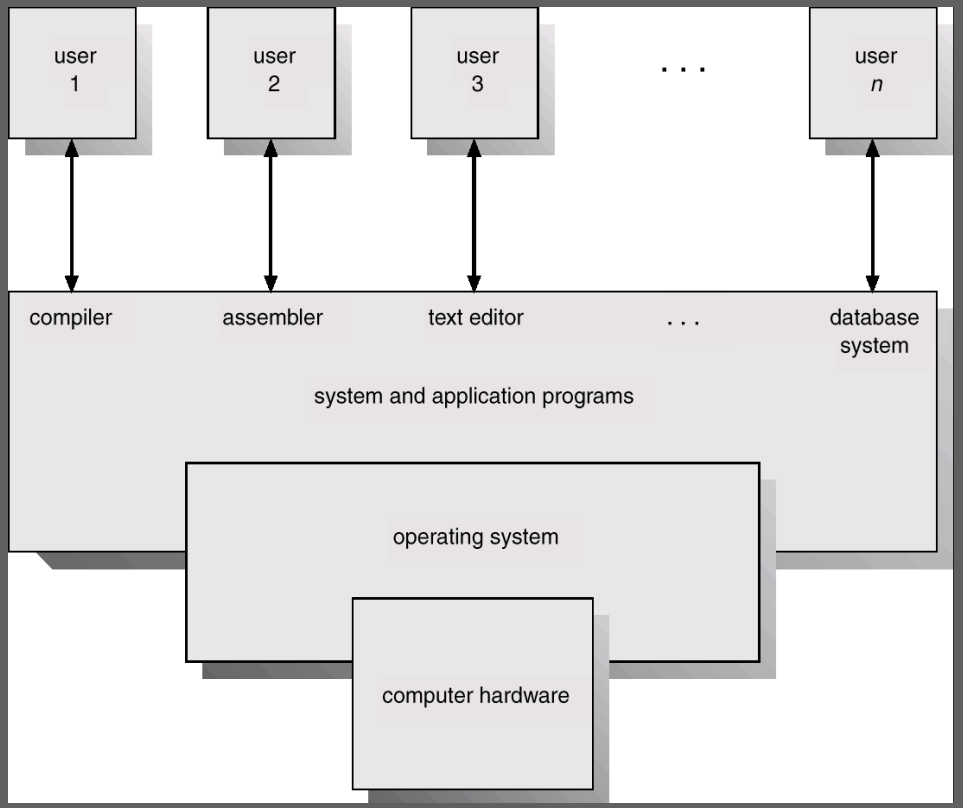
PRIMA PARTE

1.1 Introduzione

- **Intuitivamente, possiamo dire che un Sistema Operativo (SO):**



- È un intermediario tra l'utente (gli applicativi) e l'hardware
- È costituito da programmi che gestiscono l'utilizzo delle risorse fisiche di cui è fatto un computer (in particolare *la memoria primaria e secondaria, le periferiche, e la stessa CPU*) (fig. 1.1)

1.1 Introduzione

- Più in dettaglio un sistema operativo:
  - Fornisce gli strumenti per un uso corretto e facile delle risorse della macchina (la CPU, le memorie, le periferiche…)
  - Alloca le risorse nella maniera più conveniente da un punto di vista globale.
  - Controlla l'esecuzione dei programmi in modo da prevenire gli usi impropri e limitare i danni causati da errori, malfunzionamenti, e attacchi intenzionali

1.1 Introduzione

- Quindi, un SO ha 2 obiettivi primari:
  - **Punto di vista dell'utente**: rendere il sistema **semplice** da usare (in primo luogo, per poter organizzare i propri file e lanciare i propri programmi)
  - **Punto di vista della macchina**: rendere il sistema **efficiente e sicuro** (quindi, allocare in modo efficiente e sicuro le varie risorse – memoria e CPU soprattutto – tra le varie applicazioni, *ed eventualmente tra i vari utenti*, del sistema)

1.1 Introduzione

- All'interno di un computer, il confine tra il SO e le altre componenti software non è sempre ben definibile. Ad esempio, l'interfaccia grafica fa parte del SO?
- Nel passaggio da ms-dos a windows, l'interfaccia grafica si è integrata nel SO, ma inizialmente non era parte del SO stesso
- In Unix, i comandi a disposizione dell'utente NON sono parte del SO, come pure le varie interfacce grafiche a disposizione

1.1 Introduzione

- Il problema di definire esattamente quali siano i componenti di un sistema operativo sembra avere poca importanza, ma da un punto di vista commerciale/economico può avere una rilevanza fondamentale.

- Ad esempio, qualche anno fa il Dipartimento di Giustizia USA promosse una azione legale contro la Microsoft accusata di includere troppe funzioni all'interno di Windows (ad esempio Explorer) facendo così concorrenza sleale ad altri produttori e rivenditori di applicazioni simili a quelle incluse in Windows

1.1 Introduzione

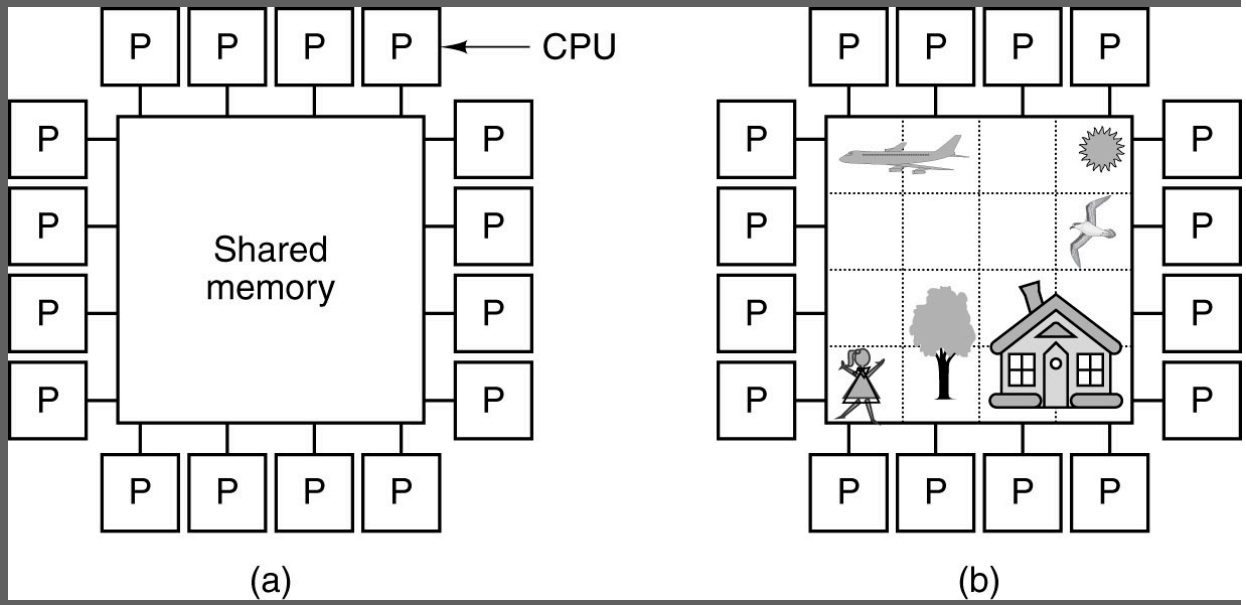
- In ogni caso, in un sistema operativo è di solito possibile individuare un "cuore" – propriamente detto **kernel** (nucleo) – del sistema, che svolge funzioni simili in qualsiasi sistema operativo, e che sono:
- gestione dei **programmi in esecuzione**, gestione della **memoria principale**, gestione della **memoria secondaria**
- L'obiettivo del corso è di studiare il funzionamento del kernel di un generico sistema operativo.
- Tutti i sistemi operativi "general purpose" esistenti in commercio (Windows, MacOS, Unix, Linux, Solaris, ...) adottano soluzioni simili a quelle che vedremo.

1.3 Architetture single/multi-core

- All'incirca fino alla fine degli anni '90, termini come "processore" o "CPU" indicavano una singola unità per l'esecuzione delle istruzioni macchina.
- Un processore poteva eseguire le istruzioni di un solo programma alla volta, come le architetture che avete studiato nel corso di Architetture I
- In realtà un processore di fascia alta (come i Pentium, gli AMD e i PowerPC) poteva eseguire in parallelo anche 3 o 4 istruzioni macchina, ma sempre appartenenti allo stesso programma

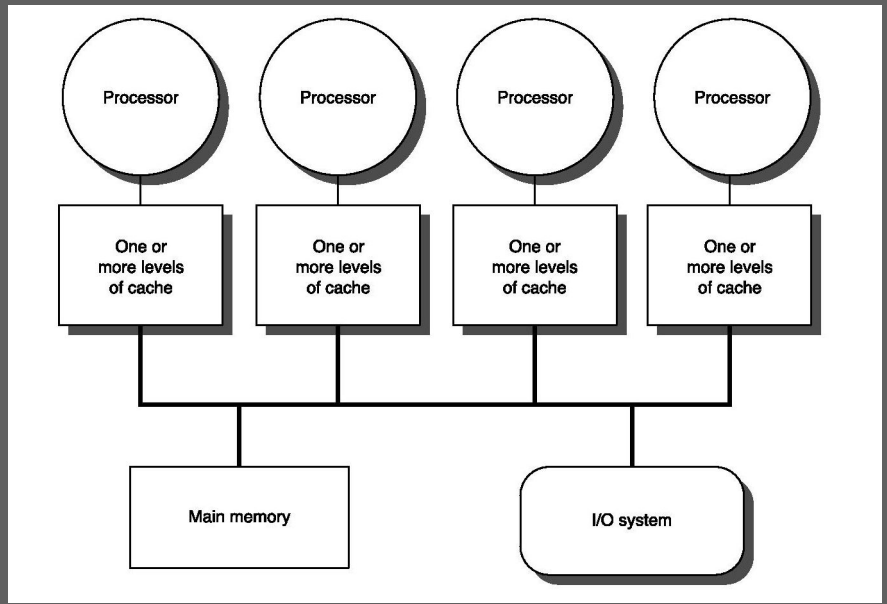
1.3 Architetture single/multi-core

- Se c'era bisogno di maggior potere computazionale, era necessario mettere insieme più processori che utilizzavano, **e condividevano**, la stessa memoria principale, la RAM. (Più propriamente dovremmo dire che i vari processori **condividono lo stesso spazio di indirizzamento**, ma questo lo capiremo bene solo alla fine del capitolo sulla memoria primaria)



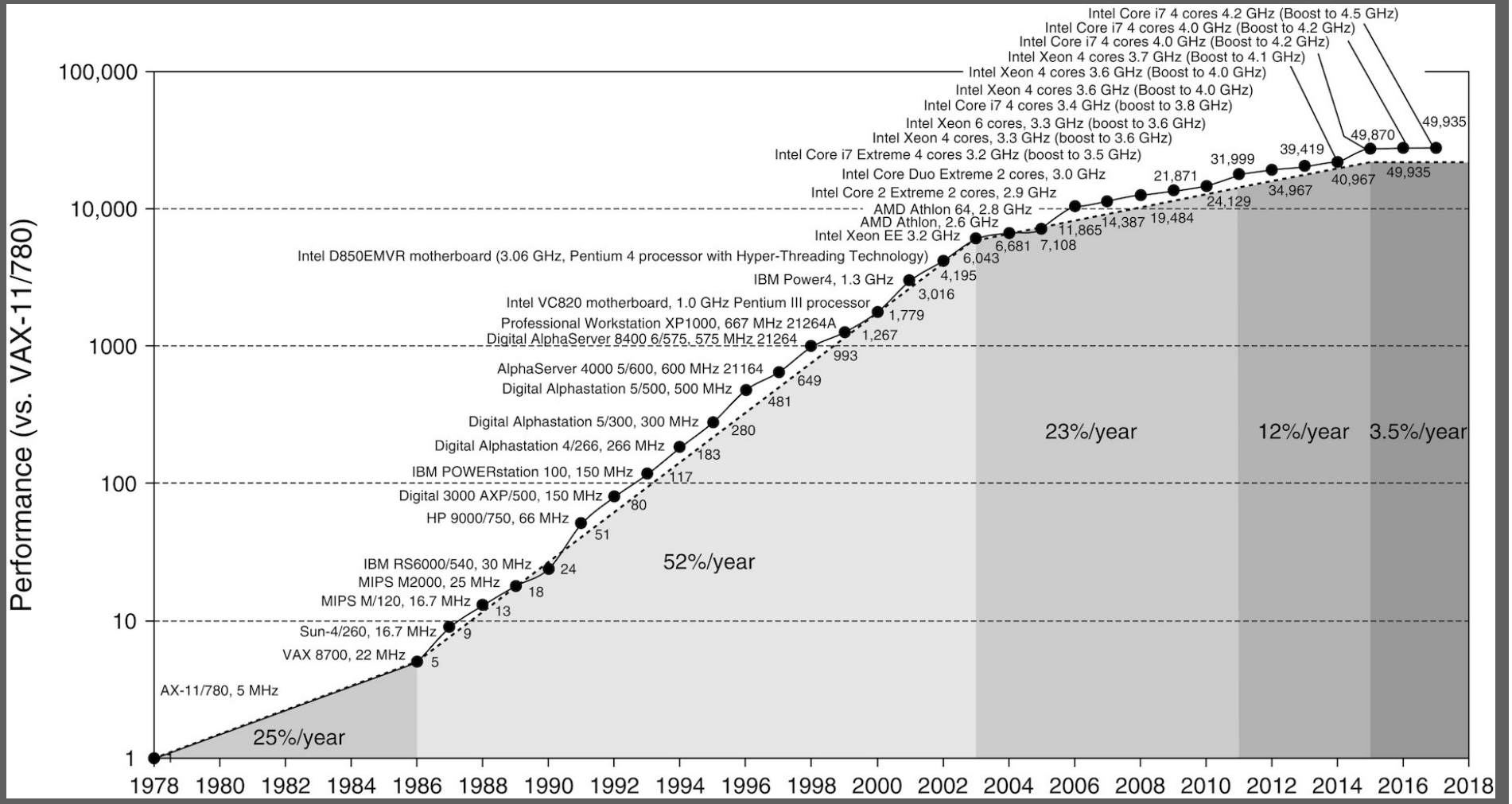
1.3 Architetture single/multi-core

- Si può parlare allora di un **sistema multi-processore**. Se tutti i processori si affacciano su un'unica memoria principale utilizzata da tutti i processori, e se il tempo di accesso alla RAM è lo stesso per tutti i processori, allora parliamo di un sistema multi-processore di tipo **UMA**: **U**niform **M**emory **A**ccess.
- Ogni CPU ha anche uno o più livelli di cache privata, posizionata tra la RAM e la CPU per velocizzare l'accesso a dati e istruzioni di uso più immediato.



1.3 Architetture single/multi-core

- Dai primi anni del 2000 però, l'aumento di prestazioni di un singolo processore rallenta sensibilmente



1.3 Architetture single/multi-core

- Per contrastare questo rallentamento e poter offrire sul mercato computer dalle prestazioni sempre più elevate si incominciamo a progettare CPU costituite da due processori affiancati sullo stesso **die** (la stessa fettina di silicio)
- I due processori cambiano nome, ciascuno viene chiamato "**core**", e il sistema nel suo complesso viene indicato come un **processore dual core**
- Col tempo e i miglioramenti tecnologici si è poi passati a processori quad-core e anche più, fino a 16 core e oltre, usati normalmente nei nostri dispositivi elettronici, non solo computer, ma smartphone, tablet, fotocamere, ecc.

1.3 Architetture single/multi-core

- Quindi attualmente, quando si parla di processore o CPU, si intende quasi sempre un processore **multi-core**, ossia un dual, quad, o anche più, core.

- È chiaro che un processore a N core è in grado di eseguire simultaneamente le istruzioni appartenenti a N programmi diversi: un programma per ciascun core.

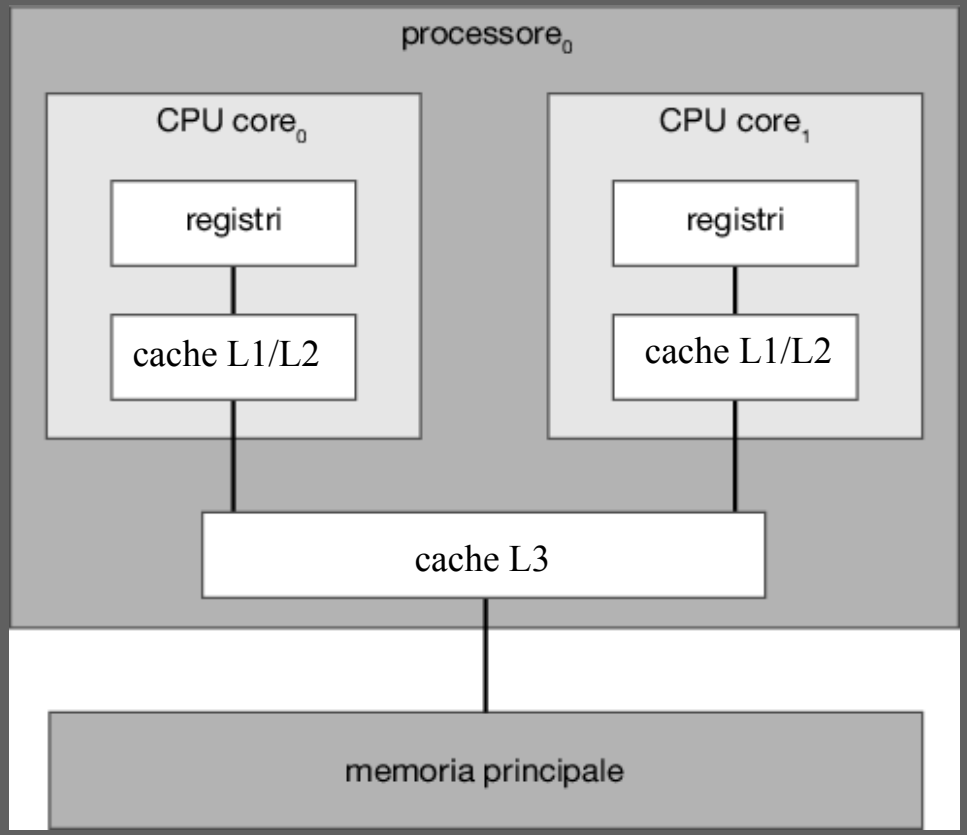
- Per mantenere uniforme la terminologia, un processore costituito da un'unica unità di esecuzione delle istruzioni sarà allora detto processore **single-core**

1.3 Architetture single/multi-core

- I moderni processori multi-core sono in sostanza dei piccoli sistemi UMA, in cui tutti i core possono indirizzare la stessa memoria principale.
- Tutti i core di un processore multi-core condividono però solitamente un livello di cache, il che fornisce notevoli vantaggi per lo scambio veloce di informazioni tra i vari core del processore multi-core
- Naturalmente, più processori multi-core possono poi essere messi assieme per formare un sistema multi-processore di tipo UMA, con una potenza di calcolo ancora maggiore.

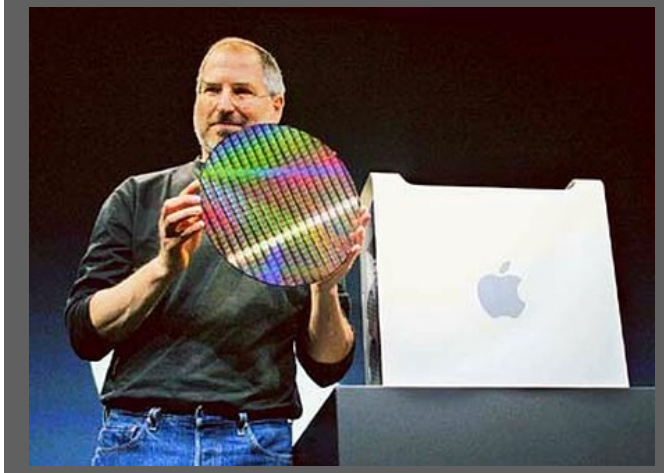
1.3 Architetture single/multi-core

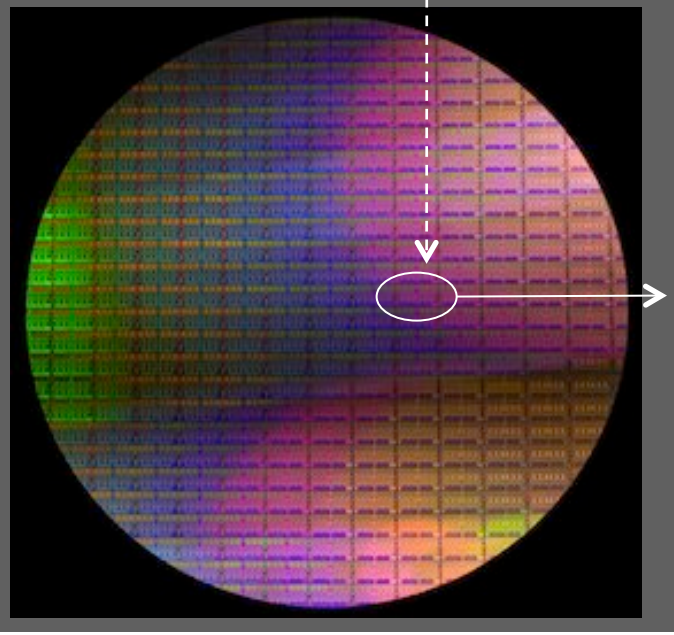
- Una architettura con processore dual-core, quindi con due core sullo stesso chip, due livelli di cache privata per core (L1 e L2) e un livello di cache condiviso (L3)

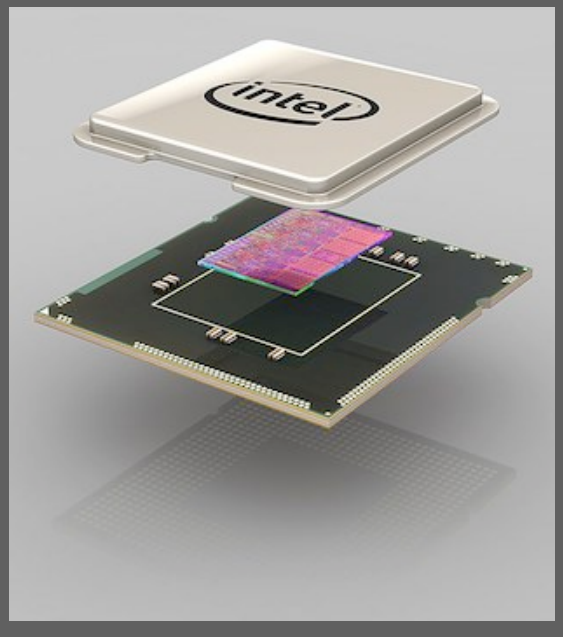


1.3 Architetture single/multi-core

- "wafer": una piastra di silicio su cui vengono prodotti centinaia di processori.
- Ogni rettangolino del wafer, il "die", contiene un singolo processore, che viene tagliato e inserito in un supporto di plastica e ceramica detto "package"

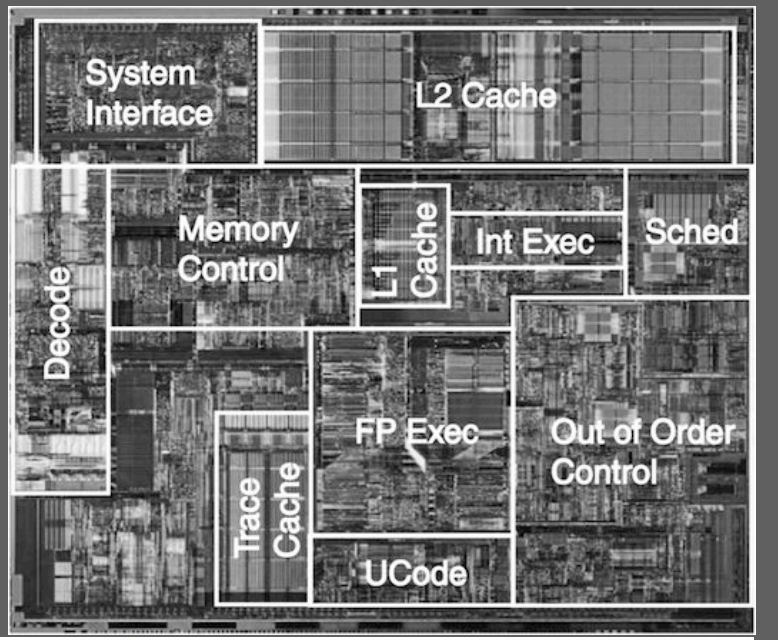




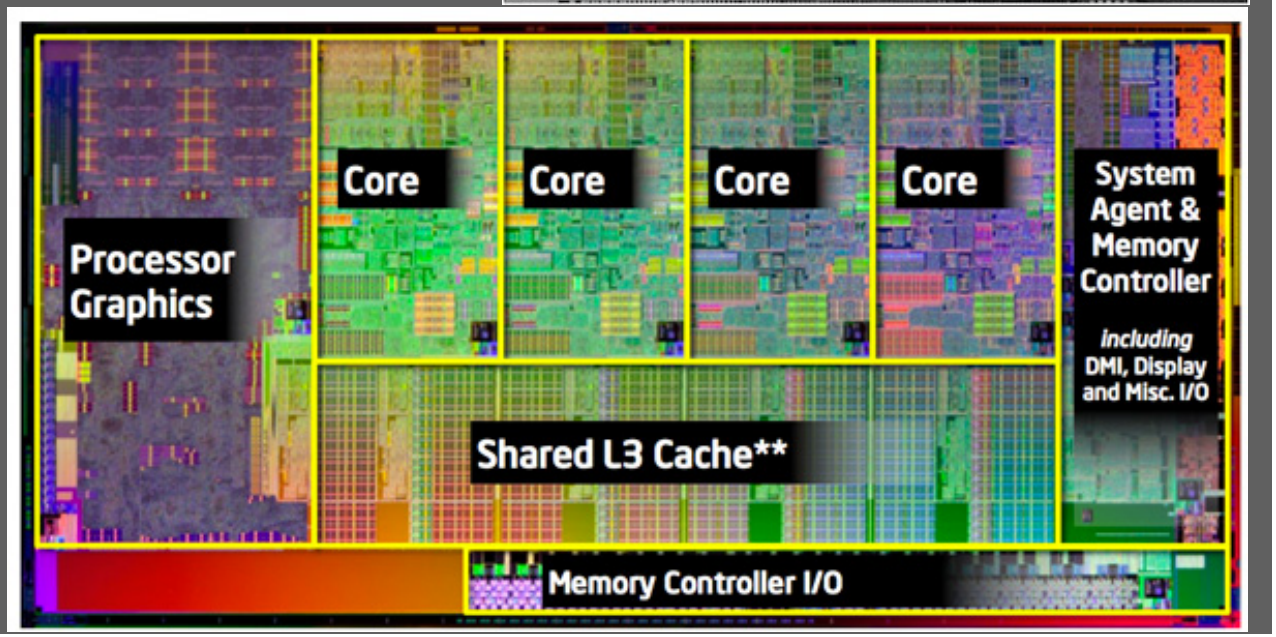


1.3 Architetture single/multi-core

- Un processore può quindi essere single-core (ad esempio come i vecchi Pentium 4)



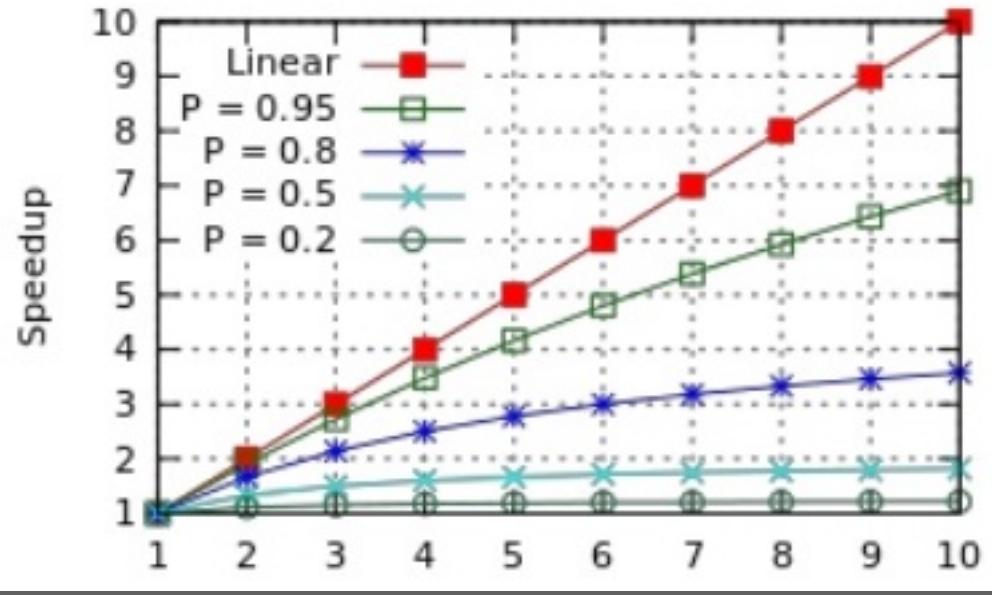
- o a 2 o più core, come i più recenti Intel i9, i7, i5 e i3



1.3 Architetture single/multi-core

- Ma, attenzione: le prestazioni di un processore N-core aumentano meno che linearmente rispetto al numero di core disponibili: **legge di Amdahl**

$$S = \frac{1}{1 - P + \frac{P}{N}}$$



Numero di core disponibili

1.1 Introduzione

- Nel corso studieremo le caratteristiche fondamentali dei tipici sistemi operativi che vengono usati in un generico computer moderno.
- Questi computer sono sistemi mono-core o sistemi multi-core, in cui almeno due (ma spesso più di due) core condividono lo stesso bus di sistema e la stessa memoria principale (e parzialmente anche la cache…)
- Concettualmente però, non esiste una grossa differenza tra SO per computer single-core o con processore multi-core.

1.1 Introduzione

- Semplicemente, il sistema operativo, se progettato per gestire più core, quando questi sono disponibili, può sfruttarli adeguatamente, distribuendo l'esecuzione dei programmi tra i vari core.

- Nel seguito, per semplificare la discussione, assumeremo sempre implicitamente che il sistema operativo stia girando su un computer che usa un processore single-core, ossia con una sola unità di esecuzione dei programmi

- Quando però sarà opportuno, faremo alcune osservazioni per il caso di computer con processori multi-core

1.1 Introduzione

- È importante osservare che, oltre ai sistemi operativi che girano sui nostri computer, esistono anche altri tipi di sistemi operativi, sviluppati per sistemi hardware più specifici.
- Ad esempio, anche nei tablet, negli smartphone, nelle foto e video camere, nei router, nelle console per video giochi girano dei sistemi operativi, con caratteristiche più o meno simili, ma a volte limitate rispetto a quelle dei SO di un generico computer, perché devono svolgere compiti più specifici di quelli svolti dal computer.

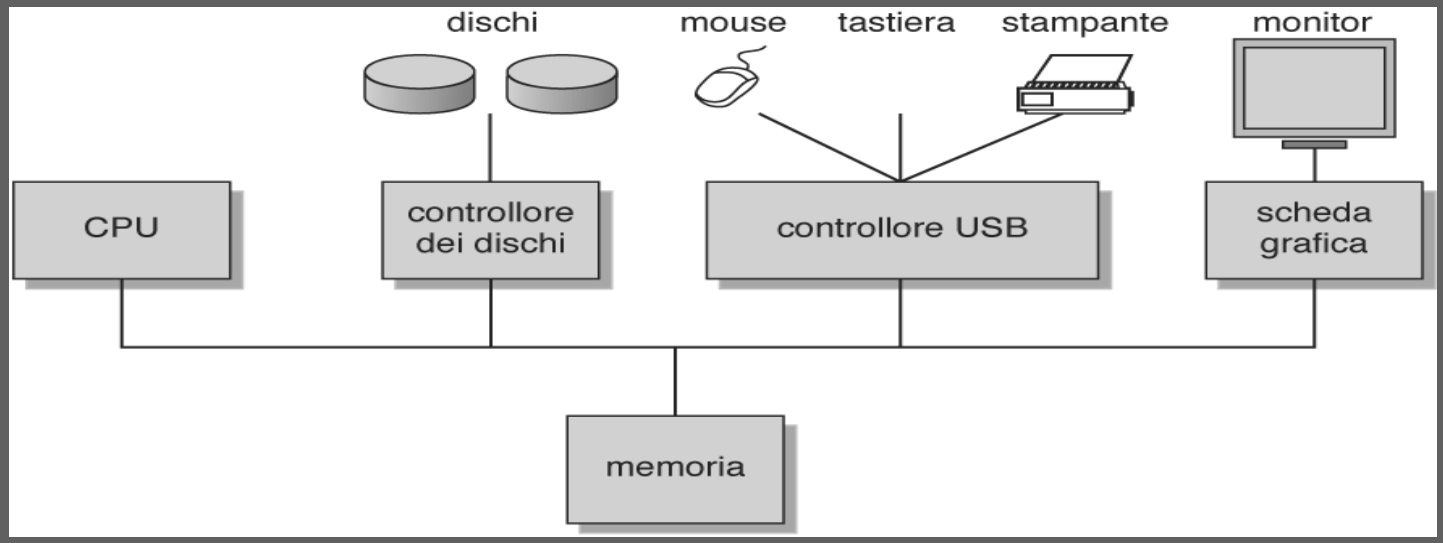
1.10 Sistemi Operativi di Rete e Sistemi Operativi Distribuiti

- Ed esistono poi sistemi operativi che sono una naturale estensione di quelli che studieremo, per gestire il funzionamento di sistemi hardware con molti processori che non condividono memoria, e in cui la comunicazione avviene "all'esterno" di ogni singola macchina.

- Questi sistemi possono fornire, come servizi aggiuntivi:
  - la condivisione di alcune risorse (stampanti, dischi)
  - un aumento della potenza di calcolo distribuendo il carico sulle diverse macchine
  - Maggiore affidabilità e usabilità (se una macchina si rompe o è occupata possiamo usarne un'altra)

1.2 Organizzazione di un Sistema elaborativo

- In questa parte del corso vediamo (o riassumiamo, dato che dovreste aver visto alcune di queste cose nel corso di Architetture I) alcuni aspetti dell'hardware di un computer che sono rilevanti per capire il funzionamento dei sistemi operativi (figura 1.2).



1.2.1 Interruzioni
(l'interazione tra l'hardware e il SO)

- Consideriamo un generico computer (acceso) su cui sia installato un generico sistema operativo.
- Che cosa fa per la maggior parte del tempo il SO?
- Niente. Le risorse del computer (CPU, memoria primaria e secondaria) vengono infatti utilizzate dai programmi dell'utente (o degli utenti) che sono collegati al computer.
- In fondo, il SO è attivo solo quando una qualche parte del suo codice sta girando nella CPU, e quando compriamo un computer non lo facciamo per farci girare sopra il SO, ma i nostri programmi...

1.2.1 Interruzioni

- Però di tanto in tanto questi programmi hanno bisogno dell'aiuto del SO, e lo "svegliano".
- Lo stesso SO ogni tanto si sveglia "di sua volontà" per verificare che tutto stia funzionando correttamente, e se è il caso interviene.
- Infine, a volte un programma utente si comporta in un modo che può danneggiare il sistema nel suo complesso, e il SO deve intervenire per gestire queste situazioni
- Più formalmente, diciamo che il SO è "**Event Driven**", guidato dagli **eventi** che si verificano nel sistema.

1.2.1 Interruzioni

- Quando si verifica un evento, il SO si sveglia per gestire l'evento, prendendo il controllo della macchina.
- In sostanza, entra in esecuzione la porzione di codice del SO che serve a gestire quel particolare evento.
- Quando la gestione dell'evento è completata, il SO restituisce il controllo ad uno dei programmi utente che stanno girando sul computer, e si riaddormenta in attesa dell'evento successivo.
- E' attraverso questo sistema di gestione degli eventi che il SO riesce a tenere sotto controllo la macchina anche quando non sta girando (con l'aiuto della CPU)

1.2.1 Interruzioni

- Distinguiamo due tipi di eventi: **Interrupt** ed **Eccezioni**.
- **Interrupt**: sono eventi di natura hardware, che si manifestano come segnali elettrici inviati da qualsiasi elemento dell'architettura del computer che richieda l'intervento del SO.
- **Eccezioni**: sono eventi di natura software, cioè causati dal programma in esecuzione, e devono essere gestiti dal SO

1.2.1 Interruzioni

- Le **Eccezioni** si distinguono a loro volta in:
  - **Trap**: sono dovute a malfunzionamenti del programma in esecuzione. Sono causate ad esempio da tentativi di accesso ad aree della RAM alle quali il programma non ha diritto di accedere, divisioni per 0…
  - **System Call**: (appunto, "chiamata al SO") richiesta di uno specifico servizio fornito dal SO (ad esempio, la richiesta di eseguire un'operazione su un file)

1.2.1 Interruzioni

- Quando si verifica un evento, la CPU reagisce eseguendo alcuni passi predefiniti a livello hardware:

1. **Viene salvato lo stato della computazione in corso:** Il valore corrente del Program Counter (PC) e degli altri registri della CPU viene salvato in appositi registri speciali della CPU

- In questo modo, il programma in esecuzione viene sospeso, e potrà essere riavviato da quel punto quando la gestione dell'evento (se non era una trap) sarà terminata.

1.2.1 Interruzioni

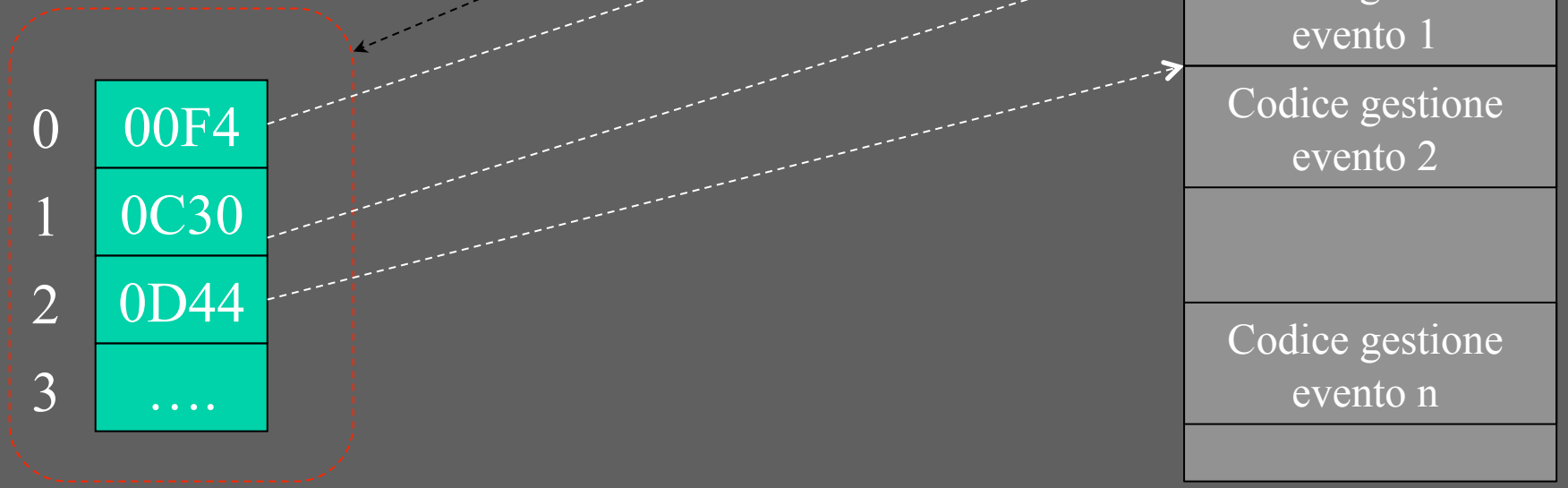
2. **Nel Program Counter viene scritto l'indirizzo in RAM della porzione di codice del SO che serve a gestire l'evento che si è appena verificato.**

- Vediamo in dettaglio come è implementato il meccanismo del punto 2:

  all'accensione del computer, come parte della fase di avvio del SO, il SO stesso carica in aree della RAM che il SO riserva a se stesso le varie porzioni di codice eseguibile che dovranno entrare in esecuzione quando si verifica un certo evento

1.2.1 Interruzioni

- Nei primi N indirizzi della RAM viene caricato un array di puntatori noto come **vettore delle interruzioni**. Ogni entry del vettore contiene l'indirizzo di partenza in RAM di una delle porzioni di codice del SO del punto precedente.



1.2.1 Interruzioni

- Gli eventi che si possono verificare durante l'uso della macchina (e che devono essere gestiti dal SO) sono rappresentati da un codice numerico: 0, 1, 2, ecc.

- Quando un certo evento si verifica, l'hardware della CPU copia nel Program Counter il contenuto della entry del vettore delle interruzioni corrispondente a quell'evento.

- Ma quello è proprio l'indirizzo di partenza della porzione di codice scritto per gestire quell'evento, e la CPU comincia ad eseguire quel codice.

1.2.1 Interruzioni

- L'ultima istruzione di ogni procedura di gestione di un evento sarà sempre una istruzione di "*return from event*", in modo da far ripartire l'esecuzione del programma che era stato temporaneamente sospeso per gestire l'evento.
- I valori del PC e degli altri registri della CPU salvati in precedenza (si veda il precedente punto 1) vengono ripristinati.
- L'esecuzione del programma riprende dal punto in cui era stato sospeso, mentre il SO si riaddormenta fino al prossimo evento.

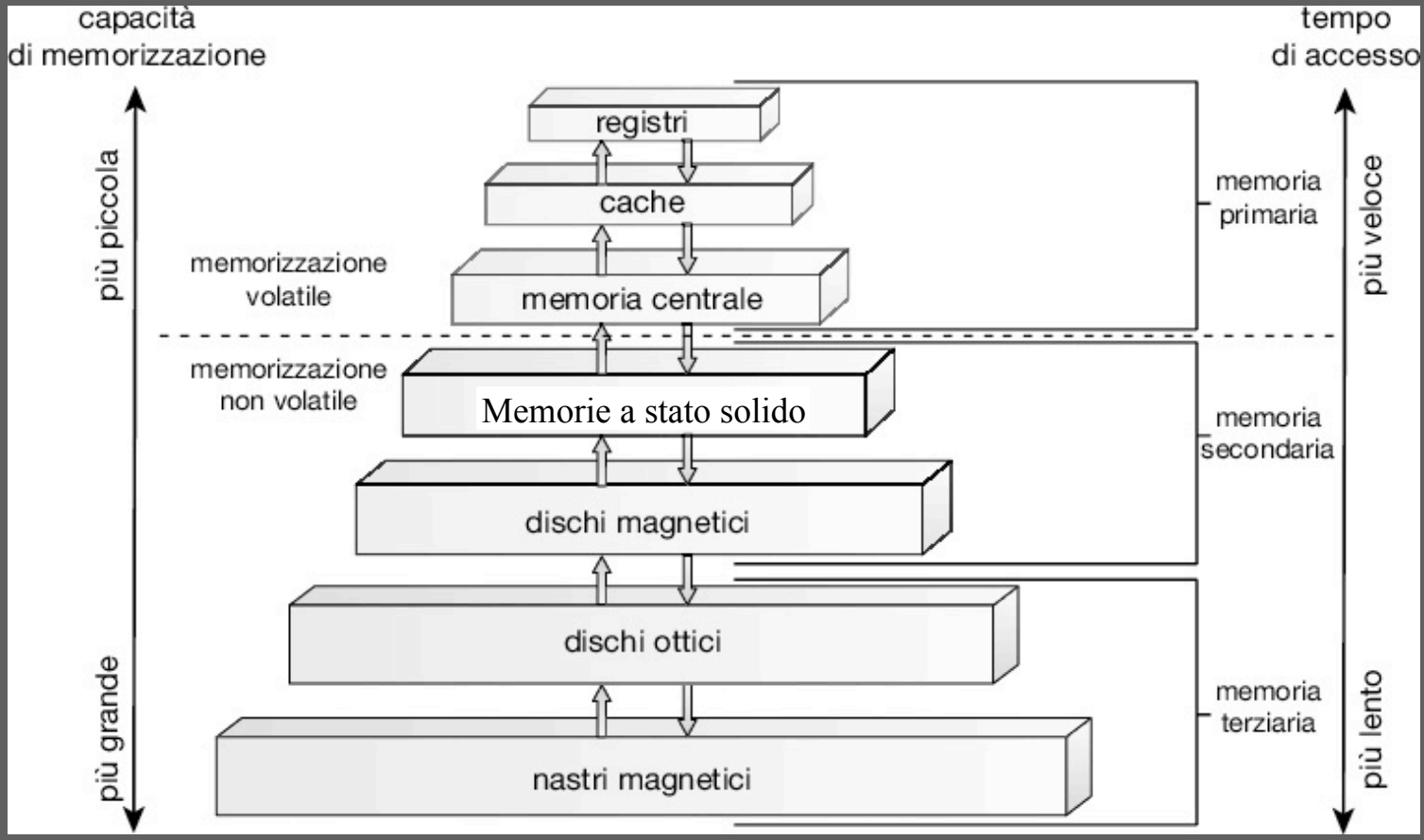
1.2.2 Struttura della Memoria

- Nel corso ci interesseremo principalmente a due tipi di memoria presente in un generico computer:

1. la **Memoria Principale**: (detta anche **primaria**, o **centrale**, o **RAM**) è la memoria indirizzata direttamente dalle istruzioni eseguite dalla CPU, e in cui devono risiedere i programmi per essere eseguiti e i dati usati da questi programmi

2. la **Memoria Secondaria**: (detta anche memoria **di massa**) che fornisce un ampio spazio di memorizzazione **permanente**, ed è solitamente realizzata con dischi magnetici: gli **hard disk,** o con **memorie a stato solido**

1.2.2 Gerarchia di Memorie



- Figura 1.6: Velocità implica complessità maggiore, costo maggiore, e di solito capacità minore.

1.8.3. Il concetto di caching

- Che bello sarebbe avere 500 GB di registri di CPU, o Hard Disk veloci quanto i registri della CPU…
- Purtroppo nel caso reale il luogo di immagazzinamento a lungo termine di una informazione non è il più conveniente per il suo uso a breve termine.
- Allora, per usare una informazione la si deve copiare in una memoria più veloce (ma più costosa e più ingombrante, e quindi più piccola) e più "vicina" al luogo in cui verrà elaborata.
- **La gerarchia delle memorie è una gerarchia di memorie cache, ognuna rispetto alla precedente**

1.8.3. Il concetto di caching

- Per essere eseguito un programma viene copiato dall'HD alla RAM: **la RAM fa da cache per l'HD**
- I dati (e le istruzioni) di piu' immediato utilizzo del programma in esecuzione vengono copiati dalla RAM alla memoria CACHE: **la CACHE fa da cache (ovviamente) per la RAM**
- L'istruzione attualmente in esecuzione e i dati che usa vengono copiati dalla CACHE nei registri appositi della CPU: (impropriamente parlando) **i registri fanno da cache per la CACHE…**

1.2.3. Struttura di I/O

(l'interazione tra i dispositivi di I/O e il SO)

- Un generico computer è composto da una CPU e da un insieme di dispositivi di I/O connessi fra loro da un bus comune
- Ogni dispositivo di I/O è controllato da un apposito componente hardware detto **controller**.
- Di fatto un controller è a sua volta un piccolo processore, con alcuni registri e una memoria interna, detta **buffer**.
- Il controller trasferisce i dati tra il dispositivo e il buffer
- Il SO interagisce con il controller (e quindi con il dispositivo di I/O) attraverso un software apposito noto come **driver del dispositivo**.

1.2.3. Struttura di I/O

- Per avviare una operazione di I/O il SO, tramite il driver del dispositivo, carica nei registri del controller opportuni valori che specificano le operazioni da compiere.
- Il controller esamina i registri e intraprende l'operazione corrispondente (ad esempio, "leggi un carattere digitato alla tastiera")
- Il controller trasferisce così i dati dal dispositivo (ad esempio i tasti digitati) al proprio buffer.
- A fine operazione il controller, inviando un opportuno interrupt, informa il driver (e quindi il SO) che i dati sono pronti per essere prelevati dal buffer.

1.2.3. Struttura di I/O

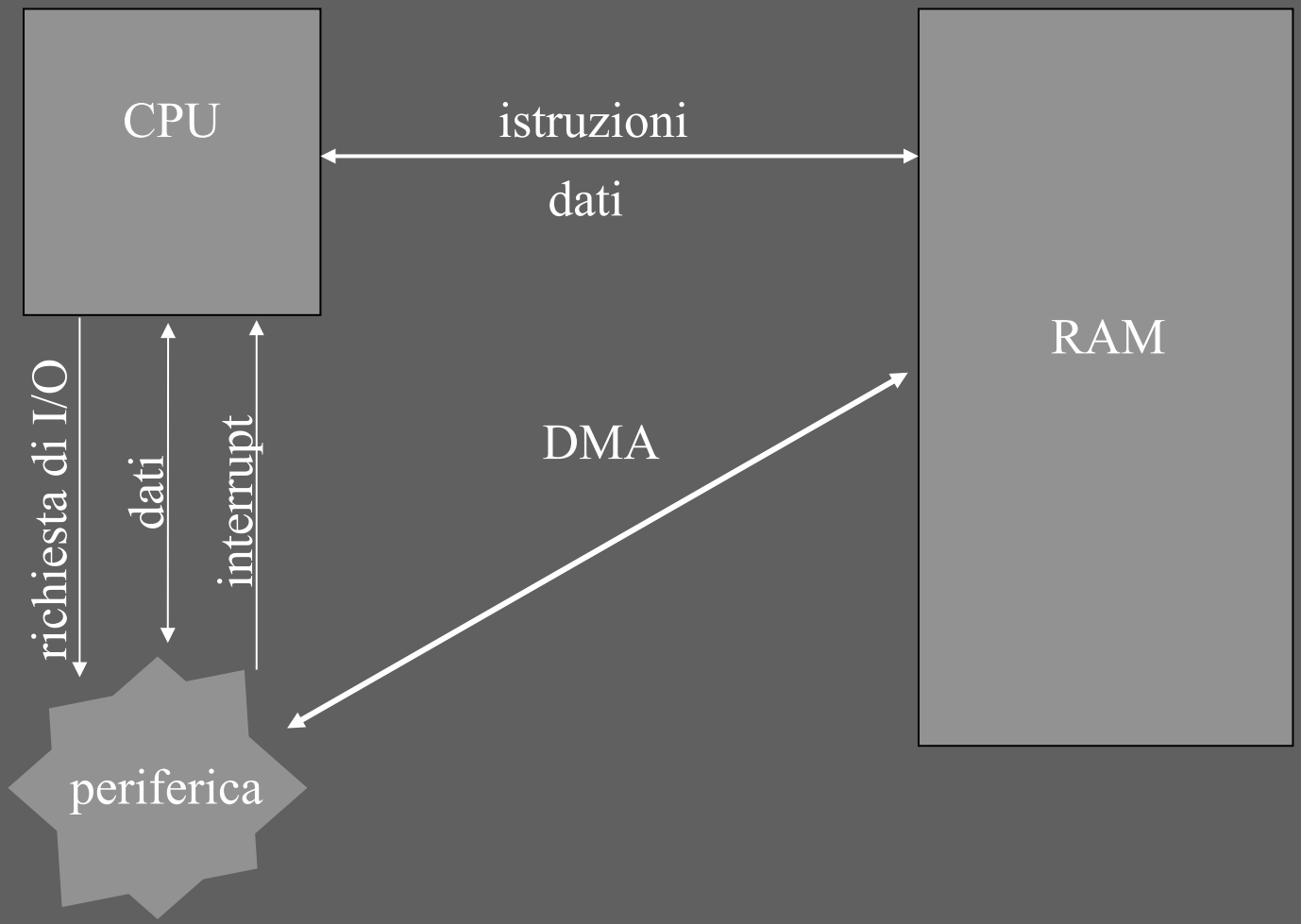
- Questo modo di gestire l'I/O è adeguato solo per piccole quantità di dati da trasferire.
- Quando la quantità di dati da trasferire è grande, questo modo di gestire l'I/O è inefficiente.
- E' il caso, ad esempio, delle comunicazioni tra la memoria secondaria (l'hard disk) e la memoria primaria (la RAM).
- I dati dall'hard disk al buffer del disco vengono trasferiti in blocchi di 1024 / 2048 / 4096 byte. Sarebbe poco pratico inviare un interrupt al SO *per ognuno dei byte che dal buffer vanno trasferiti alla RAM*

1.2.3. Struttura di I/O

- In tali casi è utile avere un canale di comunicazione *diretto* tra il dispositivo e la RAM, in modo da "non coinvolgere troppo" il SO.
- Tale canale è detto **"Direct Memory Access" (DMA)**
- Il SO, tramite il driver del disco, istruisce opportunamente il controller del disco, con un comando (scritto nei registri del controller) del tipo:
  - Trasferisci il blocco numero 1000 del disco in RAM a partire dalla locazione di RAM di indirizzo FFFF
- Il controller trasferisce direttamente il blocco in RAM usando il DMA, e ad operazione conclusa avverte il SO mediante un interrupt opportuno.

1.2.3. Struttura di I/O

- Figura 1.7: relazione tra CPU, RAM e dispositivi di I/O



1.4.1 Multitasking e Time-sharing

- *(Nota: i concetti introdotti in questa sezione, multitasking e time-sharing, sono a volte chiamati con terminini diversi: ad esempio nel testo con "multiprogrammazione" e con "multitasking (cooperativo)". Di solito è chiaro dal contesto a quale concetto ci si sta riferendo)*
- Quando lanciamo un programma, il SO cerca il codice del programma sull'hard disk, lo copia in RAM, e "fa partire il programma" (notate le virgolette)
- Noi utenti del SO non dobbiamo preoccuparci di sapere dov'è memorizzato il programma sull'hard disk, né dove verrà caricato in RAM per poter essere eseguito.

1.4.1 Multitasking e Timesharing

- Dunque, il SO rende **facile** l'uso del computer. Ma il SO ha anche il compito di assicurare un uso **efficiente** delle risorse del computer, in primo luogo la CPU stessa.
- Consideriamo un programma in esecuzione: a volte deve fermarsi temporaneamente per compiere una operazione di I/O (esempio: leggere dall'hard disk dei dati da elaborare).
- Fino a che l'operazione non è completata, il programma non può proseguire la computazione, e non usa la CPU.
- Invece di lasciare la CPU inattiva, perché non usarla per far eseguire il codice di un altro programma (ammesso che ce ne sia uno da far girare)?

1.4.1 Multitasking e Timesharing

- Questo è il principio della **multiprogrammazione** (in inglese: **multitasking**), implementato da tutti i moderni SO: il SO mantiene in memoria principale il codice e i dati di più programmi che devono essere eseguiti (figura 1.12).
- Quando un programma si ferma temporaneamente, il SO ha già in RAM un altro programma a cui assegnare la CPU.
- La conseguenza di questo modo di gestire le risorse della macchina (la CPU, ma anche la RAM) è che la loro produttività aumenta.

1.4.1 Multitasking e Timesharing

- Alcune applicazioni degli utenti però sono per loro natura **interattive**. Pensate ad un editor di testi, ad un programma di mail, o ad un browser; c'è un'interazione continua tra il programma e l'utente che lo usa.
- Inoltre, i sistemi di calcolo (Hardware+SO) **multi-utente** permettono a più utenti di essere connessi al sistema e di usare **"contemporaneamente"** il sistema stesso.
- Che succede se il programma che sta attualmente utilizzando la CPU non si ferma mai per compiere delle operazioni di I/O (rilasciando così la CPU, in modo che possa essere usata da qualche altro programma/utente)?

1.4.1 Multitasking e Timesharing

- È meglio allora distribuire il tempo di CPU fra i diversi utenti (i loro programmi in "esecuzione") frequentemente (ad esempio ogni 1/10 di secondo) così da dare una impressione di simultaneità (che però è solo apparente).
- Questo è il **time-sharing**, che estende il concetto di multiprogrammazione, ed è implementato in tutti i moderni sistemi operativi
- Ora non dobbiamo più aspettare che un programma decida di fermarsi per una operazione di I/O per assegnare la CPU ad un altro programma. Ci pensa il SO (come lo vediamo tra poco).

1.4 Compiti del Sistema Operativo

- Un moderno sistema time sharing, deve quindi risolvere alcuni problemi fondamentali:

1. E' necessario tenere traccia di tutti i programmi attivi nel sistema, che stanno usando o vogliono usare la CPU, e gestire in modo appropriato il passaggio della CPU da un programma all'altro, nonché come lanciare nuovi programmi e gestire la terminazione dei vecchi.
   - Questo è il problema della gestione di **processi** (cap. 3) e dei **thread** (cap. 4)
2. Quando la CPU è libera, e più programmi la vogliono usare, a quale programma in RAM assegnare la CPU?
   - Questo problema è noto come **CPU scheduling** (cap. 5)

1.4 Compiti del Sistema Operativo

3. I programmi in esecuzione devono interagire fra loro senza danneggiarsi ed evitando situazioni di stallo (ad esempio, il programma A aspetta un dato da B che aspetta un dato da C che aspetta un dato da A)
   - Questi sono i problemi di **sincronizzazione** (cap. 6/7)
   - e di **deadlock** (stallo dei processi) (cap. 8)
4. Come gestire la RAM, in modo da poterci far stare tutti i programmi che devono essere eseguiti? Come tenere traccia di quali aree di memoria sono usate da quali programmi?
   - La soluzione a questi problemi passa attraverso i concetti di gestione della **memoria centrare** (cap. 9)
   - e di **memoria virtuale** (cap. 10).

1.4 Compiti del Sistema Operativo

5. Infine, un generico computer è spesso soprattutto un luogo dove gli utenti memorizzano permanentemente, organizzano e recuperano vari tipi di informazioni, all'interno di "contenitori" detti **file**, a loro volta suddivisi in **cartelle** (o **folder**, o **directory**) che sono organizzate in una struttura gerarchica a forma di albero (o grafo aciclico) nota come **File System**.
   - Il SO deve gestire in modo efficiente e sicuro le informazioni memorizzate sulla **memoria di massa (o secondaria)** (cap. 11).
   - deve permettere di organizzare i propri file in modo efficiente, ossia fornire una adeguata interfaccia col file system (cap. 13),
   - Deve **implementare il file system** (cap. 14)

come fa il SO a mantenere sempre il controllo della macchina?

- Soprattutto, come fa anche quando non sta girando (ossia quando la CPU è usata da qualche programma utente?)
- Ad esempio, come evitare che un programma utente acceda direttamente ad un dispositivo di I/O usandolo in maniera impropria?
- Oppure, che succede se un programma, entra in un loop infinito?
- E' necessario prevedere dei modi per proteggersi dai malfunzionamenti dei programmi utente (voluti, e non)

1.4.2 Duplice modalità di funzionamento

- Nei moderni processori le istruzioni macchina possono essere eseguite in due modalità diverse:
- **normale** (detta anche modalità **utente**)
- di **sistema** (detta anche modalità **privilegiata**, o **kernel / monitor / supervisor** mode)
- La CPU è dotata di un "bit di modalità" di sistema (0) o utente (1), che permette di stabilire se l'istruzione corrente è in esecuzione per conto del SO o di un utente normale.

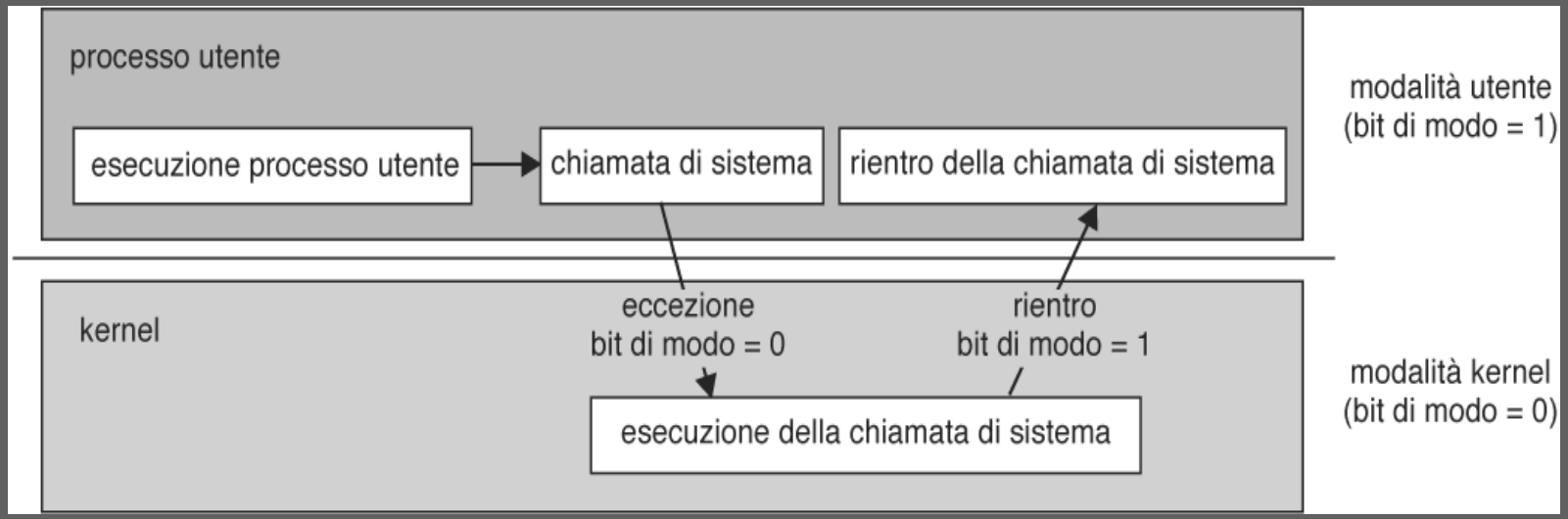
1.4.2 Duplice modalità di funzionamento

- Le istruzioni macchina “delicate”, nel senso che se usate male possono danneggiare il funzionamento del sistema nel suo complesso, possono essere eseguite solo in *modalità di sistema*, e quindi solo dal SO, altrimenti...
- Se nel codice di un programma normale in esecuzione è contenuta una istruzione delicata, quando questa istruzione entra nella CPU viene generata una trap
- Il controllo viene automaticamente trasferito al SO che interrompe l’esecuzione del programma

1.4.2 Duplice modalità di funzionamento

- I programmi utente hanno a disposizione le **system call (chiamate di sistema)** per compiere operazioni che richiedono l'esecuzione di istruzioni privilegiate
- Una system call si usa in un programma come una normale subroutine, ma in realtà **provoca una eccezione**, ed il controllo passa al codice del SO di gestione di quella eccezione.
- Ovviamente, quando il controllo passa al SO, il bit di modalità viene settato in modalità di sistema in modo automatico (ossia via hardware).

1.4.2 Duplice modalità di funzionamento



- Il SO gestisce la richiesta e infine restituisce il controllo della CPU al programma utente, riportando il bit di modalità in modalità utente. (fig. 1.13)
- Si dice di solito che il processo utente sta eseguendo in **kernel mode**

1.4.3 Timer

- L'uso di una doppia modalità di esecuzione delle istruzioni, e il fatto che le istruzioni macchina delicate (perché potenzialmente pericolose) possano essere eseguite solo in modalità privilegiata protegge il sistema da molti problemi che possono essere causati dai programmi utente (volutamente o accidentalmente)
- Ma che succede se un programma utente, una volta ricevuto il controllo della CPU, si mette ad eseguire il seguente codice?

```
for(; ;) i++;        // an endless loop!
```

1.4.3 Timer

- il programma non sta facendo nulla di pericoloso, incrementa semplicemente una variabile in un ciclo che non termina, e quindi non sembra molto intenzionato a rilasciare mai la CPU...
- Per evitare questo tipo di problemi, nella CPU è disponibile un **Timer**, che *viene inizializzato con la quantità di tempo che si vuole concedere* ***consecutivamente*** *al programma in esecuzione*.
- Il sistema operativo inizializza il Timer (ad esempio, ad 1/10 di secondo) e poi lascia la CPU dandola al programma che vuole mandare in esecuzione.

1.4.3 Timer

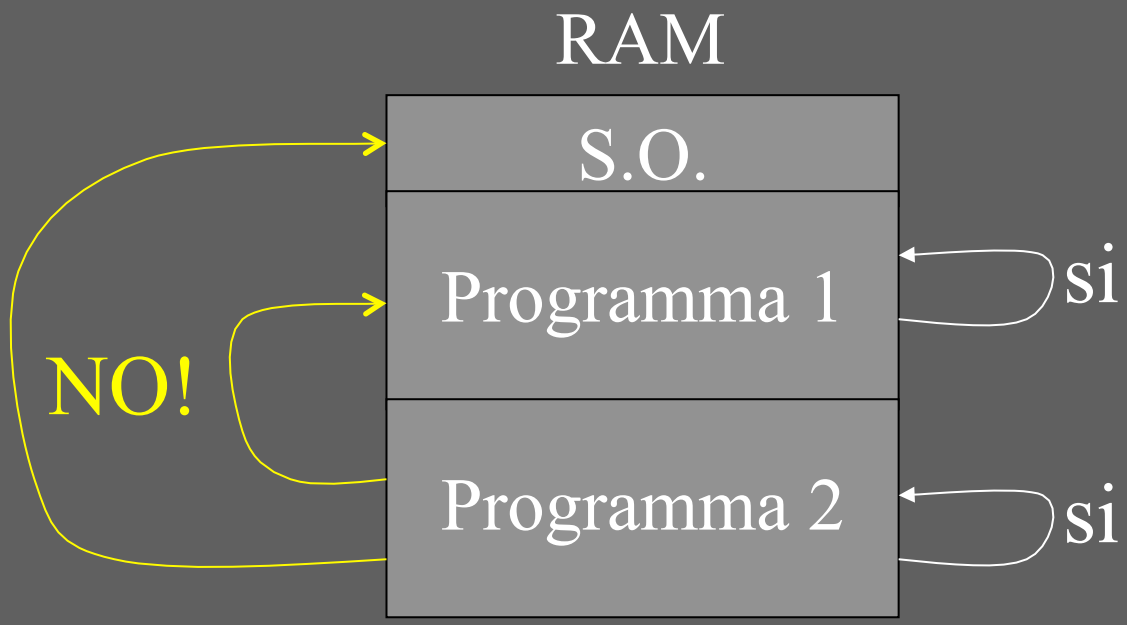
- Qualsiasi cosa faccia il programma in esecuzione, dopo 1/10 di secondo il Timer invia un interrupt alla CPU, e il controllo viene restituito al sistema operativo.
- Il SO verifica che tutto stia procedendo regolarmente, ri-inizializza il Timer e decide quale programma mandare in esecuzione (potrebbe anche essere quello di prima...)
- Come vedremo, questa è l'essenza del time-sharing
- **Ovviamente, le istruzioni macchina che servono per gestire il timer sono istruzioni privilegiate (altrimenti, cosa potrebbe succedere?)**

Protezione della Memoria

- Il testo non lo dice esplicitamente nel capitolo 1 (se ne parla poi nella sezione 9.1.1), ma c'è almeno un'altra cosa importantissima che il SO deve fare per assicurarsi sempre il controllo della macchina.
- Infatti, che succede se un programma in esecuzione sovrascrive i dati di un altro programma in "esecuzione"?
- Peggio ancora, che succede se un programma va a scrivere nelle aree dati del Sistema Operativo?
- E' necessario proteggere la memoria primaria da accessi ad aree riservate.

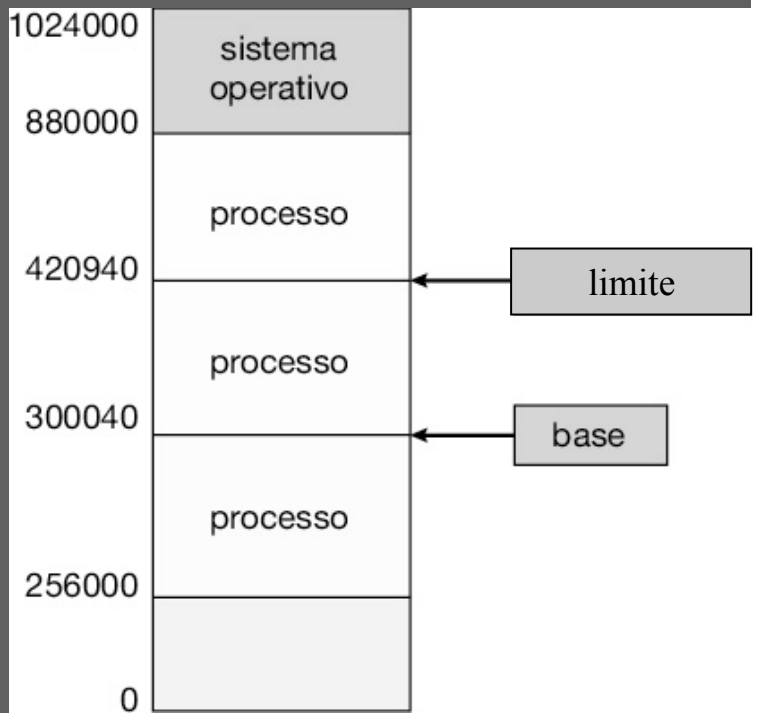
Protezione della Memoria

- Vogliamo evitare che un programma vada a leggere e/o modificare i dati di altri programmi, incluso soprattutto il sistema operativo



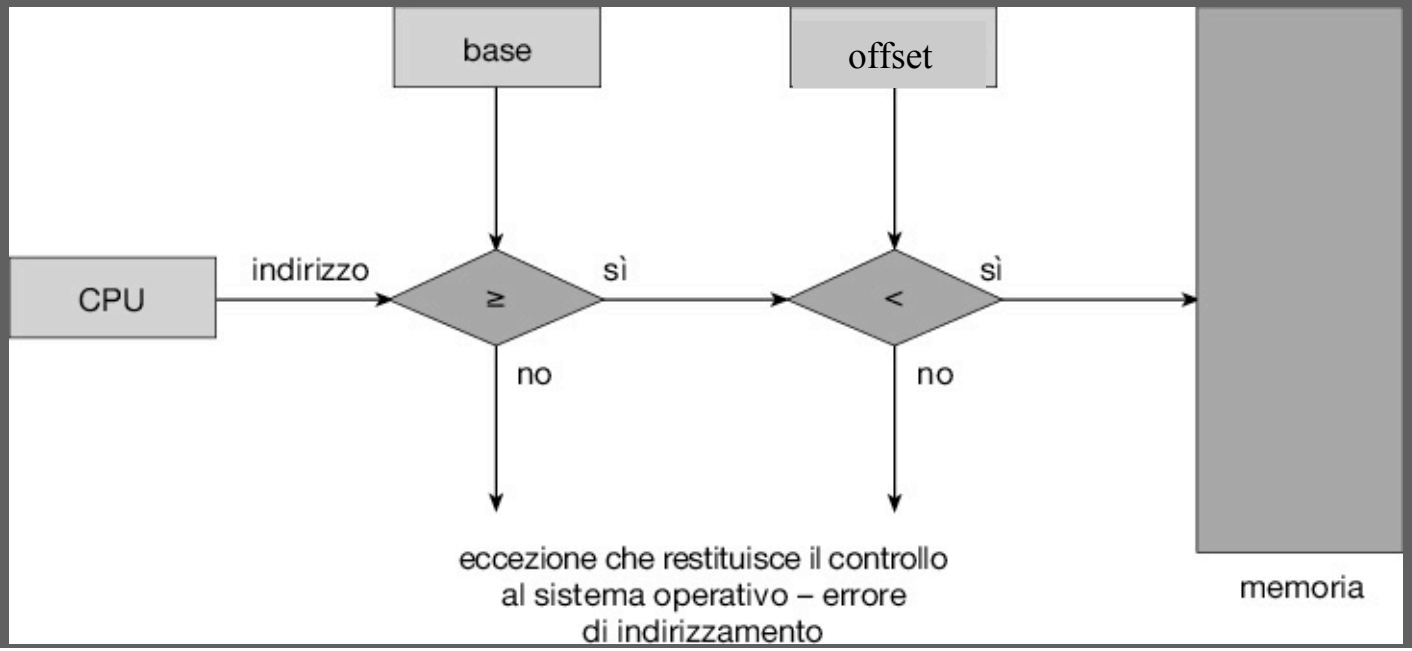
Protezione della Memoria

- Una possibile soluzione: in due registri appositi della CPU (**base** e **limite**) il SO carica gli indirizzi di **inizio** e **fine** dell'area di RAM assegnata ad un programma
- Ogni indirizzo *I* generato dal programma in esecuzione viene confrontato con i valori contenuti nei registri base e limite.
- Se $I <$ base $\|\| I >$ limite $\Rightarrow$ TRAP!
- Ovviamente i controlli vengono fatti in parallelo e a livello hardware, altrimenti richiederebbero troppo tempo (fig. 9.1 modificata)



Protezione della Memoria

- Una soluzione simile: in due registri appositi della CPU il SO carica rispettivamente l'indirizzo di **inizio (base)** e la **dimensione (offset)** dell'area di RAM assegnata ad un programma (fig. 9.2).
- Per ogni ind. *I*: Se $I < \text{base} \,\|\, I > \text{base+offset} \Rightarrow$ TRAP!



Protezione della Memoria

- In realtà quelle che abbiamo visto sono soluzioni molto semplici e i sistemi operativi moderni possono adottare soluzioni anche molto più sofisticate.

- Ma le soluzioni viste sono in ogni caso alla base dei meccanismi di protezione che studieremo nel capitolo 9 sulla gestione della memoria principale.

Riassumendo:

- I SO svolgono il ruolo di **mediatori** fra gli utenti e **la macchina** per:
  - fornire un ambiente più conveniente per l'esecuzione dei programmi
  - aumentare l'efficienza dell'utilizzo dell'hardware
- I SO evolvono continuamente sotto la pressione di:
  - Necessità e richieste degli utenti
  - caratteristiche dell'hardware
  - costo dell'hardware

Per chi vuole approfondire:

- Alla fine di ogni capitolo vi verranno proposti alcuni argomenti di approfondimento che trovate sul libro di testo (i numeri fanno riferimento alla 10° edizione). Questi argomenti possono completare la vostra preparazione e dunque siete invitati a leggerli, ma gli approfondimenti suggeriti non fanno parte del programma del corso, e non verranno chiesti all'esame.

- Sezione 1.7: Virtualizzazione
- Sezione 1.10: Ambienti d'elaborazione
- Sezione 1.11: Sistemi operativi liberi e open-source