

07-03-23

Introduzione

- Il sistema operativo gestisce le istanze dei programmi in esecuzione nel minor tempo possibile.
Deve gestire una finestra temporale per la gestione della macchina in maniera ottimale.
- Un programma interrotto deve ripartire da dove si è fermato e ciò è gestito dal SO. Le informazioni che riguardano la ripartenza sono gestite dal SO e devono essere gestite nel miglior modo possibile rendendo tali operazioni "invisibili" all'utente, ottimizzando diverse sfaccettature (evitare attese fra processi ecc..).

Ci sono 2 tipi di memoria **RAM** (dati memorizzati in questo momento, quindi processi in esecuzione) e **HARD DISK**(memorizzato in modo permanente)

Come viene organizzata la gestione di accesso al HD è un altro aspetto del Sistema Operativo (SO).

Il filesystem: come vengono organizzati i dati nell'HD e la lettura dell'HD è lenta e per questo il SO deve lavorare nel miglior modo possibile.

Come memorizzo i dati all'interno di un PC? In modo permanente e i dati che vengono usati in quell'istante. Un programma non ha mai una quantità di dati statica ma è dinamica perché vengono usate strutture dinamiche e non sempre statiche. (pila, code, liste ecc..)

La memoria di un programma è sempre maggiore della dimensione disponibile in RAM e per tale motivo si usa una **memoria virtuale**.

Come vengono gestiti i processi? (esempio una stampante in un ufficio)

Segmentation fault = accedere a spazi di memoria non esistenti sulla RAM ma probabilmente sono sull'hard disk. Tale informazione va caricata nella RAM se c'è spazio altrimenti il SO deve gestire questo problema. Se la RAM è piena allora si deve rimuovere una parte di memoria per dare spazio ad altro. Se al passo dopo mi serve un dato che ho tolto al passo precedente si crea un altro **fault** e così via. Quindi i dati da rimuovere sono quelli usati meno recentemente perché è più PROBABILE che non verranno usati a breve.

Thread: programma che va in esecuzione suddiviso in sottoprogrammi(thread) e ognuno di essi si occupa di una specifica attività e si crea uno pseudo parallelismo.

<https://www.dmi.unict.it/mpavone/so.html> sito delle slide.

Sistema operativo (SO)

In caso di più processori come viene gestita la distribuzione di ciò che viene eseguito? Come si assegnano i processi ai processori? Inizialmente posso assegnarli come voglio ma devo considerare che il SO deve ottimizzare l'obiettivo da raggiungere e inoltre:

- se ho 4 processori (4 core) dovrei guadagnarci in tempistiche quindi l'obiettivo è velocizzare l'esecuzione dei programmi. *Problema:* se li assegno a caso, allora dopo un certo intervallo di tempo, o c'è un componente che coordina i processi e gli altri 3 lavorano (coordinatore per assegnazione dei programmi ai core) -> 1 coordinatore e 3 lavorano non è l'idea migliore.
 - Se ho 2 processori che lavorano a pieno e 2 che lavorano "stanno a guardare" non mi fa raggiungere il mio obiettivo principale (cioè voglio che tutti e 4 lavorino a pieno in modo che io

possa guadagnarci)

- Nella gestione dei programmi con **più processori** ho una gestione più complicata. Ogni singolo processore deve avere **lo stesso carico di lavoro rispetto agli altri** per avere la gestione a **massimo regime**.

I programmi in background (demoni) sono dormienti durante la loro esistenza, si risvegliano quando devono essere eseguiti e poi tornano "a dormire".

*Il sistema operativo mette in comunicazione i processori e altre componenti e deve soddisfare le richieste dell'utente, ovvero ricevere **immediatamente** ad un input.* (come per esempio passare da una slide ad una prossima slide).

- La **priorità** del SO è rispondere all'utente quindi tutte le altre operazioni diventano secondarie quando si deve rispondere all'utente (come su una macchina utente)
- Con una macchina industriale è diverso perchè si completa un'attività per poi passare ad un'altra quindi la priorità non è più l'utente ma varia in base all'utilizzo.

Le operazioni di base del SO sono uguali a prescindere al tipo di SO adottato (windows, linux ecc..)

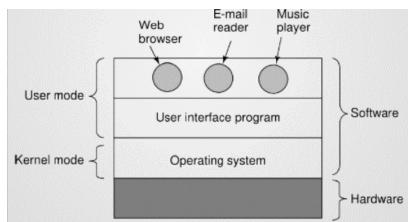
- Il SO è il **cuscinetto** fra **UTENTE** e **HARDWARE** e fa parte del software. In questo caso l'utente comunica con l'hardware mediante il linguaggio macchina.
- Il SO attiva l'hardware secondo le richieste dell'utente.

Tutto ciò funziona grazie alla traduzione in linguaggio macchina del SO.

Modalità di suddivisione del Sistema Operativo

Lasciare che l'utente possa operare alla memoria è una scelta **azzardata** ("Salva con nome" è un'operazione che fa una chiamata di sistema. Cioè il sistema operativo prende possesso dell'incarico ed effettua l'operazione. Alla fine di ciò, si ritorna dal punto della chiamata.). Il SO è suddiviso in 2 modalità di esecuzione:

- **utente**, programmi eseguiti normalmente però con dei limiti
- **kernel**, hanno accesso a tutto senza limiti ovvero tutto ciò che è eseguito dal sistema operativo.



Ogni chiamata di sistema costa in termini di tempo (**system call**) e sono operazioni che devono minimizzare le chiamate di sistema e non togliere spazio alle operazioni dell'utente.

*Il cambio password è eseguito in **modalità utente** nonostante si accedano a dati sensibili della memoria.*

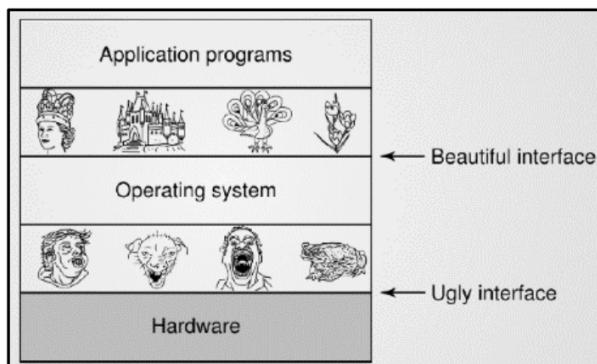
Il SO gestisce le difficoltà in maniera quanto più ottimale possibile al fine di rendere ottime le prestazioni di un calcolatore in termini di velocità, memorizzazione, risposta all'utente ecc..

Quando si usa un PC queste operazioni sono invisibili all'utente.

Il compito principale del SO è quello di rendere **semplice** le operazioni dell'utente e trasformarle in qualcosa di complesso (in *background*).

- Se si esegue un programma in un determinato istante non avrà lo stesso effetto se si effettua la stessa operazione in un tempo diverso perché tutto è centralizzato in un determinato momento. (risorse disponibili in un determinato momento ecc..)
- L'ordine con cui si sceglie quale programma mandare in esecuzione cambia la totale esecuzione di un'operazione perchè dipende dall'istante di tempo.
- In caso di risorsa condivisa (stampante, file), il processo che deve accedere rimane in attesa finchè essa non si libera. Questo è un meccanismo attivo che occupa molta memoria.
- Un altro metodo è : se un processo non può accedere alla risorsa allora il processo si "addormenta". Quando la risorsa si libera, il processo precedente "si risveglia" e quindi il processo di attesa non è più attivo, quindi non consuma in termini di risorse e si guadagna in unità di elaborazione (hardware).
- Per il SO consumare o lasciare una casella piccolissima di memoria vuota lo valuta come uno spreco di memoria.

L'obiettivo comunque rimane quello di rendere bello all'utente ciò che bello non è.



09-03-2023

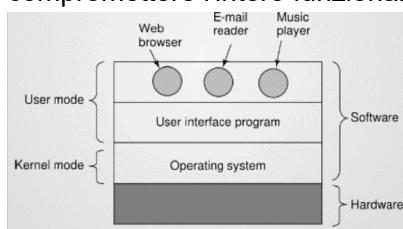
Riprendendo il discorso...

Il SO è un tipo di **software speciale** che fa da **cuscinetto** fra programmi e hardware e ha il compito di attivare l'hardware in base alle richieste### Riprendendo il discorso...

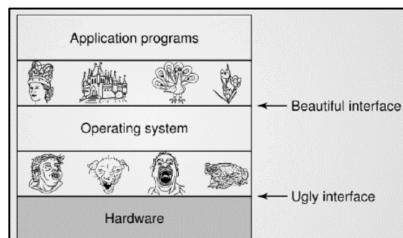
Il SO è un tipo di **software speciale** che fa da **cuscinetto** fra programmi e hardware e ha il compito di attivare l'hardware in base alle richieste dell'utente.

Ci sono 2 modalità d'uso:

- **Kernel mode**, ha accesso a tutto ciò che esiste, anche a risorse delicate ed è eseguito dal SO.
- **User mode**, non si ha accesso alle parti più delicate, profonde, della memoria perché si potrebbe compromettere l'intero funzionamento del sistema.



- Un SO è fortemente legato all'architettura sulla quale opera. Esso usa il concetto di **astrazione** attraverso il quale non ci si deve preoccupare di quello che c'è "sotto" (*uso di un file*)
- Il compito principale del SO è rendere bello ciò che bello non è attraverso l'astrazione, quindi **nascondere la complessità dell'hardware**.



- Si deve avere una visione tale da poter allocare in **maniera ordinata e controllata** per sapere in maniera **veloce** ed efficiente come mettere in comunicazione queste componenti. Devono anche gestire l'esecuzione contemporanea di molteplici programmi (*multitask*).

Sistema operativo come gestore di risorse

Per quanto riguarda la condivisione delle risorse si parla di **MULTIPLEX**. Può essere in due modalità:

1. rispetto al **TEMPO**: i programmi si alternano fra loro in modo da concedersi la risorsa
2. rispetto allo **SPAZIO**: ogni processo prende una porzione di risorsa in termini di spazio (utenti che si suddividono la stessa memoria).

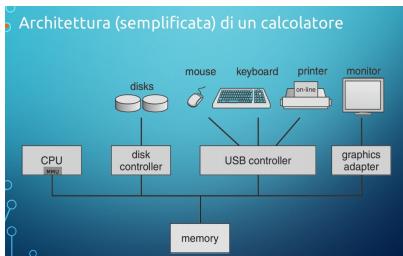
Evoluzione dei SO

Vi furono diverse *generazioni*:

1. 1945-55 : Schede a perforazione. linguaggio macchina. nessun SO e si facevano semplici calcoli matematici

2. 1955-65 : transistor, **mainframe**. I pc diventano più affidabili. Erano grandi ed erano chiusi in stanze dedicate e gestite da operatori professionisti. Acquisti solo da industrie o università (*erano costosi*). Ogni programma era detto **JOB** in **assembly**. Un processo veniva svolto uno dopo l'altro seguendo la logica di una coda e quest'operazione prendeva il nome di **Batch**. Si facevano equazioni differenziali. Si raccoglievano tutti i job.
 - In questa era ci sono due diversi calcolatori che lavorano insieme in *stanze diverse*.
3. 1965-80 : si cerca di **incorporare** le due attività all'interno di un solo calcolatore piccolo, accessibile. Si vogliono inglobare le attività. Si usano i circuiti integrati e nasce l'**unica** macchina IBM SYSTEM/360 di piccole dimensioni rispetto a prima che **ingloba le due attività**.
In questa *terza generazione* ci sono diverse tecniche:
 - gestione della **MULTIPROGRAMMAZIONE**: esecuzione di processi contemporaneamente. Quindi si cerca di occupare tutta la memoria per migliorare le **prestazioni**.
 - **Timesharing**: necessità di far sì che più utenti lavorino allo stesso tempo e la prima macchina di questo tipo fu **MULTICS** ma ebbe poco successo. Da essa si iniziò lo sviluppo di **UNIX**. I sistemi erano **COMPLESSI** e scritti in **linguaggio macchina (assembly)** quindi commettere errori era molto semplice.
4. 1980-oggi : Nasce il primo SO **MS-DOS** puramente da usato da **Shell**. Windows 3.1 fu la prima versione **PARZIALMENTE GRAFICA**. Windows 95 era il primo sistema operativo **INTERAMENTE GRAFICO**.
5. 1990-oggi : *Mobile computers, iOS, Android, Windows Mobile*

Hardware



L'hardware è collegato da vari **bus** e ce ne sono diversi tipi in base al dato che viaggia su una determinata linea: bus dati ecc...

Come vengono messi in comunicazione i vari componenti dal SO? Qual è la complessità di tali operazioni? Viene imposto un **vincolo**: le operazioni devono **ridurre il più possibile i costi** in termini di **TEMPO**.

- Aprire un file in lettura è più "leggera" come operazione rispetto alla scrittura.

Processore

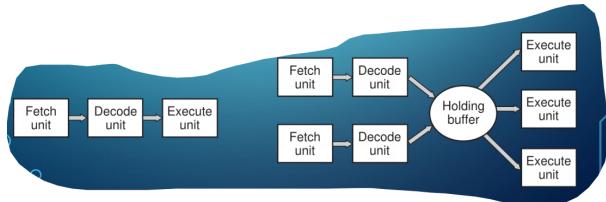
Ciclicamente preleva le istruzioni (**fetch**), le **decodifica** ed esegue le **operazioni**. È formato da tanti registri utili per memorizzare le variabili più importanti, risultati temporanei (tipo risultati da operazioni aritmetiche):

- Program Counter (PC): contiene memorizzata l'istruzione successiva da eseguire. La fetch preleva l'istruzione da questo registro
- Stack Pointer (SP): punta alla cima di uno stack usato per memorizzare i frame per ogni procedura che è stata eseguita ma non ancora completata (variabili locali, file aperti letti e altro ancora...).
 - *Quando si blocca un programma, il SO deve tenere in mente tutto quello che ha fatto fino a quel momento e dove si è fermato in modo tale da riprendere dal punto preciso.*

- Program Status Word (PSW): registro a bit che contiene la priorità della CPU, la modalità di esecuzione di un programma e altro..

Esistono diverse progettazioni avanzate:

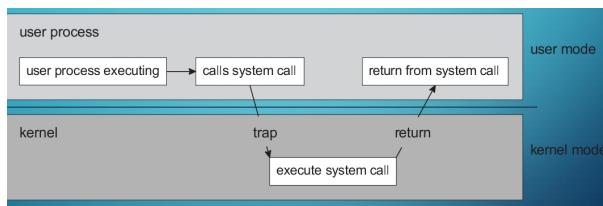
- **PIPELINE**, vuol dire che si eseguono **più istruzioni nello stesso tempo**:
 - Fetch->Decode->Execute
- **CPU SUPERSCALARE**: si ha un'unità specializzata a vari tipi di operazioni: booleane, float, interi ecc... Ciò dipende dall'operazione che si deve fare, appunto, ed è più efficiente. Dal buffer (che fa da coda) verranno prelevate le varie operazioni da eseguire.



Chiamata di sistema

Quando si ha una **chiamata di sistema**, un programma utente va in esecuzione e, quando si deve accedere a qualcosa di delicato (non concessa all'utente), si attiva una chiamata di sistema attraverso un'istruzione **TRAP** si avviene il passaggio del comando da modalità utente a modalità kernel e chiaramente ogni TRAP ha un proprio costo.

Successivamente vengono eseguite le operazioni (tipo disabilitazione degli interrupt) e dopodiché si ritorna alla modalità utente.



- Il sistema operativo, in modalità kernel, sa esattamente come gestire le situazioni più delicate a differenza dell'utente.

E' un grosso rischio assegnare all'utente di gestire la memoria.

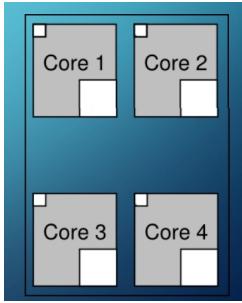
- Più si limitano le operazioni che può fare l'utente, più trap devo usare e quindi più spazio di memoria devo occupare quindi non conviene esagerare.
- Spesso le TRAP vengono causate dall'HardDisk (tipo la divisione per 0)

Più processori

Si possono avere **multiprocessori multicore** oppure si possono avere (*in entrambi i casi*) si possono avere sistemi **multithread**, cioè è come se un programma che viene eseguito, viene diviso in varie parti diverse fra loro. Parti diverse (thread) vengono eseguite come se venissero eseguite insieme(l'esecuzione avviene sempre nel tempo assegnato al programma in sè) in uno **pseudo-parallelismo**.

Ogni parte prende il nome di **THREAD** e **ognuno è differente dall'altro** in modo tale che ogni parte

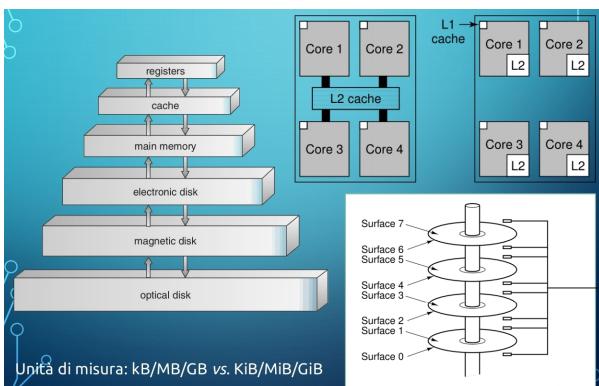
abbia il proprio compito.



- Se il SO vede un programma **senza thread** e un pezzo di programma richiede qualcosa che la manda in attesa, esso manda l'intero programma in attesa.
- Se invece ci sono i thread (che non interagiscono fra di loro) e solo un thread blocca l'esecuzione, il SO decide di far andare avanti altri thread e bloccando solo un pezzo fino alla sua liberazione. Nello stesso tempo si può eseguire parti di programma e far andare avanti le varie operazioni.
- Il **SO deve tenere conto dei thread** in modo da poterli sfruttare al massimo. Se non li conosce e si blocca un terzo flusso(thread) allora per il SO si blocca tutto il programma perchè non sa come interpretare certi segnali.
- Deve inoltre sapere **a chi appartengono i thread** perchè se si blocca un thread, il SO, se conosce senza i thread senza i proprietari, può decidere di sostituire con un qualunque altro thread e questo vuol dire sostituire programmi fra loro. Quindi è costretto ad attivare i thread di un altro programma
- **Se invece il SO conosce i thread** e i relativi proprietari, in caso di blocco, può decidere di riassegnare i tempi ai thread dello stesso programma per sfruttare al massimo i vantaggi dei thread.

Avere **più processori** non vuol dire migliorare le prestazioni ma **gestire in maniera corretta** vuol dire ottenere prestazioni migliori. Quindi se ho più processori devo avere anche una buona **organizzazione e bilanciamento** di essi per ottenere risultati migliori

Memoria



Avere diversi livelli di memoria consente di avere una memoria in grado di rispondere tempestivamente alle richieste. Ognuna di esse ha un proprio compito.

Chi non sa della gerarchia è convinto di avere una sola memoria che risponde in maniera super veloce. Si fanno meno operazioni sull'harddisk e di più sulla RAM.

- Salendo di gerarchia si hanno memorie meno capienti e più veloci.
- Nella cache si memorizza tutto ciò che mi serve **di recente** perchè tutto quello che è usato di recente probabilmente verrà usato di nuovo. In questo modo si perde meno tempo e si velocizzano le operazioni
- Le **cache L1** sono dentro la CPU e sono velocissime.

- La **cache L2** può essere di 2 tipi: unica per tutti i componenti core, oppure ognuna ha la sua componente core.

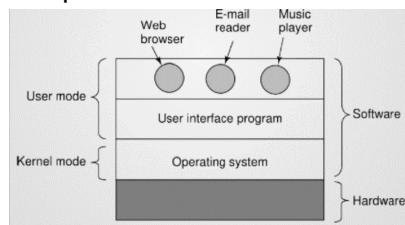
In caso di multiprocessore ogni processore ha la sua cache. All'interno di essa vengono memorizzate le informazioni su un determinato programma che viene eseguito.

Visto che i processori devono avere un **bilanciamento**, essi devono **sfruttare la cache** del processore perchè quando termina l'esecuzione di un programma, viene occupata da altri processi.

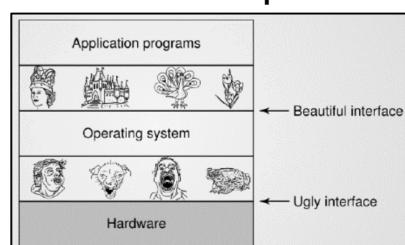
- Si deve cercare di **rieseguire i programmi in quei processi dove sono stati eseguiti precedentemente** così si **sfruttano le cache** che già **contengono i dati** usati da quel programma precedentemente se non sono stati sovrascritti.
- Le memorie a **disco fisso** sono **più lente** perchè per raggiungere una posizione si deve partire dalla testa. Qui i dati vengono memorizzati in maniera permanente e servono per l'avvio della macchina
- Le memorie elettroniche (**RAM**) sono **più veloci** perchè sono ad accesso casuale quindi si può accedere a una **qualsiasi posizione** di memoria in ogni istante di tempo. Queste memorie sono volatili (quando si spegne il pc si perdono tutti i dati).

Ci sono 2 modalità d'uso:

- **Kernel mode**, ha accesso a tutto ciò che esiste, anche a risorse delicate ed è eseguito dal SO.
- **User mode**, non si ha accesso alle parti più delicate, profonde, della memoria perchè si potrebbe compromettere l'intero funzionamento del sistema.



- Un SO è fortemente legato all'architettura sulla quale opera. Esso usa il concetto di **astrazione** attraverso il quale non ci si deve preoccupare di quello che c'è "sotto" (*uso di un file*)
- Il compito principale del SO è rendere bello ciò che bello non è attraverso l'astrazione, quindi **nascondere la complessità dell'hardware**.



- Si deve avere una visione tale da poter allocare in **maniera ordinata e controllata** per sapere in maniera **veloce** ed efficiente come mettere in comunicazione queste componenti. Devono anche gestire l'esecuzione contemporanea di molteplici programmi (*multitask*).

Sistema operativo come gestore di risorse

Per quanto riguarda la condivisione delle risorse si parla di **MULTIPLEX**. Può essere in due modalità:

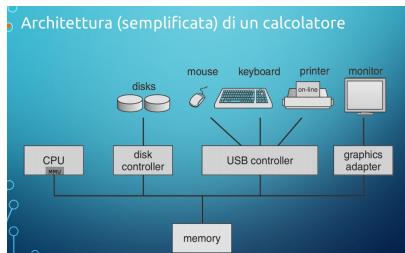
1. rispetto al **TEMPO**: i programmi si alternano fra loro in modo da concedersi la risorsa
2. rispetto allo **SPAZIO**: ogni processo prende una porzione di risorsa in termini di spazio (utenti che si suddividono la stessa memoria).

Evoluzione dei SO

Vi furono diverse *generazioni*:

1. 1945-55 : Schede a perforazione. linguaggio macchina. nessun SO e si facevano semplici calcoli matematici
2. 1955-65 : transistor, **mainframe**. I pc diventano più affidabili. Erano grandi ed erano chiusi in stanze dedicate e gestite da operatori professionisti. Acquisiti solo da industrie o università (*erano costosi*). Ogni programma era detto **JOB** in *assembly*. Un processo veniva svolto uno dopo l'altro seguendo la logica di una coda e quest'operazione prendeva il nome di **Batch**. Si facevano equazioni differenziali. Si raccoglievano tutti i job.
 - In questa era ci sono due diversi calcolatori che lavorano insieme in *stanze diverse*.
3. 1965-80 : si cerca di **incorporare** le due attività all'interno di un solo calcolatore piccolo, accessibile. Si vogliono inglobare le attività. Si usano i circuiti integrati e nasce l'**unica** macchina IBM SYSTEM/360 di piccole dimensioni rispetto a prima che **ingloba le due attività**.
In questa *terza generazione* ci sono diverse tecniche:
 - gestione della **MULTIPROGRAMMAZIONE**: esecuzione di processi contemporaneamente. Quindi si cerca di occupare tutta la memoria per migliorare le **prestazioni**.
 - **Timesharing**: necessità di far sì che più utenti lavorino allo stesso tempo e la prima macchina di questo tipo fu **MULTICS** ma ebbe poco successo. Da essa si iniziò lo sviluppo di **UNIX**. I sistemi erano **COMPLESSI** e scritti in **linguaggio macchina (assembly)** quindi commettere errori era molto semplice.
4. 1980-oggi : Nasce il primo SO **MS-DOS** puramente da usato da **Shell**. Windows 3.1 fu la prima versione **PARZIALMENTE GRAFICA**. Windows 95 era il primo sistema operativo **INTERAMENTE GRAFICO**.
5. 1990-oggi : *Mobile computers, iOS, Android, Windows Mobile*

Hardware



L'hardware è collegato da vari **bus** e ce ne sono diversi tipi in base al dato che viaggia su una determinata linea: bus dati ecc...

Come vengono messi in comunicazione i vari componenti dal SO? Qual è la complessità di tali operazioni? Viene imposto un **vincolo**: le operazioni devono **ridurre il più possibile i costi** in termini di **TEMPO**.

- Aprire un file in lettura è più "leggera" come operazione rispetto alla scrittura.

Processore

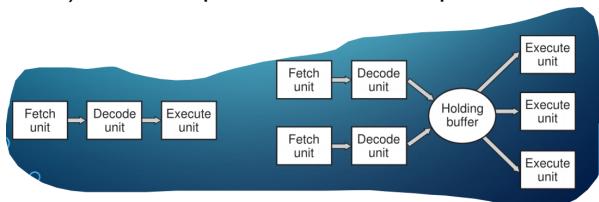
Ciclicamente preleva le istruzioni (**fetch**), le **decodifica** ed esegue le **operazioni**. E' formato da tanti registri utili per memorizzare le variabili più importanti, risultati temporanei (tipo risultati da operazioni aritmetiche):

- Program Counter (PC): contiene memorizzata l'istruzione successiva da eseguire. La fetch preleva l'istruzione da questo registro

- Stack Pointer (SP): punta alla cima di uno stack usato per memorizzare i frame per ogni procedura che è stata eseguita ma non ancora completata (variabili locali, file aperti letti e altro ancora...).
 - Quando si blocca un programma, il SO deve tenere in mente **tutto quello che ha fatto fino a quel momento** e dove si è fermato in modo tale da riprendere dal punto preciso.
- Program Status Word (PSW): registro a bit che contiene la priorità della CPU, la modalità di esecuzione di un programma e altro..

Esistono diverse progettazioni avanzate:

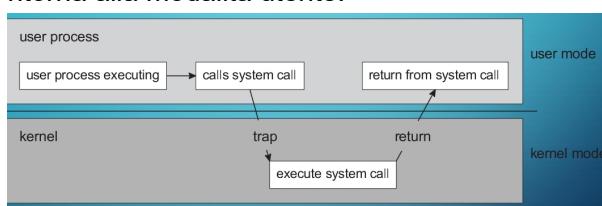
- **PIPELINE**, vuol dire che si eseguono **più istruzioni nello stesso tempo**:
 - Fetch->Decode->Execute
- **CPU SUPERSCALARE**: si ha un'unità specializzata a vari tipi di operazioni: booleane, float, interi ecc... Ciò dipende dall'operazione che si deve fare, appunto, ed è più efficiente. Dal buffer (che fa da coda) verranno prelevate le varie operazioni da eseguire.



Chiamata di sistema

Quando si ha una **chiamata di sistema**, un programma utente va in esecuzione e, quando si deve accedere a qualcosa di delicato (non concessa all'utente), si attiva una chiamata di sistema attraverso un'istruzione **TRAP** si avviene il passaggio del comando da modalità utente a modalità kernel e chiaramente ogni TRAP ha un proprio costo.

Successivamente vengono eseguite le operazioni (tipo disabilitazione degli interrupt) e dopodiché si ritorna alla modalità utente.



- Il sistema operativo, in modalità kernel, sa esattamente come gestire le situazioni più delicate a differenza dell'utente.

E' un grosso rischio assegnare all'utente di gestire la memoria.

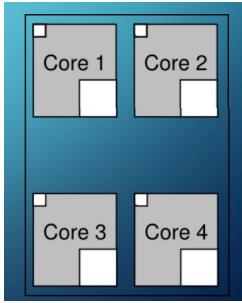
- Più si limitano le operazioni che può fare l'utente, più trap devo usare e quindi più spazio di memoria devo occupare quindi non conviene esagerare.
- Spesso le TRAP vengono causate dall'HardDisk (tipo la divisione per 0)

Più processori

Si possono avere **multiprocessori multicore** oppure si possono avere (*in entrambi i casi*) si possono avere sistemi **multithread**, cioè è come se un programma che viene eseguito, viene diviso in varie parti diverse fra loro. Parti diverse (thread) vengono eseguite come se venissero eseguite insieme(l'esecuzione avviene sempre nel tempo assegnato al programma in sè) in uno **pseudo-parallelismo**.

Ogni parte prende il nome di **THREAD** e **ognuno è differente dall'altro** in modo tale che ogni parte

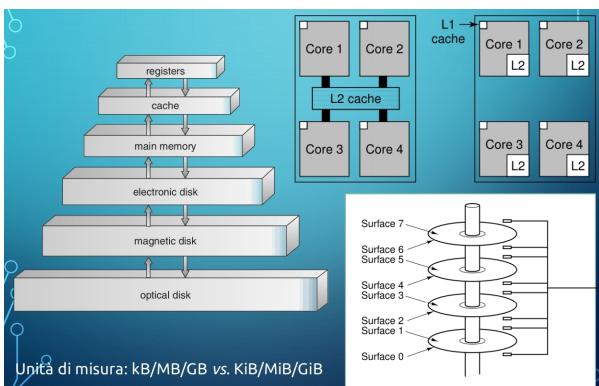
abbia il proprio compito.



- Se il SO vede un programma **senza thread** e un pezzo di programma richiede qualcosa che la manda in attesa, esso manda l'intero programma in attesa.
- Se invece ci sono i thread (che non interagiscono fra di loro) e solo un thread blocca l'esecuzione, il SO decide di far andare avanti altri thread e bloccando solo un pezzo fino alla sua liberazione. Nello stesso tempo si può eseguire parti di programma e far andare avanti le varie operazioni.
- Il **SO deve tenere conto dei thread** in modo da poterli sfruttare al massimo. Se non li conosce e si blocca un terzo flusso(thread) allora per il SO si blocca tutto il programma perchè non sa come interpretare certi segnali.
- Deve inoltre sapere **a chi appartengono i thread** perchè se si blocca un thread, il SO, se conosce senza i thread senza i proprietari, può decidere di sostituire con un qualunque altro thread e questo vuol dire sostituire programmi fra loro. Quindi è costretto ad attivare i thread di un altro programma
- **Se invece il SO conosce i thread** e i relativi proprietari, in caso di blocco, può decidere di riassegnare i tempi ai thread dello stesso programma per sfruttare al massimo i vantaggi dei thread.

Avere **più processori** non vuol dire migliorare le prestazioni ma **gestire in maniera corretta** vuol dire ottenere prestazioni migliori. Quindi se ho più processori devo avere anche una buona **organizzazione e bilanciamento** di essi per ottenere risultati migliori

Memoria



Avere diversi livelli di memoria consente di avere una memoria in grado di rispondere tempestivamente alle richieste. Ognuna di esse ha un proprio compito.

Chi non sa della gerarchia è convinto di avere una sola memoria che risponde in maniera super veloce. Si fanno meno operazioni sull'harddisk e di più sulla RAM.

- Salendo di gerarchia si hanno memorie meno capienti e più veloci.
- Nella cache si memorizza tutto ciò che mi serve **di recente** perchè tutto quello che è usato di recente probabilmente verrà usato di nuovo. In questo modo si perde meno tempo e si velocizzano le operazioni
- Le **cache L1** sono dentro la CPU e sono velocissime.

- La **cache L2** può essere di 2 tipi: unica per tutti i componenti core, oppure ognuna ha la sua componente core.

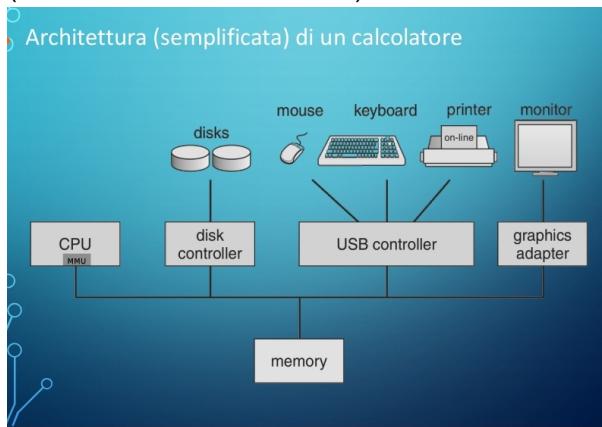
In caso di multiprocessore ogni processore ha la sua cache. All'interno di essa vengono memorizzate le informazioni su un determinato programma che viene eseguito.

Visto che i processori devono avere un **bilanciamento**, essi devono **sfruttare la cache** del processore perchè quando termina l'esecuzione di un programma, viene occupata da altri processi.

- Si deve cercare di **rieseguire i programmi in quei processi dove sono stati eseguiti precedentemente** così si **sfruttano le cache** che già **contengono i dati** usati da quel programma precedentemente se non sono stati sovrascritti.
- Le memorie a **disco fisso** sono **più lente** perchè per raggiungere una posizione si deve partire dalla testa. Qui i dati vengono memorizzati in maniera permanente e servono per l'avvio della macchina
- Le memorie elettroniche (**RAM**) sono **più veloci** perchè sono ad accesso casuale quindi si può accedere a una **qualsiasi posizione** di memoria in ogni istante di tempo. Queste memorie sono volatili (quando si spegne il pc si perdono tutti i dati).

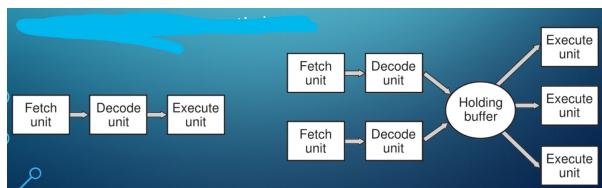
16-03-2023

Il SO deve conoscere esattamente tutte le componenti hardware che comunicano fra loro tramite i **BUS**. (di istruzioni, dati e controllo).



Abbiamo visto il processore e la memoria (parti chiavi per il SO). Le operazioni del processore:

- cerca l'istruzione e la preleva (fetch)
- Analizza l'istruzione
- Esecuzione
- **repeat()**



Nella CPU ci sono diversi **REGISTRI** per le **variabili** o registri per **dati temporanei**.

Il SO deve conoscere tutti i registri e ce ne sono alcuni specifici:

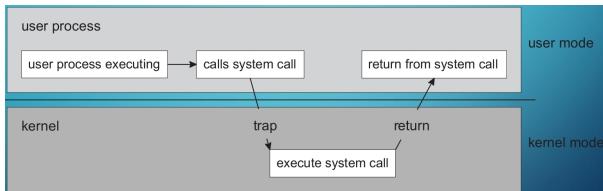
- **Program Counter** (ci sono l'istruzione successiva da eseguire)
- **Stack Pointer** (informazioni riguardo l'attività del SO, variabili utilizzate in quel momento ecc)
- **Program Status Word** (informazioni relative al programma, modalità, priorità -> non tutti i programmi sono eseguiti allo stesso modo). Quando un programma viene fermato, esso deve partire esattamente da dove si è fermato. Con chiamate di sistema I/O

Le CPU moderne permettono di svolgere più istruzioni allo stesso tempo. Mentre inizia un'istruzione e dopo un ciclo si preleva la successiva:

- **pipeline**: accelerano le operazioni. Eseguo il programma n che decodifica e il programma n+1 che preleva l'istruzione. L'unità che elabora le istruzioni è **unica**
- **cpu superscalare**: ci sono unità di esecuzione distinte specializzate. Una gestisce operazioni aritmetiche con interi, un'altra con float ecc... La prima unità che si libera preleva i dati necessari (*coerenti*) dal buffer.

Modalità esecuzione dei programmi

Una piccola parte viene eseguita in modalità **KERNEL** e la restante parte in modalità **UTENTE**. Le **TRAP(chiamate di sistema)** costano e quindi non devono essercene troppe altrimenti creano rallentamento del sistema. Si usa ridurre al minimo quello che c'è al livello kernel.



Thread: programma suddiviso in parti indipendenti e vengono eseguite in maniera **quasi parallela**.

Multithread: più thread sullo stesso programma. Il SO deve conoscere l'esistenza dei thread per gestirli nel miglior modo possibile.

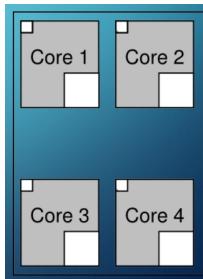
Ogni programma è diviso in **sottoprogrammi** indipendenti. I thread comandano il programma principale.

Esempio: Ogni processo ha 4 di thread ed è come se sta simulando 4 CPU

Multicore: gestione più delicata. Gli obiettivi del SO, in questo caso, massimizzare l'obiettivo (più programmi nel minor tempo possibile) e mantenere il corretto **bilanciamento di carico** per **sfruttare a pieno i vari processori o core**.

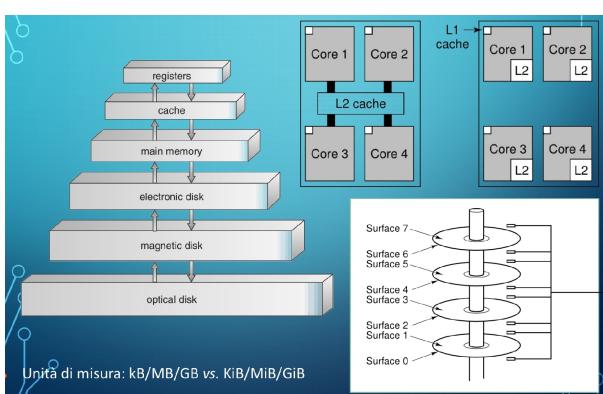
GPU: in parallelo fa diversi calcoli, rendering di parti grafiche.

Il SO lavora in diversi modi in base alle casistiche sopra citate. Usa al meglio i processori e gestisce al meglio i sottoprogrammi in modo da ottenere le massime prestazioni.



Memorie

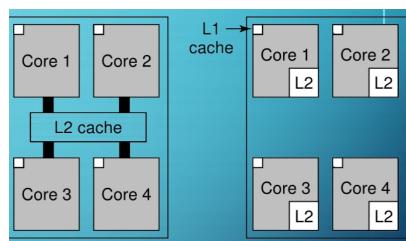
L'utente vede un'**unica memoria** veloce e di **grande quantità** di dati. Ci sono insiemi di memorie a vari **livelli**. Viene rappresentata attraverso l'**uso gerarchico** delle memorie.



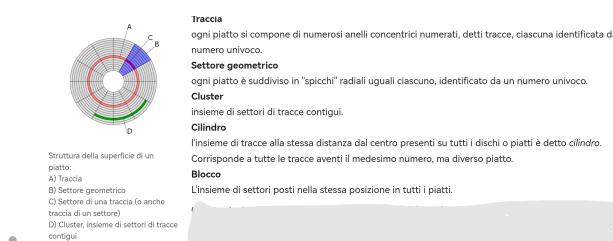
- I **registri** sono memorie piccole, veloci e costose
- La **cache** si trova nell'unità di elaborazione o nei pressi. All'interno di essa vengono memorizzati i dati che vengono usati **più frequentemente**. Se il dato viene trovato in cache si dice **CACHE HIT** e

quindi il dato viene prelevato dalla cache stessa senza analizzare le memorie più lente. Al contrario si ha **CACHE MISS** e il dato viene cercato nelle altre memorie e, allo stesso tempo, memorizzo tale dato nella cache.

- Nella cache di un processore ci sono i dati che sono stati usati per determinati processi. Si cerca di far eseguire i programmi negli stessi processori dove la cache è già popolata con i dati interessati
- La **cache L1** è dentro l'unità di elaborazione e risponde immediatamente
- La **cache L2** può essere di 2 tipi ed è nelle vicinanze dell'unità di elaborazione:
 - condivisa fra tutti i core
 - ogni core ha una sua cache L2
- I processori Intel usano la cache L2 per ogni core
- I processori AMD usano una cache L2 condivisa fra i core



- La **RAM** (che è **volatile**) contiene la **ROM** (che non è volatile) e serve per l'**avvio del calcolatore**
- **Hard Disk:** E' molto grande ed economico ma molto lento
 - I dati sono descritti in **cerchi concentrici** e ad ogni posizione del braccio, la testina legge una regione (**traccia**)
 - Tutte le tracce in una specifica zona formano un **cilindro**.
 - Ogni traccia è divisa in **settori**
 - Muovere il braccio da un cilindro ad un altro richiede circa **1ms** di tempo



- Disco ottico: per prelevare dei dati specifici si deve sempre partire dall'inizio, quindi si ha un **accesso sequenziale**

Memoria virtuale: ogni programma genera dei propri indirizzi virtuali che vengono associati a indirizzi fisici. Se ne possono usare molti di più di quelli che sono effettivamente gli indirizzi reali fisici. Di ciò si occupa **MMU (Memory Management Unit)** ed esegue la **mappatura (astrazione)** di questi indirizzi (**fisico->virtuale**).

Dispositivi I/O

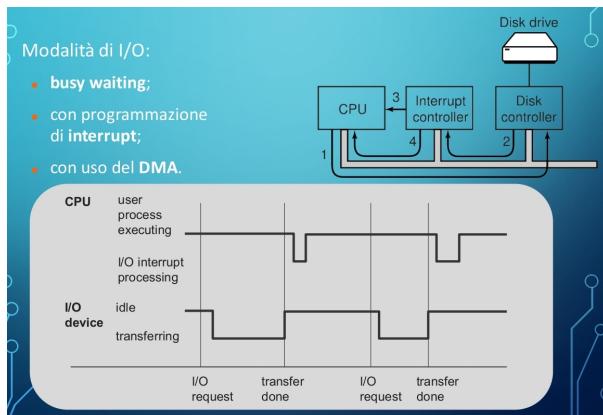
Dialogano spesso con il sistema operativo. Essi sono formati da:

- **controller:** offre un'**interfaccia semplice** per leggere i dati e comunicare con il SO. E' un **chip** (o insiemi di chip) che controlla il dispositivo. **Accetta i comandi richiesti** dal SO. Rende l'interfaccia molto semplice per il SO
- **dispositivo in sè:** ha anch'esso un'interfaccia semplice ma complicata da usare. (Esempio disco SATA: disco magnetico usato su molti PC).

Ogni controlloer e dispositivo è diverso da un'altro e per questo motivo serve un software di comunicazione. Essi si chiamano **DRIVER** e vengono installati nel SO e ne diventano *componenti*.

I sottoprogrammi che vengono eseguiti si distinguono:

- il loro tempo lo passano maggiormente in esecuzione -> (**CPU Bound**)
- il loro tempo lo passano maggiormente per richiedere I/O -> (**I/O Bound**). In questo caso alterno i programmi per liberare le risorse occupate il più velocemente possibile

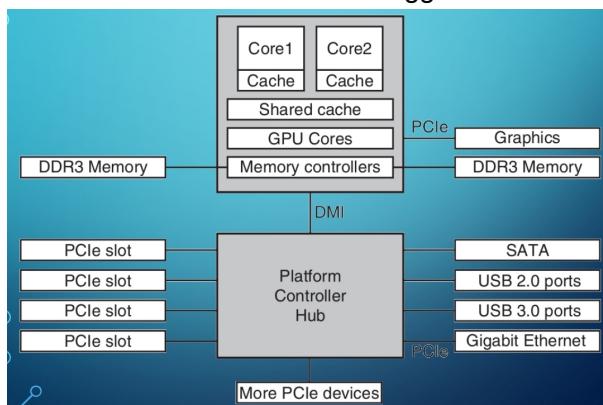


L'attesa può essere gestita in diversi modi:

- l'unità di elaborazione controlla se la risorsa si è liberata o meno ma si ha **busy waiting**. Si occupa **INUTILMENTE** attività dell'unità di elaborazione
- Uso degli **interrupt**: l'unità che controlla gli interrupt comunica con la *CPU* e dice quali dispositivi sono disponibili e quali sono le operazioni da fare. Essi interrompono le attività eseguite in *modalità kernel* perchè le loro azioni sono **determinanti**.
- *Direct Memory Access (DMA)* è un chip che controlla il flusso dei bit fra memoria e alcuni controller **senza l'utilizzo della CPU**. Si ha un trasferimento dal dispositivo direttamente alla memoria. Il *kernel* avviserà mediante *interrupt* che ci sono dei dati messi a disposizione.

Bus

Nei sistemi moderni sono stati aggiunti bus *extra* per le operazioni I/O:



Ogni bus ha una velocità e funzione propria e diversa dagli altri. Il SO deve conoscere l'esistenza e la configurazione di ogni bus.

Il bus principale è il **PCIe (express)** ed è la versione migliorativa rispetto al PCI, cioè vengono trasferiti più dati alla volta.

L'hub mette in comunicazione tutti i dispositivi mediante i bus e mediante il **DMI** (*interfaccia diretta fra dispositivi*).

Lo zoo dei sistemi operativi

In base all'**obiettivo** ci sono Sistemi operativi per:

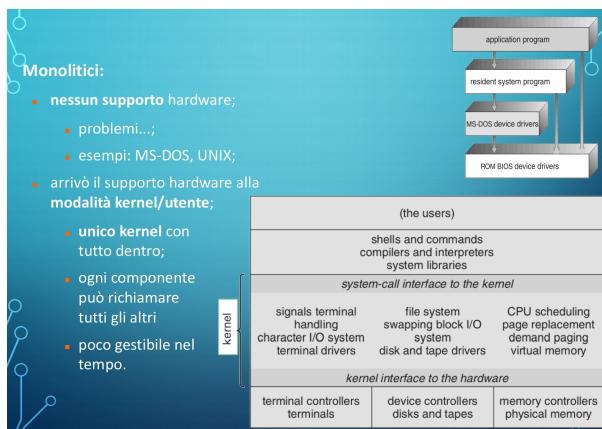
- mainframe/server: gestiscono **grandi quantità di dati** e più dischi (*come UNIX*). Si massimizzano i tempi e usare programmi complicati in un certo (*basso*) intervallo di tempo.
- personal computer: **facilitano l'interazione** con l'utente nel modo più rapido possibile
- palmari/smartphone:
- multiprocessore
- sistemi integrati (embedded): fanno una **specifica attività (ATM)**
- realtime: usati in ambito industriale ed è fondamentale che un'attività avvenga in un **preciso istante** (*catena di montaggio*).
 - **HARD**: avviene in un **preciso istante** altrimenti nulla.
 - **SOFT**: tentano di rispettare i tempi

Strutture per un sistema operativo

Alcune possibili strutture per un SO:

- **Monolitici**
 - A livelli (o a strati)
 - Microkernel
 - A Moduli
 - Macchine virtuali
 - categorie con intersezioni (sistemi ibridi);
 - tassonomia non per forza completa o condivisa

Monolitici



il SO viene eseguito in modalità kernel, visto come una raccolta di procedura che, dopo ogni procedura, ne viene chiamata una nuova. Questa situazione può creare problemi ed errori.

Era diviso in 3 parti:

- programma principale: riporta la procedura di servizio
- insieme di procedure per realizzare le chiamate di sistema
- procedure di supporto a quelle di servizio (prendono dati e richieste degli utenti)

Questo tipo è poco gestibile nel tempo. **MS-DOS** è uno dei primi esempi ed era a linea di comando

A livelli



SO diviso in gerarchia di livelli. L'interno è più importante di quello esterno:

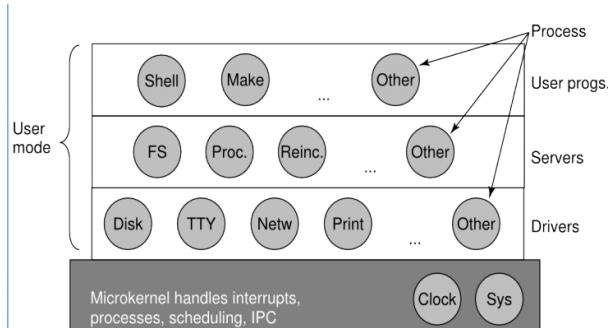
- Il livello centrale è **hardware**
- Al livello superiore -> **attività di esecuzione dei processi**, come si scambiano i programmi, cosa si fa se si termina il tempo assegnato ai programmi, azioni previste in caso di interrupt ecc... Gestione dei programmi e della **CPU**
- al livello successivo -> **gestione della memoria principale**. I programmi non dovevano preoccuparsi della memoria occupata visto che di questo si occupava il livello superiore ad esso.
- al livello successivo ->**gestione della comunicazione fra processi e utente**
- poi: *operazione I/O*
- poi: *organizzazione dei dati*
- L'ultimo livello** contiene l'**interfaccia per interagire con l'utente**

Il passaggio fra livelli **dal livello più alto a quello più basso** viene chiamato **TRAP**, cioè la chiamata di sistema.

In questa struttura ci sono dei problemi:

- I tempi di prestazione non sono eccellenti

Microkernel



Tutti gli strati vengono messi nel kernel e un possibile errore può risultare fatale. Allora si genera un kernel minimale (il più piccolo possibile) per fare poche cose (non affidate ad altre componenti) in modo da ridurre le TRAP e dare maggiore stabilità al sistema.

Il livello più basso, cioè la **modalità kernel**, si occupa di:

- interrupt**
- memoria**
- gestione dei processi (scheduling ->ordine di esecuzione dei processi)**
- gestione della **comunicazione fra processi**

- gestione del **clock**
- sys -> gestore delle chiamate di sistema

Tutte le **altre operazioni** sono in **modalità utente**.

I livelli della modalità utente sono:

1. Drivers:
2. servers: sottoprogrammi che svolgono gran parte dell'attività che deve svolgere il SO. Si verifica anche che tutti i livelli sottostanti stanno funzionando correttamente
3. user progs: rende più stabile il sistema, si riducono le attività rallentamenti dei tempi.

Struttura a moduli



Classica programmazione OOP. Si crea un kernel modulare caricabili dinamicamente e usato da unix, linux, macOS.

Si ha un kernel principale con funzionalità ridotte e altri moduli che si occupa di specifiche attività.
E' una via di mezzo fra microkernel e struttura a moduli.

Si ha:

- l'interfaccia a livello superiore
- mach: gestione della memoria e serve per il supporto alle chiamate remote
- BSD: libreria per la gestione dei thread e così via

Macchine virtuali



L'idea è quella di esagerare con l'astrazione per rappresentare diversi SO su tipologie diverse di macchine. Permette di testare diversi sistemi operativi sul proprio sistema operativo.

Viene lanciata in modalità utente. Deve attivare le richieste per la reale attivazione della modalità kernel. Lo **HYPERVERISOR** gestisce le virtual machine ed è di tipo:

- 1 -> lavora direttamente sull'hardware dell'host della macchina che lo esegue

- 2 -> le macchine virtuali vengono eseguite in un ambiente del SO convenzionale. E' un processo SO Host ed è più complicato.