Componenti di un sistema operativo

Un sistema operativo può essere suddiviso in componenti:

- ➔ **Gestione dei processi:** il sistema operativo è **responsabile** di tutti i **processi in esecuzione**. **Ogni processo** necessita di **risorse** che vengono assegnate dal sistema operativo e viene eseguito in **sequenzialmente**, un'istruzione per volta. Inoltre viene fatta differenza tra processi **User** e processi **Kernel**, che vengono eseguiti con determinati privilegi rispetto ai processi User. L'OS è responsabile:
  - ◆ **Creazione e distruzione di processi:** a ogni processo è associato un id univoco (PID) al momento della creazione. In caso di errori (es SEGFAULT) è compito del sistema operativo la terminazione del processo.
  - ◆ Fornire le **primitive** per la **sincronizzazione** tra **processi**.
  - ◆ **Sospensione** e **riesumazione** dei processi (code di WAIT, READY, ...).
- ➔ **Gestione della Memoria Primaria (RAM):** ogni programma per poter essere mandato in esecuzione deve essere **caricato in memoria**. La memoria centrale primaria **conserva** inoltre anche **dati condivisi** da CPU e dispositivi I/O. L'OS è responsabile:
  - ◆ **Gestione** dell'intero **spazio** di **memoria** (che parti sono utilizzate e da chi).
  - ◆ Decisione di che processo caricare in memoria quando esiste lo spazio disponibile.
  - ◆ **Allocazione** e **rilascio** di **memoria** ai processi che ne fanno richiesta.
- ➔ **Gestione della Memoria Secondaria (di Massa):** siccome la **memoria primaria è volatile** e "piccola", uno storage di massa è indispensabile per mantenere grandi quantità di dati. Il compito dell'OS è:
  - ◆ **Gestione** dello **spazio libero** e **occupato**, e allocazione dello stesso.
  - ◆ **Scheduling** degli accessi al disco.
  - ◆ Gestione dei file:
    - Un "**File**" è un'**astrazione logica** per rendere conveniente l'utilizzo della memoria non volatile.
    - L'OS è responsabile di **creazione**/**cancellazione** delle **directory**, primitive per la gestione dei file (copia, sposta, ...), corrispondenza tra file e spazio fisico.
- ➔ **Gestione dell'I/O:** il sistema operativo **nasconde** la **complessità** delle **periferiche I/O** all'utente. Il sistema operativo:
  - ◆ Mette a disposizione un sistema per accumulare gli accessi ai dispositivi (buffering).
  - ◆ Un'interfaccia generica per i device drivers, e alcuni device drivers specifici nel caso di alcuni dispositivi.
- ➔ **Protezione:** il sistema operativo è responsabile del controllo dell'accesso alle risorse da parte degli utenti e processi in esecuzione:
  - ◆ Definizione degli accessi autorizzati e non.
  - ◆ Definizione dei controlli da imporre agli accessi.
  - ◆ Fornire strumenti per verificare le politiche di accesso alle risorse.

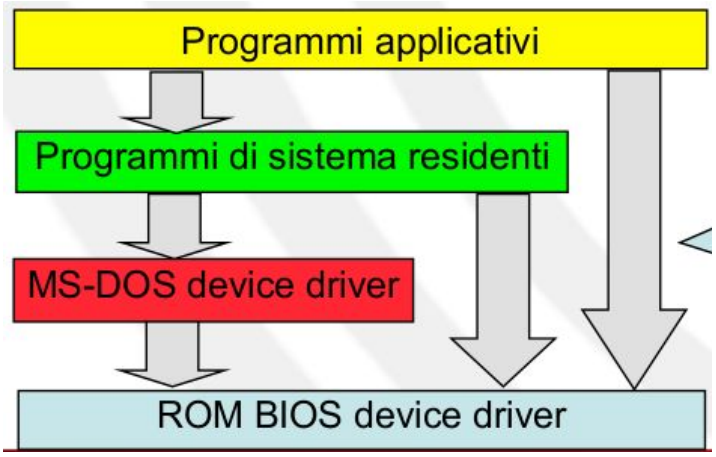
Architettura

Sistemi Monolitici

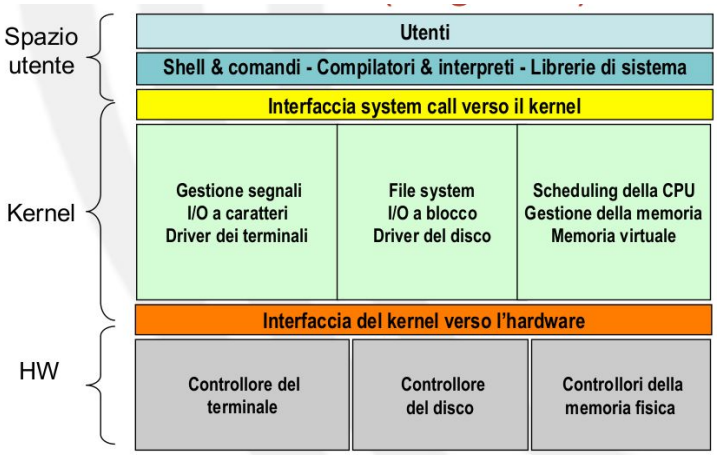
- **Semplici** e **veloci** da **implementare**.
- Non hanno **nessuna gerarchia**, composti da un unico strato di software tra l'utente e l'hardware, dove i **metodi possono chiamarsi a vicenda**.
- Il codice strettamente **dipendente dall'architettura hardware** è sparso per tutto il sistema operativo, rendendo test e debug dei componenti difficile.

Sistemi a struttura semplice

- È definita una **minima organizzazione gerarchica** molto flessibile, mirata a **ridurre i costi** di sviluppo e **mantenimento**.
- Alcuni esempi di sistemi a struttura semplice sono MS-DOS e il sistema UNIX originale:
    - MS-DOS: pensato per fornire il massimo numero di funzionalità nel minimo spazio
        - Composto da una struttura minima, ma interfacce e livelli di funzionalità non sono ben definiti, infatti i programmi applicativi posso accedere alle routine base a loro piacimento.
        - Non ha dual mode (user e kernel mode), né è suddiviso in moduli.
    - UNIX originale:
        - Composto da kernel e programmi di sistema.
        - Il kernel è composto da tutto ciò che sta tra l'interfaccia delle system call e l'hardware.



Struttura di MS-DOS



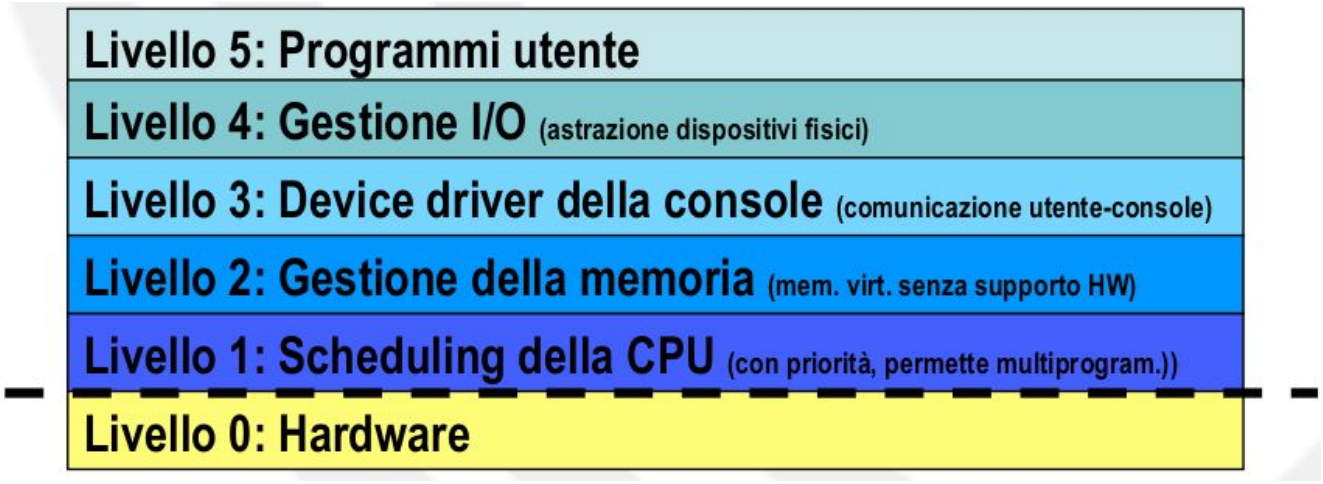
Struttura di UNIX

Sistemi a livelli

- Tutti i **servizi** sono **organizzati** per **livelli gerarchici**.
- Ogni **livello** può **comunicare** solo con il **livello superiore** e **inferiore**.
- Il livello più alto è l'interfaccia utente, il più basso è l'hardware.

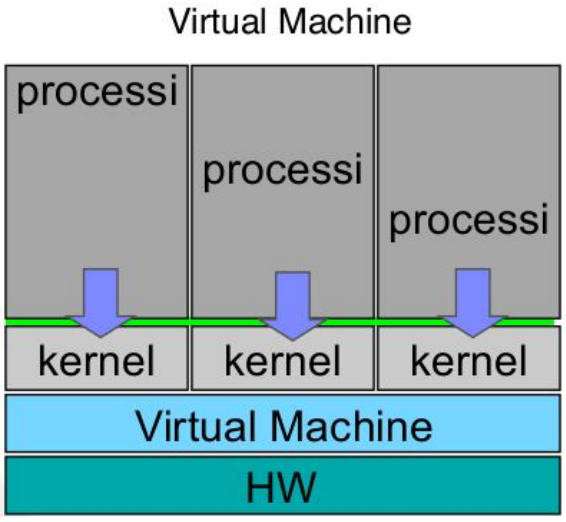
+ **Modularità**: semplice messa a punto e manutenzione del sistema.	- **Difficile definire** propriamente gli **strati**. - Ogni strato aggiunge **overhead**, quindi ne riduce l'efficienza. - Codice dipendente dall'hardware sparso per i livelli ne riduce le possibilità di portabilità.

Esempi: THE, MUTICS, OS/2



Virtualizzazione

Viene **estremizzato** il concetto dell'**approccio a livelli**, dove l'hardware e il **software della VM viene tratto come vero e proprio hardware**, fornendo un interfaccia identica all'hardware sottostante. Ciò permette l'esecuzione di più sistemi operativi su una stessa VM.



+ Protezione completa del sistema: ogni **VM è isolata** dalle altre. + **Più sistemi operativi** sulla stessa macchina host. + Buona **portabilità**. + Ottimizzazione delle risorse: una vm può ospitare quello che dovrebbe essere eseguito su macchine separate.	- **Problemi** di **performance**. - Problemi nel gestire la **dual mode virtuale**. - Nessuna condivisione: ogni VM è totalmente isolata.

Esempi: JVM, VirtualBox, VMWare.

Sistemi basati su kernel

Composto solamente da **servizi kernel** e **servizi non-kernel**, dove **alcune funzionalità** del sistema operativo sono implementate **fuori dal kernel** (es. Filesystem).

+ Vantaggi dei sistemi a livelli, senza averne troppi.	- Nessuna regola organizzativa per le parti fuori dal kernel. - Kernel complessi tendono a diventare monolitici.

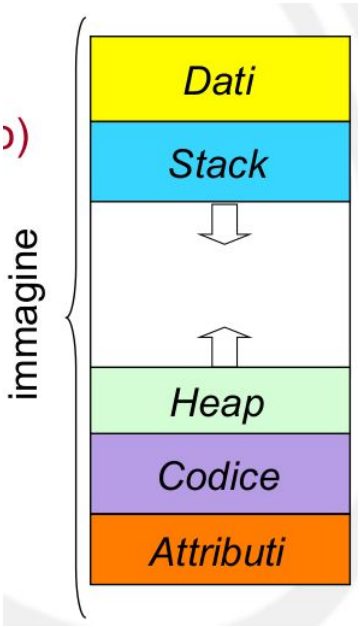
Sistemi client-server

- Tutti i servizi del S.O. sono realizzati come processi utente (client).
- Il client chiama un processo servitore (server) per usufruire di un servizio.
- Il server, dopo l'esecuzione, restituisce il risultato al client.
- Il kernel si occupa solo della gestione della comunicazione tra client e server.
- Il modello si presta bene per S.O. distribuiti.
- S.O. moderni realizzano tipicamente alcuni servizi in questo modo.

Processi e scheduling della CPU

Processo=istanza di un **programma in esecuzione**, eseguito in maniera sequenziale. In sistemi multiprogrammati i processi evolvono in modo concorrente.

Immagine di memoria



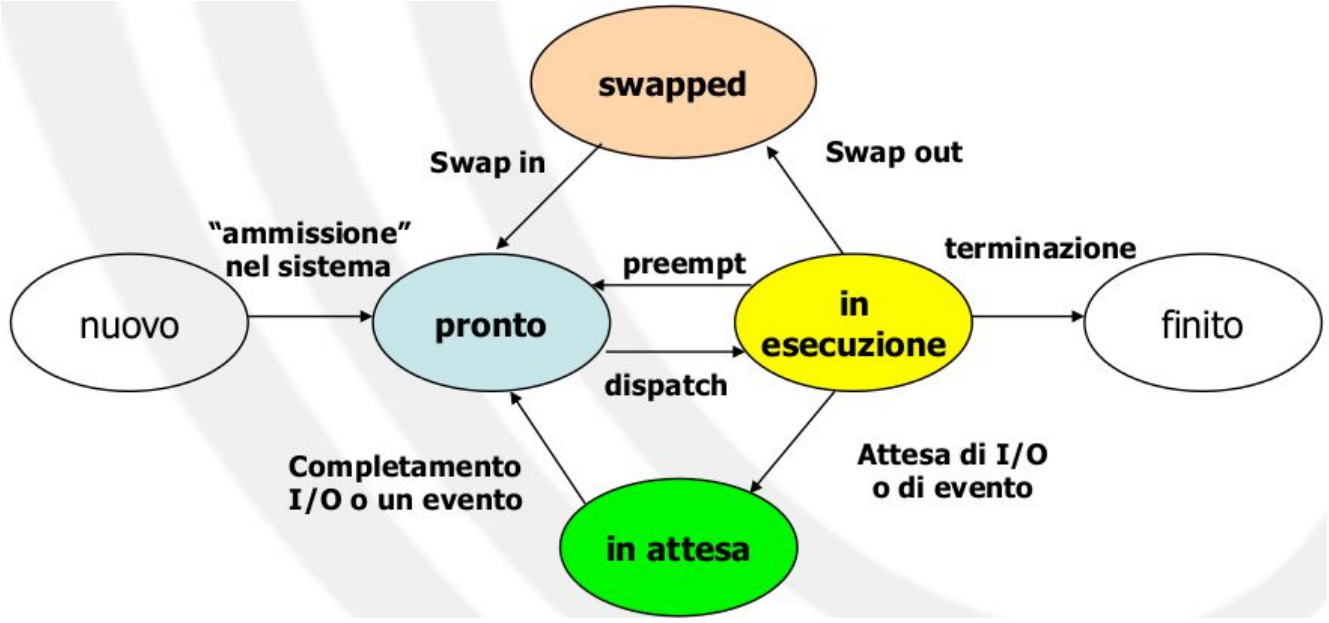
Un processo è composto da:

- **Istruzioni** (sezione Codice o Testo)
  - Parte statica del codice
- **Dati** (sezione Dati)
  - Variabili globali
- **Stack**
  - Chiamate a procedura e parametri
  - Variabili locali
- **Heap**
  - Memoria allocata dinamica
- **Attributi** (id, stato, controllo)

Ogni processo è rappresentato da un serie di attributi, detto **PCB** (**P**rocess **C**ontrol **B**lock), che sono:

- **Stato del processo**
- **Program Counter**
- **Valore dei registri**
- **Informazioni sulla memoria** (registri limite, tabella delle pagine)
- **Stato dell I/O**
- **Informazioni sullo scheduling** (priorità)
- **Informazioni sull'utilizzo del sistema** (CPU)

Stati di un processo



La **scelta** del **processo da mandare in esecuzione** nella CPU è necessario per garantire:

- **Multiprogrammazione**
  - Massimizzare l'utilizzo della CPU, più di un processo in memoria.
- **Time-sharing**
  - Far commutare frequentemente la CPU tra i processi (**context switch**).

Ogni **processo** può essere **inserito** in una serie di **code**:

- **Pronto:** tutti i processi pronti per passare nello stato di esecuzione.
- **In attesa:** processi in attesa di un dispositivo o un evento I/O.

[...]

Sincronizzazione e Deadlock

Prendiamo in esempio il modello astratto del Producer-Consumer:

➔ Il produttore aggiunge al buffer
➔ Il consumatore toglie dal buffer

Non si può togliere da un buffer vuoto e non si può aggiungere a un buffer pieno, inoltre le operazioni sono eseguite in maniera concorrente.

Produttore

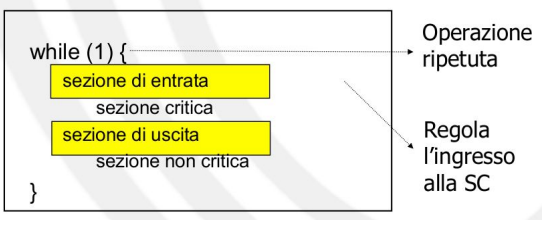
```
void deposit (item p)
{
 while (counter == N)
      no_op;
 buffer[in] = p;
 in = (in+1) % N;
 counter++;
}
```

Consumatore

```
item remove ()
{
   while (counter == 0)
     no_op;
   next = buffer[out];
   out = (out+1) % N;
   counter--;
   return next;
}
```

Le istruzioni di **incremento** e **decremento** del buffer a livello di assembly **sono implementate con più microistruzioni**, eseguite sequenzialmente, ma **non è possibile sapere l'interleaving** (**quale** istruzione dei processi viene eseguita **prima** o **dopo**).

Il problema è sia il producer che il consumer **possono tentare di modificare il contatore contemporaneamente**, rischiando di creare un **inconsistenza** dello stesso. Il contatore, che è una variabile condivisa tra i due processi, è detta **sezione critica**.



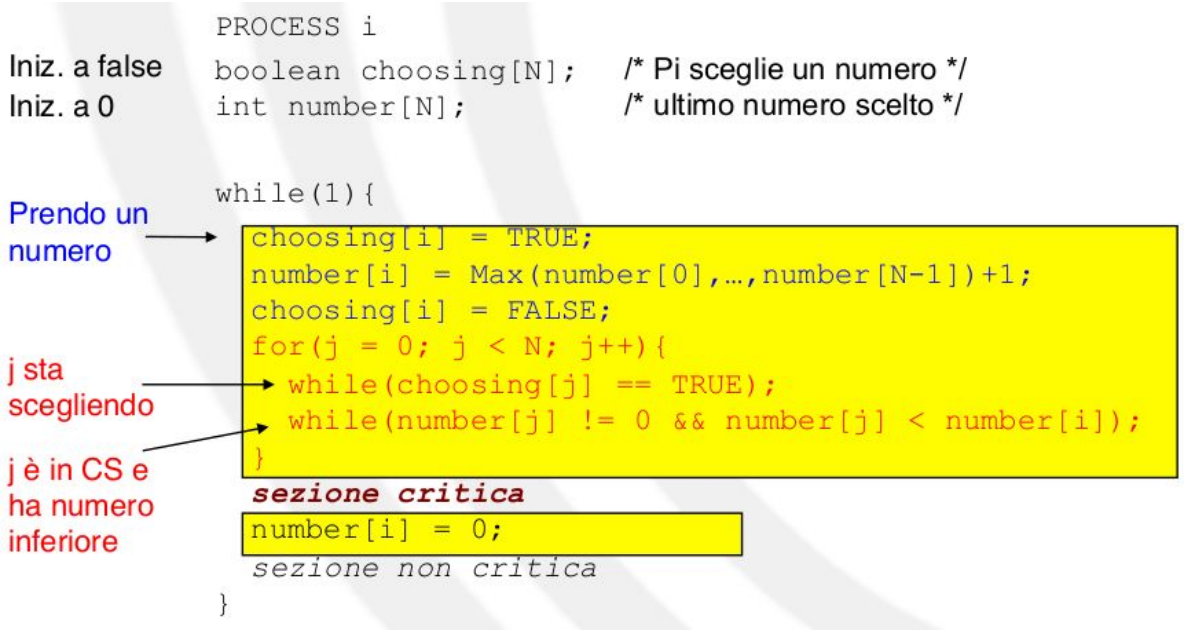
Per **proteggere una sezione critica**, bisogna rispettare **tre criteri**:

- ➔ **Mutua Esclusione**
  - ◆ Un processo alla volta può accedere alla sezione critica.
- ➔ **Progresso**
  - ◆ Solo i processi che stanno nella sezione critica possono decidere chi entra nella stessa.
  - ◆ La decisione non può essere rimandata all'infinito.
- ➔ **Attesa Limitata**
  - ◆ Un processo può entrare solo un numero limitato di volte consecutivamente.

Soluzioni Software

Soluzione del problema con un numero N di processi: **Algoritmo del Fornaio**.

- ➔ **Ogni processo sceglie un numero.**
- ➔ Il **numero più basso** viene **servito per primo**.
- ➔ Nel caso di **numeri uguali** viene usato un **confronto a due livelli** (come il la posizione i o il PID)



Soluzioni Hardware

Un modo hardware per risolvere il problema della sezione critica è **disabilitare gli interrupt** mentre una certa variabile viene modificata. Tuttavia, in caso il test per l'accesso sia troppo lungo, gli interrupt verrebbero disabilitati per troppo tempo.
Le **operazioni** per **l'accesso alla risorsa** devono essere **atomiche**, quindi **non deve essere possibile interromperle**.

Test-and-set

```
bool TestAndSet (boolean &var)
{
    boolean temp;
    temp = var;
    var = TRUE;
    return temp;
}
```

ATOMICA

Il valore di `var` viene modificato

- Valore di ritorno: vecchio valore di `var`
- Assegna TRUE a `var`

```
boolean lock;     /* globale iniz. FALSE */

while (1) {
  while (TestAndSet(lock)) ;
  sezione critica
  lock = FALSE;
  sezione non critica
}
```

Passa solo il primo processo che arriva e trova *lock = FALSE*

Swap

```
void Swap (boolean &a, boolean &b)
{
    boolean temp;
    temp = a;
    a = b;
    b = temp;
}
```

ATOMICA

```
boolean lock;        /* globale, inizializzata a FALSE */

while (1) {
     dummy = TRUE;   /* locale al processo */
     do
          Swap(dummy, lock);
     while (dummy == TRUE);
     sezione critica
     lock = FALSE;
     sezione non critica
}
```

Quando `dummy=false` P_i accede alla SC.
Gli altri processi continuano a scambiare *true* con *true* e non accedono a SC finché P_i non pone `lock=false`

Sia **Test-and-set sia Swap** però **non garantiscono l'attesa limitata**.

Semafori

Un **semaforo** è una **variabile intera S** a cui si accede tramite **due primitive atomiche**:

- ➔ **Signal: V(s):**
  - ◆ **Incrementa** il valore S di 1.
- ➔ **Wait: P(s):**
  - ◆ **Tenta di decrementare** il valore S di 1, se S == 0, è **necessario attendere** che prima venga **incrementato tramite una V(s)**

I semafori possono essere di **tipo binario** (possono assumere solo i valori 0 e 1) o **generici** (interi, valori >= 0).

Utilizzi principali dei semafori:

- **Mutex(Mutua esclusione):** semaforo binario inizializzato con il valore 1. Protezione della sezione critica da N processi, solo uno di essi può entrare in sezione critica per volta.
- **Sincronizzazione:** semaforo binario inizializzato a 0, il processo si mette in attesa di un evento.

```
/* valore iniziale di s = 1 (mutex) */
while (1) {
   P(s);
   sezione critica
   V(s);
   sezione non critica
}
```

Problemi

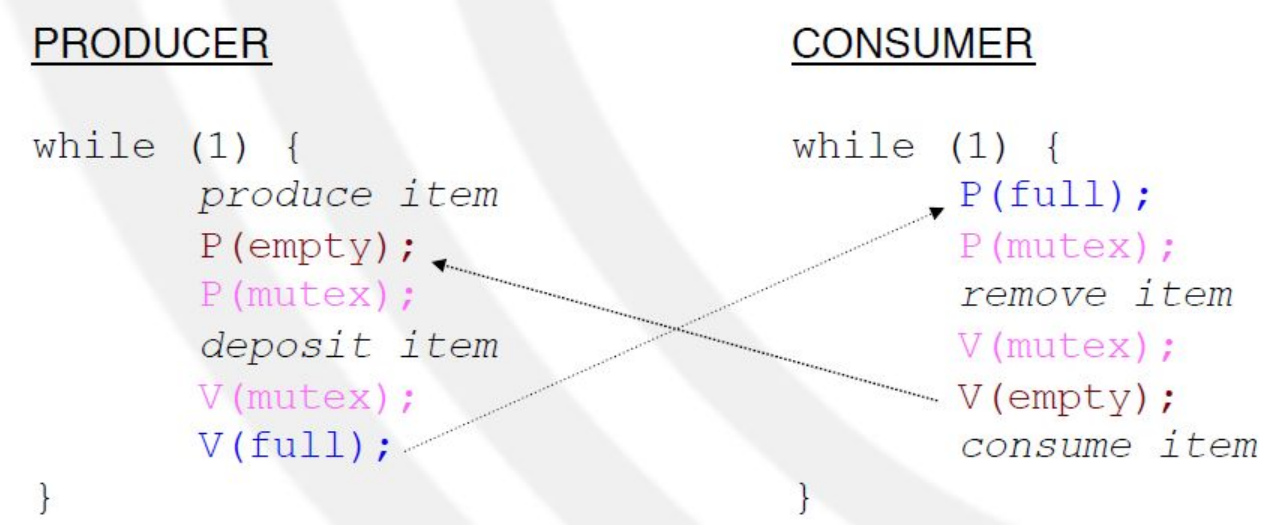
Alcuni **problemi tipici** che si possono incontrare nell'utilizzo dei semafori sono:

- **Deadlock (blocco critico):**
  - Un **processo si blocca in attesa di un evento che solamente lui può generare**, un esempio è l'attesa circolare.
- **Starvation**
  - Si intende per starvation uno stato in cui **non c'è attesa limitata**, cioè un processo può **rimanere bloccato per un tempo indefinito** in attesa di un evento.

Problema del produttore/consumatore

Necessari 3 semafori:

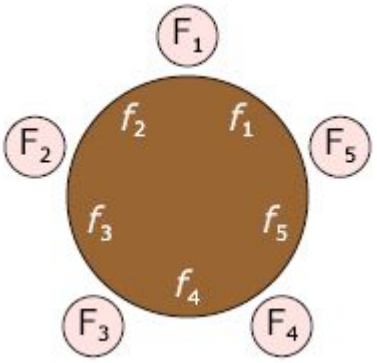
```
Sem Bin mutex = True // mutua esclusione per il buffer, o solo il produttore o solo il consumatore posso accedervi
Sem Int empty = N // il produttore non può accedere al buffer se esso è vuoto
Sem Int full = 0 // Il consumatore non può accedere il il buffer è vuoto
```



- Il producer può **accedere alla sezione critica solo se il buffer non è pieno** grazie al semaforo empty posto in cima; se è pieno, verrà bloccato li.
- Una volta **passata** la **sezione critica** protetta dal mutex, **sblocca il consumer chiamando la V su full**.

- Se **full** è **vuoto**, il **consumatore si blocca** in attesa che il producer aggiunga un elemento.
- **Superata** la **sezione critica** protetta dal mutex, **sblocca il produttore** dandogli la possibilità di aggiungere un elemento al buffer, incrementando utilizzando la **V su empty**.

Problema dei Dining Philosophers



Cinque filosofi sono seduti a un tavolo con un piatto davanti e una forchetta a destra e una a sinistra. Un filosofo può pensare o mangiare. Per mangiare un filosofo ha bisogno della forchetta alla sua destra e di quella alla sua sinistra.
Il primo filosofo F_1 dovrà prendere la prima forchetta f_1 prima di poter prendere la seconda forchetta f_2. I filosofi F_2, F_3 e F_4 si comporteranno in modo analogo, prendendo sempre la forchetta f_i prima della forchetta f_{i+1}. Rispettando l'ordine numerico ma invertendo l'ordine delle mani, il filosofo F_5 prenderà prima la forchetta f_1 e poi la forchetta f_5. Si crea così un' alternanza che evita il deadlock.

- Soluzione corretta
  - Ogni filosofo può essere in tre stati
    - Pensante (THINKING)
    - Affamato (HUNGRY)
    - Mangiante (EATING)

Variabili condivise
- `semaphore mutex = 1;`
- `semaphore f[N] = 0;`
- `int stato[N] = THINKING;`

```
Void Philosopher (int i)
{
  while (1) {
  Think();
  Take_fork(i);
  Eat();
  Drop_fork(i);
}
```

```
Void Drop_fork (int i)
{
  P(mutex);
  stato[i] = THINKING;
  test((i-1)%N);
  test((i+1)%N);
  V(mutex);
```

I vicini possono mangiare

```
Void test (int i)
{
  if (stato[i] == HUNGRY &&
      stato[i-1] != EATING &&
      stato[i+1] != EATING)
  {
    stato[i] = EATING;
    V(f[i]);
  }
}
```

```
Void Take_fork (int i)
{
  P(mutex);
  stato[i] = HUNGRY;
  test(i);
  V(mutex);
  P(f[i]);
}
```

56

Sleepy Barber

Un barbiere ha un negozio con una sala d'attesa con N sedie e una stanza adibita al taglio. In assenza di clienti, il barbiere dorme, ma quando entra un cliente: se trova le sedie occupate, se ne va; se il barbiere è occupato, si siede; se il barbiere dorme, lo sveglia.

```
Sem customers = 0;      // sveglia il barbiere
BinSem barbers = 0;     // stato del barbiere
BinSem mutex = 1;       // protegge la sezione critica
int waiting = 0;        // conta i clienti in attesa
```

Barber

```
while (1) {
    P(customers);
    P(mutex);
    waiting--;
    V(barbers);
    V(mutex);
    cut hair;
}
```

Customer

```
P(mutex);
if (waiting < N){
   waiting++;
   V(customers); //sveglia!!
   V(mutex);
   P(barbers);    //pronto x il taglio
   get haircut;
} else {
   V(mutex);      //non c'è posto!
}
```

Monitor

Sono **costrutti simili a delle classi**, utilizzati per la **condivisione sicura ed efficiente di dati** tra diversi processi. In un monitor è **possibile definire delle variabili**, **visibili** solamente **all'interno** del monitor stesso, e sono visibili solamente ai metodi definiti all'interno del monitor stesso. In un monitor può essere attivo solamente un solo processo alla volta (non è necessario codificare la mutua esclusione esplicitamente).
Le **variabili dichiarate all'interno di un monitor sono dette condition**, e sono accessibili solamente tramite due primitive:

- **wait():** Il **processo che invoca** questo metodo **verrà bloccato fino** all'invocazione della controparte **signal()**.
- **signal():** invocando il metodo signal sono possibili **diversi comportamenti**.
    - L'invocazione del metodo comporta un **risveglio in processo nello stato di wait()**
        - Se non c'è nessun processo in wait() non ha effetti
    - Dopo l'invocazione del metodo signal() **il processo chiamante può bloccarsi** e passare l'esecuzione al processo sbloccato, **oppure**, il **processo che invoca esce dal monitor**.

Esempio di Semaforo binario implementato con un monitor:

```
monitor BinSem
{
        boolean busy; /* iniz. FALSE */
        condition idle;

        entry void P( )
        {
                if (busy) idle.wait();
                busy = TRUE;
        }
        entry void V( )
        {
                busy = FALSE;
                idle.signal ();
        }
        busy = FALSE;          /* inizializzazione */
}
```

Deadlock

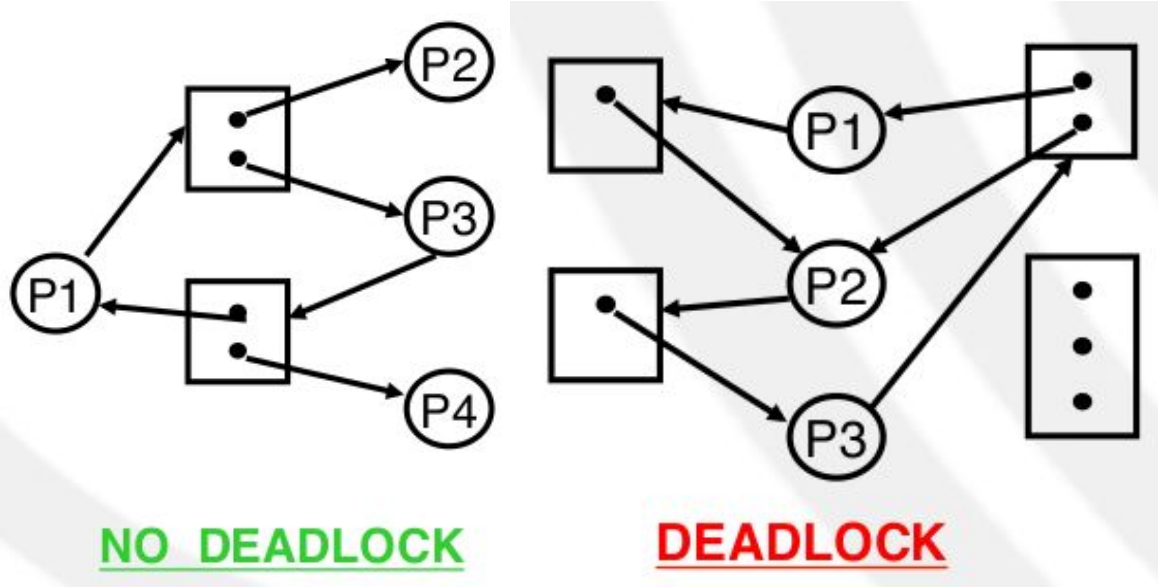
Quando un **insieme di processi** è in **attesa** ognuno di **una risorsa utilizzata da un altro processo** in quello **stesso insieme** allora siamo davanti a una situazione di deadlock. Affinché possa verificarsi un deadlock **tutte le condizioni seguenti devono essere verificate:**

- **Mutua esclusione** (almeno una risorsa non condivisibile)
- **Hold and Wait** (un processo ha una risorsa e ne attende un'altra che è già occupata)
- **No preemption** (SOLO il processo può rilasciare una risorsa se esso la detiene)
- **Attesa circolare** (un insieme di processi si attendono l'un l'altro per il possesso di una risorsa)

RAG

Per **rappresentare i deadlock** viene usato un modello astratto detto **RAG** (**R**esource **A**llocation **G**raph):

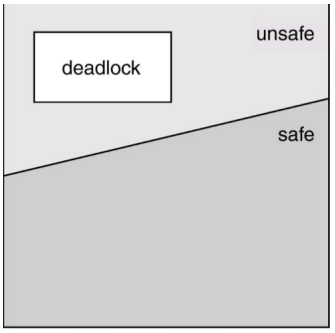
- Nodi **CERCHIO** rappresentano **processi**.
- Nodi **RETTANGOLO** rappresentano le **risorse**.
- Archi **DA PROCESSO A RISORSA** indicano una **richiesta** della risorsa.
- Archi **DA RISORSA A PROCESSO** indicano **detenzione** della risorsa.
- Se il RAG **non ha cicli**, allora **non ci sono deadlock**.
- Se il RAG ha **cicli**, allora:
    - Se la risorsa ha solo **una istanza**, allora c'è **deadlock**.
    - Se la risorsa ha **più istanze**, allora dipende dallo **schema di allocazione**.



Gestione deadlock

Un deadlock si può gestire tramite **4 principali tecniche**:

- **Prevenzione statica** (evitare le quattro condizioni dei deadlock).
    - **Mutua esclusione:** irrinunciabile, si lavora su altro.
    - **Hold and Wait:**
        - si alloca **tutto all'inizio**
          OPPURE
        - si alloca **una risorsa alla volta**.
    - **No preemption:**
        - Se la risorsa richiesta **non è disponibile**, **rilascia tutte** quelle che hai
          OPPURE
        - **Cede risorse su richiesta** di altri processi
    - **Attesa circolare:**
        - Assegnare una priorità ad ogni risorsa e un processo può richiedere una risorsa solo in ordine di priorità (es. se la risorsa con priorità 4 è già stata presa non si può prendere la risorsa con priorità 2 perché la 4 è ancora in uso)



- **Prevenzione dinamica**, usando allocazione risorse (richiede troppe informazioni sulle risorse).
    - Viene **analizzato dinamicamente** il **RAG** per **evitare cicli**.
    - È necessario **sapere dall'inizio quante risorse** massimo può richiedere ogni processo.
    - **Ogni risorsa ha uno stato**, calcolato in base al numero di sue **istanze ALLOCATE** ai processi o invece **DISPONIBILI**.
    - Il sistema è in **stato *safe*** se esiste una sequenza safe, ovvero se **esiste un ordine in cui mettere i processi richiedenti** (P1, … PN) che permette di **allocare le risorse disponibili ad ogni processo Pi** in modo che **ciascuno** possa **terminare** la sua **esecuzione**, cioè permettendo di allocare risorse solo se sono tutte disponibili o se sono utilizzate da processi che sono precedenti a questo e che metterebbero questo in attesa fino al rilascio della risorsa.
    - Se non esiste una tale sequenza il sistema è ***unsafe*** e **PUÒ** (non necessariamente però) **condurre** a uno **stato di deadlock**.
    - Bisogna **utilizzare algoritmi** che lasciano sempre il sistema in **stato safe**, ovvero si parte in stato safe e si danno risorse solo se poi si rimane in stato safe, al prezzo di una minore frequenza di utilizzo delle risorse.
    - Due alternative implementative:
        - **Algoritmo con RAG:** vincola ad avere una sola istanza per risorsa.
            - vengono aggiunti degli **archi di richiamo tratteggiati**, che indicano se quel processo può richiedere quella risorsa IN FUTURO.
            - ogni **processo dichiara quante risorse** vorrebbe usare.
            - una **richiesta** viene **soddisfatta se e solo se non crea un ciclo nel RAG**, rivelato solo con algoritmi di complessità quadratica sul numero di processi.
        - **Algoritmo del banchiere: meno efficiente del RAG** (quadratico sul numero di processi, per ogni risorsa), **ma permette più istanze**:
            - **Il banchiere non deve mai distribuire tutto il denaro** che ha in cassa perché altrimenti non potrebbe più soddisfare successivi clienti.
            - Ogni processo dichiara la sua massima richiesta e ad ogni richiesta viene determinato se soddisfarla lascia il sistema in stato safe usando quindi prima un **algoritmo di allocazione e** poi uno di **verifica dello stato.**
            - **ALGORITMO ALLOCAZIONE:** quando un processo richiede delle risorse, viene prima controllato che non siano più del massimo dichiarato, in caso positivo, **se le risorse richieste sono più di quelle disponibili**, il **processo** viene **messo in attesa**. Ora, se le risorse **sono disponibili**, viene **simulata l'assegnazione** delle risorse e viene **controllato** se il sistema **rimane safe**. In caso **contrario**, si fa il **rollback** dell'assegnazione simulata e si mette il processo in **wait**, **altrimenti** se effettua effettivamente **l'assegnazione** delle **risorse**.
            - **ALGORITMO VERIFICA:** dopo **l'assegnazione simulata** delle risorse, si controlla se esiste un processo richiedente che può ancora essere eseguito (in base al massimo che può richiedere) e in caso negativo, lo stato dopo l'assegnazione sarebbe unsafe, quindi si esegue il rollback; in caso positivo, lo stato continuerebbe ad essere safe, quindi le risorse possono essere assegnate.

Algoritmo di allocazione (P_i)

```
void request(int req_vec[]) {
   if (req_vec[] > need[i][])
      error();   /* superato il massimo preventivato */
   if (req_vec[] > available[])
      wait();   /* attendo che si liberino risorse */
   available[] = available[] - req_vec[];
   alloc[i][] = alloc[i][] + req_vec[];
   need[i][] = need[i][] - req_vec[];
   if (!state_safe()) {/* se non è safe, ripristino il vecchio stato */
      available[] = available[] + req_vec[];
      alloc[i][] = alloc[i][] - req_vec[];
      need[i][] = need[i][] + req_vec[];
      wait();
   }
}
```

Richieste del processo P_i

"simulo" l'assegnazione

rollback

Algoritmo di verifica dello stato

```
boolean state_safe(){
   int work[m] = available[];
   boolean finish[n] = (FALSE,...,FALSE);
   int i;
   while (finish != (TRUE,...,TRUE)) {
      /* cerca Pi che NON abbia terminato e che possa
         completare con le risorse disponibili in work */
      for (i=0; (i<n)&&(finish[i]||(need[i][]>work[])); i++);
      if (i==n)
         return FALSE; /* non c'è → unsafe */
      else {
         work[] = work[]+alloc[i][];
         finish[i] = TRUE;
      }
   } return TRUE;
}
```

Ho già sottratto le richieste di P_i !

Sono arrivato in fondo, senza trovare finish[i]=FALSE e need[i][ ] <= work[ ]

L'ordine non ha importanza. Se + processi possono eseguire, ne posso scegliere uno a caso, gli altri eseguiranno dopo, visto che le risorse possono solo aumentare

29

- **Rivelazione e ripristino** (**permette** i **deadlock**, usa **metodi** per **riportare** il **sistema indietro**).
  - Rilevazione statica e dinamica riducono troppo l'utilizzo delle risorse, quindi viene usato un approccio alternativo:
    - **RAG**: controlla periodicamente il RAG e al verificarsi di un deadlock, inizia la recovery del sistema. Funziona solo con una risorsa per tipo, non necessita di conoscere prima le richieste ma ha il costo di recovery
      OPPURE
    - **Algoritmo di detection**: basato sull'esplorazione di ogni possibile sequenza di allocazione per i processi non ancora terminati; se la sequenza va a buon fine, siamo in uno stato safe.

Algoritmo di detection

```
int work[m] = available[m];
bool finish[] = (FALSE,...,FALSE), found = TRUE;
while (found)  {
      found = FALSE;
      for (i=0; i<n && !found; i++) {
        /* cerca un Pi con richiesta soddisfacibile */
        if (!finish[i] && req_vec[i][] <= work[]){
         /* assume ottimisticamente che Pi esegua fino al termine
         e che quindi restituisca le risorse */
          work[] = work[] + alloc[i][];
          finish[i]=TRUE;
          found=TRUE;
        }
      }
} /* se finish[i]=FALSE per un qualsiasi i, Pi è in deadlock */
```

Se non è così il possibile deadlock verrà evidenziato alla prossima esecuzione dell'algoritmo

    - L'algoritmo di detection viene eseguito:
      - Dopo ogni richiesta di risorsa
      - Dopo un intervallo di tempo prestabilito
      - Quando l'utilizzo della CPU scende sotto una certa soglia prestabilita
    - Quando si rivela un deadlock, si ripristina il sistema in stato safe:
      - **Uccidendo tutti i processi**, perdendo quindi tutto il lavoro
      - **Uccidendo selettivamente** in base a svariate condizioni (priorità, tipi di risorse allocate, tempo di esecuzione rimanente, interatico o batch, ...), invocando quindi ogni volta l'algoritmo di rilevazione per controllare se c'è ancora il deadlock

- **Prelazionando le risorse** (tolte forzatamente) ai processi in deadlock, che non potranno quindi procedere normalmente con l'esecuzione e torneranno (se scritti come si deve) in uno stato in cui non avevano la risorsa, quindi safe, con una operazione di **rollback**; eventualmente, effettueranno un **total rollback**, ripartendo da zero con l'esecuzione.
    - Attenzione: se tolgo le risorse sempre agli stessi processi, quelli andranno in **starvation**; quindi è necessario considerare anche il numero di rollback effettuati da ogni processo nei fattori di costo

- **Algoritmo dello struzzo** (i deadlock sono rari, la **prevenzione** e **recovery** è troppo **costosa**, gli algoritmi spesso sbagliano, meglio ignorare tutto).

Conclusione: nessun approccio è superiore a un altro, solitamente si utilizza una soluzione ibrida, partizionando le risorse, dando una priorità e gestendo ogni classe di priorità con un algoritmo appropriato al caso:

1. risorse interne (I/O e PCB), prevenzione tramite ordinamento di risorse
2. memoria, prevenzione tramite prelazione
3. risorse di processo (file), prevenzione dinamica
4. spazio di swap (blocchi su disco), prevenzione tramite allocazione hold and wait

ma a volte viene preferita la soluzione più semplice, ovvero ignorare tutto e continuare l'esecuzione dei processi anche se essi non stanno effettivamente facendo niente se non aspettarsi a vicenda.

Memoria Principale e Memoria Virtuale

La **condivisione** della **memoria principale** tra di **più processi** è **fondamentale per un uso efficiente delle risorse di sistema**. Tuttavia porta con sé una serie di **problematiche**:

- **Allocazione della memoria** a singoli job.
- **Protezione dello spazio di indirizzamento** da accessi non autorizzati ad aree di memoria.
- **Condivisione dello spazio di indirizzamento** tra più processi.
- Gestione dello **swap** (parti di memoria che vengono "swappate" su disco in caso di necessità).

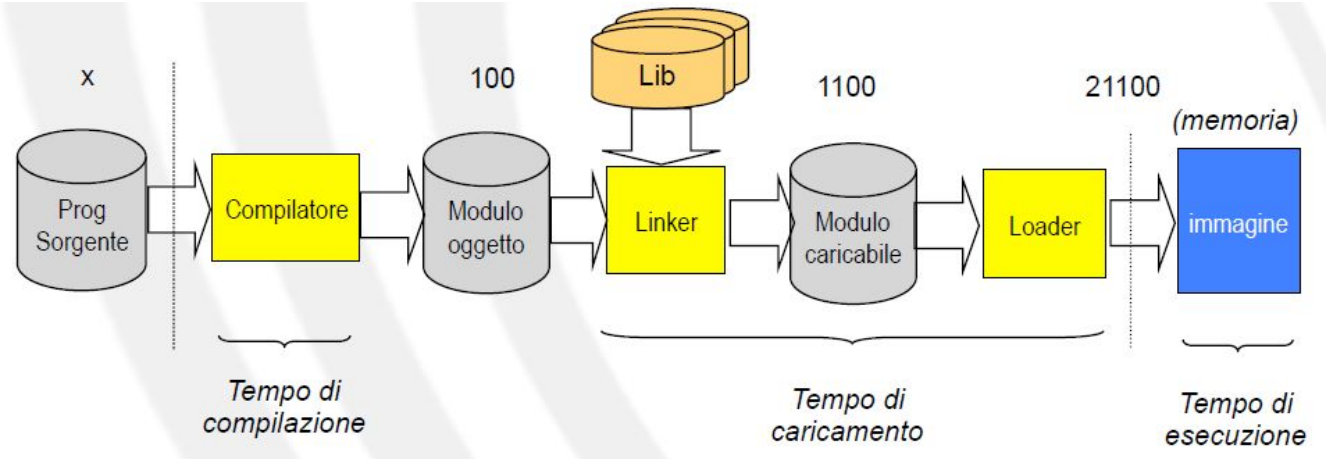
Ogni programma deve essere portato in memoria e **trasformato in processo** per poter essere **eseguito**:

- Le istruzioni vengono caricate dalla memoria in base al valore del program counter.
- L'istruzione può prevedere il caricamento di operandi dalla memoria.
- Al termine dell'esecuzione dell'istruzione il possibile risultato può essere scritto in memoria.
- Quando un processo termina, la sua memoria deve essere rilasciata.

Come avviene la trasformazione da programma a processo? Attraverso varie fasi precedenti all'esecuzione effettiva:

- SORGENTE: Gli **indirizzi di memoria** di un programma **sorgente** sono **simbolici**.
- COMPILAZIONE: in fase di **compilazione** gli **indirizzi simbolici** sono **associati** a **indirizzi rilocabili**, per poi essere **successivamente associati** a **indirizzi assoluti** dal loader o dal linker.

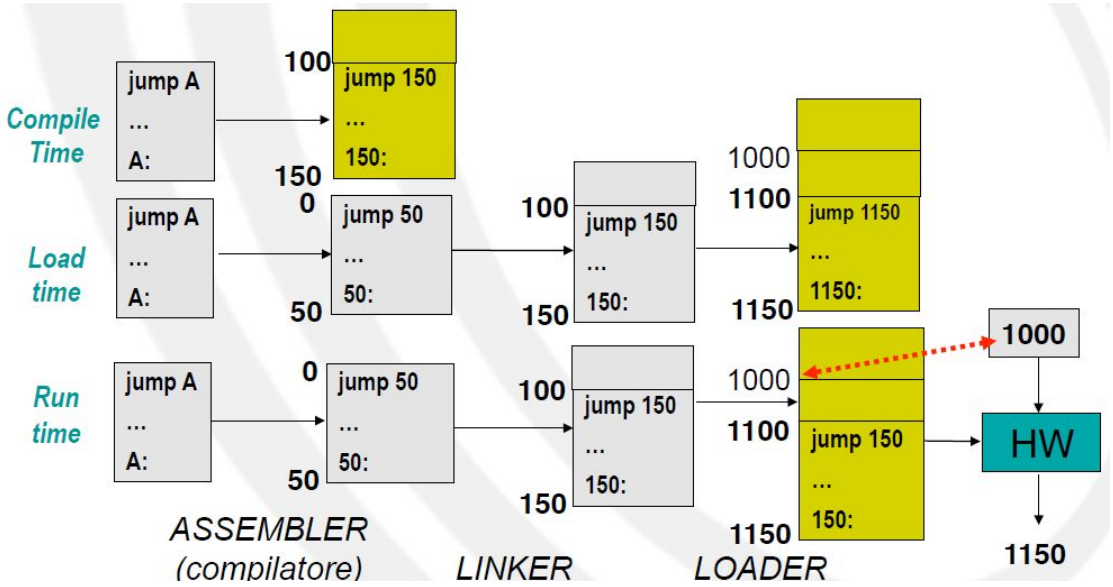
Il collegamento tra indirizzi simboli a indirizzi fisici è detto **binding**.



Binding degli indirizzi

Il binding degli indirizzi può essere effettuato in tre momenti distinti:

- **Compile time (statico):** deve essere **noto a priori** in **quale parte della memoria** risiederà il processo, se la locazione cambia, è necessario ricompilare il programma.
- **Load time (statico):** gli indirizzi vengono **impostati al caricamento del processo**, vengono utilizzati indirizzi relativi (es. n byte dall'inizio del programma). Se cambia l'indirizzo di riferimento è necessario ricaricare il programma.
- **Run time (dinamico):** Il binding degli indirizzi viene **posticipato a tempo di esecuzione**, il processo può essere spostato in aree diverse della memoria durante l'esecuzione. Per efficienza è necessario un supporto hardware.



Linking

Statico	Dinamico
Tutti i riferimenti sono definiti prima dell'esecuzione, l'immagine del processo contiene una copia delle librerie usate.	il linking delle librerie viene fatto a tempo di esecuzione, a differenza del linking statico, l'immagine del processo contiene solamente i riferimento alle librerie.

Loading

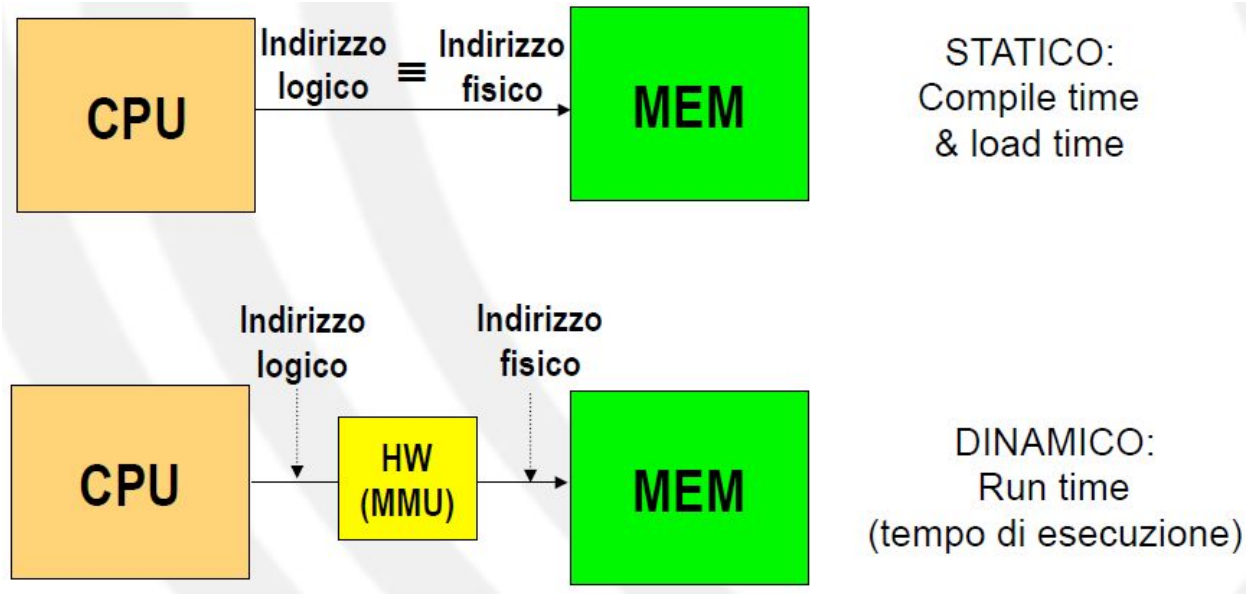
Statico	Dinamico
Tutto il codice è caricato in memoria a tempo di esecuzione.	Il caricamento viene fatto in corrispondenza del primo utilizzo, il codice non utilizzato non viene caricato.

Spazi di indirizzamento

Lo spazio di indirizzamento può essere **logico o fisico**:

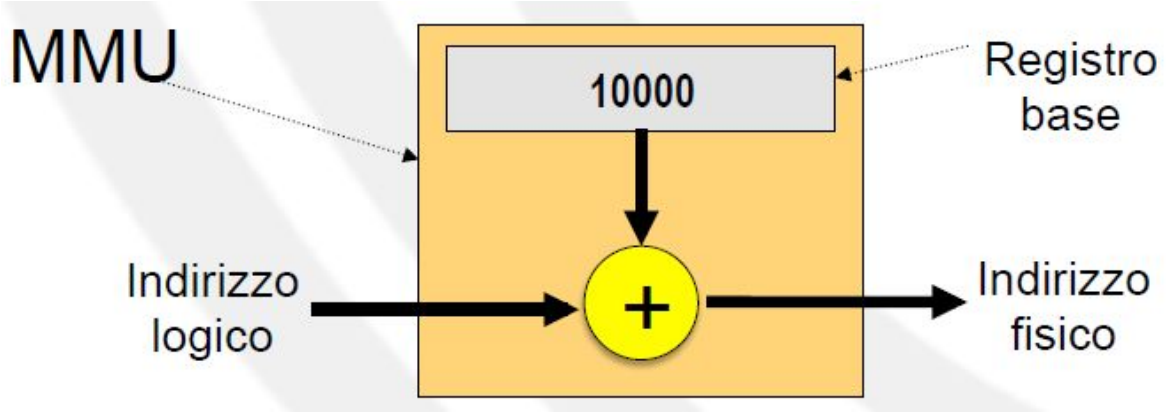
- **Indirizzo logico (o virtuale):** generato e utilizzato dalla CPU.
- **Indirizzo fisico:** indirizzo reale visto dalla memoria.

Durante il binding, se effettuato a compile time o loading time (quindi **statico**) **indirizzo logico e fisico coincide**, se effettuato a run time (**dinamico**) spesso **diverge**.



MMU (Memory Management Unit)

è un dispositivo hardware che **mappa indirizzi virtuali in indirizzi fisici**:



In un **sistema multiprogrammato, non è possibile sapere in anticipo dove un processo può essere posizionato in memoria** (**binding** a tempo di **compilazione impossibile**), e l'esigenza di avere lo **swap**, **impedisce** anche l'utilizzo del **binding a tempo di caricamento** (un processo che viene ricaricato dal disco non è detto che venga messo nella stessa area di memoria). **Due tipi** di rilocazione:

1. **Statica:** possibile solo per sistemi operativi per applicazioni specifiche, con una gestione della memoria limitata.
2. **Dinamica:** Utilizzata per sistemi operativi che hanno il pieno controllo della memoria.

Schemi di gestione della memoria

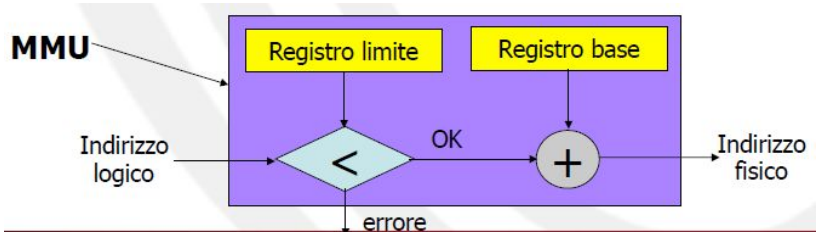
Sono principalmente 4 e prevedono che il programma sia interamente caricato in memoria (le soluzioni implementate nei sistemi operativi in realtà utilizzano la memoria virtuale)

Allocazione contigua

I processi sono allocati in memoria in posizioni contigue all'interno di una partizione, ovvero una **suddivisione** fissa o variabile **della memoria stessa**, che viene quindi vista come un insieme di partizioni di dimensioni predefinite tipicamente diverse.

- **Partizioni a dimensione fissa**
    - Assegnazione della memoria, effettuata dallo scheduler a lungo termine (prima di entrare in stato ready) utilizzando
        - **una coda per ogni partizione**, assegnando quindi un processo alla partizione più piccola in grado di contenerlo, ma permettendo però che ci siano job in altre code che non vengono caricati poiché di dimensioni troppo piccole rispetto alle partizioni disponibili
          OPPURE
        - **una coda singola per tutte**, con diverse politiche
            - **FCFS**, facile da implementare ma non sfrutta al meglio la memoria
            - **Best-fit-only**, scelta del job con dimensioni più simili alla partizione
            - **Best-available-fit**, scelta del primo job che può stare nella partizione

- **Supporto per la rilocazione dinamica**

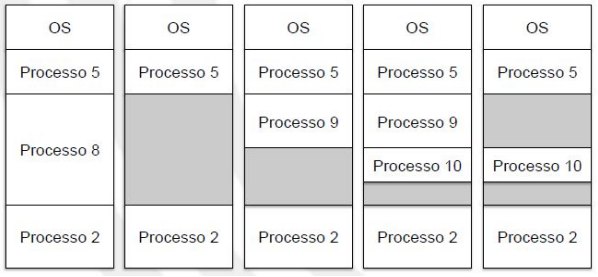


  - Semplice da realizzare
  - Il grado di multiprogrammazione è limitato al numero di partizioni
  - Si verifica frammentazione

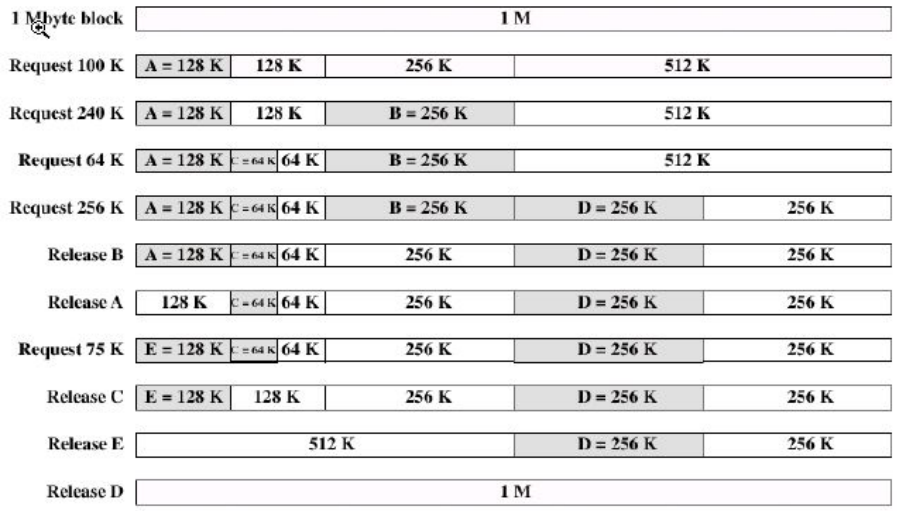
    INTERNA
    - interna alla partizione
    - quando il job è più piccolo della partizione

    ESTERNA
    - quando ci sono partizioni vuote che non soddisfano i job in attesa

- **Partizioni a dimensione variabile**, di dimensioni **identiche alle necessità dei processi** (per **eliminare la frammentazione interna**)
  - Assegnazione della memoria, vista come un insieme di buche (ovvero blocchi di memoria disponibile),
    - il sistema operativo mantiene dati sulle partizioni allocate e sulla loro disposizione.
    - All'arrivo di un processo, gli viene assegnata una buca che lo possa contenere.



    - **First-fit**, alloca la prima buca sufficiente, **tipicamente la migliore.**
    - **Best-fit**, alloca la **più piccola buca sufficiente**
      - richiede scansione della lista.
      - **minimo spreco**, lascia buchi piccoli dopo l'allocazione.
    - **Worst-fit**, alloca la **buca più grande**
      - richiede la scansione della lista.
      - **massimo spreco**, lascia buchi grandi dopo l'allocazione.
  - Supporto per la rilocazione dinamica
    - come per partizioni fisse, col vantaggio che non si ha frammentazione interna (per costruzione) ma solo frammentazione esterna
- **Riduzione della frammentazione**
  - **Compattazione**, ovvero spostare la memoria per rendere contigui i blocchi, possibile solo con rilocazione dinamica poiché richiede la modifica del registro base
  - **Tecnica del *buddy system***: la memoria viene vista come una serie di liste di blocchi di dimensione $2^L<2^k<2^U$, con L dimensione minima e U massima; all'inizio la memoria è un unico blocco e quando un processo parte, la memoria viene suddivisa in due finché non si ottiene un blocco abbastanza grande da poter contenere il processo. In questo modo la compattazione è semplicissima ma al costo di avere frammentazione interna sul blocco di minor dimensione.



Paginazione

è una tecnica per **eliminare la frammentazione esterna**, permette che lo spazio di indirizzamento fisico di un processo sia non continuo, e quindi di **allocare la memoria fisica dove essa è disponibile**.
Suddivisione della memoria:

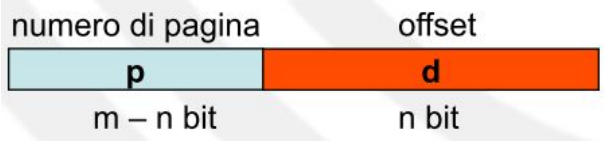
- **Fisica:** divisa in blocchi di grandezza fisica, detti frame (tipicamente di dimensioni comprese tra 512 byte e 8 kbyte).
- **Logica:** divisa in blocchi della stessa dimensione dei frame, detti pagine.

Vincoli della paginazione:

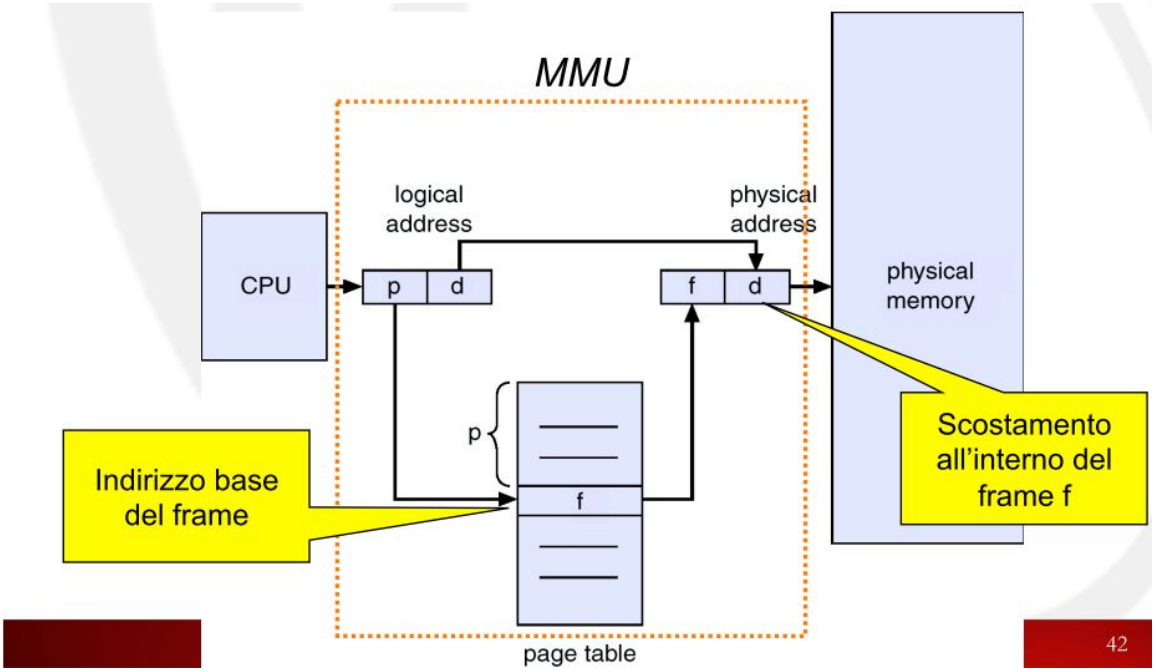
- Per eseguire un programma avente dimensione di n pagine, bisogna trovare n frame liberi.
- Si utilizza una tabella delle pagine, per ogni processo, per mantenere traccia di quale frame corrisponde a quale pagina. Viene inoltre utilizzata per tradurre indirizzo logico in fisico.

Traduzione degli indirizzi:

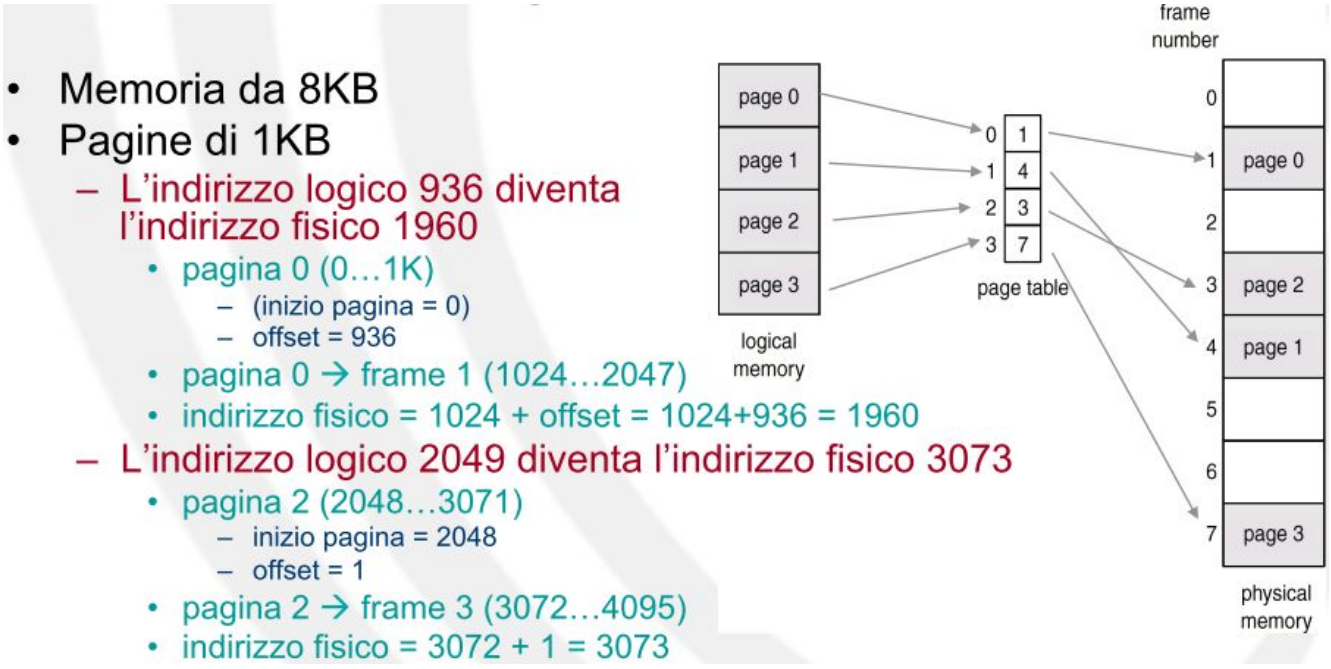
- L'indirizzo generato dalla CPU viene suddiviso in due parti:
  - **Numero di pagina(p):** usato come indice nella tabella delle pagine che contiene l'indirizzo base di ogni frame.
  - **Offset(d):** combinato con l'indirizzo base definisce l'indirizzo fisico che viene inviato alla memoria.
- Se la dimensione della memoria è 2^m e quella di una pagina è 2^n (parole/byte), l'indirizzo è suddiviso così:



Traduzione degli indirizzi:



Esempio di traduzione degli indirizzi:



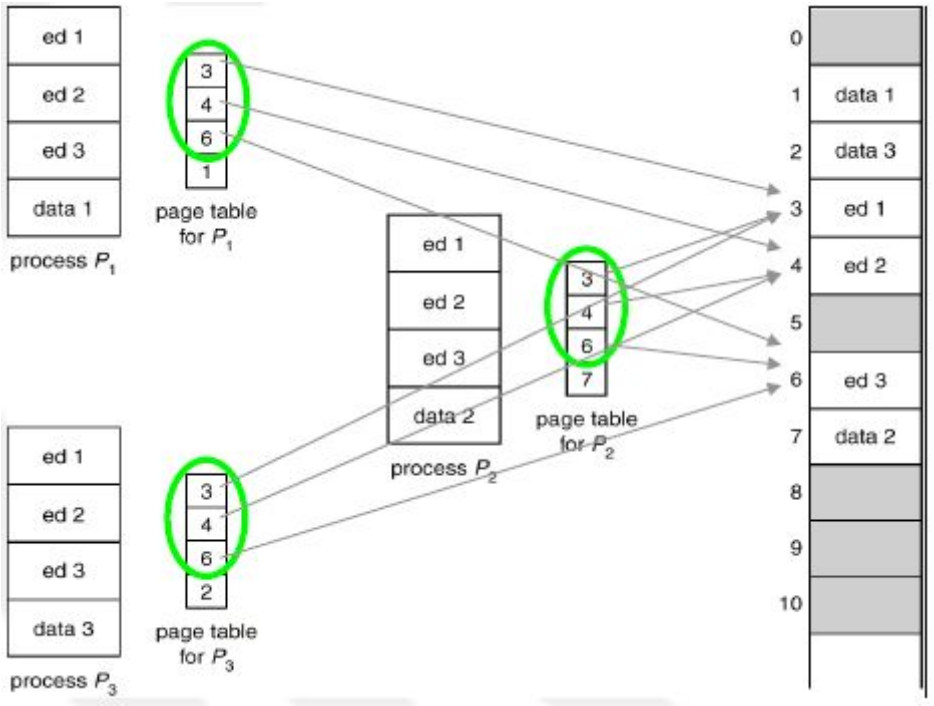
L'efficienza è fondamentale per l'implementazione della **tabella delle pagine**, ci sono diverse soluzioni implementative:

- **Registri:** le entry della tabella sono **mantenute nei registri della CPU**.
  - Efficiente.
  - Solo per numero di entry limitate.
  - **Allunga** i tempi di **context switch**.
- **Memoria:** la tabella delle pagine risiede **in memoria**, e vengono utilizzati i registri:
  - Page-table base register (**PTBR**), che **punta** alla **tabella delle pagine**
  - Page-table length register (**PTLR**), che contiene la **lunghezza della tabella**, opzionale
  - **Context switch** più **breve**, salvataggio solo dei due registri

- **Ogni accesso** a dati e istruzioni richiede però **due accessi a memoria** (prima alla tabella delle pagine, poi all'effettivo dato)
  - Questo problema si può risolvere tramite la cache **TLB** (**T**ranslation **L**ook-aside **B**uffers), che confronta l'elemento fornito con il campo chiave di tutte le entry contemporaneamente.
  - Ad **ogni context switch** il **TLB** viene **pulito** per evitare mapping errati
  - Durante un **accesso in memoria**, viene **prima** controllato il **TLB**: se la pagina è al suo **interno**, allora viene ritornato **subito il numero di frame** (con un singolo accesso, solo al TLB, solitamente in meno del 10% del tempo senza TLB), **altrimenti** è necessario accedere alla **tabella delle pagine**
  - **Effective Access Time**: $\mathbf{EAT = (T_{MEM} + T_{TLB})a + (2T_{MEM} + T_{TLB})(a - 1)}$
    con ***a*** probabilità di cache hit, T_x tempo di accesso a x e 2 indica il numero di livelli della paginazione (2 => 1 livello, 3 => 2 livelli, ...)

Protezione e condivisione:

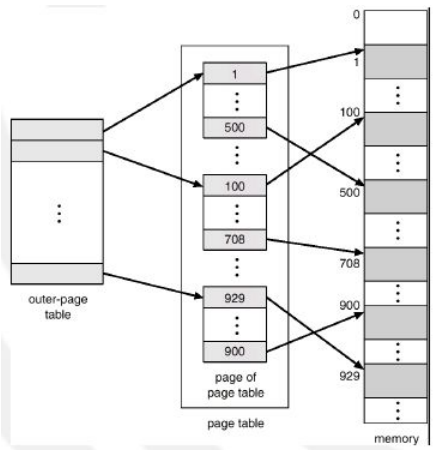
- Viene associato un **bit di protezione** ad **ogni frame**
  - **valid bit**: per ogni entry della tabella, per indicare se la pagina è associata o meno nello spazio di indirizzamento logico del processo
  - **bit di accesso**: r/w/x
- Il **codice viene condiviso fisicamente** ma logicamente ci sono **più copie**, **una per processo**
- Il codice read-only può essere condiviso (es. compilatori, window manager, ...)
- I dati solitamente sono diversi per ogni processo (più copie fisiche e logiche)



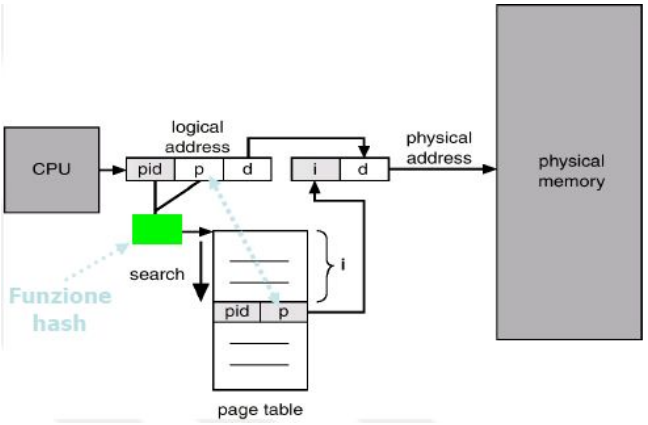
Problema dello spazio di indirizzamento:

Nelle architetture a 32 e 64 bit lo spazio virtuale è enorme rispetto a quello fisico (2^{32} o 2^{64}), che significa **tantissime entry possibili per ogni processo**; è quindi necessario gestire il problema della dimensione della tabella delle pagine:

- **Paginazione multilivello**, ovvero paginare la tabella delle pagine: solo alcune pagine sono memorizzate sulla memoria, le altre sono su disco. Solitamente si utilizzano 2, 3 o 4 livelli.



    - La **conversione logico-fisico** può richiedere **4 accessi** a memoria (per paginazione a 3 livelli), poiché **ogni pagina porta ad un'altra** pagina al livello successivo
    - TLB mantiene comunque grandi prestazioni
- **Tabella delle pagine invertita**, ovvero **un'unica tabella** per **tutto** il **sistema** (non per processo), avente una entry per ogni frame, contenente:
    - indirizzo virtuale della pagina che occupa quel frame.
    - informazioni sul processo che usa quella pagina.
    - più di un indirizzo virtuale può corrispondere allo stesso fisico, è necessario cercare il valore desiderato, usando una tabella hash.



Segmentazione

La memoria viene divisa come l'utente la immagina: un **programma** è un **insieme di segmenti**, che sono **unità logiche di dimensione variabile** che rappresentano rispettivamente main, procedure, funzioni, variabili, stack, …

- Gli **indirizzi logici** sono composti dalla tupla **<numero di segmento, valore di offset>**, dove il secondo campo indica dove all'interno di quel segmento andare a cercare il dato
- Per tenere traccia dei segmenti si usa la **tabella dei segmenti** simile alla tabella delle pagine, che mappa associa al **numero di segmento** un **indirizzo base** e un **offset massimo (limite)**. Viene gestita in memoria con due registri:
    - Segment-table base register (STBR), punta alla locazione della tabella in memoria
    - Segment-table length register (STLR), indica di quanti segmenti è composta la tabella
        - un indirizzo logico <s, d> è valido se d < STLR
        - BASE(s) + STBR indica l'indirizzo fisico dove recuperare l'elemento richiesto
    - Viene anche usato il TLB per memorizzare le entry maggiormente usate

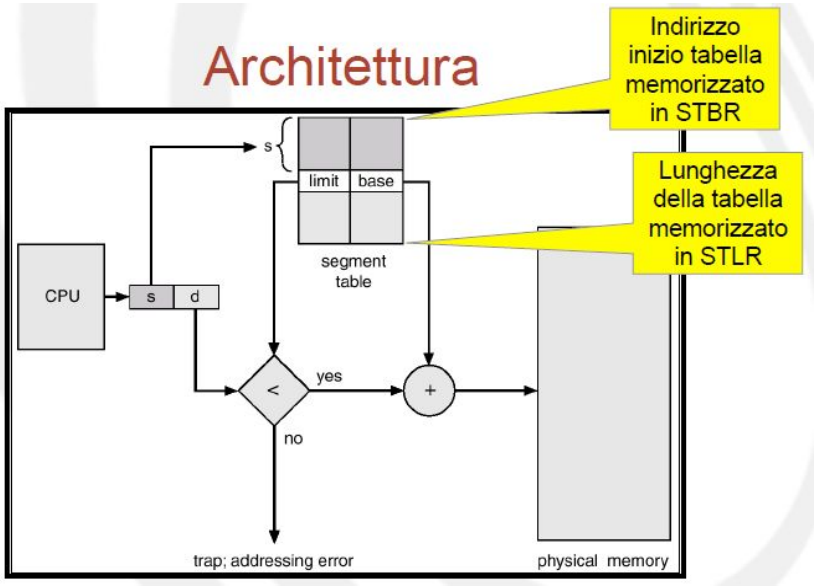


Tabella dei segmenti

Segmento	Limite	Base
0	600	219
1	14	2300
2	100	90
3	580	1327

- **Protezione**: supportata naturalmente, essendo un segmento un'entità semantica che rappresenta un insieme logico (es. dati, istruzioni, ...). Ad ogni segmento sono associati:
    - bit di modalità (r/w/x)
    - valid bit (che indica se un segmento è legale o no)

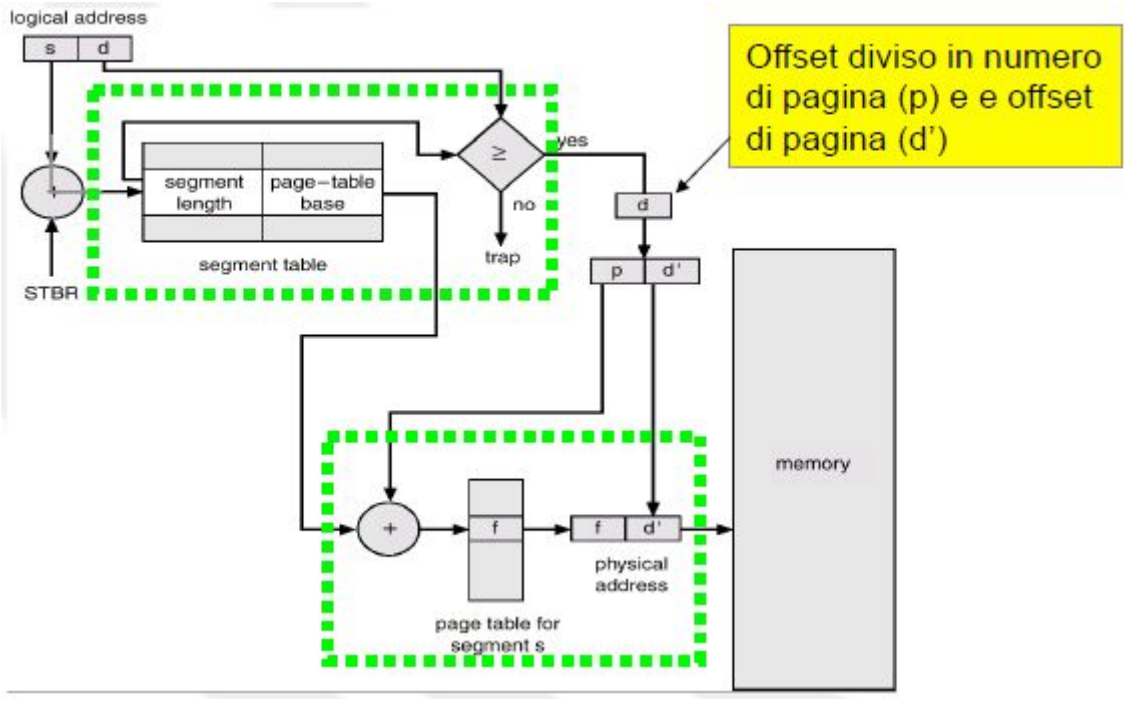
- **Condivisione**: a livello di segmento, **se si vuole condividere qualcosa** (parti di programma, funzioni di libreria) **basta metterlo nello stesso segmento**; chi vuole utilizzare quelle informazioni deve solo poter accedere a quello specifico segmento
- **Frammentazione**: il S.O. deve allocare spazio per tutti i segmenti di un programma, che hanno lunghezza variabile
    - L'allocazione dinamica viene effettuata tramite **first-fit o best-fit**
    - C'è la **possibilità di frammentazione esterna**, soprattutto nei segmenti grandi

Paginazione	Segmentazione
Vantaggi ● paginazione interna nulla ● allocazione frame semplice, non necessita di algoritmi complessi	Vantaggi ● consistenza tra vista utente e vista fisica della memoria ● protezione e condivisione dei segmenti
Svantaggi ● separazione vista utente e fisica	Svantaggi ● richiesta della allocazione dinamica dei segmenti ● potenziale frammentazione esterna

Segmentazione paginata

Usando una **combinazione delle due tecniche precedenti**, si ottiene uno **schema migliorato**, dove ogni segmento viene "paginato", ovvero suddiviso in pagine:

- ogni segmento possiede una propria tabella delle pagine
- la tabella dei segmenti contiene l'indirizzo base delle tabelle delle pagine di ogni segmento
- si **elimina** il **problema dell'allocazione dei segmenti** e della **frammentazione esterna**



Memoria Virtuale

con gli schemi di memoria visti precedentemente, un intero programma deve essere caricato in memoria per poter essere eseguito, anche se questo non è tuttavia strettamente necessario (ne basta solo una parte). Se riusciamo a **non mantenere in memoria tutto il nostro programma**, ma solamente la parte interessata, possiamo tenere una **spazio di indirizzi virtuali** molto più grande dello **spazio degli indirizzi fisici**, e più processi possono essere mantenuti in memoria.

Memoria virtuale = Memoria Fisica + Disco:

- Possibilità di "**swappare**" pagine da e verso la memoria e non l'intero processo.
- Possibilità di dividere la memoria logica (utente) dalla memoria fisica.

Implementazioni:

- Demand paging
- Demand segmentation

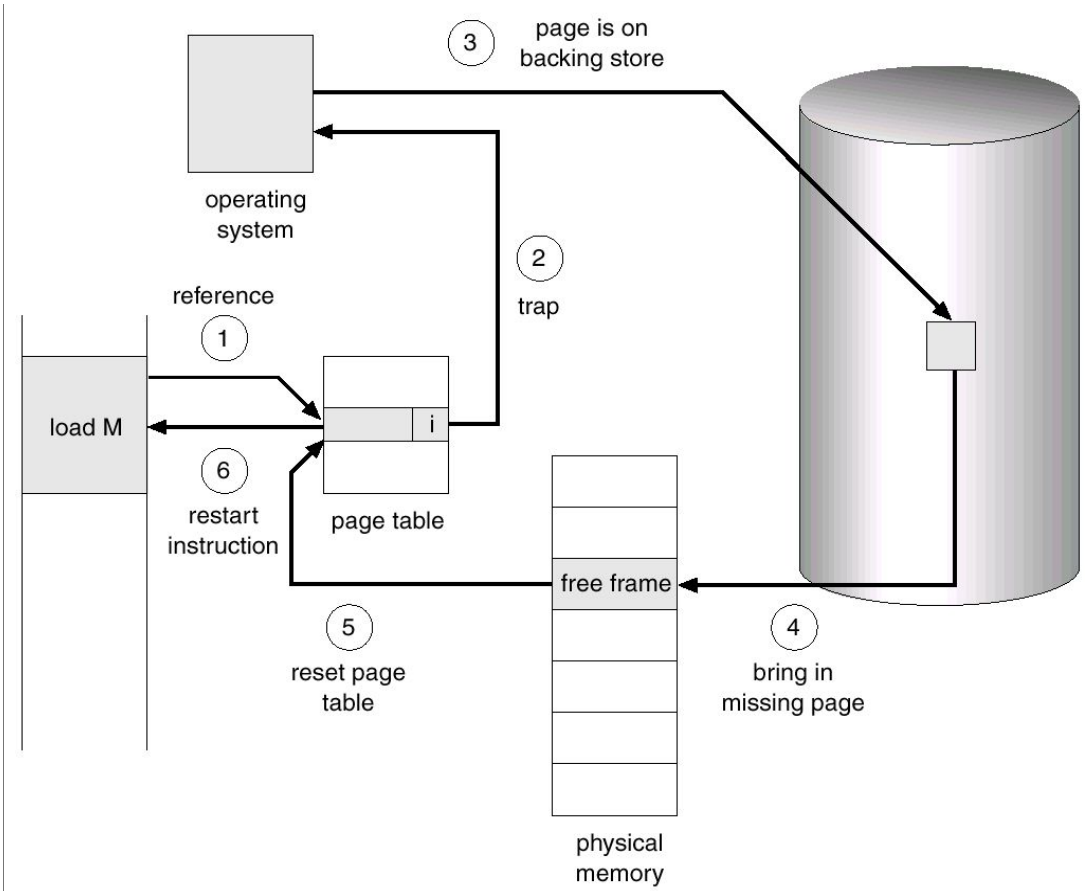
Demand paging

Una pagina viene **caricata in memoria solo quando è necessario.**

- **Meno richieste I/O** quando è necessario lo swapping = **risposte più rapide**.
- Meno memoria utilizzata = Più processi possono condividerla.
- Necessario sapere lo **stato di una pagina**: in memoria o no?
    - A **ogni pagina è associato un bit** (**valid** = 1 = in memoria/**invalid** = 0 = non in memoria).
    - Inizialmente impostati a 0.
    - Se durante la traduzione logico/fisico, si trova una pagina con il bit a zero, si ha un **page fault**.

Un **page fault = un interrupt** al sistema operativo che:

1. **Verifica la tabella associata al processo** (se il riferimento è valido attiva il caricamento della pagina)
2. **Cerca un frame vuoto** in memoria dove caricare la pagina.
3. Esegue lo **swap dal disco verso la memoria centrale**.
4. **Setta il bit** associato alla pagina **a 1** nella tabella del processo.
5. **Ripristina l'istruzione** che ha causato il page fault.
   Attenzione: il **primo accesso** in memoria di un programma risulterà **sempre in un page fault**!



Prestazione di demand paging

La **demand paging influenza il tempo effettivo** di accesso alla memoria principale (**EAT**=**E**ffective **A**ccess **T**ime). Il tasso di page fault è compreso tra 0 <= p <= 1, dove p=0, significa nessun page fault, mentre p=1, ogni accesso in memoria è un page fault.
Calcolo dell'EAT:

$$\mathbf{EAT = (1 - p) * t_{mem} + p * t_{page\ fault}}$$

Dove $t_{page\ fault}$ è dato da 3 componenti principali:

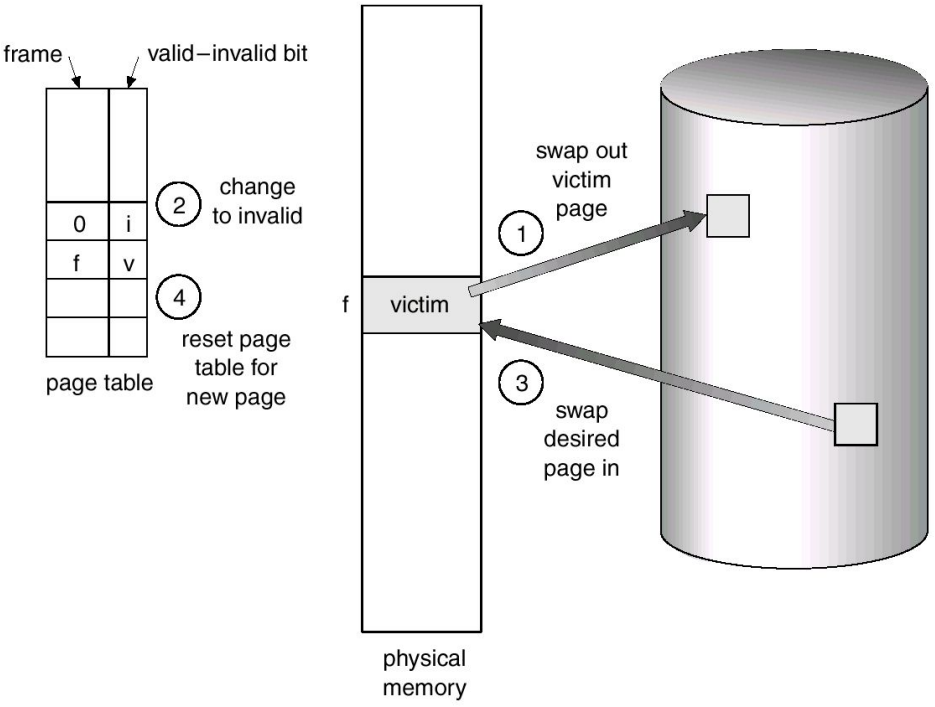
- Servizio dell'interrupt
- Swap In (lettura della pagina dal disco)
- Costo del riavvio del processo
- Swap out [opzionale, non sempre eseguito]

t_{mem} = 100ns
$t_{page\ fault}$ = 1 ms (10^6 ns)
EAT = (1 – p)*100 + p*10^6 =
= 100 –100*p + 1000000*p = 100*(1 + 9999 p) ns
Per mantenere il peggioramento entro il 10% rispetto al tempo di accesso standard:
– 100 * (1.1) > 100*(1 + 9999p) ➡ p < 0.0001 ≈ 10^{-4}
– 1 page fault ogni 10000 accessi!

Rimpiazzamento delle pagine

In caso di mancanza di spazio per poter effettuare lo swap in di una pagina, è **necessario cercare tra le pagine in memoria da rimpiazzare** e farne swap out su disco prima di sovrascriverle. L'algoritmo di ricerca delle pagine deve avere come **obiettivo l'ottimizzazione delle prestazioni** e la **minimizzazione** del numero di **page fault**.
Gestione dei page fault **in caso di assenza di frame:**

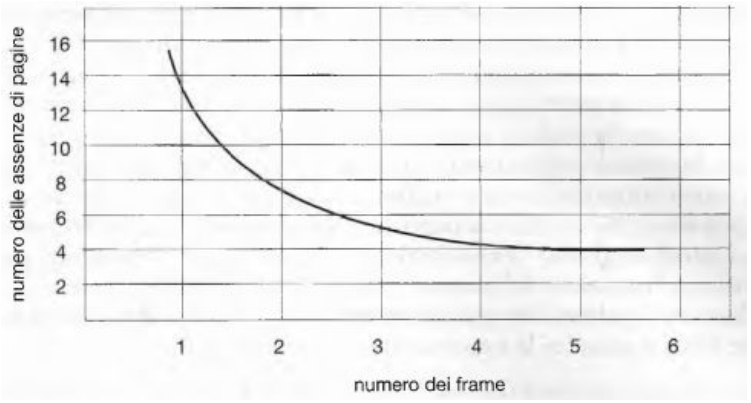
1. Il sistema operativo **verifica la tabella associata al processo** per controllare che non sia una violazione di accesso.
2. **Inizia la ricerca di un frame vuoto.**
    a. **Se esiste**, si passa la **punto 4**, altrimenti si invoca l'algoritmo per scegliere una vittima.
3. **Swap out** della **vittima** su disco.
4. **Swap in** della **pagina** nel frame **da disco**.
5. **Modifica delle tabelle** (page table, bit di validità).
6. Viene **ripristinata l'istruzione** che ha causato il page fault.



In **assenza di frame liberi**, si hanno **due accessi a memoria**, uno per fare lo swap out della vittima e uno per fare lo swap in per il frame da caricare, il risultato è che ho **il tempo di page fault raddoppiato**.
Una possibile ottimizzazione è utilizzare una tabella delle pagine modificate mentre sono state in memoria (dirty bit).

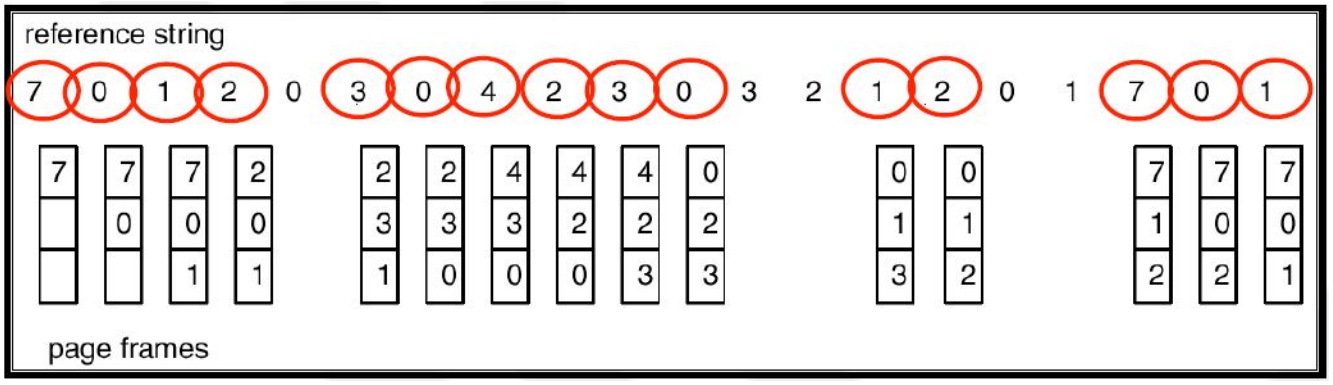
Algoritmi di rimpiazzamento delle pagine

Vogliamo **ridurre al minimo il tasso di page fault**, e il tasso di page fault è **inversamente proporzionale al numero di pagine.**



FIFO (First in First out)

La **prima pagina introdotta** è la **prima** a essere **rimossa**. **Algoritmo "cieco"**, non valuta l'importanza della pagina che va a rimuovere e tende ad aumentare il tasso di page fault. **Soffre dell'anomalia di Belady.**



Con una memoria di **3 frame**, si hanno **15 page fault**.

Anomalia di Belady

è un **fenomeno che si presenta in alcuni algoritmi di rimpiazzamento delle pagine** di memoria per cui la frequenza dei **page fault** può **aumentare** con il numero di frame assegnati ai processi. Ne **soffrono** gli **algoritmi FIFO** con alcune combinazioni di richieste di pagina.
Spesso all'**aumentare dei frame, il numero di page fault addirittura aumenta** invece di diminuire.

- Reference string

1	2	3	4	1	2	5	1	2	3	4	5

- Con 3 frame 9 page fault

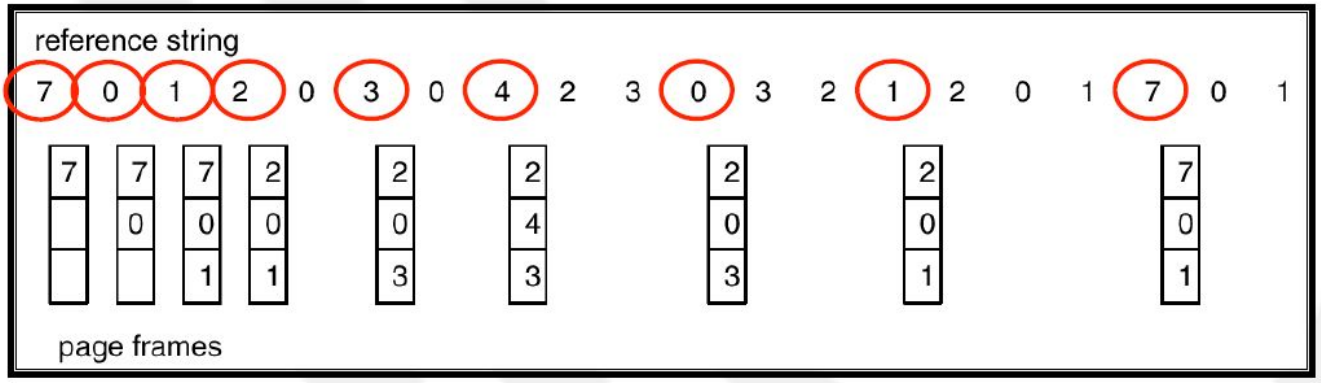
1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4

- Con 4 frame 10 page fault

1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3

Qual'è allora il nostro **algoritmo ideale**?

- Deve garantire ovviamente il **numero minimo di page fault**.
- **Rimpiazza** le **pagine** che **non saranno usate** per il periodo di tempo più lungo.
- Richiede **conoscenza anticipata** della stringa dei riferimenti.



Viene usato come riferimento per gli altri algoritmi.

LRU (Least Recently Used)

Può essere visto come un' **approssimazione dell'algoritmo ottimo**. Usa il **recente passato come previsione del futuro**, e rimpiazza la pagina che non viene usata per più tempo (*). Il risultato è meglio di FIFO ma peggio, ovviamente, dell'ideale.

1	2	3	4	1	2	5	1	2	3	4	5

1	1*	1*	1*	1	1	1	1	1	1	1*	5
	2	2	2	2*	2	2	2	2	2	2	2
		3	3	3	3*	5	5	5	5*	4	4
			4	4	4	4*	4*	4*	3	3	3

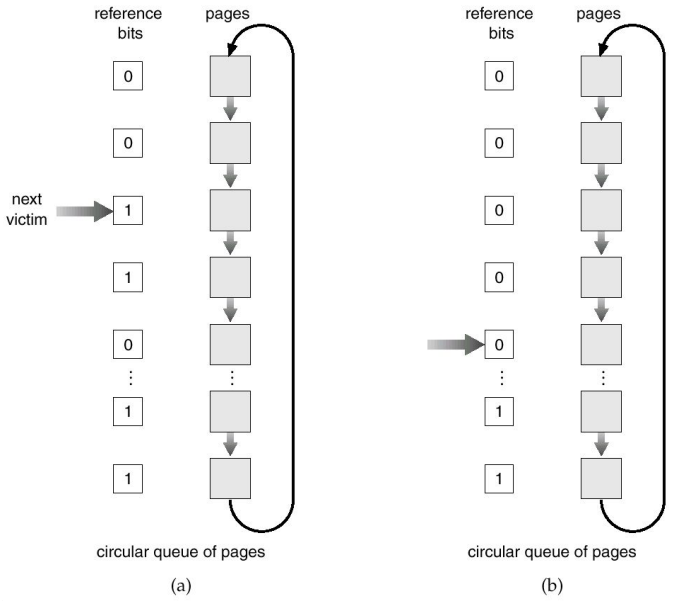
4 frame -> 8 page fault

Il problema risulta ora trovare una **buona implementazione**, che riesca a **calcolare** il tempo dell'**ultimo utilizzo** velocemente e senza troppo HW addizionale:

- **Tramite contatore:** ad **ogni pagina** è associato **un contatore**; ad ogni referenza ad una pagina, il clock di sistema è copiato nel contatore, quindi viene **rimpiazzata** la **pagina** con il **valore più piccolo**, ma prima bisogna cercarla.
- **Tramite stack:** si mantiene uno **stack di numeri di pagina** e ad ogni riferimento ad una pagina, questa **viene messa in cima allo stack**; l'aggiornamento richiede l'estrazione dallo stack, il fondo dello stack è la pagina LRU, quindi non è necessaria alcuna ricerca, ma **richiede supporto HW** per la modifica dello stack e della copia del clock.

Approssimazione dell'algoritmo LRU:

- **Uso di un reference bit**:
  - Associato ad ogni pagina e **inizialmente pari a 0**.
  - Quando una pagina viene referenziata, viene messo a 1 dall'hardware.
  - Per il rimpiazzamento viene scelta una pagina che ha il bit a 0.
  - Approssimato: non viene verificato l'ordine di riferimento delle pagine.
- **(Alternativa) Reference bit:**
  - Uso di più bit di reference (**registro a scorrimento**) per ogni pagina.
  - I bit vengono **aggiornati periodicamente**.
- **Rimpiazzamento second chance:**
  - Basato su bit di reference.
    - Bit a **1**, **metti** a **0**, ma **lascia** la **pagina** in memoria.
    - Bit a **0**, **rimpiazza** la pagina.
  - Di fatto una FIFO circolare.



- **Altre tecniche basate sul conteggio:**
  - Algoritmo **LFU** (**L**east **F**requently **U**sed):
    - Mantiene un conteggio con il numero di riferimenti fatti, rimpiazza la pagina con il conteggio più basso.
  - Algoritmo **MFU** (**M**ost **F**requently **U**sed):
    - Opposto di LFU: si presuppone che la pagina con un conteggio basso sia stata appena caricata, e quindi deve ancora essere usata.

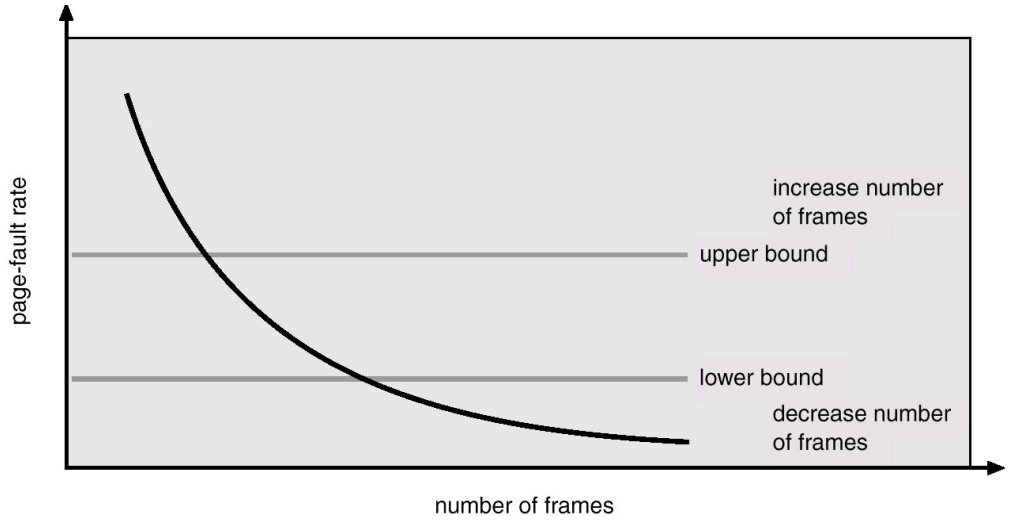
Allocazione dei frame

In base al numero di frame e processi, è importante scegliere bene quanti frame allocare ad ogni processo, poiché ogni processo necessita almeno di un minimo numero di frame per essere eseguito e quel valore è pari al massimo numero di indirizzi specificabile in una istruzione (es. "move" ha bisogno di almeno 6 pagine per essere eseguita)

- Gli schemi di allocazione possono essere
  - Fissa, un processo ha sempre lo stesso numero di frame
    - Allocazione equitaria: dati *m* frame e *n* processi, ogni processo ha m/n frame

        - Allocazione proporzionale: in base alla dimensione del processo; dati S dimensione totale dei processi, s_i dimensione del processo p_i e a_i numero di pagine per il processo p_i, si ha che $a_i = \frac{s_i}{S} \times m$

    - Variabile, il numero varia durante l'esecuzione; in base a cosa modifico?
        - **Calcolo del working set**, ovvero calcolare in qualche modo quali sono le richieste effettive di ogni processo
            - Ad ogni processo viene assegnato un numero di frame sufficiente per mantenere in memoria il suo working set, ma questo è un valore che cresce man mano che il tempo passa
            - Si può verificare un fenomeno noto come thrashing: se il numero di frame allocati ad un processo scende sotto un certo minimo, il tasso di page-fault tende a crescere, quindi:
                - Si abba l'utilizzo della CPU perché alcuni processi sono in attesa di gestire il page-fault
                - Il S.O. tende ad aumentare il grado di multiprogrammazione aggiungendo altri processi tra quelli in esecuzione
                - I nuovi processi "rubano" frame a quelli vecchi, aumentando il grado page-fault e ricominciando il circolo vizioso
            - Quindi come misuro il working set? Approssimo usando un timer per interrompere periodicamente la CPU e all'inizio di ogni periodo pongo tutti i reference bit a 0. Ad ogni interruzione del time, le pagine vengono scandite e se il reference bit è pari a 0 (cioè non sono state usate in questo periodo) allora vengono scartate
        - **Calcolo del PFF** (Page Fault Frequency), una soluzione più accurata, che si prepone di stabilire un tasso page-fault accettabile per ogni processo
            - Se il numero di page-fault di un processo è troppo alto, allora quel processo ottiene più frame; se è troppo basso allora deve rilasciarne perché ne ha troppi

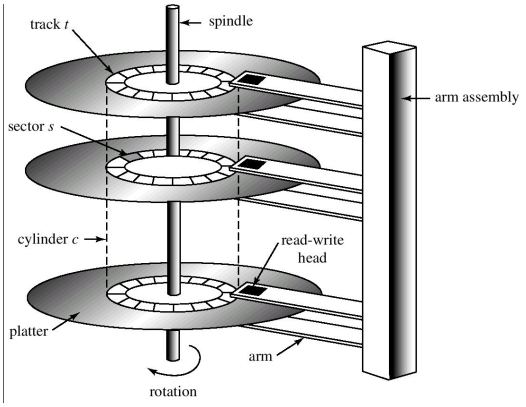


- In caso di page-fault, la pagine vittima da sacrificare viene scelta in base a
    - Rimpiazzamento locale: ogni processo seleziona una vittima tra i suoi frame
    - Rimpiazzamento globale: un processo sceglie un frame fra tutti quelli presenti
- Altre considerazione: alcuni frame non vanno mai rimpiazzati (es. quelli del kernel o quelli per I/O) e bisogna scegliere accuratamente la dimensione delle pagine in base a svariati fattori quali frammentazione, dimensione della page table, località e I/O overhead.

Memoria Secondaria e File System

La memoria secondaria è prettamente adibita al mantenimento dei dati dopo lo spegnimento del calcolatore; nel tempo ha subito molti miglioramenti ma le principali categorie sono:

- **Nastri magnetici**: ad accesso sequenziale, quindi molto lenti, ora usati solo per backup
- **Dischi magnetici**: una testina viene sospesa sopra un piatto di alluminio ricoperto di materiale ferromagnetico; la testina viene usata sia per la scrittura (induzione da parte della testina di corrente positiva o negativa che magnetizza la superficie) che per la lettura (induzione da parte della superficie di corrente nella testita)
    - **Il disco è diviso in tracce circolari**, suddivise in settori, l'unità più piccola di informazione (tra 32 byte e 4 KB, solitamente 512 byte)
    - Vengono raggruppati logicamente dal S.O. in **cluster (o blocchi)** per aumentare l'efficienza; un file occupa sempre almeno un cluster.
    - **L'accesso** a un settore avviene **specificando** la **superficie**, la **traccia** e il **settore** stesso.
    - $\mathbf{t_{accesso} = t_{seek} + t_{latency} + t_{trasfer}}$ **[ms]**
        - **Seek time**: tempo per spostare la testina.
        - **Latency time**: tempo per posizionare il settore sotto la testina, in base alla velocità di rotazione; **= 30/rpm.**
        - **Transfer time**: tempo per la lettura/scrittura elettrica del dato dal/al settore; dim.blocco/vel.trasferimento.
    - Il **seek time** è il **fattore dominante**, quindi è necessario minimizzarlo tramite algoritmi di scheduling degli accessi al disco, massimizzando la banda di trasferimento.



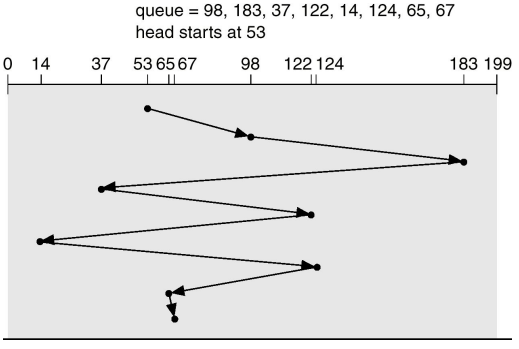
- **Dispositivi a stato solido**: usa chip (DRAM o flash NAND) per memorizzare in modo non volatile utilizzando la stessa interfaccia dei dischi fissi, ma con le seguenti caratteristiche:
    - **Meno soggetti a danni meccanici**, ma hanno un limite sul numero di write (isteresi).
    - Silenziose (nessuna parte meccanica).
    - **Più efficienti** nel transfer time, ma più costose.
    - Non necessitano deframmentazione.

Scheduling degli accessi a disco

Un disco è visto come un vettore unidimensionale di blocchi logici, dove il settore 0 è il primo settore della prima traccia del cilindro più esterno e la numerazione procede per settori, tracce e infine cilindri.

Quando un processo richiede accesso al disco, il S.O. salva la richiesta (solitamente contenete tipo di accesso R/W, indirizzo di memoria di destinazione, quantità di dati da trasferire, ...) in una coda di I/O, che è necessario gestire in modo da ridurre al minimo il seek time

- **FCFS**: le richieste sono processate nell'**ordine di arrivo**.
- **SSTF** (Shortest Seek Time First): è la **scelta più efficiente**, infatti seleziona la richiesta col minimo spostamento rispetto alla posizione attuale, ma aumenta il rischio di starvation per le richieste distanti.
- **SCAN** (algoritmo dell'ascensore): la **testina parte da un'estremità e si sposta verso l'altra**, servendo le



richieste quando passa sul settore indicato durante i viaggi di andata e ritorno.

- **CSCAN**: **come SCAN** ma **circolare**, quindi raggiunta un'estremità ritorna subito all'altra senza servire le richieste in mezzo.
- **LOOK** e **C-LOOK**: la testina **non per forza arriva fino ad un'estremità**, ma si gira (o riparte dall'inizio nel caso di C-LOOK) se non ci sono più richieste in quella direzione.
- **N-step-SCAN** e **FSCAN**: per evitare che la testina rimanga sempre nella stessa zona, la coda richieste viene partizionata in più code di dimensione massima N. Quando la testina va in una direzione, viene processata solo una coda piena e al prossimo giro si processa la prossima piena; FSCAN è una N-step-SCAN con solo 2 code.
- **LIFO** (Last In First Out): in alcuni casi (es. quando gli accessi sono sempre vicini) è meglio schedulare le richieste in **ordine contrario** a quello di arrivo, con la possibilità però di starvation.

Nessuno degli algoritmi sopra è ottimo (ne esiste uno ma è molto inefficiente a causa della elefata complessità di calcolo che necessita); **in generale però, SCAN e C-SCAN sono migliori** per sistemi con molti accessi al disco, ma solitamente il disk-scheduling è implementato come modulo del S.O. e sceglie di utilizzare un algoritmo piuttosto che un altro in base ad un'analisi delle richieste.

Gestione del disco

Il disco non è altro che un piatto di metallo che mantiene campi magnetici e per poter funzionare deve essere **formattato fisicamente**, ovvero deve subire un processo di divisione in settori che il controllore può gestire, leggere e scrivere (es. MBR o GPT). Il S.O. deve memorizzare all'interno del disco tutte le proprie strutture:

- Partizionamento del disco in più gruppi di cilindri (**partizioni**).
- **Formattazione logica** per la creazione di un **file system** (es FAT32, NTFS, EXT4, ...).
- Programma di boot per l'inizializzazione del sistema, che carica driver dei dispositivi essenziali (CPU, RAM, disco, GRAM) e lancia il sistema operativo.

Gestione blocchi difettosi

Ogni settore possiede un **campo** finale chiamato **ECC** (**E**rror **C**orrection **C**ode), che permette di capire s**e i dati in quel settore sono corretti o meno**; quando la testina passa per leggere il settore, legge

anche ECC, poi calcola un altro ECC con i dati appena letti e **se i due ECC sono diversi**, allora nel settore c'è un errore e viene definito **bad block**; va gestito:

- **Off-line**: durante la formattazione logica i bad block vengono individuati e inseriti in una lista; inoltre si possono eseguire utility per isolare i bad block (es CHKDSK).
- **On-line**: il controllore mappa il bad block su un blocco buono vuoto (**spare block**), così da rendere la cosa trasparente al S.O.; questo potrebbe influire sulle ottimizzazioni fornite dallo scheduling, ma il problema si risolve allocando spare block in ogni cilindro.

Gestione spazio di swap

Spazio su disco usato dalla memoria virtuale come estensione della RAM; può essere **ricavato dal file system** (tramite primitive di accesso a file, inefficiente) **oppure** può essere messo in una **partizione separata** (soluzione più tipica), totalmente esterna al file system, che usa un gestore apposito dell'area di swap (swap deamon). Lo spazio di swap può essere allocato ad un processo nel momento in cui viene creato oppure quando una pagina viene forzata fuori dalla memoria fisica.

File system

Fornisce il meccanismo e la memorizzazione e accesso a dati e programmi. Esso è composto da collezioni di file, e strutture di directory.

Interfaccia del file system

Il sistema operativo astrae le caratteristiche fisiche dei supporti di memorizzazione fornendone una visione logica.

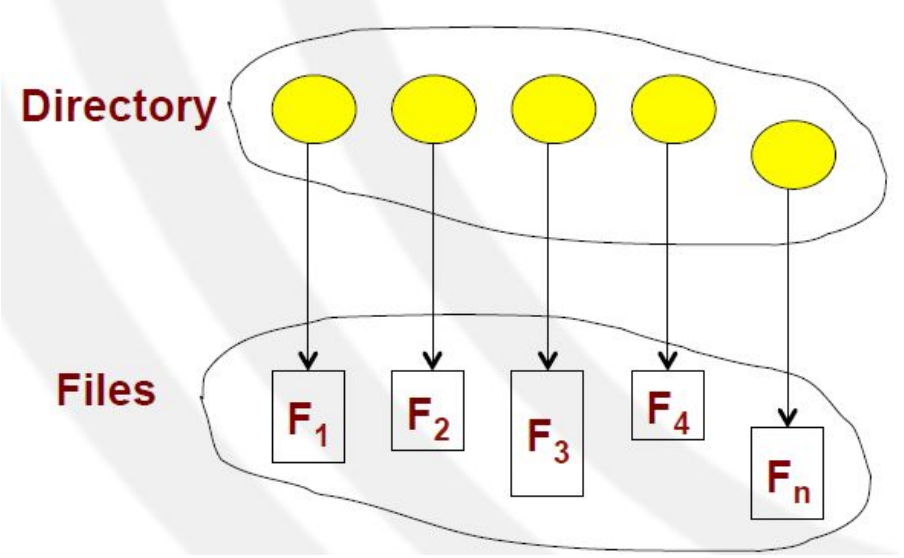
Concetto di file: un file è uno spazio di indirizzamento logico e contiguo, identificato da un nome, un una serie di attributi.

- Tipi: possono essere di tipo dati (numerici, binari, testuali, etc) o programmi.
- Attributi:
    - **Nome** (unica informazione in formato "leggibile")
    - **Tipo**
    - **Posizione** (puntatore allo spazio fisico sul dispositivo)
    - **Dimensione**
    - **Protezione** (controllo di chi può leggere, scrivere, eseguire)
    - **Tempo e data**
    - **Identificazione dell'utente**
- Operazioni:
    - **Creazione**: ricerca dello spazio su disco, creazione del nuovo elemento su directory per attributi.
    - **Scrittura**: System call che specifica nome del file e dati da scrivere, puntatore alla prossima area di memoria da scrivere.
    - **Lettura**: System call che specifica nome del file e dove mettere i dati letti in memoria, puntatore della prossima locazione da leggere.
    - **Riposizionamento all'interno del file system**: aggiornamento dell'attributo posizione corrente.
    - **Cancellazione**: libera lo spazio associato al file e l'elemento corrispondente nella directory.

    - **Troncamento**: mantiene nome e attributi inalterati, ma ne cancella il contenuto.
    - **Apertura:** Ricerca del file nella struttura della directory su disco, copia il file in memoria e inserisce un riferimento nella tabella dei file aperti.
        - Nei sistemi multiutente ci sono 2 tabelle per gestire i file aperti
            - Una per ogni processo
            - Una per tutti i file aperti
    - **Chiusura**: copia del file in memoria su disco.
- **Struttura di un file:** I tipo possono essere utilizzati per indicare la struttura interna del file
    - Nessuna: semplice sequenza di parole, bytes, etc.
    - A Record semplice: un record = una riga (a lunghezza fissa o variabile).
    - Complesso: documenti formattati, formati ricaricabili (load module).
- **Metodi di accesso:**
    - **Sequenziale:** permesso il **read next, write next e rewind**, non è permesso il rewrite, rischi inconsistenza dati (es. usato da editor e compilatori).
    - **Diretto:** sequenza numerata di blocchi, permesse il **read/write n, position to n, read/write next, rewrite n** (es. usato da database).

Struttura delle directory

Una **directory** è una **collezione di nodi** contenenti informazioni sui file, sono anch'esse residenti su disco e determinano la struttura del file system.



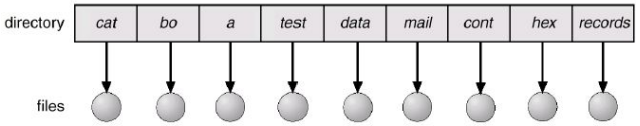
Una directory deve tenere traccia per ogni file:	**Operazioni sulle directory:**
• Nome • Tipo • Indirizzo • Lunghezza attuale • Massima lunghezza • Data di ultimo accesso • Data ultima modifica • Possessore • Info di protezione	• Visualizzarne i file contenuti • Aggiungere un file • Rinominare un file • Cancellare un file • Ricercare un file • Attraversare il file system

Una **directory** deve inoltre **permette un rapido accesso ai file**, fornire una **nomenclatura conveniente** agli utenti, **raggruppamento di file** per criteri.

Tipi di strutture

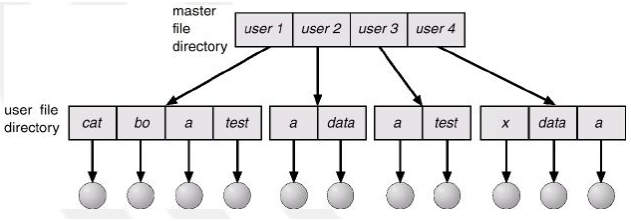
Directory a un livello

- Problemi di nomenclatura
- Problemi di raggruppamento



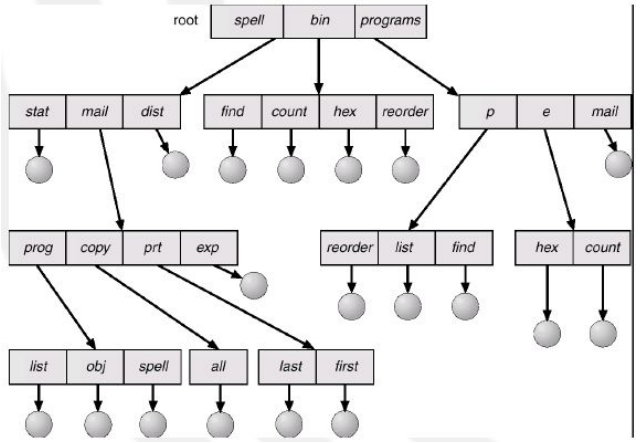
Directory a due livelli

- Concetto di percorso (path)
- Possibilità di usare lo stesso nome di file per utenti diversi
- Ricerca efficiente
- No raggruppamento



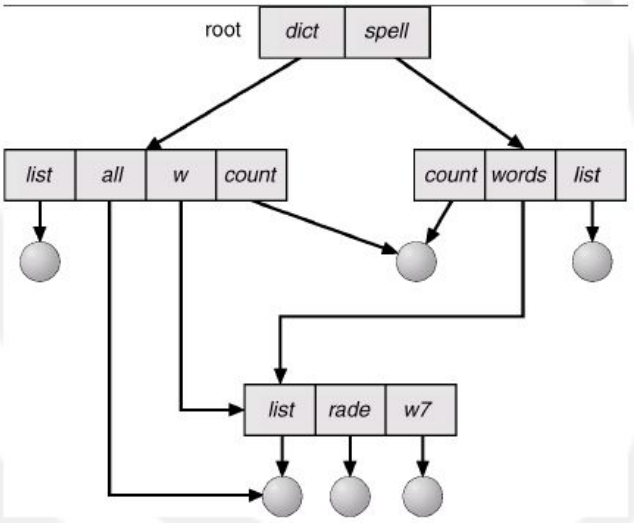
Directory ad albero

- Ricerca efficiente
- Possibilità di raggruppamento
- Concetto di directory corrente
- Percorsi assoluti o relativi



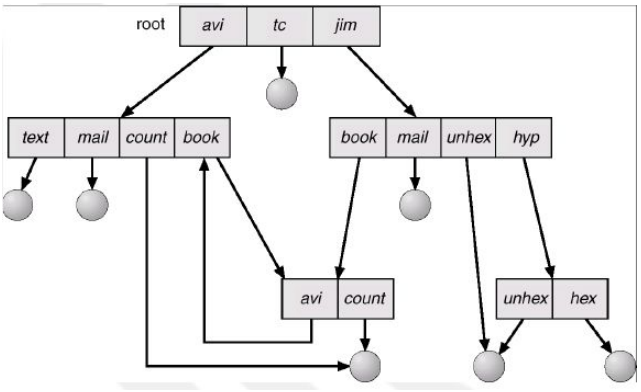
Directory a grafo aciclico

- Implementazione della condivisione grazie a link e collegamenti
    - Symlink
    - Hardlink



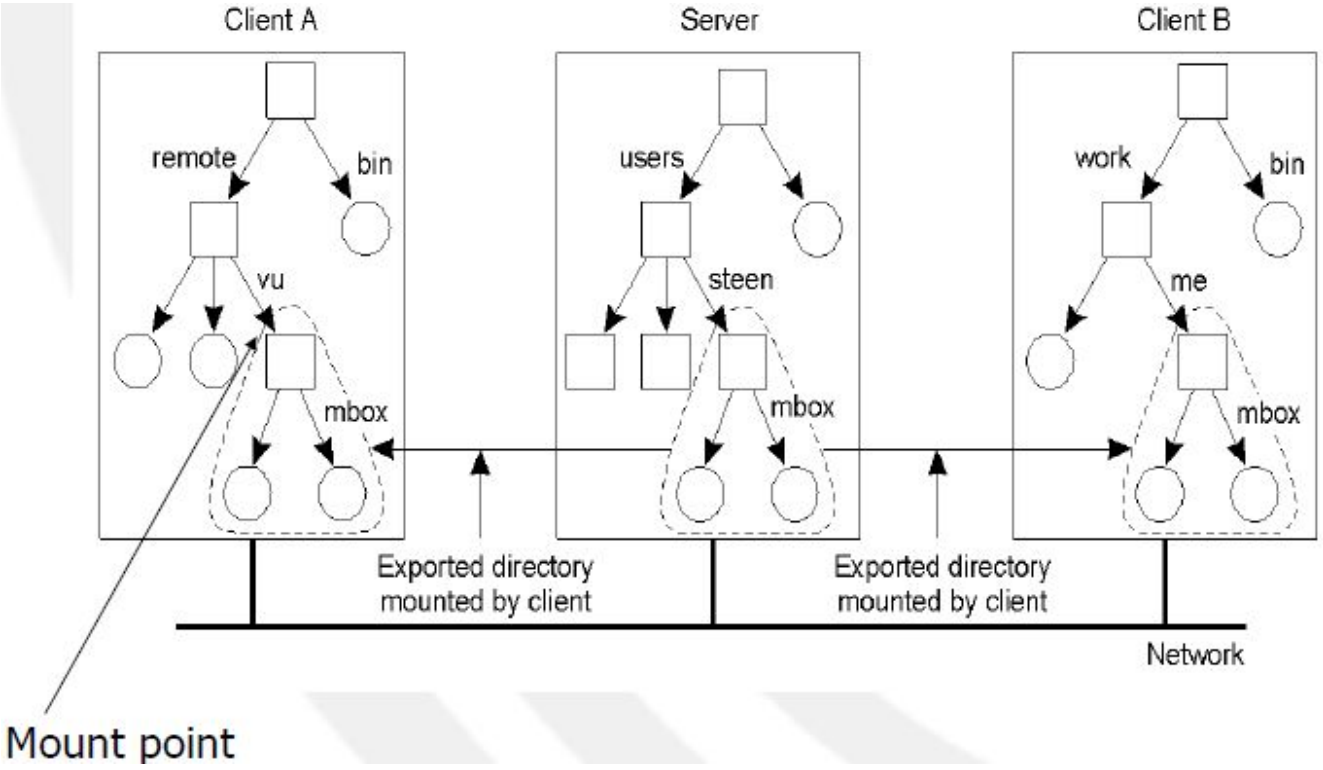
Directory a grafico generico

- Necessario garantire che non esistano cicli



Mount di file system

Permette la **realizzazione di file system modulari**, attaccando e staccando interi file system a file system preesistenti.



Condivisione di file

- Importante in sistemi multiutente, realizzabile tramite schemi di protezione.
- In sistemi distribuiti i file possono essere distribuiti tramite la rete.

Protezione

Il **proprietario** di un file deve poter **controllare cosa è possibile fare su un file e da parte di chi**. Alcuni tipi di **operazioni controllabili** sono:

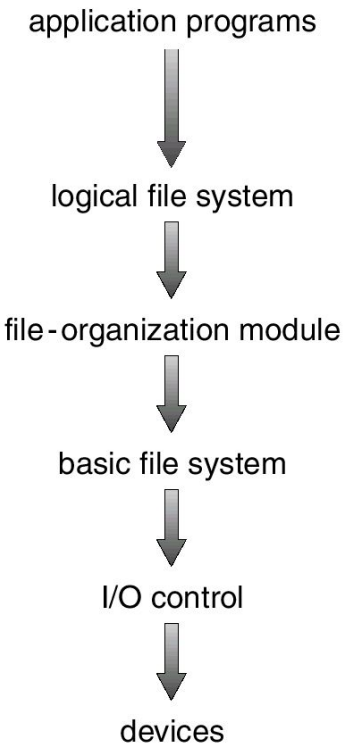
- **Lettura**
- **Scrittura**
- **Esecuzione**
- **Append**
- **Cancellazione**
- ...

La **lista d'accesso** per file e directory è l'elenco di chi può fare che cosa per ogni file e/o directory. Sotto UNIX sono **raggruppati in 3 classi:**

- **Proprietario**
- **Gruppo**
- **Altri**

Per ogni classe i permessi sono 3: **r,w,x** (**lettura**, **scrittura**, **esecuzione**)

Implementazione del File System



Un **file system** è composto da **diversi livelli**, dove **ogni livello sfrutta** le funzionalità del **livello inferiore**. Per gestire un file system, sono necessarie **diverse strutture dati**, parti di esse sono presenti **su disco**, altre **in memoria**. Le caratteristiche di uno specifico file system sono fortemente dipendenti dal sistema operativo e dal file system stesso.

Strutture su disco (memorizzati tipicamente in questo ordine):

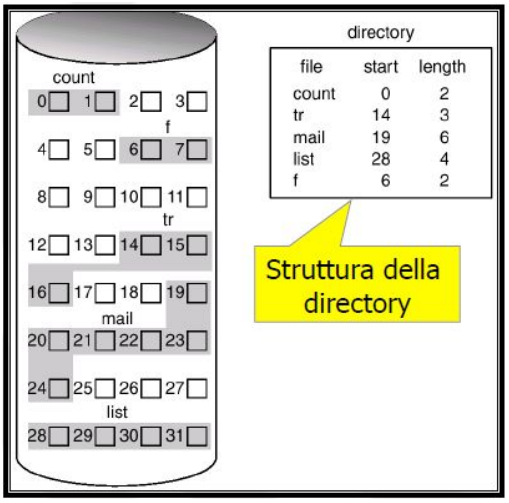
1. **Blocco di Boot:** contiene le informazioni necessarie per avviare il sistema operativo all'accensione della macchina.
2. **Blocco di controllo delle partizioni:** contiene tutte le informazioni delle partizioni, come numero e dimensioni dei blocchi, blocchi liberi, etc.
3. **Strutture di directory:** descrivono l'organizzazione dei file.
4. **Descrittori di file:** contengono le informazioni sui file e i puntatori ai blocchi dati.

Strutture in memoria:

- **Tabella delle partizioni:** contiene le informazioni delle partizioni montate
- **Strutture di directory:** Copia in memoria delle strutture a cui si è fatto l'accesso recentemente.
- **Tabella globale dei file aperti:** copie dei descrittori dei file.
- **Tabella dei file aperti per ogni processo:** puntatore alla tabella precedente ed informazioni di accesso.

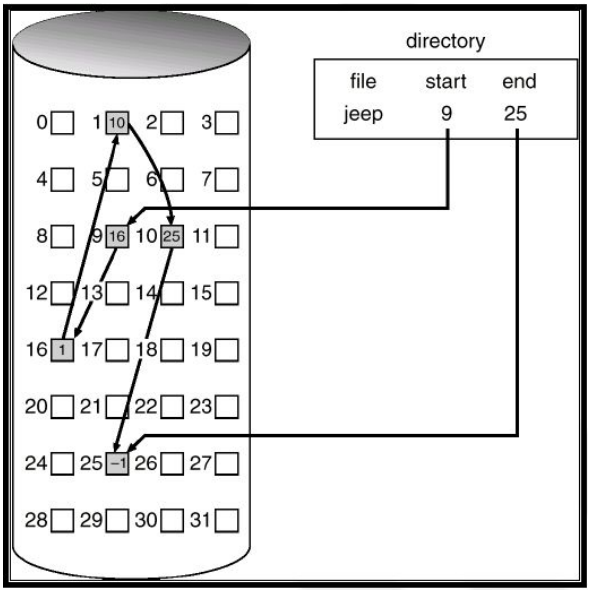
Come allochiamo lo spazio su disco a file e directory (blocchi)? Il nostro **obbiettivo** è quello di **minimizzare i tempi di accesso ai dati** e **massimizzare l'utilizzo dello spazio** (evitare sprechi). Abbiamo diverse alternative:

- **Allocazione contigua:** ogni file occupa un insieme di **blocchi contigui su disco.**
  - Entry della directory semplice: composto dall'indirizzo del blocco di partenza, e la lunghezza (quindi dal numero di blocchi).
  - **Vantaggi:**
    - Accesso semplice, il blocco b+1 non richiede di spostare la testina rispetto al blocco b.
    - Supporta l'accesso casuale e sequenziale, rispetto un blocco b, basta fare b+i, e saltare al blocco corrispondente.



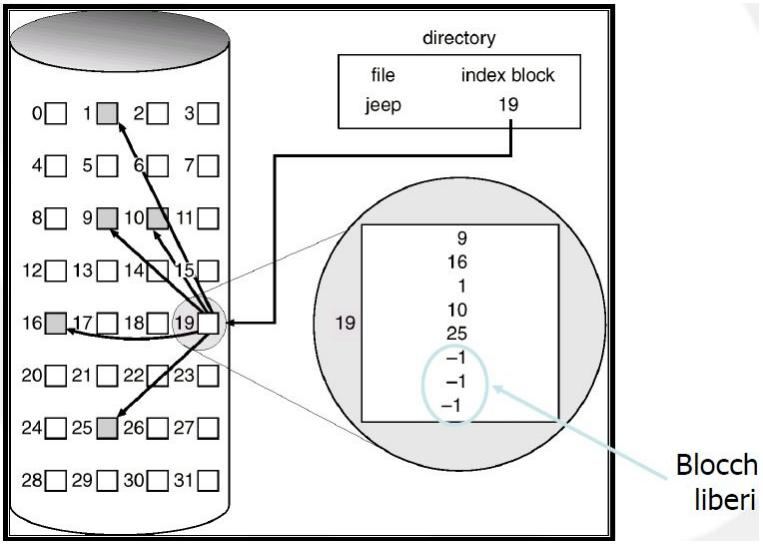
  - **Svantaggi:**
    - Presenta problemi simili a quelli dell'allocazione dinamica della memoria, quindi l'utilizzo di algoritmi best-fit, first-fit, etc.
    - Frammentazione esterna, quindi spreco di spazio (risolvibile deframmentando periodicamente).
    - I file non possono crescere dinamicamente, quindi o si sovrastima la dimensione alla creazione (sprecando spazio), o si trova un buco più grande copiando tutto il file (rallentando il sistema).

- **Allocazione a lista concatenata:** ogni file è una lista di blocchi, che possono essere sparsi ovunque nel disco, perchè ogni blocco contiene il puntatore al blocco successivo.
    - Indirizzamento:
        - X=indirizzo logico;
        - N=dimensione del blocco;
        - X / (N-1)=numero del blocco nella lista;
        - X % (N-1)=offset all'interno del blocco.



    - **Vantaggi:**
        - Creazione semplice di un nuovo file (basta cercare un blocco libero, e creare la entry della directory che punta a quel blocco)
        - Estensione di un file semplice (basta aggiungere blocchi in coda)
        - Nessuna frammentazione esterna.
    - **Svantaggi:**
        - Accesso casuale impossibile, bisogna scorrere tutti i dati a partire dal primo.
        - Richiede tanti riposizionamenti della testina, quindi scarsa efficienza.
        - Scarsa affidabilità, in quanto se un puntatore va corrotto si perde l'intero file.

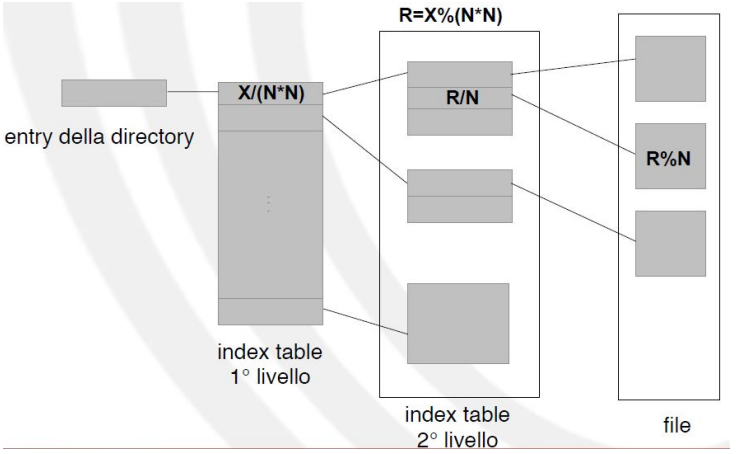
- **Allocazione indicizzata:** ogni file ha un blocco indice che contiene la tabella degli indirizzi dei blocchi fisici, mentre la directory contiene l'indirizzo al blocco indice del file.
    - Indirizzamento:
        - X=indirizzo logico;
        - N=dimensione del blocco;
        - X / N = offset nella index table.
        - X % N = offset all'interno del blocco dati



    - **Vantaggi:**
        - Accesso casuale efficiente.
        - Accesso dinamico senza frammentazione esterna (solo l'overhead del blocco index per la index table)
    - **Svantaggi:**
        - La dimensione del blocco limita la dimensione massima del file
            - Con una dimensione del blocco di 512 parole, la dimensione massima del file è 512^2 parole = massimo 256 K parole per file.

Come fare ad avere file senza una dimensione massima con l'allocazione indicizzata? Ci sono diversi **schemi a più livelli:**

- Indici multilivello: una tabella più esterna contiene puntatori alle index table.
    - X=indirizzo logico;
    - N=dimensione del blocco;
        - X / (N*N) = blocco della index table di 1° livello.
        - X % (N*N) = R.
    - R
        - R / N = offset nel blocco della index table di 2° livello.
        - R % N = offset nel blocco dati.
    - ES: con blocchi da 4 KB, posso fare 1K indici da 4 byte
        - Con due livelli posso fare file da 4 GB massimo.



- **Schema concatenato:** è una lista concatenata di blocchi indice.
    - L'ultimo degli indici di un blocco indice punta a un'altro blocco indice.
    - X = indirizzo logico;
    - N = dimensione del blocco;
        - X / (N(N-1)) = numero del blocco indice all'interno della lista dei blocchi indice.
        - X % (N(N-1)) = R
    - R
        - R / N = offset nel blocco indice.
        - R % N = offset nel blocco dati.
    - ES: dimensione blocco N = 1KB, quindi massimo 256 parole da 4 byte l'una; un file è grande massimo 256 blocchi, quindi massimo 256KB. Se l'indirizzo logico è X = 12200, allora il file è alla parola X%N=168 all'interno del blocco X/N=12200/256=47. Se l'indirizzo logico è invece X = 644000 non basta un primo livello, infatti X/N=2515>256 (blocco massimo); quindi serve trovare il blocco della lista concatenata X/(N(N-1))=9, ottenere R=X%(N(N-1))=56480 e con questo valore calcolare l'offset nel blocco index R/N=220 e l'offset nel blocco dai R%N=160.
- **Schema combinato**, ovvero uno schema in cui ci sono blocchi diretti, altri sono a uno, due , tre livelli o più, in base alla priorità e l'importanza dei file

Implementazione delle directory

Tutti i meccanismi visti per memorizzare un file viene impiegato anche per memorizzare le directory. Le **directory** di per sé **non contengono dati**, ma **sono** delle **liste di file** (o altre directory) che essa contiene.

Come possono essere implementate?

- **Lista regolare di nomi di file con puntatori ai blocchi dati:** implementazione semplice ma poco efficiente (lettura, scrittura e rimozione richiedono una ricerca per trovare il file).
- **Tabella hash:** tempi di ricerca migliori, ma con possibilità di collisioni in caso di due nomi di file nella stessa posizione.

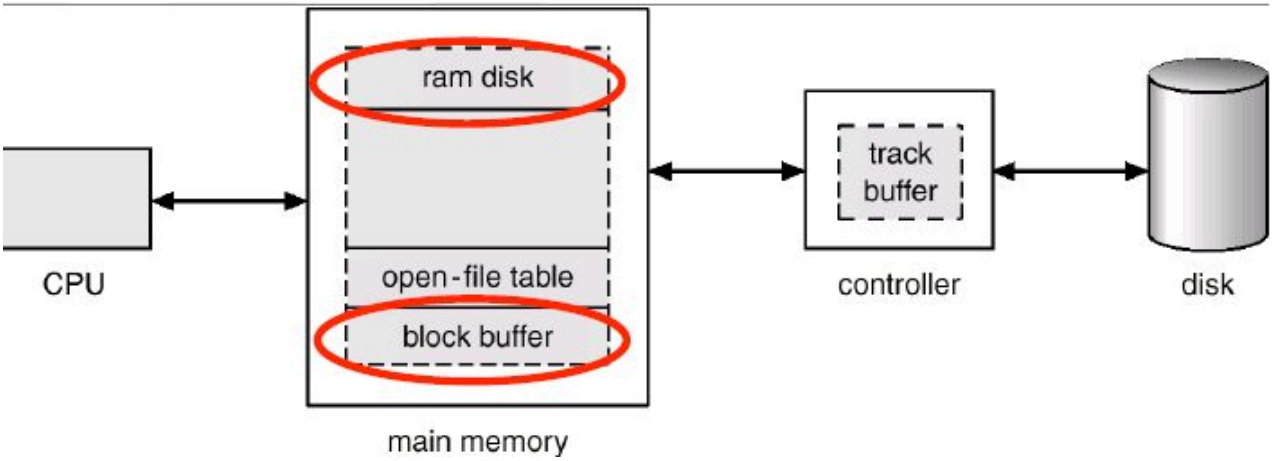
Gestione dello spazio libero

Per tenere traccia dello spazio libero su disco si usa una **lista di blocchi liberi,** dove per creare nuovi file basta rimuovere i corrispettivi blocchi dalla lista, per eliminarli basta aggiungerli alla lista.
Implementabile con:

- **Vettore di bit:** si utilizza un vettore di bit per ogni blocco, dove 1=libero e 0=occupato; per trovare il primo blocco libero si cerca il primo bit a 1 nel vettore.
  - La mappa di bit richiede uno spazio extra per essere mantenuta.
  - Efficiente solo se è mantenibile tutta in memoria.
  - Facile ottenere file contigui.
- **Lista concatenata:**
  - Spreco minimo di spazio (solo per la testa della lista).
  - Spazio contiguo non ottenibile.
- **Raggruppamento:**
  - Modifica della lista concatenata e simile alla allocazione indicizzata.
  - Fornisce rapidamente un gran numero di blocchi liberi.
- **Conteggio:**
  - Mantiene il conteggio di quanti blocchi liberi seguono il primo in una zona di blocchi liberi contigui.
  - Generalmente la lista risulta più corta.

Efficienza e prestazioni

Nel sistema **il disco è il collo di bottiglia principale**, e la sue efficienza è estremamente importante. Essa dipende dall'algoritmo di gestione dello spazio libero e dai tipi di dati contenuti nelle directory (ad esempio quando accedo a un file devo aggiornare anche la data di accesso nella directory dove esso è contenuto).
Il controllore del disco contiene un **piccola cache** in grado di contenere una **traccia intera**, ma esso da solo non basta per garantire prestazioni elevate, quindi è possibile fare ricorso a **dischi virtuali** (**RAM disk, volatili**) o **buffer cache** (vengono **mantenuti in memoria i blocchi usati di frequente**, simile alla cache tra CPU e RAM, sfrutta anch'esso il **principio della località spaziale e temporale**).

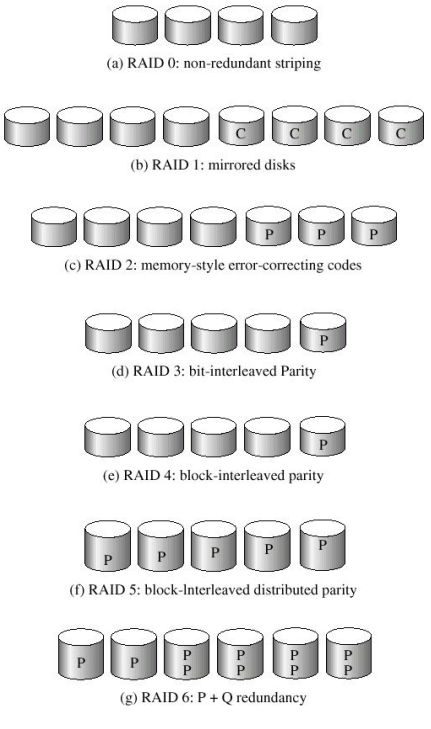


File system log structured
Registrano ogni cambiamento come una transazione, e ogni transazione è scritta a log in maniera asincrona. Una transazione viene considerata completata quando è scritta a log, se il sistema va in crash, le transazioni non avvenute sono quelle memorizzate ancora nel log.

RAID

Con il tempo, **i dischi** sono diventati **sempre più piccoli ed economici**, quindi è stato possibile equipaggiare i sistemi con un numero sempre maggiore di dischi. Ciò ha portato alla creazione della tecnica denominata **RAID** (**R**edundant **A**rray of **I**ndependent **D**isks), che permette di **aumentare le prestazioni** (R/W in parallelo) e garantisce una **maggiore affidabilità** (ridondanza).

- Implementazioni
  - **Struttura SW:** la funzionalità RAID è implementata dal S.O., i dischi sono tutti collegati allo stesso bus
  - **Struttura HW:** un controllore intelligente nella macchina gestisce i dischi collegati alla macchina
  - **Batteria RAID:** un'unità separata dalla macchina è collegata a tutti i dischi e funge da controllore e cache, rendendo trasparente la presenza di più dischi alla macchina e facendolo risultare come unico
- **Le tecniche RAID sono basate su due semplici operazioni:**
  - **Mirroring**: copiatura speculare dei dati, aumenta l'affidabilità introducendo ridondanza, memorizzando informazioni utili per ricostruire i dati persi in caso di guasto.
    - La perdita di dati (in caso di puro mirroring) avviene solo se entrambi i dischi si guastano insieme e il tempo medio di perdita dei dati dipende dal tempo medio di guasto di un disco e dal tempo medio di riparazione.
      - ES. tempo di guasto disco = 100000 ore; tempo di riparazione = 10 ore: tempo di perdita dati con puro mirroring = $100000^2/(2*10)$ = 57000 anni
    - I guasti non sempre sono indipendenti tra i dischi (es. disastri naturali, cali di tensione, difetti di fabbricazione sul lotto di produzione dei dischi).
    - Miglioramento delle prestazioni in lettura su file diversi, ma non in scrittura perché bisogna mantenere il mirroring.
  - **Data striping**: sezionamento dei dati su più dischi, aumenta le prestazioni poiché permette la lettura in parallelo su più dischi di parti dello stesso dato, che poi vengono riaccorpate in ordine per restituire il dato nella sua interità
    - Sezionamento a bit: i byte sono distribuiti su più dischi
    - Sezionamento a blocco: i blocchi sono distribuiti su più dischi
    - Si aumenta la produttività per gli accessi multipli a picche porzioni di dati, riducendo il tempo di risposta per gli accessi a grandi quantità di dati
- **Livelli RAID**, ovvero l'unione delle due tecniche base viste sopra con l'inclusione di tecniche basate sui bit di parità, che permettono di **identificare dati persi** e, con l'aggiunta di più bit supplementari, ricostruirli o correggere gli errori:
  - **RAID 0:** block **striping** senza ridondanza:
    - economico, alte prestazione grazie al parallelismo tra i dischi.
    - senza ridondanza, l'affidabilità diminuisce all'aumentare dei dischi utilizzati (con 4 dischi, basta che se ne rompa anche solo uno per perdere tutti i dati, l'affidabilità è ¼ rispetto a quella di un disco solo).
  - **RAID 1:** pure **mirroring** senza striping:
    - affidabilità aumenta linearmente in base al numero di dischi, aumento delle prestazioni in



lettura (finché un disco è occupato, posso leggere da un altro, tanto i dati sono uguali).
        - alto costo, bassa scalabilità.
    - **RAID 2:** block **striping** con **parità a bit** su **più dischi**
        - i bit di correzione errori vengono salvati in dischi diversi da quelli usati per i dati
        - se un disco si guasta, i bit ECC vengono usati per ricostruire i dati persi
        - ad ogni lettura viene controllata la parità e ad scrittura va calcolata, tedioso
        - RAID 2 usa solo 3 dischi in più per 4 dischi dati, contro i 4 richiesti da RAID 1
        - in pratica è un RAID 0 con maggiore affidabilità, ma più costoso
    - **RAID 3:** block **striping** con **parità a bit** su un **solo disco**
        - stessa efficienza di RAID 2, ma usa un solo disco per la parità
        - velocità pari a n volte quella di RAID 1, grazie al data striping
        - meno operazioni I/O al secondo perché ogni disco è necessario per ogni richiesta e le scritture necessitano di più tempo poiché bisogno calcolare ogni volta il bit di parità, un problema evitabile usando un controllore RAID, che solleva la CPU da questo incarico
    - **RAID 4:** block **striping** con **parità a blocco** su un **solo disco**
        - come RAID 0, ma con un blocco di parità su un disco separato
        - tollera guasti e rende le letture più veloci grazie al parallelismo
        - il disco di parità può essere un collo di bottiglia, rendendo scritture lente a causa del calcolo della parità
    - **RAID 5:** block **striping** con **parità a bit distribuita** nei dischi dati
        - un blocco di parità del disco *i* non può essere contenuto nel disco *i*, altrimenti in caso di guasto del disco non si riuscirebbe a ricostruire i dati
        - come RAID 4 ma senza collo di bottiglia sul disco di parità
        - scritture lente come per il RAID 4
    - **RAID 6:**
        - simile a RAID 5, ma con più informazioni di ridondanza per gestire guasti contemporanei su più dischi; vengono utilizzati altri codici (es. Reed-Solomon) al posto della parità
        - altissima ridondanza ma ad un alto costo, le scritture sono molto lente a causa dei nuovi codici utilizzati
- **Tecniche miste:** è possibile utilizzare **più livelli di RAID** per ottenere versioni sempre più performanti ed affidabili di questa tecnologia, sfruttando pregi di una per eliminari i difetti di un'altra:
    - **RAID 0+1:** prima RAID 1 e poi su quello si applica RAID 0:
        - maggiori prestazioni rispetto a RAID 5, ma richiede il doppio dei dischi dati, quindi risulta essere più costoso
        - non supporta la rottura simultanea di 2 dischi se non appartengono allo stesso stripe
    - **RAID 1+0:** prima RAID 0 e poi su quello si applica RAID 1:
        - più robusto di RAID 0+1, ogni disco di ogni stripe può guastarsi senza far perdere dati al sistema, ma risulta essere costoso (come il RAID 0+1)

Sistema I/O

Il sottosistema di I/O è un **insieme di metodi** per controllare i dispositivi di I/O che ha come obiettivo il fornire un'**interfaccia semplice**, efficiente, e indipendente dai dispositivi stessi ai processi utente.

Hardware di I/O

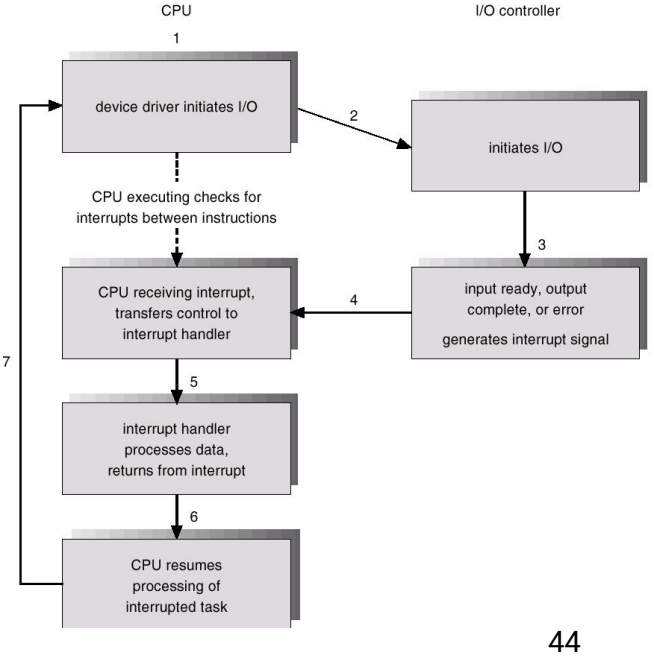
Ci sono tanti ed **eterogenei tipi di dispositivi I/O**, come per memorizzazione (dischi, memorie flash) e trasmissione di dati (modem, schede di rete) di dati, interazioni tra uomo e macchina (monitor, tastiere, etc). I concetti comuni che hanno questi dispositivi sono la **porta** (punto di connessione con la macchina), il **bus** (i collegamenti con il relativo protocollo) e il **controllore** che agisce sulla porta, bus, o altri dispositivi ancora.

È necessaria una distinzione tra:

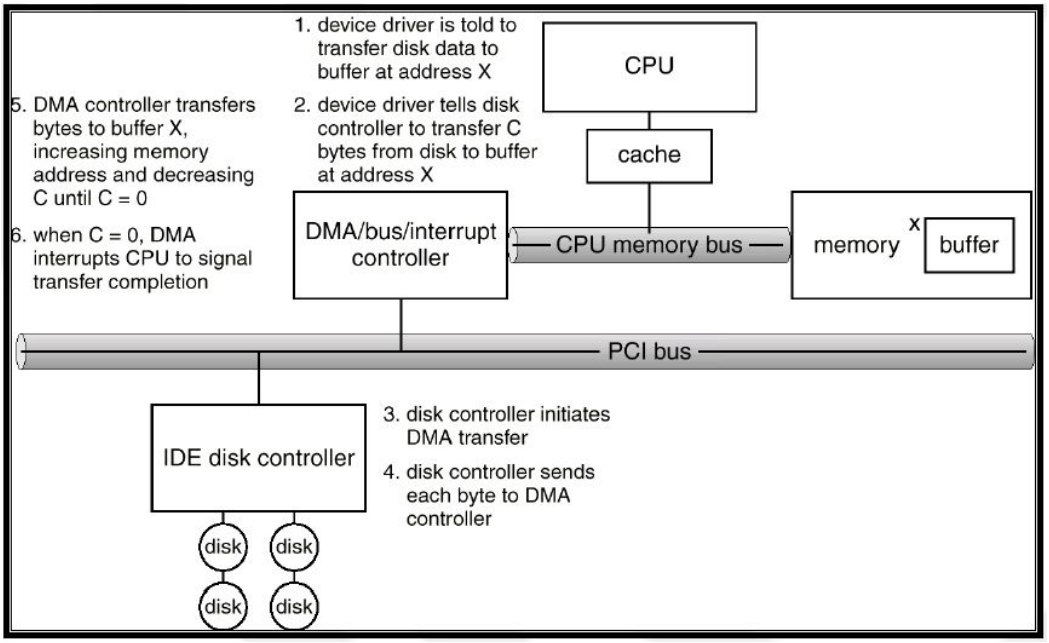
- **Dispositivi** (parte non elettronica)
- **Controllore dei dispositivi** (parte elettronica, detto anche device controller)
    - Connesso tramite bus al resto del sistema.
    - Associato a un indirizzo specifico:
        - Possono essere mappati direttamente in memoria, quindi essere visti nello spazio di indirizzamento del sistema, necessità di disattivare la cache durante l'accesso (memory-mapped).
        - Dispositivi mappati tramite I/O, accesso tramite istruzioni specifiche (I/O-mapped).
    - **Contiene i registri necessari per comandare il dispositivo.**
        - **Uno o più registri di stato:** per capire lo stato del dispositivo, esito dei comandi, etc.
        - **Registro di controllo:** per dare comandi al dispositivo
        - **Uno o più buffer**

Per **accedere ai dispositivi I/O** ci sono diversi metodi:

- **Polling**
    - Determina lo stato del dispositivo mediante la lettura ripetuta del busy bit del registro di stato di un dispositivo.
    - **La CPU cicla regolarmente** sui dispositivi (**Attesa attiva** = spreco di tempo).
- **Interrupt**
    - Il dispositivo **avverte la "CPU"** tramite un segnale su un collegamento fisico.
        - Gli interrupt possono essere mascherabili, cioè ignorabili durante l'esecuzione di istruzioni critiche.
        - Gli interrupt sono numerati, il cui valore è l'indice in una tabella (vettore di interrupt)
        - Interrupt multipli sono ordinati per priorità.
- **DMA (Direct Memory Access)**
    - Pensato per **evitare I/O programmato** per grandi spostamenti di dati.

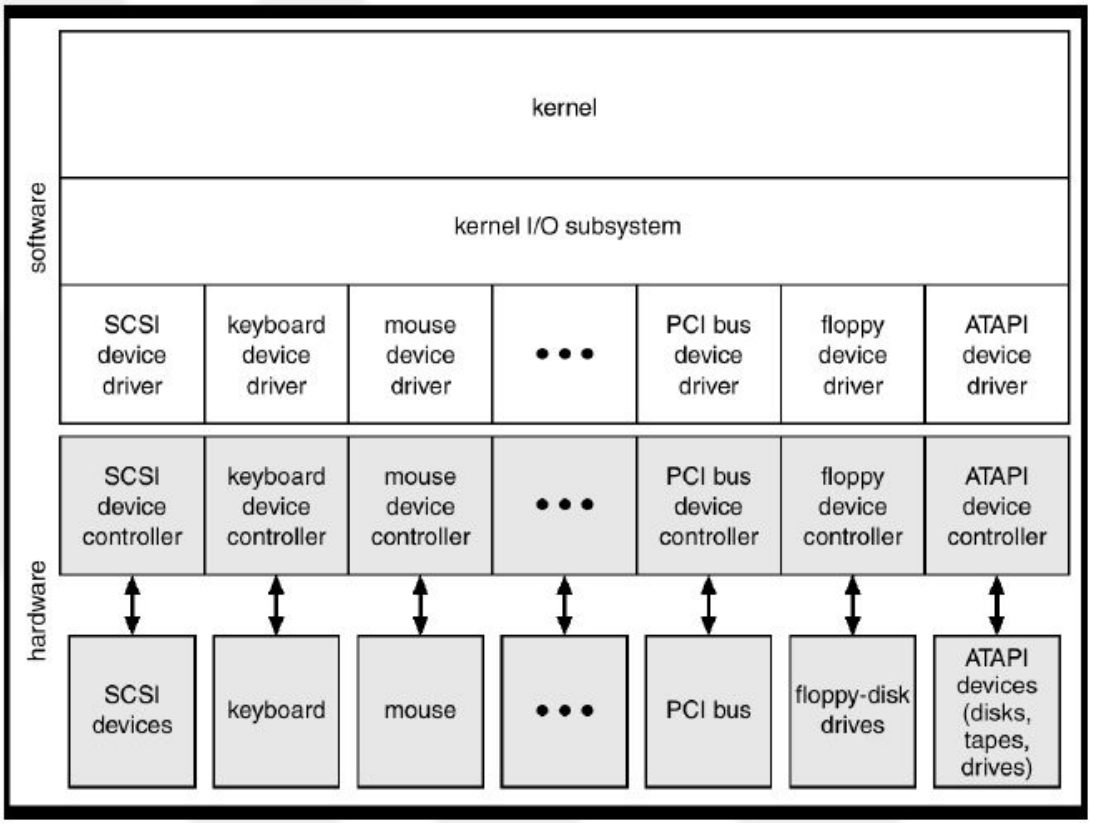


- Richiede **esplicito hardware** (DMA controller).
- Basato sull'idea di **bypassare la CPU per trasferire dati direttamente tra I/O e memoria**.
  - Il DMA controller gestisce il trasferimento, comunicando con il controllore del dispositivo mentre la CPU effettua altre operazioni.
  - Il DMA controller interrompe la CPU al termine del trasferimento.



Interfaccia I/O

Per trattare i dispositivi in maniera standard, viene creata un'**interfaccia con un insieme di funzioni standard**, e le **differenze** vengono implementate e **incapsulate** nei **device driver**.



Tipi di interfaccia dei dispositivi:

- **Interfaccia dispositivi a blocchi:**
  - Memorizzano e trasferiscono dati in blocchi.
  - Lettura/scrittura di un blocco indipendentemente dagli altri.
  - Comandi tipo: read, write, seek.
  - Dispositivi tipici: dischi.
  - Memory-mapped I/O sfrutta block-device driver.
- **Interfaccia dispositivi a caratteri:**
  - Memorizzano/trasferiscono stringhe di caratteri.
  - No indirizzamento (no seek).
  - Comandi tipo: get, put.
  - Dispositivi tipici: terminali, mouse, porte seriali.

Software di I/O

L'obiettivo è fornire un **indipendenza dal dispositivo**, notazioni **uniformi**, **gestione** di **errori** e **opzioni** di trasferimento dati e **prestazioni**.

L'organizzazione è su 4 livelli di astrazione:

Gestori degli interrupt

a. Astratti il più possibile da resto del sistema operativo
b. Bloccaggio/Sbloccaggio processi (semafori, segnali, etc)

Device driver

c. Devono tradurre le richieste astratte del livello superiore in richieste device-dependent.
d. Sono spesso condivisi per diverse classi di dispositivi.
e. Hanno il compito di interagire con i controllori dei dispositivi.
f. Sono tipicamente scritti in linguaggio macchina.

SW del S.O. indipendente dal dispositivo.

g. Definisce interfacce uniformi.
h. Naming dei dispositivi.
i. Protezione: tutte le primitive I/O sono privilegiate.
j. Buffering: gestione di diverse velocità e diverse dimensioni di dati.
k. Definizione della dimensione del blocco.
l. Allocazione e rilascio dei dispositivi.
m. Gestione degli errori

Programmi utente

n. Tipicamente system call per l'accesso ai dispositivi.

Spooling: gestione di hardware I/O dedicato non condivisibile (es. stampante). I dati da processare sono messi in una spooling directory, e periodicamente un processo di sistema detto spooler si occupa di processare i dati nella spooling directory.