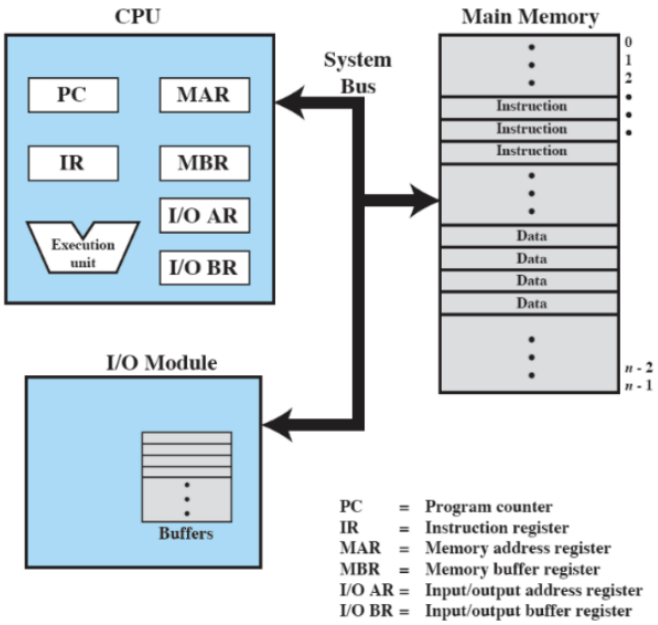
Introduzione

Index

Sistema operativo



Il sistema operativo ha il compito di gestire le risorse hardware di un sistema computerizzato generalmente composto da:

- uno o più processori → si occupa di tutte le computazioni
- memoria primaria (RAM o memoria reale) → definita volatile, se si spegne il computer infatti se ne perde il contenuto
- dispositivi di input/output → come ad esempio dispositivi di memoria secondaria (non volatile), dispositivi per la comunicazione (es. schede di rete) ecc.
- “bus” di sistema → mezzo per far comunicare tra loro le parti interne del computer (infatti i sistemi di input output scambiano direttamente informazioni con la ram e indirettamente con il sistema operativo attraverso la RAM)

Il suo scopo è quello di **fornire un insieme di servizi agli utenti**: sia per gli sviluppatori che per i semplici utilizzatori

---

Registri del processore

I registri del processore si dividono in:

Registri visibili dall'utente

Sono utilizzati soprattutto da linguaggi di basso livello e sono gli unici che possono essere **utilizzati direttamente dall'utente**. Possono contenere dati o indirizzi.
Risultano essere obbligatori per l'esecuzione di operazioni su alcuni processori, ma facoltativi per ridurre gli accessi alla memoria principale
A loro volta possono essere:

- **puntatori diretti**
- **registri indice** → per ottenere l'indirizzo effettivo, occorre aggiungere il loro contenuto ad un indirizzo di base
- **puntatori a segmento** → se la memoria è divisa in segmenti, contengono l'indirizzo di inizio di un segmento
- **puntatori a stack** → puntano alla cima di uno stack

Registri di controllo e stato

Questi vengono usualmente **letti/modificati in modo implicito dal processore** per controllare l'uso del processore stesso **oppure da opportune istruzioni assembler** (es. `jump`) per controllare l'esecuzione dei programmi.
Nell'x86 sono considerati indirizzi di controllo anche quelli per la gestione della memoria (es. i registri che gestiscono le tabelle delle pagine)
Questi sono:

- *Program Counter* (PC) → contiene l'indirizzo di un'istruzione da prelevare dalla memoria
- *Instruction Register* (IR) → contiene l'istruzione prelevata più di recente
- *Program Status Word* (PSW) → contiene le informazioni di stato (es. interrupt disabilitato, flag…)
- *flag* → singoli bit settati dal processore come risultato di operazioni (es. risultato positivo, negativo, zero, overflow…)

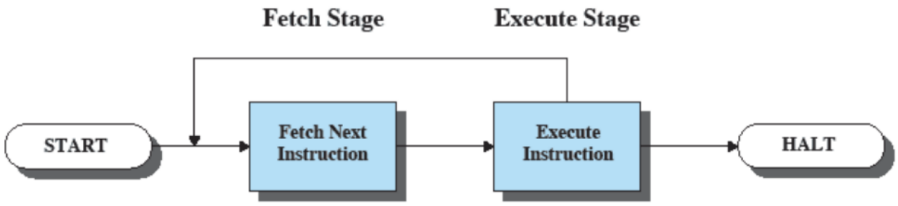
Registri interni

Questi sono usati dal processore tramite microprogrammazione e utilizzati per la comunicazione con memoria ed I/O
Questi sono:

- *Memory Address Register* (MAR) → contiene l'indirizzo della prossima operazione di lettura/scrittura
- *Memory Buffer Register* (MBR) → contiene i dati da scrivere in memoria, o fornisce lo spazio dove scrivere i dati letti dalla memoria
- *I/O address register*
- *I/O buffer register*

Per capire il funzionamento e come interagiscono MAR e MBR prendiamo questo esempio: nell'istruzione `lw $s1, 4($s2)` concettualmente, prima viene copiato l'indirizzo con l'offset di `4($s2)` in MAR, poi si utilizza il valore in MAR per accedere alla memoria e si scrive il valore letto `s1` in MBR, e infine viene copiato il contenuto di MBR in `$s1`

Esecuzione di istruzioni



Un'istruzione viene eseguita in varie fasi:

1. il processore preleva (fase di fetch) le istruzioni dalla memoria principale e viene caricata nell'IR
    - il PC mantiene l'indirizzo della prossima istruzione da prelevare e viene incrementato dopo ogni prelievo. Se l'istruzione contiene una `jump`, il PC verrà ulteriormente modificato dall'istruzione stessa
2. il processore esegue ogni istruzione prelevata

Categorie di istruzioni

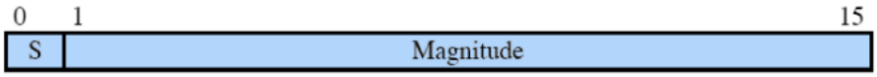
Le istruzioni sono divise in base alla loro funzione

- scambio di dati tra processore e memoria
- scambio di dati tra processore e input/output
- manipolazione di dati → include anche operazioni aritmetiche, solitamente solo con i registri, ma in alcuni processori direttamente in RAM come negli x86
- controllo → modifica del PC tramite salti condizionati o non

- operazioni riservate → (dis)abilitazione interrupt, cache, paginazione/segmentazione

Formato istruzione ed esecuzione programma

(a) Instruction format



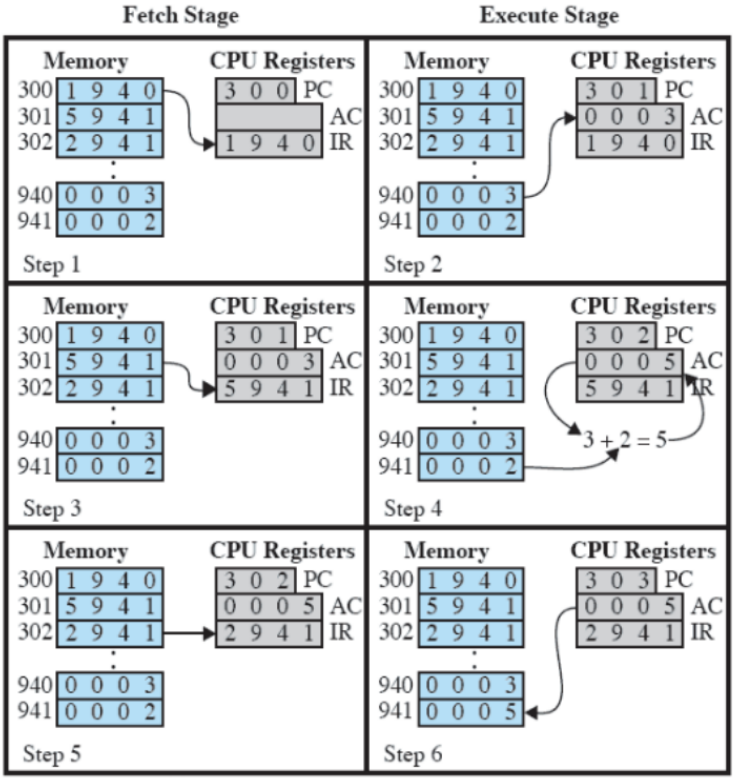
(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes



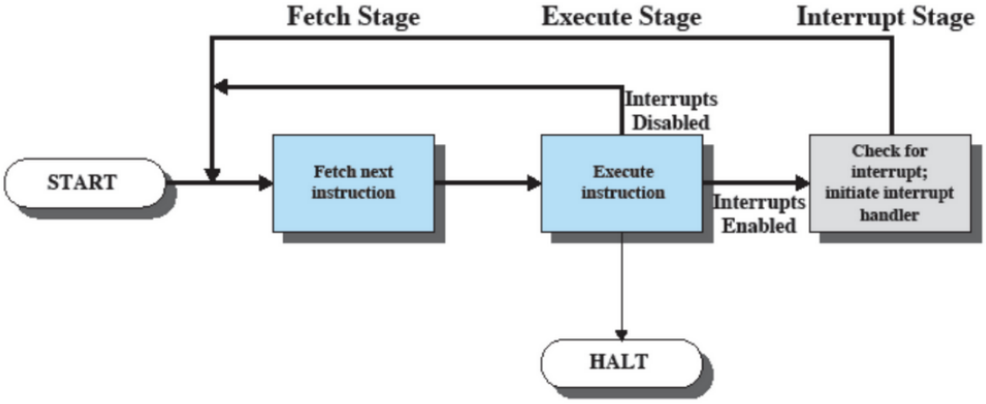
Interruzioni

Le interruzioni sono un paradigma dell'interazione hardware/software. Queste **interrompono la normale esecuzione sequenziale del programma** ed iniziano ad eseguire del software che fa parte del sistema operativo.

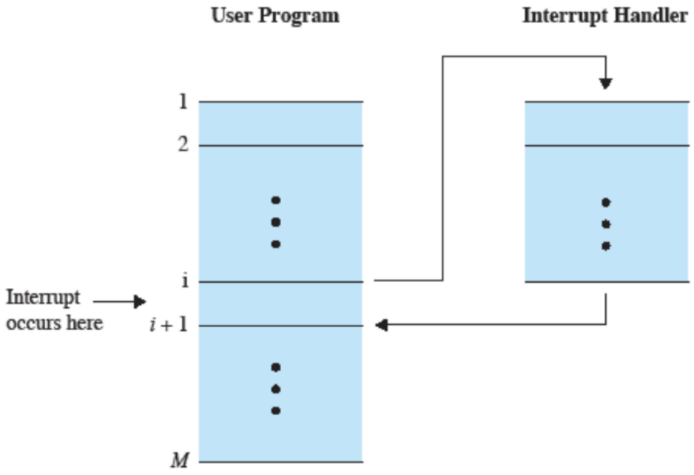
Le cause possono essere molteplici, e danno luogo a diverse **classi di interruzioni**:

- da programma (sincrone)
- da I/O (asincrone)
- da fallimento hardware (asincrone)
- da timer(asincrone)

Il ciclo fetch-execute cambia in questa maniera in caso di interruzioni:



Nel dettaglio:



Interruzioni sincrone

Le uniche interruzioni sincrone sono quelle **da programma**. Come conseguenza interrompono **immediatamente** il programma. Queste nei processori Intel, sono chiamate *exception*.

Queste sono causate principalmente da:

- overflow
- divisioni per 0

- debugging
- riferimento ad indirizzo di memoria non disponibile al programma o momentaneamente non disponibile (memoria virtuale)
- tentativo di esecuzione di un'istruzione macchina errata (opcode illegale o operando non allineato)
- chiamata a *system call*

Per le interruzioni sincrone una volta che l'handler è terminato si hanno varie possibilità:

- *faults* → errore correggibile, viene rieseguita la stessa istruzione
- *aborts* → errore non correggibile, si esegue software collegato all'errore
- *traps* (per debugging) e *system calls* → si continua dall'istruzione successiva

Interruzioni asincrone

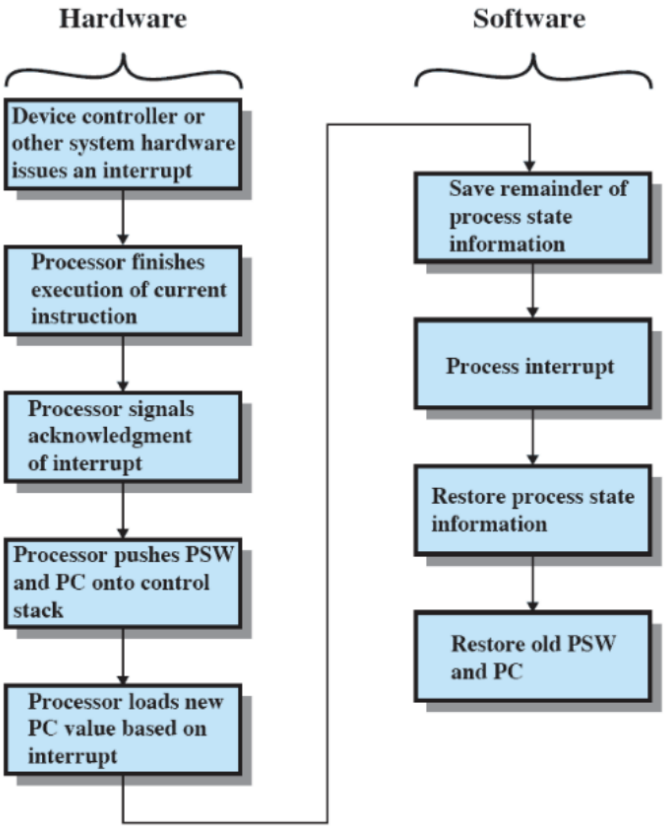
Questo tipo di interruzioni vengono tipicamente sollevate (molto) **dopo** l'istruzione che le ha causate (addirittura, alcune non sono neanche causate dall'esecuzione di istruzioni). Queste nei processori Intel, sono chiamate *interrupt*.

Le cause possono essere molteplici:

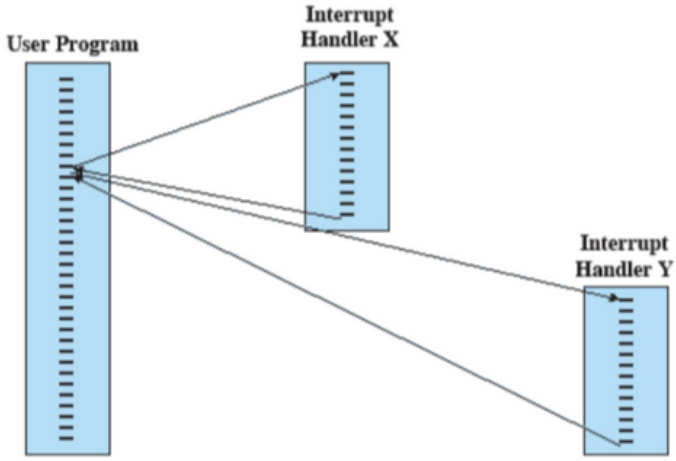
- interruzioni da **input/output**
  Queste sono generate dal controllore di un dispositivo I/O e vengono generati perché generalmente questi sono più lenti del processore. Per questo motivo il processore manda un comando al dispositivo di I/O (es. leggere/scrivere un file) e, per evitare che il processore aspetti per l'esito dell'operazione, nel mentre **continua ad eseguire altre operazioni**.
  Quando il dispositivo termina l'operazione, genera un'interruzione che segnala alla CPU di **fermarsi momentaneamente per gestire** la richiesta del dispositivo
- interruzioni da **fallimento hardware**
  Come ad esempio un'improvvisa mancanza di potenza o un errore di parità nella memoria
- interruzioni da **comunicazione tra CPU** (nei sistemi in cui ce n'è più di una)
- interruzioni da **timer**
  Generate da un timer interno al processore. Queste permettono al sistema operativo di eseguire alcune operazioni ad intervalli regolati

Per questo tipo di interruzioni, una volta che l'handler è terminato, **si riprende dall'istruzione subito successiva** a quella dove si è verificata l'interruzione (ovviamente solo se la computazione non è stata completamente abortita, anche se in alcuni casi ci potrebbe essere un process switch)

Trasferimento del controllo

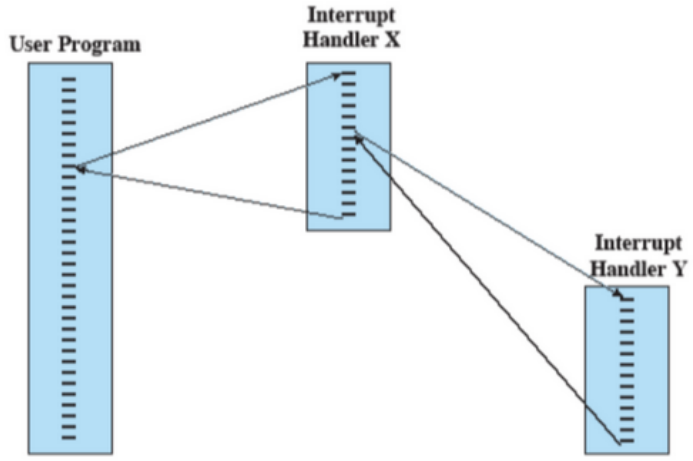


Interruzioni sequenziali ed annidate



(a) Sequential interrupt processing

Si parla di **interruzione sequenziale** quando, mentre sto sto eseguendo un'interruzione, me ne arriva una seconda, quindi finisco di eseguire la prima interruzione per poi eseguire la seconda



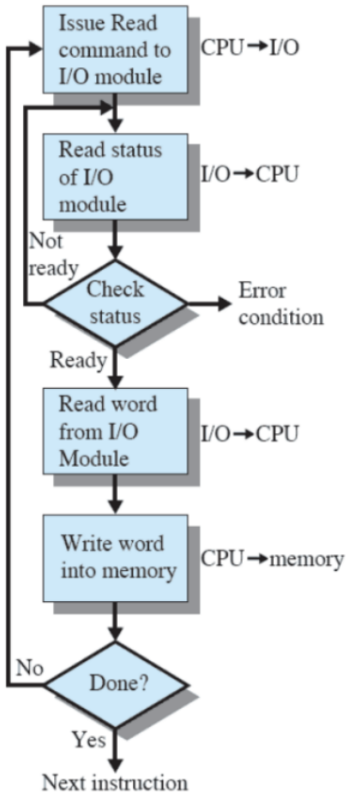
(b) Nested interrupt processing

Si parla di **interruzione annidata** quando, mentre eseguo un'interruzione me ne arriva una seconda, quindi metto momentaneamente in pausa la prima interruzione per eseguire la seconda

---

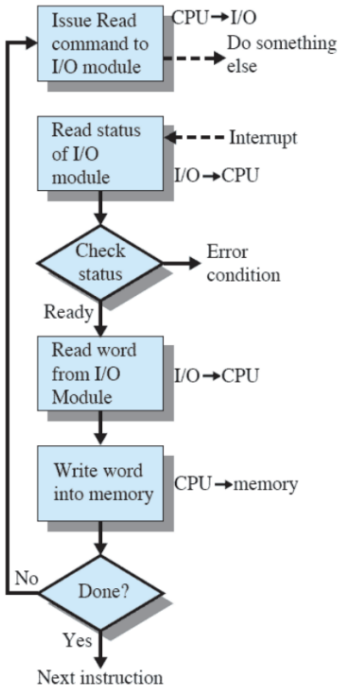
I/O

I/O programmato



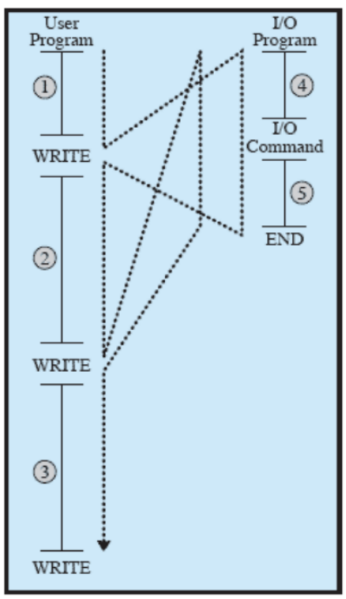
Questo è il modo più vecchio di fare I/O. In questo caso l'azione viene eseguita dal modulo di I/O, non dal processore e dunque il processore controlla lo status (check busy-way) finché l'operazione non è completa. Dunque il processore rimane bloccato finché non è terminata l'operazione di I/O.

I/O da interruzioni

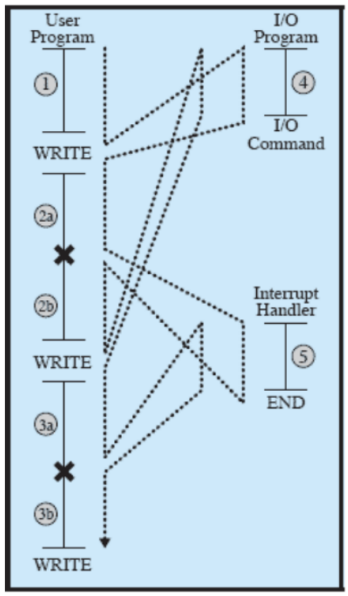


Nei sistemi recenti per fare I/O si utilizzava I/O da interruzioni. In questo caso il processore viene interrotto quando il modulo I/O è pronto a scambiare i dati; il processore salva il contesto del programma che stava eseguendo e comincia ad eseguire il gestore dell'interruzione (evitando sprechi inutili di tempo) quando l'operazione su I/O è stata terminata.

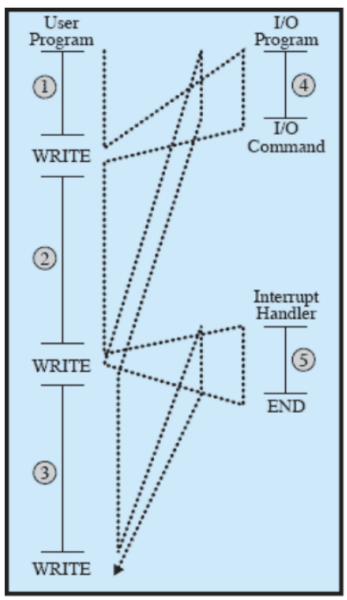
Flusso di controllo



(a) No interrupts



(b) Interrupts; short I/O wait



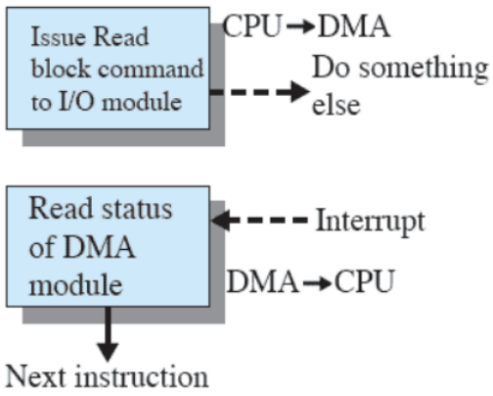
(c) Interrupts; long I/O wait

Nel caso (a) viene mostrato come nei vecchi sistemi la CPU rimane in pausa finché l'operazione I/O non è terminata

Nel caso (b) viene mostrato come nei sistemi più moderni, quando viene eseguita un'operazione I/O, la CPU continua a fare altre operazioni finché non arriva

un’interruzione a segnalare che l’operazione I/O è terminata
Il caso (c) invece tratta di sistemi moderni però con operazioni I/O particolarmente lunghe tanto che nel mentre che esegue la prima ne arriva una seconda (la prima non è ancora stata completata), in questo caso la CPU attende di terminare la prima operazione per poi eseguire la seconda

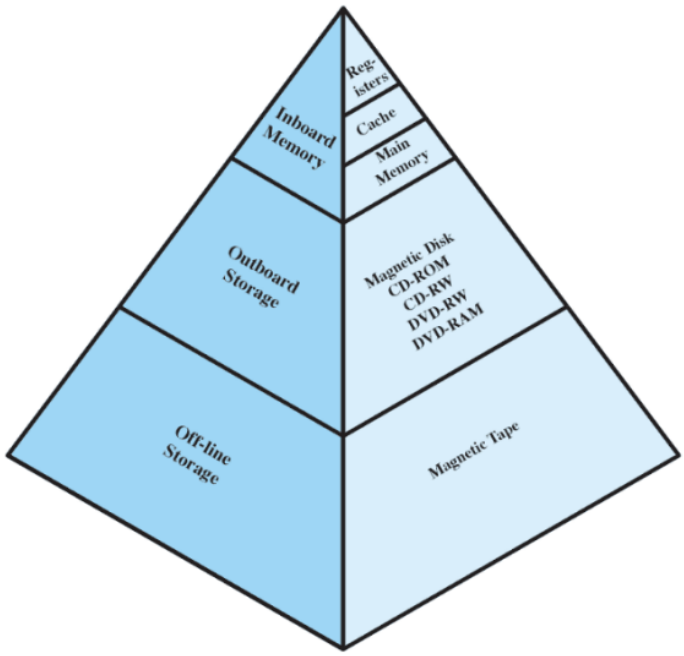
Accesso diretto a memoria



Nel computer attuali il metodo I/O utilizzato è quello dell’**accesso diretto a memoria** (DMA). Le istruzioni di I/O infatti tipicamente richiedono di trasferire informazioni tra dispositivo e memoria, dunque si preferisce trasferire direttamente un blocco di dati dalla/alla memoria in quanto più efficiente rispetto ai sistemi precedentemente elencati. In questo caso l’interruzione viene mandata quando il trasferimento è completato

---

Gerarchia della memoria



Dall’alto al basso:

- diminuisce la velocità di accesso
- diminuisce il costo al bit
- aumenta la capacità
- diminuisce la frequenza di accesso alla memoria da parte del processore

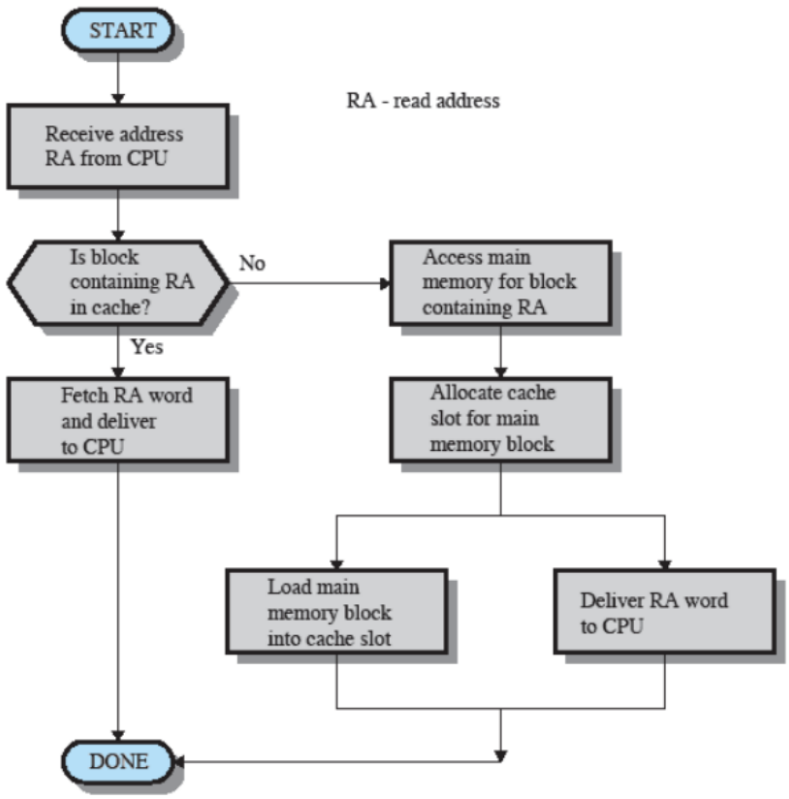
Memoria secondaria

Questa corrisponde all’offline e all’offline storage. Questo tipo di memoria è non voltatile, infatti se si spegne il computer il contenuto rimane e viene comunemente utilizzata per memorizzare i files contenenti programmi o dati

Inboard memory

Anche all’interno dell’inboard memory ci sono importanti differenze di velocità: infatti, la velocità del processore è maggiore della velocità di accesso alla memoria principale. Per questo motivo tutti i computer hanno una **memoria cache**: una memoria piccola e veloce che sfrutta il principio di località

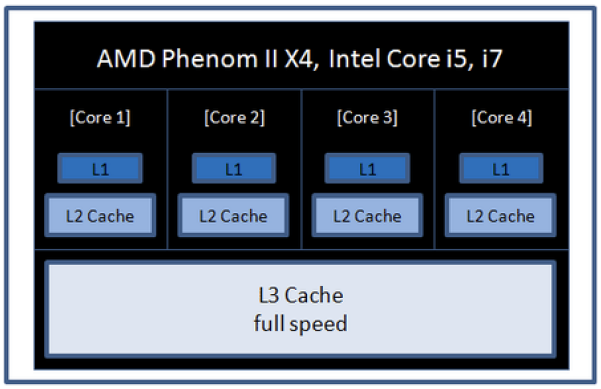
Memoria cache



La cache contiene copie di porzioni della memoria principale. Questi dati in particolar modo sono dati che la CPU ha utilizzato di recente o sta utilizzando e sono mantenuti

vicini tra loro in modo tale da massimizzare la velocità con cui la CPU ci accede
Quando però si scrive sulla cache il valore che precedentemente era scritto in RAM non viene aggiornato, per questo esistono due politiche di scrittura:

- **write-through** → viene modificato il dato in RAM ogni volta che questo viene modificato in cache
- **write-back** → si scrive in memoria solo quando il blocco di cache viene rimpiazzato



Funzione di mappatura → determina la locazione della cache nella quale andrà messo il blocco proveniente dalla memoria
Algoritmo di rimpiazzamento → sceglie il blocco da rimpiazzamento, l'algoritmo Least-Recently-Used (LRU) rimpiazza il blocco usato meno di recente

---

Sistema operativo

Lo scopo del sistema operativo è quello di **fornire servizi agli utenti** (es. ambiente grafico per eseguire applicazioni). Gli obbiettivi di un sistema operativo sono quelli di garantire convenienza, efficienza e capacità di evolvere
Si occupa inoltre di gestire le risorse hardware di un sistema computerizzato (es. processore/i, RAM, I/O ecc.)

Servizi offerti

- Esecuzioni di programmi → app, servizi; anche molti contemporaneamente
- Accesso ai dispositivi di input/output
- Accesso al sistema operativo stesso → shell
- Sviluppo di programmi → compilatori, editor, debugger, system calls
- Rilevamento e reazione ad errori → errori hardware, software
- Accounting → chi fa cosa

Kernel

Il **kernel** (“nucleo”) è la parte di sistema operativo che si trova sempre in memoria principale e contiene le funzioni più usate. Per **monitor** si intende un programma o un modulo che funge da supervisore o gestore delle operazioni del sistema operativo

---

Caratteristiche Hardware

Con il passare del tempo si iniziò a notare la necessità di ottemperare ad alcune mancanze presenti all’interno dei computer.
I computer infatti ai tempi erano a sistema **semplice non interattivo** (o *batch*)
In particolare era necessario:

- proteggere la memoria → si aveva un problema nel caso in cui un job andava a scrivere sul gestore del job stesso (rendere il sistema operativo inaccessibile dai job)
- aggiungere dei timer → per impedire che un job monopolizzi l’intero sistema
- istruzioni privilegiate → il gestore dei job viene eseguito con dei privilegi più elevati rispetto ai job normali

Protezione della memoria

Attraverso la protezione della memoria si definiscono due spazi di lavoro la **modalità utente** in cui vengono eseguiti i comuni job (le istruzioni che cercavano di accedere a zone di memoria protette non potevano essere eseguite) e la **modalità sistema** (qui possono essere eseguire istruzioni privilegiate)

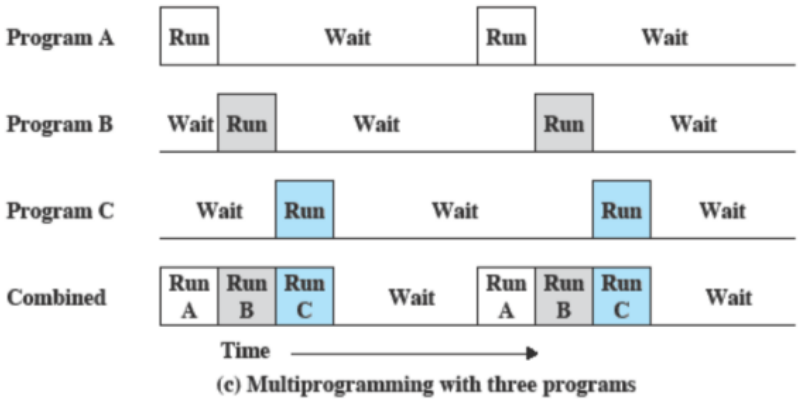
Nonostante l’avanzamento dell’hardware rimasero presenti comunque molti problemi come ad esempio la **sottoutilizzazione** nato dal fatto che la maggior parte del tempo utilizzato dalla CPU è tempo di attesa dei dispositivi I/O
Per risolvere questo problema si è passati alla **multiprogrammazione**

Multiprogrammazione

Quando un processore deve eseguire più programmi contemporaneamente, la sequenza in cui sono eseguiti dipende dalla loro priorità e dal fatto che siano o meno in attesa di input/output. Dunque per evitare di perdere cicli di clock aspettando, la CPU procede con l’esecuzione di un altro programma).

> ⓘ **Info**

> Alla fine della gestione di un'interruzione, il controllo potrebbe non tornare al programma che era in esecuzione al momento dell'interruzione



(c) Multiprogramming with three programs

Sistemi time sharing

Con il passare del tempo però cominciavano ad essere necessari dei job "interattivi" data la presenza di un prompt con cui l'utente si interfaccia, questi utenti però utilizzavano tutti un proprio prompt che era connesso ad un computer centrale detto *mainframe*.
Si iniziò quindi, a partire dagli anni '70, a pensare a dei **sistemi a condivisione di tempo**; vengono quindi date delle fette di tempo a ciascun utente

	Batch	Time sharing
Scopo principale	Massimizzare l'uso del processore	Minimizzare tempo di risposta (tempo che deve aspettare l'utente tra l'invio del prompt e la risposta)
Provenienza delle direttive al sistema operativo	Comandi del job control language, sottomessi con il job stesso	Comandi dati da terminale

---

Dal job al processo

Il **processo** riunisce in un unico concetto il job non-interattivo e quello interattivo. Incorpora inoltre anche un altro tipo di job che cominciò a svilupparsi negli anni '70: quello *real-time* (transazionale, quando due terminali richiedono contemporaneamente un job il sistema operativo deve essere in grado di gestirlo).

Il processo dunque può essere considerato come un'**unità di computazione** caratterizzata da:

- almeno un flusso di esecuzione (*thread*)
- uno stato corrente
- un insieme di risorse di sistema ad esso associate

Difficoltà della multiprogrammazione con i processi

Il passare da job a processo ha introdotto diverse difficolta come ad esempio:

- errori di sincronizzazione → gli interrupt si perdono o vengono ricevuti 2 volte
- violazione della mutua esclusione → se 2 processi vogliono accedere alla stessa risorsa ci possono essere dei problemi
- Programmi con esecuzione non deterministica → un processo accede ad una porzione di memoria modificata da un altro processo
- Deadlock (stallo) → un processo A attende un processo B che attende A

Necessità di implementazioni

Con i processi si sono notate anche delle necessità relative alla gestione della, memoria, sicurezza e gestione delle risorse

Memoria

- Isolamento dei processi
- Protezione e controllo degli accessi
- Gestione (compresa allocazione/deallocazione) automatica
- Supporto per la programmazione modulare (stack)
- Memorizzazione a lungo termine
- Metodi attuali: paginazione + memoria virtuale

Sicurezza

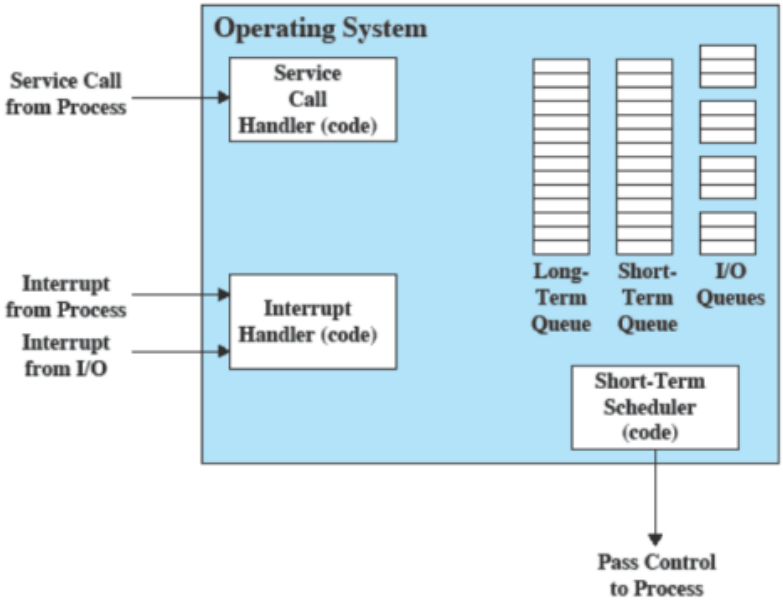
- Disponibilità (availability)
- Confidenzialità
- Integrità dei dati
- Autenticità

Gestione delle risorse

- dare accesso alle risorse in modo egualitario e equo (*fairness*)

- velocità di risposta differenziata
- massimizzare l'uso delle risorse per unità di tempo (*throughput*), minimizzare il tempo di risposta e servire il maggior numero di utenti possibile

Struttura del sistema operativo



Il tutto possibile attraverso questi pochi componenti

Per fare in modo che il tutto sia organico il SO viene visto come una serie di livelli in cui ogni livello effettua funzioni specifiche

Livelli di base

- Livello 1
  - circuiti elettrici (registri, celle di memoria porte logiche ecc.)
  - operazioni → reset registro, leggere locazione di memoria
- Livello 2
  - insieme delle istruzioni macchina
  - operazioni → add, subtract, load, store
- Livello 3
  - aggiunge il concetto di procedura (o *subroutine*)
- Livello 4
  - interruzioni

Livelli multiprogrammazione

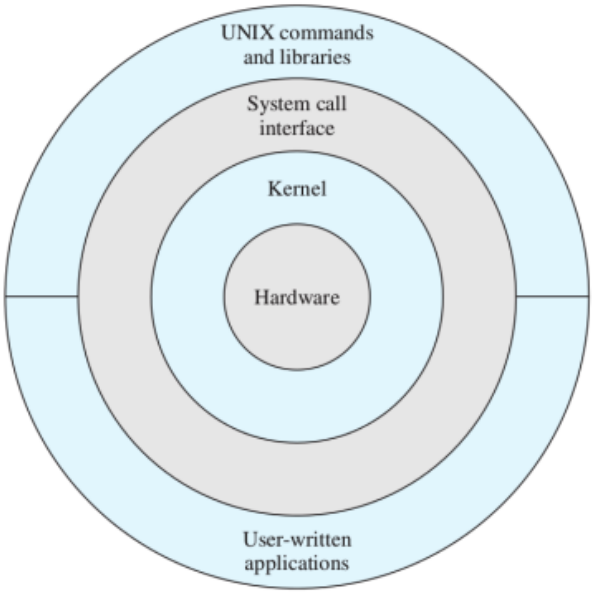
- Livello 5
  - processo come programma in esecuzione
  - sospensione e ripresa dell'esecuzione di un processo

- Livello 6
  - dispositivi di memorizzazione secondaria
  - trasferimento di blocchi di dati
- Livello 7
  - crea uno spazio logico degli indirizzi per i processi
  - organizza lo spazio degli indirizzi virtuali in blocchi

Livelli dispositivi esterni

- Livello 8
  - comunicazioni tra processi
- Livello 9
  - salvataggio di lungo termine di file con nome
- Livello 10
  - accesso a dispositivi esterni usando interfacce standardizzate
- Livello 11
  - associazione tra identificatori interni ed esterni
- Livello 12
  - supporto di alto livello per i processi
- Livello 13
  - interfaccia utente

Architettura UNIX



Kernel moderno di Linux

I kernel dei sistemi operativi moderni possono essere o **monolitici** oppure **microkernel**

- monolitico → tutto il sistema operativo viene caricato in memoria al boot (più efficiente come velocità ma occupa più memoria e meno modulare)
- microkernel → solo una minima parte del kernel si trova in memoria, il resto viene caricato in base alle necessità (rimane sempre in memoria lo scheduler e la sincronizzazione, tutto il resto solo a richiesta)

Linux è principalmente monolitico ma presenta i **moduli** infatti alcune parti particolari possono essere aggiunte e tolte a richiesta dell'immagine in memoria del kernel (driver, file system ecc.)