# *Sistemi Operativi*

## *Modulo 1: Introduzione ai sistemi operativi*

# Sommario

- **Cos'è un sistema operativo?**
  - S.O. come gestore di risorse
  - S.O. come macchina astratta
- **Storia dei sistemi operativi**
  - Generazioni 1-4 (dal 1945 a oggi)
  - Sistemi operativi paralleli, distribuiti, real-time
- **Ripasso di architettura**
  - Concetti fondamentali di architettura degli elaboratori
- **Organizzazione di un sistema operativo**
  - Panoramica sulle funzionalità offerte da un sistema operativo

# Sezione 1

## 1. Cos'è un sistema operativo?

# Cos'è un sistema operativo?

- **Definizione:**
  - Un sistema operativo è un programma che *controlla l'esecuzione di programmi applicativi* e agisce come *interfaccia tra le applicazioni e l'hardware* del calcolatore
- **Obiettivi**
  - Efficienza:
    - Un S.O. cerca di utilizzare in modo efficiente le risorse del calcolatore
  - Semplicità:
    - Un sistema operativo dovrebbe semplificare l'utilizzazione dell'hardware di un calcolatore

# Visione 1: S.O. come gestore di risorse

- **Considerate un ristorante con un capo-cuoco e i suoi aiutanti, una cucina, camerieri e clienti**
  - i clienti scelgono un piatto da un menu
  - un cameriere prende l'ordine e lo consegna al capo-cuoco
  - il capo-cuoco riceve l'ordine e assegna uno o più aiutanti alla preparazione del piatto
  - ogni aiutante si dedicherà alla preparazione di un piatto, il che potrà richiedere più attività diverse
  - il capo-cuoco supervisiona la preparazione dei piatti e gestisce le risorse (limitate) disponibili

# Visione 1: S.O. come gestore di risorse

- **Il capo-cuoco è il sistema operativo!**
  - i clienti sono gli utenti
  - le ricette associate ai piatti corrispondono ai programmi
  - il menu e il cameriere costituiscono l'interfaccia verso il sistema operativo (grafica e non)
  - gli aiutanti corrispondono ai processi
  - la cucina corrisponde al computer; pentole, fornelli, etc. corrispondono alle componenti di un computer

# Visione 1: S.O. come gestore di risorse

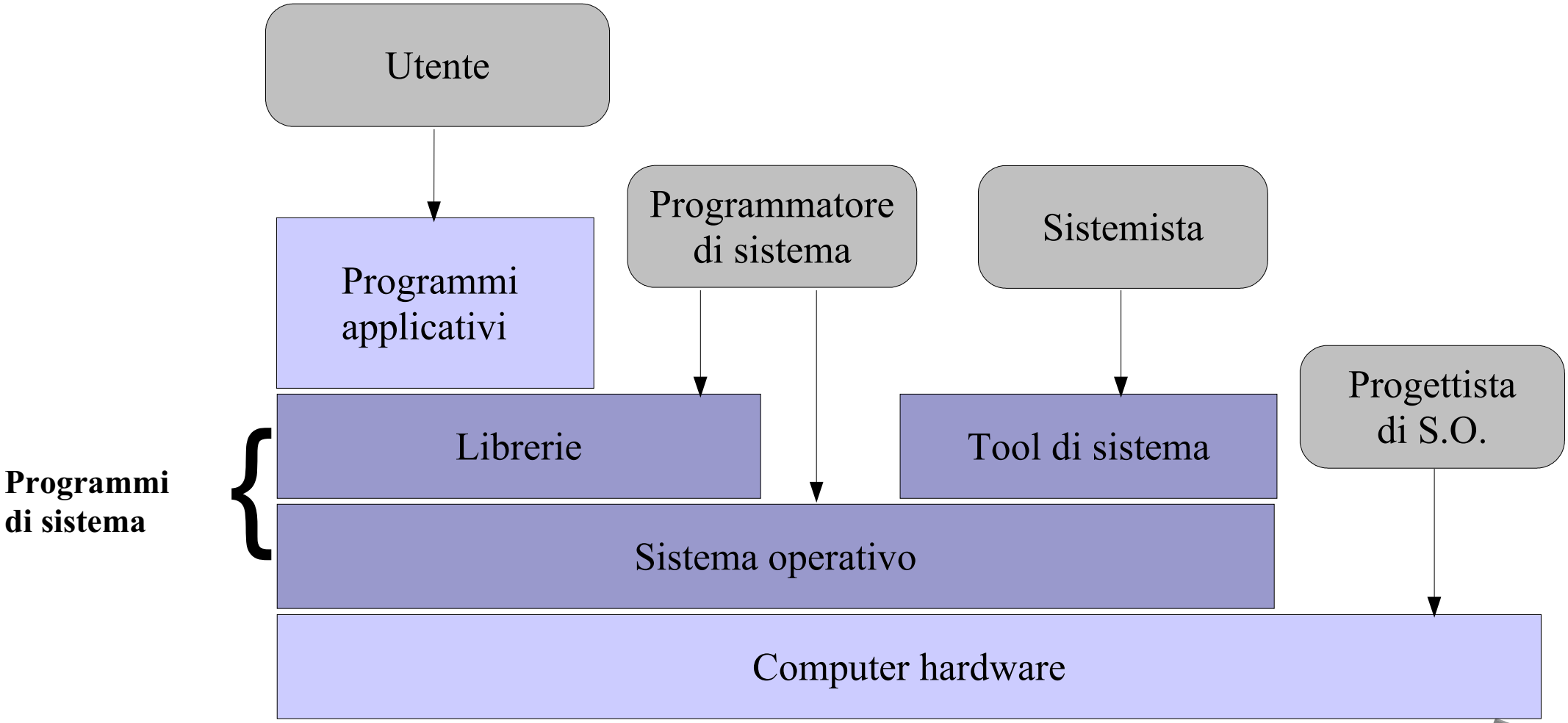
- **Problemi di un capo-cuoco:**
  - esecuzione fedele delle ricette
  - allocazione efficiente delle risorse esistenti (attori, fornelli, etc.)
  - coordinamento efficiente degli attori
  - "licenziamento" degli attori che non si comportano secondo le regole
- **Problemi di un sistema operativo**
  - Efficienza nell'uso delle risorse (processore, memoria, dischi, etc.)
  - Protezione nell'uso delle risorse

# Visione 1: S.O. come gestore di risorse

- **Alcune osservazioni:**
  - Gestendo le risorse di un calcolatore, un S.O. controlla il funzionamento del calcolatore stesso...
  - ... ma questo controllo è esercitato in modo "particolare"
- **Normalmente:**
  - Il meccanismo di controllo è esterno al sistema controllato
  - Esempio: termostato e impianto di riscaldamento
- **In un elaboratore:**
  - Il S.O. è un programma, simile all'oggetto del controllo, ovvero le applicazioni controllate
  - Il S.O. deve lasciare il controllo alle applicazioni e affidarsi al processore per riottenere il controllo

# Visione 2: S.O. come macchina estesa

- **Visione "a strati" delle componenti hardware/software che compongono un elaboratore:**



# Visione 2: S.O. come macchina estesa

- **In questa visione, un sistema operativo:**
  - nasconde ai programmatori i dettagli dell'hardware e fornisce ai programmatori una API conveniente e facile da usare
  - agisce come intermediario tra programmatore e hardware
- **Parole chiave:**
  - Indipendenza dall'hardware
  - Comodità d'uso
  - Programmabilità

# Visione 2: S.O. come macchina estesa

- **Esempio: floppy disk drive**
  - I floppy drive delle macchine Intel sono compatibili con il controllore NEC PD765
  - 16 comandi
    - inizializzazione, avviamento motore, spostamento testina, lettura-scrittura, spegnimento motore
    - formato: vari parametri, impacchettati in 1-9 byte
    - esempio: comando read, 13 parametri
  - al completamento, il driver restituirà 23 campi di stato e di errore racchiusi in 7 byte

# Visione 2: S.O. come macchina estesa

- Esempio senza S.O.

```
li $t0, 0xDEFF12 # init

sw $t0, 0xB000.0040

li $t0, 0xFFDF # motor

sw $t0, 0xB000.0044

li $t0, 0xFFBB

sw $t0, 0xBOOO.0048

...
```

- Esempio con S.O.

```
fd = open("/etc/rpc");

read(fd, buffer, size);
```

NB: Questo è esempio serve a dare un'idea,
la realtà è molto più complessa....

# Visione 2: S.O. come macchina estesa

- **Servizi estesi offerti da un S.O.:**
  - esecuzione di programmi
  - accesso semplificato ai dispositivi di I/O
  - accesso controllato a dispositivi, file system, etc.
  - accesso al sistema
  - rilevazione e risposta agli errori
  - accounting

# 2. Storia dei sistemi operativi

# Storia dei Sistemi Operativi

- **L'evoluzione dei sistemi operativi**
  - *è stata spinta* dal progresso tecnologico nel campo dell'hardware
  - *ha guidato* il progresso tecnologico nel campo dell'hardware
- **Esempio:**
  - Gestione degli interrupt
  - Protezione della memoria
  - Memoria virtuale
  - ....

# Storia dei Sistemi Operativi

- **Perché analizzare la storia dei sistemi operativi?**
  - Perché permette di capire l'origine di certe soluzioni presenti oggi nei moderni sistemi operativi
  - Perché è l'approccio didattico migliore per capire come certe idee si sono sviluppate
  - Perché alcune delle soluzioni più vecchie sono ancora utilizzate
- **Durante il corso:**
  - illustreremo ogni argomento
    - partendo dalle prime soluzioni disponibili
    - costruendo sopra di esse soluzioni mano a mano più complesse
  - non stupitevi quindi se alcune soluzioni vi sembreranno banali e ingenue; sono soluzioni adottate 10,20,30,40 o 50 anni fa!

# Storia dei Sistemi Operativi

- **Generazione 1: 1945 - 1955**
  - valvole e tavole di commutazione
- **Generazione 2: 1955 - 1965**
  - transistor e sistemi batch
- **Generazione 3: 1965 – 1980**
  - circuiti integrati, multiprogrammazione e time-sharing
- **Generazione 4: 1980 – oggi**
  - personal computer

# Generazione 0

- **Babbage (1792-1871)**
  - Cerca di costruire la macchina analitica (programmabile, meccanica)
  - Non aveva sistema operativo
  - La prima programmatrice della storia e' Lady Ada Lovelace (figlia del poeta Lord Byron)

# Generazione 1 (1944-1955)

- **Come venivano costruiti?**
  - macchine a valvole e tavole di commutazione
- **Come venivano usati?**
  - solo calcoli numerici (calcolatori non elaboratori)
  - un singolo gruppo di persone progettava, costruiva, programmava e manuteneva il proprio computer
- **Come venivano programmati?**
  - in linguaggio macchina
    - programmazione su tavole di commutazione
    - non esisteva il concetto di assembler!
- **Nessun sistema operativo!**

# Generazione 1 (1944-1955)

- **Principali problemi**
  - grossi problemi di affidabilità (guasti frequenti)
  - rigidità nell'assegnazione dei ruoli;
    - non esiste il concetto di programmatore come entità separata dal costruttore di computer e dall'utente
  - utilizzazione lenta e complessa; l'operatore doveva:
    - caricare il programma da eseguire
    - inserire i dati di input
    - eseguire il programma
    - attendere il risultato
    - ricominciare dal punto 1.
  - tutto ciò a causa dell'assenza del sistema operativo

# Generazione 1 (1944-1955)

- **Frasi celebri**
  **(da una lezione di P. Ciancarini)**
  - Nel futuro i computer arriveranno a pesare non più di una tonnellata e mezzo (Popular Mechanics, 1949)
  - Penso che ci sia mercato nel mondo per non più di cinque computer (Thomas Watson, presidente di IBM, 1943)
  - Ho girato avanti e indietro questa nazione (USA) e ho parlato con la gente. Vi assicuro che questa moda dell'elaborazione automatica non vedrà l'anno prossimo
    (Editor di libri scientifici di Prentice Hall, 1947)

# Generazione 2 (1955-1965)

- **Come venivano costruiti?**
  - introduzione dei transistor
  - costruzione di macchine più affidabili ed economiche
- **Come venivano usati?**
  - le macchine iniziano ad essere utilizzate per compiti diversi
  - si crea un mercato, grazie alle ridotte dimensioni e al prezzo più abbordabile
  - avviene una separazione tra costruttori, operatori e programmatori
- **Come venivano programmati?**
  - linguaggi ad "alto livello": Assembly, Fortran
  - tramite schede perforate
- **Sistemi operativi batch**

# Generazione 2 (1955-1965)

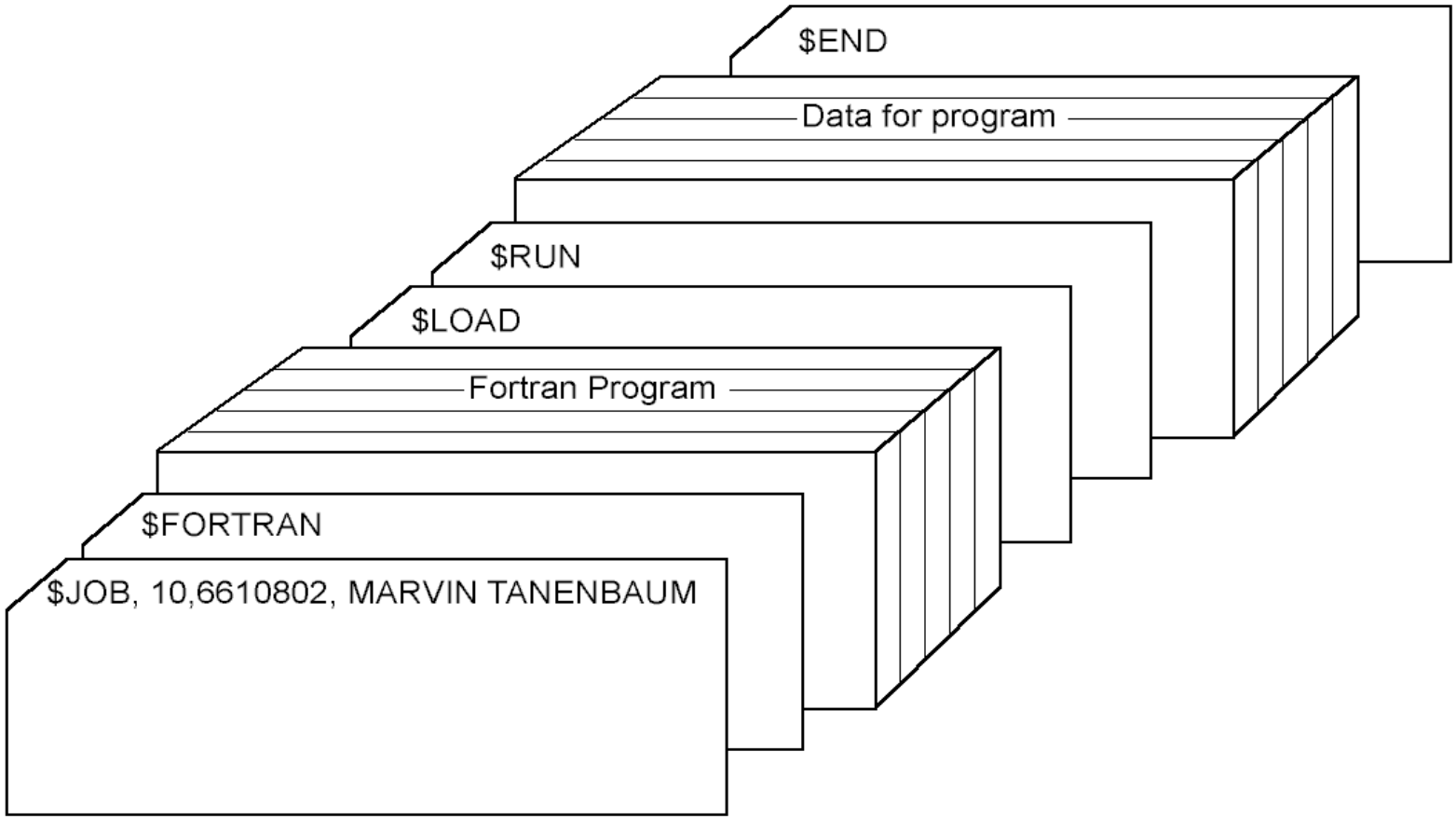
- **Definizione: job**
  - Un programma o un'insieme di programmi la cui esecuzione veniva richiesta da uno degli utilizzatori del computer
- **Ciclo di esecuzione di un job:**
  - il programmatore
    - scrive (su carta) un programma in un linguaggio ad alto livello
    - perfora una serie di schede con il programma e il suo input
    - consegna le schede ad un operatore
  - l'operatore
    - inserisce schede di controllo scritte in JCL
    - inserisce le schede del programma
    - attende il risultato e lo consegna al programmatore
- **Nota: operatore != programmatore == utente**

# Generazione 2 (1955-1965)

- **Sistema operativo**
  - primi rudimentali esempi di sistema operativo, detti anche monitor residenti:
    - controllo iniziale nel monitor
    - il controllo viene ceduto al job corrente
    - una volta terminato il job, il controllo ritorna al monitor
  - il monitor residente è in grado di eseguire una sequenza di job, trasferendo il controllo dall'uno all'altro
- **Detti anche sistemi batch ("infornata")**

# Generazione 2 (1955-1965)

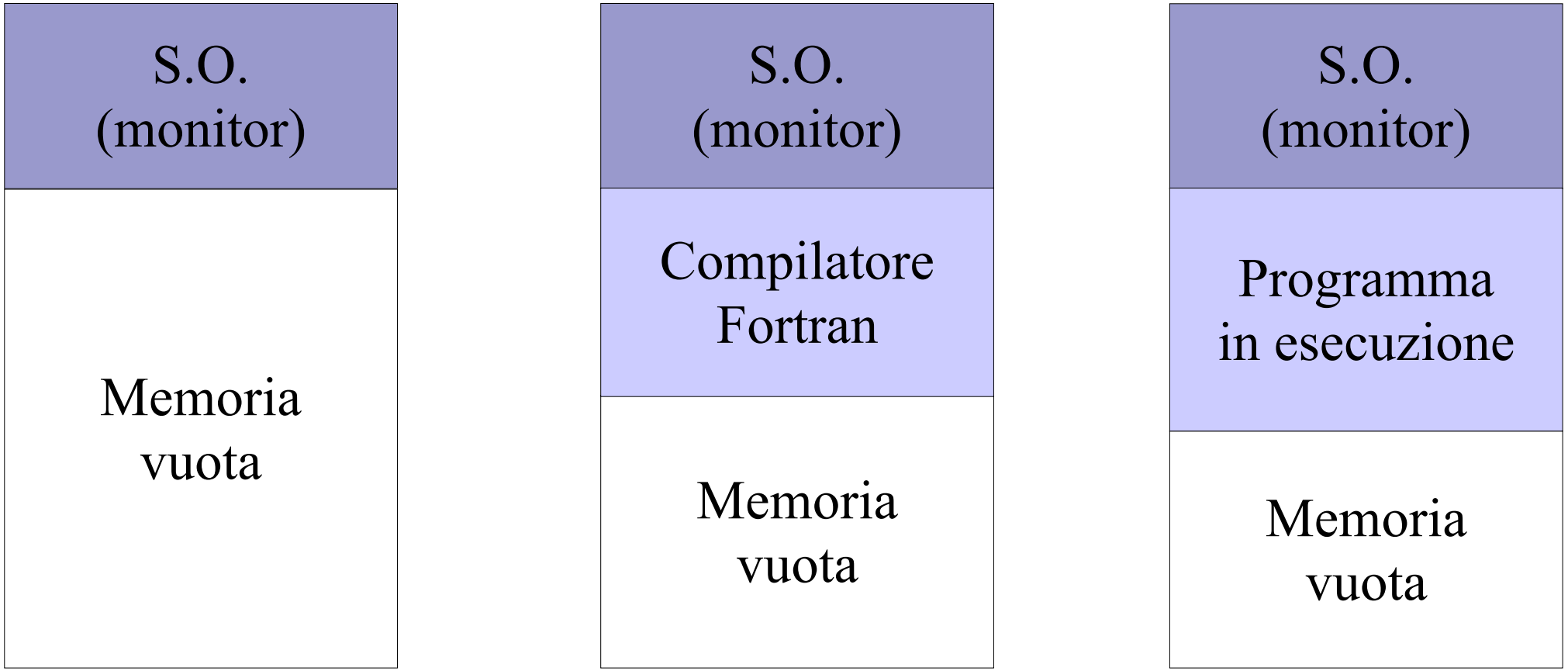
- **Job Control Language (JCL-FMS)**




(da una lezione di A.S. Tanenbaum)

# Generazione 2 (1955-1965)

- **Memory layout per un S.O. batch (versione semplificata)**




1. Stato iniziale

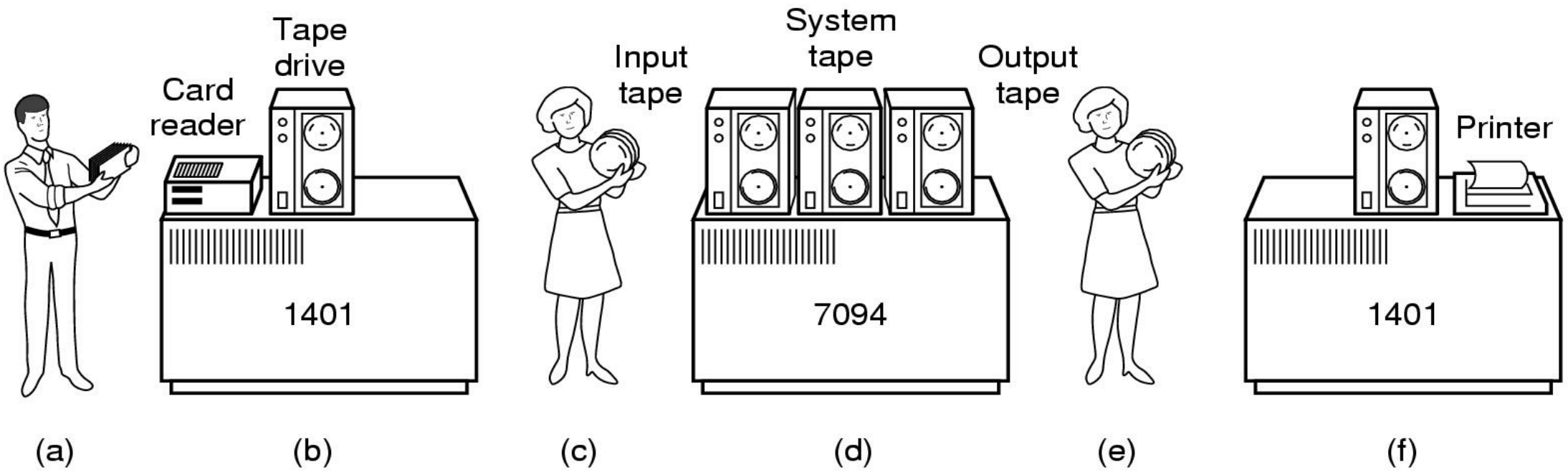
2. Dopo il caricamento del compilatore fortran

3. Dopo la compilazione

# Generazione 2 (1955-1965)

- **Principali problemi**
  - Molte risorse restavano inutilizzate:
    - durante le operazioni di lettura schede / stampa, durante il caricamento di un nuovo job, il processore restava inutilizzato
    - parte della memoria restava inutilizzata
  - Primo miglioramento (ma non una soluzione)
    - caricamento di numerosi job su nastro (off-line)
    - elaborazione (output su nastro)
    - stampa del nastro di output (off-line)

# Generazione 2 (1955-1965)




(da una lezione di A.S. Tanenbaum)

# Generazione 3 (1965-1980)

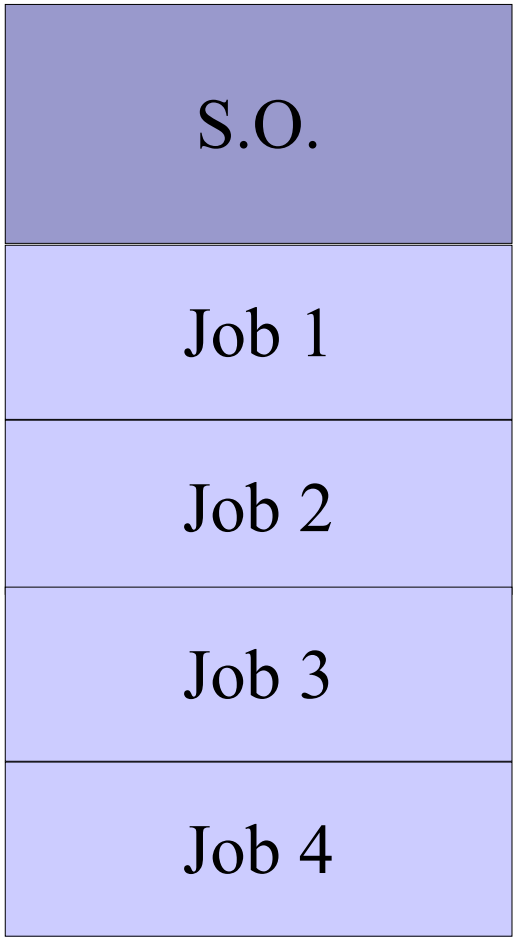
- **Come venivano costruiti?**
  - circuiti integrati
- **Come venivano usati?**
  - man mano sparisce la figura dell'operatore come "interfaccia" degli utenti verso la macchine
  - utente == operatore
- **Come venivano programmati?**
  - linguaggi ad "alto livello": C, shell scripting
  - editor testuali, editor grafici, compilatori
  - accesso al sistema da terminali
- **Quale sistemi operativi venivano usati?**
  - non più batch ma interattivi
  - multi-programmazione
  - time sharing

# Generazione 3 - Multiprogrammazione

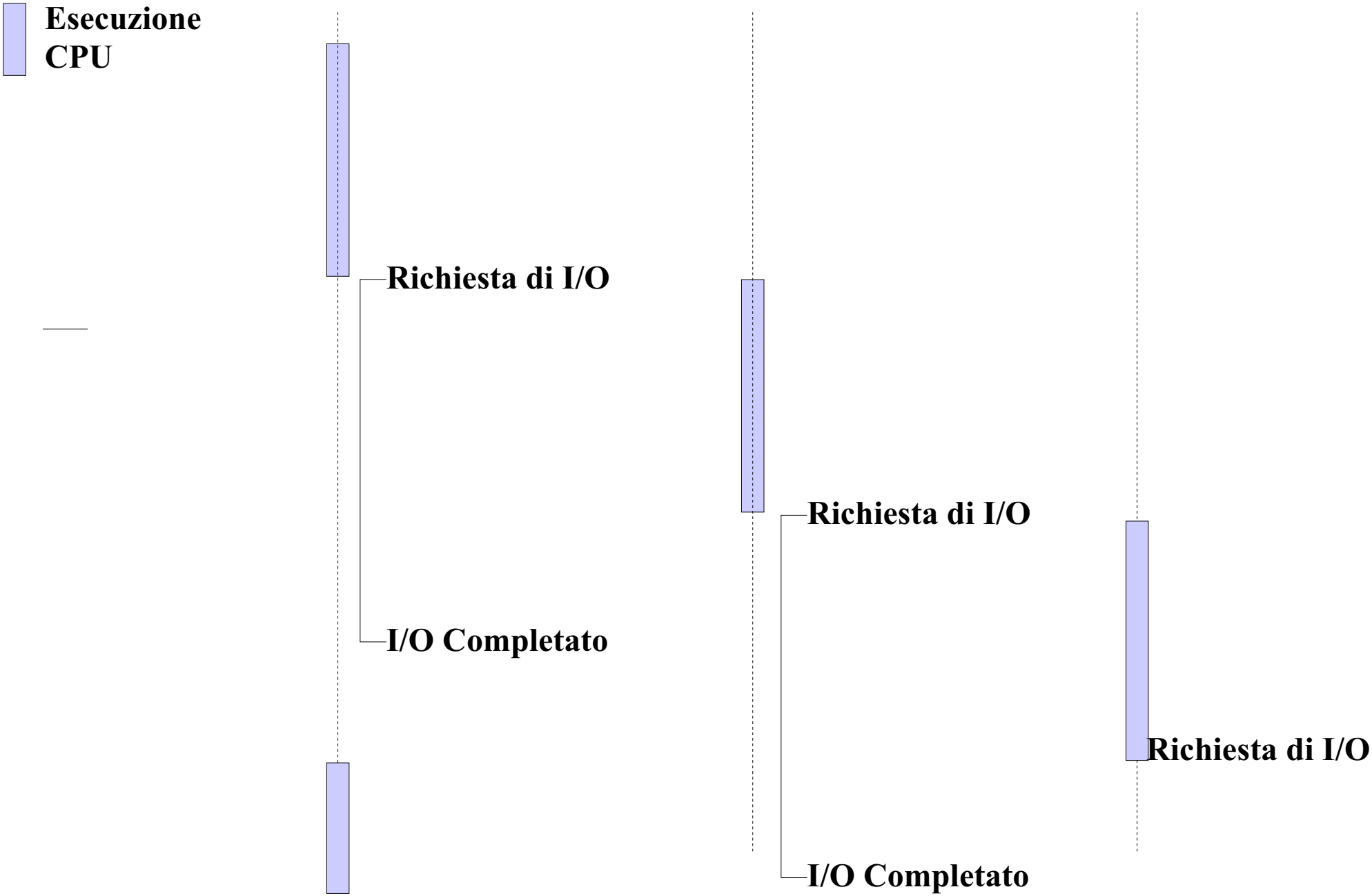
- **Definizione: *multiprogrammazione***
  - utilizzare il processore durante i periodi di I/O di un job per eseguire altri job
- **Vantaggi**
  - il processore non viene lasciato inattivo (*idle*) durante operazioni di I/O molto lunghe
  - la memoria viene utilizzata al meglio, caricando il maggior numero di job possibili
- **Nota**
  - per gestire la multiprogrammazione, il S.O. deve gestire un *pool* ("insieme") di job da eseguire, fra cui alternare il processore
  - *Simultaneous Peripheral Operation On-line (SPOOL)*
  - ad esempio: print spooler

# Generazione 3 - Multiprogrammazione

- **Caratteristiche tecniche:**
  - Più job contemporaneamente in  memoria
  - Una componente del S.O. detto scheduler si preoccupa di alternarli nell'uso della CPU
  - quando un job richiede un'operazione di I/O, la CPU viene assegnata ad un altro job.



# Generazione 3 - Multiprogrammazione



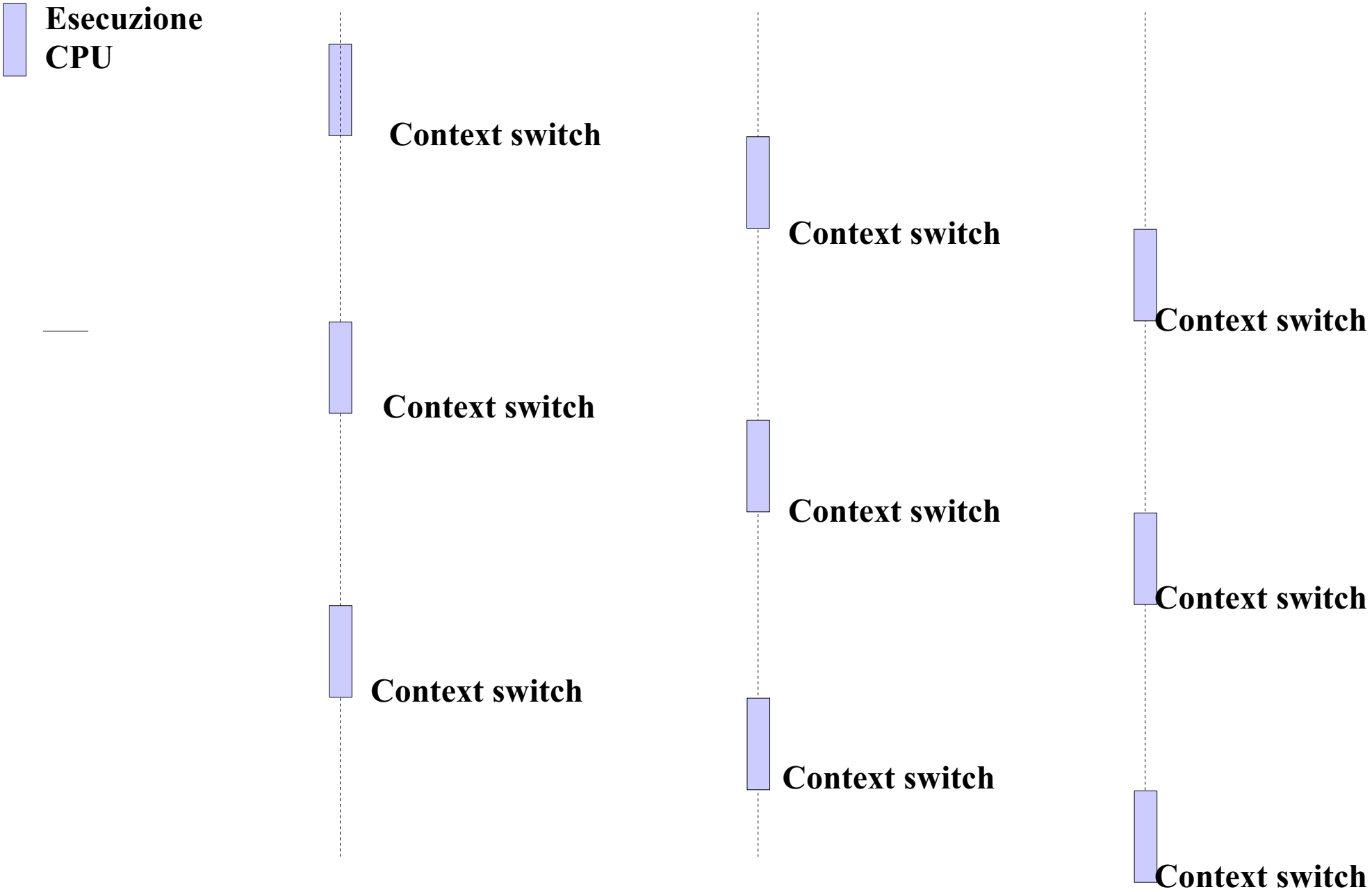
# Generazione 3 - Multiprogrammazione

- **S.O. multiprogrammati: quali caratteristiche?**
  - routine di I/O devono essere fornite dal S.O.
  - gestione della memoria
    - il sistema deve allocare la memoria per i job multipli presenti contemporaneamente
  - CPU scheduling
    - il sistema deve scegliere tra i diversi job pronti ad eseguire
  - allocazione delle risorse di I/O
    - Il sistema operativo deve essere in grado di allocare le risorse di I/O fra diversi processi

# Generazione 3 - Time-sharing

- **Definizione - Time sharing**
  - E' l'estensione logica della multiprogrammazione
  - L'esecuzione della CPU viene suddivisa in un certo numero di quanti temporali
  - Allo scadere di un quanto, il job corrente viene interrotto e l'esecuzione passa ad un altro job
    - anche in assenza di richieste di I/O
  - I passaggi (*context switch*) avvengono così frequentemente che più utenti possono interagire con i programmi in esecuzione

# Generazione 3 - Time-sharing



# Generazione 3 - Time-sharing

- **S.O. time-sharing: quali caratteristiche?**
  - *Gestione della memoria*
    - Il numero di programmi eseguiti dagli utenti può essere molto grande; si rende necessario la gestione della *memoria virtuale*
  - *CPU Scheduling*
    - Lo scheduling deve essere di tipo *preemptive* o *time-sliced*, ovvero sospendere periodicamente l'esecuzione di un programma a favore di un altro
  - *Meccanismi di protezione*
    - La presenza di più utenti rende necessari meccanismi di protezione (e.g. protezione nel file system, della memoria, etc.)

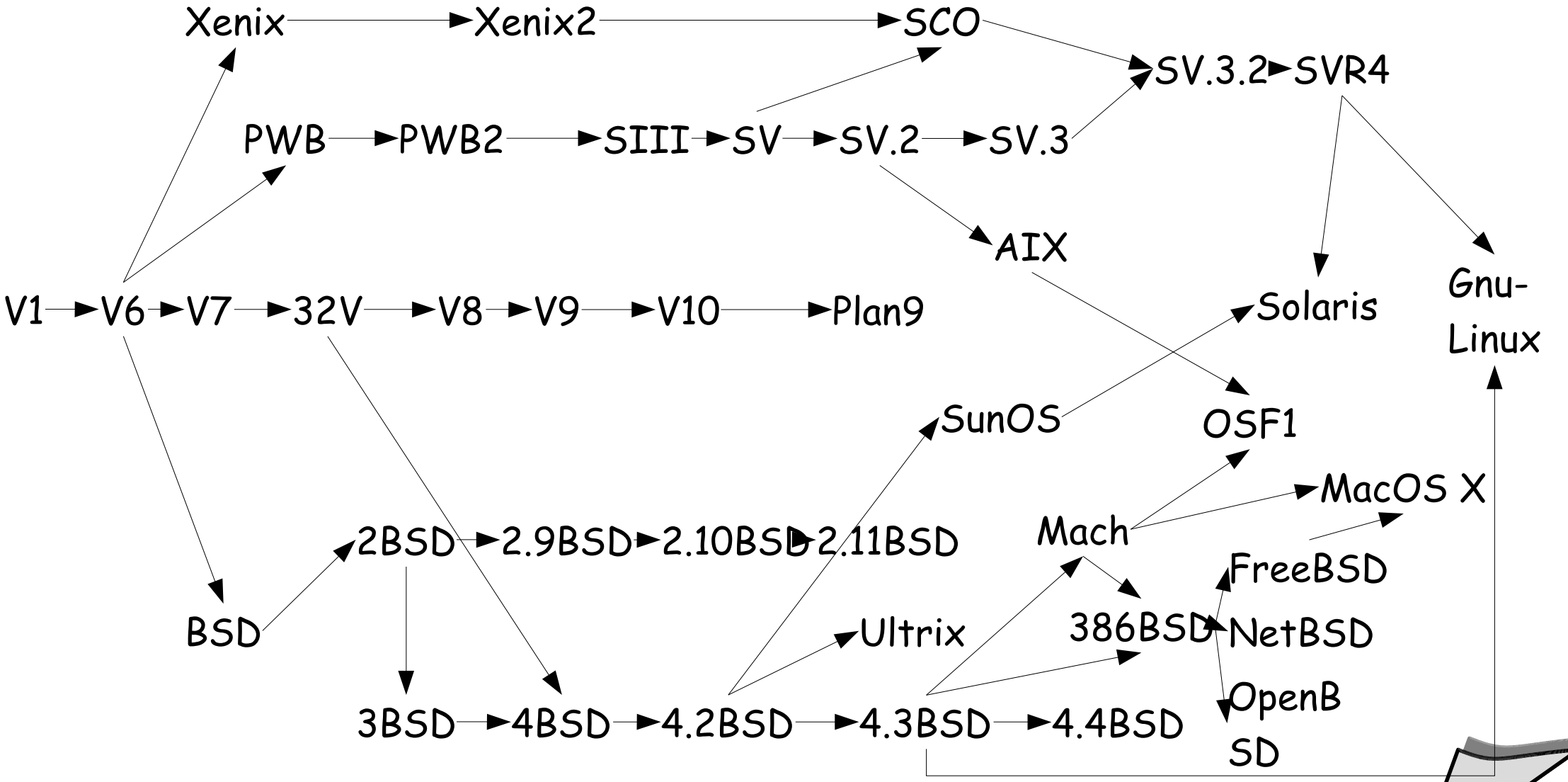
# Generazione 3 - Storia

- **Compatible Time-Sharing System (CTSS) (1962)**
  - introdusse il concetto di multiprogrammazione
  - introdusse il concetto di time-sharing
  - Codice sorgente rilasciato nel settembre 2004: http://www.piercefuller.com/library/ctss.html?id=ctss
- **Multics (1965)**
  - introduzione del concetto di processo
- **Unix (1970)**
  - derivato da CTSS e da Multics
  - sviluppato inizialmente ai Bell-labs su un PDP-7

# Unix – Un po' di storia

- **La storia di UNIX – in breve**
  - portato dal PDP-7 al PDP-11
    (1ª volta che un S.O viene utilizzato in due architetture diverse)
  - riscritto in linguaggio C per renderlo portabile
    (anche questa una 1ª volta, visto che i S.O. venivano scritti in assembly)
  - Inizialmente, veniva usato solo all'interno di Bell Labs
  - Nel 1974, viene pubblicato un articolo
    - licenze commerciali
    - licenze “libere” alle università
  - Due varianti
    - Xenix (Microsoft, poi diventato SCO...)
    - BSD (Berkeley Software Distribution, ora OpenBSD e FreeBSD)

# Unix – Un po' di storia



# Generazione 4 (1980- ) - Personal computer

- **Ancora una frase celebre**
  - Non c'è ragione per cui qualcuno possa volere un computer a casa sua
    (Ken Olson, fondatore di Digital, 1977)
- **Punti chiave**
  - I personal computer sono dedicati a singoli utenti
  - L'obiettivo primario diventa la facilità d'uso
  - Essendo dedicati a singoli utenti, i sistemi operativi per PC sono in generale più semplici
  - Tecnologie a finestre
  - Tuttavia, tecnologie sviluppate per sistemi operativi più complessi possono essere adottate

# Sistemi paralleli

- **Definizione - *Sistema parallelo***
  - un singolo elaboratore che possiede più unità di elaborazione
- **Si dicono anche sistemi *tightly* coupled**
  - alcune risorse contenute nell'elaboratore possono essere condivise
  - esempio: memoria
  - la comunicazione avviene tramite memoria condivisa o canali di comunicazione dedicati
- **Vantaggi dei sistemi paralleli**
  - incremento delle prestazioni
  - incremento dell'affidabilità (*graceful degradation*)

# Sistemi paralleli

- **Tassonomia basata sulla struttura**
  - *SIMD - Single Instruction, Multiple Data*
    - Le CPU eseguono all'unisono lo stesso programma su dati diversi
  - *MIMD - Multiple Instruction, Multiple Data*
    - Le CPU eseguono programmi differenti su dati differenti
- **Tassonomia basata sulla dimensione**
  - A seconda del numero (e della potenza) dei processori si suddividono in
    - *sistemi a basso parallelismo*
      pochi processori in genere molto potenti
    - *sistemi massicciamente paralleli*
      gran numero di processori, che possono avere anche potenza non elevata

# Sistemi paralleli

- **Symmetric multiprocessing (SMP)**
  - Ogni processore esegue una copia identica del sistema operativo
  - Processi diversi possono essere eseguiti contemporaneamente
  - Molti sistemi operativi moderni supportano SMP
- **Asymmetric multiprocessing**
  - Ogni processore è assegnato ad un compito specifico; un processore master gestisce l'allocazione del lavoro ai processori slave
  - Più comune in sistemi estremamente grandi

# Sistemi distribuiti

- **Definizione - *Sistema distribuito***
  - Sono sistemi composti da più elaboratori indipendenti (con proprie risorse e proprio sistema operativo)
- **Si dicono anche sistemi *loosely* coupled**
  - Ogni processore possiede la propria memoria locale
  - I processori sono collegati tramite linee di comunicazione (rete, linee telefoniche, linee wireless, etc)
- **Vantaggi dei sistemi distribuiti**
  - Condivisione di risorse
  - Suddivisione di carico, incremento delle prestazioni
  - Affidabilità
  - Possibilità di comunicare

# Sistemi distribuiti

- **Sistemi operativi di rete**
  - forniscono condivisione di file
  - forniscono la possibilità di comunicare
  - ogni computer opera indipendentemente dagli altri
- **Sistemi operativi distribuiti**
  - minore autonomia tra i computer
  - dà l'impressione che un singolo sistema operativo stia controllando la rete

# Sistemi real-time

- **Definizione: *sistemi real-time***
  - Sono i sistemi per i quali la correttezza del risultato non dipende solamente dal suo valore ma anche dall'istante nel quale il risultato viene prodotto

# Sistemi real-time

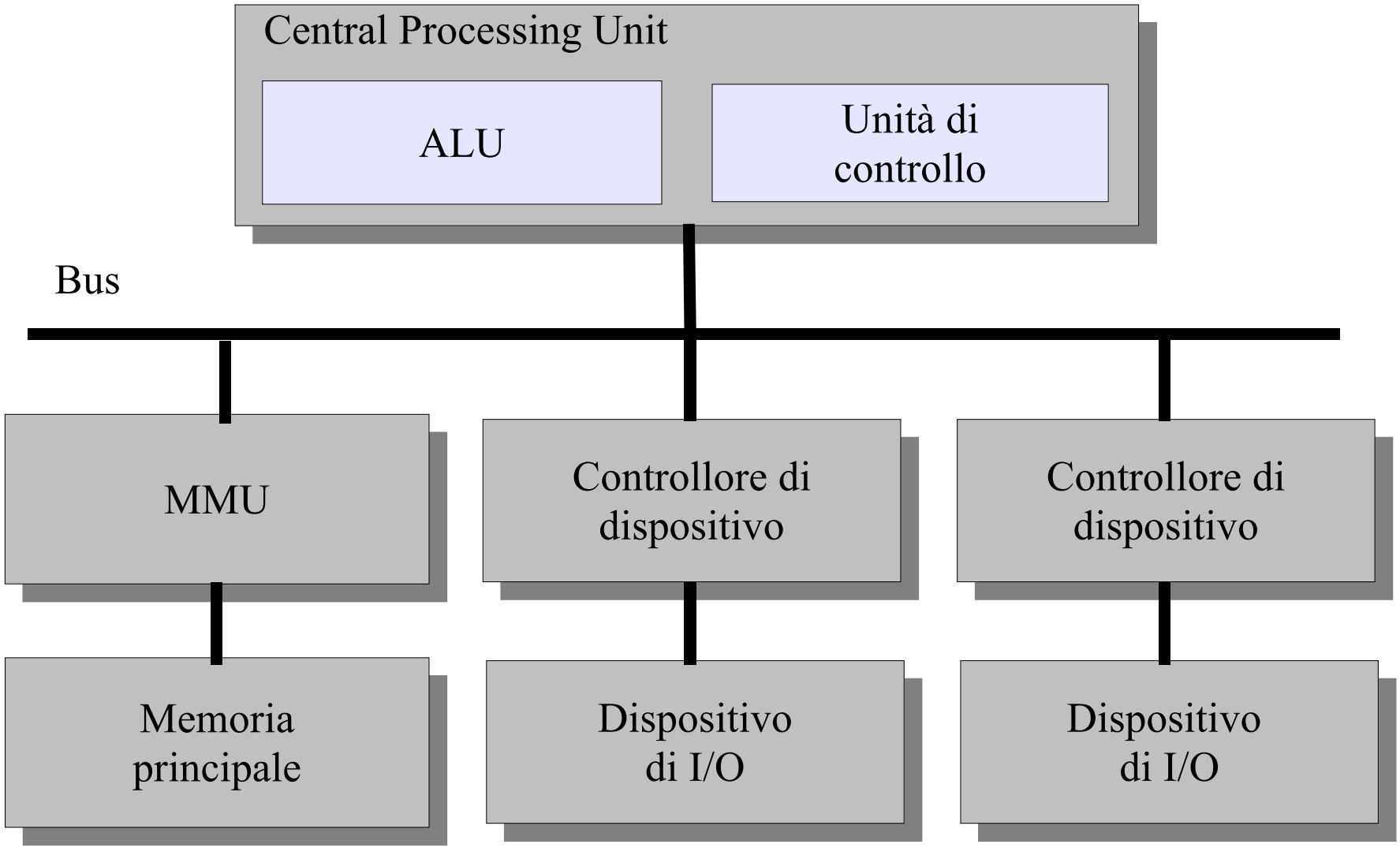
- **I sistemi real-time si dividono in:**
  - *hard real-time*:
    - se il mancato rispetto dei vincoli temporali può avere effetti catastrofici
    - e.g. controllo assetto velivoli, controllo centrali nucleari, apparecchiature per terapia intensiva
  - *soft real-time*:
    - se si hanno solamente disagi o disservizi
    - e.g. programmi interattivi
- **N.B.**
  - real-time non significa necessariamente esecuzione veloce

# Lo "zoo" dei sistemi operativi

- Mainframe operating systems
- Server operating systems
- Multiprocessor operating systems
- Personal computer operating systems
- Network operating systems
- Distributed operating systems
- Real-time operating systems
- Embedded operating systems
- Smart card operating systems

# 3. Concetti base di architetture degli elaboratori (ripasso)

# Architettura di Von Neumann



# Architettura di Von Neumann

- **L'anno scorso avete visto nei dettagli:**
  - architettura dei processori
  - concetti base relativi alla memoria
  - linguaggio assembly
- **Nei prossimi lucidi vedremo alcuni concetti relativi a:**
  - gestione della comunicazione tra processore  e dispositivi di I/O
  - concetto di interrupt
  - gerarchie di memoria

# Interrupt

- **Definizione:**
  - Un meccanismo che permette l'interruzione del normale ciclo di esecuzione della CPU
- **Caratteristiche**
  - introdotti per aumentare l'efficienza di un sistema di calcolo
  - permettono ad un S.O. di "intervenire" durante l'esecuzione di un processo utente, allo scopo di gestire efficacemente le risorse del calcolatore
    - processore, memoria, dispositivi di I/O
  - possono essere sia hardware che software
  - possono essere mascherati (ritardati) se la CPU sta svolgendo compiti non interrompibili

# Interrupt vs Trap

- **Interrupt Hardware**
  - Eventi hardware asincroni, non causati dal processo in esecuzione
  - Esempi:
    - *dispositivi di I/O*
      (per notifica di eventi quali il completamento di una operazione di I/O)
    - *clock*
      (scadenza del quanto di tempo)
- **Interrupt Software (Trap)**
  - Causato dal programma
  - Esempi
    - eventi eccezionali come divisione per 0 o problemi di indirizzamento
    - richiesta di servizi di sistema (system call)

# Gestione Interrupt - Panoramica

- **Cosa succede in seguito ad un interrupt**

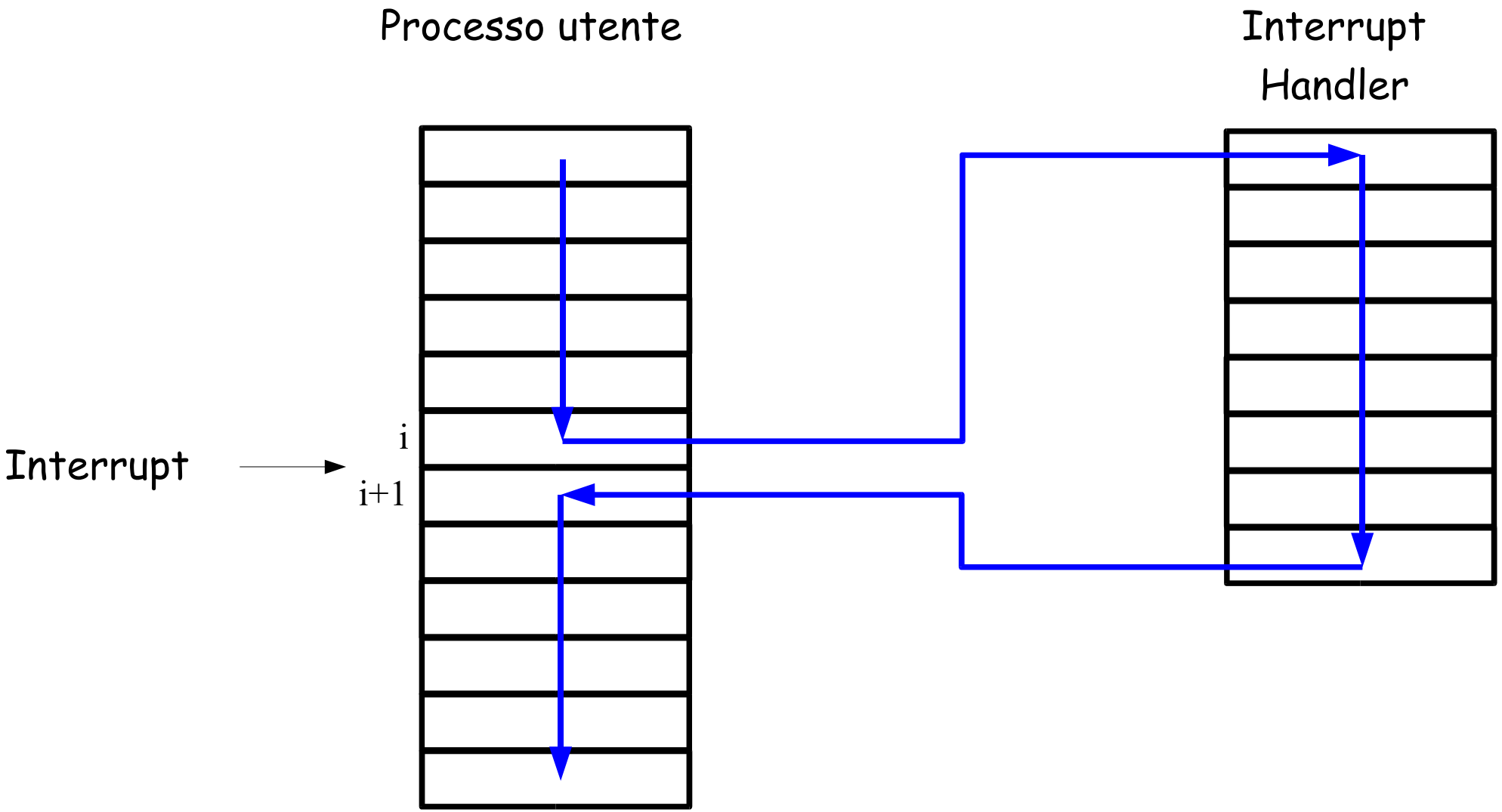
Hardware

- Un segnale "interrupt request" viene spedito al processore
- Il processore
  - sospende le operazioni del processo corrente
  - salta ad un particolare indirizzo di memoria contenente la routine di gestione dell'interrupt (*interrupt handler*)

Software

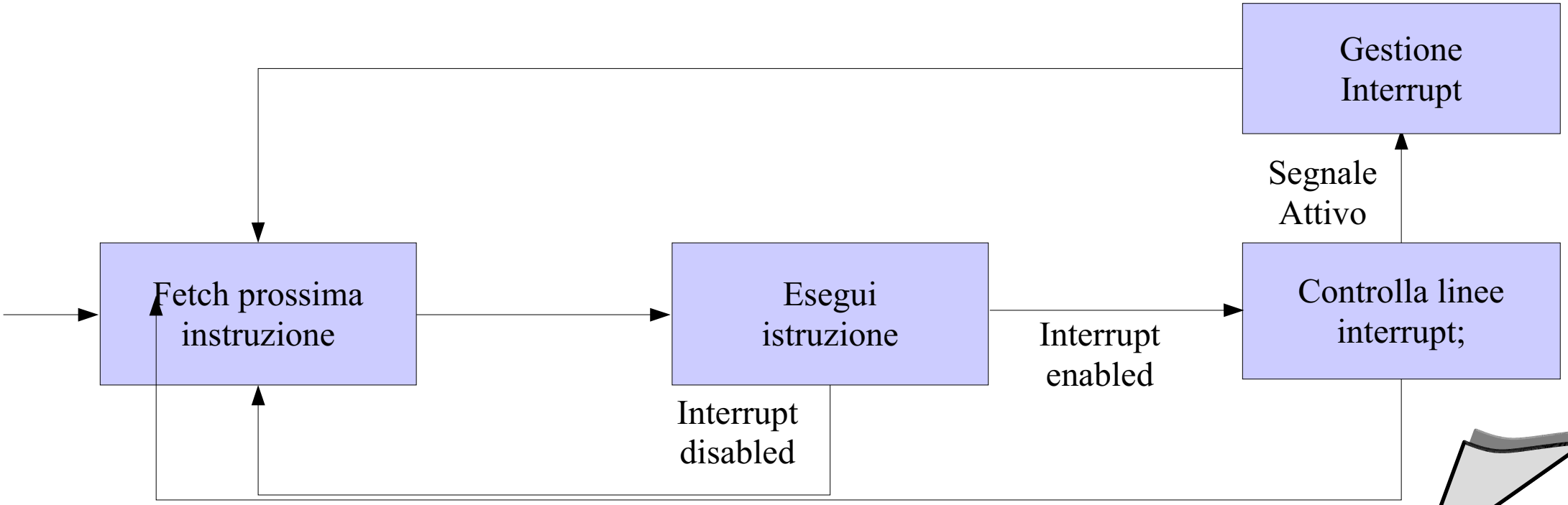
- L'interrupt handler
  - gestisce nel modo opportuno l'interrupt
  - ritorna il controllo al processo interrotto (o a un altro processo, nel caso di scheduling)
- Il processore riprende l'esecuzione del processo interrotto come se nulla fosse successo

# Interrupt



# Gestione Interrupt - Dettagli

1. Un segnale di interrupt request viene spedito alla CPU
2. La CPU finisce l'esecuzione dell'istruzione corrente
3. La CPU verifica la presenza di un segnale di interrupt, e in caso affermativo spedisce un segnale di conferma al device che ha generato l'interrupt



# Gestione Interrupt - Dettagli

4. **Preparazione al trasferimento di controllo dal programma all'interrupt handler**
   - metodo 1: salvataggio dei registri "critici"
     - informazione minima richiesta: PC + registro di stato
   - metodo 2: scambio di stato
     - fotografia dello stato del processore
5. **Selezione dell'interrupt handler appropriato**
   - a seconda dell'architettura, vi può essere un singolo interrupt handler, uno per ogni tipo di interrupt o uno per dispositivo
   - la selezione avviene tramite l'*interrupt vector*
6. **Caricamento del PC con l'indirizzo iniziale dell'interrupt handler assegnato**

# Gestione Interrupt - Dettagli

- **Nota:**
  - tutte le operazioni compiute fino a qui sono operazioni hardware
  - la modifica del PC corrisponde ad un salto al codice dell'interrupt handler
  - a questo punto:
    - il ciclo fetch-execute viene ripreso
    - il controllo è passato in mano all'interrupt handler

# Gestione Interrupt - Dettagli

7. **Salvataggio dello stato del processore**
   - salvataggio delle informazioni critiche non salvate automaticamente dai meccanismi hardware di gestione interrupt
8. **Gestione dell'interrupt**
   - lettura delle informazioni di controllo proveniente dal dispositivo
   - eventualmente, spedizione di ulteriori informazioni al dispositivo stesso
9. **Ripristino dello stato del processore**
   - l'operazione inversa della numero 7
10. **Ritorno del controllo al processo in esecuzione (o ad un altro processo, se necessario)**

# Sistemi operativi "Interrupt Driven"

- **I S.O. moderni sono detti "Interrupt Driven"**
  - gran parte del nucleo di un S.O. viene eseguito come interrupt handler
  - sono gli interrupt (o i trap) che guidano l'avvicendamento dei processi

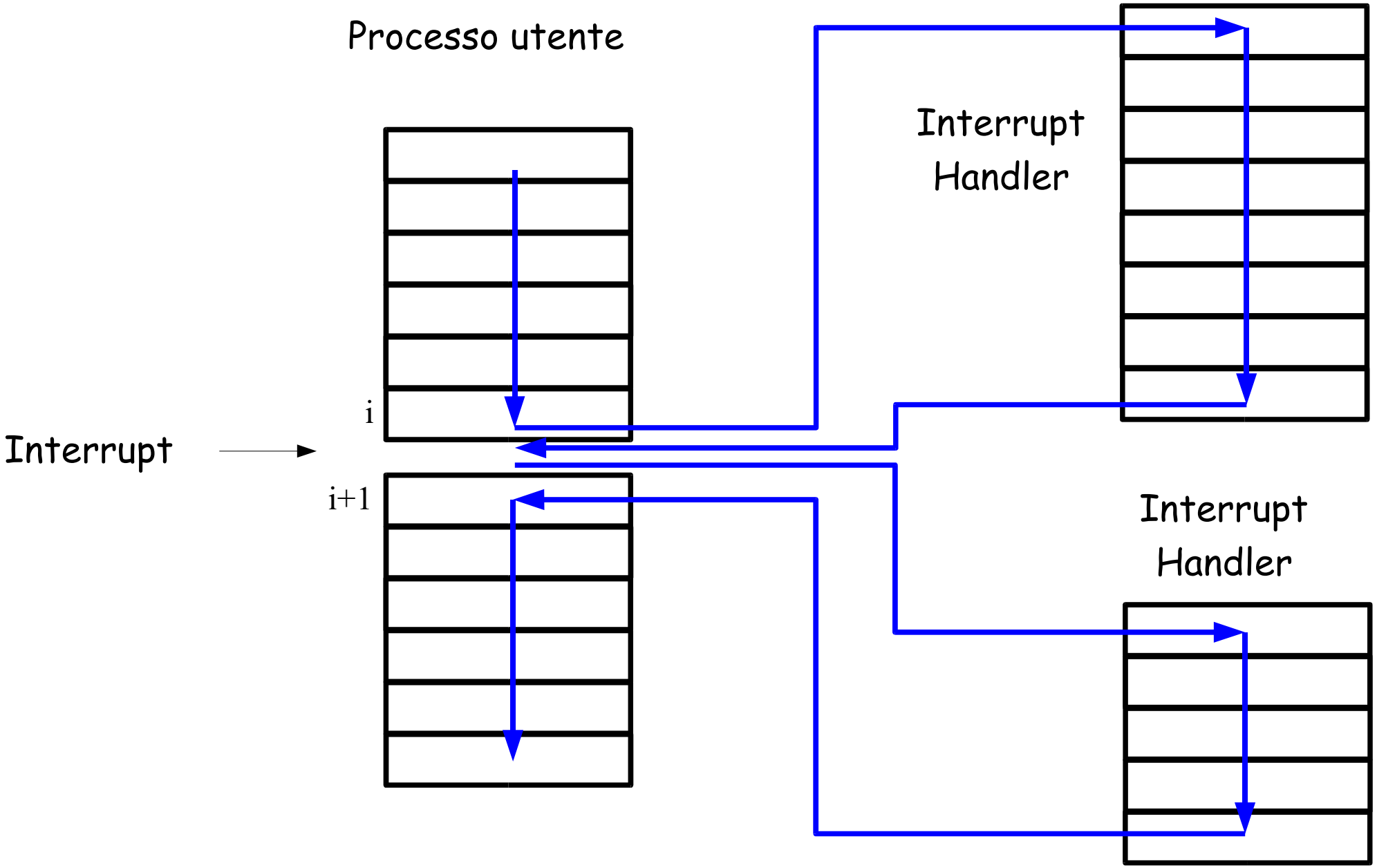
# Interrupt Multipli

- **La discussione precedente prevedeva la presenza di un singolo interrupt**
- **Esiste la possibilità che avvengano interrupt multipli**
  - ad esempio, originati da dispositivi diversi
  - un interrupt può avvenire durante la gestione di un interrupt precedente
- **Due approcci possibili:**
  - disabilitazione degli interrupt
  - interrupt annidati

# Interrupt Multipli - Disabilitazione Interrupt

- **Disabilitazione degli interrupt**
  - durante l'esecuzione di un interrupt handler
    - ulteriori segnali di interrupt vengono ignorati
    - i segnali corrispondenti restano pendenti
  - gli interrupt vengono riabilitati prima di riattivare il processo interrotto
  - il processore verifica quindi se vi sono ulteriori interrupt, e in caso attiva l'interrupt handler corrispondente
- **Vantaggi e svantaggi**
  - approccio semplice; interrupt gestiti in modo sequenziale
  - non tiene conto di gestioni "time-critical"

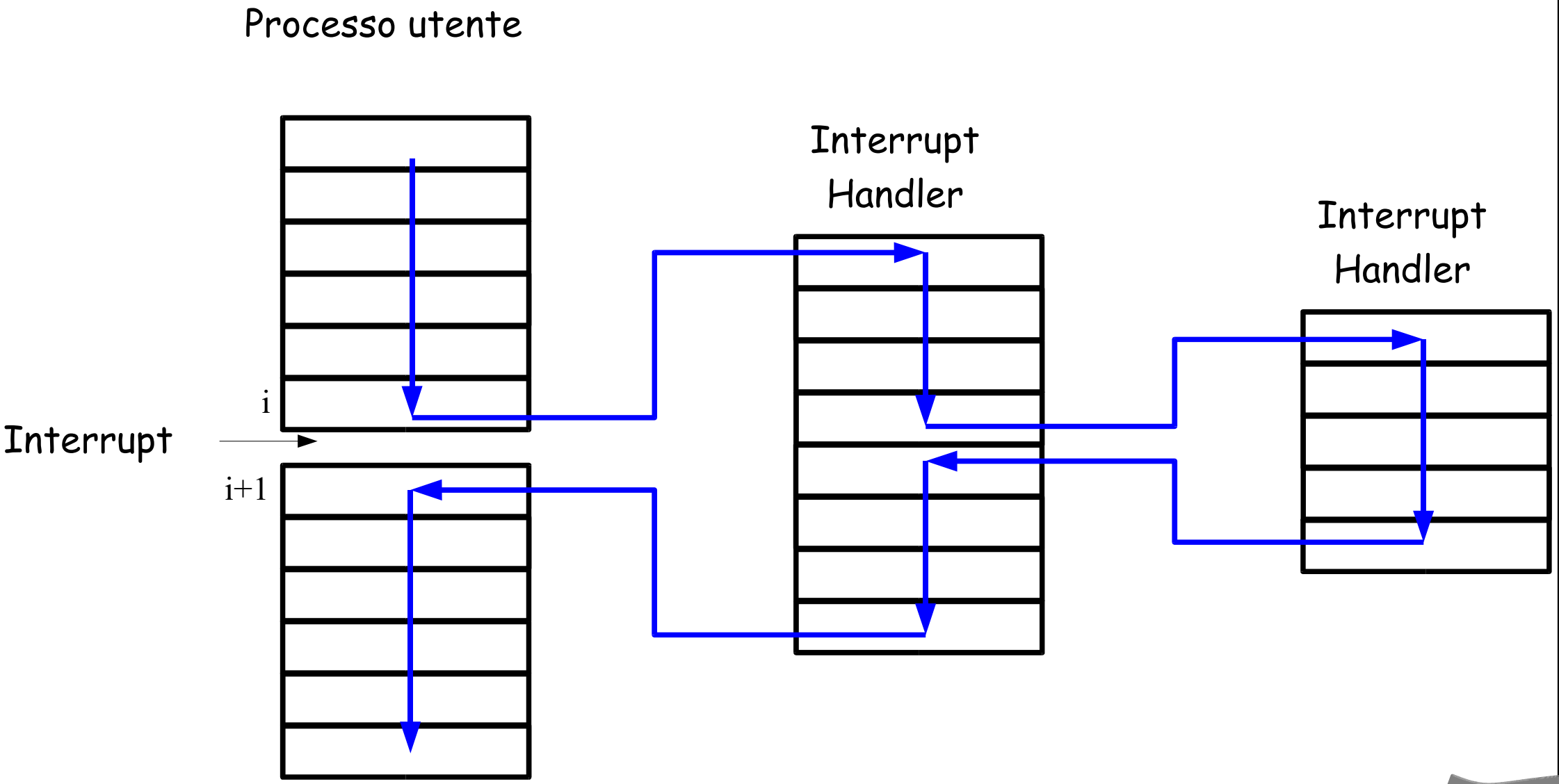
# Interrupt Multipli - Disabilitazione Interrupt



# Interrupt Multipli - Interrupt Annidati

- **Interrupt annidati**
  - è possibile definire priorità diverse per gli interrupt
  - un interrupt di priorità inferiore può essere interrotto da un interrupt di priorità superiore
  - è necessario prevedere un meccanismo di salvataggio e ripristino dell'esecuzione adeguato
- **Vantaggi e svantaggi**
  - dispositivi veloci possono essere serviti prima (es. schede di rete)
  - approccio più complesso

# Interrupt Multipli - Interrupt Annidati



# Comunicazione fra processore e dispositivi di I/O

- **Comunicazione tra processore e dispositivi di I/O**
  - il controllore governa il dialogo con il dispositivo fisico
  - il controllo della politica di accesso al dispositivo è a carico del dispositivo stesso
- **Esempio**
  - il controller di un disco accetta una richiesta per volta
  - l'accodamento delle richieste in attesa è a carico del S.O.
- **Tre modalità possibili:**
  - *Programmed I/O*
  - *Interrupt-Driven I/O*
  - *Direct Memory Access (DMA)*

# Programmed I/O (obsoleto)

- **Operazione di input**
  - la CPU carica (tramite il bus) i parametri della richiesta di input in appositi registri del controller (registri comando)
  - il dispositivo
    - esegue la richiesta
    - il risultato dell'operazione viene memorizzato in un apposito buffer locale sul controller
    - il completamento dell'operazione viene segnalato attraverso appositi registri di status
  - *il S.O. attende* (busy waiting) *che il comando sia completato verificando periodicamente il contenuto del registro di stato*
  - *infine, la CPU copia i dati dal buffer locale del controller alla memoria*

# Interrupt-Driven I/O

- **Operazione di input**
  - la CPU carica (tramite il bus) i parametri della richiesta di input in appositi registri del controller (registri comando)
  - *il S.O. sospende l'esecuzione del processo che ha eseguito l'operazione di input ed esegue un altro processo*
  - il dispositivo
    - esegue la richiesta
    - il risultato dell'operazione viene memorizzato in un apposito buffer locale sul controller
    - il completamento dell'operazione viene segnalato attraverso interrupt
  - *al ricevimento dell'interrupt, la CPU copia i dati dal buffer locale del controller alla memoria*

# Programmed I/O e Interrupt-Driven I/O

- **Nel caso di operazioni di output**
  - il procedimento è similare:
    - i dati vengono copiati dalla memoria ai buffer locali
    - questa operazione viene eseguita prima di caricare i parametri della richiesta nei registri di comando dei dispositivi
- **Svantaggi degli approcci precedenti**
  - il processore spreca parte del suo tempo nella gestione del trasferimento dei dati
  - la velocità di trasferimento è limitata dalla velocità con cui il processore riesce a gestire il servizio

# Direct Memory Access (DMA)

- **Il S.O.**
  - *attiva l'operazione di I/O specificando l'indirizzo in memoria di destinazione (Input) o di provenienza (Output) dei dati*
  - *l'interrupt specifica solamente la conclusione dell'operazione di I/O*
- **Vantaggi e svantaggi**
  - c'è contesa nell'accesso al bus
  - device driver più semplici
  - efficace perché la CPU non accede al bus ad ogni ciclo di clock

# Memoria

- **Memoria centrale (RAM)**
  - assieme ai registri, l'unico spazio di memorizzazione che può essere acceduto *direttamente* dal processore
  - accesso tramite istruzioni LOAD/STORE
  - volatile
- **Memoria ROM**

# Memory Mapped I/O

- **Un dispositivo è completamente indirizzabile tramite bus**
  - i registri di dispositivo vengono mappati su un insieme di indirizzi di memoria
  - una scrittura su questi indirizzi causa il trasferimento di dati verso il dispositivo
- **Esempio:**
  - video grafico nei PC
- **Vantaggi e svantaggi**
  - gestione molto semplice e lineare
  - necessita di tecniche di *polling*

# Dischi

- **Caratteristiche**
  - dispositivi che consentono la memorizzazione non volatile dei dati
  - accesso diretto (random, i.e. non sequenziale)
  - per individuare un dato sul disco (dal punto di vista fisico) occorre indirizzarlo in termini di cilindro, testina, settore

# Dischi

- **Operazioni gestite dal controller**
  - READ (head, sector)
  - WRITE(head, sector)
  - SEEK(cylinder)
- **L'operazione di seek**
  - corrisponde allo spostamento fisico del pettine di testine da un cilindro ad un altro ed è normalmente la più costosa
- **L'operazione di read e write**
  - prevedono l'attesa che il disco ruoti fino a quando il settore richiesto raggiunge la testina

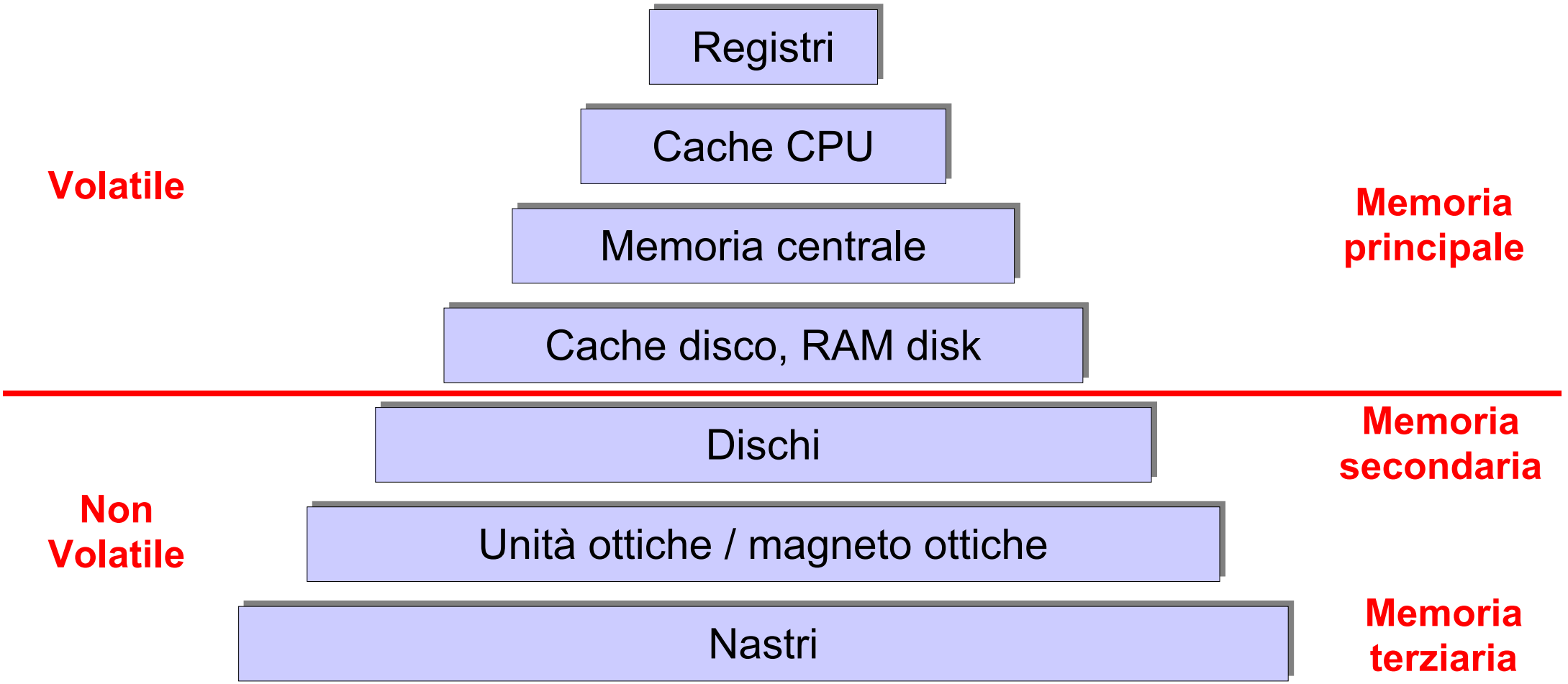
# Nastri

- **Caratteristiche**
  - dispositivi per la memorizzazione non volatile dei dati
  - accesso sequenziale (non diretto)
  - un nastro può contenere più file separati da gap, ma la ricerca del file viene sempre effettuata con scansione sequenziale

# Gerarchia di memoria

- **Trade off**
  - Quantità
  - Velocità
  - Costo
- **Limitazioni**
  - tempo di accesso più veloce, costo maggiore
  - maggiore capacità, costo minore (per bit)
  - maggiore capacità, tempo di accesso maggiore
- **Soluzione:**
  - utilizzare una gerarchia di memoria

# Gerarchia di memoria



# Cache

- **Un meccanismo di caching**
  - consiste nel memorizzare *parzialmente* i dati di una memoria in una seconda più costosa ma più efficiente
  - se il numero di occorrenze in cui il dato viene trovato nella cache (memoria veloce) è statisticamente rilevante rispetto al numero totale degli accessi, la cache fornisce un notevole aumento di prestazione
- **E' un concetto che si applica a diversi livelli:**
  - cache della memoria principale (DRAM) tramite memoria bipolare
  - cache di disco in memoria
  - cache di file system remoti tramite file system locali

# Cache

- **Meccanismi di caching**
  - hardware
    - ad es. cache CPU; *politiche non modificabili dal S.O.*
  - software
    - ad es. cache disco; *politiche sotto controllo del S.O.*
- **Problemi da considerare nel S.O.**
  - algoritmo di replacement
    - la cache ha dimensione limitata; bisogna scegliere un algoritmo che garantisca il maggior numero di accessi in cache
  - coerenza
    - gli stessi dati possono apparire a diversi livelli della struttura di memoria

# Protezione Hardware

- **I sistemi multiprogrammati e multiutente richiedono la presenza di *meccanismi di protezione***
  - bisogna evitare che processi concorrenti generino interferenze non previste...

    ma soprattutto:

  - *bisogna evitare che processi utente interferiscano con il sistema operativo*
- **Riflessione**
  - i meccanismi di protezione possono essere realizzati totalmente in software, oppure abbiamo bisogno di meccanismi hardware dedicati?

# Protezione HW: Modo utente / Modo kernel

- **Modalità kernel / supervisore / privilegiata:**
  - i processi in questa modalità hanno accesso a tutte le istruzioni, incluse quelle *privilegiate*, che permettono di gestire totalmente il sistema
- **Modalità utente**
  - i processi in modalità utente non hanno accesso alle istruzioni privilegiate
- **"Mode bit" nello status register per distinguere fra modalità utente e modalità supervisore**
- **Esempio:**
  - le istruzioni per disabilitare gli interrupt è privilegiata

# Protezione HW: Modo utente / Modo kernel

- **Come funziona**
  - alla partenza, il processore è in modalità kernel
  - viene caricato il sistema operativo (*bootstrap*) e si inizia ad eseguirlo
  - quando passa il controllo ad un processo utente, il S.O. cambia il valore del mode bit e il processore passa in modalità utente
  - tutte le volte che avviene un interrupt, l'hardware passa da modalità utente a modalità kernel

# Protezione HW: Protezione I/O

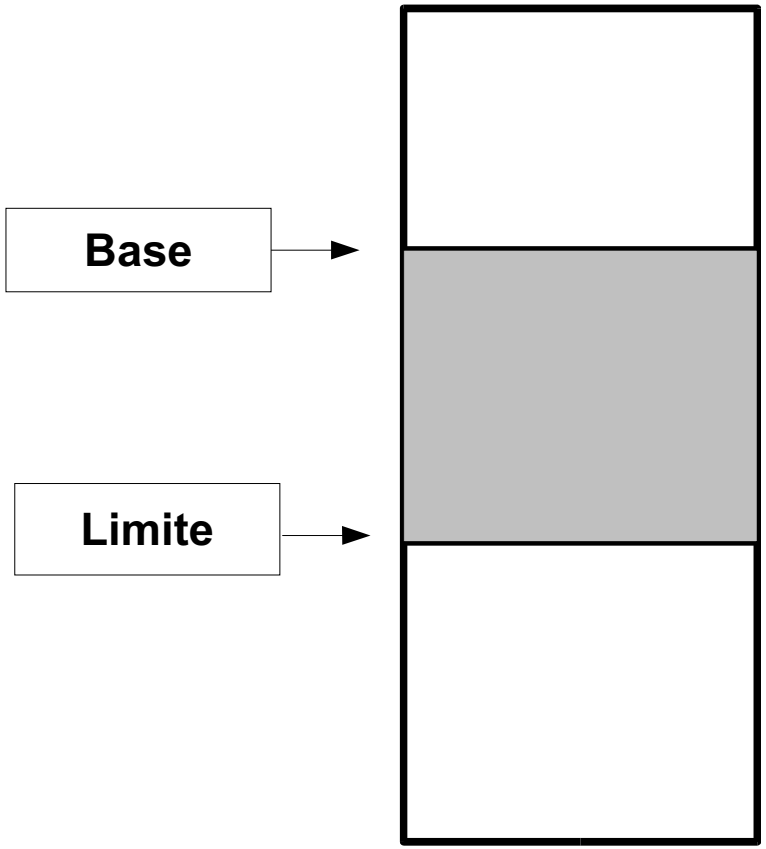
- **Le istruzioni di I/O devono essere considerate privilegiate**
  - il S.O. dovrà fornire agli utenti primitive e servizi per accedere all'I/O
  - tutte le richieste di I/O passano attraverso codice del S.O. e possono essere controllate preventivamente
- **Esempio:**
  - accesso al dispositivo di memoria secondaria che ospita un file system
  - vogliamo evitare che un qualunque processo possa accedere al dispositivo modificando (o corrompendo) il file system stesso

# Protezione HW: Protezione Memoria

- **La protezione non è completa se non proteggiamo anche la memoria**
- **Altrimenti, i processi utente potrebbero:**
  - modificare il codice o i dati di altri processi utenti
  - modificare il codice o i dati del sistema operativo
  - modificare l'interrupt vector, inserendo i propri gestori degli interrupt
- **La protezione avviene tramite la Memory Management Unit (MMU)**

# Protezione HW: MMU

- **Registro base + registro limite**
  - ogni indirizzo generato dal processore viene confrontato con due registri, detti base e limite. Se non incluso in questo range, l'indirizzo non è valido e genera un'eccezione




- **Traduzione indirizzi logici in indirizzi fisici**
  - ogni indirizzo generato dal processore corrisponde ad un indirizzo logico
  - l'indirizzo logico viene trasformato in un indirizzo fisico a tempo di esecuzione dal meccanismo di MMU
  - un indirizzo viene protetto se non può mai essere generato dal meccanismo di traduzione

# Protezione Hardware

- **Riflessione - continua**
  - i meccanismi visti proteggono
    - il sistema operativo dai programmi utenti
    - i programmi utenti dagli altri programmi utenti
  - c'è chi vuole fare di più...
    - protezione del sistema da qualunque tentativo di manomissione (compreso il proprietario del sistema...)
- **Digital Right Management**
  - un'iniziativa Microsoft, Intel et al. con lo scopo di creare un "trusted environment", in cui non sia possibile eseguire nessun programma non autorizzato
    - mai più virus.... (mah?)
    - mai più open-source software... (mah!)

# Protezione HW - System call

- **Problema**
  - poiché le istruzioni di I/O sono privilegiate, possono essere eseguite unicamente dal S.O.
  - com'è possibile per i processi utenti eseguire operazioni di I/O?
- **Soluzione**
  - i processi utenti devono fare richieste esplicite di I/O al S.O.
  - meccanismo delle *system call*, ovvero trap generate da istruzioni specifiche

# Protezione HW - System call

**Codice utente**

```
li $a0, 10
li $v0, 1
syscall
```

```
... altro
codice...
```

**Interrupt handler**

```
...salvataggio registri ...
...gestione interrupt...
...operazioni di I/O...
...ripristino registri...
rfe/eret
// return from exception
```

**Interrupt vector**