione dell'istruzione corrente).

Definizione. Chiamiamo ***esecuzione monotask*** *una sequenza di cicli CPU (fetch, execute) in cui viene eseguito un solo processo per volta. Nel grafico seguenti abbiamo rappresentato questa idea: il tempo è in orizzontale (ascissa) e il ciclo in verticale (ogni gradino è composto da fetch con segnale alto + execute con quello basso). Il "quanto"temporale è dato dal trattino orizzontale che divide due picchi consecutivi del grafico.*

Se sto usando un'esecuzione dei processi monotask significa che posso eseguire solo un processo alla volta, in alternanza e secondo varie politiche stabilite da S.O. Se sto gestendo il funzionamento dei processi A, B e C posso ottenere uno schema di esecuzione del genere, in prima istanza, dove ogni casella consuma un "quanto" di tempo (il tempo trascorso è indicato dal numero tra parentesi, per semplicità un numero intero):

A(1)	B(2)	C(3)	A(4)	C(5)	B(6)

Si assume per adesso ottimisticamente (nella realtà non è comune che succeda) che ogni processo finisca di lavorare entro il quanto di tempo definito dal sistema, garantendo così una sorta di "alternanza perfetta". Poniamo quindi che per ogni ciclo di CPU si esegua solo uno tra A, B o C, come avviene sui sistemi con MS DOS puro (senza Windows).

Contando il numero di cicli (che sono 6 in tutto) avremo impiegato complessivamente 6 * 0.1 ms = 0.6 ms eseguendo 2 volte ogni processo A, B e C.

Dal mono al multitask: concetto di interrupt

Se ad esempio le CPU diventassero due, di fatto, si passerebbe ad una esecuzione *multitask* (più di un task alla volta), visto che potrò parallelizzare l'esecuzione dei processi e lanciarli in coppia.

CPU1	A(1)	C(2)	C (3)
CPU2	B(1)	A(2)	B (3)

In questo caso il tempo di esecuzione complessivo è dimezzato, essendo T = 3 * 0.1 ms = 0.3 ms

Siamo quindi riusciti ad ottimizzare il *throughput* in modo intuitivo, ovvero aumentando il numero dei processori. Ovviamente nella pratica non sempre è possibile aggiungere processori: la tecnologia ha un costo e c'è un limite all'uso dei processori. Bisogna ricorrere a strategie di gestione specifiche da parte del S.O. che prevedano, ad esempio, che un processo si possa interrompere (*interrupt*) per poi riprendere in seguito.

Politica del time sharing

Avendo un numero di risorse limitate ed uno potenzialmente illimitato di processi, l'idea di suddividere l'uso del sistema operativo in *slot* temporali equivalenti sembra ragionevole: se il S.O. concede, per sempio, ad ogni processo lo stesso tempo di esecuzione, saremo sicuri che saranno tutti serviti e che nessuno rimarrà in starvation ("morire di fame", in gergo, senza essere mai serviti). L'idea è valida solo sotto condizioni stringenti: che tutti i processi siano identici o che, quantomeno, consumino le stesse risorse, cosa in genere irrealistica. Nella realtà un processo come Word non consumerà mai quanto un processo di grafica in 3D, ad esempio, né un videogame potrà consumare le stesse risorse di un gestionale.

Nella politica del time sharing, il S.O. mette a disposizione quanti temporali fissi ad ogni processo. Tuttavia questa scelta, da sola, non sembra sufficiente allo scopo di gestire in modo ottimale il sistema.

Concetto di *Context switch*. Lo *scheduling* o schedulazione dei processi.

Priorità dei processi e gestione della coda.

Se A e B chiedono contemporaneamente l'uso della RAM, ad esempio, il S.O. si occuperà di concederlo in momenti diversi, un po' per volta, finchè non avranno finito. La fase di passaggio da processo A al processo B viene detta *context switch*. La ripartizione delle risorse tra i processi concorrenti (perché concorrono all'uso delle risorse) viene detta anche *scheduling*.

Le forme di *scheduling* più diffuse puntano a massimizzare l'uso della CPU, ad esempio, oppure a minimizzare i tempi di attesa o quelli di elaborazione. In questa fase, oltre a misurare il *thoughput* (numero di processi che iniziano e finiscono le attività) possiamo semplicemente misurare il tempo che passa: dipende dal contesto, naturalmente, e dalle esigenze operative. Se il S.O. deve gestire un processo industriale e rispettare tempistiche fisse, ad esempio, potrà funzionare in real time, mentre nel caso delle operatività medie dovrà sfruttare un concetto interno o di tempo relativo o "soggettivo". È quello che succede in un videogioco di calcio, ad esempio, in cui i 90 minuti simulati della partita non si riflettono quasi mai in 90 minuti effettivi.

Il mondo dello scheduling è affascinante quanto complesso: esistono algoritmi come quello *Round-Robin* che cerca di accontentare tutti i processi e processarli nel minor tempo possibile. Esiste anche l'algoritmo detto *Shortest Next Process First* che ottimizza il tempo di attesa in base della durata di ogni processo: sembra ragionevole come idea, ed è stato anche dimostrato che trova sempre la soluzione ottimale. Il problema è che non è facile da applicare, se non in contesti specialistici: in genere, infatti, **non possiamo sapere** in anticipo quanto tempo occuperà ogni processo in esecuzione.

Una tecnica più smart e facile da applicare è quella dello **scheduling con priorità**: di base i processi vengono servizi nell'ordine di arrivo, un po' come avverrebbe con la coda in farmacia oppure al supermercato. se ho N processi attivi assegno una priorità ad ognuno (1 priorità massima, 10 priorità minima, ad esempio, che poi sarebbe il bigliettino con numero progressivo che ognuno di noi preleva all'arrivo). La CPU verrà assegnata volta per volta al processo con priorità di turno, ma con la possibilità di prevedere l'interruzione temporanea del processo (*interrupt*) in caso di necessità. L'interruzione non è un'operazione banale,

ovviamente, perché comporta che il sistema debba salvare una sorta di "istantanea" di quello che stava facendo il processo, per poi ripristinarlo all'occorrenza.

Concetto di pre-emption. Se ad esempio sta eseguendo il processo A con priorità 4, arriva il processo B con priorità 1 e prende, di norma, il posto di A. Prima di farlo, pero', il S.O. salva una "istantanea" del punto in cui era arrivato A dentro dei registri interni della macchina, serve B e poi ritorna a servire A quanto ha finito. Questo meccanismo di context switch viene detto pre-emption, ed è uno dei meccanismi più potenti presenti nei S.O.

Esercizio: latenza di rete su un processo attivo

Sia A un processo browser connesso alla rete internet; in questo caso, in media, sarà difficile stimare i tempi di esecuzione (quanto ci mette ad aprire una pagina web, ad esempio) per via delle latenze di rete, non prevedibili e non eludibili. Siccome connettersi ad una rete significa passare per vari dispositivi che possono presentare potenziali guasti (server, dorsali internet, router, modem, ecc.) disporre di una gestione *monotask* è problematico: il processo A potrebbe occupare la CPU a tempo indeterminato, impedendo ad altri processi futuri B, C, D, … di accedere, solo perché A sta "aspettando" la connessione di rete. B, C, D, … rimarrebbero in attesa molto a lungo: l'utente, nel caso di una latenza o ritardo di rete, vedrà il processo "congelato".

La soluzione può essere sfruttare una **politica pre-emptive,** che si "renda conto" dell'attesa eccessiva assegnando priorità ai processi in coda e sbloccando la coda, evitando di bloccare il sistema. I moderni processori o CPU sono sia *multi-thread* che *multi-core*: permettono di parallelizzare l'esecuzione dei processi attivi, ottimizzando il *throughput* con varie strategie, sia a livello *hardware* che *software*.

Nota. Il core è la parte fisica adibita al funzionamento della CPU, mentre multi-core significa poter eseguire più istruzioni in parallelo. Il thread è una sotto-sezione a sé stante nel flusso di esecuzione di un processo: pertanto un sistema operativo può far funzionare più processi concorrenti o in parallelo, e ogni processo può contenere uno o più thread adibiti alle varie operazioni.

Nota: processori Intel i5, i7, …

Processori Intel i3, i5, i7, i9 gestiscono fino a 3, 5, 7, 9 processi in parallelo.

Un processore Intel Core di 13ma generazione, ad esempio, è in grado di contenere 24 *core* distinti.

Introduzione allo stato di un processo

I processi attivi in un sistema operativo trasportano, al proprio interno, un insieme di campo che tengono traccia di vari aspetti della propria esecuzione. Ogni processo attivo nel sistema operativo viene pertanto identificato univocamente, in prima istanza, da un **PID** (*Process Identification Number*), tipicamente un numero intero progressivo 1, 2, 3, ….

Ogni processo ha il proprio PID e non sono ammessi duplicati nell'assegnamento degli stessi; in questo modo il S.O. potrà disporre di un modo sicuro per accedere all'i-esimo processo senza ambiguità.

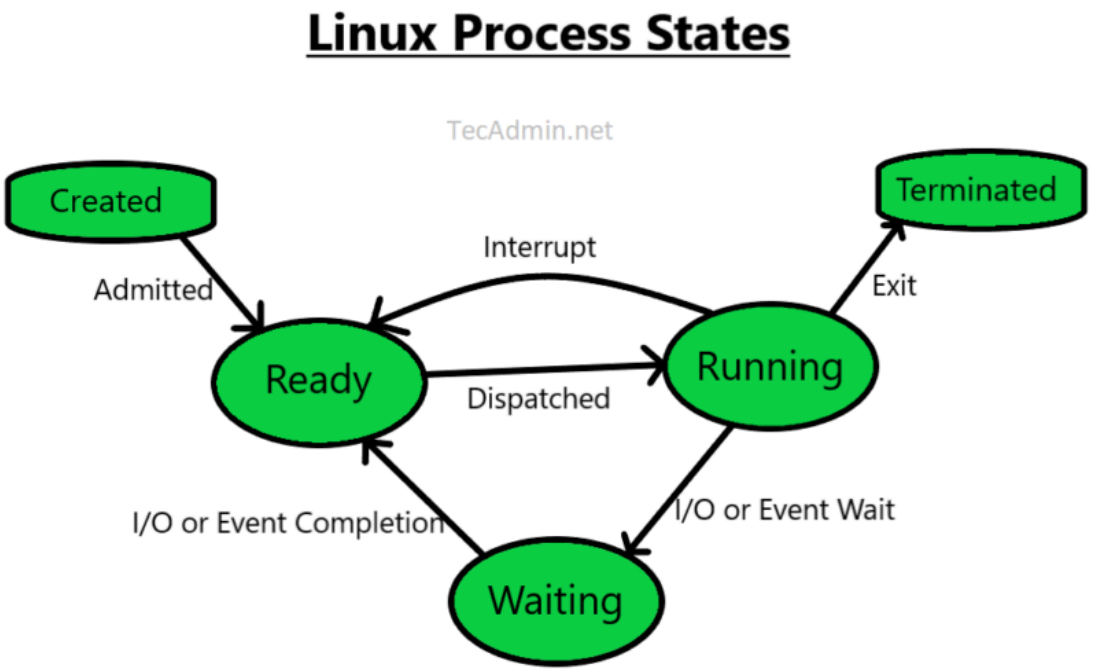
Con riferimento al S.O. Linux, nello specifico, i processi vivono un "ciclo di vita" in cui nascono, vivono e muoiono; il primo *status* da considerare è quello di creazione (*CREATED*), in cui il processo viene "generato" dal sistema operativo tipicamente su input dell'utente (che decide di aprire Word, ad esempio).

Il processo viene ammesso poi ammesso in una coda di processi, a questo punto (*Admitted*) ed è a questo punto pronto all'uso (*Ready*), occupando un'area di memoria RAM specifica e riservata: non appena qualcuno interagisce o fa una richiesta (ad esempio inizia a scrivere) il processo passa nello stato *Running*, ovvero diventa operativo e "consuma" cicli di CPU e aree di memoria.

A questo punto possono capitare tre circostanze:

1. il processo viene chiuso, per cui passa in stato *Terminated*
2. il processo viene interrotto (*interrupt*) e passa nuovamente in stato *Ready*;
3. il processo viene messo in attesa (*Waiting*), in attesa di un'evento I/O esterno (la connessione di rete, ad esempio) ed in questo caso dovrà aspettare il completamento dell'evento per poter essere di nuovo pronto all'uso.

Il ciclo di vita può ripetersi varie volte, ovviamente, finchè non si passa alla terminazione definitiva dell'evento, che libera così area di memoria e CPU dalle proprie incombenze.



Architettura di un computer: CPU, registri, memorie, I/O

È necessario a questo punto spiegare un po' di concetti tecnici per comprendere al meglio il funzionamento di un sistema operativo.

Per *risorse*, per quanto abbiamo detto, intendiamo risorse di calcolo o risorse di memorizzazione, quindi ad esempio memorie interne ed esterne, processori e/o periferiche. È bene ricordare a questo punto che un computer consiste di almeno un processore (CPU), una o pi memorie, varie periferiche di I/O; più nel dettaglio la CPU è suddivisa in ALU

(Arithmetic Logic Unit, la parte incaricata dei calcoli matematici) e Control Unit (CU, che si occupa di scandire tempi e modi di accesso alle risorse).

La CPU è il "motore" sempre attivo del dispositivo e funziona sulla base di cicli di funzionamento, che possono essere di attesa (*idle*) oppure di attività (accesso materiale alle risorse). Il sistema operativo fa da tramite o interfaccia uomo-macchina con l'aggiunta che, a conti fatti, non potrà in anticipo cosa farà l'utente, per cui dovrà gestire le risorse dinamicamente con la possibilità di dover gestire degli "imprevisti".

Modello di von Neumann (cenni)

Come insegna il modello di Von Neumann, un computer è modellizzabile da componenti quali processore (CPU), memorie, componenti I/O (Input / Output), scheda di rete e così via. Queste componenti sono interconnesse mediante un bus di comunicazione, e questo consente di modellare un sistema in cui ogni componente può comunicare e/o scambiarsi dati e/o comandi con qualsiasi altra.

Se sto facendo un'operazione ad alto livello di somma, ad esempio, tra i numeri A e B, i dati saranno il contenuto di A e B, mentre i comandi a basso livello saranno "esegui la somma tra due numeri", "salva il risultato", "mostra il risultato" e così via.

Ricordiamo fin da subito di non fare confusione tra nome di una variabile (A) e contenuto di una variabile, ovviamente, che sono sempre distinti tra loro.

Segmentazione della memoria

Nei sistemi operativi si distinguono due aree di memoria distinte: una è riservata ai **dati** (un documento word aperto, un'immagine, il contenuto di un database, …) l'altra è riservata alle **istruzioni** dei vari programmi, mediante specifiche tecniche di indirizzamento.

Di fatto, un programma è un'astrazione concepita per eseguire un'operazione. Ogni programma ad alto livello è scomponibile in istruzioni a più basso livello (una somma tra due numeri che viene scomposta in: *acquisisci A, acquisisci B, somma A e B, salva risultato in C, mostra il contenuto di C*).

Lo scopo principale di un sistema operativo è quello di consentire l'esecuzione dei programmi dando l'illusione della **contemporaneità** all'utente, cosa che è resa possibile dalla politica di segmentazione. Grazie alla segmentazione ogni programma avrà la propria area di memoria in cui lavorare, idealmente senza interferenze o sovrascritture.

Il **parallelismo** dei processi può essere effettivo o apparente: è effettivo se il processore contiene dei sotto-processori che possono lavorare indipendentemente l'uno dall'altro, in parallelo per l'appunto, come avviene per i modelli di processore Intel i5, i7, i9, … dove il numero dopo la i rappresenta il numero di sotto-processori disponibili. È apparente, al contrario, se ci sono più operazioni da eseguire che processori a disposizione, ma il tutto avviene in modo così veloce (e secondo varie politiche di alternanza o "turni") da far sembrare che siano in parallelo. Per una questione di budget, il più delle volte, lavoreremo in ambiti con molti più processi che processori, e questo comporterà l'attuazione di politiche idonee a gestire i "turni" tra i processi stessi

Processore (CPU)

Il processore è il *core* del nostro sistema, il "cuore pulsante" della macchina, nonché l'orologio operazionale che scandisce la velocità con cui eseguiamo le operazioni.

- Permette di controllare ogni singola operazione che avviene nel computer, scandita a ritmo di un clock interno.
- Si distingue, in ogni processore, una parte adibita al controllo delle operazioni (CU, Control Unit) ed una riservata ai calcoli aritmetici (ALU, Arithmetic Logic Unit).
- Le operazioni vengono eseguite tipicamente in forma di *istruzioni*, che avvengono così a basso livello e senza che l'utente finale debba occuparsene.
- A livello macro, più microistruzioni compongono i programmi.
- Una delle funzioni tipiche dell'ambito è quella di scambiare dati tra periferiche e memoria, e per farlo usa il tramite del bus e fa uso dei cosiddetti **registri interni**, che vedremo a breve nel dettaglio:
    - ❖ Un registro MAR (Memory Address Register), che specifica l'indirizzo di memoria in cui avverrà la prossima operazione di lettura o di scrittura;
    - ❖ Un registro MBR (Memory Block Register), che contiene i dati che dovranno essere scritti o letti dalla memoria.

Memoria

La memoria di un computer può essere volatile (RAM, Random Access Memory) oppure permanente (ROM o HDD, Hard Disk Drive).

- La memoria secondaria permette di memorizzare in forma **volatile** (quando si spegne l'alimentazione *non* rimane nulla salvato) sia programmi che dati.
- La memoria **non volatile** o primaria, per inciso, è ad esempio quella dell'hard disk, che permette di salvare i dati e i programmi in forma permanente e che sopravvive allo spegnimento del sistema. In questo caso l'accesso è regolamentato in modo diverso, e si fa uso del concetto di *file system* che abbiamo visto. Possono esistere memorie aggiuntive come le ROM che funzionano, per inciso, in sola lettura (Read Only Memory).
- Il fatto che l'accesso alla RAM sia "casuale" va valutato fin da subito nella sua essenza: significa che **posso accedere a qualsiasi area di memoria desidero**, effettuando quello che viene chiamato indirizzamento di memoria. Indirizzare significa, di fatto, specificare l'indirizzo a cui voglio accedere. In termini simbolici, posso scrivere M[6] per indicare che sto accedendo alla sesta "casella" della memoria M. Per usare un modello più attinente alla realtà, posso anche scrivere M[0x06] in cui accedo, in notazione esadecimale, alla sesta casella di memoria (0x è una notazione introdotta dalla prima versione del linguaggio C, ed indica che tutto ciò che segue 0x è un esadecimale). Ricordiamo che i numeri esadecimali possono essere espressi con 16 anziché 10 cifre (sistema decimale), che sono 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
- Si può immaginare la memoria volatile come una lista contigua di indirizzi a cui è possibile accedere per indirizzamento diretto ovvero casuale.
- Gli indirizzi della memoria contigui sono rappresentati tipicamente in notazione esadecimale come *0x*i, dove i = 00000, 000001, …, FFFFFF, ad esempio.
- La capienza effettiva di ogni cella è determinata dall'architettura hardware specifica, ed in questo contesto si assume che possa contenere ad esempio 1 bit, 1 byte e così via.
- In modo simbolico è possibile scrivere M[A] per indicare il fatto che sto accedendo

all'indirizzo A della memoria M.

- Grazie al sistema operativo (ed è qui che vediamo la sua funzione) la memoria permette altresì di accedere a dispositivi esterni: il sistema operativo può ad esempio copiare i dati presenti in una porzione di RAM verso una periferica di rete, su un dispositivo di memoria USB e così via.

A livello di comunicazione l'invio e la ricezione di dati tra periferiche e componenti differenti è consentito da un *bus*, un canale di comunicazione da cui passeranno tutti i flussi di dati e istruzioni.

Il bus di Sistema (System Bus) è il canale di comunicazione che permette a CPU, memoria e moduli di I/O di comunicare tra loro.

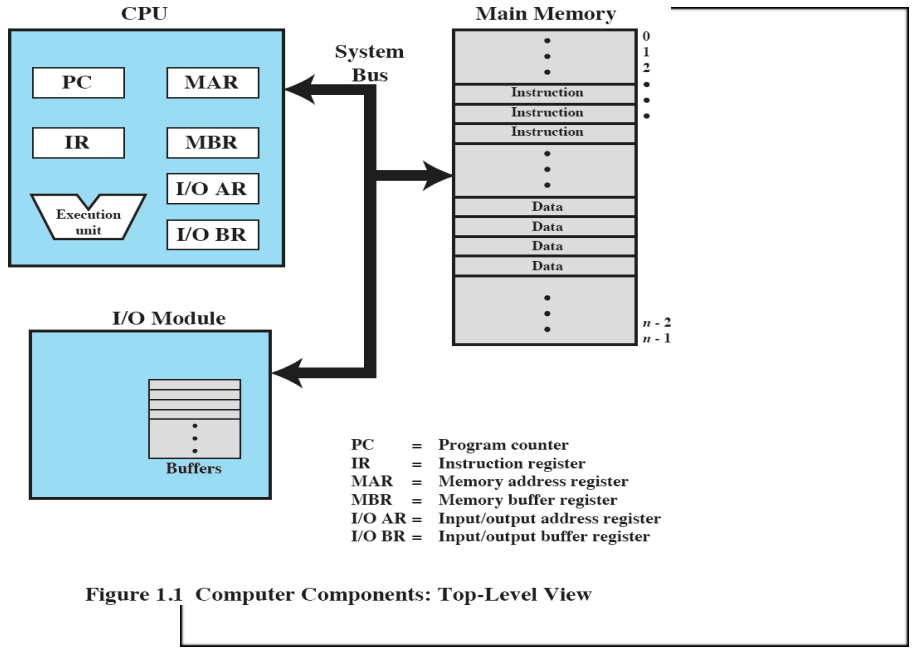
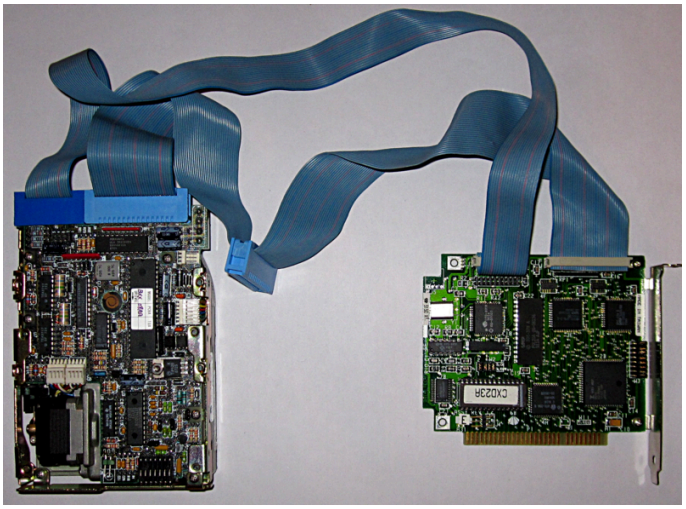


Figure 1.1 Computer Components: Top-Level View

Nella realtà hardware non sempre il bus è ben distinguibile dal resto delle componenti, ma è agevole accorgersi della sua presenza in alcuni casi: nella foto in basso ad esempio vediamo un *hard disk* (HDD) connesso mediante *bus* (il doppio connettore centrale) ad un *controller* (sulla destra). Il bus può essere presente nell'architettura di un calcolatore a più livelli e in modo modulare, senza contare che possono esistere bus adibiti ad usi specifici (bus istruzioni, bus dati, bus alimentazione, …).



Registri del processore

Nella tipica architettura di Von Neumann una CPU è composta da un'unità di controllo (Control Unit) e da una di calcolo (ALU, Arithmetic Logic Unit). Esse garantiscono un duplice funzionamento alla CPU: da un lato controllano le operazioni e le scandiscono nel tempo, dall'altro eseguono materialmente i calcoli necessari all'elaborazione.

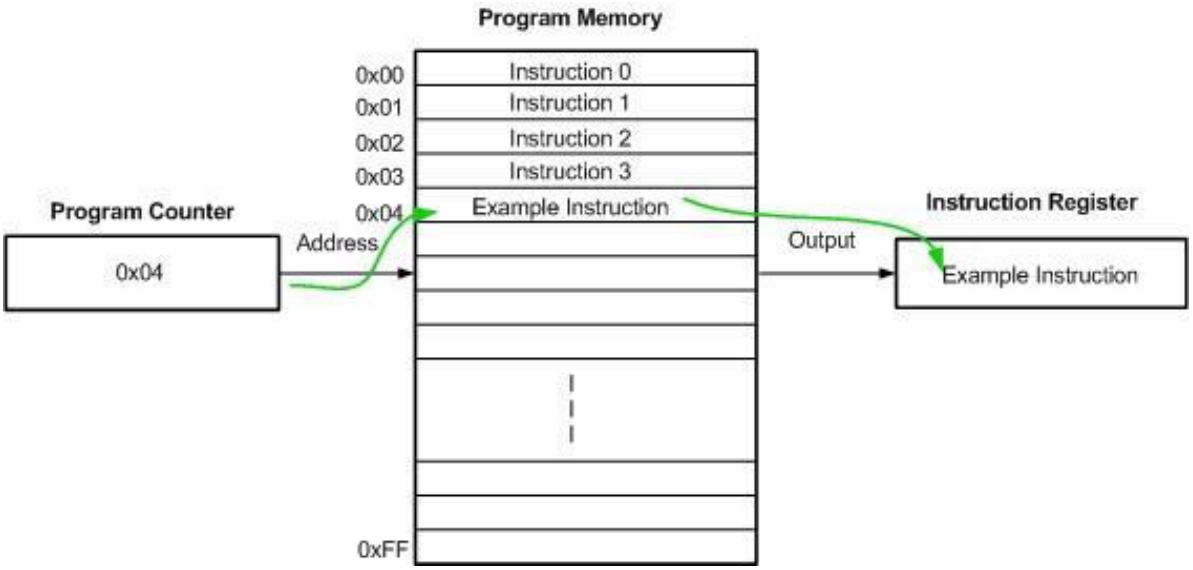
Ma una CPU o processore include, al proprio interno, oltre ad ALU e CU, un insieme di **registri** in grado di fornire scambio di dati più veloce ed efficiente di quello in memoria. Ci sono vari registri disponibili, a seconda delle architetture specifiche, e servono a garantire uno scambio di dati adeguato alle operazioni in gioco.

I registri sono in genere rappresentati da una o due lettere maiuscole e non vanno confusi con gli indirizzi di memoria, essendo ben distinti da questi ultimi. Si tratta di aree di hardware indirizzabili e gestibili per salvare rapidamente dati e recuperarli in seguito. Ad esempio le operazioni di *context switch* fanno uso dei registri, nella maggioranza dei casi.

Le funzioni dei registri del processore includono, ad un primo livello, due funzionalità distinte:

- **Registri a livello utente**: sono usati per abilitare l'accesso alla memoria a basso livello. Quando si lavora con linguaggi come C++, ad esempio, si opera ad alto livello nel senso che sono disponibili nomi di variabili e istruzioni come *for* oppure *while* per poter accedere alla memoria ed eseguire operazioni anche molto complesse; se operiamo a basso livello, invece, facciamo uso di questi registri per effettuare lo scambio dei dati a livello di bit o byte, questo ci costringe a dover scrivere codice più lungo ma offre il vantaggio di una maggiore velocità ed efficienza. Questo aspetto spiega anche perché si usano i linguaggi a basso livello ancora oggi, in alcuni frangenti: se ho bisogno di programmare un *hardware* specifico, ad esempio, può convenire fare uso di Assembly, un linguaggio a basso livello che permette di operare direttamente a livello di registri. Se uso un linguaggio ad alto livello come C++, al contrario, l'accesso ai registri viene sostituito dall'uso di variabili *int*, *float* e così via, che poi sono "tradotte" internamente, in automatico, in istruzioni a basso livello.
- **Registri di controllo e di status**: sono usati dalla CPU per controllare le operazioni del processore, ma anche dal sistema operativo, mediante routine di sistema (programmi usati solo dal SO) per controllare le operazioni ed il flusso di esecuzione dei programmi. Durante il funzionamento a regime di un SO che attiva le proprie operazioni, in altri termini, i registri di questo tipo vengono sfruttati per tenere traccia dell'indirizzo di memoria in cui stiamo eseguendo la prossima istruzione. Un esempio è il cosiddetto ***program counter***, che viene traccia con un numero progressivo dell'indirizzo di memoria in cui è presente la prossima istruzione.

La situazione viene rappresentata nell'immagine seguente, dove vediamo al centro la memoria del programma e sulla sinistra un **registro *program counter*** che punta all'indirizzo 0x04 in un'area di istruzioni contigue, una casella vicino all'altra (0x00, 0x01, …). L'istruzione eseguita viene poi trasferita nel **registro *istruction counter***, che conterrà materialmente la prossima istruzione che sarà eseguita dalla CPU.


Program Memory
0x00
0x01
0x02
0x03
0x04
0xFF
Instruction 0
Instruction 1
Instruction 2
Instruction 3
Example Instruction
Program Counter
0x04
Address
Output
Instruction Register
Example Instruction

Ricordiamo che queste operazioni avvengono al "ritmo" del *clock* di sistema, che scandisce le tempistiche di esecuzione e che esprime la propria frequenza di funzionamento in *megahertz* (Mhz).

I registri a livello utente (*user-visible register*) necessitano di un linguaggio adeguato per poter essere utilizzati, oppure si possono sfruttare ad alto livello mediante un linguaggio di programmazione come C++ (ma anche Python, Java, …), in modo tale da essere disponibili per qualsiasi programma ne faccia richiesta.

In un sistema operativo distinguiamo i programmi di sistema dai programmi utente o programmi a livello applicativo: i primi sono ad esempio l'orologio di sistema, il gestore dei processi attivi o quello che attiva la porta USB quando viene inserito un dispositivo in questo ambito, i secondi sono i programmi che usa l'utente nel sistema come ad esempio Word, Excel, Access, il browser e così via.

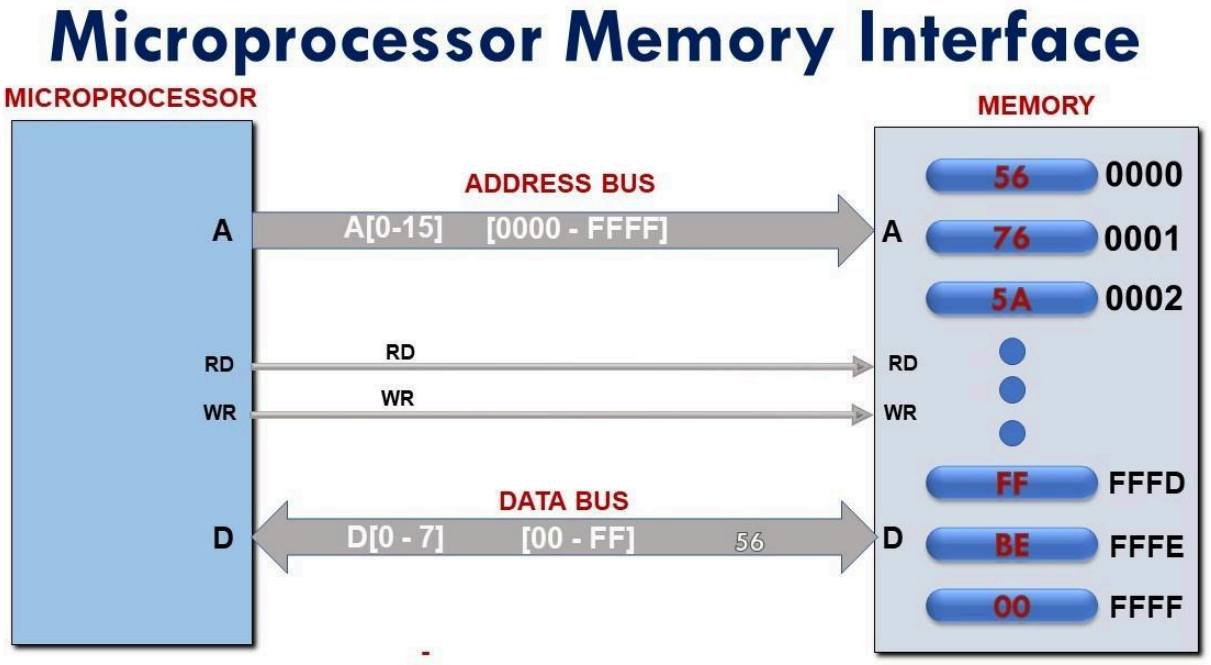
Si distinguono tre ulteriori tipi di registro, in un'architettura tipica:

- **Registri dati,** che contengono i dati che vengono scambiati (se sto ad esempio sommando due numeri, un registro conterrà il primo ed un altro il secondo);
- **Registri indirizzo**, che tengono traccia degli indirizzi di memoria in cui sto lavorando, distinguendo tra indirizzi dati e indirizzi istruzioni.
- **Registri condizione**, che realizzano condizioni logiche per cui eseguire o meno una certa parte di istruzioni (ad esempio verificare se un valore di registro intero è positivo o negativo).

In alcuni casi alcuni registri possono essere dedicati alle operazioni numeriche in virgola mobile (numeri *float* o decimali), mentre altri possono essere riservati al trattamento di numeri interi. Possono anche esistere registri "jolly" ovvero ad uso generale (*General Purpose*), che possono funzionare a seconda dei casi come registro dati, indirizzo o condizione.

La figura seguente chiarisce lo scenario: a sinistra vediamo un microprocessore con registri contraddistinti da una o due lettere maiuscole (A, D, WR, RD, …), i quali permettono di accedere alle aree di memoria dati (mediante un bus di accesso ai dati, data bus) e a quelle

di istruzioni (address bus). È importante considerare che la memoria, in questo modo, è partizionata in due aree riservate, ed il processore sa dove andare a prelevare le informazioni di cui ha bisogno in ogni momento, effettuando così la **segmentazione della memoria.**



Altri tipi di registro includono:

Registri indice (*Index register*): l'indirizzamento indicizzato è una modalità di indirizzamento che prevede l'aggiunta di un numero a un valore di base per ottenere l'indirizzo effettivo. In pratica tengo traccia del numero di caselle (*offset*) che distanziano il valore a cui mi serve accedere dall'indirizzo desiderato.

Registri segmentati (*Segment pointer*): in questo caso stiamo accedendo alla memoria sfruttando una segmentazione delle aree di memoria, ovvero blocchi di memoria a lunghezza variabile. Ogni blocco sarà composto da N>1 blocchi contigui, ad esempio, e farà in modo di consentire l'accesso in memoria mediante determinate politiche. Avremo quindi che un riferimento alla memoria sarà dato dal duplice riferimento all'indirizzo del segmento (segmento 3, ad esempio) e dall'offset necessario per raggiungere l'indirizzo effettivo

(offset=2, nell'esempio proposto di seguito).

Registro accumulatore (AC): questo è un registro molto utilizzato durante le operazioni aritmetiche di conteggio e somma, ma anche quando ci sono istruzioni da eseguire in modo ciclico. AC possiede un valore intero che tiene traccia di quante volte abbiamo eseguito una certa istruzione o insieme di istruzioni, ad esempio.

Ulteriori registri possono essere utilizzati per distinguere "*chi*" abbia richiesto di eseguire un'istruzioni, quindi ad esempio il sistema operativo oppure l'utente: questa distinzione viene fatta anche per motivi di sicurezza e di *policy*, in modo da regolamentare gli accessi ed evitare situazioni potenzialmente compromettenti per il funzionamento. Un sistema operativo, infatti, potrebbe dover eseguire operazioni che un utente non dovrebbe poter eseguire, e distinguere i due registri permette di stabilire delle regole di accesso ben codificate.

Formato istruzione a basso livello

Viene da chiedersi come sia fatta una singola istruzione a basso livello, a questi punto. Naturalmente la programmazione in linguaggio macchina è un mondo parecchio complesso che vedremo soltanto per grandi linee, ricordando che i principi sono validi per *qualsiasi* macchina, *qualsiasi* sistema operativo e *qualsiasi* programma che può essere scomposto in questo modo.

A livello *assembly* un'istruzione singola eseguita a livello macchina consiste in **tre parti distinte**:

- un *opcode*, ovvero il codice dell'operazione da eseguire;
- un parametro che può essere, a seconda dei casi, un operando oppure l'indirizzo di memoria a cui bisogna accedere;
- il *mode*, ovvero la modalità di accesso (opzionale)

opcode	*operand*	*mode*

Se in un linguaggio di programmazione classico dovessi fare un'operazione del tipo:

"dati i numeri A, B, C e D, calcolare il valore di X = (A+B)(C+D)"*

posso scomporre questa operazione in varie micro-operazioni a basso livello.

Per semplicità poniamo che la CPU disponga di quattro istruzioni diverse a livello di registro:

- **LOAD** V che serve a caricare nel registro accumulatore il valore della variabile V
- **ADD** V che serve a sommare il valore della variabile V a quello che in precedenza è stato caricato nell'accumulatore
- **STORE** V che serve invece a salvare in memoria il contenuto del registro accumulatore (qui è evidente come il registro in questione serve ad "accumulare" i risultati progressivamente)
- **MUL** V che serve a moltiplicare il valore della variabile V a quello che in precedenza è stato caricato nell'accumulatore

Vediamo nel dettaglio, passo passo, come avverrà la doppia somma nella tabella che segue: alla prima riga troviamo LOAD A, che corrisponderà a caricare il valore nell'indirizzo di memoria A dentro l'accumulatore (nella prima colonna troviamo l'opcode, nella seconda l'operando, nella terza ciò che avviene a livello di registri. Andiamo ora ad aggiungere (ADD, seconda riga) il valore di B, dato che la somma andrà eseguita prima della moltiplicazione: il valore dell'AC viene pertanto aggiornato con quello corrispondente M[B]. proseguiamo con una microprocedura di STORE, in cui all'indirizzo T viene salvato il contenuto attuale dell'accumulatore, che al momento contiene A+B.

Nella quarta riga della tabella carichiamo nell'accumulatore il valore di C, che nella microistruzione successiva sarà sommato a quello di D: una volta che abbiamo calcolato A+B e C+D, non ci resta che moltiplicarli (MUL) e salvare il risultato, finalmente, contenuto nell'accumulatore all'interrno dell'area di memoria che corrisponde alla variabile X.

opcode	operando	registri
LOAD	A	AC = M[A]
ADD	B	AC = AC + M[B]

STORE	T	M[T] = AC
LOAD	C	AC = M[C]
ADD	D	AC = AC + M[D]
MUL	T	AC = AC * M[T]
STORE	X	M[X] = AC

Un registro con puntatore allo *stack* permette di eseguire operazioni secondo una politica *push/pop*, il che significa: invece di puntare un indirizzo di memoria specifico, andiamo ad essere operazioni di *push* (inserimento in testa alla lista) e *pop* (rimozione dalla testa).

Questo permette di gestire gli indirizzi di memoria come se fosse una pila di piatti, che si accumulano dalla testa e si preleva sempre per primo quello più in alto.

Introduzione al *kernel*. Tipi di kernel

Il nucleo, nocciolo o *kernel* è la componente del S. O. adibita a **coordinare l'avanzamento dei processi attivi**, e più in generale a gestire materialmente le risorse del sistema mediante chiama ad opportune funzioni. Tale chiamata può avvenire sia direttamente che mediante delega ad altri *software*.

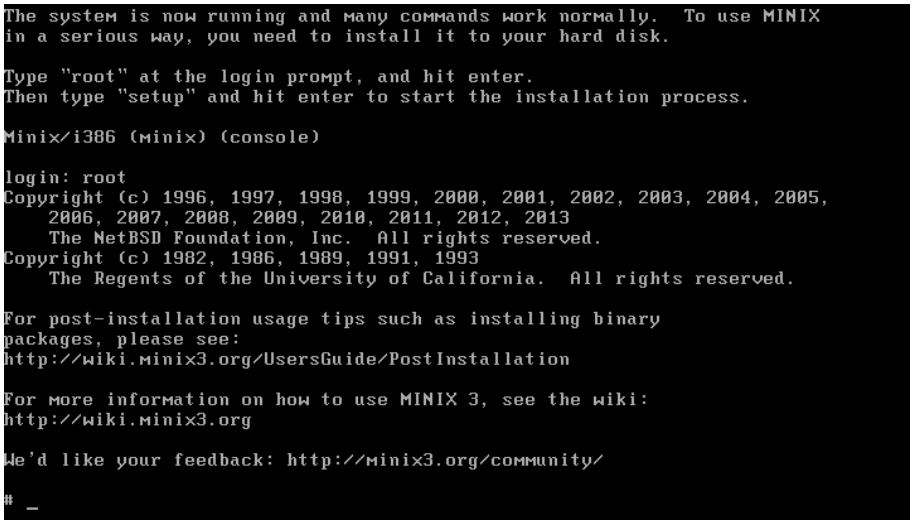
Il *kernel* viene programmato al fine di permettere l'esecuzione dei programmi, la navigazione tra file e cartelle (*file system*), l'impostazione delle preferenze di sistema (colori, dimensioni, *font* …). Il *kernel* è in genere suddiviso in N moduli, ognuno dei quali ha una funzione diversa, come ad esempio:

- gestire processi attivi;
- gestire CPU;
- gestire la memoria;
- …

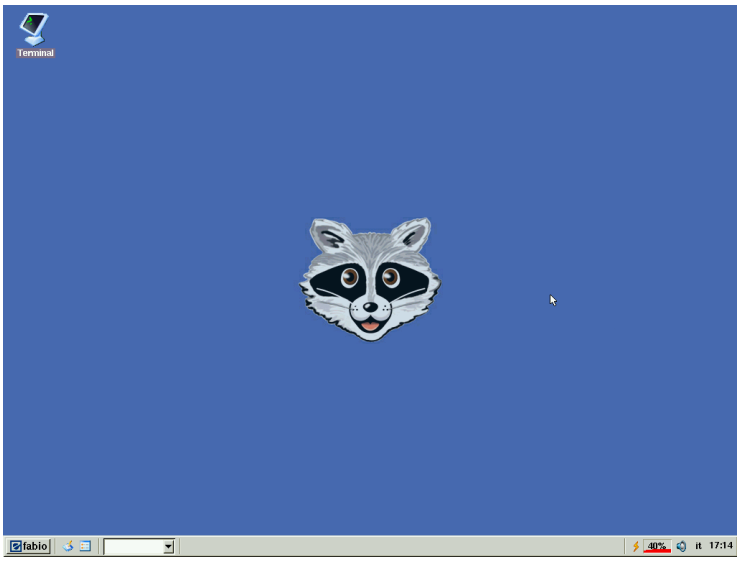
*Se un sistema operativo lavora **in modalità kernel**, in sostanza, significa che **potrà accedere senza limitazioni alla memoria e a tutte le risorse**.*

Un po' di storia: microkernel vs.

Nel 1987 il professor Andrew Tanenbaum (docente universitario di informatica, ed autore di libri molto studiati nelle università) crea MINIX, un sistema operativo basato su 26.000 righe di codice in linguaggio C allo scopo di illustrare lo stato dell'arte sui sistemi operativi. La versione di MINIX 1.5, detta tecnicamente a **microkernel**, viene utilizzata ancora oggi su alcune architetture hardware come Motorola 68000, Atari ST, Apple Macintosh e Sun SPARCstation.



Un sistema operativo a microkernel come MINIX si presenta, ad oggi, in questa veste:



Succede pero' qualcosa di rivoluzionario nel 1991, quando Linus Torvalds smette di usare MINIX e si programma da solo un nuovo kernel, che sarebbe diventato alla base di Linux, che sfruttava un'architettura contrapposta a quella precedente, detta ***kernel* monolitico**.

Si distinguono ancora oggi questi due tipi di kernel di base.

Il kernel monolitico ideato da Torvalds permette di eseguire ogni tipo di funzionalità **senza intermediari**, quindi si potrà accedere (in modalità kernel, per l'appunto) alle risorse del sistema: RAM, processi, hardware, file system. È un sistema operativo considerato veloce e facile da progettare, con la possibilità di fare qualsiasi operazione necessaria, e rendendo molto vicini il mondo delle applicazioni utente e di quelle di sistema.

L'approccio presenta comunque il problema che il voler fare una modifica anche microscopica al codice del *kernel* richiede la ricompilazione dello stesso, ovvero è necessario tradurre nuovamente il codice con cui è stato scritto il S.O. in linguaggio macchina, con il rischio di compromettere l'intero sistema per *bug* anche insignificanti.

Il sistema a microkernel teorizzato da Tanenbaum, al contrario, stabilisce che il S.O. possa usare solo funzionalità parziali (RAM e processi), mentre tutto il resto (inclusi driver e file system, ad esempio) deve essere delegato a servizi esterni. Se in un caso il programmatore è meno libero di modificare il sistema, se non altro i piccoli errori localizzati non diventano bloccanti per il sistema, per quanto le prestazioni del sistema siano considerate meno performanti.

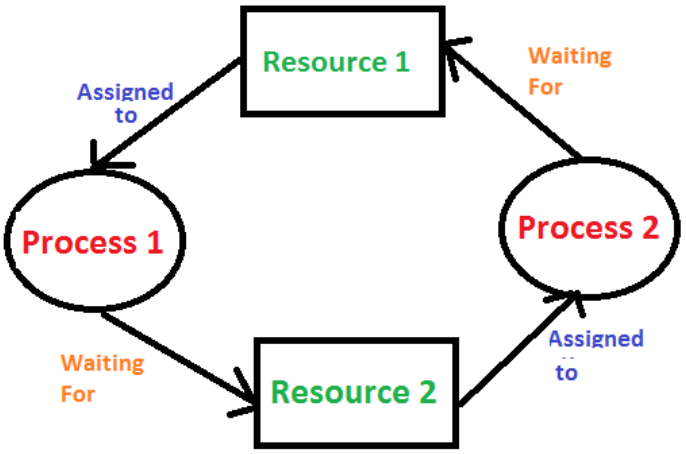
Deadlock e starvation

I S.O. possono prendere provvedimenti anche in situazioni drastiche, dette *starvation* e *deadlock*. La *starvation* si verifica quando un processo P attende "eternamente" il proprio turno, fino a morire simbolicamente di fame (è un problema enorme per il S.O., perché quel processo occuperà memoria e spazio su disco per un tempo indefinito). Il deadlock è una specie di doppia starvation, che si verifica qualora un processo P necessiti di una risorsa R bloccata dal processo Q, che a sua volta necessita di una risorsa S bloccata da P. Anche in questi casi la mutua esclusione è una politica efficiente (anche se non è l'unica), perché stabilisce la possibilità di effettuare operazioni di blocco (lock) e sblocco (lock) delle risorse all'occorrenza.

Molti sistemi della vita di ogni giorno funzionano in mutua esclusione, in effetti: il salumiere che può servire solo un cliente per volta, i ticket di assistenza che aprono i call center (il numero identificativo è un po' la "priorità" del cliente, in qualche modo), le file all'ingresso di un qualsiasi evento in cui si rispetta (sperabilmente…) l'ordine FIFO.

Esistono due esempi visuali abbastanza intuitivi per comprendere la differenza tra starvation e deadlock: la starvation è come una persona che rimane in fila molto a lungo, venendo continuamente scavalcata da altri processi con priorità sempre più grande. Il deadlock si verifica quando i processi si bloccano a vicenda perché necessitano di una risorsa che è stata presa da un processo diverso. L'intuito suggerisce che le politiche risolutive del S.O. debbano essere di due tipi: da un lato per evitare la starvation basta

gestire con attenzione le priorità dei processi, mentre per evitare il deadlock è sufficiente grarantire che nessun processo mantenga indefinitamente l'uso di una risorsa, prevedendo una politica pre-emptive con interrupt.



Concetto di mutua esclusione

L'interazione tra processi è la parte conclusiva della nostra comprensione generale del funzionamento del S.O.; possiamo focalizzare l'attenzione su P_1 e P_2, che operano nel sistema in modo indipendente, senza aree di memoria condivise e senza interferire l'uno con l'altro. È un caso solo ideale, ovviamente, perché nella realtà data una risorsa R i processi potranno avere bisogno nello stesso momento della stessa.

Lo schema generale è realizzato in questo modo (L ed S indicano rispettivamente operazioni di Lettura o di Scrittura):

Nella pratica si rappresenta uno schema semplificato in cui la risorsa R può essere scritta o letta, alternativamente, dai due processi. R può essere ad esempio una memoria RAM, intesa come una specifica locazione di memoria. Possono succedere quattro circostanze diverse nello stesso momento:

1. P_1 legge, P_2 legge;
2. P_1 legge, P_2 scrive;
3. P_1 scrive, P_2 legge;
4. P_1 scrive, P_2 scrive;

Nel caso 1 non servono particolari contromisure, dato che il dato deve essere letto e, nella peggiore delle ipotesi, leggere nello stesso istante non comporta conflitti. Molto diversi negli altri tre casi, ovviamente: se due processi *scrivono* nello stesso momento un dato (o almeno uno dei due scrive, e l'altro legge) il processo lettore potrebbe leggere dati parziali, mancanti o errati. Si dice in gergo informatico **che si può pervenire ad una *inconsistenza*** dei dati, e quel che è peggio è che sarà imprevedibile: potrebbe non capitare per molti cicli di CPU per poi capitare senza preavviso, ad un certo punto.

In tal caso il S.O. non "sa" più che dato sarà salvato nella risorsa: se P_1 per esempio volesse scrivere il numero "10" e P_2 il numero "5", rimarrebbe in dubbio. Peggio ancora: se ripetessimo la scrittura per 1000 volte di seguito (cosa che dovremmo aspettarci, dato che molti processi lavorano più volte sulla stessa area), nel 50% dei casi potrebbe rimanere salvato "10", nel 50% rimanente avremmo un "5".

Si tratta di un problema non deterministico che non ci possiamo permettere, e che fortunatamente si può risolvere dal concetto di **mutua esclusione**: si tratta di una politica di accesso che stabilisce che i due processi possono leggere in qualsiasi momento il dato, ma quando scrivono devono farlo in modo "isolato" o *safe*: nessuno dovrà interferire durante processo di scrittura, e se qualcuno arriverà nel mentre dovrà essere bloccato mediante un vero e proprio "semaforo", slittando ad un quanto di tempo successivo. R in caso di conflitti tra lettori e scrittori viene detta "**risorsa critica**", per intendere che è una risorsa condivisa da due processi che sono equivalenti, i quali devono essere serviti allo stesso modo e devono farlo senza *"pestarsi i piedi"*.

Facciamo un esempio pratico: **Mario e Maddalena hanno necessità di stampare su carta un documento** in ufficio, ma c'è solo *una* stampante connessa in rete. Possono connettersi entrambi e mandare in stampa ognuno il proprio *file*, senza sapere nulla del fatto che il collega farà lo stesso o no. Il paragone con i sistemi operativi e calzante perché il sistema operativo non so in genere quanti / quali processi vorranno essere serviti: possono essere 0 in un istante e 200 (per esempio) in un momento immediatamente successivo.

Nelle stampanti non c'è un sistema operativo tradizionale né *open source*, ma un micro-sistema dedicato e memorizzato nel *firmware* (in genere non modificabile né visibile dall'esterno): il concetto di mutua esclusione si applica allo stesso modo. Senza non si facesse uso della mutua esclusione, per intenderci, qualora Mario stampasse un documento nello stesso momento di Maddalena, la stampante produrrebbe un foglio con i due documenti miscelati, un po' di righe per volta.

Questo non è quello che vogliamo: la soluzione che sfruttano le code di stampa come questa è quella di **servire il primo processo di stampa che arriva**, bloccando qualsiasi altro e mettendolo in coda (in questo caso la priorità sarà un numerino progressivo: 1, 2, 3, …) finché non avrà finito. Si tratta della politica che viene chiamata FIFO, *First In First Out*, ovvero il primo ad arrivare è anche il primo ad essere servito. Questo è anche corretto dal punto di vista degli utenti, evita che si faccia confusione con le stampe in output ed evita un errore che sarebbe imperdonabile. La cosa fondamentale è anche quella di garantire la mutua esclusione: se il file di Maddalena è di 3 pagine e quello di Mario di 4, le uniche due possibilità prevedono o la stampa completa dei primo file e poi quella del secondo, oppure la stampa prima del secondo e poi del primo, senza temporanea possibilità di *interrupt* durante il lavoro

Modello produttore-consumatore

La nostra analisi dei sistemi operativi non sarebbe completa se non considerassimo il modello del **produttore consumatore**, il quale stabilisce una politica di funzionamento in grado di sfruttare al meglio tutte le risorse disponibili, senza sprechi ed ottimizzando il *throughput*.

Parliamo di questo modello immaginando che la risorsa, questa volta, non sia un singolo banco di memoria come nei casi precedenti bensì un *array* o lista di banchi di risorse: tale *array* viene chiamato *buffer*. I processi in questo caso sono "*skillati*" su un compito preciso: il produttore P può solo produrre dati sul buffer, mentre il consumatore C può solo prelevarli dal buffer.

Un esempio animato è presente in questo video (QR code a seguire): https://www.youtube.com/watch?v=RWyv14K1DpE

Il produttore-consumatore è un problema di sincronizzazione a cui devono farsi carico *tutti* i sistemi operativi: non si tratta solo di fare in modo che ci siano errori nei dati, ma si tratta anche di garantire una gestione ottimale delle risorse in presenza di un numero imprecisabile di processi. È necessario che il S.O. faccia da arbitro, in questo contesto, sia per dettare le adeguate tempistiche di accesso che per evitare di saturare le risorse di sistema.

Le situazioni da evitare, che inducono un *bug* nel sistema operativo, sono sostanzialmente due:

- P prova a produrre con *buffer* pieno (errore di *overflow*, trabocco di memoria);
- C prova a consumare con *buffer* vuoto (accesso negato, perchè non posso accedere ad un dato che ancora non c'è).

Il sistema operativo stabilisce pertanto alcune semplici regole:

1. P e C non possono mai operare in contemporanea (per la mutua esclusione);
2. P non deve inserire se il buffer è pieno;
3. C non deve prelevare se il buffer è vuoto.

Utilizzando un qualsiasi linguaggio di programmazione non sarà difficile verificare che il buffer pieno oppure vuoto: se non richiediamo in nessuno dei due casi, significa che possiamo procedere con il lavoro.

Senza scendere nei dettagli implementativi (che possono essere complessi, in quanto afferiscono alla programmazione concorrente o mediante *thread*) ci limitiamo ad evidenziare i passi fondamentali del *main():*

1. crea processi P;
2. crea processi C;
3. attiva processi P;
4. attiva processi C;

Con quattro semplici passi il nostro produttore consumatore sarà pronto all'uso, con le accortezze di:

1. Fissare un BUFFER_SIZE massimo all'inizio (ad esempio 1000 slot);
2. Verificare preventivamente in fase di consumo se il buffer è vuoto, ovvero:
    - if (buffer.empty())
3. verifica preventivamente in fase di produzione se il buffer è pieno, ovvero:
    - if (buffer.size() == BUFFER_SIZE)
4. Sfruttare una funzione predefinita di mutua esclusione per gestire ciclicamente i processi (in C++ esiste la libreria Mutex).

Il modello basato su Onion skin

Il S.O. deve gestire l'hardware delle risorse in modo efficace, ed in questo assolve al compito di fare da intermediario tra uomo e macchina. La struttura del S.O. è determinata da una struttura detta *onion skin*, ovvero sei strati innestati uno dentro l'altro, in grado di comunicare solo per adiacenza. Questo significa che lo strato 3 ad es. potrà comunicare direttamente solo con lo strato 4 e 2, mentre per accedere da 1 a 5 dovrò passare per tutti gli strati intermedi.

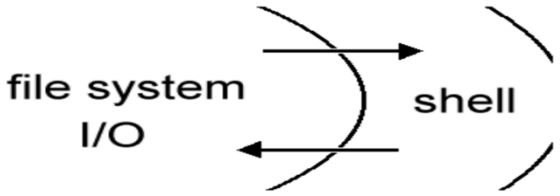


Figura 1.4 Interazione ai confini del nucleo

1Tratto da https://www2.units.it/mumolo/Introduzione.pdf

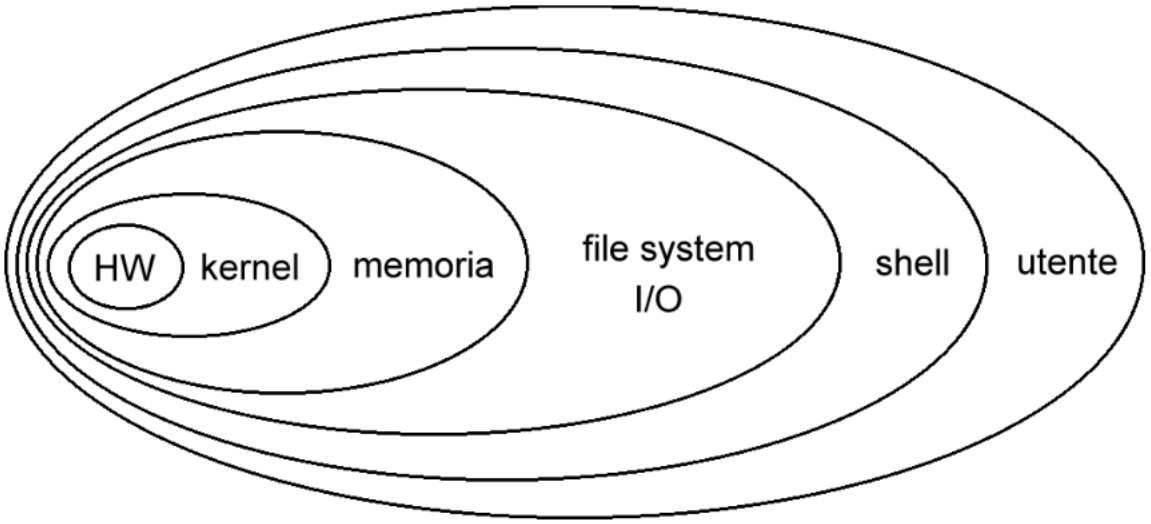


Figura 1.3 Struttura onion skin.

2Tratto da https://www2.units.it/mumolo/Introduzione.pdf

Lo schema evidenzia ancora una volta una struttura dallo strato interno a basso livello ad uno strato esterno sempre più ad alto livello.

Nella parte più interna (1) troviamo l'hardware di base, che comunica a stretto contatto con il kernel (strato 2) che abbiamo visto in precedenza. Lo strato di memoria (3), di fatto, permette di comunicare con il kernel ma anche con il file system (4). Lo strato 5 viene detto *shell*, e coincide con il terminale di comando del sistema, il quale fa da tramite con l'utente finale (6).

Dovrebbe essere chiaro, a questo punto, a cosa servono i singoli componenti degli strati: la *shell* fornisce all'utente un'interfaccia per il *file system*, utilizzando ad esempio i comandi locali (per navigare le directory e visualizzarle) come *dir* e *cd*.

Al tempo stesso lavora a contatto con l'utente, che in questa struttura per poter accedere alle funzioni del kernel dovrà passare per vari livello di accesso: *shell*, *file system*, memoria e solo a questo punto kernel. Questo offre un'organizzazione efficiente del sistema operativo ed impedisce che possano avvenire facilmente accessi indebiti da parte di *malware* o *virus*, ad esempio, stabilendo una ulteriore *policy* di sistema.

Ogni sistema operativo ha le proprie *policy* che stabiliscono, ad esempio:

- i file che si possono cancellare o modificare da parte dell'utente;
- le aree di memoria riservate al sistema operativo;
- le attività che i processi possono eseguire o meno.

La *policy* ovviamente potrebbe essere aggirata, o al contrario non essere abbastanza stringente; di contro, le *policy* troppo restrittive possono rendere complicato l'uso del sistema operativo. In genere è necessario trovare un compromesso tra varie esigenze, ed è questo uno dei motivi per cui i S.O. vengono periodicamente aggiornati (*update*).

Nello schema in alto abbiamo riportato una ulteriore stratificazione che chiarisce gli aspetti legati ai singoli processi ed alle loro tipologie: a livello di *end user* (utente finale), infatti, troviamo programmi applicativi (*application programs*) che non richiedono conoscenze informatiche di livello troppo avanzato. Se scendiamo di un gradino, al contrario, troviamo un insieme di strumenti detti *Utilities*, tra cui compilatori e debugger, che assieme alle funzionalità del sistema operativo sono offerti ai programmatori e agli utilizzatori di database per il loro lavoro quotidiano. Ad un ultimo livello troviamo i sistemisti e i programmatori di sistemi operativi, che possono accedere all'hardware a qualsiasi livello e che stabiliscono le policy che abbiamo visto.

Idealmente, pertanto, è qui che si collocano i problemi che abbiamo visto ed analizzato.