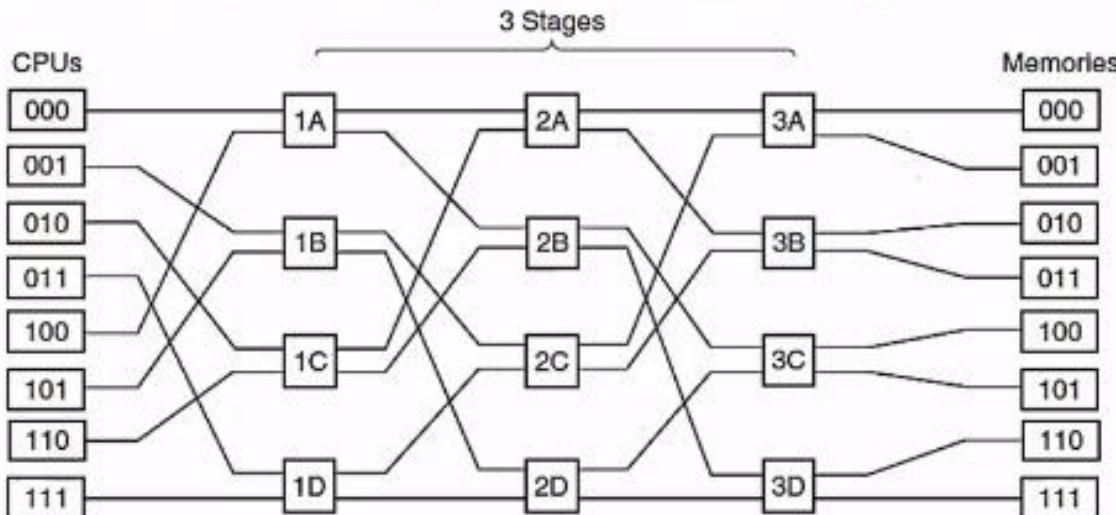


UMA con rete di commutazione a più stadi

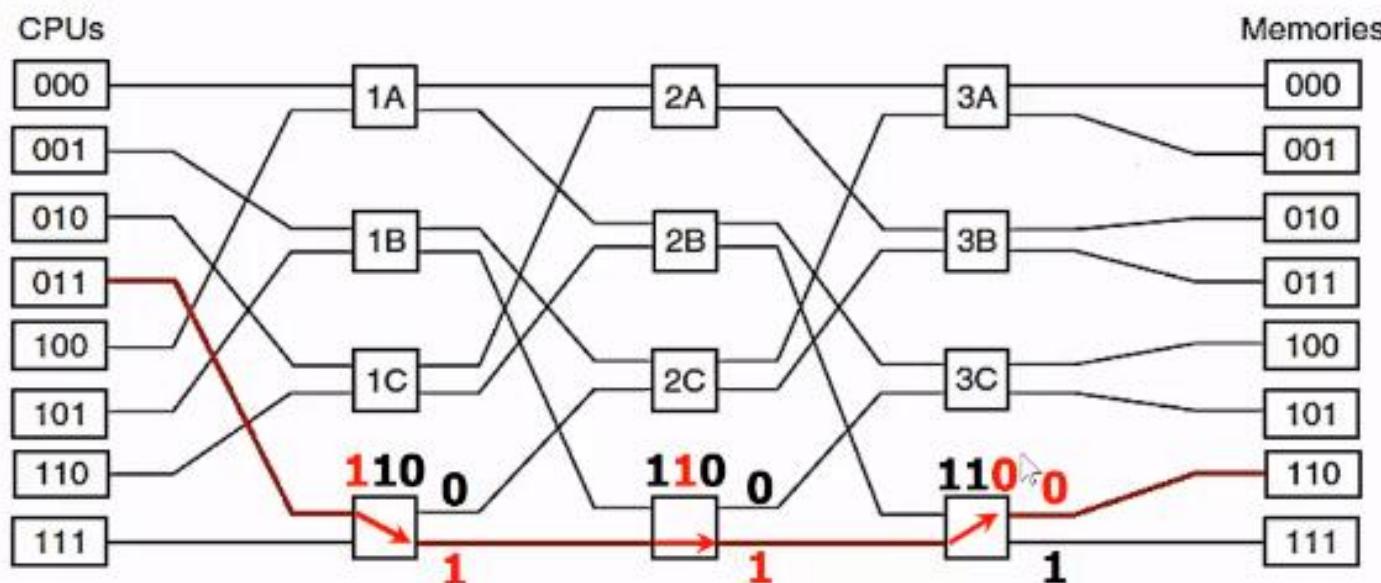
- Una rete economica e semplice è la **rete omega**.



- In questo esempio sono state connesse 8 CPU ad 8 memorie utilizzando 12 switch.
- Per n CPU e n memorie sono necessari $\log_2 n$ stadi (o stage), con $n/2$ switch per stadio, per un totale di **($n/2$) $\log_2 n$** switch, che è decisamente meglio di n^2 crosspoint, soprattutto per valori grandi di n .

UMA con rete di commutazione a più stadi

- Lo schema di connessione di una rete omega è detto **shuffle** (o miscuglio) **perfetto**.
- Per vedere come una rete omega lavora supponiamo che la CPU numero 3 (011_2) voglia leggere una parola nella memoria 6 (110_2).

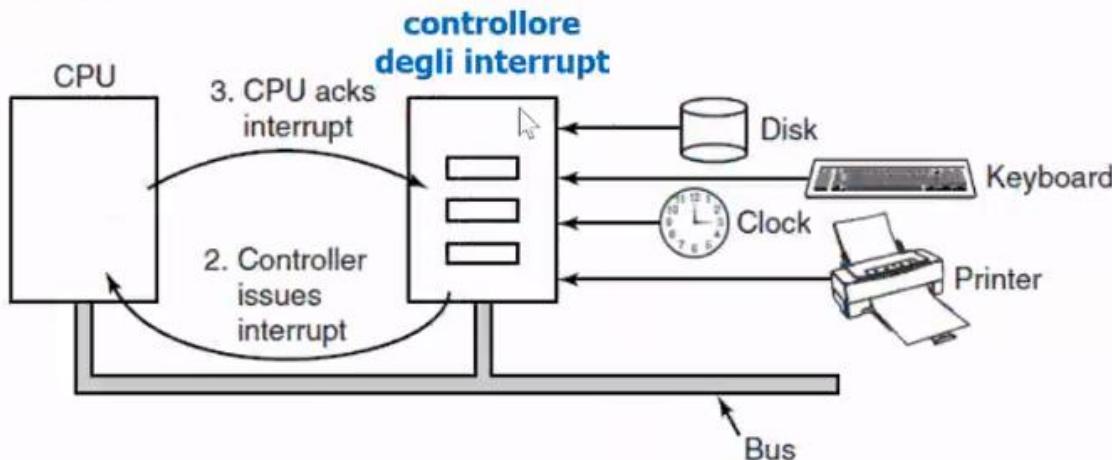


UMA con rete di commutazione a più stadi

- A differenza dell'interconnessione con crossbar switch, la rete omega è una rete **bloccante**: non tutti gli insiemi di richieste possono essere processati contemporaneamente.
- È auspicabile distribuire i riferimenti alla memoria in modo uniforme rispetto ai moduli. Un sistema di memorie in cui le parole consecutive sono in moduli diversi è detto **interleaved**.
- Le memorie interleaved massimizzano il parallelismo perché la maggior parte dei riferimenti è in indirizzi consecutivi.
- È sempre possibile progettare reti di switch non bloccanti.

Interrupt rivisitati

- Quando un dispositivo di I/O ha terminato il lavoro assegnato provoca un interrupt che invia al **controllore degli interrupt** (o **arbitro**) che decide poi cosa fare:
 - Se nessun altro interrupt è in sospeso, viene elaborata immediatamente l'interruzione.
 - Se è già stato riconosciuto un altro interrupt oppure contestualmente un altro dispositivo ha inviato un altro segnale di richiesta, si decide in base alla priorità assegnata ai dispositivi.



Interrupt rivisitati

- Il controller segnala alla CPU dell'interruzione, attende il riconoscimento e pone un **numero** sulle linee di indirizzo per specificare quale dispositivo ha inviato l'interrupt.
- Il numero è usato come indice in una tabella (**vettore di interruzione**) che restituisce il nuovo valore per il PC: l'inizio della Routine di Servizio dell'Interrupt (**ISR**).
- L'ISR per comunicare al controller degli interrupt che può accettare altre interruzioni, scrive un valore in una delle porte di I/O del controller stesso.
- Prima di avviare la ISR l'hardware deve salvare lo stato della CPU (PC, registri, stack, ...), dove vengono salvate queste informazioni?



Interrupt rivisitati

1) Nei **registri interni**.

Il controller degli interrupt non può essere avvisato finché tutte le informazioni rilevanti non sono state lette (una seconda interruzione potrebbe sovrascrivere i registri interni).

Questo approccio porta a tempi lunghi morti quando gli interrupt sono disabilitati e una possibile perdita di interrupt e di dati.

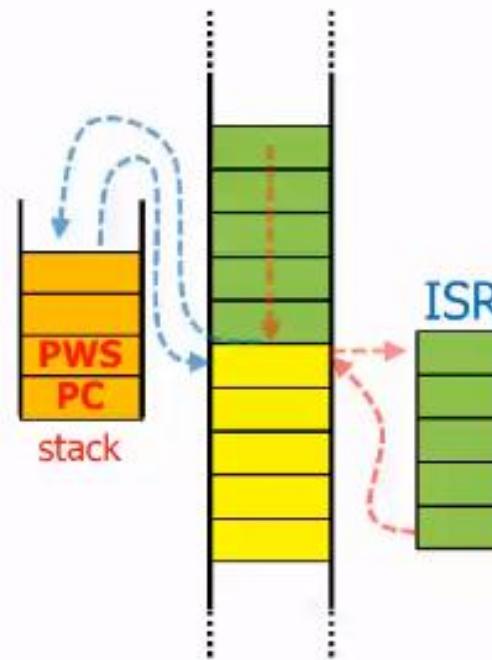
2) Nello **stack**.

Quale stack usare? Quello del processo utente o uno nel kernel?

L'uso dello stack nel kernel è migliore di quello dei processi utente dal punto di vista dell'affidabilità dei riferimenti alle pagine (SP). Tuttavia, il passaggio alla modalità kernel può richiedere tempo di CPU a causa del cambio di contesto nella MMU.

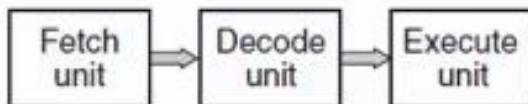
Interrupt precisi

- La maggior parte delle CPU moderne sono pipeline e superscalari (unità funzionali parallele).
- In passato era tutto più semplice: al termine dell'esecuzione di ogni istruzione l'hardware controllava se ci fosse un interrupt in sospeso.
- Il Program Counter (PC) e la parola di stato del programma (PSW) venivano inseriti nello stack ed eseguita la ISR.
- Al termine della ISR venivano ripristinati dallo stack PC e PSW.
- Tutte le istruzioni eseguite prima dell'interrupt erano state eseguite completamente, ciò non è vero nelle attuali architetture.

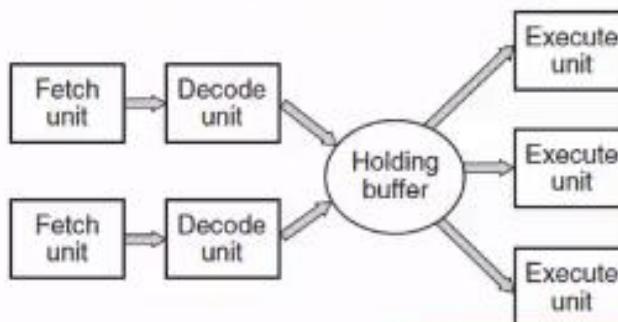


Interrupt precisi e imprecisi

- Cosa succede se si verifica un interrupt in una CPU che ha una pipeline piena?



- Le istruzioni sono in varie fasi di esecuzione.
- Quando si verifica un interrupt, il valore del PC non determina il confine netto tra istruzioni eseguite e quelle non eseguite.
- Su una macchina superscalare la situazione è ben più complessa, le istruzioni possono essere suddivisi in micro-operazioni eseguite in un ordine che dipende dalla disponibilità delle unità funzionali e dei registri.

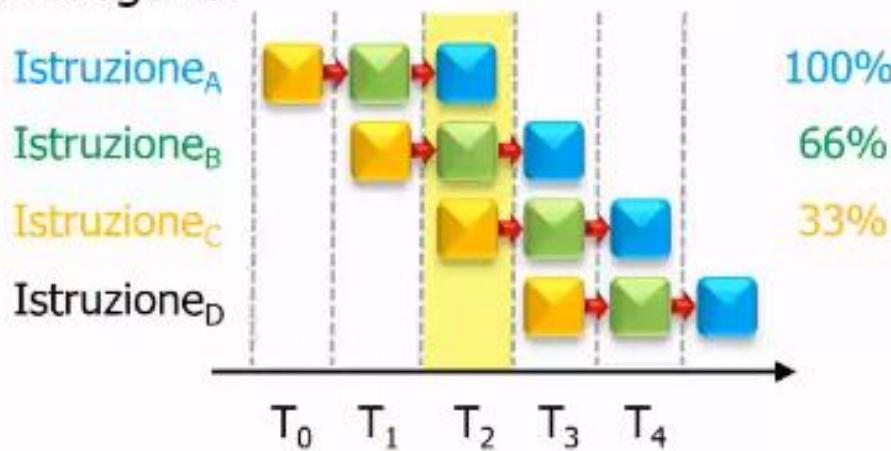


Interrupt imprecisi

- Cosa succede se si verifica un interrupt in una CPU che ha una la pipeline piena?



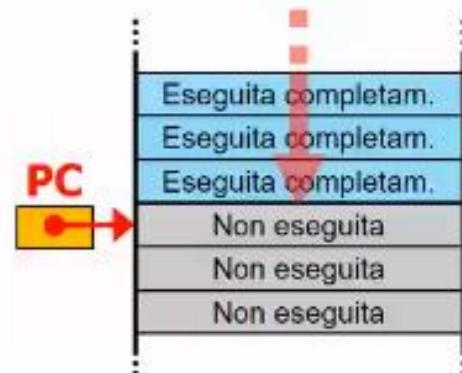
- Le istruzioni sono in varie fasi di esecuzione.
- Quando si verifica un interrupt, il valore del PC non determina il confine netto tra istruzioni eseguite e quelle non eseguite.



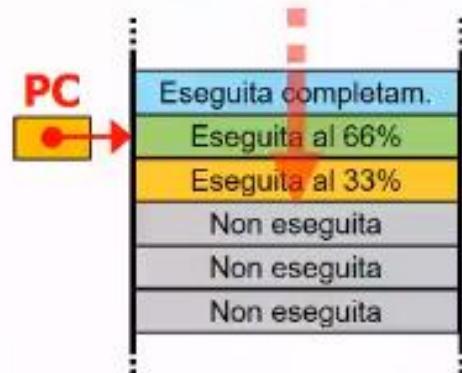
Interrupt precisi e imprecisi

- Un interrupt che lascia la macchina in uno stato ben definito è chiamato **interrupt preciso** e ha quattro proprietà:
 - 1) Il PC viene salvato in un posto conosciuto.
 - 2) Tutte le istruzioni prima di quello a cui punta il PC sono completamente eseguite.
 - 3) Nessuna istruzione oltre all'istruzione puntata dal PC è stata eseguita.
 - 4) Lo stato di esecuzione dell'istruzione puntata dal PC è conosciuto.
- Un interrupt che non soddisfa questi requisiti è chiamato un **interrupt impreciso**.

interrupt preciso



interrupt impreciso



Interrupt precisi e imprecisi

- Macchine con interruzioni imprecise di solito delegano il sistema operativo per capire come gestire la situazione e riversano sullo stack grandi quantità di dati sullo stato interno.
- Il codice necessario riavviare la macchina è estremamente complicato.
- Gli interrupt e il ripristino dello stato da un interrupt sono operazioni lente.
- Ciò porta ad una situazione paradossale che le veloci CPU superscalari talvolta sono inadatte per gestire sistemi real-time a causa della lentezza degli interrupt.



Obiettivi del software di I/O

- **Indipendenza dal dispositivo:** i programmi che gestiscono l'I/O devono poterlo fare indipendentemente dal tipo di dispositivo.
- **Denominazione uniforme:** l'identificatore di un file o di un dispositivo non deve dipendere dal tipo di dispositivo.
- **Gestione degli errori:** gli errori andrebbero gestiti il più possibile a livello hardware (controller, driver,...).
 - Molti errori (come quelli di lettura del disco) sono transitori e scompaiono se si ripete l'operazione.
- **Trasferimenti sincroni/asincroni:** i trasferimenti possono essere bloccanti (sincroni) o gestiti con interrupt (non bloccanti).
 - A causa della differente velocità, i dispositivi di I/O sono tipicamente asincroni. Tuttavia è molto più facile scrivere programmi utenti con primitive bloccanti (es. `read()`), quindi sta al SO dare al programmatore l'illusione di utilizzare chiamate sincrone quando in realtà i dispositivi sono trattati con primitive asincrone.

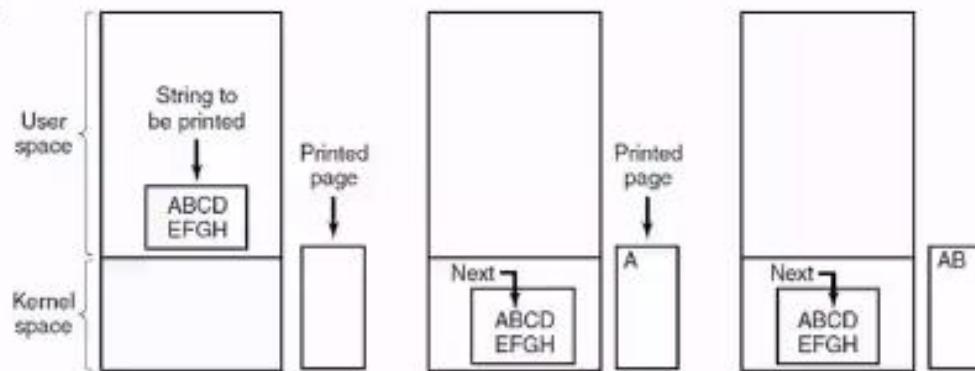
Modalità di gestione dell'I/O

- I dispositivi di I/O possono essere gestite con tre diverse tecniche:
 - 1) **I/O programmato.**
 - 2) **I/O guidato dagli interrupt (interrupt-driven).**
 - 3) **I/O con Direct Memory Access.**
- La prima tecnica è la più semplice perché è la CPU che svolge tutto il lavoro.
 - Si consideri per esempio un processo utente che vuole stampare una certa stringa sulla stampante.



I/O programmato

- Il processo tenta di aprire una connessione con la stampante con una system call, se la stampante è occupata o fuori linea:
 - restituisce un codice di errore o si blocca fino a quando non è disponibile.
- Quando la stampante torna disponibile informa il SO.
- Il SO copia la stringa nello spazio kernel (array p) e controlla se la stampante è ancora disponibile, altrimenti aspetta.
- Se sì, il SO invia un carattere alla volta al registro dati della stampante.



I/O programmato

- La CPU esegue il polling della stampante per controllare se è pronta ad accettare un altro carattere (**polling** o **busy waiting**).
- Il codice di esempio potrebbe essere:

```
copy_from_user(buffer, p, count);           /* p è copiato nel spazio kernel */
for (i = 0; i < count; i++) {                /* ciclo su ogni carattere */
    while (*printer_status_reg!=READY);      /* ciclo di polling sulla stampante */
    *printer_data_register = p[i];           /* invia un carattere alla stampante*/
}
return_to_user();                          /* torna allo spazio utente */
```

- L'I/O programmato è semplice ma ha lo svantaggio di utilizzare il tempo della CPU finché tutte le operazioni di I/O non sono finite.
- Generalmente, il **busy waiting** è una prassi che andrebbe limitata ai soli casi di I/O veloce.
- Esistono metodi migliori...

Conclusioni

- Completata la realizzazione del file system, analizzate le principali tecniche di gestione.
- Introdotto la gestione dei dispositivi di I/O dal parte del sistema operativo.
- Analizzato i principi hardware dei dispositivi di I/O.
- Introdotto i principi software dei dispositivi di I/O e descritto un esempio di I/O programmato.



I/O guidato dagli interrupt

- Supponiamo di utilizzare una stampante senza buffer: stampa un carattere alla volta.
- Se la stampante è in grado di stampare 100 caratteri/sec, ognuno impiega 10 msec per la stampa. Ciò significa che la CPU potrebbe attendere 10 msec per ciascun carattere.
- Il modo per consentire alla CPU di fare qualcosa' altro è di utilizzare gli **interrupt**.
- Quando è attivata la system call per la stampa della stringa, il buffer è copiato nello spazio kernel e il primo carattere inviato alla stampante.
- La CPU richiama lo scheduler così viene eseguito un altro processo.
- Il processo che ha chiesto la stampa della stringa rimane bloccato finché non è stata stampata l'intera stringa.

I/O guidato dagli interrupt

Codice eseguito quando è attivata la system call:

```
copy_from_user(buffer, p, count);           /* p è copiato nel spazio kernel */
enable_interrupts();                      /* abilita gli interrupts */
while (*printer_status_reg!=READY);        /* ciclo di polling sulla stampante */
*printer_data_register = p[0];              /* stampa l'ultimo carattere */
scheduler();                                /* invoca lo scheduler */
```

Interrupt Service Routine (ISR) per la stampante:

```
if (count== 0) {
    unblock_user();                         /* sono stati stampati tutti i caratteri */
} else {
    *printer_data_register = p[i];          /* stampa il carattere i-esimo */
    count = count - 1;                     /* decrementa il contatore dei caratteri */
    i = i + 1;                            /* incrementa l'indice i */
}
acknowledge_interrupt();                  /* l'interrupt è riconosciuto */
return_from_interrupt();                 /* esce dall'interrupt e ripristina lo stato */
```

I/O con l'uso del DMA

- L'I/O guidato dagli interrupt genera un interrupt per ogni carattere.
- Gli interrupt richiedono tempo, così si spreca del tempo di CPU.
- Una soluzione alternativa è di utilizzare il controller DMA che invia i caratteri alla stampante una alla volta, senza l'intervento della CPU.
 - Ottima soluzione se si devono stampare molti caratteri e gli interrupt sono lenti.
- Questa strategia richiede un hardware speciale (il controller DMA), ma consente alla CPU di svolgere altre attività durante l'I/O.
- Poiché il controllore DMA è più lento della CPU potrebbero esserci delle situazioni (rare) in cui conviene utilizzare le altre due tecniche (I/O interrupt-driven o I/O programmato).

I/O con l'uso del DMA

Codice eseguito quando è attivata la system call:

```
copy_from_user ( buffer, p, count);           /* p è copiato nel spazio kernel */
setup_DMA_controller(buffer, p, count);        /* imposta il controller DMA */
scheduler( );                                /* invoca lo scheduler */
```

Interrupt Service Routine (ISR) per la stampante:

```
acknowledge_interrupt();          /* l'interrupt è riconosciuto */
unblock_user( );                /* tutti i caratteri sono stati stampati */
return_from_interrupt();         /* esce dall'interrupt e ripristina lo stato */
```

LIVELLI SOFTWARE PER L'I/O

- Il software di I/O è in genere organizzato in quattro livelli:



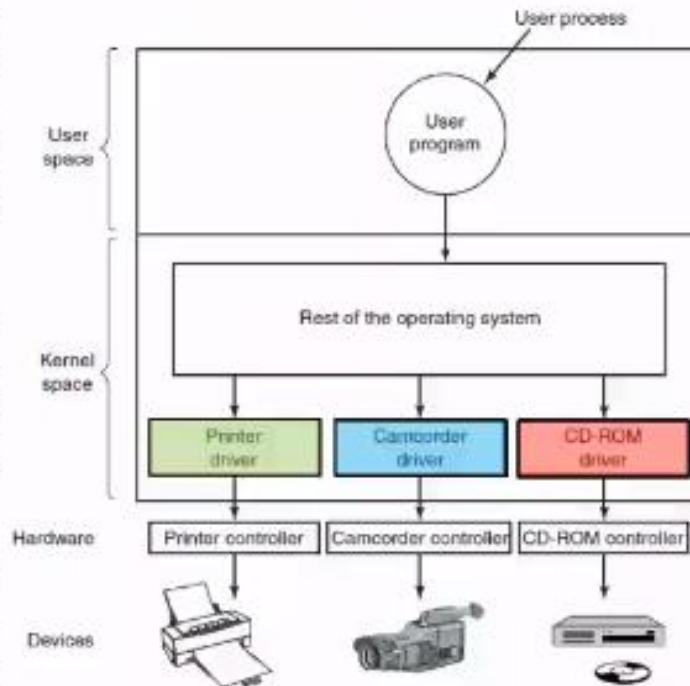
- Ogni strato ha una funzione ben definita e definisce dei servizi agli strati adiacenti attraverso la sua interfaccia.
- La funzionalità e le interfacce sono diverse da sistema a sistema, tuttavia verranno affrontate le caratteristiche comuni a più sistemi.

Driver dei dispositivi

- Per controllare un dispositivo di I/O è necessario un codice (driver di periferica) specifico del dispositivo, scritto dal produttore del dispositivo.
- I produttori di dispositivi generalmente forniscono driver per la gran parte dei sistemi operativi. 
- Per accedere ai registri del controller, il driver di periferica deve essere parte del kernel nel sistema operativo.
- È possibile costruire driver eseguiti nello spazio utente, utilizzando le chiamate di sistema.
 - Isolamento del kernel.
 - Il sistema diventa altamente affidabile.

Drivers dei dispositivi

- I progettisti dei sistemi operativi sanno che pezzi di codice (driver) scritto da terzi saranno installati all'interno della loro architettura.
- Esiste modello ben definito di ciò che un driver fa e come deve interagire con il resto del sistema.
- I driver di periferica sono normalmente posizionati al di sotto del resto del sistema operativo.

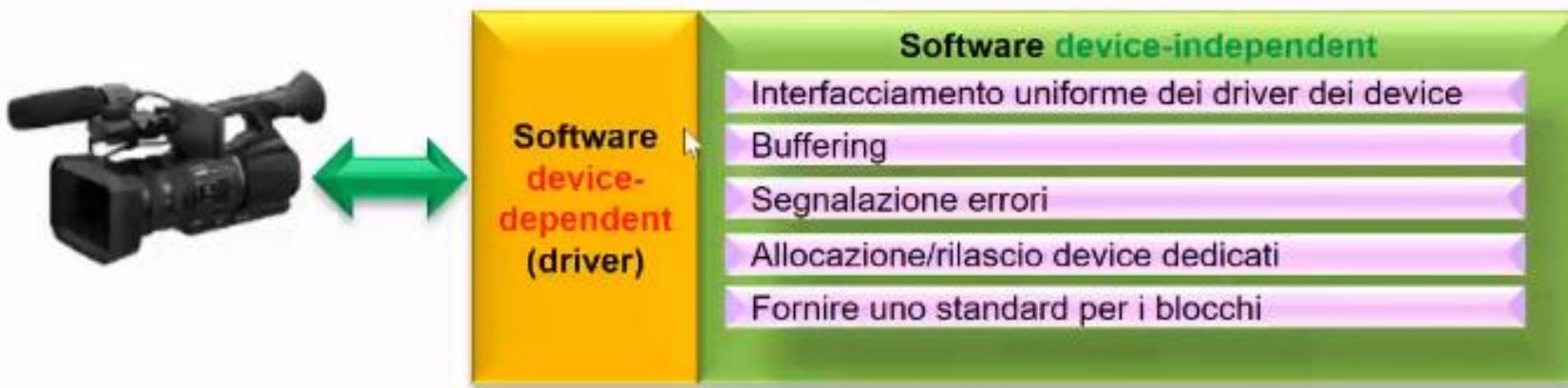


Drivers dei dispositivi

- I sistemi operativi di solito classificano driver come:
 - **dispositivi a blocchi** (dischi) che indirizzano uno o più blocchi di dati in modo indipendente.
 - **dispositivi a caratteri** (tastiere, stampanti, ...) che gestiscono i flussi di caratteri.
- Le caratteristiche più importanti per i driver sono la capacità di:
 - essere rientranti (o **reentrant**): durante l'esecuzione devono aspettarsi di essere chiamati più volte ancor prima che sia completata la precedente chiamata.
 - garantire sistemi **pluggable «a caldo»**, i dispositivi possono essere aggiunti o rimossi mentre il computer è in funzione.
- Anche se non è concesso ai driver di effettuare chiamate di sistema, essi devono interagire con il kernel per allocare/deallocare pagine di memoria fisica, per gestire la MMU, i timer, il controller DMA, i controller degli interrupt,...

Software indipendente dal device

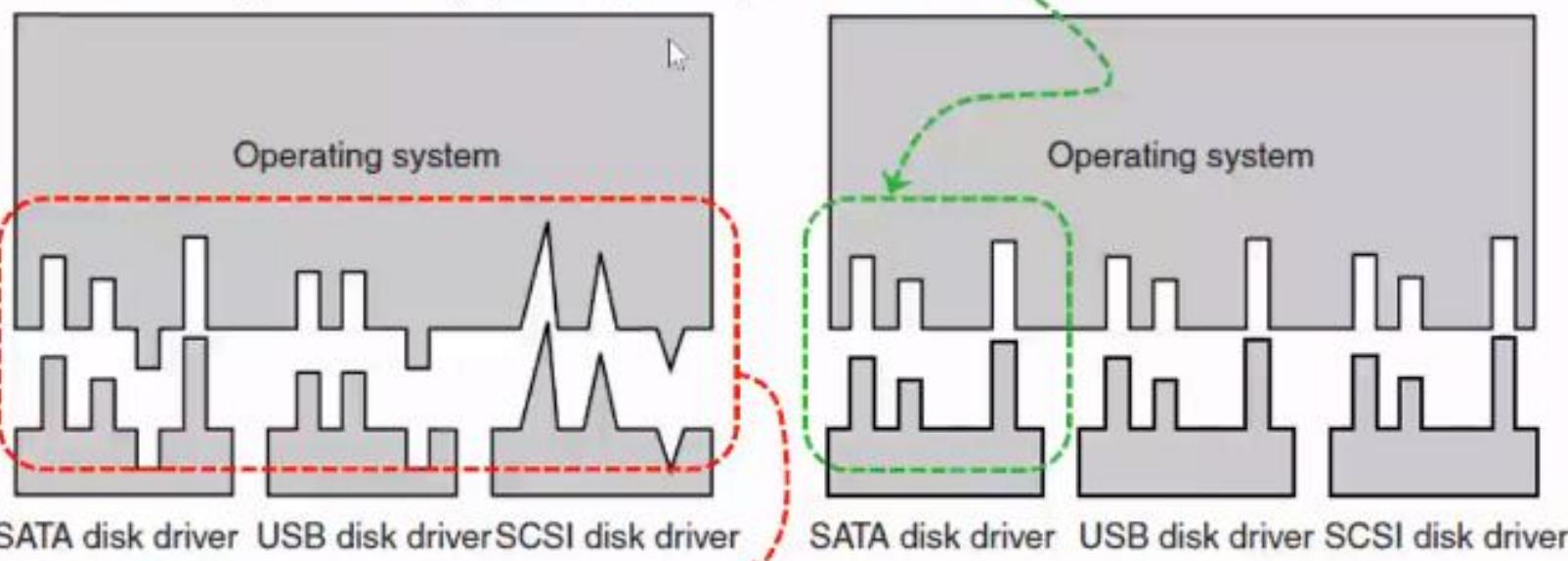
- Il software per la gestione dei dispositivi di I/O si compone di una parte specifica e un'altra indipendente dal dispositivo.



- Il confine esatto tra i driver e il software indipendente dal dispositivo dipendono dal sistema e dal dispositivo.

Interfacciamento uniforme dei driver

- Una questione importante in un sistema operativo è come rendere tutti i dispositivi e i driver più o meno simili.
- La modalità di interfacciamento tra driver e sistema operativo deve la stessa (standard) per ogni dispositivo:



- Senza standard l'interfacciamento di un nuovo driver richiede uno sforzo di programmazione enorme.

Interfacciamento uniforme dei driver

- Il software indipendente dal dispositivo si prende cura di mappare i nomi simbolici dei dispositivi nel driver adatto.
- In UNIX tutti i dispositivi hanno:
 - Il numero del dispositivo primario (**major device number**), utilizzato per individuare il driver appropriato.
 - Il numero del dispositivo secondario (**minor device number**), utilizzato come parametro per il driver per specificare l'unità.
- In UNIX il nome del dispositivo:

/dev/disk0

Specifica l'**i-node** di un file speciale che contiene i riferimenti al driver del **disco (major device number)** e all'**unità 0 (minor device number)**.

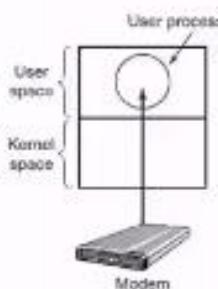
Buffering

- Il **buffering** è un problema sia per i dispositivi a blocchi sia per quelli a caratteri.

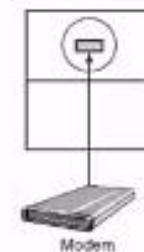
- Consideriamo un processo che vuole leggere dati da un modem:

- Il processo utente esegue una **read()** e si blocca in attesa di un carattere. Ogni carattere genera un interrupt.
- L'ISR dà il carattere al processo utente e si sblocca.

Questa soluzione è inefficiente perché il processo viene eseguito molte volte per brevi periodi di tempo.

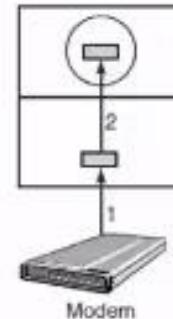


- L'aggiunta di un buffer di «n» caratteri nello spazio utente permette alla ISR di scrivere più caratteri nel buffer finché non si riempie e sveglia il processo utente.
 - Che cosa succede se il buffer è paginato su disco quando arriva un carattere?

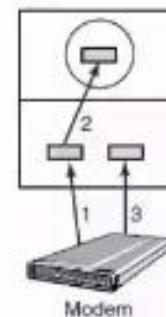


Buffering

- 3) L'ISR mette i caratteri in un buffer nel kernel, quando è pieno, la pagina che contiene il buffer nello spazio utente viene portata in memoria (se era su disco) e poi è copiato in un'unica operazione dalla pagina kernel a quella utente.
- Cosa succede ai caratteri che arrivano quando il buffer è pieno?

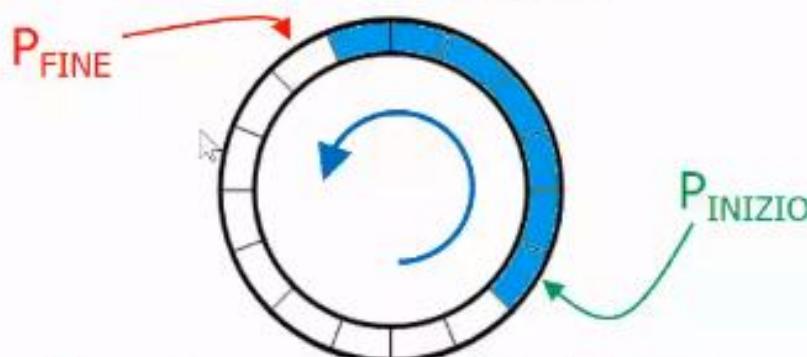


- 4) Si usano due buffer (**schema a doppio buffer**) nel kernel che si scambiano di ruolo: uno per leggere dal modem e l'altro per copiare nello spazio utente. I caratteri che arrivano mentre la pagina non è pronta o il buffer è pieno vengono memorizzati nell'altro. Il primo viene copiato e, poi, svuotato e cambia ruolo: ora può memorizzare caratteri quando il secondo sarà pieno.



Buffering

- 5) Un'altra soluzione adotta un **buffer circolare**: un puntatore indica la prossima locazione dove inserire nuovi dati (P_{FINE}), un altro la prima locazione riempita nel buffer (P_{INIZIO}).



P_{FINE} avanza quando arrivano nuovi dati dal modem

P_{INIZIO} il SO elimina ed elabora i dati.

Quando il buffer è pieno $P_{INIZIO} = P_{FINE}$

- Il buffering è una tecnica largamente utilizzata, ma occorre far attenzione all'eccesso di copia, da uno spazio all'altro, che rallenta le prestazioni.

Segnalazione degli errori

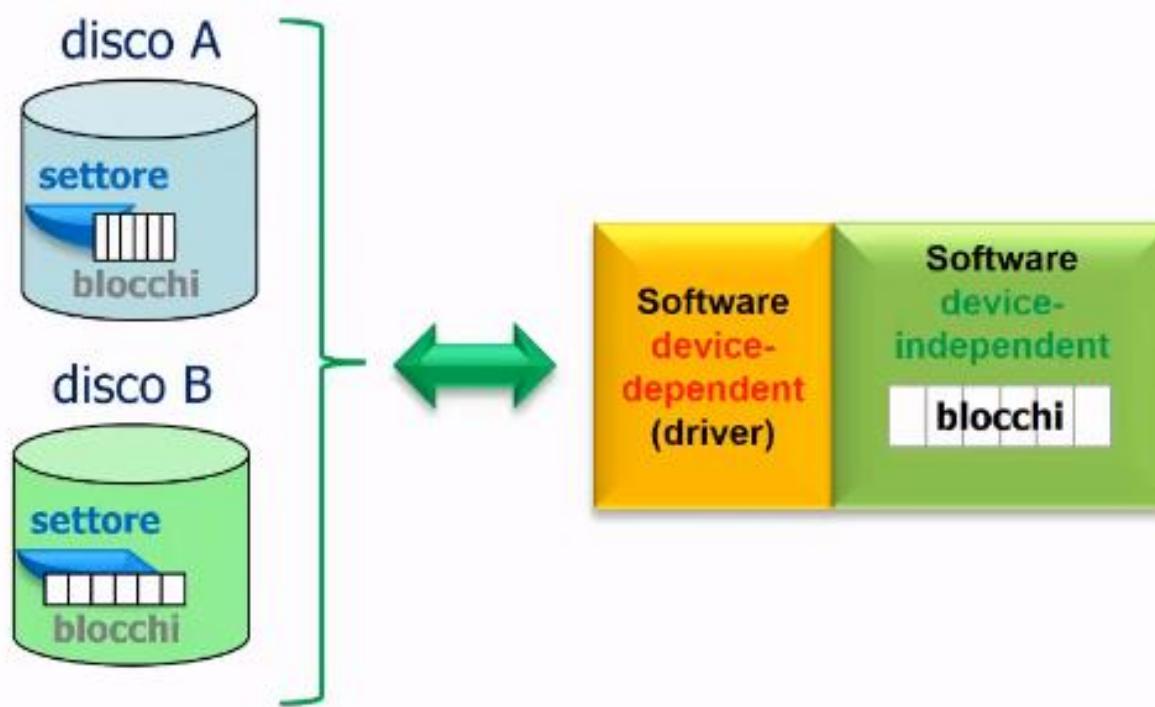
- Quando si tratta con i dispositivi di I/O gli errori sono più frequenti di altri ambiti.
- Le classi più comuni di errori sono:
 - Errori di programmazione: un processo chiede qualcosa di impossibile (es. scrittura su un dispositivo di ingresso).
 - Errori fisici del dispositivo (settore rovinato).
- Quando si verifica un errore, un software «chiamato» ha varie possibilità:
 - 1) delegare il chiamante (nel caso del driver: il software indipendente dal dispositivo) al trattamento dell'errore.
 - 2) tentare di risolvere l'errore localmente:
 - reiterando più volte l'operazione.
 - ignorando l'errore.
 - terminando il processo chiamante.

Allocazione/rilascio dei dispositivi dedicati

- Alcuni dispositivi (es. masterizzatori) possono essere utilizzati da un solo processo per volta, è il SO operativo che gestisce le richieste per l'uso di tali dispositivi.
- Due possibili approcci:
 - 1) Il processo che vuole utilizzare il dispositivo tenta una **open()** su un file speciale associato al dispositivo stesso. Se è libero riesce nell'operazione e blocca l'utilizzo esclusivo del device.
 - 2) Si utilizzano i classici meccanismi di sincronizzazione tra processi: il processo che chiede un device occupato va in sleep finché il device non è libero.

Dimensione dei blocchi device-independent

- Dischi differenti possono avere settori di diverse dimensioni, sta al software indipendente dal dispositivo nascondere questo aspetto e fornire una dimensione di blocco uniforme (astrazione).



Software per l'I/O nello spazio utente

- La maggior parte del software per la gestione dell'I/O è all'interno del sistema operativo, tuttavia una piccola parte consiste di **librerie** che verranno poi collegate ai programmi utente.
- Quando un programma C contiene la chiamata:
`count = write(fd, buffer, nbytes);`
- La procedura **write()** che appartiene ad una libreria nello spazio utente sarà collegata con il programma e contenuta nel programma binario presente in memoria in fase di esecuzione.
- Tutte procedure e librerie sono chiaramente parte del sistema di gestione dell'I/O.

Software per l'I/O nello spazio utente

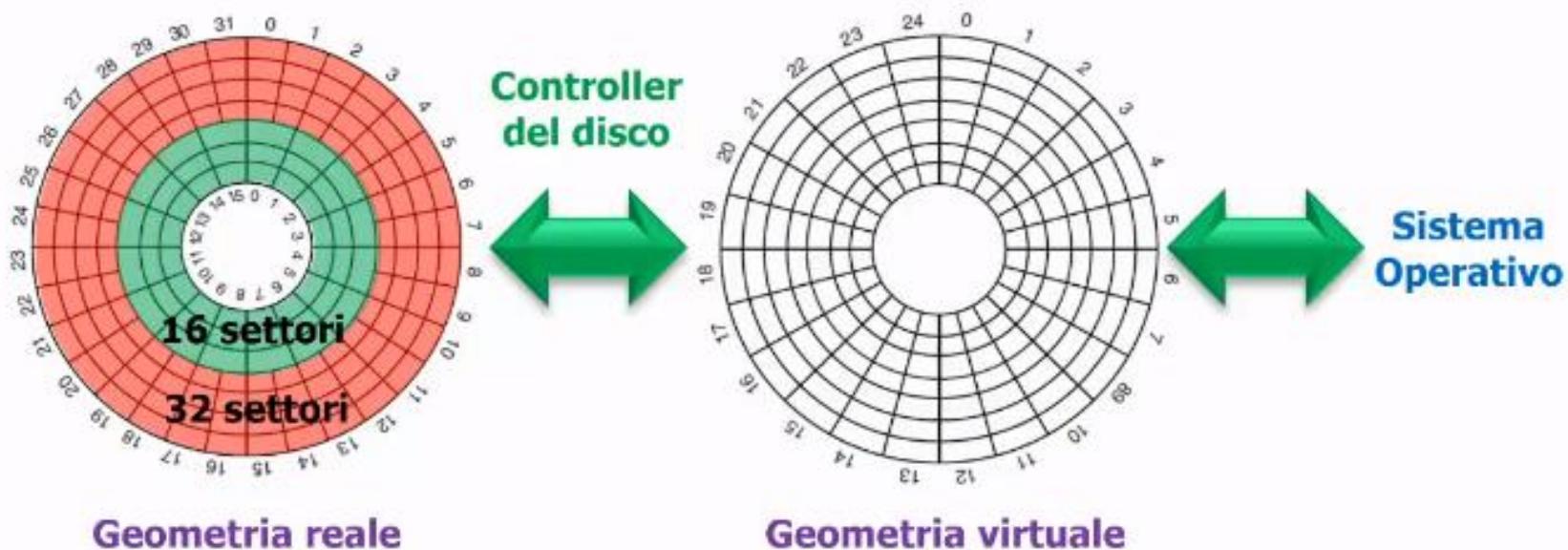
- Non tutto il software a livello utente per la gestione dell'I/O è costituito dalle procedure nelle librerie, esiste anche il **sistema di spooling**.
- Lo **spooling** è il modo per interagire con i dispositivi di I/O dedicati in un sistema multiprogrammato.
- Nel caso della stampante esiste un processo **demone** e una **directory speciale** (detta **di spooling**).
 - Per stampare un file, un processo scrive il file da stampare nella directory di spooling.
 - Successivamente il demone legge i file memorizzati nella directory e li stampa, uno alla volta.
- Questa tecnica non è utilizzata solo per le stampanti, ma anche con altri dispositivi di I/O. Ad esempio, il trasferimento di un file nella rete.

Hardware dei dischi

- Tra i principali dispositivi di I/O troviamo i dischi:
 - 1) **Magnetici** (fissi).
 - Le operazioni di lettura o scrittura hanno la stessa velocità.
 - 1) **RAID**.
 - Utilizzati per configurazioni di alta affidabilità.
 - 1) **A stato solido (SSD)**.
 - Prestazioni eccellenti (assenza di seek time e velocità di trasferimento 3 volte maggiori rispetto ad un tradizionale disco magnetico) ed assenza di componenti meccaniche.
 - 1) **Ottici** (CD-ROM, CD-Recordable, CD riscrivibile, DVD, Blu-Ray)
 - Normalmente utilizzati per la distribuzione di programmi.

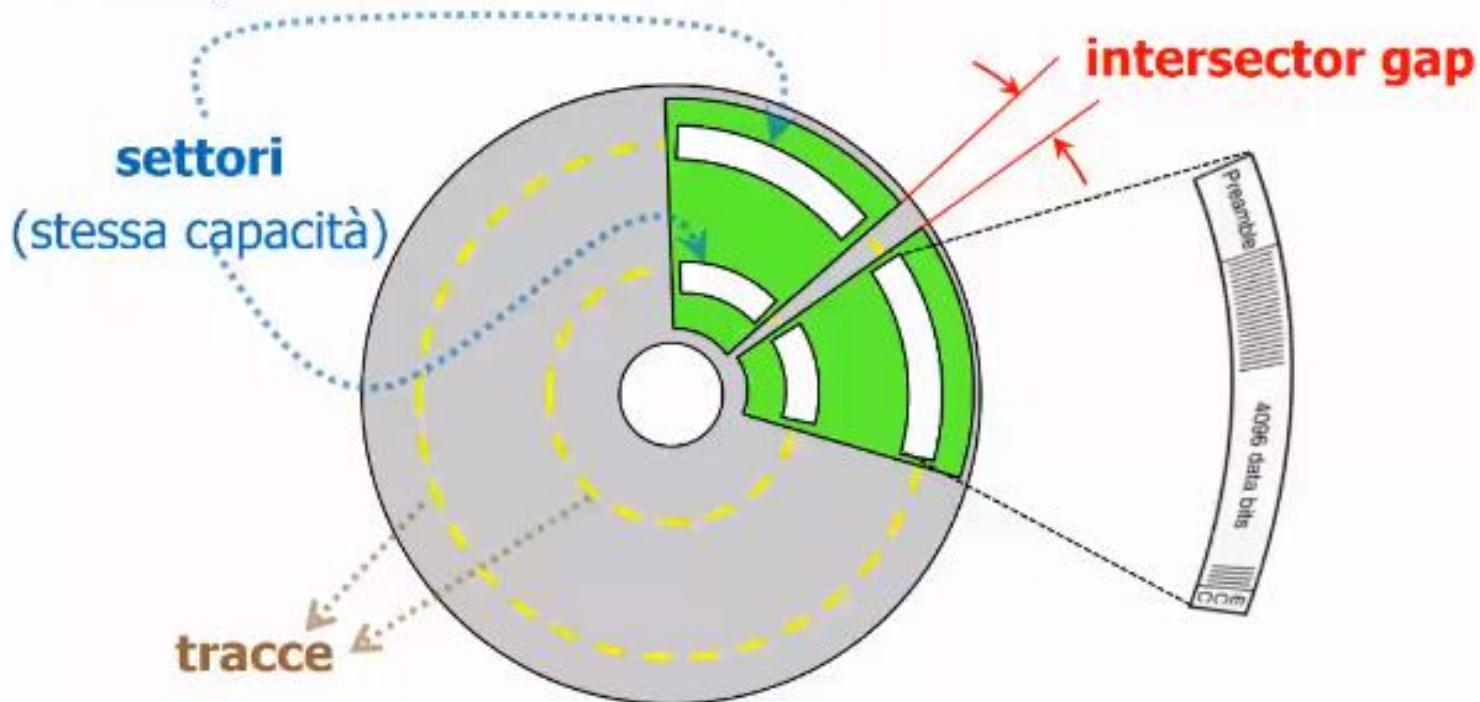
Dischi Magnetici

- La superficie dei dischi è divisa in zone:
 - Quelle esterne hanno più settori rispetto a quelle interne.
- La geometria mostrata al SO è diversa da quella fisica, sarà il controller a rimappare le richieste del SO verso il numero reale di settori.



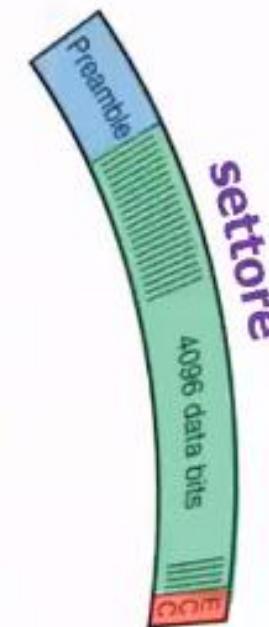
Formattazione dei dischi

- Prima di poter utilizzare un disco occorre definirgli un formato: attraverso la **formattazione a basso livello**.
 - una serie di tracce concentriche, ciascuna contenente un numero di settori, con brevi spazi tra i settori.



Formattazione dei dischi

- Ogni **settore** contiene:
 - Un **preambolo**, ovvero una sequenza di bit che consente riconoscere l'inizio del settore che può contenere anche il numero dei cilindri/settori e altre informazioni.
 - **Dati** (payload), la cui dimensione viene determinata dal programma di formattazione di basso livello (tipicamente i settori sono di 512 byte).
 - Codice di Correzione dell'Errore (**ECC**), un codice di Hamming oppure un altro codice che permette di correggere errori multipli (codice **Reed-Solomon**).
- La formattazione del disco ne riduce la dimensione di circa il **15-20%**.



Formattazione dei dischi

- Durante la formattazione a basso livello la posizione del settore 0 su ogni traccia è spostato di un certo offset rispetto alla traccia precedente.
- Questo offset, chiamato **pendenza del cilindro** (o **cylinder skew**), permette di migliorare le prestazioni riducendo gli spostamenti della testina nelle operazioni che coinvolgono più tracce.
- La pendenza del cilindro dipende dalla geometria del disco:
 - Un disco con RPM da 10k compie un giro in 6 ms.
 - Se la traccia ha 300 settori, la testina incontra il settore ogni 20 μ s.
 - Se il tempo di seek è di 800 μ s, significa che in una seek transitano 40 settori (cioè il miglior valore per la pendenza).

Confusione sulle unità di misura

- Molti produttori dichiarano la capacità del disco non formattata con lo scopo di farli sembrare più grandi della capacità che in realtà riescono a contenere.
 - Un disco da **200 GB** (200×10^9) dopo la formattazione avrà uno spazio utile per i dati di circa **170 GB**.
 - Tuttavia, il sistema operativo segnalerà **158 GB** perché il software utilizza la base due per rappresentare le quantità: **1 GB** di memoria è pari a **2^{30}** (1.073.741.824) byte e non **10^9** (1.000.000.000).
- Si osservi come invece nelle comunicazioni dati si utilizzi la base dieci per cui **1 Gbps** significa **10^9 bit/s**.

Formattazione dei dischi

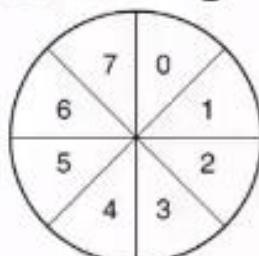
- La formattazione influisce anche sulle prestazioni.
- Se un disco 10k RPM dispone di 300 settori per traccia di 512 byte ciascuno, ci vogliono **6 ms** per leggere una traccia con una velocità di trasferimento 24,4 MB/s.
 - Non è possibile andare più veloci, indipendentemente dal tipo di interfaccia (anche disponendo di un'interfaccia SCSI a 160 MB/s).
 - La lettura continua, a queste velocità, richiede un buffer di grandi dimensioni nel controllore.

Formattazione dei dischi

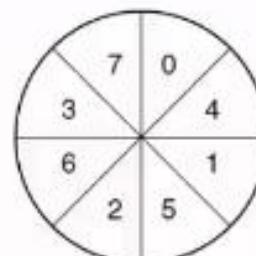
- Si consideri, ad esempio, un controller con un buffer della capacità di un settore che deve leggere due settori **consecutivi**.
- Dopo aver letto il primo settore del disco ed eseguito il calcolo dell'ECC, i dati devono essere trasferiti alla memoria principale.
- Quando la copia della memoria sarà completata, il controllore dovrà aspettare almeno un intero periodo di rotazione prima che arrivi il secondo settore.
- Questo problema può essere eliminato numerando i settori in modalità **interleaved** durante la formattazione del disco.

Formattazione dei dischi

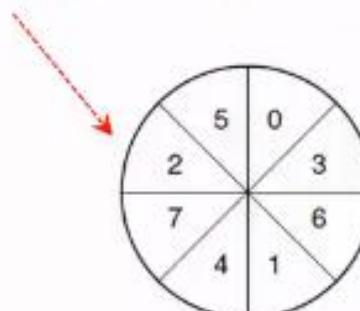
- Se il processo di copia è molto lento, potrebbe essere necessario il doppio interleaving.



Nessun
interlacciamento



Interlacciamento
singolo



Interlacciamento
doppio

- Dopo la formattazione a basso livello, il disco viene partizionato: ogni partizione è come un disco separato.
- Le partizioni permettono l'esistenza di più sistemi operativi.
- Nella maggior parte dei computer, il settore 0 contiene il **Master Boot Record (MBR)** che contiene la **tabella delle partizioni** ove è scritto il settore di avvio e la dimensione di ciascuna partizione.

Formattazione dei dischi

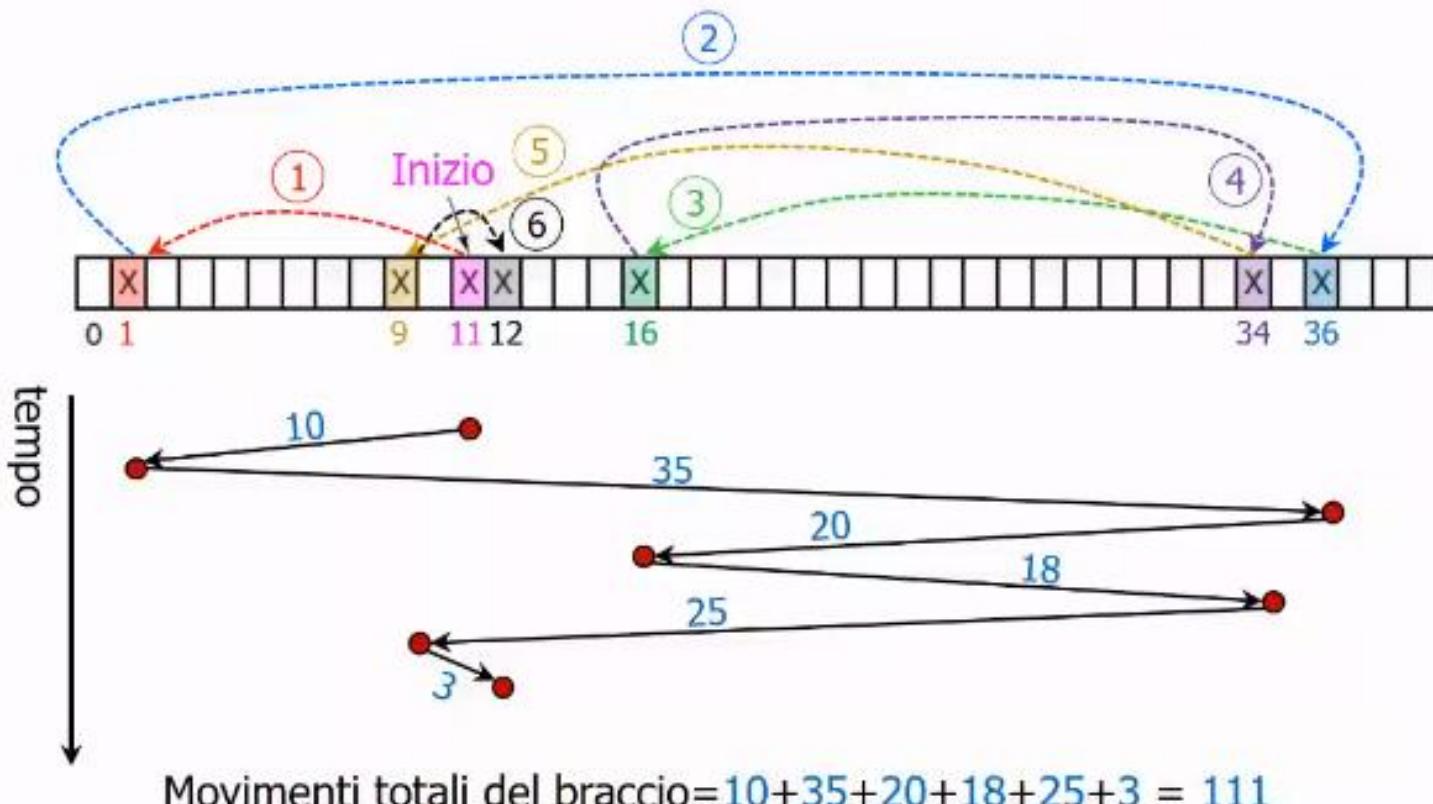
- Sul Pentium, la **tabella delle partizioni** ha spazio per quattro partizioni.
 - Su Windows si chiamano unità **C:**, **D:**, **E:** ed **F:**.
- Per essere in grado di fare il **boot** dal disco rigido, una partizione deve essere contrassegnata come attiva nella **tabella delle partizioni**.
- Il passo finale nella preparazione di un disco è di formattare ad alto livello ciascuna partizione, questa operazione crea:
 - 1) un blocco di avvio.
 - 2) una free list o una bitmap per la gestione dello spazio libero.
 - 3) una directory principale (**root**).
 - 4) un file system vuoto.
- A questo punto il sistema è pronto per essere avviato.

Algoritmi di scheduling delle richieste

- Il tempo richiesto per leggere o scrivere un blocco del disco dipende da tre fattori:
 - 1) **Tempo di ricerca** o **seek** (il tempo necessario a posizionare la testina sul giusto cilindro).
 - 2) **Ritardo di rotazione** (il tempo affinché il settore corretto ruoti sotto la testina).
 - 3) **Tempo di trasferimento dei dati.**
- Nella maggior parte dei dischi il **tempo di seek** è il fattore più critico poiché domina gli altri due valori.
- Se il driver del disco accetta una richiesta alla volta nell'ordine in cui arrivano (First-Come First-Served), poco si può fare per ottimizzare il **tempo di ricerca**.

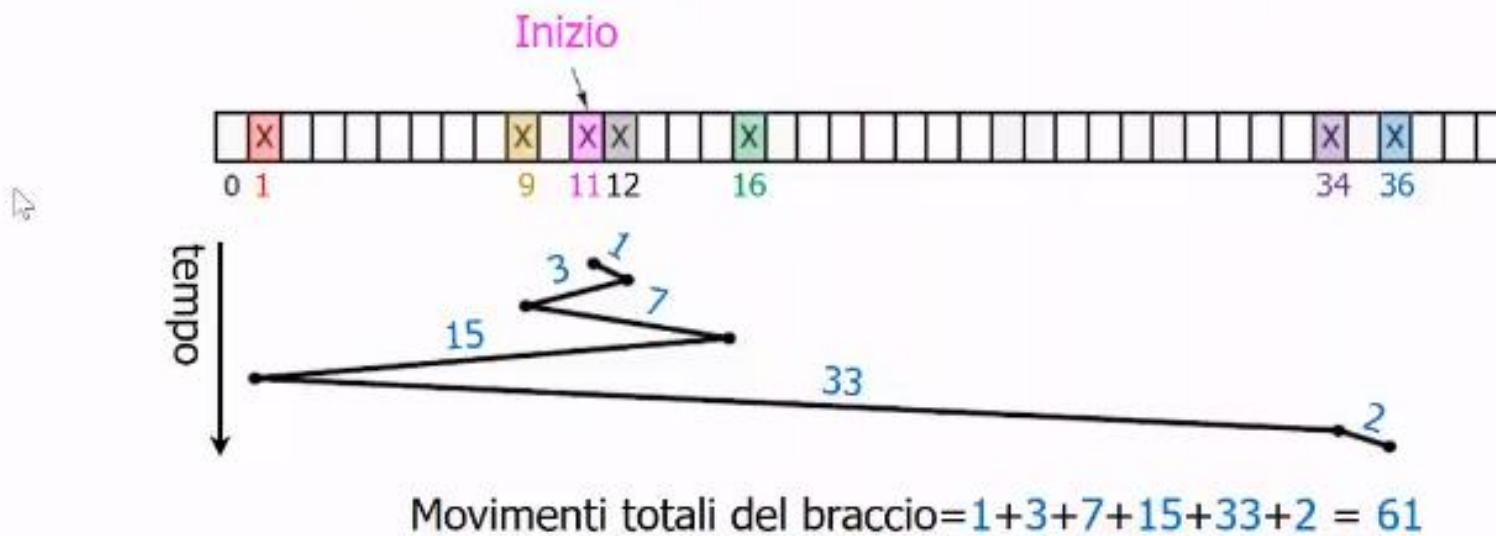
First-Come First-Served (FCFS)

- Supponendo che la posizione iniziale è il cilindro 11 e che arrivino le richieste sui cilindri: 1, 36, 16, 34, 9 e 12.



Shortest Seek First (SSF)

- La posizione iniziale è 11 e le richieste sono: 1, 36, 16, 34, 9 e 12.
- L'algoritmo riordina le richieste al fine di minimizzare il tempo di ricerca (**Shortest Seek First**): 12, 9, 16, 1, 34 e 36.



- In questo esempio, l'algoritmo riduce il movimento totale del braccio di circa il **55%** rispetto al **FCFS**.

L'algoritmo dell'ascensore

- **SSF** ha un problema simile a quello degli ascensori degli edifici alti: i cilindri (piani) centrali sono serviti meglio (perché hanno maggiore priorità) rispetto a quelli posti alle due estremità.
- L'**algoritmo dell'ascensore** utilizza un bit che rappresenta la direzione attuale del ascensore: **UP** o **DOWN**.
- Quando ha completato una richiesta, il driver del disco (o dell'ascensore) controlla il bit di direzione, se è:
 - **UP**
 - se esiste una richiesta pendente verso l'alto il movimento del braccio è in alto altrimenti il bit di direzione è invertito.
 - **DOWN**
 - se esiste una richiesta pendente verso il basso il movimento del braccio è in basso altrimenti il bit di direzione è invertito.

L'algoritmo dell'ascensore

- La posizione iniziale è 11 e le richieste sono: 1, 36, 16, 34, 9 e 12.
- L'ordine in cui sono serviti i cilindri è 12, 16, 34, 36, 9 e 1.



- Anche se in questo esempio il risultato è migliore di **SSF**, in generale accade l'esatto contrario.
- Indipendentemente dalle richieste, nell'algoritmo dell'ascensore il limite superiore dei movimenti è pari a due volte il numero di cilindri.

Gestione degli errori

- I produttori di dischi si spingono ai limiti della tecnologia, aumentando la densità lineare dei bit.
- I difetti di fabbricazione producono settori danneggiati, cioè settori che non rileggono correttamente il valore che hanno appena scritto.
 - Se il difetto è di pochi bit, è possibile correggerlo ogni volta grazie al valore ECC.
 - altrimenti l'errore non può essere mascherato.
- Nel caso di blocchi danneggiati possono essere adottati due diversi approcci:
 - 1) a livello di controller.
 - 2) a livello di sistema operativo.

Memoria stabile

- I dischi a volte commettono degli errori: un settore buono si può deteriorare e divenire difettoso. Anche l'intero disco può smettere di funzionare in modo imprevisto.
- I RAID proteggono i dati contro queste difettosità, ma non dagli errori di scrittura che alterano i dati originali senza sostituirli dai nuovi.
 - ↳ Idealmente, un disco non dovrebbe mai avere errori (irrealizzabile).
- Invece è possibile realizzare un **sistema stabile** che quando gli arriva un comando di scrittura, ha due possibilità:
 - 1) scrivere i dati correttamente.
 - 2) non fare nulla.
- L'obiettivo è la **coerenza dei dati a tutti i costi**.
- Si utilizza una coppia di dischi identici con blocchi omologhi che lavorano insieme per realizzare blocchi privi di errori.

Memoria stabile

- Per ottenere questo obiettivo, sono definite le tre operazioni:
 - 1) **Scritture stabili:** viene scritto il blocco sul **disco 1**, e successivamente riletto per verificare che sia stato scritto correttamente.
 - Se ciò non accade, le operazioni di scrittura e rilettura sono eseguite «n» volte fino a quando non funzionano.
 - al termine degli «n» tentativi si utilizza un blocco di riserva.
 - Una volta scritto il **disco 1**, si ripetono le stesse operazioni sul **disco 2** finché anch'esso avrà il medesimo contenuto.
 - 2) **Letture stabili:** viene letto il blocco dal **disco 1**. Se questo ha un ECC errato, la lettura è provata di nuovo, fino a «n» volte.
 - Se non si riesce ad ottenere un ECC valido, viene letto il **disco 2** (è altamente improbabile che siano entrambi guasti).

Memoria stabile

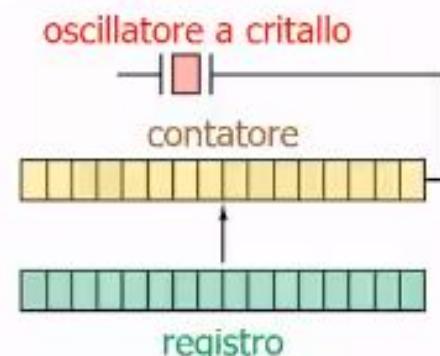
- 3) **Crash recovery**: dopo un crash, un programma di ripristino esegue la scansione di entrambi i dischi e confronta i blocchi omologhi.
- Se uno di loro ha un errore di ECC, il blocco difettoso viene sovrascritto con quello dell'altro disco.
 - Se sono entrambi validi ma con diverso contenuto, il blocco del **disco 1** sovrascrive il blocco del **disco 2**.

CLOCK

- I **clock** (chiamati anche **timer**) sono essenziali per il funzionamento di un sistema multiprogrammato per molte ragioni.
- Innanzitutto segnano il tempo e permettono il controllo del tempo di esecuzione reale dei processi evitando così il monopolio della CPU.
- Il software del **clock** può assumere la forma di un driver di un dispositivo, anche se non è né un dispositivo a blocchi, come un disco, né un dispositivo a caratteri.
- Come sempre considereremo prima l'hardware e poi il software dei **clock**.

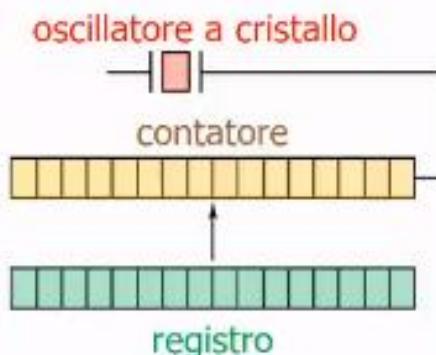
Hardware del clock

- Nei computer sono utilizzati due tipi di **clock**, entrambi sono molto diversi dagli orologi che conosciamo abitualmente.
- I primi **clock** erano più semplici perché bastava collegarli alla tensione di alimentazione (110/220 Volt) per avere un interrupt ad ogni ciclo di tensione (50 o 60 Hz). Oggi non esistono più.
→
- L'altro tipo **clock** è costruito in tre componenti:
 - 1) Un **oscillatore a cristallo**.
 - 2) Un **contatore**, decrementato ad ogni impulso.
 - 3) Un **registro**, utilizzato per caricare il contatore.



Hardware del clock

- Quando un pezzo di cristallo di quarzo è sottoposto ad una tensione, per la sua proprietà piezo-elettrica, genera un segnale periodico (centinaia di MHz a seconda del cristallo prescelto).
- Il segnale base può essere moltiplicato per un numero intero per arrivare a frequenze dell'ordine dei GHz. Questi circuiti sono presenti in ogni computer.
- Il segnale è inviato ad un contatore che decrementa il suo valore fino a zero (interrupt) e poi ricomincia.



Hardware del clock

- I **clock** programmabili hanno due modalità di funzionamento:
 - 1) **One-shot**, quando il clock parte, copia il valore del registro nel contatore e quindi decrementa il contatore ad ogni impulso del cristallo. Quando il contatore arriva a zero, provoca un interrupt e si arresta finché non viene esplicitamente riavviato dal software.
 - 2) **Onda quadra**, dopo aver raggiunto lo zero e causato l'interruzione, il registro è copiato nel contatore e il processo si ripete all'infinito. Gli interrupt periodici sono detti **clock ticks**.
- Il vantaggio dei clock programmabili è che la frequenza di interrupt può essere controllata dal software:
 - Con un cristallo a 500MHz ($T=2\text{ns}$) il contatore decrementa ogni 2ns, se si usa un registro a 32 bit gli interrupt si possono programmare da 2ns a 8,6s.

Software del clock

- L'hardware **clock** genera interrupt a intervalli ben precisi.
- Tutto il resto che coinvolge il tempo deve essere fatto dal software:
il **driver del clock** .
- Tra i compiti più comuni riscontrati in vari SO:
 - 1) Mantenere l'ora del giorno.
 - 2) Evitare che i processi siano eseguiti più tempo di quanto consentito.
 - 3) Contabilizzare l'uso della CPU.
 - 4) Gestire la system call **alarm()** invocata dai processi utente.
 - 5) Fornire il **timer watchdog** per le componenti del SO che ne fanno richiesta.
 - 6) Eseguire il profiling, il monitoraggio ed elaborazioni statistiche.

Timer soft

- Quasi tutti i computer hanno un secondo orologio programmabile che può essere impostato per causare le interruzioni a qualsiasi frequenza.
- I **timer soft** evitano interruzioni: è il kernel che, quando è in esecuzione per qualche ragione, prima di rientrare nella modalità utente controlla che non sia scaduto un **timer soft**.
- I **timer soft** sono controllati con la stessa velocità con cui il kernel entra in azione per svolgere altri compiti, quindi quando:
 - 1) viene attivata una system call.
 - 2) accade una Page miss della TLB.
 - 3) accade un Page fault.
 - 4) accade un interrupt di I/O.
 - 5) la CPU diventa inattiva (idle).

Conclusioni

- Completati i principi software connessi con la gestione dell'I/O.
 - Analizzato i diversi strati software nella gestione dell'I/O da parte del sistema operativo.
- ➔
- Studiato i principi di funzionamento dei dischi e i diversi algoritmi di scheduling per le richieste di accesso al disco.
 - Analizzato le caratteristiche hardware e software dei clock.

INTRODUZIONE

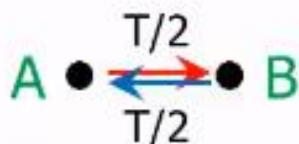
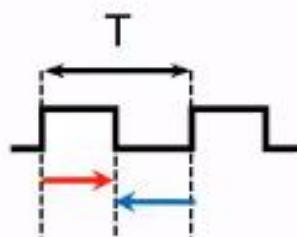
- L'obiettivo principale dell'industria dei computer è da sempre orientata verso l'incremento delle performance.
- In passato questo è stato possibile incrementando la frequenza del clock.
- Si è giunti così ad un limite fisico del materiale poiché, in accordo con la teoria della relatività di Einstein, nessun segnale elettrico può propagarsi più velocemente della velocità della luce (circa **20 cm/nsec** in un cavo di rame o nella fibra ottica).
- Questo significa che in un computer con un clock di 10 GHz, i segnali non possono percorrere più di 2cm.

$$F = 10 \text{ GHz} \rightarrow T = \frac{1}{F} = 0.1 \text{ ns}$$

$$\text{spazio} = \text{velocità} \cdot \text{tempo} = 20 \cdot \frac{\text{cm}}{\text{ns}} \cdot 0.1 \text{ ns} = 2 \text{ cm}$$

INTRODUZIONE

- Un computer con un clock di 1 THz (10³ GHz) dovrebbe essere più piccolo di 100 μm, per permettere al segnale di andare avanti ed indietro in un ciclo di clock.



$$T = \frac{1}{10^{12}} = 10^{-3} \text{ ns}$$

$$\text{spazio} = 20 \cdot \frac{\text{cm}}{\text{ns}} \cdot 10^{-3} \text{ ns} = 2 \cdot 10^{-4} \text{ m} = 200 \mu\text{m}$$

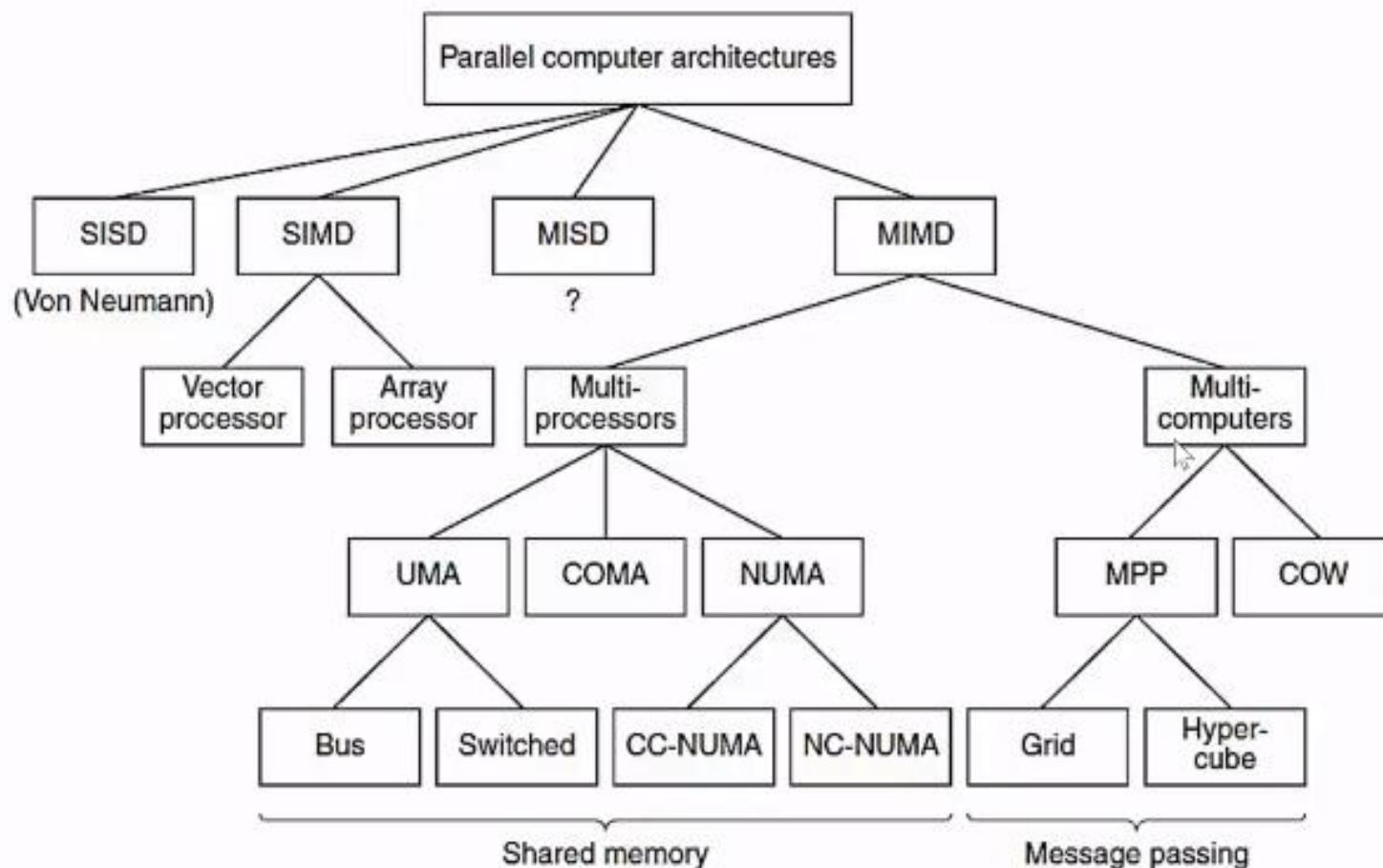
- Rendere computer così piccoli è possibile ma andiamo incontro ad un nuovo problema: la **dissipazione del calore**.
- Un modo per ottenere una maggiore potenza di calcolo è attraverso l'utilizzo delle architetture parallele: molte CPU (con velocità normale) che collaborano per il conseguimento del medesimo obiettivo.

Classificazione di Flynn (1966)

- La classificazione si basa su due concetti: il flusso di istruzioni ed il flusso dei dati.

Flusso di Istruzioni	Flusso di Dati	Nome	Esempio
Singolo	Singolo	SISD	Modello di Von Neumann
Singolo	Multiplo	SIMD	Supercomputer vettoriali
Multiplo	Singolo	MISD	Non sono note
Multiplo	Multiplo	MIMD	Multiprocessori e Multicomputer

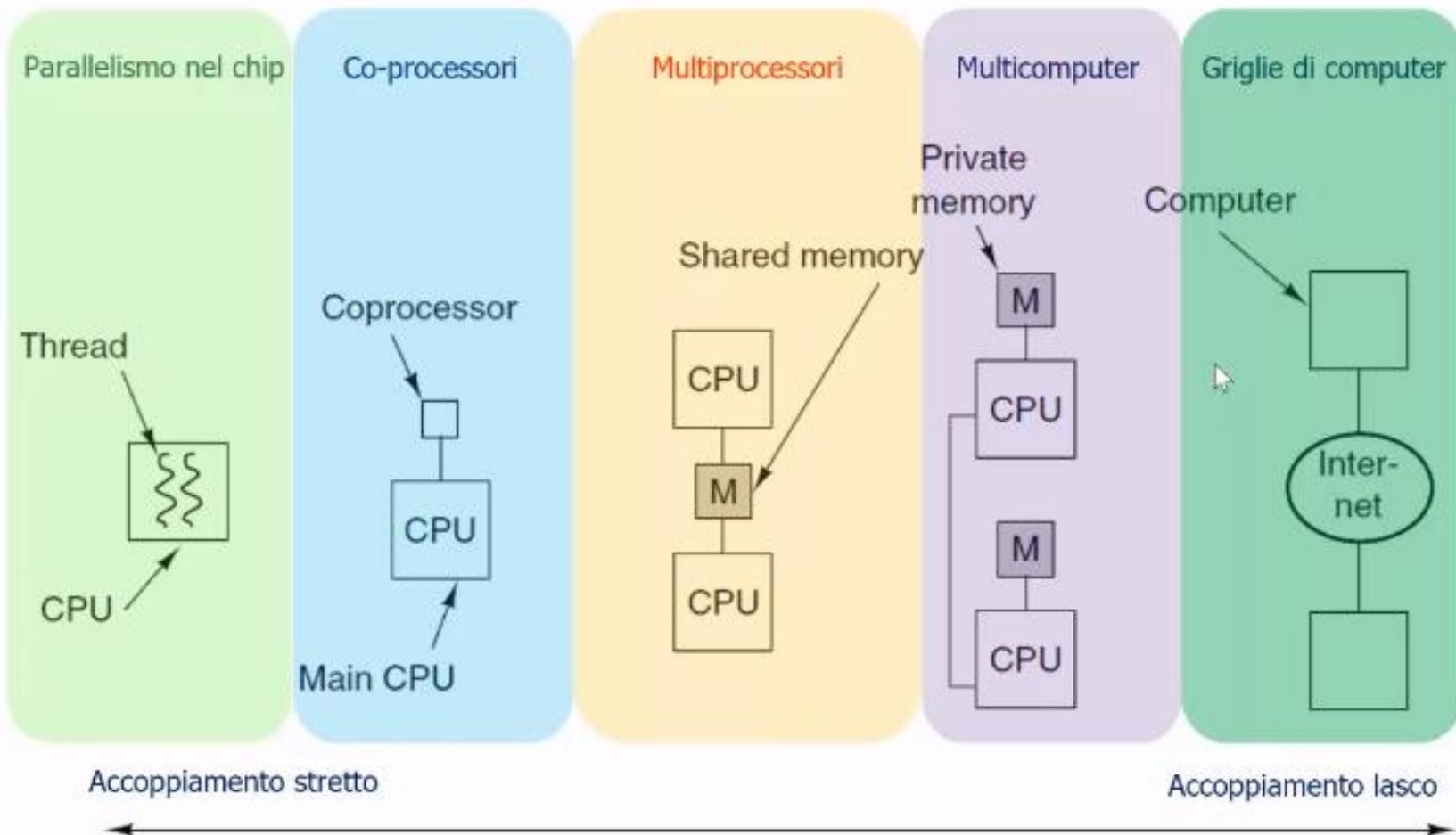
Tassonomia dei calcolatori paralleli



Architetture per il calcolo parallelo

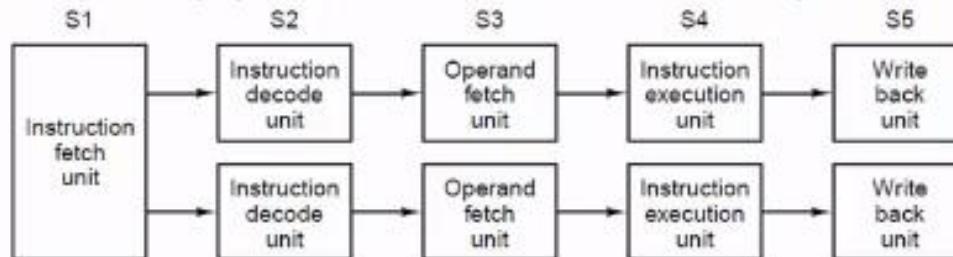
- Il parallelismo nel chip aiuta a migliorare le performance della CPU: con il pipelining e le architetture superscalari si può arrivare ad un fattore di miglioramento da **5** a **10**.
- Per incrementare drasticamente le performance di un calcolatore occorre progettare sistemi con molte CPU, in questo caso si può arrivare ad ottenere un incremento di **50**, **100**, o anche di più
- Esistono così tre differenti approcci:
 - **Data Parallel Computers** (SIMD)
 - **Multiprocessors** (MIMD)
 - **Multicomputers** (MIMD)

Quadro di sintesi delle architetture

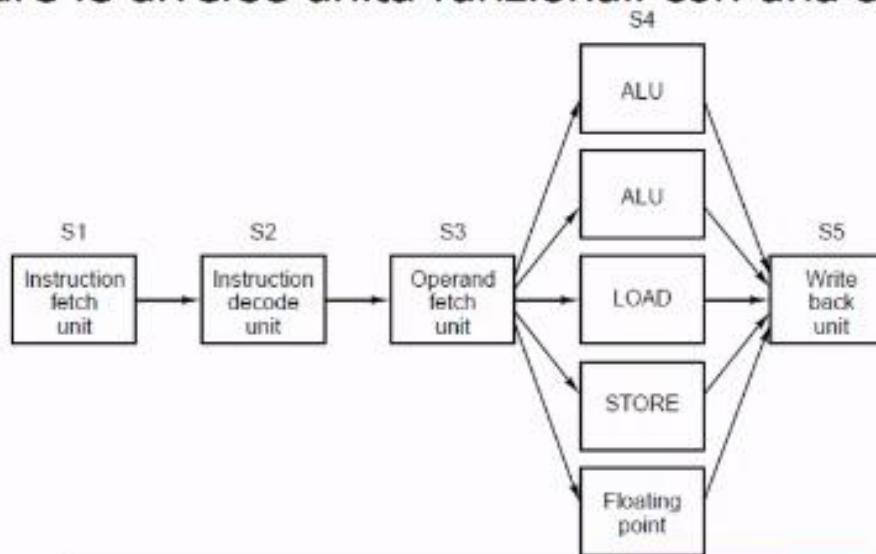


Parallelismo a livello delle istruzioni

- CPU superscalari: pipeline + unità funzionali parallele



- Processori VLIW (Very Long Instruction Word) in grado di indirizzare le diverse unità funzionali con una sola linea di pipeline

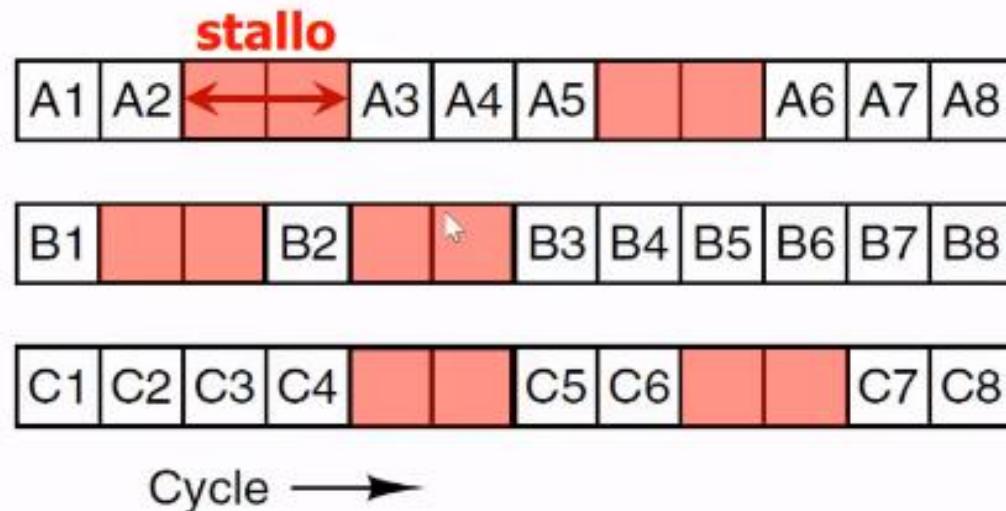


Il problema dello stallo della pipeline

- Il problema dello stallo si verifica quando una CPU tenta di accedere ad un riferimento in memoria che non è nella cache, quindi deve attendere il caricamento prima di riprendere l'esecuzione.
- Il **Multithreading nel chip** permette di mascherare queste situazioni attraverso lo switch tra thread, esistono differenti approcci:
 - **Multithreading a grana fine**
 - **Multithreading a grana grossa**
 - **Multithreading simultaneo**

Il problema dello stallo della pipeline

- Supponiamo di avere una CPU che emette una istruzione per ciclo di clock con tre thread A, B e C.
- Durante il primo ciclo A, B e C eseguono le istruzioni A1, B1 e C1
- Purtroppo al secondo ciclo di clock A2 fa un riferimento non presente nella cache di primo livello e deve attendere 2 cicli per recuperarlo dalla cache di secondo livello.



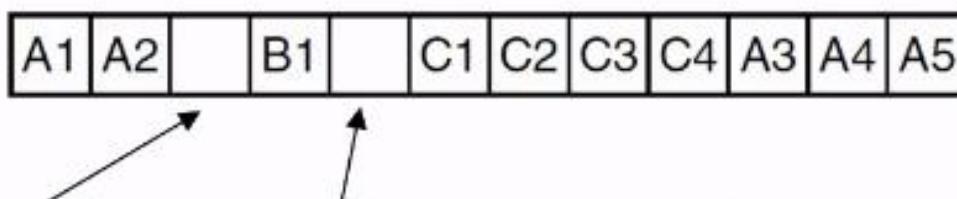
Multithreading a grana fine

- Sono eseguite ogni ciclo di clock, a turno, le singole istruzioni dei thread

A1	B1	C1	A2	B2	C2	A3	B3	C3	A4	B4	C4
----	----	----	----	----	----	----	----	----	----	----	----
- Nei due cicli di stallo la CPU è occupata a svolgere le istruzioni degli altri due thread.
- Poiché non c'è alcuna relazione tra i thread, ciascuno ha il proprio insieme di registri. Il numero massimo di thread concorrenti è definito a priori in fase di progettazione del chip.
- Le operazioni in memoria non sono l'unica ragione di stallo: alcune istruzioni condizionano l'esecuzione di altre.
- Nella pipeline non ci sarà mai più di una istruzione per thread e quindi il numero massimo di thread è pari al numero di stadi della pipeline (non è detto che ci siano tanti thread quanti gli stadi della pipeline).

Multithreading a grana grossa

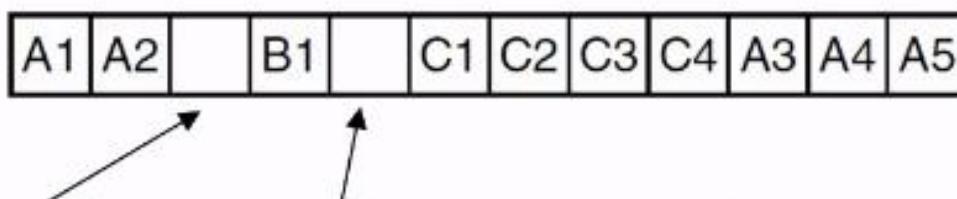
- Un thread va avanti finché non raggiunge uno stallo, perde un ciclo e commuta su un altro thread.



- Perde un ciclo ogni stallo ed è inefficiente rispetto al multithreading a grana fine.
- richiede meno thread per mantenere occupata la CPU
- Esiste una variante che permette di "guardare avanti" le istruzioni anticipando lo stallo e approssimando il multithreading a grana fine.

Multithreading a grana grossa

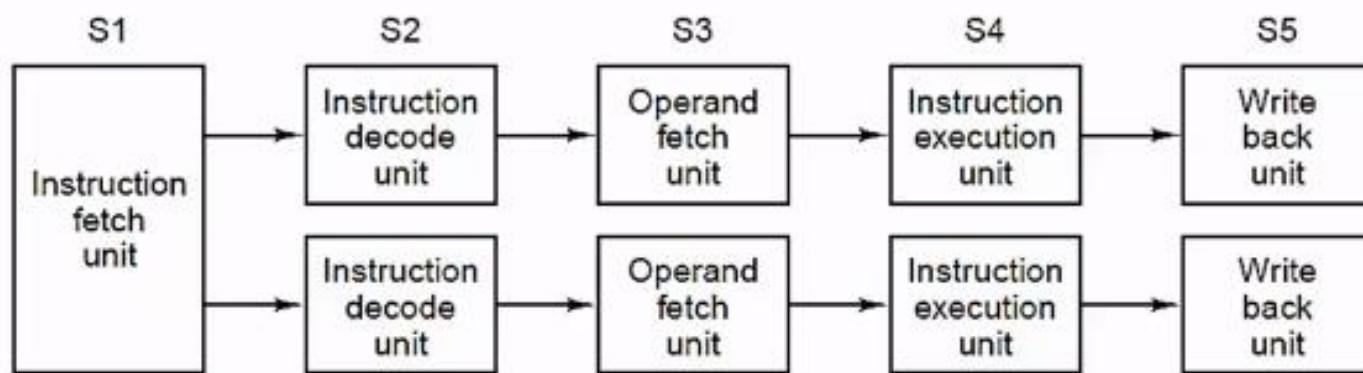
- Un thread va avanti finché non raggiunge uno stallo, perde un ciclo e commuta su un altro thread.



- Perde un ciclo ogni stallo ed è inefficiente rispetto al multithreading a grana fine.
- richiede meno thread per mantenere occupata la CPU
- Esiste una variante che permette di "guardare avanti" le istruzioni anticipando lo stallo e approssimando il multithreading a grana fine.

Multithreading in CPU superscalari

- Le CPU possono avere più unità in parallelo ed emettere più istruzioni per ciclo



Multithreading in CPU superscalari

- In ogni thread sono eseguite due istruzioni per ciclo finché non si raggiunge uno stallo:

A1	A2			A3	A4	A5			A6	A7	A8
----	----	--	--	----	----	----	--	--	----	----	----

Esempio iniziale

B1			B2			B3	B4	B5	B6	B7	B8
----	--	--	----	--	--	----	----	----	----	----	----



C1	C2	C3	C4			C5	C6			C7	C8
----	----	----	----	--	--	----	----	--	--	----	----

Cycle →

A1	B1	C1	A3	B2	C3	A5	B3	C5	A6	B5	C7
A2		C2	A4		C4		B4	C6	A7	B6	C8

Multithreading a
Grana fine

A1	B1	C1	C3	A3	A5	B2	C5	A6	A8	B3	B5
A2		C2	C4	A4			C6	A7		B4	B6

Multithreading a
Grana grossa

Multithreading simultaneo

A1	B1	C1	A3	B2	C3	A5	B3	C5	A6	B5	C7
A2		C2	A4		C4		B4	C6	A7	B6	C8

Multithreading a
Grana fine

A1	B1	C1	C3	A3	A5	B2	C5	A6	A8	B3	B5
A2		C2	C4	A4			C6	A7		B4	B6

Multithreading a
Grana grossa

- Ciascun thread emette due istruzioni per ciclo finché non si raggiunge uno stallo



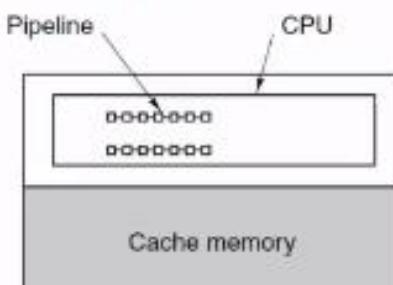
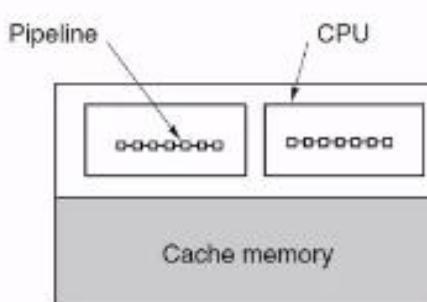
A1	B1	C2	C4	A4	B2	C6	A7	B3	B5	B7	C7
A2	C1	C3	A3	A5	C5	A6	A8	B4	B6	B8	C8

- A questo punto si passa all'istruzione del thread che segue affinché la CPU rimanga impiegata

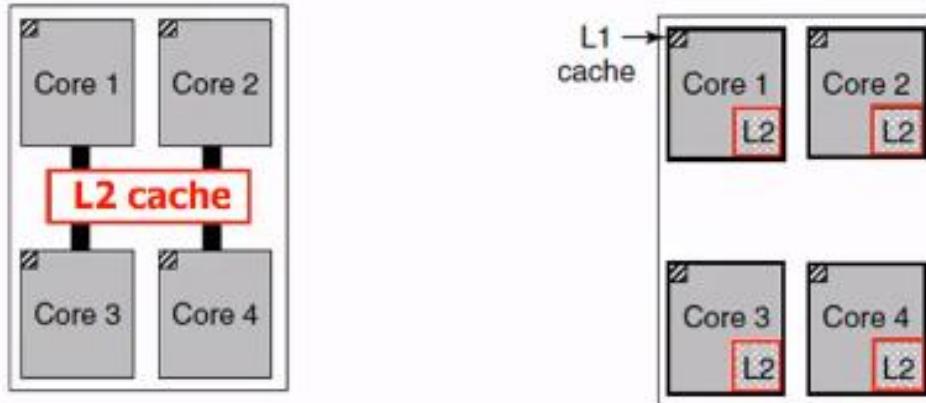
Multiprocessori in un solo chip

- Con l'andare avanti della tecnologia, i transistor sono diventati più piccoli ed è stato possibile incrementare il loro numero in un singolo chip.
- A causa del raggiungimento dei limiti fisici dei materiali (dissipazione del calore e distanza percorsa dai segnali all'interno dei semiconduttori) non è stato possibile incrementare la frequenza di clock per ottenere CPU più veloci.
- Per tale ragione le industrie produttrici di chip hanno incominciato ad inserire più CPU (chiamate **core**) all'interno dello stesso chip (o meglio **die**).
- I multiprocessori possono essere realizzati con core identici (**multiprocessori omogenei**) oppure con core con specifiche funzionalità (**multiprocessori eterogenei**).

Multiprocessori omogenei in un solo chip

- Le CPU che contengono più processori e che condividono le cache e la memoria principale sono dette multiprocessori.
- Esistono due tecnologie di multiprocessori in un solo chip:
 - 1) Quelle con una sola CPU e più pipeline che possono moltiplicare il throughput in base al numero di pipeline. Le unità funzionali sono intimamente correlate.
 - 2) Quelle che hanno più CPU (chiamate core) ciascuna con la propria pipeline. L'interazione tra CPU non è semplice poiché risultano maggiormente disaccoppiate.

Multiprocessori omogenei in un solo chip



- Mentre le CPU possono o meno condividere le cache, esse condividono sempre la memoria principale
- Il processo di **snooping** (eseguito dall'hardware) garantisce che se una parola è presente in più cache e una CPU modifica il suo valore in memoria, essa è automaticamente rimossa in tutte le cache in modo da garantire consistenza.

Multiprocessori eterogenei in un solo chip

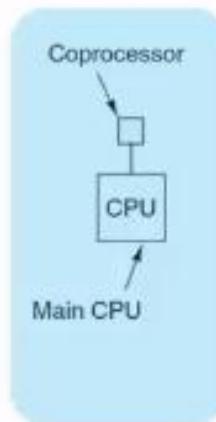
- Oltre ai multicore simmetrici esiste un altro tipo di chip multicore in cui ogni core ha un compito specifico (come ad esempio decoder audio/video, criptoprocessore, interfacce di rete).
 - Poichè queste architetture realizzano un vero e proprio calcolatore completo in un singolo chip sono spesso dette **system on a chip**.
- Come accaduto spesso nel passato, l'**hardware è molto più avanti del software**: mentre sono attualmente disponibili dei chip multicore, non abbiamo applicazioni in grado di sfruttare queste nuove caratteristiche.
- Pochi programmati sono in grado di scrivere algoritmi paralleli che gestiscano correttamente la competizione delle risorse condivise.

Chip-level Multiprocessor

- I chip multicore sono come dei piccoli multiprocessori e per questa ragione vengono chiamati **CMP (Chip-level MultiProcessors)** ovvero multiprocessori a livello di chip).
- Dal punto di vista software essi non sono così differenti dai multiprocessori a bus o a reti di switch.
- Rispetto a multiprocessori a bus che hanno una cache per ogni CPU, potrebbero avere delle prestazioni degradate sulla cache condivisa nelle situazioni in cui un core “ingordo” saturi la cache L2.
- Altra differenza rispetto ai multiprocessori è la minore tolleranza ai malfunzionamenti causata dalla stretta connessione dei core (un errore su uno potrebbe propagarsi negli altri).

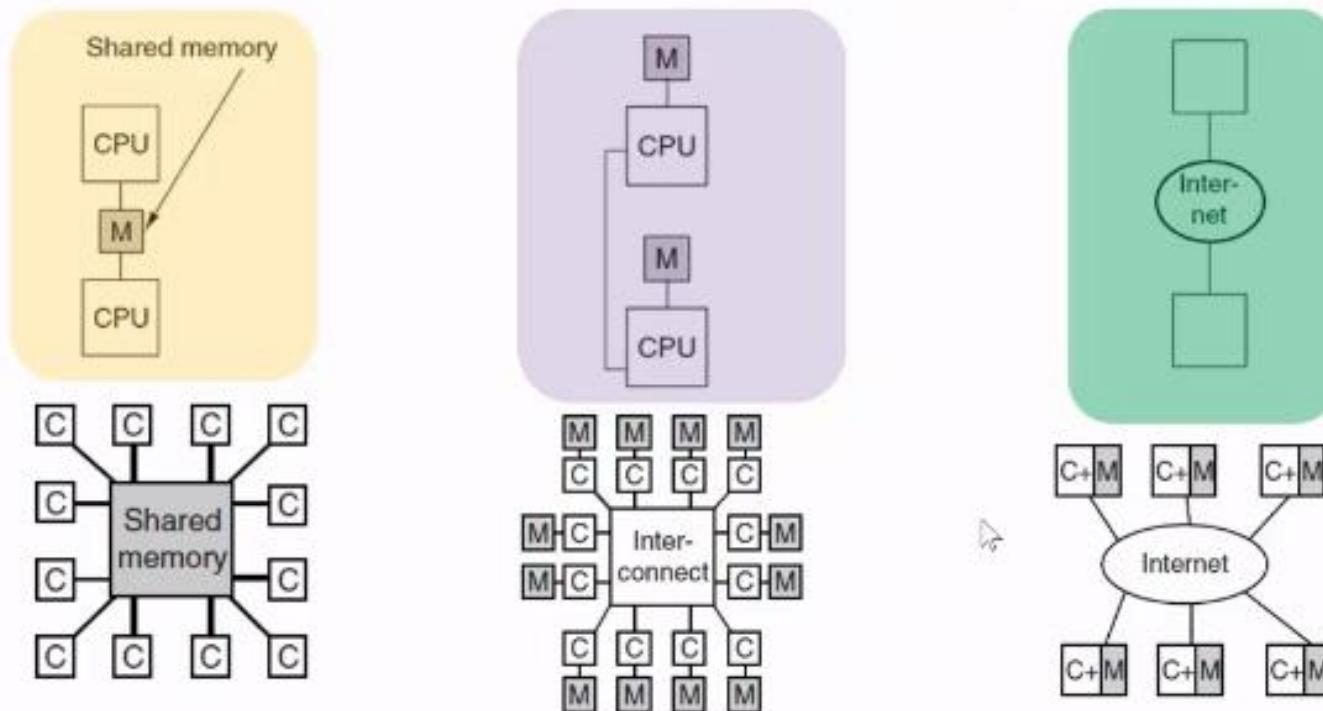
Coprocessori

- L'utilizzo di un processore dedicato a specifiche funzioni (**coprocessore**) permette di migliorare le performance di un calcolatore.
- Esistono molte varianti di coprocessori:
 - Processori di rete.
 - Processori grafici.
 - Crittoprocessori.



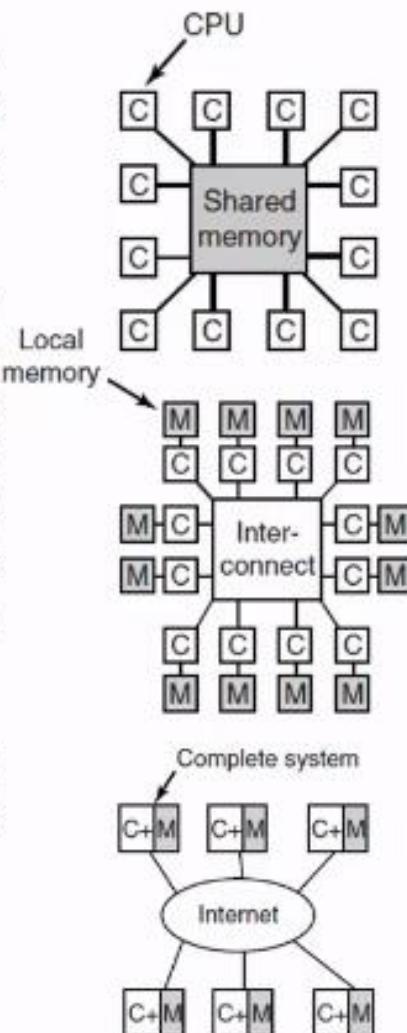
MIMD

- Tutte le comunicazioni tra componenti elettronici (o ottici) avvengono attraverso lo scambio di messaggi.
- Le principali differenze tra MIMD risiedono nella base dei tempi, nella scala delle distanze e nell'organizzazione logica utilizzata



Multiprocessori e Multicomputer (MIMD)

- 1) **Multiprocessori a memoria condivisa**, meno di **1000** CPU comunicano attraverso una memoria condivisa; il tempo di accesso ad una parola di memoria è di **2-10 ns**.
- 2) **Multicomputer a scambio di messaggi**, un certo numero di coppie memoria-CPU sono collegati attraverso una rete di interconnessione ad alta velocità; le comunicazioni avvengono attraverso brevi messaggio che possono essere spediti in **10-50 µs**; ogni memoria è locale a ciascuna CPU.
- 3) **Sistemi distribuiti**, computer distribuiti su un'area geografica estesa come internet, i messaggi impiegano da **10 a 100 ms**.



Multiprocessori

- Un multiprocessore a memoria condivisa (o semplicemente multiprocessore) è un calcolatore in cui tutte le CPU condividono una memoria comune.
- L'unica proprietà inusuale che ha un multiprocessore è che una CPU può scrivere un valore in una parola di memoria e trovarvi un differente valore alla successiva lettura (perché un'altra CPU può averla modificata).
- I sistemi operativi utilizzati nei Multiprocessori sono normali sistemi operativi.



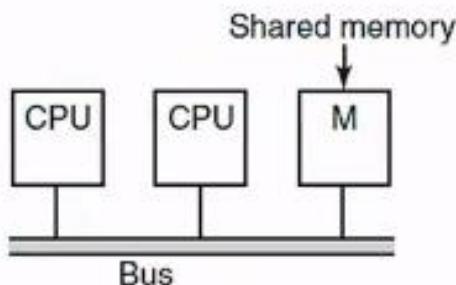
Hardware dei multiprocessori

- Esistono due differenti tipologie di multiprocessori:
 - **Multiprocessori UMA** (Uniform Memory Access) in cui ogni parola di memoria può essere letta alla stessa velocità.
 - **Multiprocessori NUMA** (Nonuniform Memory Access) in cui non tutte le parole di memoria possono essere lette alla medesima velocità.



UMA con architettura basata sul bus

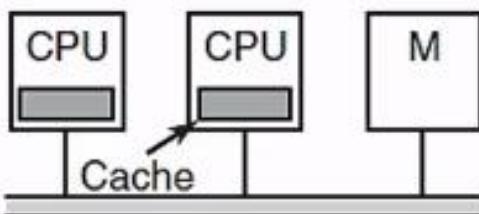
- I più semplici multiprocessori sono basati su un singolobus che interconnette tutte le CPU alla memoria condivisa.



- Una CPU che vuole leggere/scrivere una parola in memoria, se il bus è occupato, deve attendere che si liberi.
- Questa architettura funziona bene con due o tre CPU (il contenzioso sull'accesso al bus può essere gestito agevolmente).
- Per poter incrementare il numero di CPU, e quindi le performance, occorre aggiungere delle ulteriori memorie.

UMA con singolo bus e cache nelle CPU

- L'aggiunta di una memoria cache all'interno delle singole CPU può ridurre il traffico sul bus e il sistema può supportare anche più di tre CPU:



- Il Caching non viene eseguito sulle singole parole di memoria ma su blocchi di 32 o 64 byte. Quando una parola è referenziata, l'intero blocco che la contiene, chiamato **linea di cache**, è caricato nella CPU che l'ha richiesta.
- Ogni blocco di cache è contrassegnato come READ-ONLY oppure READ/WRITE.

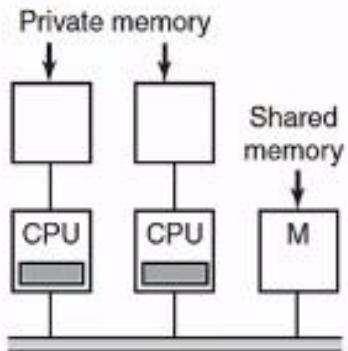
UMA con singolo bus e cache nelle CPU

- Se una CPU scrive una parola che è contenuta anche in altri blocchi di cache remote, l'hardware del bus percepisce la scrittura e lo segnala alle altre cache che possono avere:
 - una copia "pulita" del blocco di memoria, allora la cache la sovrascrive con il valore aggiornato
 - Una copia modificata del blocco di memoria (detta "copia sporca"), allora la cache lo deve prima trascrivere in memoria e poi applicare la modifica
- L'insieme di tali regole, utilizzate per la gestione della cache, è chiamato **protocollo di coerenza della cache**.



UMA con singolo bus e CPU dotate di RAM

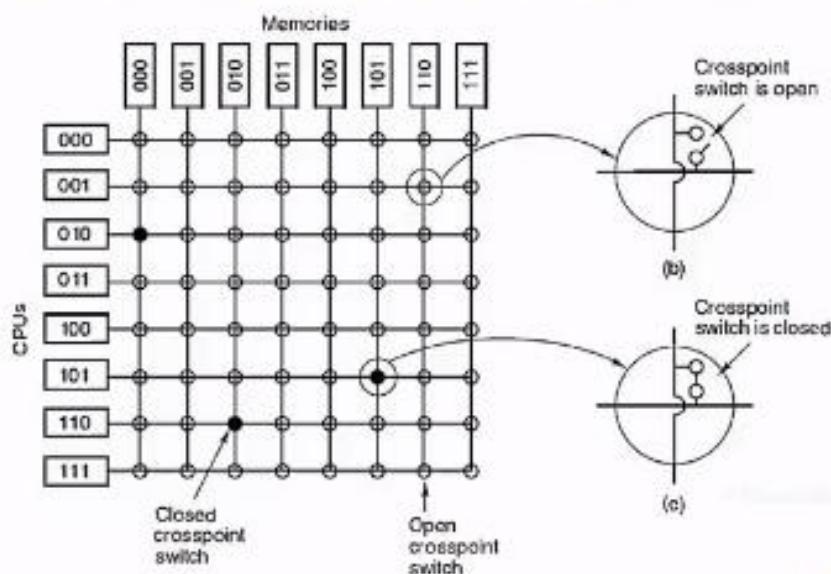
- Un'altra soluzione è con l'aggiunta di memorie RAM in un bus interno dedicato per ogni CPU



- A questo punto la memoria condivisa è utilizzata esclusivamente per scrivere variabili condivise (globali).
- Questa soluzione riduce il traffico sul bus ma richiede una collaborazione attiva del compilatore che deve separare gli oggetti locali (il programma, lo stack, le variabili locali,...) da quelli globali.

UMA con crossbar switch

- Anche con le migliori tecniche di caching, l'uso di un singolo bus di interconnessione limita la dimensione del multiprocessore UMA a 16 o 32 CPU.
- Per andare oltre occorre utilizzare una differente rete di interconnessione.
- Il circuito più semplice che permette di collegare n CPU a k memorie è il **crossbar switch**.



- In ogni intersezione delle linee orizzontali (CPU) con quelle verticali (RAM) c'è un piccolo interruttore (pilotato elettricamente) che permette di stabilire il collegamento tra CPU e RAM.

UMA con crossbar switch

- Attraverso questo sistema si realizza una **rete non bloccante**: ad alcuna CPU è mai vietata la connessione verso una memoria di cui ha bisogno perché qualche crosspoint o linea è già occupata.
- Non è necessaria alcuna azione di pianificazione anticipata. È sempre possibile connettere una CPU alla memoria aprendo o chiudendo un interruttore.
- Rimane il problema della competizione per la memoria, qualora due, o più, CPU vogliono accedere allo stesso modulo nel medesimo istante (partizionando la memoria in n unità, la competizione si riduce di un fattore n rispetto al modello con bus singolo).
- Una delle peggiori caratteristiche di questo schema è che il numero degli incroci cresce come n^2 .
 - con 1000 CPUs e 1000 moduli di memoria occorrono un milione di crosspoints. Costruire una crossbar di queste dimensioni non è fattibile.

UMA con rete di commutazione a più stadi

- Una progettazione di multiprocessori completamente differente è basata su switch 2x2 (due input e due output).
- I messaggi che arrivano nelle due linee di ingresso possono scambiarsi in una delle due linee di uscita.
- Ogni messaggio contiene:

Module	Quale memoria utilizzare
Address	L'indirizzo nel modulo
Opcode	Il codice dell'operazione da eseguire
Valore	Il valore di un operando
- Lo switch utilizza il campo Module per scegliere dove spedire il messaggio (su X o su Y).
- Gli switch possono essere organizzati in vari modi per costruire complesse reti di commutazione a più stadi.

