# Concetti introduttivi

# Obiettivi

- Descrivere l'organizzazione generale di un sistema di elaborazione
- Illustrare i principali compiti di un SO relativi alla gestione delle risorse
- Descrivere i concetti di multiprogrammazione e multitasking
- Identificare i servizi forniti da un SO
- Illustrare le varie interfacce di un SO
- Descrivere gli scopi principali del progetto e dell'implementazione di un SO
- Confrontare gli approcci utilizzati per la strutturazione interna dei SO

# Concetti introduttivi

- Cosa fa un SO
- Organizzazione di un sistema di elaborazione
- Gestione delle risorse
- Virtualizzazione
- Sicurezza e protezione
- Servizi di un SO
- Interfacce utente
- System call & API
- Servizi di sistema
- Progetto e implementazione di un SO
- Struttura di un SO

# Concetti introduttivi

- Cosa fa un SO
- Organizzazione di un sistema di elaborazione
- Gestione delle risorse
- Virtualizzazione
- Sicurezza e protezione
- Servizi di un SO
- Interfacce utente
- System call & API
- Servizi di sistema
- Progetto e implementazione di un SO
- Struttura di un SO

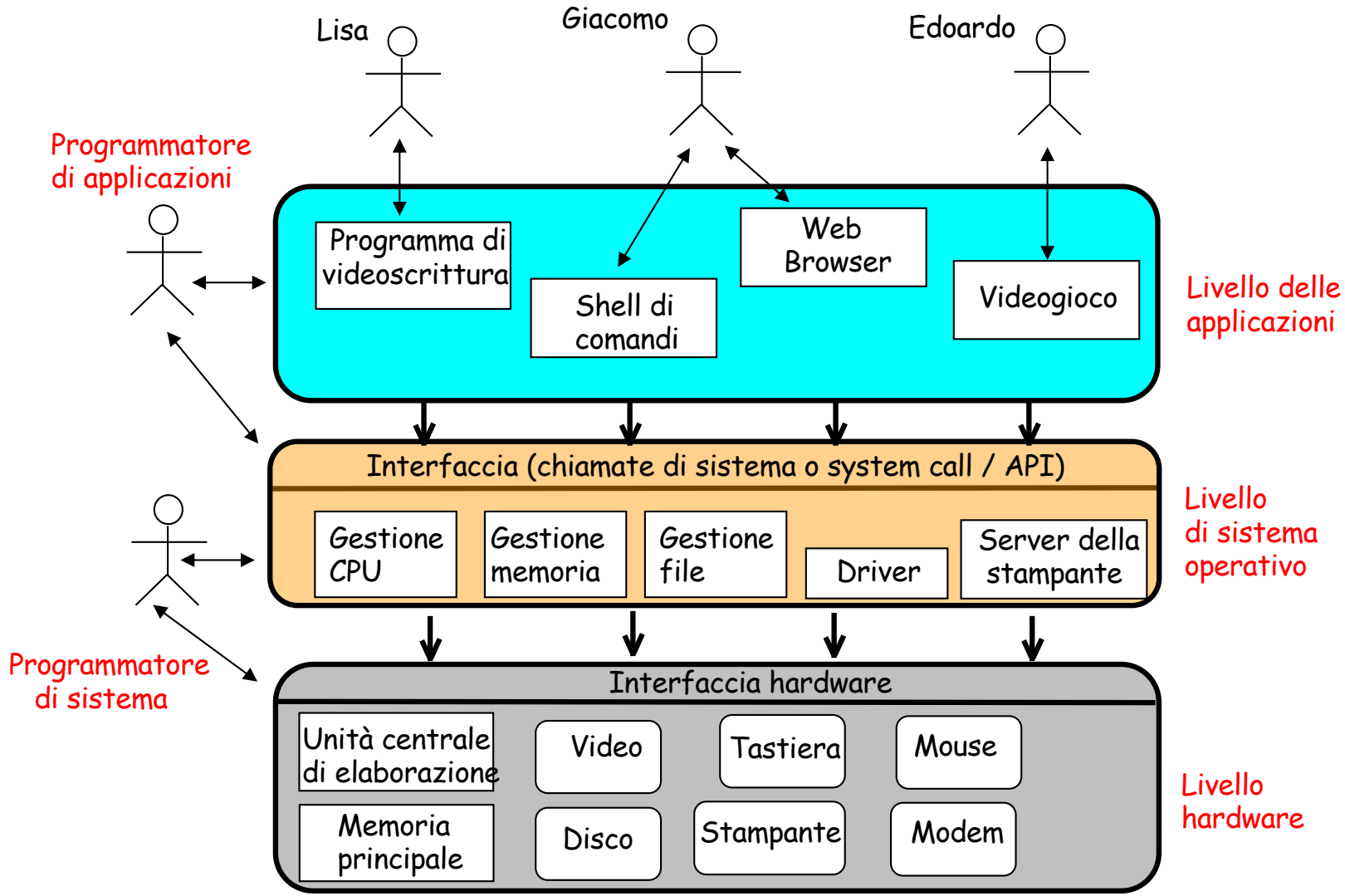
# Cos'è un Sistema Operativo?

Una possibile definizione

Insieme di programmi che agisce come intermediario tra gli utenti e l'hardware di un sistema di elaborazione

# Componenti di un sistema di elaborazione

## Un sistema di elaborazione si può suddividere in quattro componenti



# Obiettivi principali di un SO

1. Fornire metodi convenienti per utilizzare il sistema di elaborazione
2. Assicurare un uso efficiente delle risorse del sistema di elaborazione
3. Prevenire interferenze nelle attività degli utenti

# Fornire metodi convenienti per utilizzare il sistema di elaborazione

- Punto di vista utente
  - Il SO deve fornire un ambiente in grado di eseguire programmi utente facilitando così la soluzione dei problemi computazionali degli utenti tramite l'utilizzo del calcolatore

- Il SO deve
  - Definire e rendere disponibile una macchina estesa (o *virtuale*)
  - Rendere il SW applicativo indipendente dall'HW (*trasparenza*)
  - Facilitare la *portabilità* dei programmi su architetture HW differenti
  - Permettere l'uso delle risorse HW (solo) tramite system call (*chiamate di sistema*) e API (application programming interface)

# Assicurare un uso efficiente delle risorse del sistema di elaborazione

- Punto di vista del calcolatore
  - Lo stesso SO fa uso di memoria e CPU, cosa che costituisce un *overhead* e riduce le risorse disponibili per i programmi utente
  - Il SO può *monitorare* l'uso delle risorse per assicurare efficienza, ciò però *aumenta* l'overhead
  - Il SO utilizza *politiche* per garantire l'efficienza dell'uso delle risorse
- Il SO deve
  - Definire *tecniche* e *strategie* con cui assegnare una data risorsa a fronte di richieste contemporanee
  - Evitare *conflitti di accesso* alle risorse
  - Massimizzare l'utilizzo delle risorse
  - Bilanciare l'overhead dovuto al monitoraggio delle risorse e le prestazioni
  - Proteggere da eventuali malfunzionamenti (guasti, blocchi critici, ...)

# Prevenire interferenze nelle attività degli utenti

- Durante l'esecuzione possono verificarsi varie interferenze
  - I programmi di un utente e gli stessi servizi del SO possono essere disturbati dalle azioni di (altri) utenti o di loro programmi
  - Nei sistemi multiutente, si possono verificare tentativi di accesso illegale ai file utente, quali programmi e dati

- Il SO deve
  - Prevenire le interferenze
    - allocando risorse ad uso esclusivo dei programmi utente e dei servizi del SO
    - impedendo accessi illegali alle risorse
  - Tenere traccia di quali file utente possono essere acceduti e da chi usando *meccanismi di autenticazione e autorizzazione* per garantire protezione e sicurezza delle informazioni

# Possibili definizioni e ruoli di un SO

Non c'è una definizione universalmente accettata di cosa sia un SO

- *Intermediario*

  che agisce tra gli utenti di un sistema di elaborazione e l'hardware

- *Virtualizzatore di risorse*

  permette di utilizzare il sistema di elaborazione come se si avesse a disposizione una macchina funzionalmente estesa

- *Allocatore di risorse*

  gestisce ed alloca risorse fisiche/logiche (esempi: *tempo di CPU, spazi di memoria, dispositivi di I/O*) in base alle necessità dei programmi risolvendo eventuali conflitti sulle richieste e sull'uso delle risorse

- *Programma di controllo*

  controlla l'esecuzione dei programmi utenti cercando di impedire che vengano commessi errori nell'utilizzo del computer e dei dispositivi di I/O

- *Kernel*

  è il solo programma che è sempre in funzione nel calcolatore, generalmente chiamato kernel (nucleo)

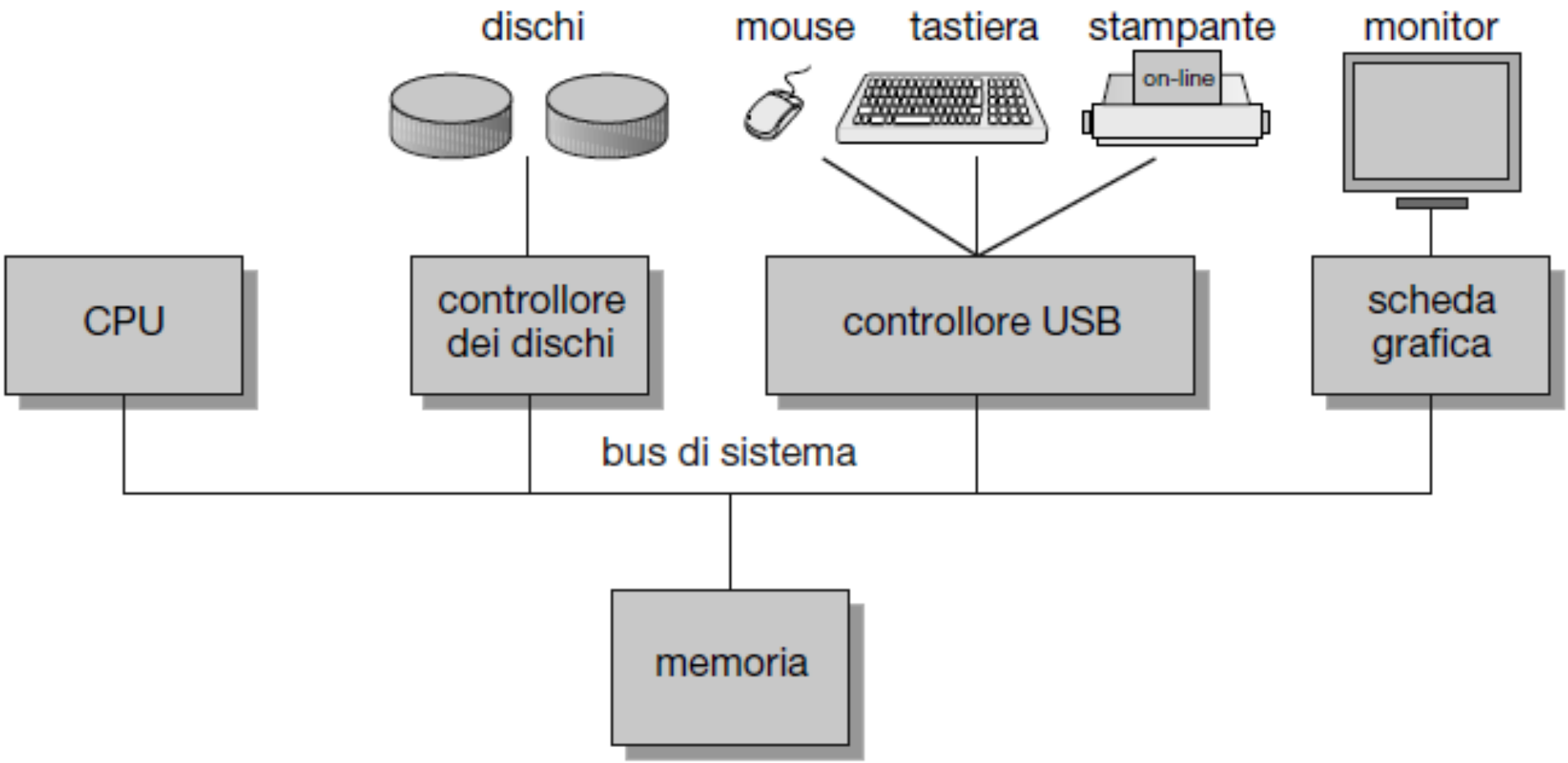
# Concetti introduttivi

- Cosa fa un SO
- Organizzazione di un sistema di elaborazione
- Gestione delle risorse
- Virtualizzazione
- Sicurezza e protezione
- Servizi di un SO
- Interfacce utente
- System call & API
- Servizi di sistema
- Progetto e implementazione di un SO
- Struttura di un SO

# Organizzazione di un sistema di elaborazione

- Architettura e funzionamento di un calcolatore generico
- Interruzioni, dispositivi di I/O & DMA
- Struttura e gerarchia della memoria
- Protezioni hardware

# Architettura di un calcolatore generico



# Architettura e funzionamento di un calcolatore generico

- Un moderno calcolatore general-purpose è composto da una o più CPU e da un certo numero di controllori di dispositivi connessi attraverso un canale di comunicazione comune (bus) che permette l'accesso alla memoria condivisa dal sistema
- La CPU e i controllori possono eseguire operazioni in parallelo, competendo per i cicli di accesso alla memoria
- Per ogni controllore di dispositivo, in genere, i SO possiedono un driver del dispositivo che gestisce le specificità del controllore e funge da interfaccia uniforme con il resto del sistema

# CPU (Central Processing Unit)

- Contiene diversi registri interni
  - *Programmabili (o general-purpose)*: durante l'esecuzione dei programmi contengono dati ed indirizzi di locazioni di memoria (es. Registro Indice, Registro Base, Stack Pointer)
  - *Di stato e controllo:* contengono informazioni sullo stato della CPU necessarie a controllarne il funzionamento

    (es. Program Counter, Instruction Register, Program Status Word)

- Esegue le istruzioni dei programmi contenuti in memoria centrale (ciclo prelievo-decodifica-esecuzione o *Fetch, Decode & Execute*)
  - *prelievo:* trasferisce nell'IR l'istruzione da eseguire prelevandola dalla locazione di memoria il cui indirizzo è in PC ed inserisce in PC l'indirizzo dell'istruzione successiva
  - *decodifica* ed *esecuzione*: decodifica l'istruzione contenuta in IR e la esegue, prelevando dalla memoria gli operandi ed inserendovi il risultato

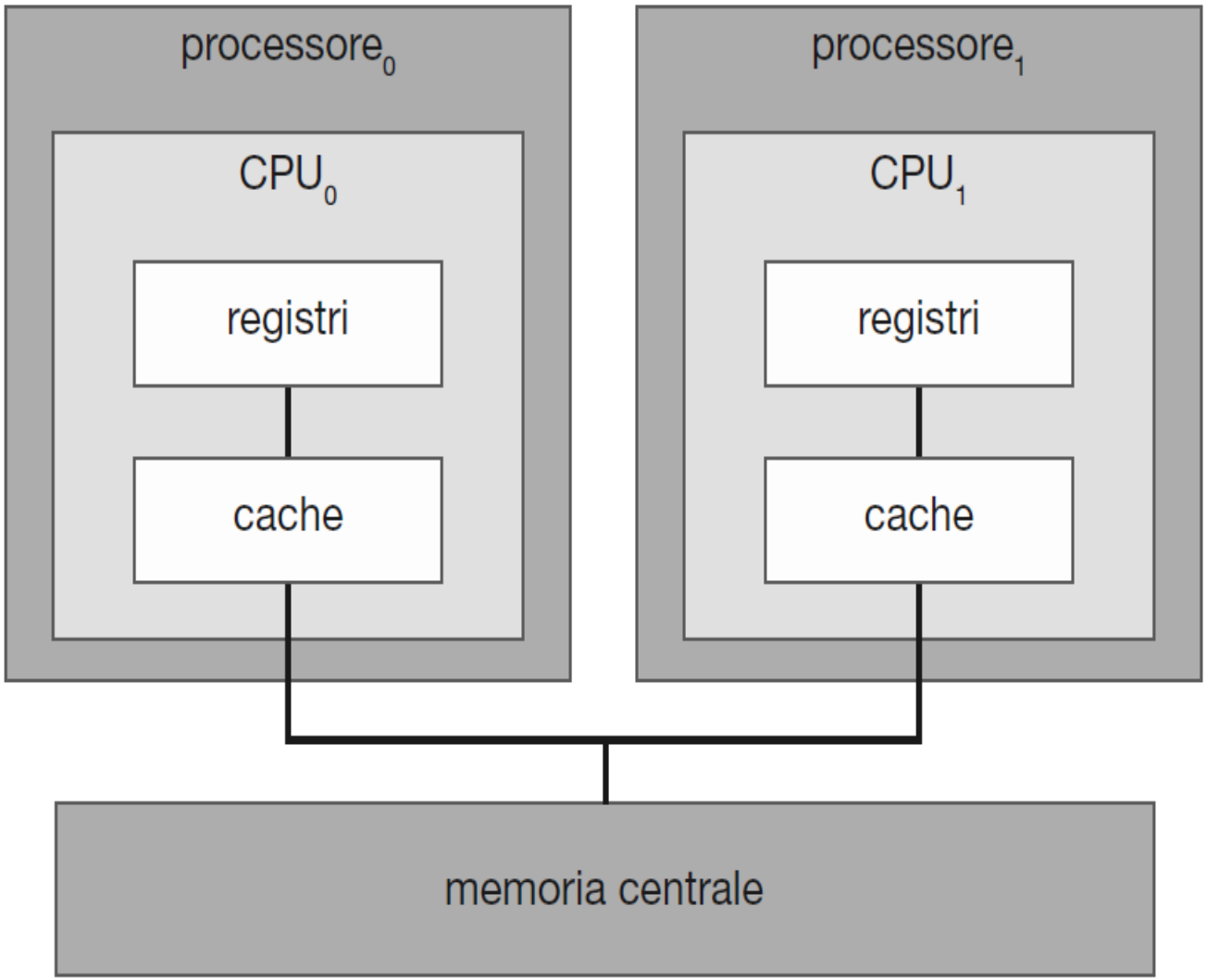
- Per migliorare le prestazioni, i sistemi di elaborazione moderni usano vari accorgimenti per eseguire più istruzioni contemporaneamente

# Architetture multiprocessore

- Le architetture multiprocessore sono dotate di più processori
- Sistemi strettamente connessi (tightly coupled systems)
  - i processori condividono memoria e clock
  - la comunicazione avviene di solito tramite memoria condivisa
- Concorrenza (reale)

  più programmi sono effettivamente in esecuzione allo stesso istante su processori differenti
- Il SO deve gestire gli accessi alle strutture comuni per evitare inconsistenze
- Vantaggi rispetto alle architetture monoprocessore
  - elevata capacità di elaborazione dei dati
  - basso costo
  - maggiore affidabilità (degradazione progressiva della performance, tolleranza ai guasti, …)

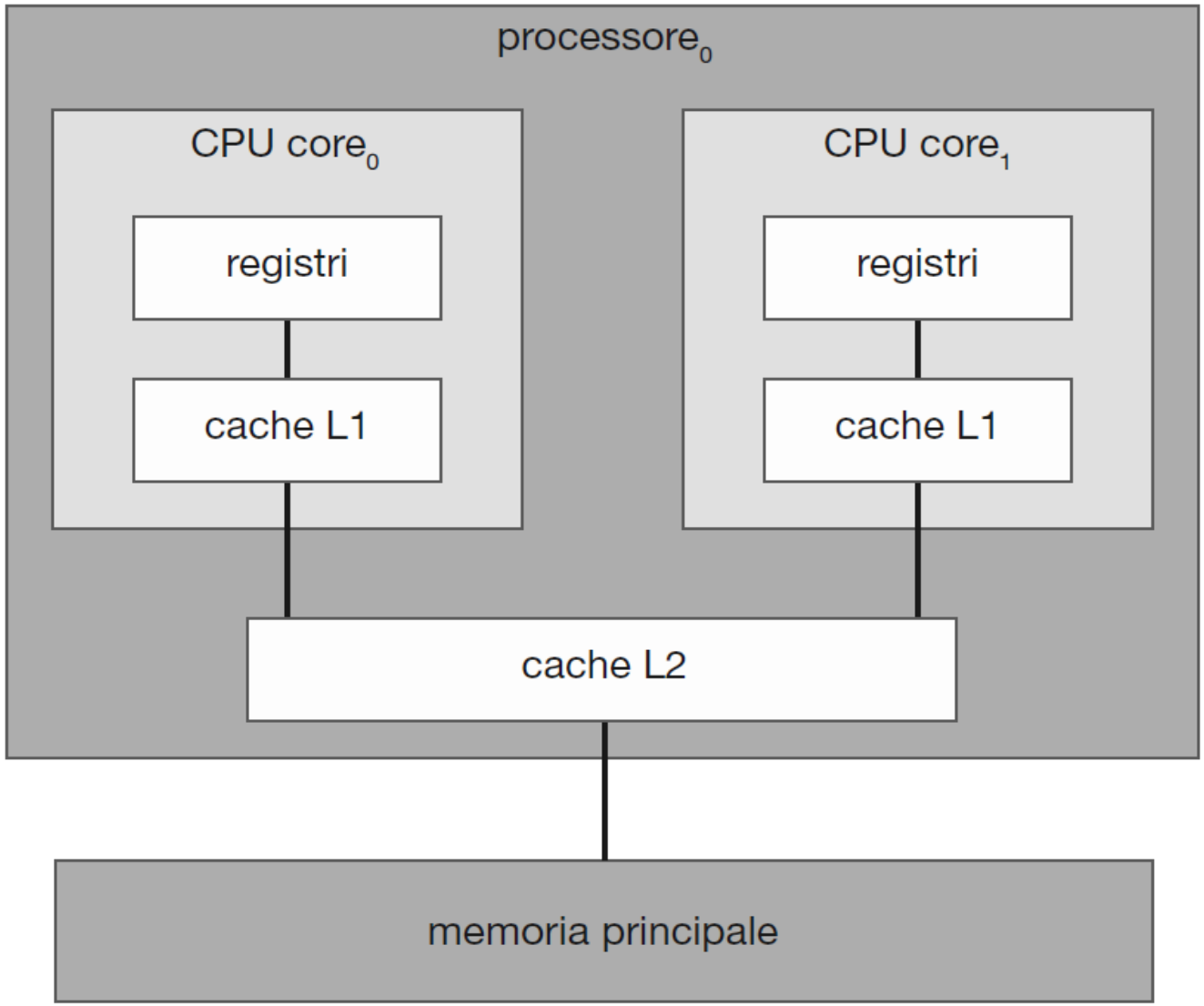
# Architettura multiprocessore con 2 CPU



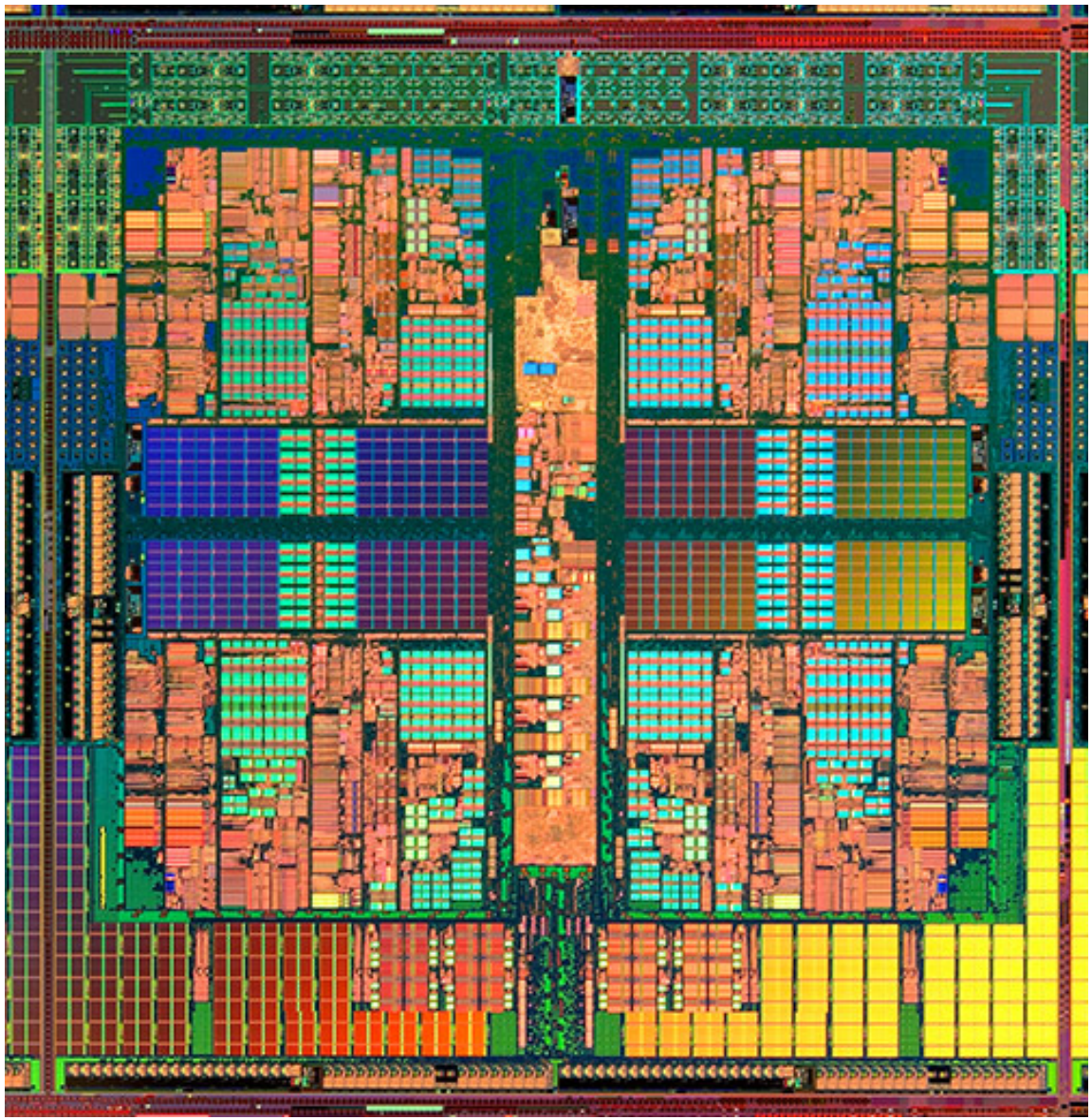
# Architetture multicore

- Le architetture multicore sono equipaggiate con processori dotati di più nuclei di elaborazione (core o unità logico-aritmetica) che condividono la stessa piastra di silicio (chip)
  - Processori multicore comuni sono dual core, quad core e octo core, rispettivamente con 2, 4 e 8 nuclei
- Ciò permette di raddoppiare, quadruplicare, ottuplicare la potenza di calcolo del singolo processore
  - Anziché processare una singola informazione per volta, un processore multicore potrà processare contemporaneamente tanti pacchetti dati quanti sono i core presenti
  - Concorrenza (reale)
- Il SO deve gestire gli accessi alle strutture comuni per evitare inconsistenze
- Vantaggi simili alle architetture multiprocessore

# Architettura dual-core con due core sullo stesso chip



# Architettura quad-core



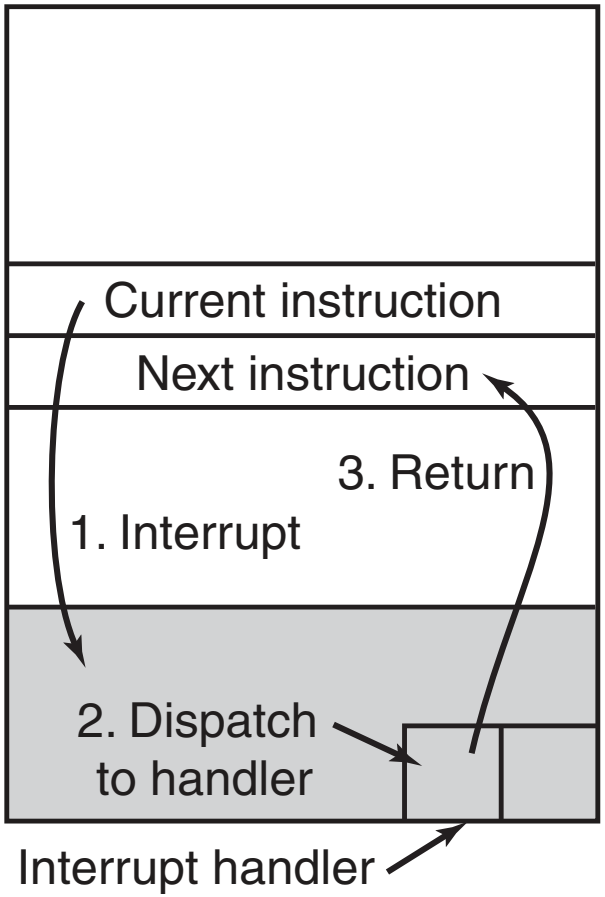
# Controllori dei dispositivi

- Ogni controllore è responsabile di uno specifico tipo di dispositivo
- Ogni controllore di dispositivo ha un buffer (memoria tampone) locale e alcuni registri usati per comunicare con la CPU
  - L'insieme di tutti i registri di un controllore forma una porta di I/O
- L'I/O avviene dal dispositivo al buffer locale del controllore (input), o viceversa (output)
- La CPU sposta i dati dal buffer locale del controllore alla memoria principale (o viceversa) tramite il bus
- Controllori dei dispositivi di I/O e CPU possono operare concorrentemente
- Quando un dispositivo ha completato l'operazione corrente, il suo controllore informa la CPU lanciando un segnale di interruzione (tramite il bus)

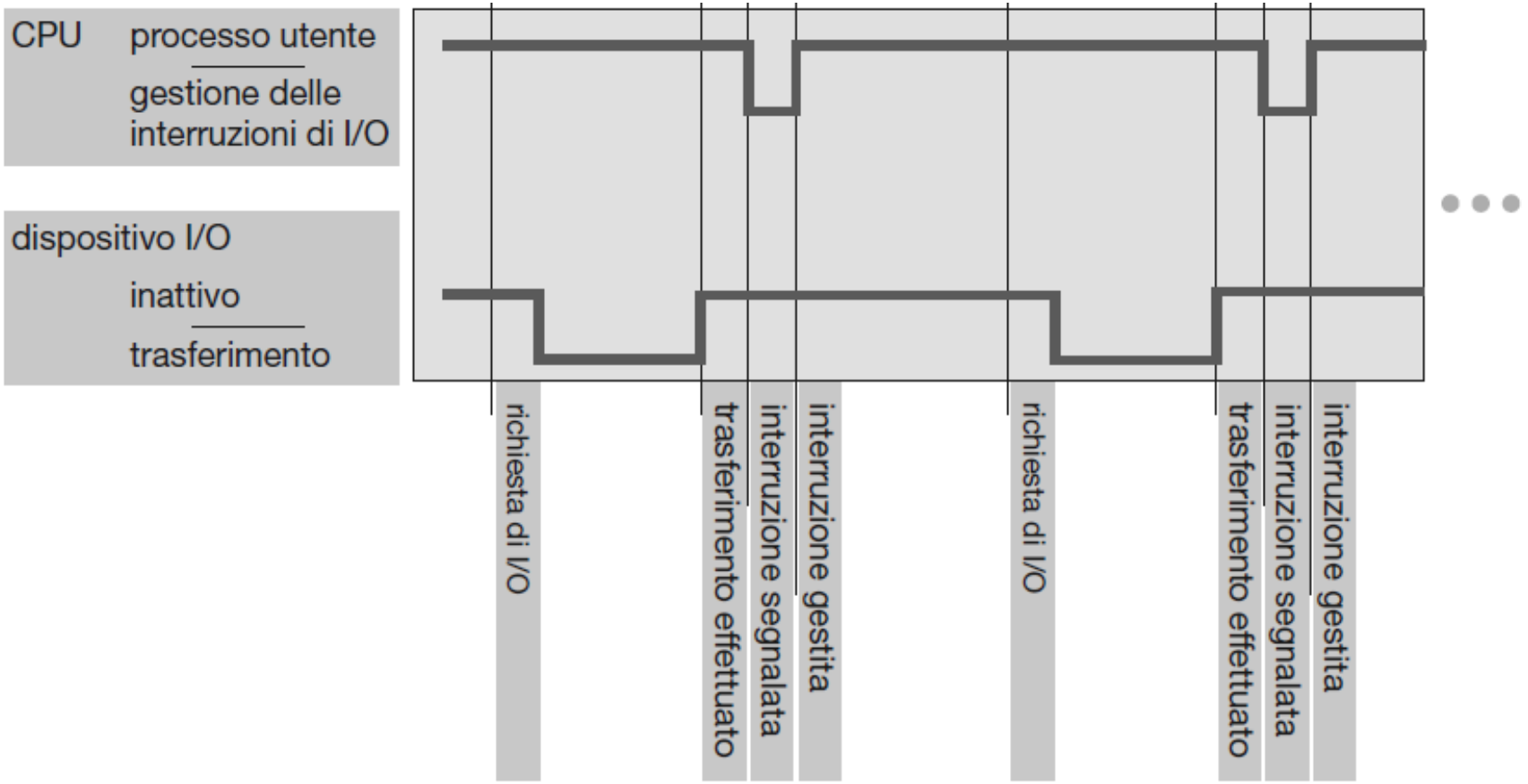
# Meccanismo delle interruzioni

È una combinazione di comportamenti HW e SW, attivati da un segnale HW (segnale di controllo sul bus) o SW (istruzione), per cui:

1. al rilevamento dell'interruzione, si interrompe l'esecuzione del processo corrente
2. si assegna la CPU ad una funzione di gestione dell'interruzione, la cui posizione è nota a priori
3. al termine dell'esecuzione della funzione di gestione, si riprende l'esecuzione del processo che era stato sospeso

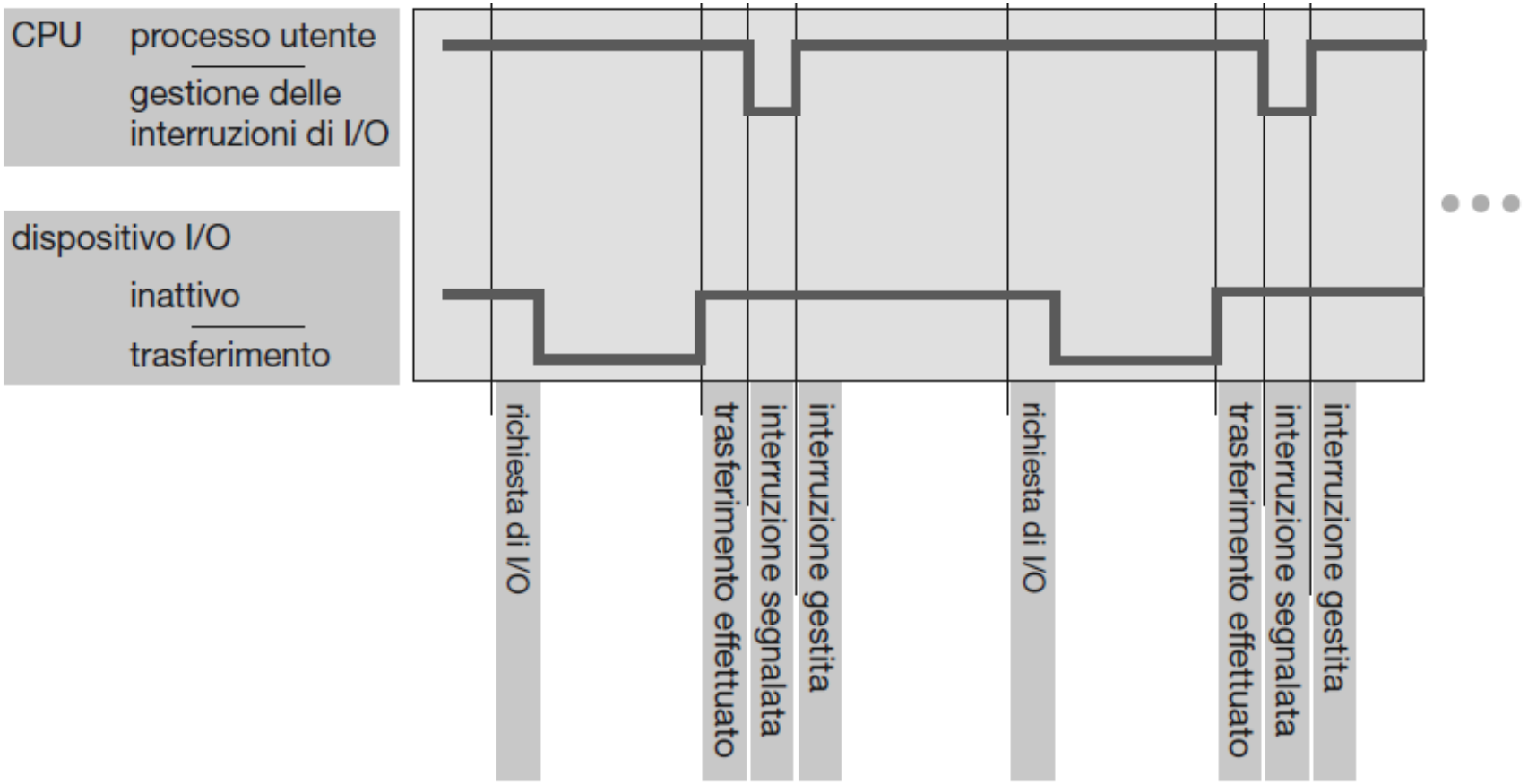


# Diagramma temporale delle interruzioni: singolo processo che invia dati in output




- La CPU commuta tra l'esecuzione dei processi e l'esecuzione delle funzioni di gestione delle interruzioni
- I dispositivi di I/O alternano tra attesa e attività di I/O

# Diagramma temporale delle interruzioni: singolo processo che invia dati in output




- Intervallo di latenza tra richieste di I/O e inizio del trasferimento da parte del dispositivo, e tra interruzione generata dal controllore del dispositivo al completamento del trasferimento e inizio dell'esecuzione della corrispondente funzione di gestione da parte della CPU
- Dispositivi di I/O e CPU possono operare concorrentemente

# Cause di interruzione

- Alcuni eventi che possono causare una interruzione:
  - interruzioni esterne (alla CPU): completamento di un I/O
  - fallimenti hardware: errore di parità, mancanza di tensione
  - trap: divisione per 0, overflow aritmetici, accesso alla memoria non valido perché viola la protezione
  - interruzioni software: generate da *system call,* istruzioni apposite per chiedere al kernel servizi o risorse alle quali i programmi non possono accedere direttamente

    (*int* Intel Pentium, *trap* Motorola 68000, *syscall* MIPS R3000)
- Alcuni autori usano il termine eccezione (anziché interruzione)
- Le interruzioni possono essere quindi inattese, previste o addirittura volute dal programmatore

# Gestione delle interruzioni

I moderni SO fanno uso di meccanismi raffinati forniti dall'HW sottostante

- Il tipo di una interruzione è individuato tramite un numero che è l'indice del vettore delle interruzioni: una tabella in cui ad ogni interruzione sono riservate due locazioni
  - la prima contiene l'indirizzo della funzione di gestione
  - la seconda contiene il valore che il registro PS deve avere durante l'esecuzione della funzione di gestione
- Nel registro PS (Program Status Word) c'è un bit di abilitazione delle interruzioni
  - Se posto ad 1: alla terminazione di un ciclo prelievo-decodifica-esecuzione, prima di leggere l'istruzione successiva, la CPU controlla se ci sono interruzioni pendenti
  - Se posto a 0, la CPU non controlla la presenza di interruzioni

# Gestione delle interruzioni

- Le interruzioni possono avere delle priorità
  - In caso ci siano più interruzioni pendenti, la CPU serve quella a priorità maggiore (trasferendo il controllo alla funzione di gestione corrispondente)
  - Interruzioni a priorità maggiore possono sospendere la gestione di interruzioni a priorità minore (serve un kernel prelazionabile)
  - Alle interruzioni SW si assegna solitamente una priorità bassa rispetto alle interruzioni generate dai dispositivi di I/O (per favorire l'I/O e lasciare inutilizzati i dispositivi per il minor tempo possibile)
- Alcune interruzioni sono mascherabili (tipicamente quelle generate dai controllori dei dispositivi), cioè possono essere disattivate dalla CPU (tramite il bit di abilitazione alle interruzioni) prima dell'inizio di una sequenza di istruzioni che non deve essere interrotta
  - Altre non lo sono (corrispondono ad errori non recuperabili)

# Vettore delle interruzioni di un Intel Pentium

| indice del vettore | descrizione |
|---|---|
| 0 | divide error |
| 1 | debug exception |
| 2 | null interrupt |
| 3 | breakpoint |
| 4 | INTO-detected overflow |
| 5 | bound range exception |
| 6 | invalid opcode |
| 7 | device not available |
| 8 | double fault |
| 9 | coprocessor segment overrun (reserved) |
| 10 | invalid task state segment |
| 11 | segment not present |
| 12 | stack fault |
| 13 | general protection |
| 14 | page fault |
| 15 | (Intel reserved, do not use) |
| 16 | floating-point error |
| 17 | alignment check |
| 18 | machine check |
| 19-31 | (Intel reserved, do not use) |
| 32-255 | maskable interrupts |

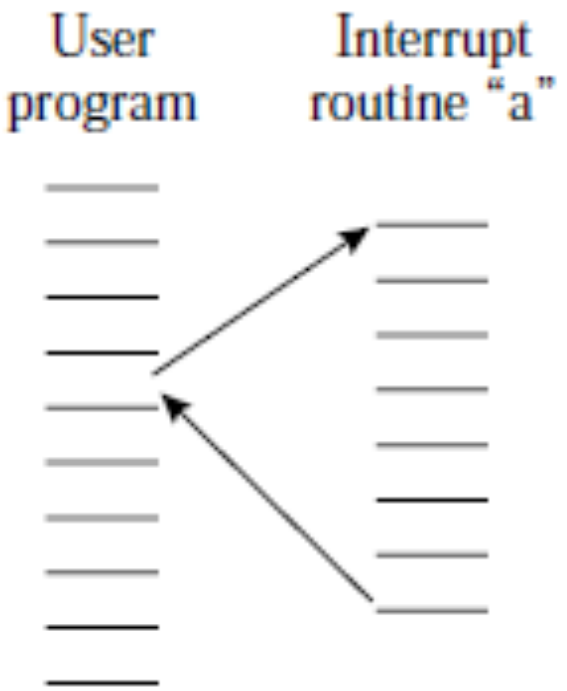
0-31 non mascherabili: condizioni di errore
32-255 mascherabili: es interruzioni generate dai dispositivi

# Gestione delle interruzioni

I SO utilizzano due approcci principali per la gestione delle interruzioni

- Kernel non prelazionabile
- Kernel prelazionabile

# Kernel non prelazionabile (Interruzioni semplici)

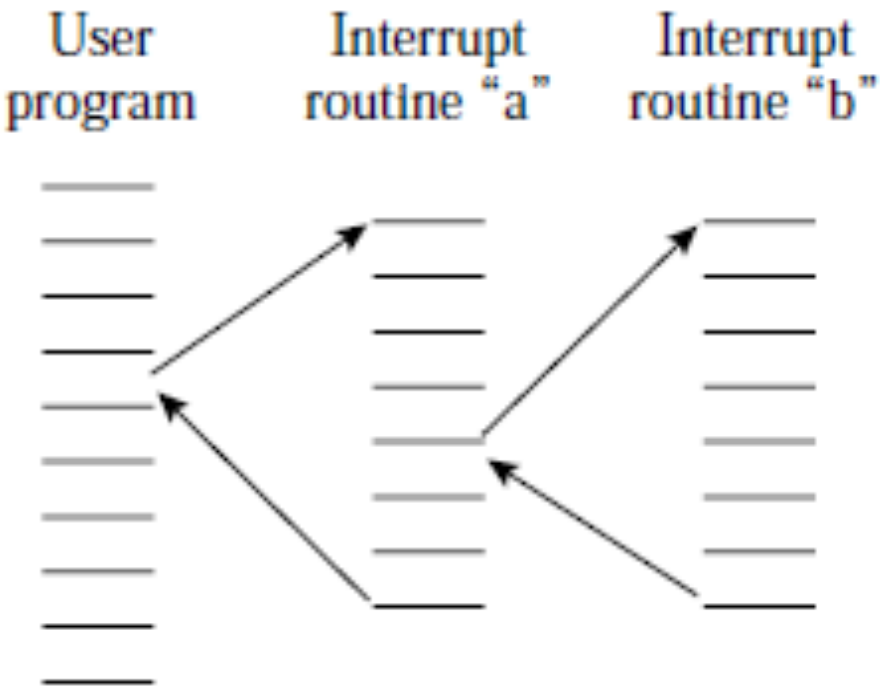



Durante la gestione di una interruzione il meccanismo di gestione delle interruzioni è disabilitato per evitare interferenze

- Le interruzioni che si verificano durante questo periodo di tempo sono poste in attesa e segnalate quando il meccanismo di gestione delle interruzioni viene riabilitato
- Semplifica la progettazione, ma può ritardare la gestione di alcune interruzioni, anche se hanno priorità alta
- L'esecuzione della funzione di gestione dev'essere rapida

# Kernel prelazionabile (Interruzioni annidate)

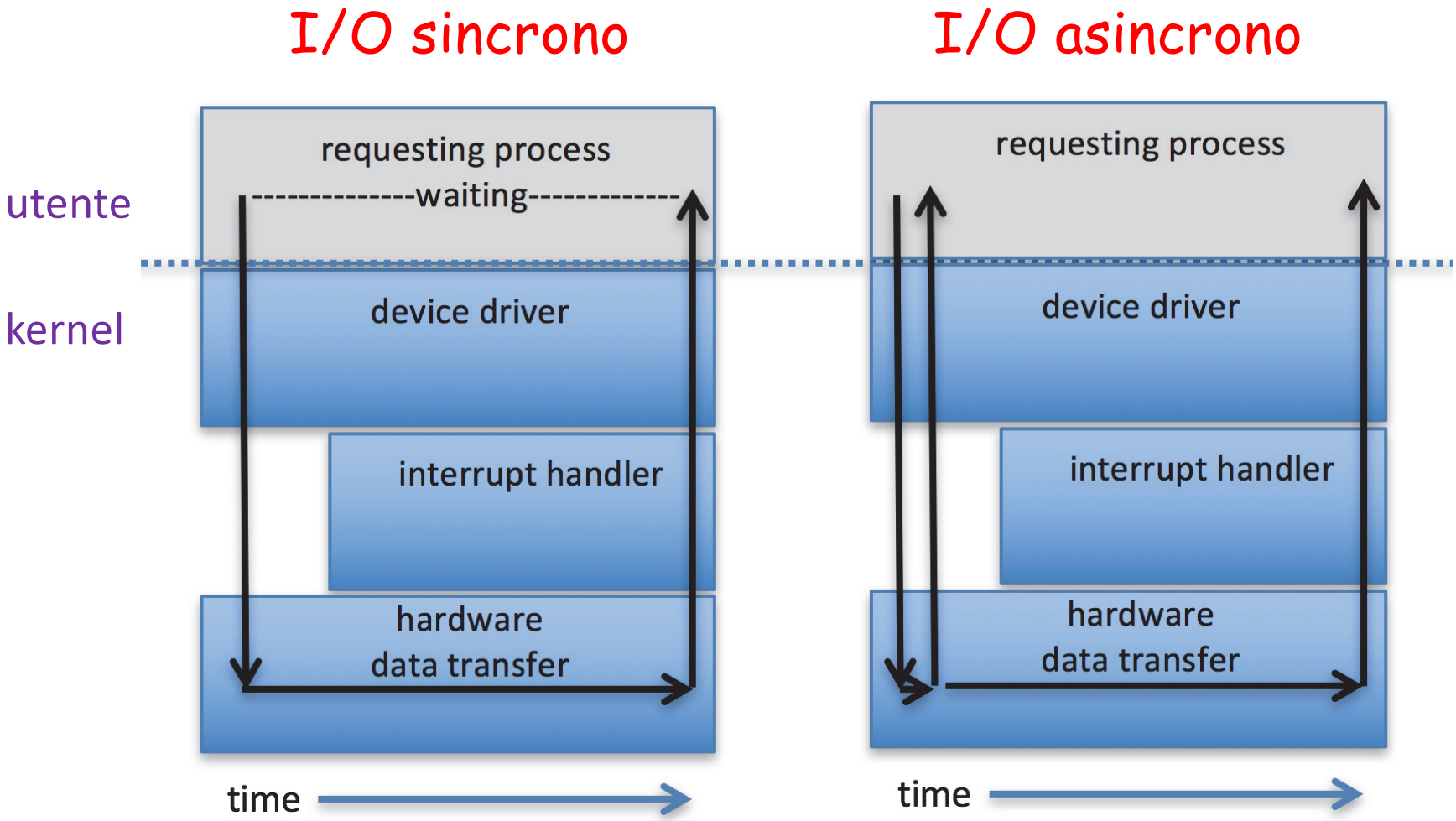
- Architetture più sofisticate permettono di gestire altre interruzioni (più critiche) mentre è già in corso la gestione di una interruzione
- Tipicamente si adotta uno schema a priorità
  - Ogni interruzione è associata ad un livello di priorità
  - Interruzioni a priorità più alta possono interrompere la gestione di interruzioni a priorità più bassa



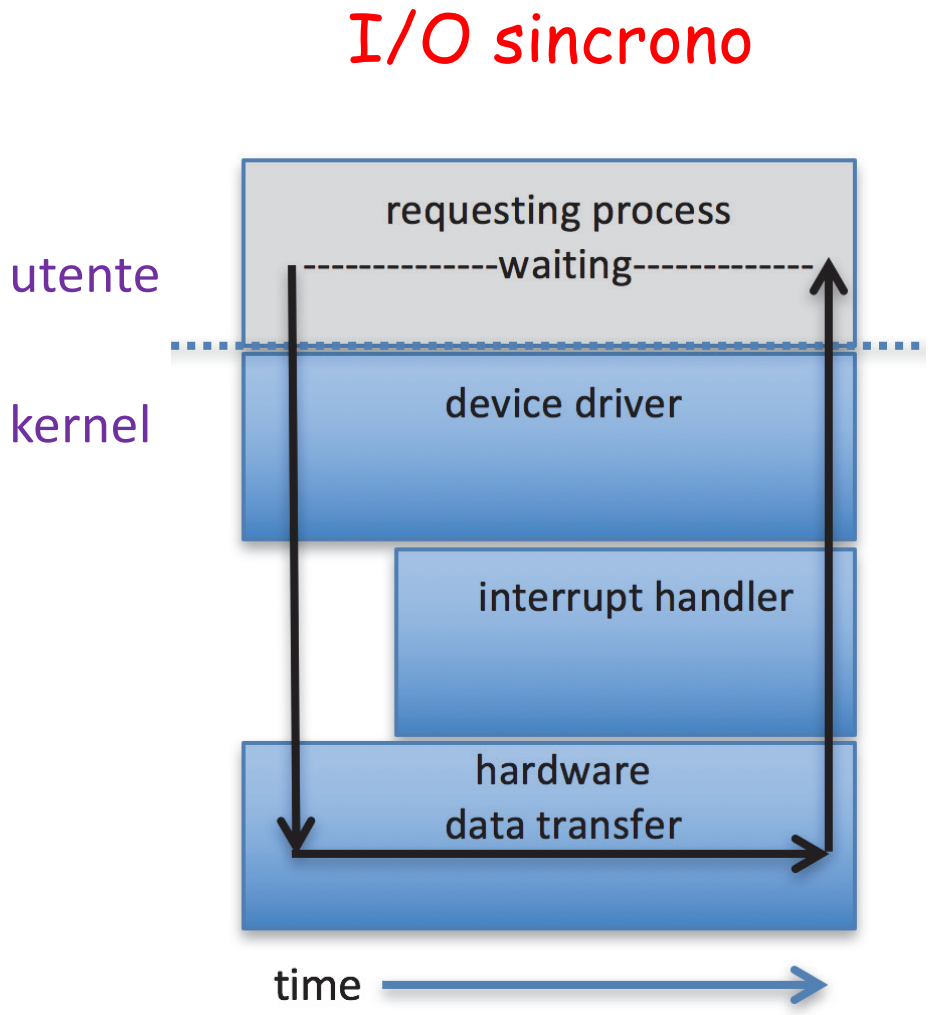

- Problemi di consistenza dei dati:
  - Potrebbero sorgere se due o più routine di gestione delle interruzioni attivate in maniera annidata aggiornassero gli stessi dati del kernel
  - Il kernel deve utilizzare uno schema di sincronizzazione per assicurare che solo una routine di gestione per volta possa accedere tali dati in ogni istante

# I/O sincrono e asincrono

## Due principali modalità per l'esecuzione dell'I/O



# I/O sincrono e asincrono




All'invocazione di un I/O, il controllo ritorna al processo utente solo al completamento dell'I/O

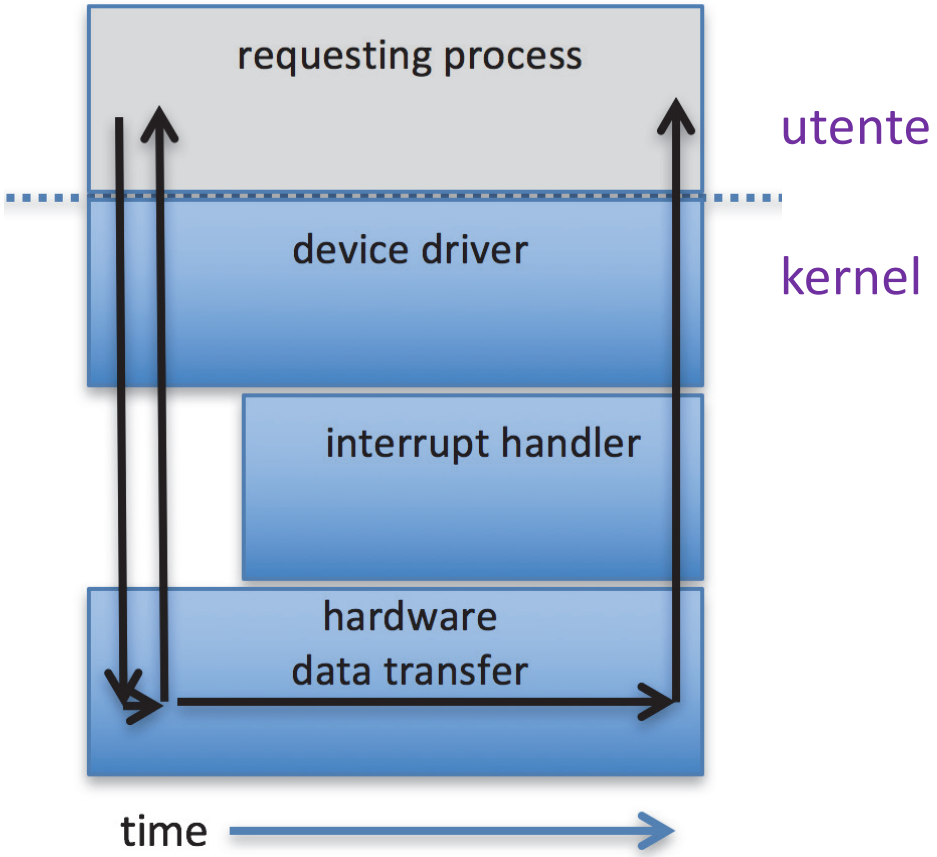
- L'istruzione **wait** rende la CPU inattiva fino all'interruzione successiva
- In un dato istante, al più una richiesta di I/O è in sospeso (quindi, quando arriva una interruzione da dispositivo, il SO riconosce subito richiesta e dispositivo coinvolti)
- Non c'è elaborazione simultanea dell'I/O su più dispositivi

# I/O sincrono e asincrono

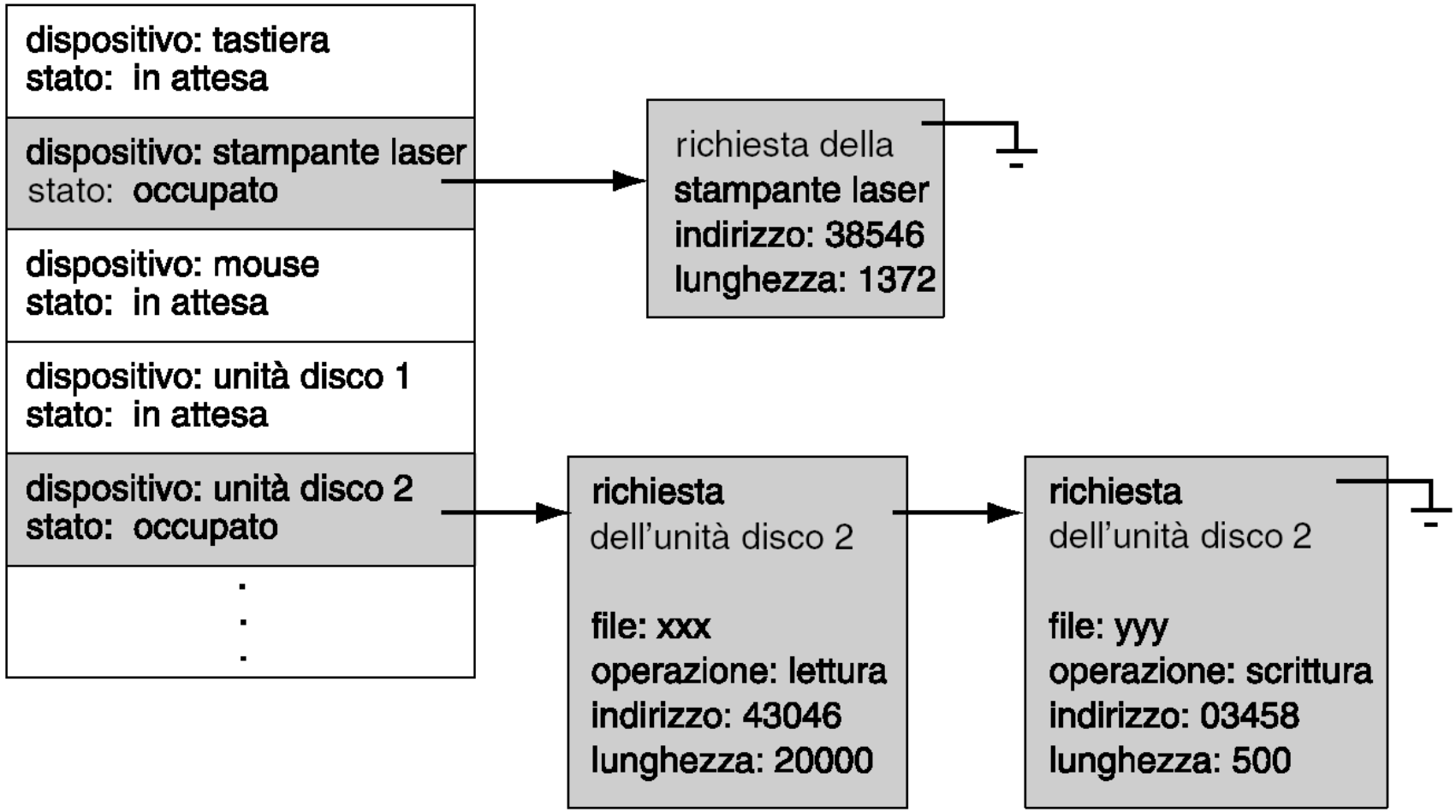
All'invocazione di un I/O, il controllo ritorna al processo utente immediatamente, senza aspettare il completamento dell'I/O

- Se necessario, si deve usare una system call bloccante per permettere al processo utente di attendere il completamento dell'I/O
- Per tener traccia delle richieste di I/O attive ad un dato istante si usa una tabella di stato dei dispositivi che contiene una voce per ogni dispositivo di I/O che ne indica tipo, stato (in attesa, occupato) e coda delle richieste pendenti
- All'arrivo di una richiesta di I/O, il SO consulta la tabella per determinare lo stato del dispositivo e, se occupato, inserisce la richiesta nella coda

I/O asincrono

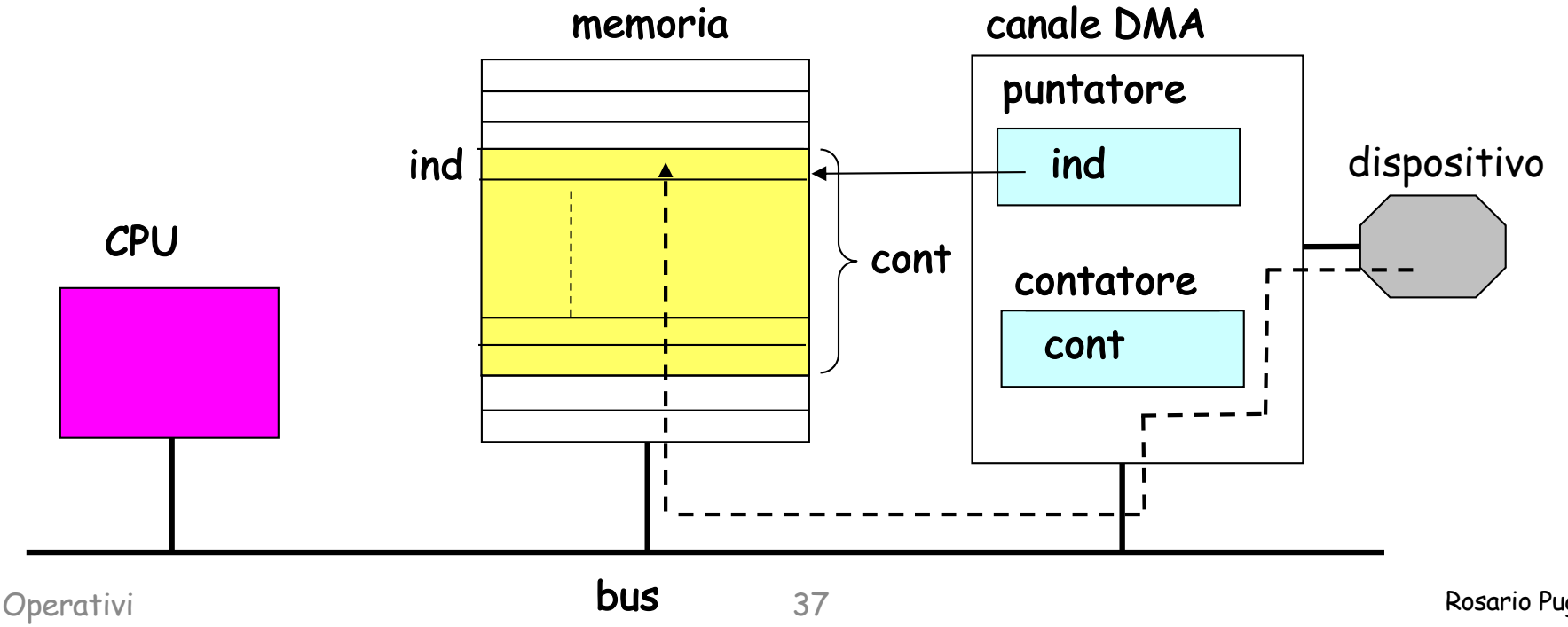


# Tabella di stato dei dispositivi



# Accesso diretto alla memoria (DMA)

- Un canale DMA (direct memory access) è un circuito che collega un dispositivo di I/O al bus di sistema e lo abilita ad accedere direttamente alla memoria senza richiedere l'intervento della CPU
- Contiene due registri:
  - uno indica la locazione di memoria a partire dalla quale inizia la sequenza dei dati da trasferire
  - l'altro indica la quantità di dati da trasferire



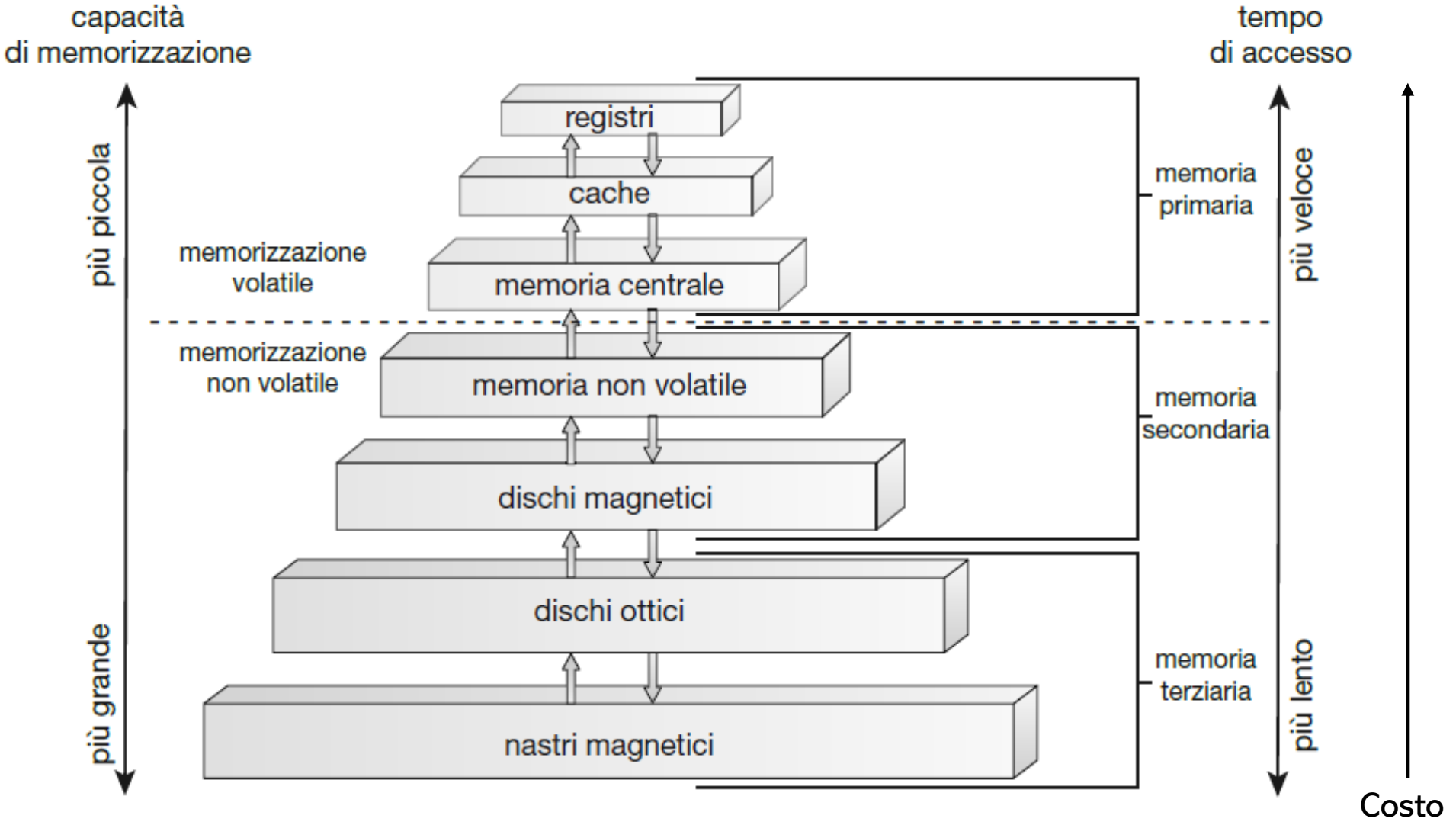
# Accesso diretto alla memoria (DMA)

- Un canale DMA (direct memory access) è un circuito che collega un dispositivo di I/O al bus di sistema e lo abilita ad accedere direttamente alla memoria senza richiedere l'intervento della CPU
- Contiene due registri:
  - uno indica la locazione di memoria a partire dalla quale inizia la sequenza dei dati da trasferire
  - l'altro indica la quantità di dati da trasferire
- In pratica un canale DMA è un controllore di dispositivo intelligente in grado di gestire il trasferimento di interi blocchi di dati dal buffer direttamente alla memoria principale (e viceversa)
  - Usato per dispositivi di I/O ad alta velocità in grado di trasmettere informazioni a velocità prossime a quelle di accesso della memoria
- Vantaggio: viene generato solo una interruzione per l'intero blocco, anzicché una per ogni byte/parola come avviene nei dispositivi più lenti (quindi riduce l'overhead dovuto alla gestione delle interruzioni)

# Struttura della memoria

- Una memoria è un vettore di parole ciascuna con un proprio indirizzo
- Idealmente un computer dovrebbe contenere una memoria capiente e veloce, cosicché gli accessi in memoria non rallentino la CPU
- Ciò non è possible a causa dei costi, ma è ovviamente desiderabile realizzare un servizio che possa fornire caratteristiche simili
- Soluzione: *gerarchia* di memorie con caratteristiche diverse
  - La più veloce (e costosa) è la più piccola, la più lenta è la più grande
  - La CPU accede direttamemnte solo alla memoria più veloce
  - Se il dato (o istruzione) è presente (hit) viene usato, altrimenti (miss) viene copiato dalla memoria immediatamente più lenta nella più veloce e quindi usato
  - Il dato rimane nella memoria più veloce finché non viene rimosso per far spazio ad altri dati
  - Gli altri livelli nella gerarchia sono utilizzati in maniera analoga
- Tempo di accesso effettivo: diversamente dal tempo di accesso reale, dipende da quanto frequentemente un miss si verifica

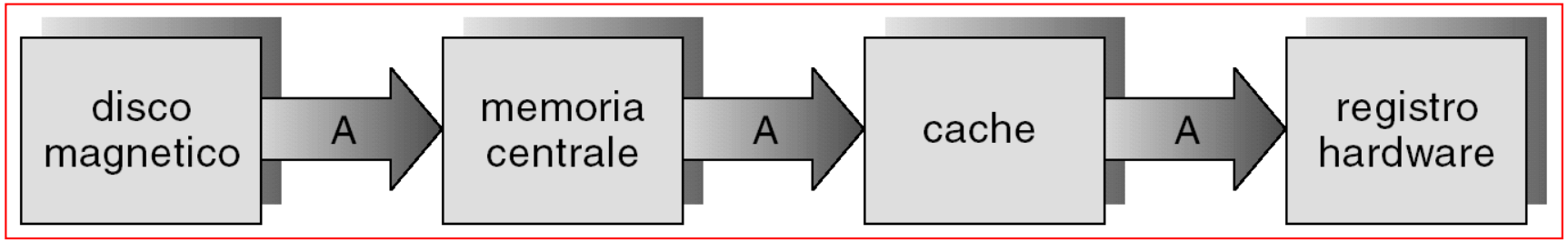
# Gerarchia dei dispositivi di memorizzazione



# Caratteristiche della memoria

- Memoria cache
  - viene inserita tra CPU e bus, contiene informazioni che sono in RAM
  - molto veloce e molto costosa
- Memoria centrale o principale (random access memory, RAM)
  - unica area di memoria di grandi dimensioni direttamente accessibile da CPU
  - volatile (perde il contenuto quando non è più alimentata)
  - alta velocità di accesso dei dati, ma costosa
- Memoria secondaria o di massa
  - estensione delle memoria centrale che fornisce grande capacità di memoria non volatile (es., unità a disco)
  - economica ma con tempi di accesso relativamente alti
- Memoria non volatile
  - ROM (read only memory), di sola lettura, contiene il programma di *bootstrap* (codice capace di localizzare il kernel del SO, caricarlo in memoria ed avviarne l'esecuzione)
  - Dischi a stato solido, pennine USB, …

# Trasferimento dei dati da disco a registro

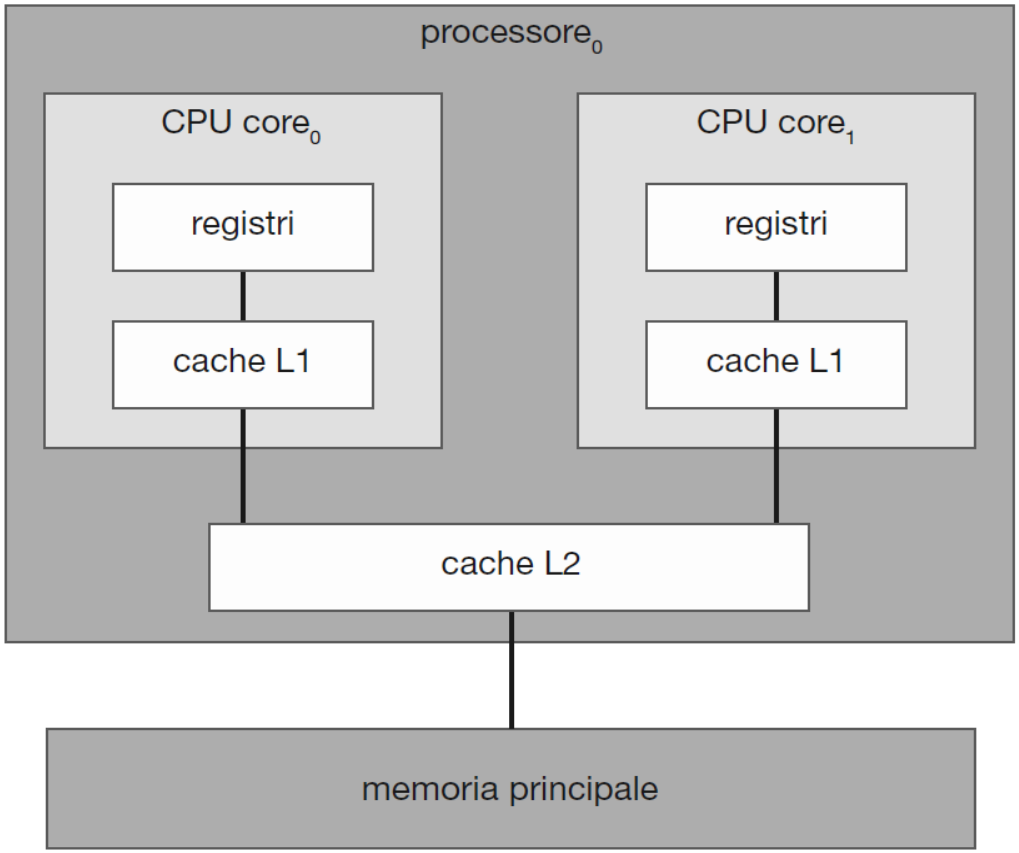



- Memoria centrale e cache hanno un funzionamento analogo: i dati vengono trasferiti in blocchi di byte (es. pagine)

- Differenza
  - gestione della memoria centrale e trasferimento dei blocchi tra disco e memoria centrale sono gestiti dal SW (SO)
  - gestione della cache e trasferimento dei blocchi tra memoria centrale e cache sono gestiti dall'HW

# Funzionamento di una memoria gerarchica

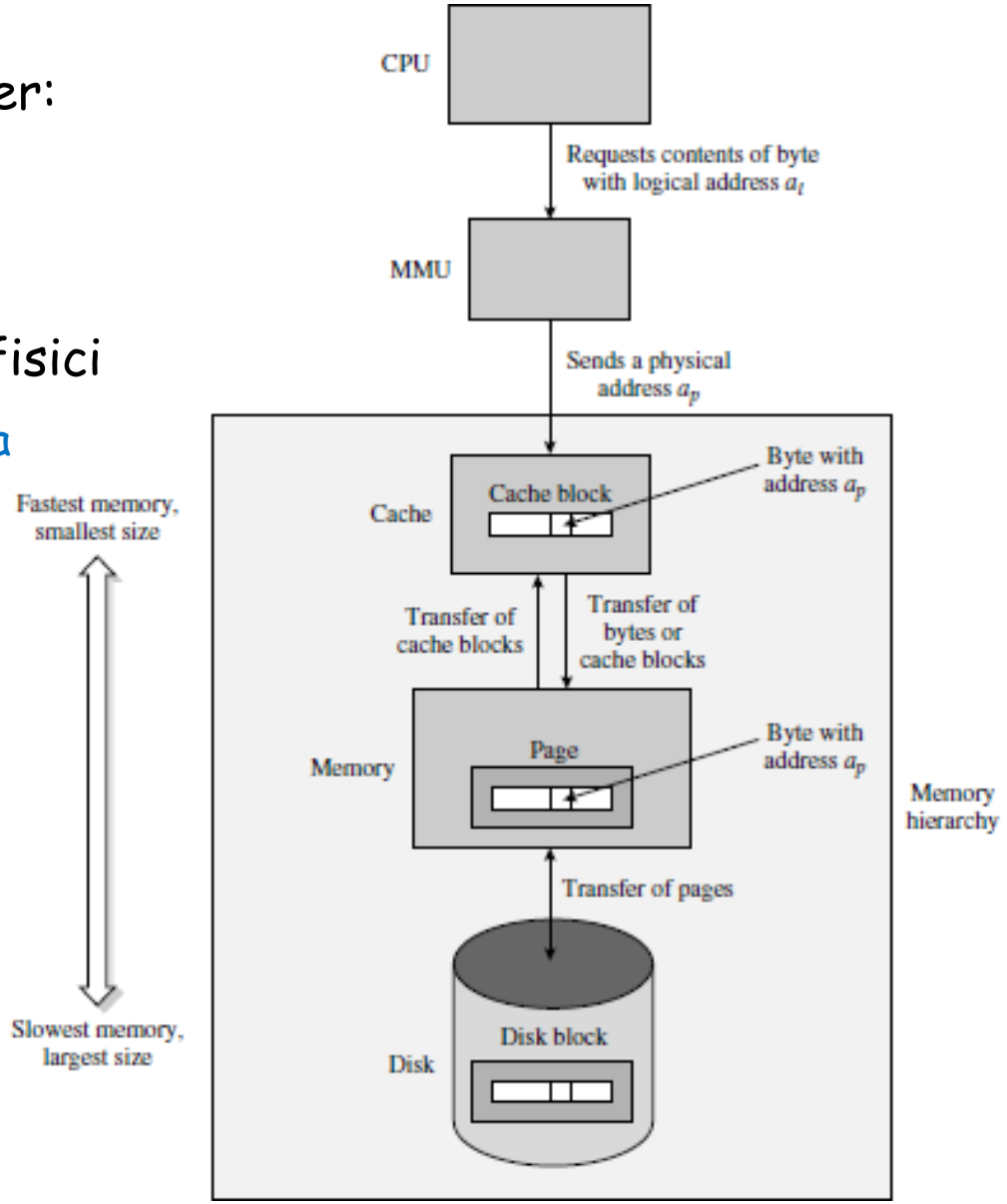
- A causa della grande differenza di velocità tra RAM e cache, per ridurre ulteriormente il tempo di accesso effettivo alla memoria, si utilizza una gerarchia di memorie cache invece di una singola cache
  - Una cache L1 (cioè di primo livello) è montata sul chip della CPU
  - Sullo stesso chip può essere montata anche un'altra cache, detta L2 (cioè di secondo livello), più lenta ma più capiente della L1
  - Una cache L3 ancora più capiente ma più lenta della L2 tipicamente è esterna al processore



# Funzionamento di una memoria gerarchica

I computer moderni differiscono per:

- gerarchie di memorie cache
- disposizione della MMU (Memory Management Unit), che traduce dinamicamente indirizzi logici in fisici
  - Nella figura, la MMU è interposta tra la CPU e la cache



# Funzionamento di una memoria gerarchica

- Configurazione parallela della MMU e della cache L1
  - Alla L1 viene inviato un indirizzo logico anzicché uno fisico
  - Questa configurazione elimina la necessità di tradurre gli indirizzi prima di effettuare la ricerca in L1, velocizzando l'accesso ai dati nel caso in cui si verifichi un *hit* nella cache L1
  - Inoltre, consente di sovrapporre la traduzione di un indirizzo effettuata dalla MMU con la ricerca nella cache L1, risparmiando tempo nel caso in cui si verifichi un *miss* nella cache L1

# Caratteristiche di varie tipologie di memoria

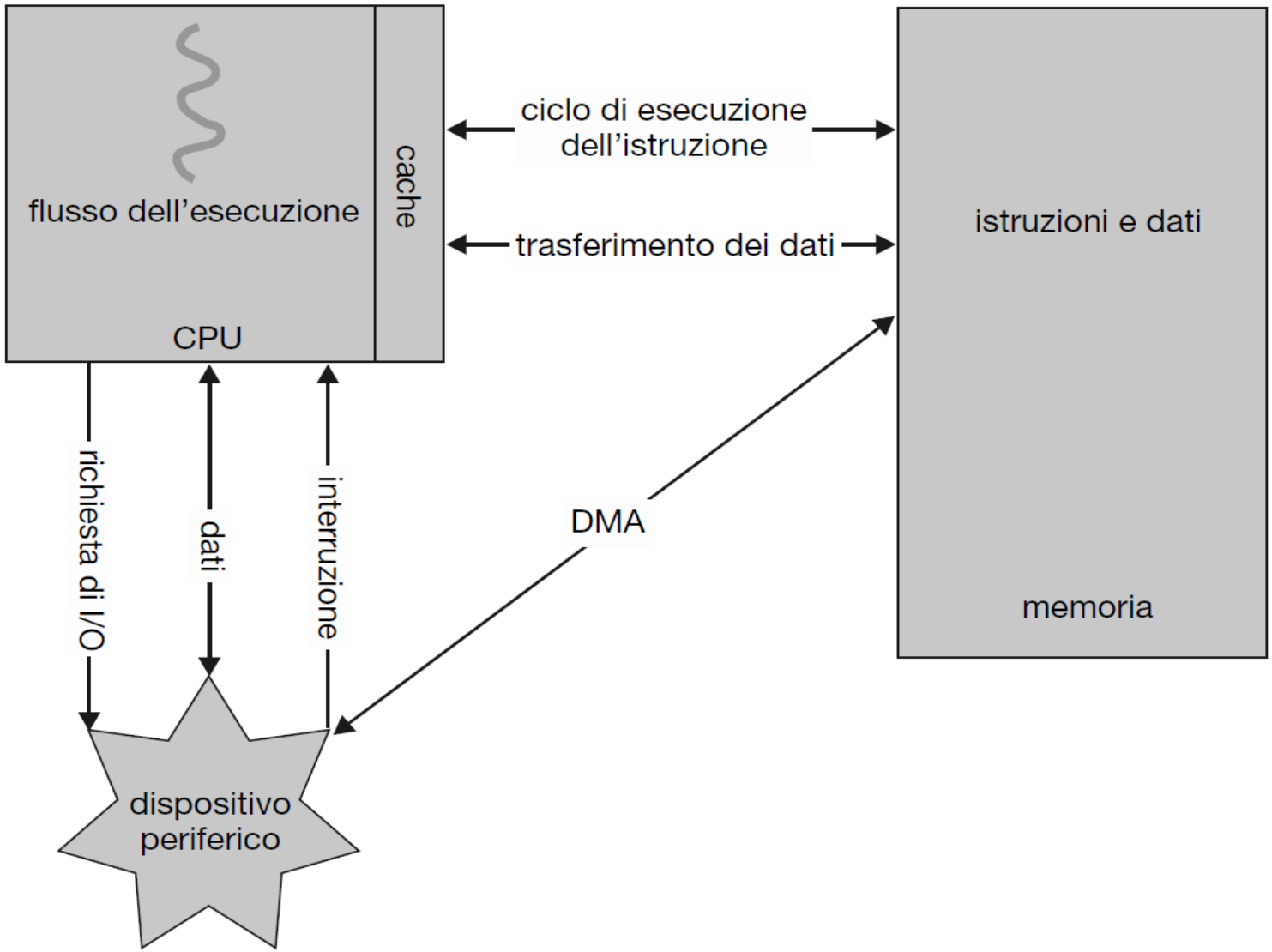
| Livello | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Nome | registri | cache | memoria centrale | disco a stato solido | disco magnetico |
| Dimensione tipica | < 1 KB | < 16 MB | < 64 GB | < 1 TB | < 10 TB |
| Tecnologia | memoria dedicata con porte multiple (CMOS) | CMOS SRAM (on-chip o off-chip) | CMOS DRAM | memoria flash | disco magnetico |
| Tempo d'accesso (ns) | 0,25 – 0,5 | 0,5 – 25 | 80 – 250 | 25.000-50.000 | 5.000,000 |
| Ampiezza di banda (MB/s) | 20.000 – 100.000 | 5000 – 10.000 | 1000 – 5000 | 500 | 20 – 150 |
| Gestito da | compilatore | hardware | sistema operativo | sistema operativo | sistema operativo |
| Supportato da | cache | memoria centrale | disco | disco | disco o nastro |

# Unità di misura: corrispondenza tra valori e prefissi

| Exp. | Explicit | Prefix | Exp. | Explicit | Prefix |
|---|---|---|---|---|---|
| $10^{-3}$ | 0.001 | milli | $10^{3}$ | 1,000 | Kilo |
| $10^{-6}$ | 0.000001 | micro | $10^{6}$ | 1,000,000 | Mega |
| $10^{-9}$ | 0.000000001 | nano | $10^{9}$ | 1,000,000,000 | Giga |
| $10^{-12}$ | 0.000000000001 | pico | $10^{12}$ | 1,000,000,000,000 | Tera |
| $10^{-15}$ | 0.000000000000001 | femto | $10^{15}$ | 1,000,000,000,000,000 | Peta |
| $10^{-18}$ | 0.000000000000000001 | atto | $10^{18}$ | 1,000,000,000,000,000,000 | Exa |
| $10^{-21}$ | 0.000000000000000000001 | zepto | $10^{21}$ | 1,000,000,000,000,000,000,000 | Zetta |
| $10^{-24}$ | 0.000000000000000000000001 | yocto | $10^{24}$ | 1,000,000,000,000,000,000,000,000 | Yotta |

- Una linea di comunicazione a 1Kbps trasmette 1.000 bit al secondo e una LAN a 10Mbps funziona a 10.000.000 bit/s, dato che le velocità sono espresse in base 10
- I prefissi hanno significati diversi quando sono usati per misurare le dimensioni delle memorie (che sono indicate come potenze di 2):
  - Kilo significa $2^{10}$ anziché $10^{3}$, quindi 1KB corrisponde a 1024 byte (e non 1.000)
  - Allo stesso modo, 1MB corrisponde a 1.048.576 byte (cioè $2^{20}$ byte) e 1GB corrisponde a $2^{30}$ byte

# Schema delle interazioni tra i componenti di un elaboratore



# Meccanismi di Protezione

- La condivisione delle risorse, se da un lato aumenta l'efficienza del sistema, dall'altro richiede che il SO garantisca che un programma malfunzionante non possa influenzare il comportamento di altri programmi
  - I programmi vanno protetti da eventuali errori causati da altri programmi
  - Il SO deve essere protetto dai programmi applicativi
  - I dati di ogni utente, ma anche quelli di sistema, devono essere protetti da accessi erronei o dolosi
- Vari tipi di meccanismi di protezione
  - meccanismi SW: es. protezione dei file (più avanti vedremo alcuni meccanismi a livello di SO)
  - meccanismi HW

# Protezioni hardware

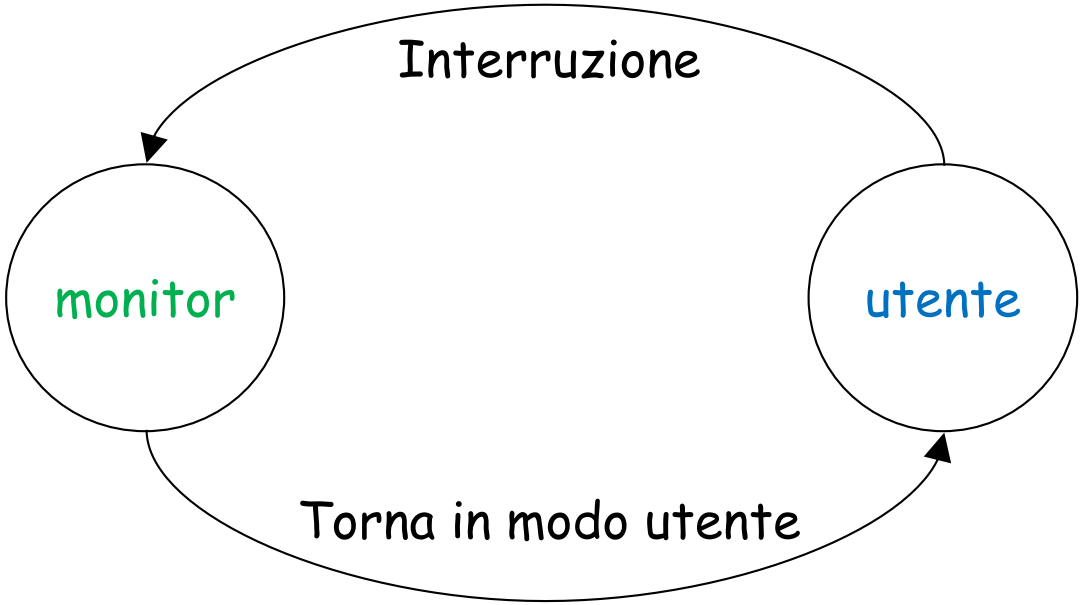
- Duplice modalità di funzionamento del processore
- Protezione dell'I/O
- Protezione della memoria
- Protezione della CPU

# Duplice modalità di funzionamento

- Due diverse (perlomeno) modalità di funzionamento del processore
  - modalità utente
    - usata per la normale esecuzione dei programmi
    - non è possibile accedere liberamente a tutte le risorse del sistema: le istruzioni privilegiate, es. modifica delle informazioni di protezione della memoria nei registri base e limite, non possono essere eseguite
  - modalità monitor (o kernel o supervisore o di sistema)
    - usata per l'esecuzione dei servizi richiesti al SO tramite system call
    - non esiste alcun limite alle operazioni effettuabili
- Per garantire una gestione delle risorse corretta da parte del SO è necessario
  - obbligare i processi utente ad accedere le risorse del calcolatore solo tramite system call
  - garantire che un processo utente non possa mai ottenere il controllo del calcolatore mentre questo si trova in modalità monitor

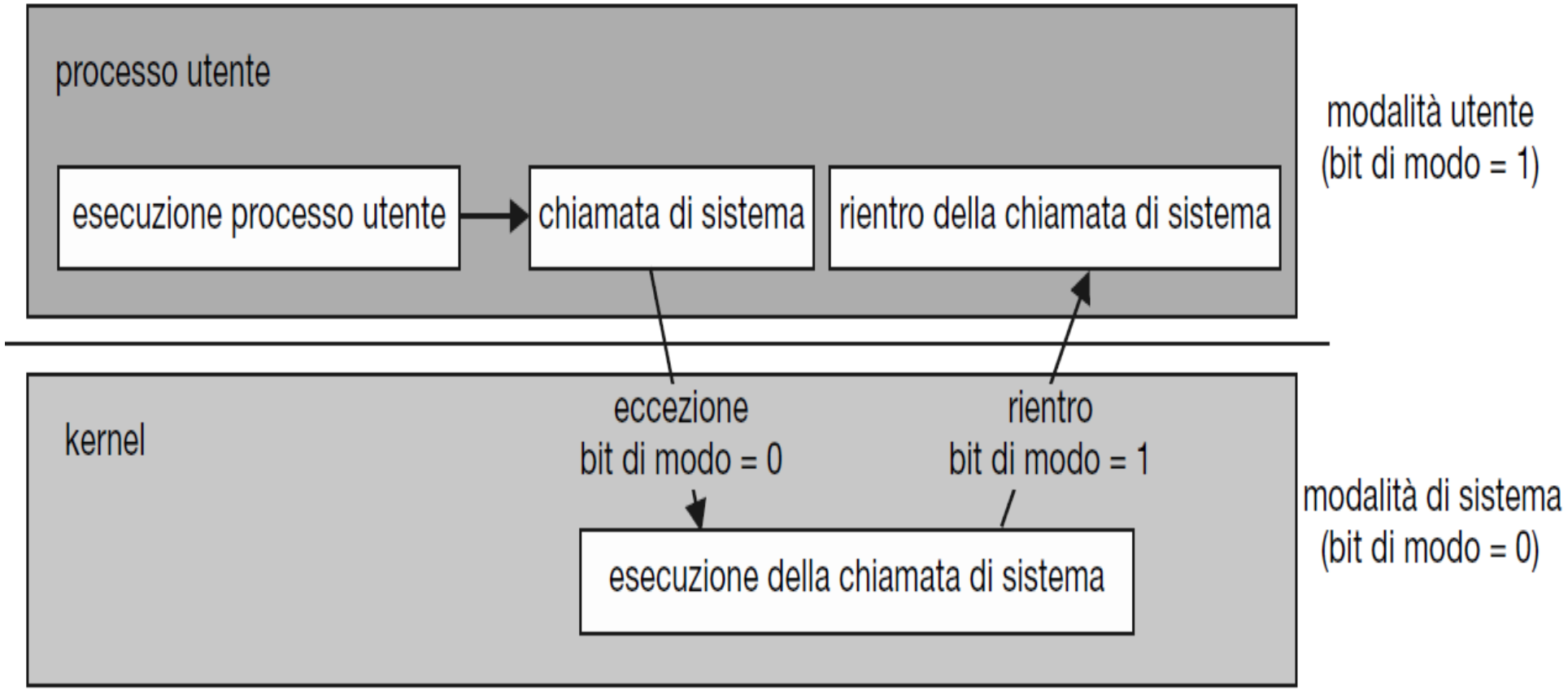
# Duplice modalità di funzionamento

- Un bit di modalità (fa parte del registro PS, *Program Status Word*) indica la modalità corrente:
  - monitor (0)
  - utente (1)
- Quando si verifica una interruzione, il sistema passa alla modalità monitor in modo da poter eseguire la relativa routine di gestione
- Dopo la gestione di una interruzione e prima di restituire il controllo ad un processo utente, il sistema passa alla modalità utente



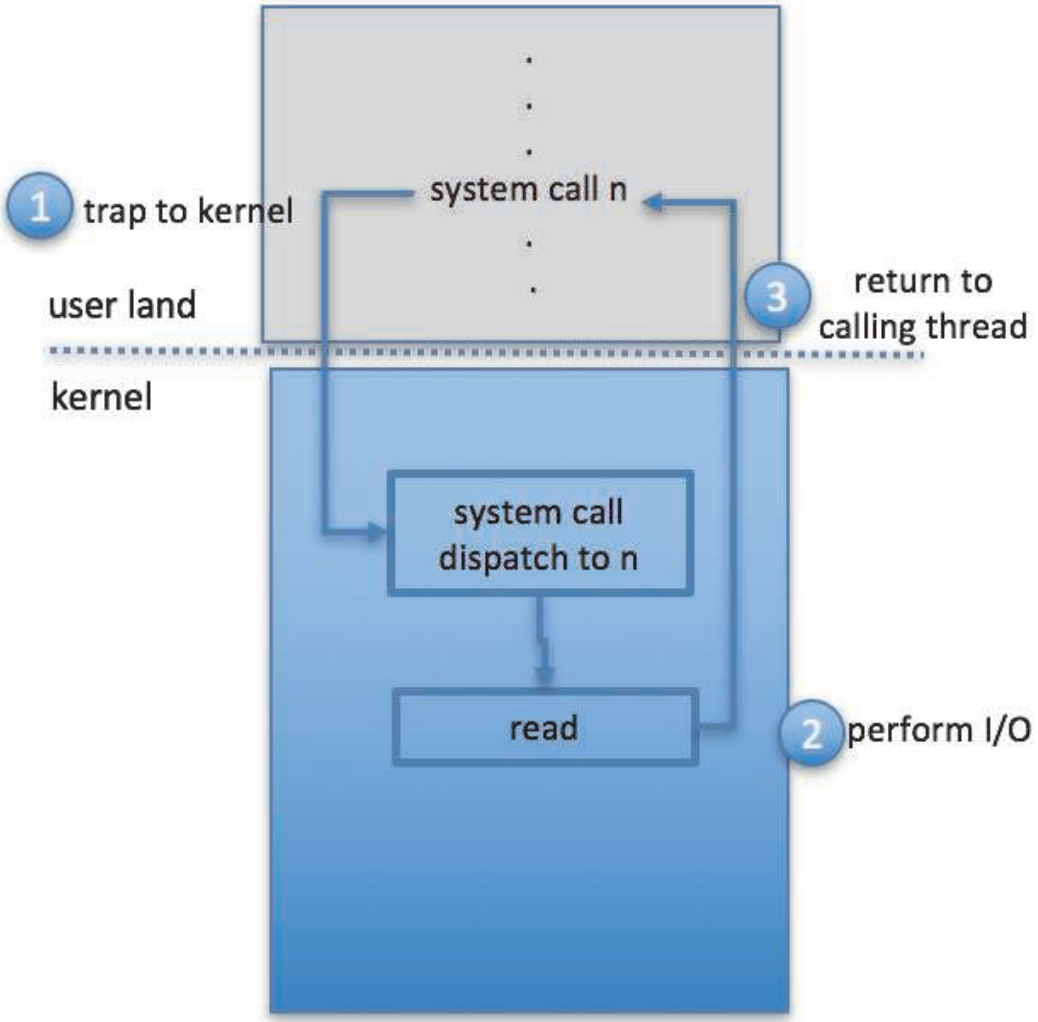
# Duplice modalità di funzionamento

Una system call genera una interruzione (che modifica la modalità di funzionamento della CPU e invoca l'esecuzione della funzione corrispondente)



# Protezione dell'I/O

- Le istruzioni di I/O sono privilegiate, così da impedire agli utenti di eseguirle direttamente
- Per eseguire operazioni di I/O, un programma utente invoca una system call affinché il SO esegua l'I/O per suo conto
- Il SO, lavorando in modalità kernel, verifica che la richiesta sia valida e, se lo è, esegue l'I/O richiesto e, alla fine, restituisce il controllo all'utente in modalità utente



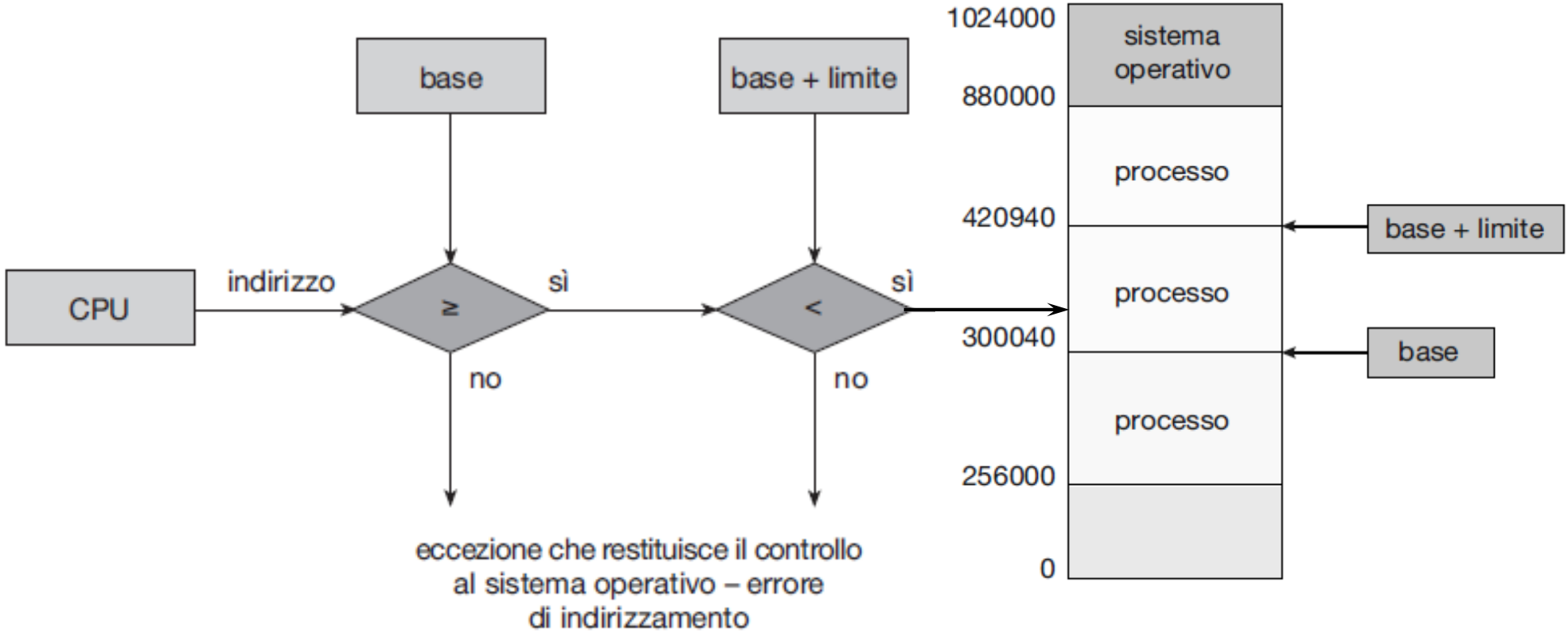
# Protezione della memoria

- Bisogna assicurare la protezione della memoria (principale), perlomeno per il vettore delle interruzioni e le funzioni di gestione delle interruzioni

  *Si pensi, ad esempio, ad un programma utente che memorizza un nuovo indirizzo nel vettore delle interruzioni: al momento in cui una interruzione si verifica, potrebbe far richiamare se stesso ottenendo così il controllo del sistema in modalità monitor*

- Un semplice supporto hardware per proteggere la memoria usa due registri che determinano l'intervallo di indirizzi legali che un processo può accedere:
  - registro base: contiene il più piccolo indirizzo della memoria fisica destinata al processo
  - registro limite: contiene la lunghezza dell'area di memoria riservata al processo

# Protezione della memoria




Ogni indirizzo generato dalla CPU è confrontato con il valore del registro base ed il valore ottenuto dalla somma dei due registri

- se l'indirizzo è maggiore o uguale al valore contenuto nel registro base e minore del valore dato dalla somma del contenuto del registro base e del registro limite, allora viene generata una richiesta di accesso alla memoria
- altrimenti, viene generata un'eccezione che restituisce il controllo al SO

# Protezione della memoria

- La memoria al di fuori dell'intervallo determinato dai due registri non è accessibile al programma ed è quindi protetta
- Solo il SO può modificare i registri base e limite poiché
  - I registri base e limite possono essere modificati solo tramite istruzioni privilegiate
  - Le istruzioni privilegiate possono essere eseguite solo in modalità kernel e solo il SO viene eseguito in modalità kernel
- Il SO, eseguito in modalità kernel, ha accesso illimitato sia alla memoria del SO che alla memoria riservata ai programmi utente, così il SO può
  - caricare i processi utente nelle aree di memoria ad essi riservate
  - effettuare copie del contenuto di queste regioni (*dump*) in caso di errori
  - accedere e modificare i parametri delle system call
  - eseguire operazioni di I/O da e verso la memoria utente
  - … e fornire molti altri servizi

# Protezione della memoria cache

- Alla cache L1 si accede solitamente tramite indirizzi logici (configurazione parallela della MMU e della cache L1)
- Un processo di lunghezza *n* utilizza gli indirizzi da 0 a *n-1*
  - Quindi, molti processi utilizzeranno gli stessi indirizzi logici
  - Perciò, un controllo basato sugli indirizzi logici non può essere utilizzato per decidere se un processo può accedere ad un valore presente nella cache (ogni blocco memorizza: <ind.logico,dato/istruzione>)
- Approccio semplice alla protezione della cache: cancellazione (flush) del contenuto di tutta la cache al momento in cui cambia il processo in esecuzione (context switch)
  - La cache contiene dati/istruzioni di un solo processo alla volta
  - Le prestazioni del processo che verrà eseguito saranno scarse a causa di un basso *hit rate*
- Approccio più sofisticato: in ogni blocco della cache si memorizza anche l'identificativo del processo cui appartengono i dati/istruzioni ivi contenuti e si permette l'accesso al blocco solo al processo in questione
  - Non richiede di svuotare la cache al context switch e non influisce sulle prestazioni dei processi

# Protezione della CPU

- Per impedire che un processo entri in un ciclo infinito ed assicurare che l'esecuzione non sfugga al controllo del SO si può usare un timer, che interrompe il processo in esecuzione dopo un periodo di tempo prefissato
  - Il timer viene impostato ogni volta che comincia l'esecuzione di un processo e viene decrementato ad ogni ciclo di clock
  - Quando il timer raggiunge il valore 0, si genera una interruzione
- Il timer è anche utilizzato per calcolare l'ora corrente
- L'istruzione per impostare il timer è privilegiata

# Concetti introduttivi

- Cosa fa un SO
- Organizzazione di un sistema di elaborazione
- Gestione delle risorse
- Virtualizzazione
- Sicurezza e protezione
- Servizi di un SO
- Interfacce utente
- System call & API
- Servizi di sistema
- Progetto e implementazione di un SO
- Struttura di un SO

# Gestione delle risorse

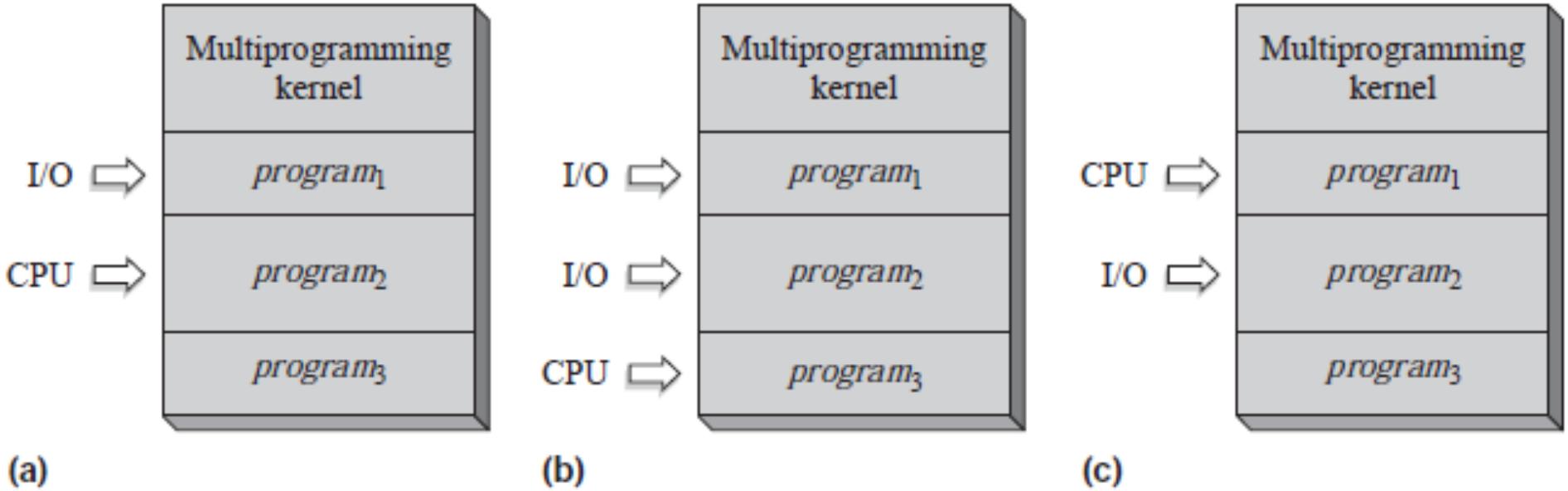
## I moderni SO

- Supportano la condivisione di alcune tipologie di risorse, in due diverse modalità
  - Condivisione nel tempo: programmi o utenti diversi usano la risorsa 'a turno' (es. CPU)
  - Condivisione nello spazio: programmi o utenti diversi usano ciascuno 'una parte' della risorsa (es. memoria principale)
- Utilizzano due importanti caratteristiche
  - Multiprogrammazione (o Multiplexing)
  - Time-sharing (o Multitasking)

# Multiprogrammazione

- Tecnica introdotta negli anni '60 per permettere un utilizzo efficiente delle risorse in un ambiente di elaborazione non interattivo
- Il SO organizza i processi in modo tale da mantenere la CPU in continua attività:
  - mantiene contemporaneamente in memoria centrale un insieme di processi (un sottinsieme di quelli memorizzati in memoria secondaria)
  - ne sceglie uno e lo esegue fino a che questo non ha bisogno di attendere il verificarsi di un qualche evento (es. il completamento di un I/O)
  - a questo punto passa ad eseguire un altro processo (context switch)
  - quando un processo sospeso ha terminato la sua attesa (es. l'evento atteso si è verificato), viene reinserito nell'insieme dei processi da eseguire
- Il SO passa dall'esecuzione di un processo ad un altro effettuando:
  - uno *scheduling dei processi* da caricare in memoria centrale
  - una *gestione della memoria* tra i vari processi
  - uno *scheduling della CPU* per scegliere il processo da eseguire
- Si basa su dispositivi hardware per
  - generare e gestire *segnali di interruzione*
  - *proteggere* dagli altri processi la memoria assegnata ai singoli processi

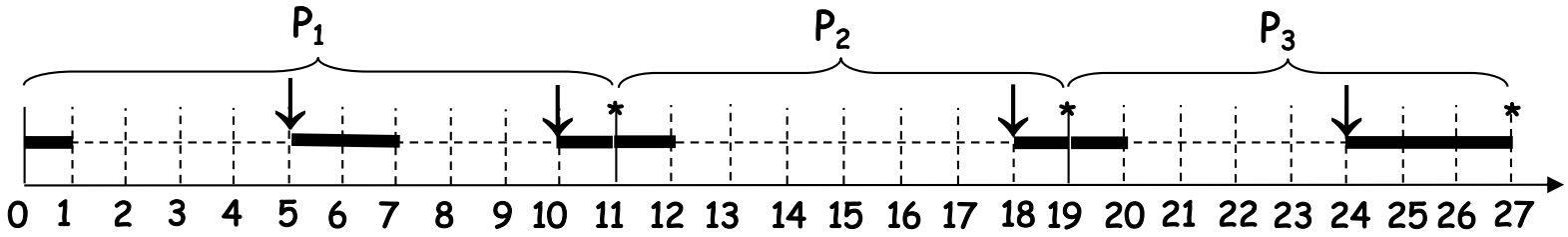
# Funzionamento di un sistema multiprogrammato




(a) $program_2$ è in esecuzione, $program_1$ è in attesa di un I/O
(b) $program_2$ avvia un'operazione di I/O, $program_3$ viene schedulato per l'esecuzione
(c) l'operazione di I/O di $program_1$ viene completata e il programma viene schedulato per l'esecuzione

# Funzionamento di un sistema multiprogrammato

Esecuzione sequenziale




↓ *arrivo di un segnale di interruzione*

* *terminazione di un processo*

Esecuzione multiprogrammata

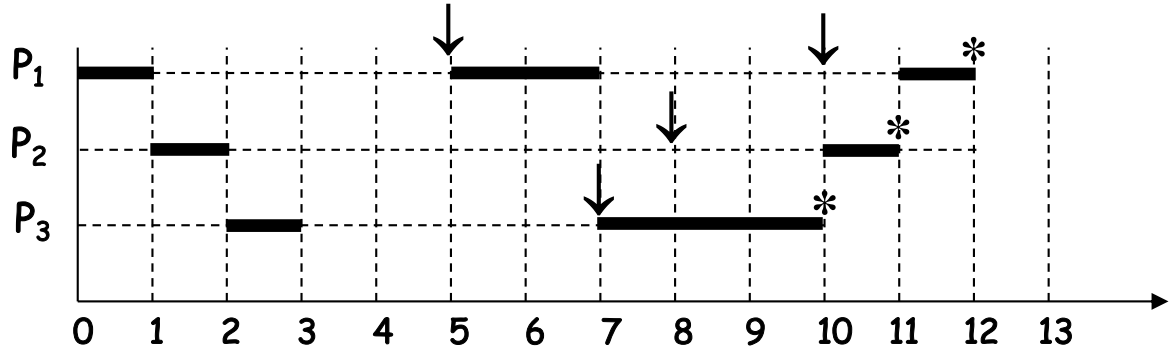


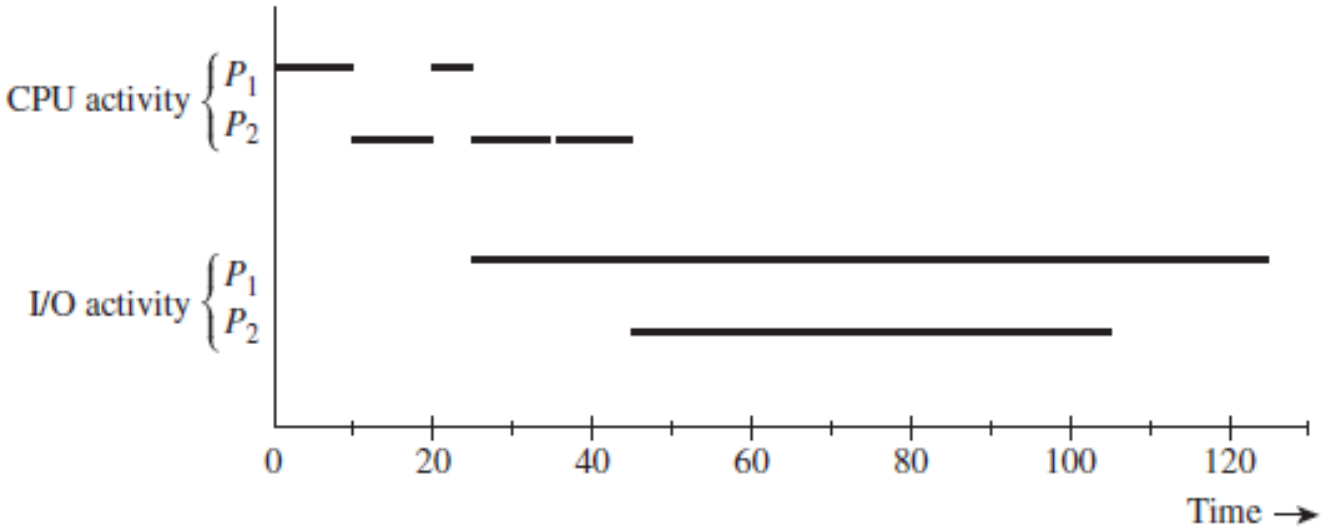

Diagramma di Gantt

# Time-sharing o Multitasking

- Estensione logica della multi-programmazione introdotta per
  - minimizzare i tempi di riposta dei programmi e, quindi,
  - garantire un uso interattivo del sistema di calcolo ad un costo ragionevole
- I programmi residenti in memoria centrale si alternano nell'uso della CPU con tempi molto brevi (dell'ordine dei millisecondi)
  - Il SO assegna la CPU ad ogni programma per un quanto di tempo predeterminato (time slice)
  - Ciascun utente ha l'illusione di avere l'intero sistema a sua disposizione
  - Sfrutta i "tempi morti" degli utenti
- Permette la comunicazione on-line tra utente e sistema
- Facilita la messa a punto dei programmi durante la stessa fase di esecuzione

# Funzionamento di un sistema Time-sharing

| Time | Scheduling list | Scheduled program | Remarks |
|---|---|---|---|
| 0 | $P_1, P_2$ | $P_1$ | $P_1$ is preempted at 10 ms |
| 10 | $P_2, P_1$ | $P_2$ | $P_2$ is preempted at 20 ms |
| 20 | $P_1, P_2$ | $P_1$ | $P_1$ starts I/O at 25 ms |
| 25 | $P_2$ | $P_2$ | $P_2$ is preempted at 35 ms |
| 35 | $P_2$ | $P_2$ | $P_2$ starts I/O at 45 ms |
| 45 | – | – | CPU is idle |



# Funzioni

Riguardo la gestione delle risorse, il SO svolge le seguenti funzioni:

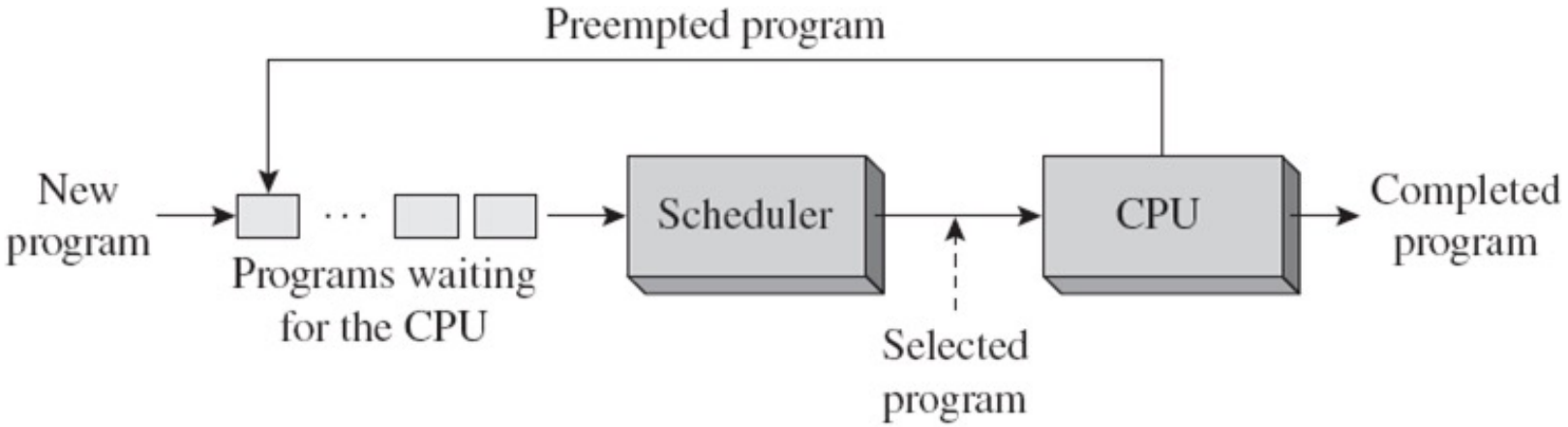
- Gestione dei processi
- Gestione della CPU
- Gestione della memoria principale
- Gestione dei file
- Gestione della memoria secondaria
- Gestione dei dispositivi di I/O

# Gestione dei processi

- Un programma in esecuzione è detto processo
  - Per compiere le sue attività, un processo necessita specifiche risorse (logiche e fisiche), tra cui tempo di CPU, memoria, file e dispositivi di I/O
- Il SO è responsabile delle seguenti attività
  - creazione ed eliminazione di processi
  - sospensione e ripresa dell'esecuzione di processi (quindi scheduling dei processi sulla CPU)
- Inoltre, il SO deve fornire meccanismi per
  - la sincronizzazione dei processi
  - la comunicazione fra processi
  - la gestione dei deadlock (stallo)

# Gestione della CPU

- In presenza di CPU con alte prestazioni, il SO può intervallare l'esecuzione di più processi
- Lo scheduler decide a quale processo assegnare la CPU in ogni istante di tempo




visione schematica dello scheduling della CPU con uno dei possibili algoritmi

- Le politiche (o algoritmi) di scheduling influenzano
  - l'efficienza di uso della CPU e
  - la qualità del servizio per l'utente

# Gestione della memoria principale

- La memoria principale è un vettore di grandi dimensioni di parole o di byte, ciascuna col proprio indirizzo
  - È un contenitore di dati rapidamente accessibili, condivisi da CPU e dispositivi di I/O
  - Il suo contenuto è volatile e si perde in caso di spegnimento o fallimento del sistema
- Il SO è responsabile delle seguenti attività
  - tenere traccia di quali parti di memoria sono correntemente usate e da quale processo
  - decidere quali processi portare in memoria (*caricare*) quando un'area di memoria diventa disponibile
  - allocare e deallocare spazio di memoria in base alle necessità
  - proteggere gli accessi alle locazioni di memoria

# Gestione dei file

- Un file è un insieme di informazioni correlate definite dal suo creatore
  - È un'unità di memorizzazione logica astratta rispetto alle caratteristiche dei dispositivi fisici di memorizzazione
  - Solitamente i file rappresentano programmi (in forma sorgente od oggetto), documenti o raccolte di dati
- Il SO è responsabile delle seguenti attività
  - creazione e cancellazione di file
  - creazione e cancellazione di directory per organizzare i file
  - supporto di primitive per manipolare file e directory
  - associazione dei file ai dispositivi di memoria secondaria
  - backup di file su dispositivi di memoria non volatili
  - controllo degli accessi allo scopo di garantire una protezione adeguata

# Gestione della memoria secondaria

- La maggior parte dei computer moderni usa dischi (magnetici o a stato solido) come dispositivi di memoria secondaria
  - La memoria secondaria funge da back-up della memoria principale, la quale è volatile e troppo piccola per contenere permanentemente tutti i dati e i programmi
- Il SO è responsabile delle seguenti attività
  - partizionamento della memoria secondaria
  - montaggio e smontaggio della memoria secondaria
  - gestione dello spazio libero
  - allocazione della memoria
  - scheduling delle richieste di operazione
  - protezione della memoria secondaria

# Gestione dei dispositivi di I/O

- Effettuata dal (sotto)sistema di I/O, che comprende due componenti:
  - un'interfaccia generale per i driver dei dispositivi
  - un driver per ogni specifico dispositivo
- Il SO è responsabile delle seguenti attività
  - mascherare la diversità e la complessità dei vari dispositivi
  - gestire la competizione dei processi nell'uso dei dispositivi
  - gestire l'I/O delle informazioni, ivi compreso
    - *buffering* (immagazzinamento temporaneo dei dati in memorie tampone durante il trasferimento)
    - *caching* (immagazzinamento di parte dei dati in memorie veloci per migliorare le prestazioni)
    - *spooling* (sovrapposizione dell'I/O di alcuni processi con l'esecuzione di altri)

# Concetti introduttivi

- Cosa fa un SO
- Organizzazione di un sistema di elaborazione
- Gestione delle risorse
- Virtualizzazione
- Sicurezza e protezione
- Servizi di un SO
- Interfacce utente
- System call & API
- Servizi di sistema
- Progetto e implementazione di un SO
- Struttura di un SO

# Virtualizzazione delle risorse

- Una *risorsa virtuale* è una risorsa fittizia
  - Astrazione di una risorsa da un punto di vista dei programmi
  - Visione supportata dal SO tramite l'uso di una risorsa reale
  - Una stessa risorsa reale può supportare parecchie risorse virtuali
- La corrispondenza tra risorse virtuali e risorse reali è mantenuta dal SO in modo *trasparente* (mascherandone la struttura)
- La virtualizzazione delle risorse è utile per vari motivi
  - All'utente ed alle applicazioni vengono mostrate *risorse virtuali*, più semplici da usare o in numero maggiore rispetto alle *risorse reali*
  - La disponibilità di una molteplicità di risorse virtuali, rimuove i vincoli di uso esclusivo e favorisce l'esecuzione concorrente di più applicazioni
    - Nella maggioranza dei casi le risorse reali possono in effetti essere assegnate solo in *uso esclusivo*, limitando così il parallelismo nell'esecuzione delle applicazioni

# Esempio: server di stampa

- Un server di stampa è un esempio comune di risorsa virtuale
  - ***Risorsa reale*** ⇒ stampante
  - ***Risorsa virtuale*** ⇒ server di stampa (componente del SO)

- Quando un programma chiede di stampare un file, il server di stampa semplicemente copia il file nella "coda di stampa"
- Il programma che ha richiesto la stampa può continuare la sua esecuzione come se la stampa fosse stata effettuata
- Il server di stampa esamina di continuo la coda di stampa e, man mano che la stampante reale è disponibile, stampa i file che trova in coda

- Effetto: l'accesso alla stampante è semplificato ed il vincolo di uso esclusivo è apparentemente rimosso

# Virtualizzazione delle risorse

Può essere utilizzata a vari livelli e per diverse tipologie di risorse

- CPU: Tramite la *multiprogrammazione* più programmi condividono l'uso della CPU, alternandosi
- Memoria: Tramite la *memoria virtuale* lo spazio di memoria disponibile per i programmi non è limitato dalla memoria fisica
- I/O: L'accesso ai dispositivi di I/O è semplificato ed il vincolo di uso esclusivo è rimosso
- Computer: Alcuni SO supportano *macchine virtuali* ciascuna delle quali può essere allocata ad un utente
  - Evita le mutue interferenze tra gli utenti
  - Consente di usare SO diversi simultaneamente sullo stesso calcolatore
  - Permette a un SO di essere eseguito all'interno di un altro SO, come una applicazione

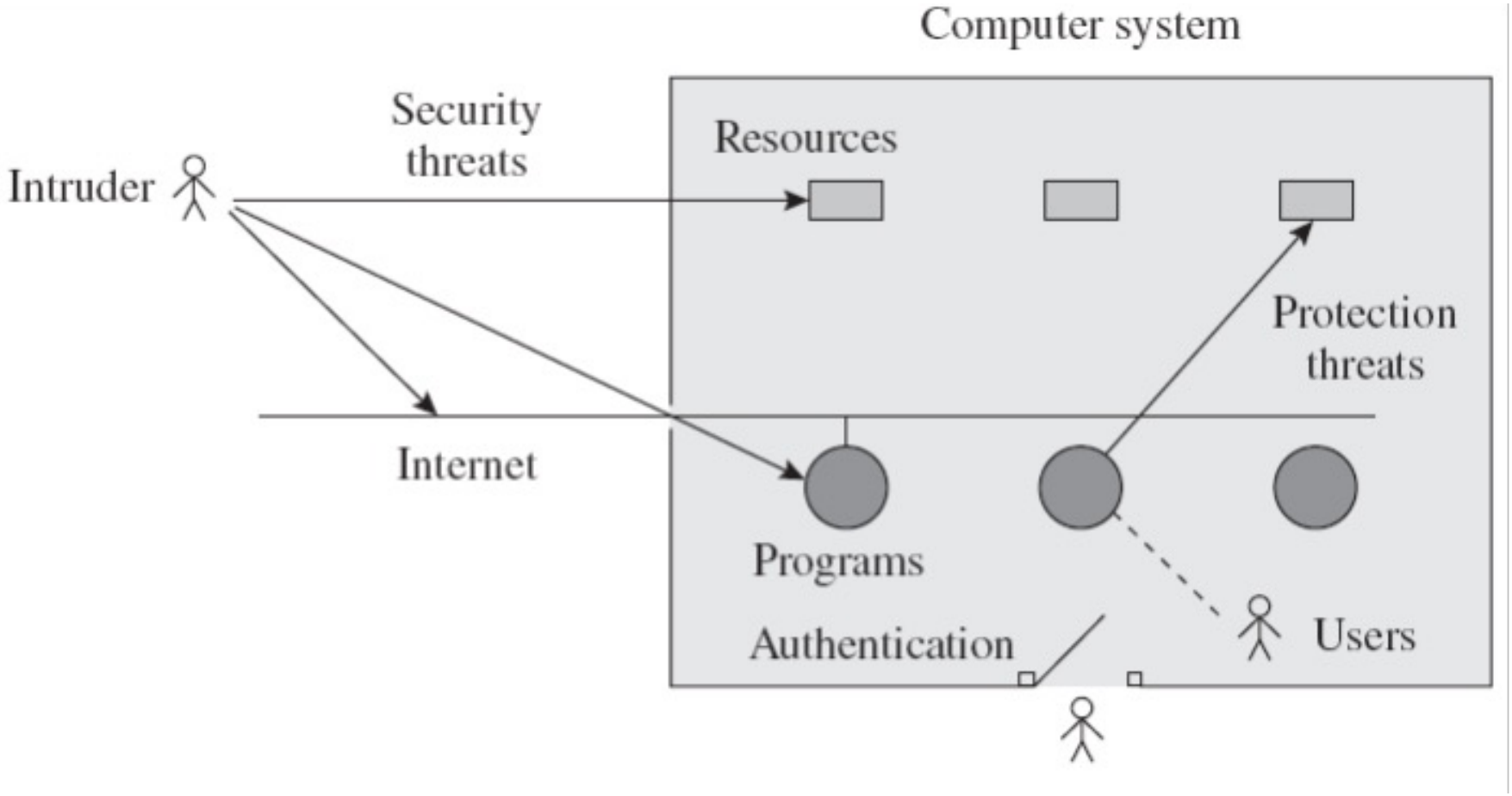
# Concetti introduttivi

- Cosa fa un SO
- Organizzazione di un sistema di elaborazione
- Gestione delle risorse
- Virtualizzazione
- Sicurezza e protezione
- Servizi di un SO
- Interfacce utente
- System call & API
- Servizi di sistema
- Progetto e implementazione di un SO
- Struttura di un SO

# Sicurezza e protezione

- La sicurezza si occupa di preservare le risorse del computer da accessi non autorizzati, da modifiche intenzionali dannose o distruttive e dall'introduzione accidentale di incoerenze
  - Riguarda in particolare l'uso illegale o le interferenze operate da utenti/programmi fuori dal controllo del SO (intruder, estranei al sistema)
  - Si basa sull'autenticazione (*processo di verifica dell'identità di utenti o applicazioni*): solo gli utenti registrati possono usare il sistema di calcolo
- La protezione è l'insieme dei meccanismi che controllano l'accesso dei processi e degli utenti alle risorse di un sistema informatico
  - Si basa sull'autorizzazione (*processo di verifica di ciò che gli utenti autenticati sono autorizzati a fare*): solo gli utenti che dispongono di autorità sufficiente possono accedere ad una risorsa in una certa modalità
  - Vari meccanismi: protezione della memoria, protezione della CPU, duplice modalità di funzionamento della CPU, controllo accesso ai file, …

# Minacce a Sicurezza e Protezione




Ampia tipologia di attacchi (sia esterni che interni):
denial-of-service, phishing, furto di identità o di servizio, …

# Sicurezza e protezione

Il SO è responsabile delle seguenti attività

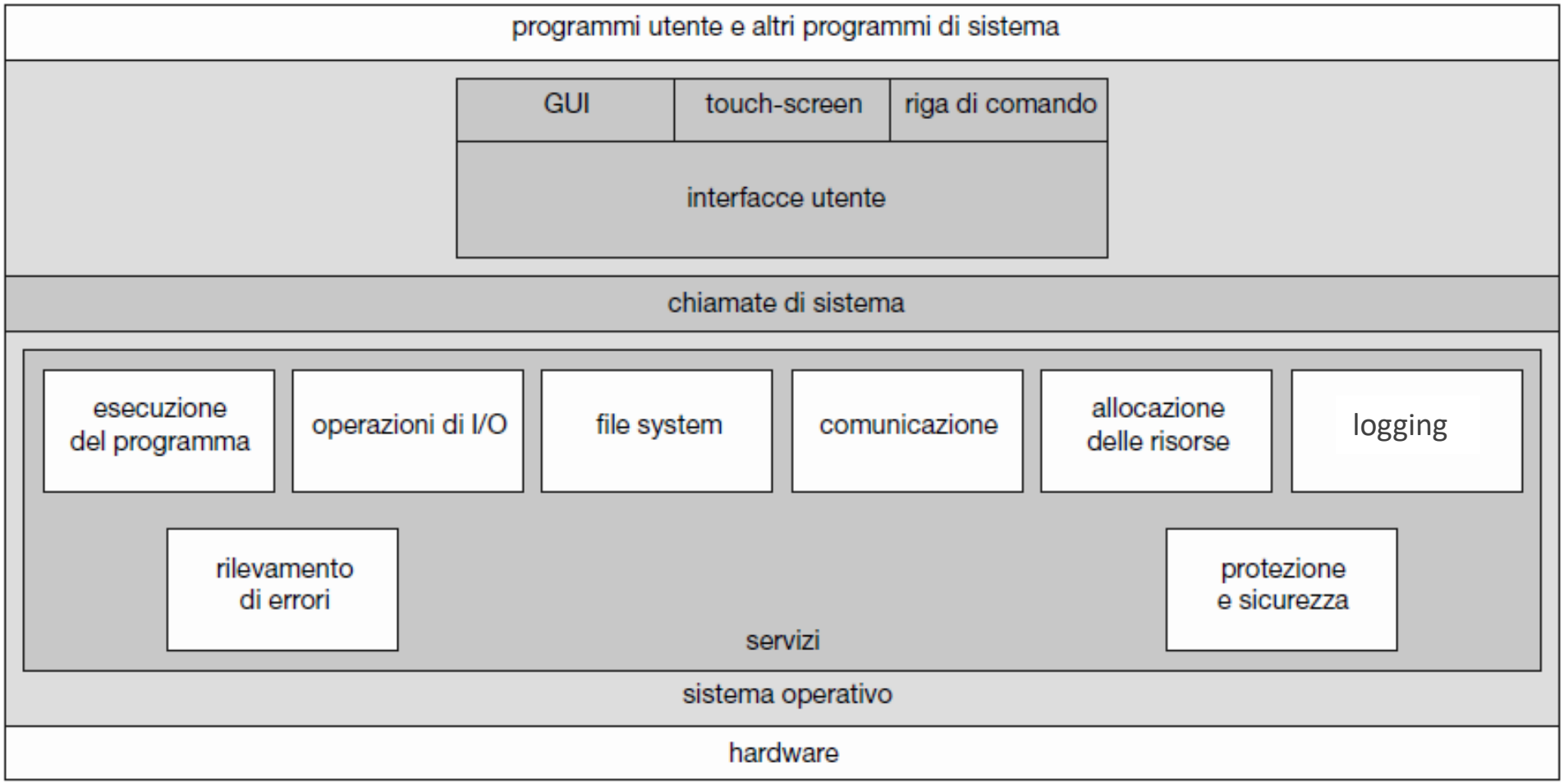
- Autenticazione degli utenti: solo gli utenti registrati possono usare il sistema
  - Ciascun utente ha un proprio user ID che lo identifica
  - Lo user ID di un utente è quindi associato a tutti i file e processi di quell'utente
  - In aggiunta, i group ID permettono di definire insiemi di utenti e sono a loro volta associati ad ogni processo o file
- Autorizzazione degli utenti autenticati all'uso delle risorse; richiede meccanismi per:
  - specificare quali controlli debbano essere effettuati
  - distinguere tra uso autorizzato e non autorizzato
  - fornire metodi per imporre i criteri stabiliti

# Concetti introduttivi

- Cosa fa un SO
- Organizzazione di un sistema di elaborazione
- Gestione delle risorse
- Virtualizzazione
- Sicurezza e protezione
- Servizi di un SO
- Interfacce utente
- System call & API
- Servizi di sistema
- Progetto e implementazione di un SO
- Struttura di un SO

# Panoramica dei servizi di un SO

- L'accesso ai servizi forniti dal SO avviene solitamente tramite interfacce utente di vario tipo e chiamate di sistema
- I servizi specifici forniti differiscono da un SO all'altro, ma possiamo identificare alcune classi comuni



# Servizi di un SO

Alcuni servizi forniscono all'utente funzioni che facilitano l'utilizzo del calcolatore

- Esecuzione di programmi
- Esecuzione di operazioni di I/O
- Gestione del file system
- Comunicazioni
- Rilevamento di errori

# Servizi di un SO

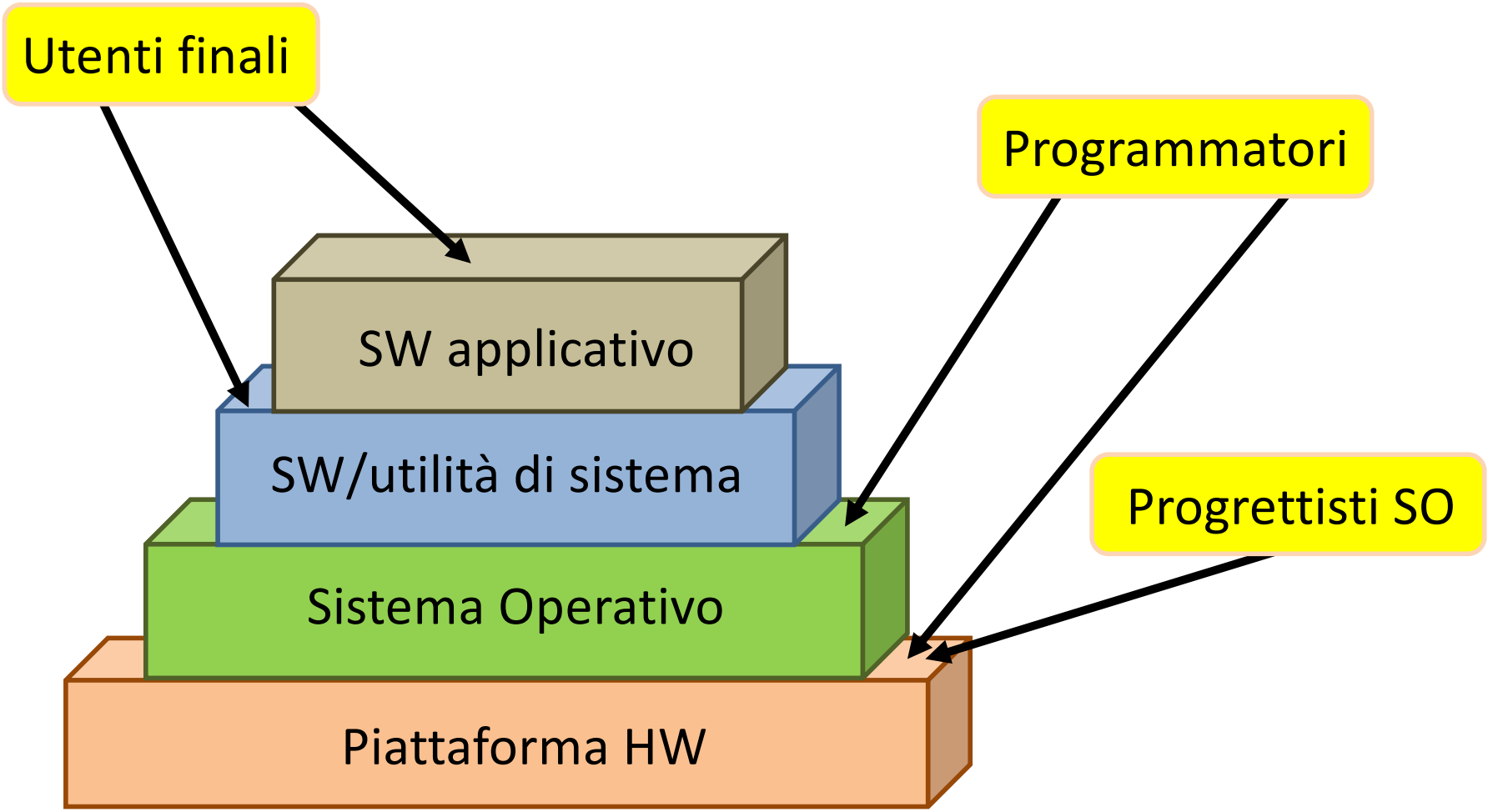
Altri servizi garantiscono un funzionamento efficiente del sistema stesso sfruttando la condivisione delle risorse tra i processi

- Allocazione delle risorse
- Logging, per tenere traccia di quali programmi utilizzano quante e quali tipi di risorse del computer
- Protezione e sicurezza

# Concetti introduttivi

- Cosa fa un SO
- Organizzazione di un sistema di elaborazione
- Gestione delle risorse
- Virtualizzazione
- Sicurezza e protezione
- Servizi di un SO
- Interfacce utente
- System call & API
- Servizi di sistema
- Progetto e implementazione di un SO
- Struttura di un SO

# Schema delle interazioni tra componenti HW, SW e utenti di un sistema di elaborazione



# Interfacce utente

- I livelli si collegano ciascuno con quello sovrastante tramite interfacce le quali offrono le funzionalità del livello inferiore a quello superiore, il quale a sua volta fornisce funzionalità più astratte che semplificano ulteriormente l'uso del sistema per gli utenti
- Gli utenti finali usano il sistema tramite il SW applicativo o il SW di sistema (*interpreti, compilatori, linker, …*)
- Gli utenti programmatori possono sfruttare anche funzionalità più potenti messe a disposizione nei livelli inferiori
  - I programmatori di applicazioni si interfacciano col sistema tramite l'interfaccia del SO costituita dalle system call e dalle API
  - I programmatori di sistema usano le funzionalità messe a disposizione dal SO e anche direttamente dall'HW
- I progettisti del SO si interfacciano direttamente con l'HW

# Interprete di comandi

- È il SW di sistema più importante
- Legge ed interpreta le istruzioni del linguaggio di controllo del SO costituito dai vari comandi forniti dal SO per:
  - creazione e manipolazione dei processi
  - gestione dell'I/O
  - gestione della memoria principale
  - gestione della memoria secondaria
  - accesso al file-system
  - protezione
  - networking

# Interprete di comandi

- A riga di comando (command-line interface, shell in UNIX)
  - Interpreta ed esegue un commando alla volta, inserito dall'utente tramite il cosidetto prompt dei comandi
  - Può eseguire anche file batch o script (file di testo che contengono una serie di comandi che saranno interpretati ed eseguiti sequenzialmente)
- Interfaccia grafica (Graphical User Interface, GUI)
- Interfaccia touch-screen

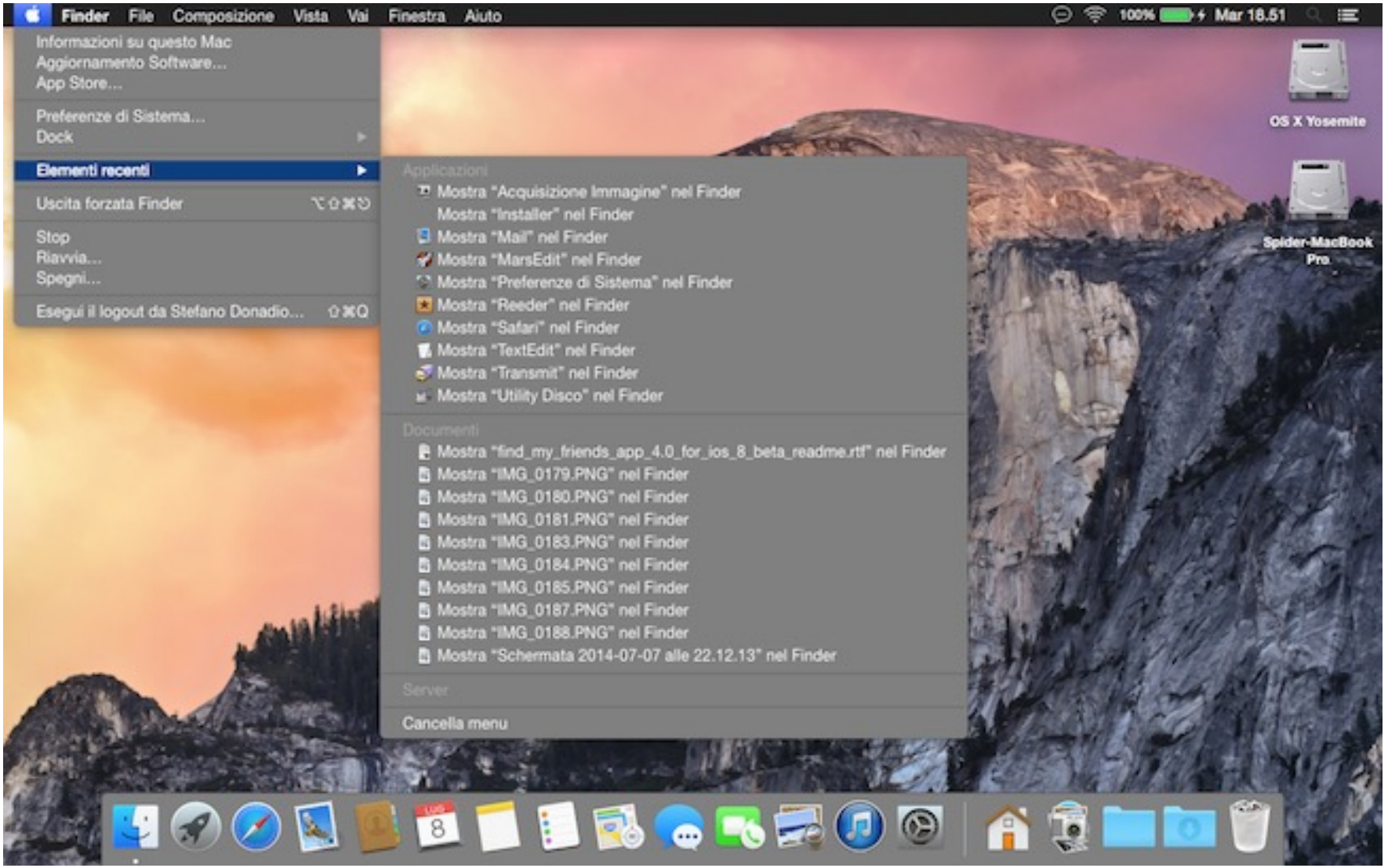
# Shell Bash: un interprete di comandi utilizzato in Linux

```
bash-3.2$ cat append_date.sh
#!/bin/sh

file_name=test_files.txt

current_time=$(date "+%Y.%m.%d-%H.%M.%S")
echo "Current Time : $current_time"

new_fileName=$file_name.$current_time
echo "New FileName: " "$new_fileName"

cp $file_name $new_fileName
echo "You should see new file generated with timestamp on it.."


bash-3.2$ ./append_date.sh
Current Time : 2014.12.15-10.31.42
New FileName:  test_files.txt.2014.12.15-10.31.42
You should see new file generated with timestamp on it..
bash-3.2$
bash-3.2$ ls test_files.txt*
test_files.txt                    test_files.txt.2014.12.15-10.31.42
bash-3.2$
```

Crunchify.com

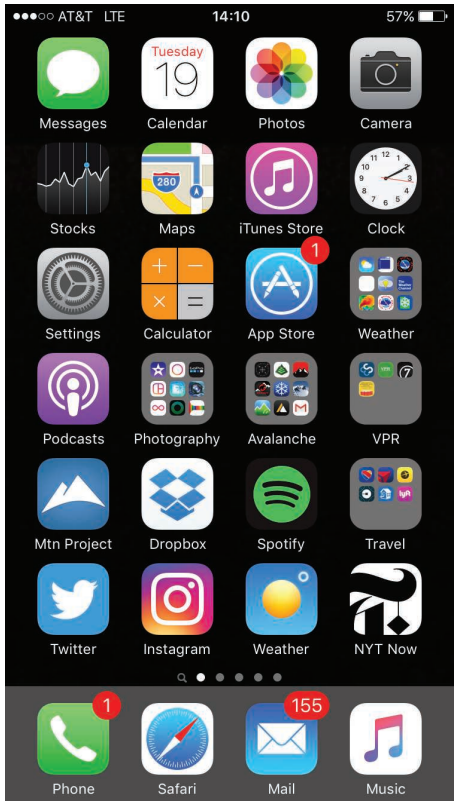
# Interfaccia grafica (GUI) di Mac OS X

(GUI con l'uso di dispositivi di puntamento: *mouse, trackpad, …*)



# Interfaccia touch-screen

## iPad Pro




## iPhone

# Concetti introduttivi

- Cosa fa un SO
- Organizzazione di un sistema di elaborazione
- Gestione delle risorse
- Virtualizzazione
- Sicurezza e protezione
- Servizi di un SO
- Interfacce utente
- System call & API
- Servizi di sistema
- Progetto e implementazione di un SO
- Struttura di un SO

# System Call

- Richieste che un programma rivolge al (kernel del) SO attraverso interruzioni software
- Costituiscono l'interfaccia fra programma in esecuzione e servizi offerti dal SO

# Tipi di System Call

- Controllo dei processi

  creazione/terminazione, caricamento/esecuzione, attesa/segnalazione evento, assegnazione/rilascio memoria, esame/impostazione attributi, …
- Gestione dei file

  creazione/cancellazione, apertura/chiusura, lettura/scrittura, esame /impostazione attributi, …
- Gestione dei dispositivi

  richiesta/rilascio dispositivi, lettura/scrittura, (s)collegamento logico dei dispositivi (e.g. (un)mounting), esame/impostazione attributi, …
- Gestione delle informazioni di sistema

  lettura/impostazione data e ora del sistema, esame informazioni su HW/SW installato, esame informazioni su utenti collegati, …
- Realizzazione delle comunicazioni

  creazione/eliminazione connessioni, invio/ricezione messaggi, esame stato trasferimento, …
- Protezione

  esame/impostazione permessi sui file, modifica proprietario/gruppo, …

# Uso delle System call

Illustriamo come le system call sono usate considerando la sequenza di system call necessaria per leggere i dati da un file e copiarli in un altro file, come fa il commando UNIX

```
cp in.txt out.txt
```

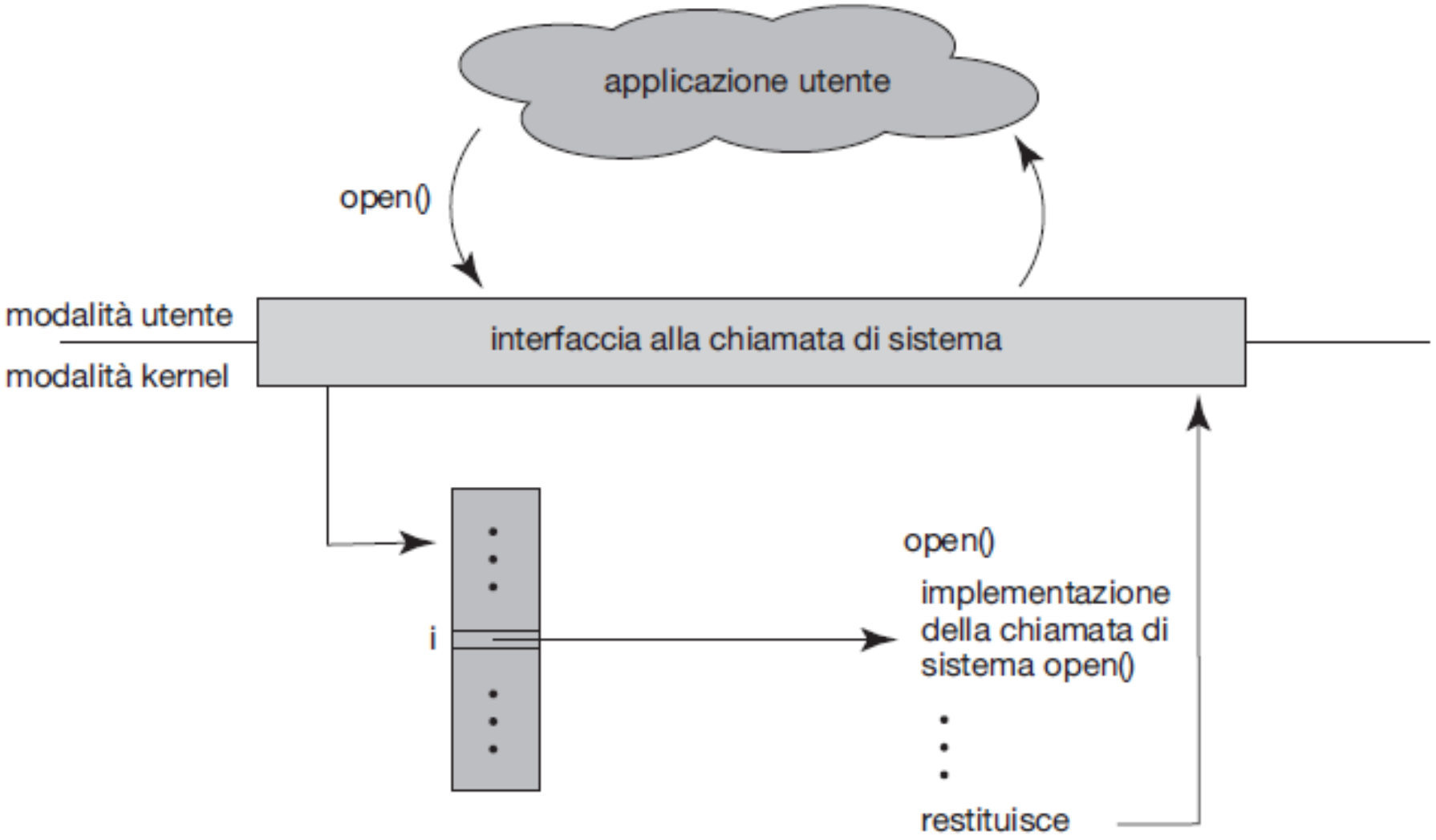

Esempio di sequenza di chiamate di sistema

```
Acquisisce il nome del file in ingresso
  Scrive messaggio di richiesta sullo schermo
  Accetta i dati in ingresso
Acquisisce il nome del file in uscita
  Scrive messaggio di richiesta sullo schermo
  Accetta i dati in ingresso
Apre il file in ingresso
  Se il file non esiste, termina con errore
Crea il file in uscita
  Se il file esiste, termina con errore
Ripete
  Legge dal file in ingresso
  Scrive sul file in uscita
Finché c'è ancora da leggere
Chiude il file in uscita
Scrive messaggio sullo schermo per informare del completamento
Termina senza errori
```

# Implementazione delle System Call

- Ad ogni system call è associato un numero intero
  - È l'indice in una tabella (simile al vettore delle interruzioni) che contiene le informazioni (tra cui l'indirizzo della prima istruzione della sequenza che implementa la system call) riguardo l'implementazione della system call
- L'interfaccia verso le system call
  - gestisce la tabella delle system call indicizzata in base al numero associato alle system call
  - invoca la system call desiderata nel kernel del SO e, alla sua terminazione, restituisce lo stato della system call e i valori di ritorno
- A seconda del SO può essere necessario utilizzare ulteriori informazioni, oltre al numero che identifica una system call
- L'invocante non ha bisogno di conoscere i dettagli dell'implementazione della system call
  - Deve solo usare correttamente la system call, o le funzioni della API che la richiamano, e sapere cosa fa il SO a seguito di tali invocazioni

# Gestione della system call `open()` invocata da un'applicazione utente

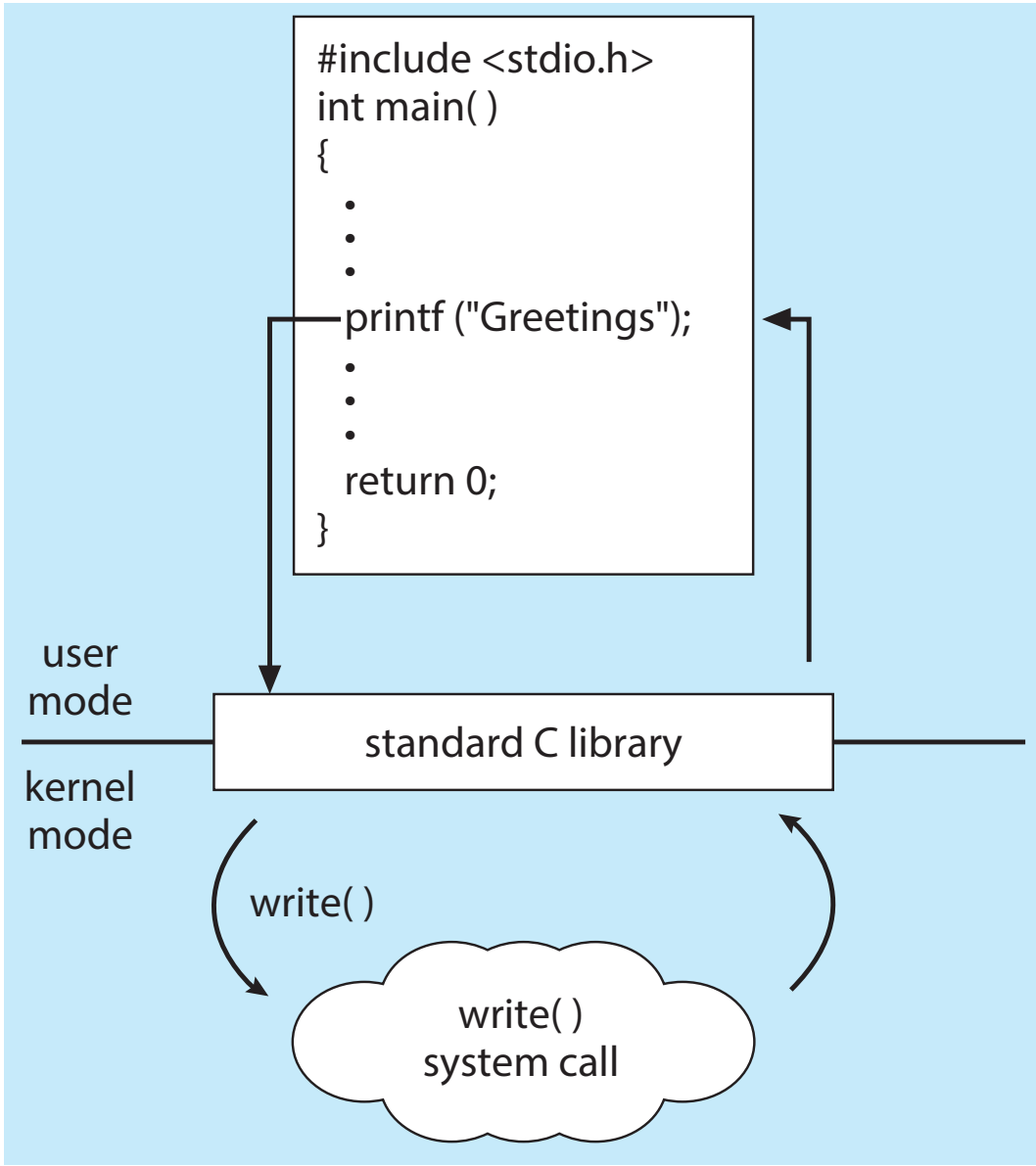


# System call & API

- Le system call sono spesso scritte in linguaggio assembler, ma talvolta sono disponibili in linguaggi di alto livello progettati per la programmazione di sistema *(es. C, Bliss, PL/360)*
- Solitamente *non* sono accedute direttamente dai programmi, ma tramite una interfaccia di più alto livello detta Application Programming Interface (API)
- Tre API molto comuni sono
  - POSIX API per sistemi basati sullo standard POSIX (virtualmente tutte le versioni di UNIX, Linux, e Mac OS X)
  - Win32 API per Windows
  - Java API per la Java Virtual Machine (JVM)

# La libreria C standard

- La libreria C standard (`glibc`) fornisce parte dell'interfaccia per le system call in molte versioni di UNIX e Linux
- Ad esempio, supponiamo che un programma C invochi l'istruzione `printf()`
- La libreria C intercetta questa chiamata e richiama la (o le) system call necessaria nel SO, in questo caso la system call `write()`
- Successivamente, la libreria C accetta il valore restituito da `write()` e lo restituisce al processo utente



# Lo standard POSIX
## (Portable OS Interface for Unix)

- POSIX è il nome che indica la famiglia degli standard definiti dall'IEEE denominati formalmente IEEE 1003
  - Il nome standard internazionale è ISO/IEC 9945
- POSIX specifica un insieme di procedure (circa 100) che un sistema compatibile deve fornire
  - Non specifica se siano system call, chiamate di librearia o altro
  - La corrispondenza tra procedure e system call non è di uno-a-uno
  - Se una procedura può essere realizzata senza richiamare una system call (quindi senza fare una trap nel kernel) essa sarà eseguita nello spazio utente per motivi prestazionali
- Molti SO sono compatibili con POSIX (POSIX compliant)
- La compatibilità con POSIX garantisce la portabilità delle applicazioni

# POSIX: controllo dei processi

| Call | Description |
|---|---|
| pid = fork( ) | Create a child process identical to the parent |
| pid = waitpid(pid, &statloc, options) | Wait for a child to terminate |
| s = execve(name, argv, environp) | Replace a process' core image |
| exit(status) | Terminate process execution and return status |

- Sono tra le principali chiamate di sistema
- Corrispondono al classico meccanismo UNIX di creazione dei processi
- Un processo genera una sua replica (*figlio*) tramite la `fork`
- Il figlio muta il suo comportamento tramite la `execve`

# POSIX: gestione dei file

| Call | Description |
|---|---|
| fd = open(file, how, ...) | Open a file for reading, writing, or both |
| s = close(fd) | Close an open file |
| n = read(fd, buffer, nbytes) | Read data from a file into a buffer |
| n = write(fd, buffer, nbytes) | Write data from a buffer into a file |
| position = lseek(fd, offset, whence) | Move the file pointer |
| s = stat(name, &buf) | Get a file's status information |

# POSIX: gestione del file system

| Call | Description |
|---|---|
| s = mkdir(name, mode) | Create a new directory |
| s = rmdir(name) | Remove an empty directory |
| s = link(name1, name2) | Create a new entry, name2, pointing to name1 |
| s = unlink(name) | Remove a directory entry |
| s = mount(special, name, flag) | Mount a file system |
| s = umount(special) | Unmount a file system |

# POSIX: chiamate varie

| Call | Description |
|---|---|
| s = chdir(dirname) | Change the working directory |
| s = chmod(name, mode) | Change a file's protection bits |
| s = kill(pid, signal) | Send a signal to a process |
| seconds = time(&seconds) | Get the elapsed time since Jan. 1, 1970 |

- La `kill` permette di mandare 'segnali' ai processi
- Se il processo destinatario è in attesa di un segnale, questo gli viene comunicato e la sua esecuzione riprende
- Se il destinatario non è in attesa viene 'ucciso': da tale effetto deriva il macabro nome della system call
- Es. `kill(pid, SIGINT)` vs.
  `kill -9 pid` o `kill -KILL pid` (comando)

# Win32 API

- Win32 API è la libreria di Windows tramite la quale si accede alle system call
- Supportata (parzialmente) da tutte le versioni di Windows
- Comprende molte migliaia di chiamate
- Non a tutte le procedure, in tutte le implementazioni, corrispondono in realtà chiamate di sistema
- Molte chiamate non coinvolgono il kernel, cioè vengono gestite nello *user space*
- Molte chiamate sono relative alla GUI (Graphical User Interface): creazione di finestre, menu, scrollbar, ecc.

# Windows vs. UNIX System Call

| | Windows | UNIX |
|---|---|---|
| **Controllo dei processi** | `CreateProcess()` | `fork()` |
| | `ExitProcess()` | `exit()` |
| | `WaitForSingleObject()` | `wait()` |
| **Gestione dei file** | `CreateFile()` | `open()` |
| | `ReadFile()` | `read()` |
| | `WriteFile()` | `write()` |
| | `CloseHandle()` | `close()` |
| **Gestione dei dispositivi** | `SetConsoleMode()` | `ioctl()` |
| | `ReadConsole()` | `read()` |
| | `WriteConsole()` | `write()` |
| **Gestione delle informazioni** | `GetCurrentProcessID()` | `getpid()` |
| | `SetTimer()` | `alarm()` |
| | `Sleep()` | `sleep()` |
| **Comunicazione** | `CreatePipe()` | `pipe()` |
| | `CreateFileMapping()` | `shm_open()` |
| | `MapViewOfFile()` | `mmap()` |
| **Protezione** | `SetFileSecurity()` | `chmod()` |
| | `InitializeSecurityDescriptor()` | `umask()` |
| | `SetSecurityDescriptorGroup()` | `chown()` |

# Vantaggi delle API

Perché usare le API anziché le system call direttamente?

- Portabilità delle applicazioni: un programma sviluppato usando una certa API girerà su qualunque sistema che metta a disposizione quella API
  - Anche se in pratica le differenze architetturali possono rendere il porting non del tutto agevole
- Facilità d'uso: le system call sono in generale più di basso livello e difficili da usare rispetto ad una API
  - Anche se solitamente c'è una stretta corrispondenza tra le funzioni di una API e le system call del kernel corrispondenti

# Perché le applicazioni dipendono dal SO

- Fondamentalmente le applicazioni compilate su un SO *non sono eseguibili* su altri sistemi operativi
  - Ogni SO fornisce un insieme univoco di chiamate di sistema
- Tre modi per consentire a un'applicazione di essere resa disponibile per l'esecuzione su più SO
  - Può essere scritta in un linguaggio interpretato (es. Python) che ha un interprete disponibile per più SO
  - Può essere scritta in un linguaggio (es. java) che utilizza una macchina virtuale, disponibile per più SO, che poi esegue l'applicazione
  - Può essere scritta in un linguaggio standard (es. C) ed usare una API standard (es. POSIX) per poi essere compilata in un linguaggio specifico per la combinazione HW/SO

# Concetti introduttivi

- Cosa fa un SO
- Organizzazione di un sistema di elaborazione
- Gestione delle risorse
- Virtualizzazione
- Sicurezza e protezione
- Servizi di un SO
- Interfacce utente
- System call & API
- Servizi di sistema
- Progetto e implementazione di un SO
- Struttura di un SO

# Servizi di Sistema (programmi di sistema o utilità di sistema)

- Sono associati al SO ma non fanno necessariamente parte del kernel
- Forniscono un ambiente per lo sviluppo e l'esecuzione dei programmi utente
- Si trovano ad un livello più astratto rispetto alle system call e le API
- Alcuni costituiscono delle interfacce semplicate per le system call, altri sono più complessi
  - Per la maggior parte degli utenti la visione del SO risulta definita dai servizi di sistema, piuttosto che dalle effettive system call

# Servizi di sistema

Varie categorie sulla base del loro utilizzo

- Gestione dei file e directory (creare, cancellare, copiare, spostare,…)
- Ottenere info sullo stato del sistema e sugli utenti
- Modifiche dei file (editor)
- Supporto ai linguaggi di programmazione (compilatori, interpreti, debugger, …)
- Caricamento ed esecuzione dei programmi (loader, linker)
- Supporto alle comunicazioni (rlogin, ftp, email, …)

Alcuni programmi di sistema (detti daemon o server) vengono lanciati al momento dell'avvio del sistema e restano in esecuzione fino a quando il sistema viene arrestato: es. server di stampa, server di rete, ...

# Concetti introduttivi

- Cosa fa un SO
- Organizzazione di un sistema di elaborazione
- Gestione delle risorse
- Virtualizzazione
- Sicurezza e protezione
- Servizi di un SO
- Interfacce utente
- System call & API
- Servizi di sistema
- Progetto e implementazione di un SO
- Struttura di un SO

# Scopi del progetto di un SO

- Da un punto di vista utente, un SO dovrebbe essere
  - conveniente da usare,
  - facile da imparare,
  - affidabile, sicuro e veloce
- Da un punto di vista del sistema di elaborazione, un SO dovrebbe essere
  - facile da progettare, implementare e manutenere,
  - affidabile, corretto ed efficiente,
  - *flessibile*, *portabile* ed *espandibile*

# Flessibilità: Politiche & Meccanismi

- Principio di progettazione: *separare* meccanismi e politiche
- Per ogni risorsa bisogna distinguere tra 2 componenti del SO:
  - le politiche determinano **cosa** bisogna fare
  - i meccanismi determinano **come** farlo
- Es. Gestione della CPU
  - Lo scheduler implementa la politica di scheduling che sceglie a quale processo assegnare la CPU
  - Il dispatcher implementa il meccanismo di context switch che assegna la CPU ad un processo
- Permette la massima *flessibilità* in caso di necessità di
  - modificare le politiche con cui certe decisioni sono prese
  - aggiungere nuove componenti HW/SW (es. nuovo dispositivo/driver)

# Portabilità e espandibilità

- Per ammortizzare i costi di progettazione, un SO dovrebbe avere un ciclo di vita pari ad *almeno un decennio*, per cui dovrebbe *adattarsi ai cambiamenti* di
  - architettura dei computer
  - tecnologia dei dispositivi di I/O
  - ambienti di elaborazione
- Due caratteristiche sono importanti in questo contesto:
  - Portabilità: facilità con la quale un SW può essere adattato all'utilizzo su un altro computer
  - Espandibilità: facilità con la quale nuove funzionalità possono essere aggiunte ad un dato SW

# Portabilità

- Richiede il cambiamento di parti di codice del SO che sono dipendenti dall'HW (solitamente riguardano i meccanismi)
  - Vettore delle interruzioni: contiene le info che dovrebbero essere caricate nei vari campi del registro PS per l'esecuzione delle funzioni di gestione delle interruzioni
  - Informazioni per la protezione della memoria e quelle da fornire alla MMU
  - Istruzioni per effettuare operazioni di I/O
  - ...
- La portabilità è facilitata se
  - il codice dipendente dall'HW ha dimensione ridotta
  - le parti di codice dipendenti dall'HW e quelle indipendenti sono separate e le interfacce tra le due sono ben definite

# Espandibilità

- Necessaria per
  - incorporare nuovo HW in un computer (es. nuovi dispositivi di I/O o adattatori di rete)
  - fornire nuove funzionalità in risposta alle aspettative degli utenti
- I primi SO non erano espandibili
- I successivi risolsero il problema aggiungendo alla procedura di boot una nuova funzionalità che
  - cerca il nuovo HW oppure richiede all'utente di selezionare il SW appropriato (*driver di dispositivo*) per gestire il nuovo HW
  - carica il nuovo SW e lo integra con il kernel così che successivamente possa essere richiamato ed utilizzato in modo appropriato
- I SO moderni utilizzano la funzionalità *plug-and-play* che permette di aggiungere nuovo HW anche durante l'esecuzione del SO
  - Il SO gestisce l'interruzione causata dall'aggiunta del nuovo HW, seleziona il SW appropriato e lo integra nel kernel

# Progetto di un SO

- Non esiste una soluzione unica al problema di definire i requisiti per un SO
- L'ampia gamma di SO esistenti mostra che requisiti diversi possono comportare una grande varietà di soluzioni per ambienti diversi
- Varie categorie di SO si sono evolute nel tempo con l'evoluzione dei computer e delle aspettative degli utenti, ovvero con l'evoluzione degli ambienti di elaborazione

# Implementazione di un SO

- I primi SO erano scritti in linguaggio assembler
- I SO moderni sono scritti, in gran parte, in uno o più linguaggi di alto livello, con piccole porzioni di codice in assembler
- Vantaggi: il codice scritto in un linguaggio di alto livello
  - può essere scritto più rapidamente
  - è più compatto
  - è più facile da comprendere e da manutenere
  - è più facile da "portare" su altro HW
- L'uso di compilatori sofisticati che producono codice ottimizzato compensa gli svantaggi tipici: prestazioni minori e maggiore occupazione di memoria
- Le prestazioni sono anche notevolmente influenzate dalle strutture dati e dagli algoritmi usati

# Concetti introduttivi

- Cosa fa un SO
- Organizzazione di un sistema di elaborazione
- Gestione delle risorse
- Virtualizzazione
- Sicurezza e protezione
- Servizi di un SO
- Interfacce utente
- System call & API
- Servizi di sistema
- Progetto e implementazione di un SO
- Struttura di un SO

# Modelli strutturali

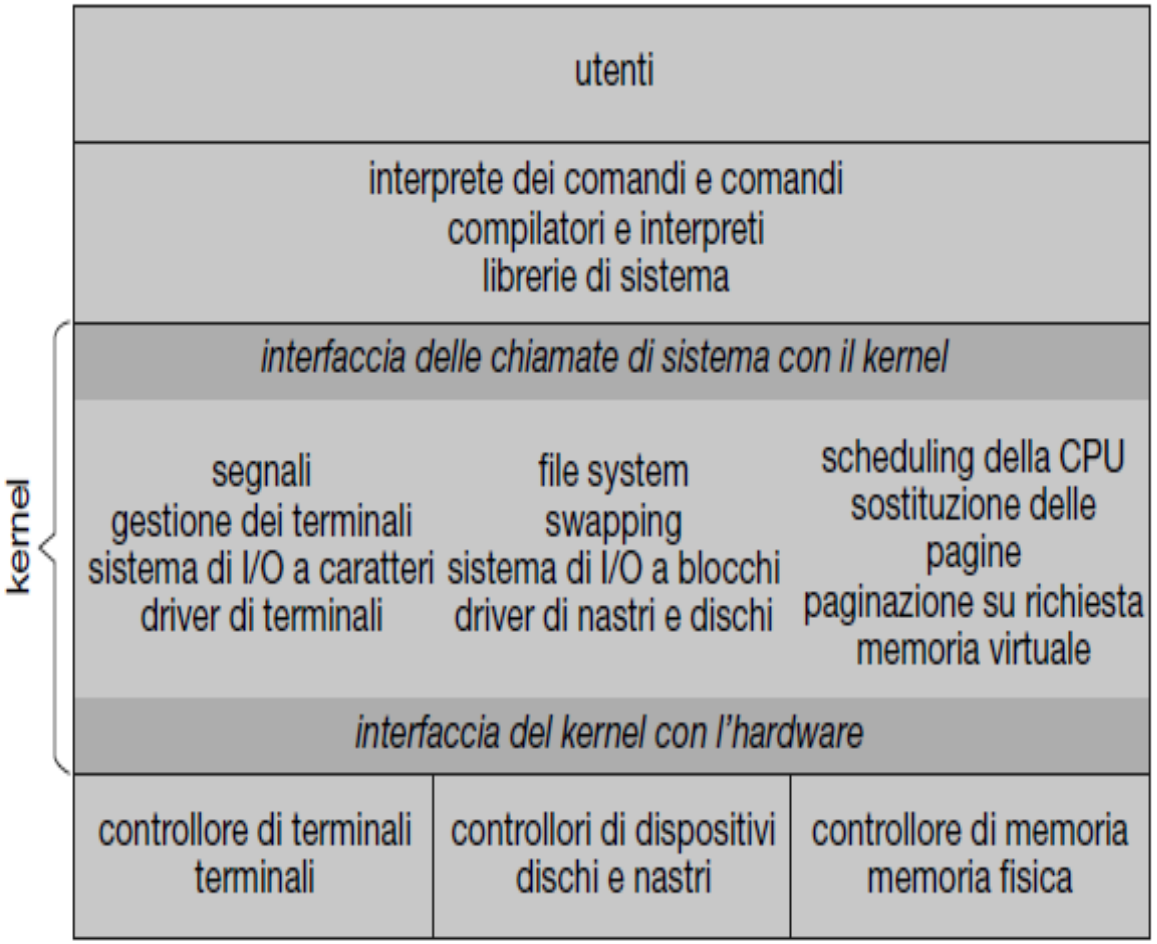
- Un SO è un software complesso
  - Svariate funzionalità e milioni di righe di codice
  - È necessario strutturarlo in qualche modo
- Il modello della struttura interna del SO è definito dai seguenti criteri progettuali
  - organizzazione delle componenti
  - modalità con cui le componenti interagiscono
- Modelli strutturali
  - Sistemi monolitici
  - Sistemi modulari
  - Sistemi a livelli (o strati)
  - Sistemi a microkernel
  - Sistemi ibridi

# Sistemi monolitici

- Il modo più semplice per strutturare un SO è quello di non strutturarlo affatto
- Ossia, collocare tutte le funzionalità del kernel in un unico file binario statico che viene eseguito in un unico spazio di indirizzi
  - Il kernel è costituito da un unico programma contenente un insieme di procedure che realizzano le varie componenti del SO
  - L'interazione tra le componenti avviene mediante il meccanismo di chiamata a procedura
- È una tecnica comune per la progettazione di SO
- Esempi: MS-DOS, primi sistemi UNIX, Linux, IBM OS/360

# Struttura di UNIX originario

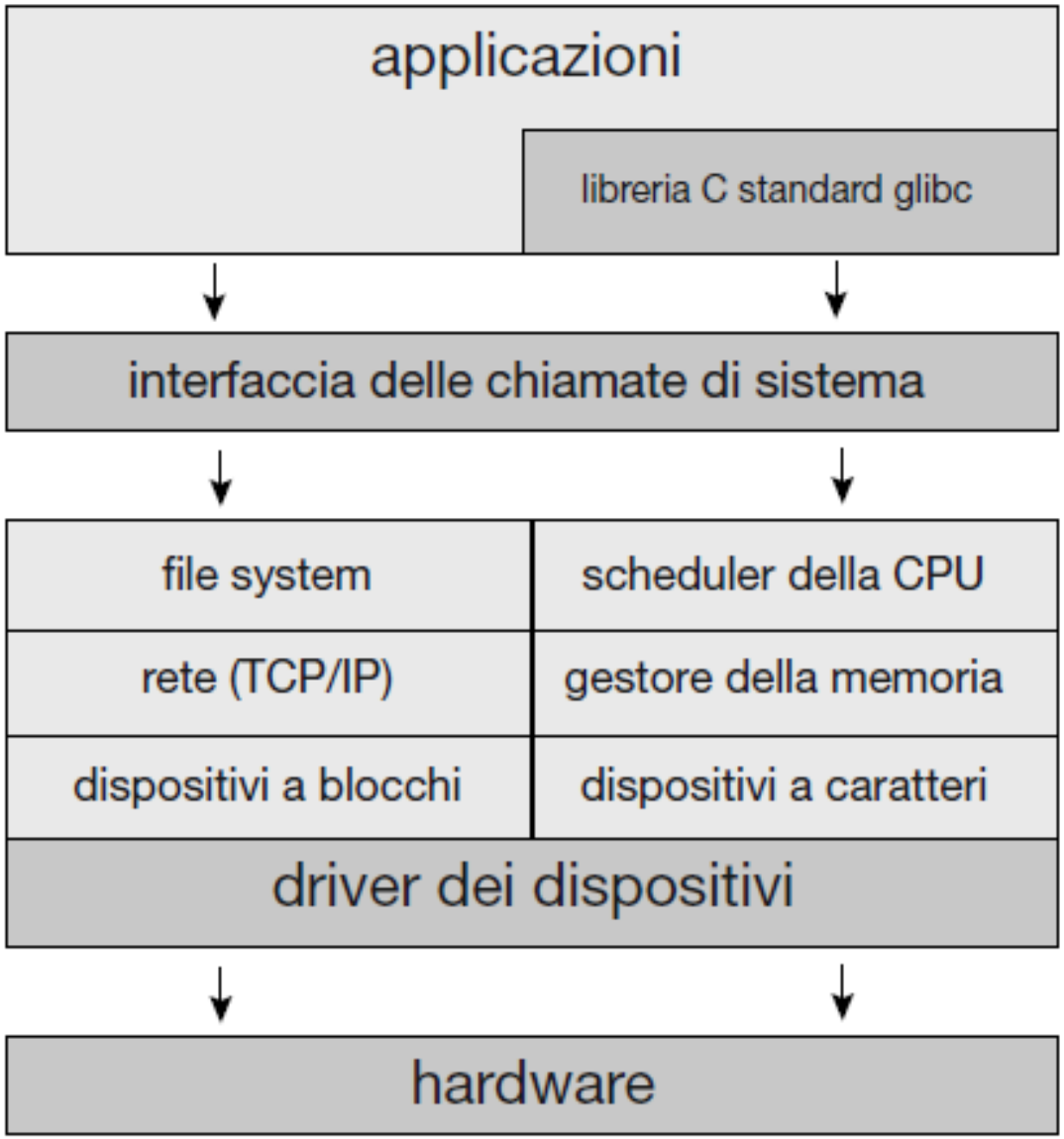
- Un esempio di tale strutturazione limitata è il SO UNIX originario
- Consiste di due parti separabili: il kernel e i programmi di sistema
- Possiamo vedere il tradizionale SO UNIX in una certa misura come stratificato
- Il kernel è costituito da tutto ciò che sta sotto l'interfaccia delle system call e sopra l'HW
- Il kernel è ulteriormente separato in una serie di interfacce e driver di dispositivo, che sono stati aggiunti e ampliati nel corso degli anni con l'evoluzione di UNIX
- Il kernel fornisce il file system, lo scheduling della CPU, la gestione della memoria e altre funzioni del SO tramite system call




In conclusione, si tratta di un'enorme quantità di funzionalità da combinare in un unico spazio di indirizzi!

# Linux

- Il SO Linux è basato su UNIX ed è strutturato in modo simile
- Le applicazioni in genere usano la libreria C standard `glibc` per comunicare col kernel tramite l'interfaccia delle system call
- Il kernel Linux è monolitico in quanto funziona interamente in modalità kernel in un unico spazio di indirizzi
- Ma ha un design modulare che consente di modificare il kernel durante il runtime



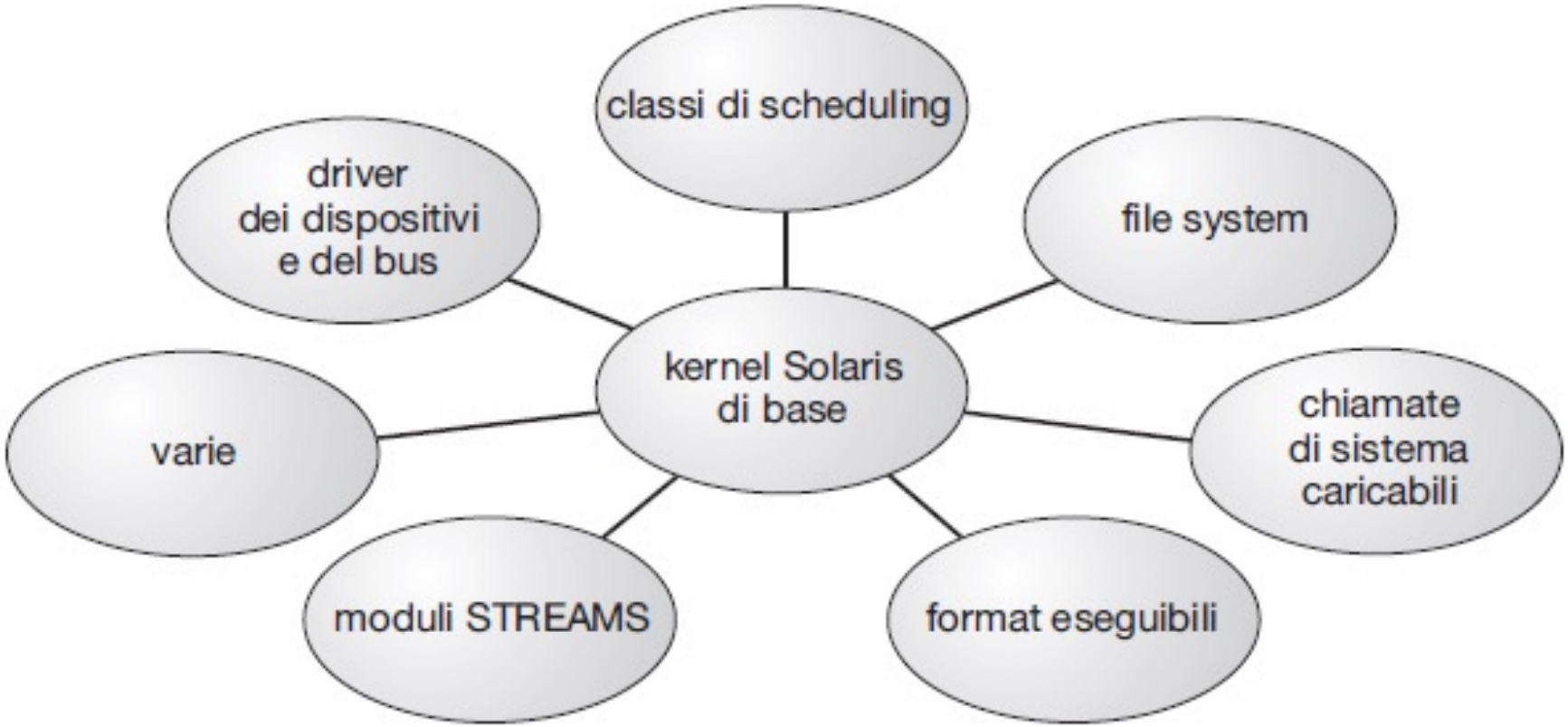
# Sistemi monolitici: considerazioni

- Nonostante l'apparente semplicità, i kernel monolitici sono difficili da implementare ed estendere
- I kernel monolitici presentano tuttavia un netto vantaggio in termini di prestazioni
  - C'è un overhead ridotto nell'interfaccia delle system call e la comunicazione all'interno del kernel è veloce
- Pertanto, nonostante gli svantaggi, la loro velocità ed efficienza spiega perché vediamo ancora usi di kernel monolitici nei SO UNIX, Linux e Windows

# Sistemi modulari

- È forse la migliore metodologia per la progettazione di un SO
- Il kernel ha una serie di componenti principali e può collegarsi a servizi aggiuntivi tramite moduli, sia all'avvio che durante l'esecuzione
- Questo tipo di progettazione è comune nelle implementazioni moderne di UNIX, come Linux, mac OS X e Solaris, nonché Windows
- L'idea del progetto è che il kernel fornisce servizi di base, mentre altri servizi sono implementati in modo dinamico,cioè mentre il kernel è in esecuzione
- Il collegamento dinamico dei servizi è preferibile all'aggiunta di nuove funzionalità direttamente al kernel, cosa che richiederebbe la ricompilazione del kernel ogni volta che viene apportata una modifica
- Pertanto, ad esempio, potremmo creare algoritmi di scheduling della CPU e di gestione della memoria direttamente nel kernel e quindi aggiungere il supporto per diversi file system tramite moduli caricabili

# Struttura di Solaris




- Solaris è una versione di UNIX sviluppata da SUN
- Kernel modulare implementato tramite tecniche di OOP
  - contiene un certo numero di componenti base
  - si collega ai servizi addizionali al boot e durante l'esecuzione

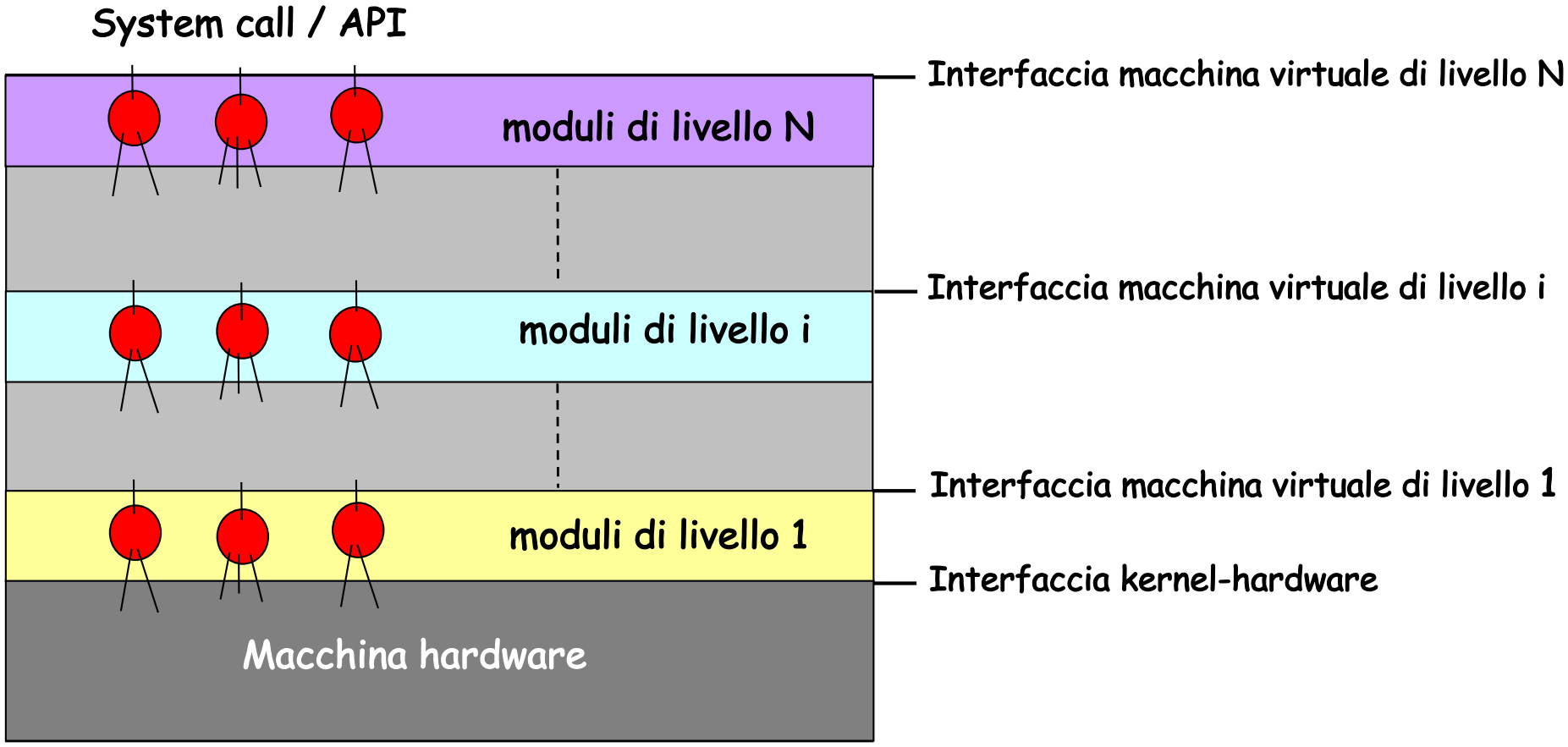
# Linux

- Linux utilizza moduli kernel caricabili (loadable kernel modules, LKM), principalmente per supportare driver di dispositivo e file system
- Gli LKM possono essere inseriti nel kernel all'avvio del sistema o durante l'esecuzione
  - Ad esempio, quando viene collegato un dispositivo USB, se il kernel non ha il driver necessario può caricarlo dinamicamente
- Gli LKM possono essere rimossi dal kernel anche durante l'esecuzione
- Gli LKM rendono il kernel Linux dinamico e modulare, pur mantenendo i vantaggi prestazionali di un sistema monolitico

# Sistemi a livelli

- Obiettivo: semplificare progetto e messa a punto del SO riducendo il numero di possibili interconnessioni tra i moduli del SO
- Le funzionalità del SO sono organizzate in un certo numero di livelli gerarchici
  - Ogni livello definisce un insieme di funzionalità e servizi, e le modalità per utilizzarli dai livelli superiori (ogni livello definisce una nuova macchina astratta)
  - I livelli sono organizzati in modo che ciascuno usi solo funzionalità e servizi dei livelli sottostanti
  - Un livello non può vedere i dettagli implementativi, es. strutture dati e istruzioni usate, interni ad altri livelli

# Sistemi a livelli



# Sistemi a livelli: considerazioni

- Complessità del progetto a causa della difficoltà nello stratificare le funzionalità e i servizi
  - Non sempre è ovvio il livello in cui realizzare una data funzionalità
- Meno flessibili dei sistemi modulari
- Inoltre, le prestazioni complessive di tali sistemi sono scarse a causa dell'overhead richiesto a un programma utente di attraversare più livelli per ottenere un servizio del SO
- Tuttavia, una certa stratificazione è comune nei SO moderni
  - Numero inferiore di livelli, ma con più funzionalità
  - Offrono la maggior parte dei vantaggi della struttura modulare, ma evitano al contempo i problemi di definizione e interazione tra livelli

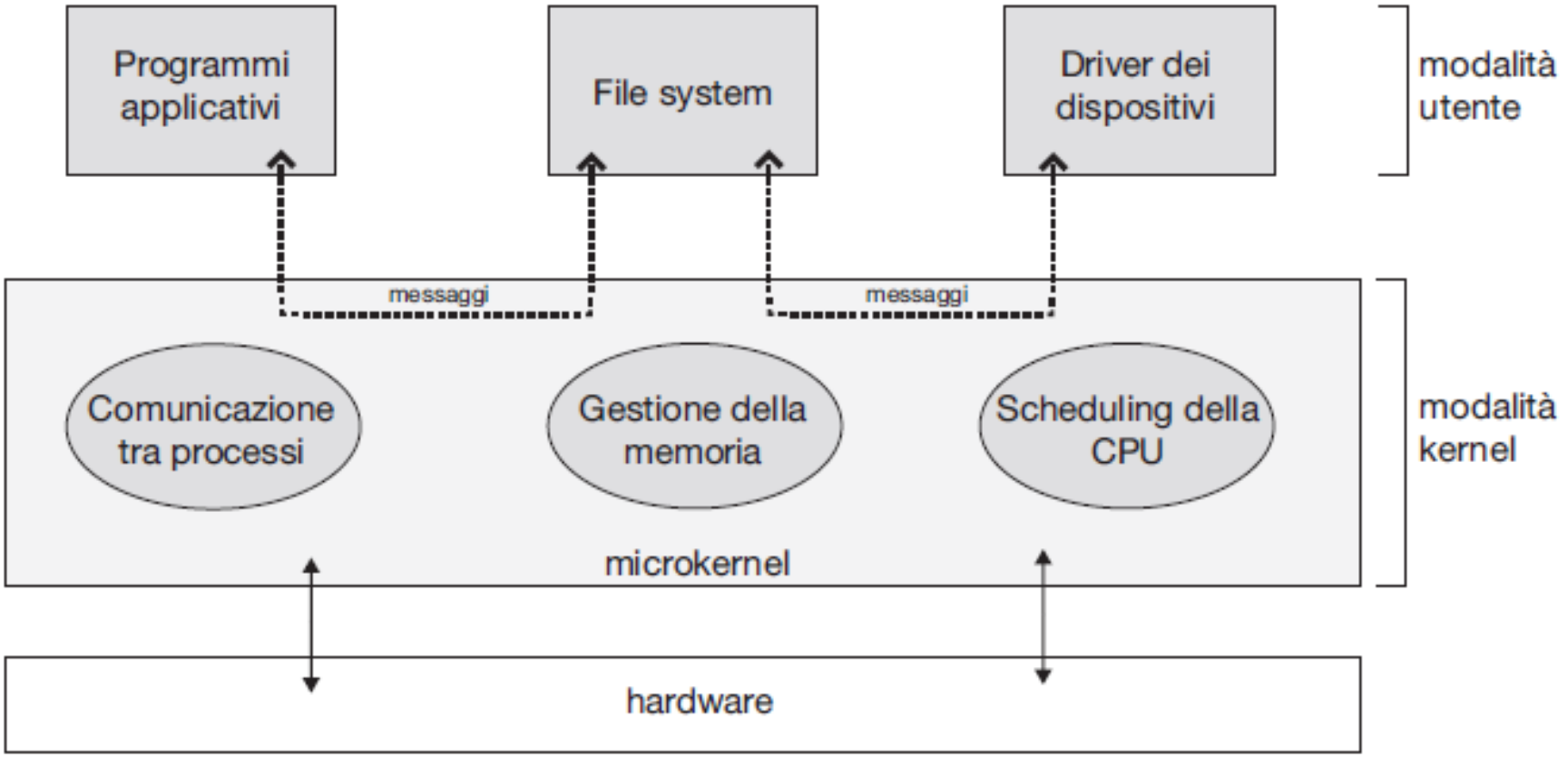
# Sistemi a microkernel

- Estremizzazione del principio di separazione tra politiche e meccanismi:
  - il kernel fornisce solo i meccanismi (eseguiti in modalità privilegiata)
  - mentre le politiche vengono implementate come processi di sistema (eseguiti in modalità utente)
- Primo esempio di SO a microkernel è una versione di UNIX chiamata Mach (sviluppata a Carnegie Mellon Univ., '85)
- Kernel ridotto al minimo indispensabile, include meccanismi per
  - comunicazione tra processi (IPC, InterProcess Communication)
  - gestione minimale della memoria, dei processi e della CPU
  - gestione dell'HW di basso livello
- Tutto il resto viene gestito da server (*processi di sistema che non terminano mai*)

  Es. le politiche di gestione del file system, i driver dei dispositivi
- Server e processi utente operano al di sopra del microkernel

# Microkernel: funzionamento di base

- I gestori delle varie risorse sono server o manager (es. *file server*, *terminal server*, *printer server*, ...)
- Quando un applicativo (client) deve richiedere l'uso di una risorsa, deve interagire col server corrispondente tramite IPC



# Microkernel: funzionamento di base

- I gestori delle varie risorse sono server o manager (es. *file server*, *terminal server*, *printer server*, ...)
- Quando un applicativo (client) deve richiedere l'uso di una risorsa, deve interagire col server corrispondente tramite IPC

- Ogni IPC comporta
  - La copia dei messaggi tra i processi interagenti, che risiedono in spazi di indirizzi separati
  - La commutazione di contesto per passare da un processo all'altro per scambiare i messaggi
- Tale overhead è stato il principale ostacolo alla diffusione dei SO basati su microkernel

# Microkernel: Vantaggi/Svantaggi

- Meno efficiente di un kernel di maggiori dimensioni
  - Che però tipicamente contiene anche codice indipendente dall'architettura
- Grande flessibilità (migliore espandibilità e portabilità)
  - Tutti i nuovi servizi vengono aggiunti allo spazio utente e di conseguenza non richiedono la modifica del kernel
- Più affidabile e sicuro
  - Una quantità inferiore di codice è eseguita in modalità protetta
- Adatto per ambienti di elaborazione di rete e embedded
- SO recenti sono basati, in diversa misura, su microkernel (AIX4, BeOS, GNU HURD, MacOS X (Darwin), QNX, Tru64, Windows NT . . . )

# Sistemi Ibridi

- Nella pratica, i SO combinano strutture diverse, che portano a sistemi ibridi orientati alle prestazioni, alla sicurezza e alla flessibilità
  - Linux è monolitico (prestazioni) ma anche modulare (espandibilità)
  - Windows è in gran parte monolitico (prestazioni), ma ha anche comportamenti tipici dei sistemi a microkernel (sicurezza) con moduli caricabili dinamicamente (espandibilità)
- Anche Apple Mac OS X e i due principali SO per dispositivi mobili, iOS e Android, sono ibridi