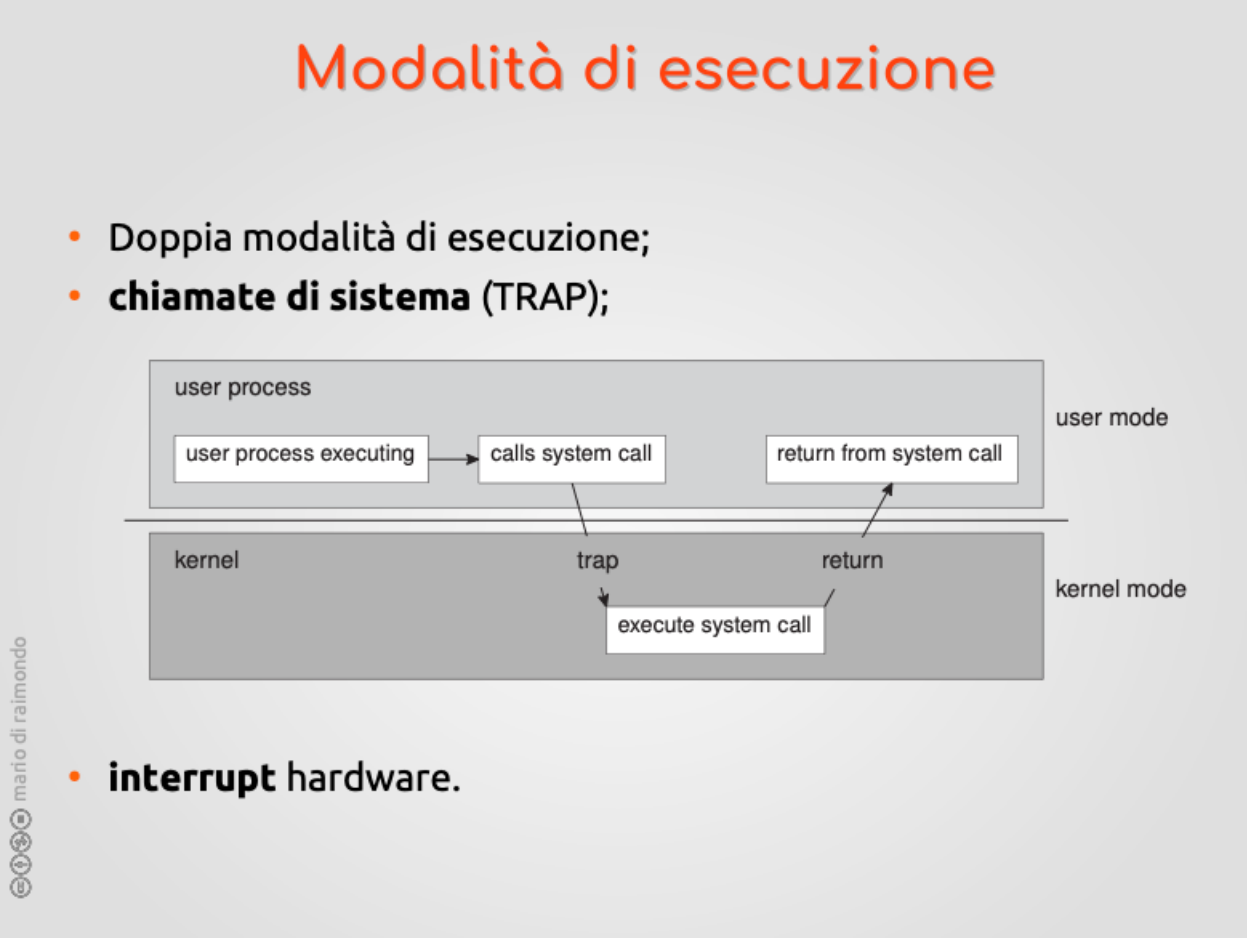
CPU superscalare

Lezione 10 Marzo



Gli utenti del sistema operativo sono i processi(system call);

Le librerie sono raccolte di codice già scritto e riutilizzabile, collezioni di istruzioni precompilate che possiamo utilizzare, ad esempio anziché andare a costruirsi una struttura dati (heap, pila, lista ecc...) possiamo usare questi sistemi già pronti che altri hanno già creato in precedenza (il vantaggio oltre al tempo risparmiato e ad avere un minor numero di errori è che le istruzioni di libreria sono ottimizzate in modo da avere una complessità il più bassa possibile)

Una **system call** è il meccanismo con cui un programma utente chiede un servizio al sistema operativo, eseguito in modalità kernel, i programmi normali (excel, word, chrome) girano in user mode, quindi non possono accedere direttamente

all'hardware o a risorse sensibili, per fare queste operazioni devono chiedere al kernel tramite una system call

Esempi di system call:

- usare hardware(tastiera, lettore dischi)
- allocare memoria
- creare processi

La **TRAP** è un eccezione usata per passare dalla user mode alla kernel mode quando si deve fare una system call generata dal programma in esecuzione. È il compilatore che effettua il “salto” (anche se l’indirizzo finale non è specificato).

Si segue questo ordine: il programma è in esecuzione in user mode (CPU in user mode), avviene una system call (inizia la TRAP).quindi la CPU passa in kernel mode. Si esegue questa system call e poi si ritorna alla user mode. Quando si esegue questo switch la CPU si salva il contesto precedente alla TRAP, in modo da poter ritornare a eseguire correttamente le operazioni una volta tornata in user mode.
Cosi qualunque processo può fare richiesta al sistema operativo che poi sceglierà se
soddisfarla o no.

In questa fase, il sistema deve garantire di poter tornare esattamente al punto di partenza: per questo motivo, viene salvato nello **stack** tutto ciò che è utile per il rientro. In particolare:

- **Il Program Counter (PC):** che punta all'istruzione successiva da eseguire nel programma utente.
- **La Program Status Word (PSW):** fondamentale perché memorizza lo stato della CPU e la modalità attuale (che in quel momento è "User Mode").

La TRAP non permette al programma di saltare a un indirizzo di memoria qualsiasi (sarebbe un rischio di sicurezza), ma lo obbliga a saltare a una **procedura prestabilita del kernel**, ovvero il codice che implementa la funzione richiesta (ad esempio la funzione `OPEN` per aprire un file).Ora il codice è sotto il controllo totale del Sistema Operativo, che decide se soddisfare la richiesta. Una volta terminata

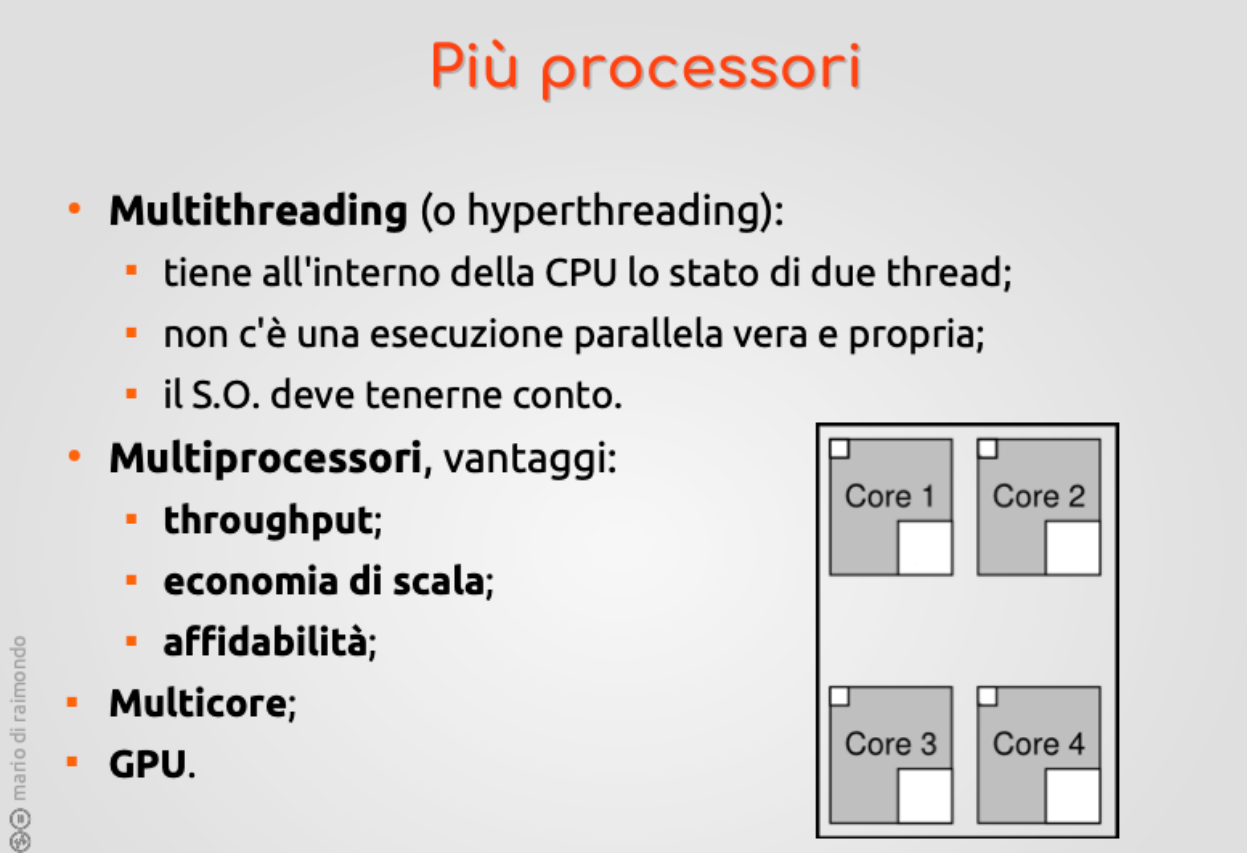
l'operazione, il controllo torna al programma utente (es. il tuo editor di testo) tramite un'istruzione di **return**: il sistema recupera dal registro PC l'indirizzo che era nello stack, avendo cura di **ripristinare la modalità utente limitata** tramite la PSW precedentemente salvata.

L' **interrupt hardware** è simile ad un operazione di TRAP ma differisce per il fatto che l'interrupt è mandato dall'hardware (anziché dal software con la system call), gli interrupt sono un meccanismo che permette di notificare qualcosa alla CPU, interrompe quindi quello che sta facendo in quel momento per eseguire la routine associata all'interrupt (è questo ad esempio il motivo per cui la freccia del mouse è sempre fluida senza mai bloccarsi anche quando il computer è sotto sforzo, perché il mouse lancia un interrupt e si fa riservare un piccolo spazio di esecuzione per essere eseguito senza problemi). La CPU salva il minimo indispensabile per tornare a riprendere quello che stava facendo prima (salvataggio del contesto quindi Program Counter,PSW...) in modo simile alla TRAP, la CPU riprende il suo lavoro. In generale un interrupt non deve procurare problemi, tutto poi deve continuare normalmente.

Ogni **Kernel** contiene il codice necessario per gestire gli interrupt, che viene sempre eseguito in **modalità kernel** (privilegiata). Gli eventi che scatenano un interrupt si dividono in due categorie:

1. **Eventi Sincroni (Eccezioni/Trap):** Sono generati direttamente dall'esecuzione di un'istruzione del programma (es. **divisione per zero** o un'**istruzione illegale** non interpretabile).
2. **Eventi Asincroni (Interrupt Hardware):** Sono generati da cause esterne al flusso del programma (es. segnali dai dispositivi hardware).

Se l'evento sincrono rappresenta un **errore fatale** che il sistema non può gestire, il Kernel interviene **terminando forzatamente il processo** per proteggere l'integrità del sistema operativo.



Tra i vari meccanismi possiamo trovare **multithreading**, si riferisce all' ottimizzazione di una CPU. Questo escamotage prevede di implementare un doppio contesto di esecuzione all'interno dello stesso core.

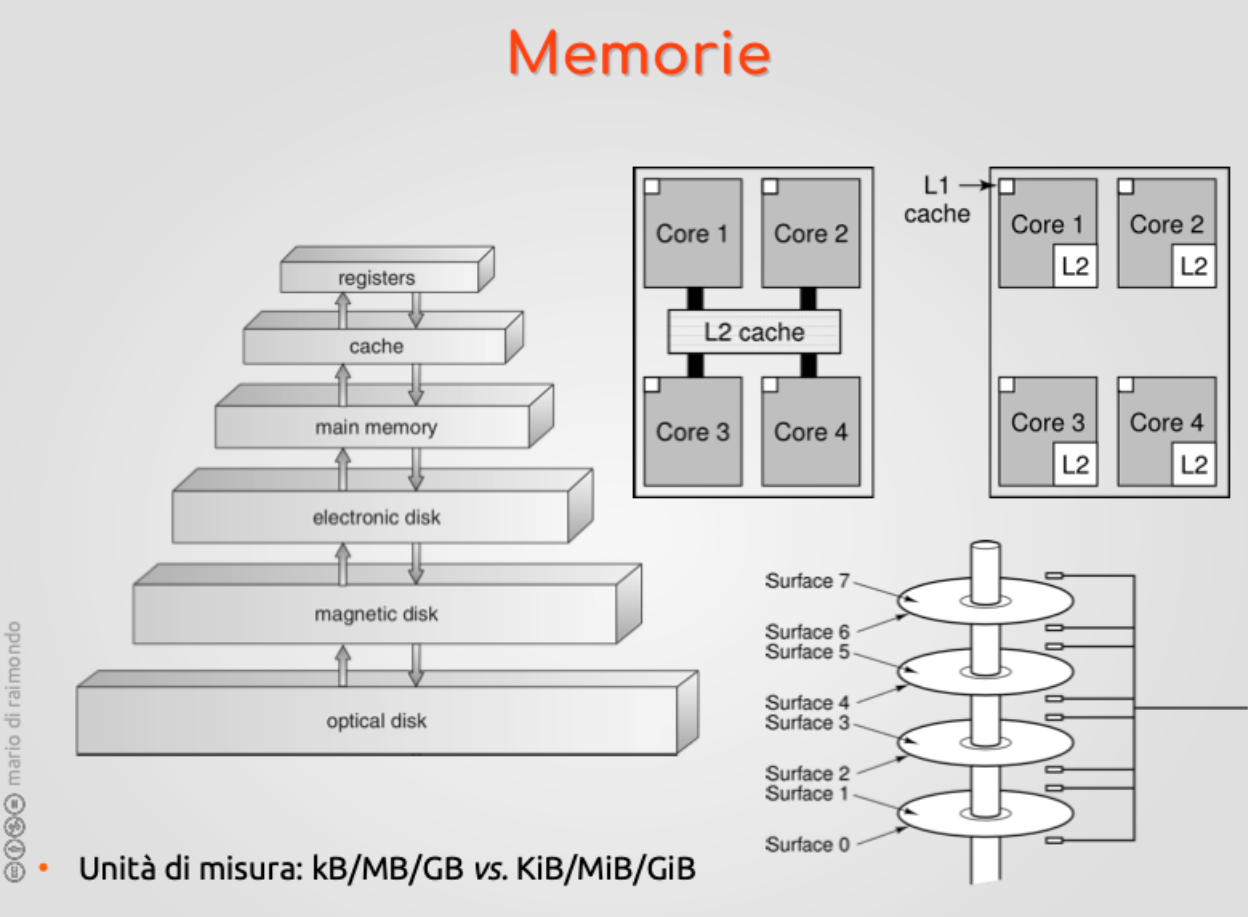
L'idea è: nel ciclo di decode execute ci sono tempi morti(quando la CPU deve accedere ad allocazioni vuote della RAM), nei registri vengono caricati i set di valori associati a due diversi processi: se ho il processo P1 che implica un certo numero di cicli affinché un suo fetch venga concluso, nel tempo morto verranno eseguiti degli step del P2. Sembra una complicazione, ma permette di lavorare su processi diversi in modo più efficace, ***ma questa soluzione non è parallelismo***, non ce una doppia CPU, il multithreading viene effettuato sullo stesso core. in un dato istante si attenziona o uno o l'altro processo. In questo caso l'hardware 'finge' di avere due CPU (CPU logiche) esponendo al sistema operativo un doppio set di registri. Il sistema operativo deve però essere in grado di distinguere tra una CPU reale e una logica: se avesse solo due processi da eseguire, sarebbe un errore caricarli entrambi sui due set di registri della stessa CPU fisica, lasciando un'altra CPU totalmente ferma. Il SO deve quindi 'capire' la struttura reale per

evitare utilizzi inefficienti delle risorse.Il sistema operativo deve però capire che l' ALU è sempre la stessa evitando di portare a utilizzi strani della risorsa, evitando quindi di bloccare i processi.

Un altro sistema sono i **multiprocessori** (***da notare NON multicore***) in cui aggiungo più CPU in modo da poter ***parallelizzare*** certi processi, accoppiando al multithreading e ad una buona pipeline si può migliorare molto la velocità di esecuzione. Questa scelta è stata fatta perché non si riesce a espandere all'infinito la cache o i registri ne si riescono ad aumentare troppo i GigaHertz della CPU, perché si va incontro a consumi elevati e problemi di surriscaldamento (al massimo si arrivare sui 5.5/6 GHz). Tuttavia, avere ad esempio 4 CPU che condividono un'unica RAM e un unico bus non garantisce una velocità 4x netta. Il vantaggio economico è evidente, ma lo svantaggio è che si creano accodamenti e tempi morti nell'accesso alle risorse. Diverso è il discorso del **multicore** in cui anziché avere ad esempio 4 CPU distinte montate sulla scheda madre abbiamo più core (e quindi più ALU) all'interno della stessa CPU, soluzione utilizzata in moltissimi scenari consumer, a differenza invece dei data center che spesso usano anche la tecnica del multiprocessori.

Dal punto di vista del Sistema Operativo, però, multicore e multiprocessore sono praticamente la stessa cosa: al kernel interessa solo il numero totale di unità di elaborazione disponibili, non come sono fisicamente impacchettate.

Infine, parlando di elaborazione parallela, esistono le GPU. Queste hanno molte più unità di elaborazione delle CPU, ma il sistema operativo di norma 'non ci mette bocca': la loro gestione è demandata direttamente alle applicazioni e ai driver specifici, restando fuori dal controllo diretto delle politiche del kernel.



Memorie che possiamo trovare, dalla più **efficiente ma costosa** (registri) alla **meno efficiente ma più economica** (dischi):

- Registri: memoria velocissima riesce a stare al passo della CPU è molto piccola
  (nell'ordine di pochi byte) e contiene i dati che servono ai processi per poter essere eseguiti, si trova dentro la CPU
- Cache: memoria anch'essa piccola e veloce posizionata dentro la CPU, a differenza dei registri la cache è più capiente (siamo nell'ordine dei MegaByte, le più grandi arrivano sui 100 mega circa) ma anche più lenta dei registri, restando comunque la seconda memoria in termini di velocità e dimensione, la cache è divisa in 3 livelli: L1, L2, L3 .

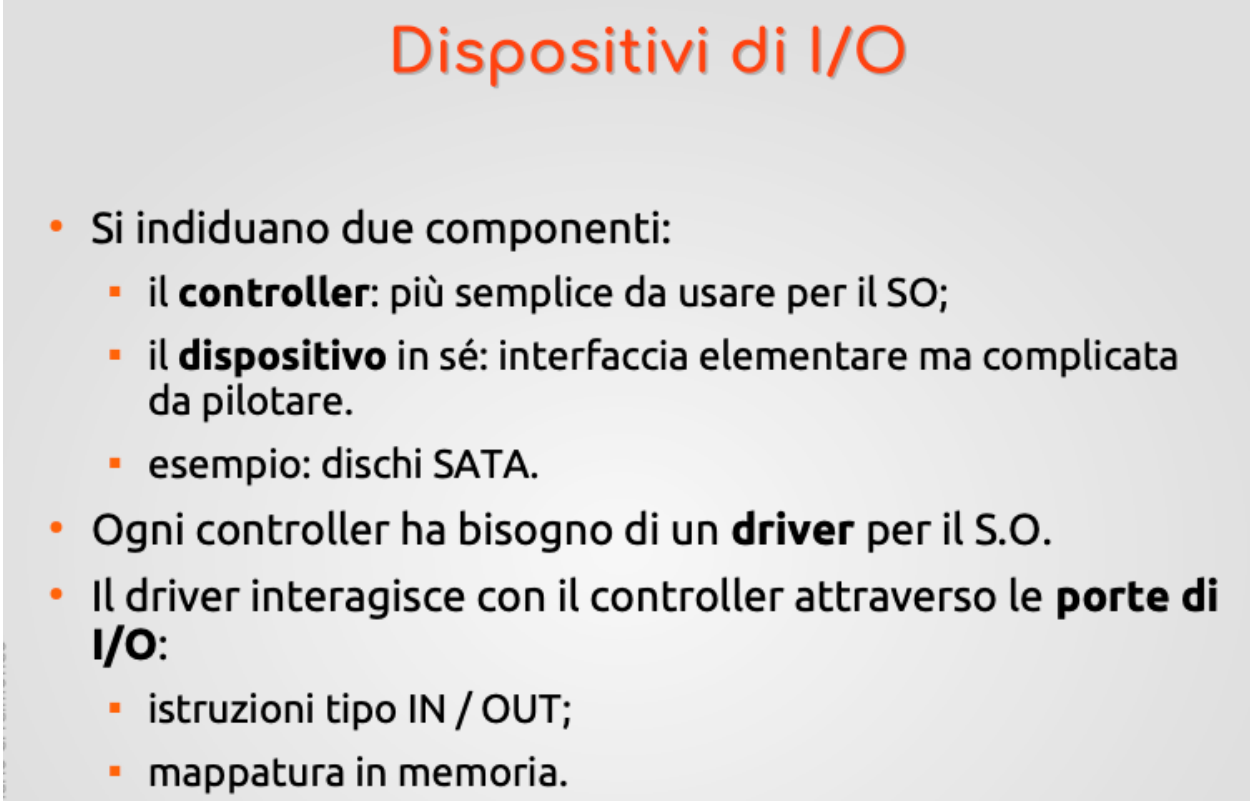
La L1 è la più vicina alla CPU, ha una latenza quasi nulla ma è molto costosa. **Nei sistemi moderni, le cache possono essere "dedicate" (ogni core ha la sua L2) o "condivise" (un'unica L3 per tutti i core). La cache condivisa facilita la**

comunicazione, ma può diventare un collo di bottiglia (bottleneck) se troppi core la richiedono contemporaneamente., lo scopo principale della cache è evitare in ogni modo che durante la fase di fetch si debba passare per la RAM (cosa che comunque ogni tanto succede) in modo da evitare bottleneck, utilizzando le linee di cache da 64 byte ciascuna (ovvero quando si prendono dei dati dalla RAM si prendono anche i successivi, perché è molto probabile che quei dati saranno utili a breve, quindi il dato prelevato va nei registri mentre quelli successivi aspettano nella cache: cache hit quando quei dati sono serviti veramente e si è risparmiato tempo, cache miss quando quei dati non sono serviti e bisogna andare di nuovo nella RAM) . Si ha un *Cache Hit* quando il dato è presente. Si ha un *Cache Miss* quando il dato non c'è e bisogna andare in RAM. **In caso di Miss, se la cache è piena, intervengono algoritmi specifici per decidere quale vecchio dato eliminare per fare spazio al nuovo. Se due core caricano la stessa linea di cache e uno dei due la modifica, il sistema deve sincronizzarli immediatamente affinché l'altro core non lavori su dati obsoleti.**

- RAM, memoria principale dell'architettura di Von Neumann, è veloce anche se molto meno di cache e registri ma in compenso è più capiente (ordine di GigaByte) qui risiedono i dati che servono ai processi per poter essere eseguiti.
- RAM, cache e registri sono memorie volativi ovvero quando il PC viene spento si
  svuota di tutti i lori dati
- Memoria di massa, questa memoria comprende vari dispositivi, dischi ottici, hard disk, SSD, pen drive ecc..., sono memorie lente rispetto a quelle volatili ma con
  diverse velocità in base al dispositivo in uso (SSD è più veloce di HDD), sono memorie usate per immagazzinare i dati, quindi sono non volativi ovvero quando il
  computer si spegne non perdono il loro contenuto.

Disco Magnetico (HDD): È un dispositivo elettromeccanico composto da dischi concentrici magnetizzabili che ruotano. La lettura avviene tramite testine (una per ogni faccia del disco). Le aree raggiungibili dalle testine senza spostare il braccio meccanico formano il "cilindro". Per leggere un dato, la testina deve

posizionarsi sulla traccia corretta e attendere la rotazione del disco, rendendolo molto più lento di un SSD.



Si individuano 2 componenti:

1. controller → interfaccia per il SO
2. dispositivo in se

Ad esempio la scheda video è un controller per il display. Un controller è quindi un mini
calcolatore dotato di una propria unità di elaborazione e una memoria (le scheda video
hanno un chip che esegue calcoli e una propria VRAM), usiamo l'esempio di un Hard Disk: Il controller dell' HDD è fisicamente collegato al disco, il controller si occupa di pilotare il motorino del disco e permette di inviare comandi (motore spento o acceso ad esempio), la CPU da se non riesce a comandare questo controllore, ha quindi bisogno di un driver proprietario, sviluppato dall'azienda dell'HDD che permetta di far dialogare bene SO, CPU e controllore con la periferica.

Quando un controller sta lavorando (ad esempio per cercare un file in un HDD), la CPU
deve aspettare che arrivi il risultato, ma anziché stare ferma svolge altri compiti, sarà
quindi un interrupt mandato dal controller che segnalerà alla CPU che il dispositivo ha
fornito la sua risposta

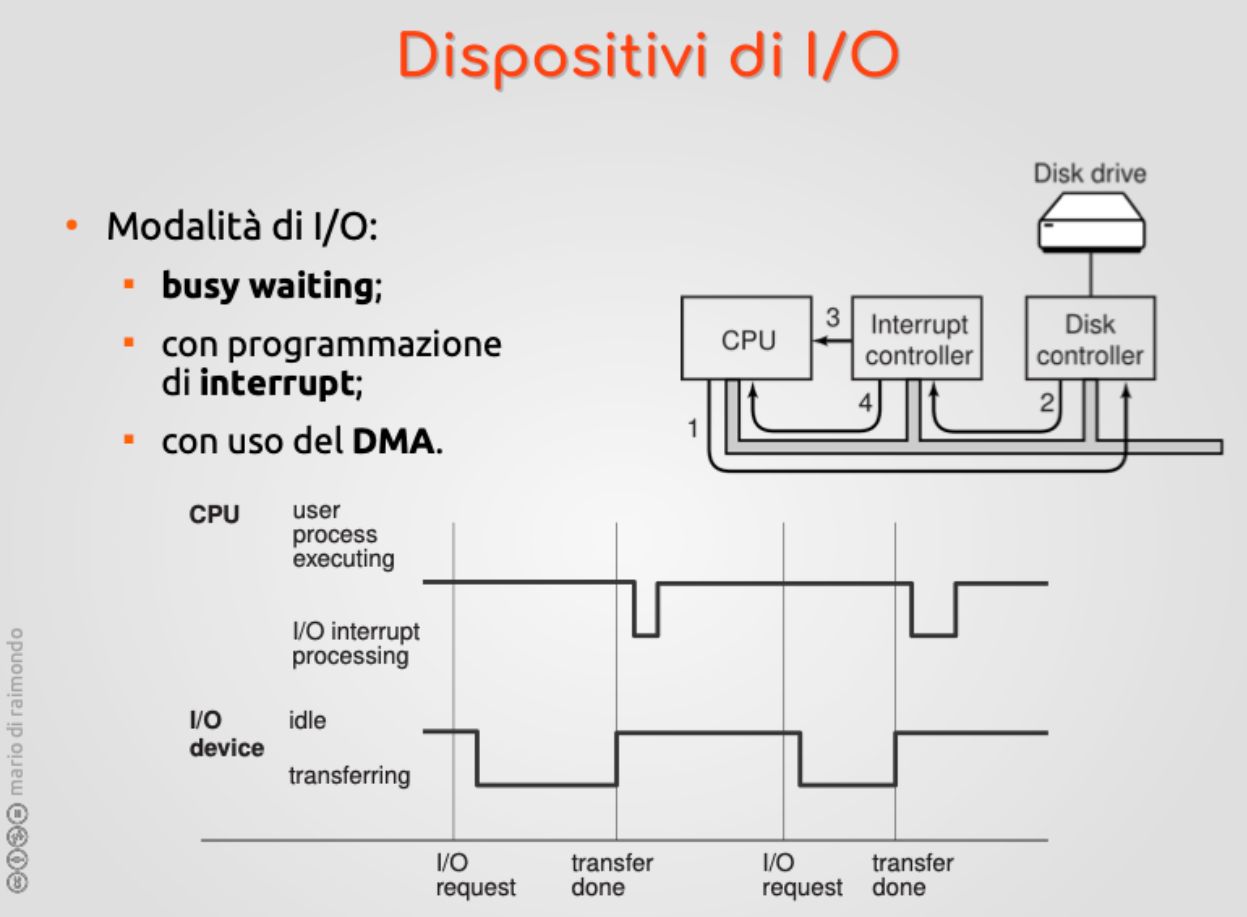
DRIVER

Il driver funge da interprete: conosce sia il linguaggio ad alto livello del Sistema Operativo, sia il linguaggio a basso livello del controller. Il produttore deve implementare un'interfaccia comune affinché il SO possa inviare richieste standard, che il driver traduce poi nei comandi primitivi e proprietari del controller.

MAPPATURA

Spiega come la CPU raggiunge le porte del controller:

> Per comunicare, si usa la **mappatura dell'I/O in memoria**: il sistema operativo istruisce la **MMU (Memory Management Unit)** affinché determini indirizzi di memoria corrispondano fisicamente alle porte di input/output del controller.
>
> - Creare questa mappatura è un'operazione **privilegiata** (Kernel Mode) che si fa una sola volta. Tuttavia, una volta che il 'ponte' è stabilito, leggere o scrivere su quegli indirizzi per scambiare dati non richiede più privilegi elevati. Questo permette un escamotage: far girare alcuni driver in **User Mode**, migliorando la stabilità del sistema.



Quando un processo (es. un editor di testo) richiede di leggere un file dal disco, si attiva una catena di eventi per gestire una risorsa che è infinitamente più lenta della CPU.

1. Lo stato di "Blocco" e la Multiprogrammazione

Poiché le operazioni di storage richiedono millisecondi (un'eternità per la CPU), il processo utente viene messo in **stato di bloccato (waiting)**.

- Non è accettabile che la CPU rimanga ferma ad aspettare il controller. Il SO "toglie" la CPU al processo bloccato e la concede a un altro processo pronto (multiprogrammazione).

2. Il problema del "Polling" (Interrogazione ciclica)

Il controller ha il suo "dialetto" per rispondere. Senza meccanismi avanzati, il SO dovrebbe usare la CPU per chiedere continuamente al controller: *"Hai finito? I dati sono pronti?"*. Questa tecnica si chiama **Polling**.

- **Il limite:** Il polling consuma cicli CPU inutilmente, producendo un alto **Overhead** (ovvero un carico di lavoro "burocratico" che non produce calcoli utili).

3. La soluzione: Interrupt e DMA

Per eliminare l'overhead e ottimizzare il sistema, si usano due strumenti:

- **Interrupt:** Invece di essere la CPU a chiedere, è il controller che "bussa" alla CPU tramite un segnale di interrupt quando l'operazione è conclusa.
- **DMA (Direct Memory Access):** Questo è il componente chiave. Senza DMA, la CPU dovrebbe leggere ogni singolo byte dal controller e scriverlo nella RAM uno alla volta (**Programmed I/O**). Questo saturerebbe la CPU solo per spostare dati.
    - **Come funziona il DMA:** Il SO istruisce il controller DMA dicendogli: *"Prendi X byte dal disco e copiali nell'indirizzo Y della RAM"*.
    - A questo punto, la CPU si disinteressa totalmente dell'operazione e torna a eseguire altri programmi.
    - Il controller DMA gestisce lo spostamento dei dati **direttamente in memoria** attraverso il bus di sistema.
    - Solo quando l'intero blocco di dati è stato trasferito, il DMA lancia un **interrupt** alla CPU per avvisarla che i dati sono finalmente pronti in RAM.

Cos'è l'Overhead di sistema?

In questo contesto, l'**overhead** è il tempo e la capacità di calcolo che il sistema "perde" per gestire l'impalcatura dell'I/O (gestione dei driver, salvataggio del contesto per gli interrupt, gestione del bus).

- **L'obiettivo del SO:** Ridurre l'overhead al minimo. Usare il DMA riduce drasticamente l'overhead perché la CPU interviene solo all'inizio e alla fine dell'operazione, lasciando il "lavoro sporco" dello spostamento dati all'hardware dedicato.

Lezione 12 Marzo