

COS'E' UN SISTEMA OPERATIVO?

Un sistema operativo è un insieme di programmi, e offre un ambiente per controllare e coordinare l'utilizzo dell'HW da parte dei programmi applicativi.

- Deve essere facile
- Deve essere efficiente
- Deve evitare conflitti nella allocazione dell'uso di HW e SW

I suoi obiettivi sono quindi **ASTRAZIONE** e **EFFICIENZA**.

Il principio è che si vuole il processore sempre utilizzato!!

1^a GENERAZIONE

Non esisteva un SO, ci si accedeva solo tramite prenotazione. Venivano caricati in memoria un'istruzione alla volta, e si controllavano gli errori dalle spie della console.

(EVOLUZIONE)

- Nascono le periferiche e i sw (librerie), con tempi di setup elevati.

2^a GENERAZIONE

L'operatore e il programmatore vengono separati: l'operatore elimina quindi parte dei tempi morti.

Nasce il **BATCHING**: raggruppamento di programmi simili nell'esecuzione.

Nasce **AutoJobSequencing**: Il sistema si occupa di cambiare i job senza il bisogno dell'operatore.

Nasce il **MONITOR RESIDENTE**: Utilizza la tecnica dell'automatic job sequencing e language per sequenzializzare meglio i job.

Problemi:

- Velocità I/O è minore di quella della CPU.
- La CPU non è mai attiva durante l'I/O.

(EVOLUZIONE)

Con l'elaborazione *offline*, si sovrapponevano le operazioni di I/O e CPU su due macchine differenti. La CPU ha come limite la velocità dei nastri.

Per Sovrapporli sulla stessa macchina?

Ci sono diverse tecniche:

- 1) **POOLING**: la CPU continua a interrogare il dispositivo per sapere se ha trasmesso i dati. (sincrono)
- 2) **DMA**: Il dispositivo avvisa la CPU con un interrupt quando ha finito. In circuiti separati.

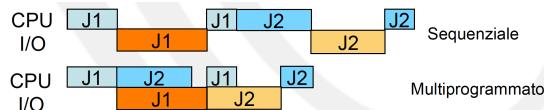
Nel caso di dispositivi molto veloci, senza DMA, arrivano troppi interrupt.

BUFFERING: Permette di sovrapporre CPU e I/O dello stesso job. (l'I/O legge più dati dei richiesti).

SPOOLING: Permette di sovrapporre CPU e I/O di job diversi: i job vengono caricati dentro una memoria, dove uno scheduler gli da in pasto alla CPU secondo un ordine stabilito dal *job scheduler*.

3^a GENERAZIONE

Nasce il concetto di **multiprogrammazione**: in caso di richiesta di un I/O il processo si blocca: la CPU rimane inattiva. Si sfruttano quindi le fasi di attesa I/O per eseguire un altro processo/job.



→ **Sistemi**

non interattività e gli importa **il completamento del job**.

→ **Sistemi Multiprogrammati**: Soddisfano molti utenti , gli importa **il tempo di risposta del job**.

Tradizionali: Tendono alla

TIMESHARING aka MULTITASKING

Lo scheduler ha prelazione, e quindi più utenti lavorano più veloci sulla stessa macchina

PROTEZIONE DEL S.O.

I/O: Due programmi diversi non devono usare I/O contemporaneamente;

MEMORIA: Un programma non può scrivere in una memoria protetta;

CPU: Prima o poi il controllo della CPU va al S.O.;

→ I/O

Protezione realizzata tramite il *dual mode*:

- **USER:** i job non possono accedere alle risorse I/O
- **KERNEL:** Il S.O. può accedere alle ricerche I/O

Quindi, tutte le operazioni di I/O sono eseguite in modalità KERNEL. Le system call offrono un'interfaccia all'utente per poterle eseguire.

→ MEMORIA

Ogni processo ha dei registri limite che possono essere modificati solo dal S.O. con istruzioni privilegiate.

→ CPU

Tramite un timer ogni job deve lasciare la CPU al S.O.

Gestione dei Processi

Il **processo**, è un programma in esecuzione che ha bisogno di risorse e viene **eseguito sequenzialmente**.

Esistono i processi del S.O. e dell'utente.

Il S.O si occupa di:

- Creare i processi
- Sospendere / Riprendere i processi
- Far sincronizzare e comunicare i processi

Gestione Della Memoria Primaria

La memoria **primaria** conserva i dati condivisi tra cpu e i/o; un programma per essere eseguito deve venire prima caricato in questa memoria.

Il S.O si occupa di:

- Gestire lo spazio di memoria: da le parti e dice chi le usa
- Decide quale processo caricare in memoria
- Allocazione e rilascio di memoria

Gestione Della Memoria Secondaria

La memoria **secondaria** è indispensabile per la grande quantità di dati da conservare

Il S.O si occupa di:

- Gestione e allocazione dello spazio su disco
- Scheduling dell'accesso sul disco

Gestione dell'I/O

Il S.O. nasconde all'utente le specifiche caratteristiche dell'I/O.

Il Sistema I/O consiste in:

- Un sistema per accumulare gli accessi i/o (buffering)
- Una generica interfaccia verso il device driver
- Device driver specifici per ogni dispositivo

Protezione

Il S.O. e' responsabile di:

- Definizione degli accessi autorizzati e non
- Definizione dei controlli da imporre
- Fornitura degli strumenti per verificare le politiche di accesso

*Quando avvengono chiamate a funzione, e quindi passaggi di parametro nello stack, si invocano delle syscall e quindi certe operazioni si effettuano in **kernel mode!!***

Struttura dei SO

1) MONOLITICA:

- Non ce gerarchia: tutte le componenti sono allo stesso livello.

2) STRUTTURA SEMPLICE:

- Minima gerarchia: i livelli sono molto flessibili, mirati a ridurre il costo e la manutenzione
- I programmi applicativi possono accedere alle routine di base (mc-dos)
- Ci sono i programmi di sistema e il kernel (unix)

3) STRUTTURA A LIVELLI:

- La struttura e'suddivisa a livelli, dove ogni livello usa le funzioni di quello sotto.
- Svantaggio: difficile definire gli strati, ogni strato deve avere delle syscall. Poca portabilita

4) VM:

- Pensato come sistema di timesharing multiplo. **Hw e sw sono visti come hw.**
- La vm da illusione di processi multipli. Ognuno eseguito sul proprio hw.
- E'possibile usarla su diversi SO.

VANTAGGI:

- Protezione Completa: ogni VM e'indipendente
- Più SO sulla stessa macchina
- Buona portabilità

SVANTAGGI:

- Problemi di prestazioni
- Il gestore delle vm e' in kernel mode, ma la vm in user mode.
- Ogni vm e'isolata dalle altre

5) SISTEMI BASATI SU KERNEL:

- Ci sono solo due livelli: servizi kernel, e non kernel.
- Vantaggio: Ho i livelli, ma non troppi
- Svantaggio: un kernel complesso sembra monolitico

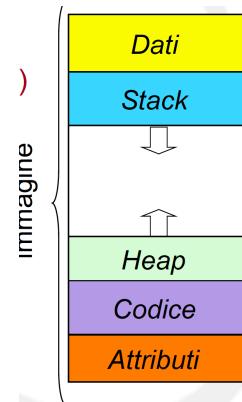
I PROCESSI

Un processo e' eseguito in modo **sequenziale**, ma solo un'istruzione alla volta:
NB In un sistema multiprogrammato i processi evolvono in modo concorrente.

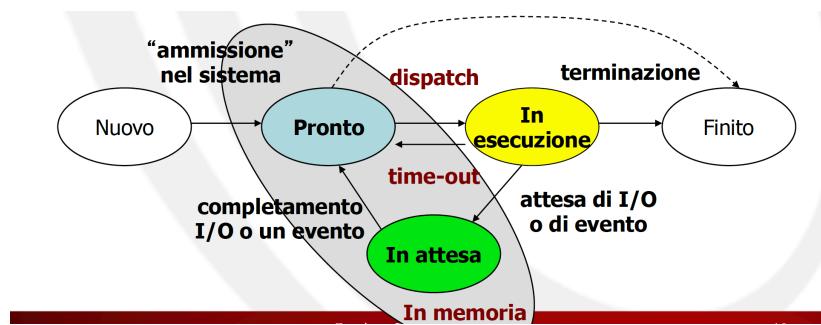
L'**IMMAGINE IN MEMORIA** di un processo e' costituita da:

- Dati
- Stack
- Heap
- Codice
- Attributi (PCB aka Process Control Block)

Il PCB contiene informazioni riguardanti lo stato del processo, i valori dei registri i program counter, info sullo stato i/o ec..



Lo stato dei processi



SCHEDULING

Lo scheduler si occupa di garantire:

- **Multiprogrammazione:** Massimizzare l'uso della CPU, mettendo più di un processo in memoria.
- **TimeSharing:** Comutare frequentemente in processi, facendoli sembrare che ogni processo abbia la CPU tutta per se.

L'operazione di **DISPATCH** sposta il processo da *pronto* a *in esec.* E avviene:

- Cambio di contesto aka **context switch** (in mod. kernel!)
- Passaggio alla **modalità utente**
- Salto all'istruzione da eseguire

Ci sono due relazioni tra processi:

- 1) **PROCESSI INDIPENDENTI:** esecuzioni deterministiche e riproducibili: senza condivisione di dati o altro con altri processi
- 2) **PROCESSI COOPERANTI:** esecuzioni NON deterministiche e NON riproducibili: condividono dati tra di loro e sono influenzati

Questi ultimi, sono utili Perche'si puo'dividere il calcolo, sono modulari e convenienti.

I THREAD

Thread = unità minima dell'uso della CPU; Processo = unità minima dell'uso delle risorse;

Un **processo** ha: Spazio di indirizzamento, e risorse del sistema

Un **thread** ha: Stack, stato di esec., registri della CPU e un program counter: le thread condividono i dati del processo!!!!

VANTAGGI:

- Mentre una thread è bloccata, l'altra può eseguire
- Le thread di un processo, condividono la memoria facilmente
- Il context-switch costa di meno
- Con un multiprocessore si può sfruttare il parallelismo (1 thread x processore)

NB lo stato di un processo non sempre corrisponde a quello della thread.

Implementazione delle Thread

1) USER-LEVEL: Il kernel ignora i thread

- a. Non serve passare in modalità kernel per usare le thread
- b. Lo scheduler è a liv. Applicazione
- c. Girano su qualunque S.O. dato che il kernel non gli vede

SVANTAGGI:

- Se un thread si blocca, blocca il processo
- Non si usa il multiprocessore

2) KERNEL-LEVEL: Le applicazioni usano il kernel tramite le syscall

- a. Se un thread si blocca non blocca il processo
- b. Il SO può essere multithreading

SVANTAGGI:

- Se si vuole cambiare thread bisogna passare dal kernel

Gestione dei Processi del SO

Il SO può essere visto come un processo?

1) KERNEL SEPARATO:

- IL SO ha uno spazio di memoria riservato, prende il controllo del sistema e è sempre in kernelmode
- Il processo è solo processo utente

I servizi del SO sono chiamabili in modalità **protetta** e il codice del SO è condiviso tra i processi

2) KERNEL COME PROCESSO:

- I Servizi del SO sono processi individuali
- Vantaggio quando in multiprocessore possono essere eseguiti su un processore ad hoc

DEADLOCK

Il deadlock o stallo critico avviene se usiamo male i processi.

DEF: Un insieme di processi è in **deadlock** quando ogni processo è in attesa di un evento che può essere causato da un processo dello stesso insieme. (i.e. nell'algoritmo dei 5 filosofi tutti prendono una forchetta, e quindi, sono in deadlock perché nessuno può averne due e si bloccano)

Condizioni Necessarie Affinchè Si Verifichi Un Deadlock:

Se ci sono tutte e queste 4 condizioni, PUO verificarsi il deadlock. Se una di queste è falsa, allora siamo sicuri che non ci sarà mai il deadlock:

- **Mutua Esclusione:** almeno una risorsa deve essere non condivisibile
- **Hold & Wait:** Deve esistere un processo che detiene una risorsa e che attende di acquisirne un'altra, detenuta da un altro
- **No preemption:** Le risorse non possono essere rilasciate se non “volontariamente” dal processo che le usa (il processo che prende le risorse, non le libera finché vuole)
- **Attesa Circolare:** Deve esistere un insieme di processi che attendono ciclicamente il liberarsi di una risorsa.

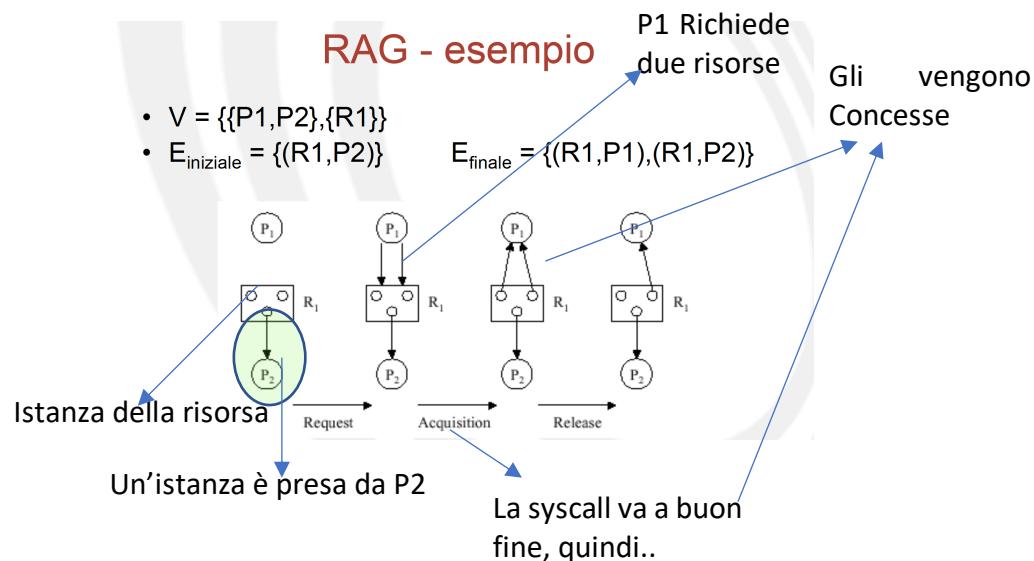
Se queste 4 condizioni sono vere contemporaneamente si può avere il deadlock; se solo una è falsa, allora non si verificherà il deadlock.

RAG (Resource Allocation Graph): Modello Astratto

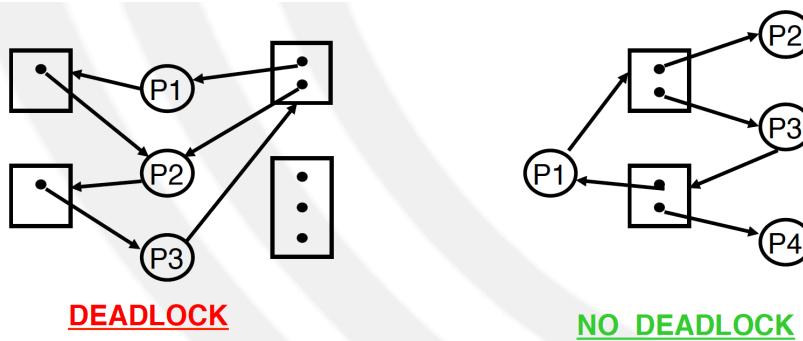
Il RAG è un grafo che può aiutare a rilevare il deadlock, se e solo se, si ha una sola istanza per ogni tipo di risorsa; Nel caso in cui ci sono più istanze dipende dallo schema

G(V, E)

- V = Nodi
 - Cechi = processi (cpu o memoria)
 - Rettangoli = risorse
 - NB: Nei rettangoli ci sono tanti pallini quante sono le istanze della risorsa
- E = Archi
 - Da Processo a Risorsa: il processo la richiede (vuole la risorsa)
 - Da Risorsa e Processo: il processo la detiene (ha quella risorsa)



Ho posso trovare il deadlock in un RAG quando ho un ciclo (non sempre) solo se ho un'istanza per risorsa all'interno del grafo:



In questo caso, il no deadlock, è perché P1 è in H&W, idem P3, ma P2 e P4 sono solo in Hold, e quindi prima o poi P2 o P4 libereranno una risorsa detenuta e in questo modo P3 o P1 potranno prenderla e proseguire.

Gestione dei Deadlock – Alternative

Ci sono due strategie per gestire i deadlock:

- 1) **PREVENZIONE**
 - **Statica**: evitare che si possa verificare una delle 4 condizioni
 - **Dinamica** (basata su allocaz. Delle risorse) : Mai usata perché richiede una conoscenza troppo approfondita delle richieste delle risorse
- 2) **RILEVAZIONE (detection) & RIPRISTINO (recovery)**
 - Permette che si verifichino deadlock
 - Prevede metodi per riportare il sistema al funzionamento normale
- 3) **STRUZZO**: Non fare nulla, gestirli costa troppo e sono rari.

Pro e Contro:

- **PREVENZIONE:** Deve usare meno risorse di quelle che ha, ha quindi meno performance, meno uso di risorse, ma è safe deadlock; è usata sui sistemi che devono controllare un software critico
- **RILEVAZIONE:** Usa le risorse al 100%, ma se avviene un deadlock deve esserci un sistema di rollback e ripristino: se il deadlock è troppo grave non ci sarà un rollback ma il P() ripartirà dall'inizio

Prevenzione STATICÀ:

Ha lo scopo di impedire che si verifichi una delle 4 condizioni che devono essere vere contemporaneamente affinchè si verifichi un deadlock.

CASO – MUTUA ESCLUSIONE

- E' irrinunciabile per certi tipi di risorsa → Non può essere tolta

CASO – HOLD AND WAIT

SOLUZIONI:

- Un processo alloca all'inizio tutte le risorse che deve utilizzare
- Un processo può ottenere un risorsa solo se non ne ha altre
in questo modo spezzo l'attesa circolare

PROBLEMI:

- Basso utilizzo delle risorse
- Possibilità di starvation (richiesta di tante risorse popolari)
- Conoscenza del numero di risorse richieste → Non posso usare l'allocazione dinamica / oggetti

CASO – NO PREEMPTION

SOLUZIONI:

- Un processo che richiede una risorsa non disponibile deve cedere tutte le altre che detiene
- In alternativa: può cedere risorse che detiene su richiesta di un altro processo

PROBLEMI:

- Fattibile solo per risorse il cui stato può essere ristabilito (registri, semafori, cpu ecc..)
- Non per stampanti, nastri ec...

CASO – ATTESA CIRCOLARE

SOLUZIONE:

- Assegno una priorità (ordinamento globale) ad ogni risorsa
- Un processo può richiedere risorse solo in ordine crescente di priorità
- L'attesa circolare diventa impossibile!
- La priorità non è semplice da assegnare e ci pensa il S.O. (a noi non interessa)

NB: Se implemento uno di questi casi con soluzioni non avrò mai il deadlock.

Prevenzione DINAMICA:

La prevenzione statica può portare a un basso utilizzo delle risorse perché mette vincoli sul modo in cui i processi possono accedere alle risorse;

-Obiettivo: Prevenire in base alle richieste (la statica imponeva solo delle regole); viene fatta l'analisi del grafo delle risorse per evitare situazioni cicliche

-Requisito: Conoscere il caso peggiore (bisogna conoscere il massimo numero di istanze di una risorsa richieste per processo)

Prevenzione dinamica – stato safe

Lo stato di una risorsa è calcolato come numero di stanze allocate e disponibili.

Il sistema si trova in uno stato **sicuro / safe** quando:

- Esiste una **sequenza safe**, ovvero: se usando le risorse disponibili, può allocare risorse ad ogni processo, in qualche ordine, in modo che ciascuno di essi possa terminare la sua esecuzione
- (aka trovare un ordine dei processi eseguiti facendo sì che ogni processo usa quelle libere e quelle del processo appena concluso)

DEF: UNA SEQUENZA DI PROCESSI (P_1, \dots, P_n) E' SAFE SE, PER OGNI P_i , LE RISORSE CHE P_i PUO' RICHIEDERE POSSONO ESSERE ESAUDITE USANDO: LE RISORSE DISPONIBILI O QUELLE DETENUTE DA P_j , $j < i$ (attendendo che P_j termini)

SE NON ESISTE TALE SEQUENZA, ALLORA SIAMO IN UNO STATO DI **UNSAFE** (NON VUOL DIRE CHE CI SIA DEADLOCK, MA PUO' VERIFICARSI)

D V/F: La verifica della condizione della sequenza di safe, rimuove il deadlock? **FALSO** : mi permette di prevenirlo, non di rimuoverlo

Quindi, la **Prevenzione DINAMICA**, si basa sul concetto di **sequenza di safe**:

IDEA: Utilizzare algoritmi che lasciano il sistema sempre in uno stato di safe (all'inizio il sistema è in uno stato si safe, e ogni volta che un processo P richiede la risorsa, viene assegnata a p Se e Solo Se si rimane in uno stato di safe)

SAVNTAGGIO: L'utilizzo delle risorse è minore rispetto al caso in cui non uso tecniche di prevenzione dinamica!

Algoritmo RAG (Prevenzione Dinamica - 1 Istanza per tipo di risorsa)

- Funziona solo con una istanza per risorsa
- Il rag viene aumentato con archi di reclamo (\dashrightarrow) che indicano che il processo richiede quella risorsa in futuro
- All'inizio ogni P deve dire quali risorse vorrebbe usare durante la sua esecuzione
- Una richiesta viene soddisfatta SSE l'allocazione della risorsa non crea un ciclo nel RAG
- Serve un algoritmo per la rilevazione di cicli (ha complessità $O(n^2)$)

Algoritmo Del Banchiere (Prevenzione Dinamica - N Istanze per tipo di risorsa)

- Più costoso del RAG
- Meno efficiente del RAG ma funziona con più istanze

Come funziona:

- Ogni processo dichiara la sua massima richiesta
- Ogni volta che richiede una risorsa si determina se soddisfarla ci lascia in uno stato safe
- È costituito da:
 - Algoritmo di Allocazione
 - Algoritmo di verifica dello stato

Rilevazione Deadlock & Ripristino

La prevenzione statica e dinamica riducono l'utilizzo delle risorse. La rilevazione ha due approcci:

- Rilevazione e ripristino tramite RAG
- Algoritmo di rilevazione

R e R tramite RAG (1 istanza per tipo di risorsa)

- Funziona solo con una risorsa per tipo
- Analizza periodicamente i RAG per verificare se esistono deadlock e iniziare il ripristino

Contro: Pago il recovery(molto costoso)

Algoritmo di rilevazione

È Simile al banchiere, ma non gli serve il max numero di risorse.

- Se alla fine dell'algoritmo la sequenza è safe, non c'è deadlock

Ripristino

Quanto spesso bisogna chiamare l'algoritmo di rilevazione?

- Dopo ogni richiesta
- Ogni N secondi
- Quando la cpu è sotto una solia di utilizzo

Come si fa?

- 1) **Uccisione dei processi coinvolti**
 - a. Uccidere tutti i processi (costoso)
 - b. Uccidere in modo selettivo fino alla scomparsa del deadlock. Ma in che ordine? (costoso)
- 2) **Prelazione dei processi**
 - a. Si effettua il rollback in uno stato di safe al processo che subisce la prelazione
 - b. Eventualmente il total rollback
 - c. Ho il problema della starvation (tolgo sempre le risorse ai processi)

CONCLUSIONE:

Ogni approccio ha vantaggi e svantaggi, si usa una soluzione combinata: Si partizionano le risorse in classi e per ognuna utilizzo una strategia diversa, con algoritmi specifici.

MEMORIA CENTRALE

In un ambiente multiprogrammato la condivisione della memoria è essenziale per il sistema. Ogni programma deve essere portato in memoria e trasformato in un processo per essere eseguito (*FDEW*).

La **trasformazione** da programma a processo avviene attraverso varie fasi precedenti all'esecuzione:

Binding & Indirizzi

Il binding di dati e istruzioni può avvenire in tre modi diversi:

STATICO – A TEMPO DI COMPILAZIONE:

Se è nota a priori la parte in cui risiede il processo è possibile generare codice assoluto, se la locazione di partenza cambia bisogna compilare.

Es. Assegno a $x = 100$; e il suo indirizzo è sempre lo stesso: va bene solo se so quali e quanti processi ho (No multiprogrammazione)

STATICO – AL TEMPO DI CARICAMENTO:

Necessario generare codice rilocabile (riposizionabile): decido il binding quando carico il programma. (es. il mio programma sarà sempre al 128byte dall'inizio del programma)

DINAMICO – AL TEMPO DI ESECUZIONE:

Biding posticipato se il processo può essere spostato durante l'esecuzione in posizioni diverse della memoria. (quando il programma entra e esce dalla memoria ha posizioni diverse)

Collegamento (*linking*):

- **Statico:**

- Tradizionale: tutti i riferimenti sono definiti prima dell'esecuzione;
- L'immagine del processo contiene una copia delle librerie usate

- **Dinamico:**

- Link posticipato a tempo di esecuzione

Caricamento (*loading*):

- **Statico:**

- tutto il codice è caricato in memoria al tempo di esecuzione

- **Dinamico:**

- Caricamento dei moduli posticipato al primo utilizzo
- Codice non usato, non viene caricato
- Liv. Multiprogrammazione più elevato

Lo **spazio di indirizzamento** logico è legato a uno spazio di indirizzamento fisico.

-> **Logico**: Generato dalla cpu, chiamato ind. Virtuale

-> **Fisico**: Visto della memoria (realtà)

Nel binding a *compile & load time* ind. Fisico == ind. Logico

Nel binding a *runtime* ind. Fisico != ind. Logico

MMU

L'MMU è un dispositivo hardware che mappa gli indirizzi virtuali in fisici. Somma un ind. Base a quello logico.

CONCLUSIONE:

In un sistema **multiprogrammato**, non è possibile conoscere in anticipo l'ind. Del processo in memoria, il *binding a tempo di compilazione* non è possibile.

Lo **swap** impedisce gli indirizzi rilocati in modo statico, il *binding a tempo di caricamento* non è possibile.

→ Bisogna usare la rilocazione dinamica (usata per sistemi complessi e per la gestione della memoria nel SO)

GESTIONE DELLA MEMORIA

Esistono 4 schemi di gestione della memoria:

- Allocazione contigua
- Paginazione
- Segmentazione
- Segmentazione Paginata

(prevedono che l'intero programma sia caricato in memoria)

1) ALLOCAZIONE CONTIGUA:

I processi allocati in memoria in posizione contigue all'interno di una partizione. Le partizioni sono **fisse e variabili**.

La memoria è un insieme di partizioni di dimensioni predefinite, che non possono essere cambiate. Se ho (7 partizioni il mio liv. max di multiprogrammazione sarà 6, una è per il SO)

ASSEGNAZIONE DELLA MEMORIA:

è effettuata dallo scheduler a **lungo termine**, usando due sistemi:

- Una coda per partizione: il processo viene assegnato alla partizione più piccola in grado di contenerlo; poco flessibile
- Una coda singola per tutte le partizioni: va gestita con una politica (FCFS, best fit only, best av. Fit)

SUPPORTO PER LA RILOCAZIONE:

La MMU consiste di registri di rilocazione per proteggere lo spazio dei vari processi. Contiene due registri, uno di base o rilocazione, e uno di registri limite (lim superiore)

VANTAGGI	SVANTAGGI
Costa poco ed è semplice	Grado di multiprogrammazione limitato al numero di partizioni. Ho frammentazione (spreco di memoria) interna , partizione > job esterna , ho tanti buchi che non sono contigui per inserirci il mio processo

1-a) ALLOCAZIONE CONTIGUA con partizioni variabili:

Lo spazio utente è diviso in partizioni di dimensioni variabili, identiche alla dimensione dei processi. In questo modo posso eliminare **frammentazione interna**. (ma non quella esterna)

ASSEGNAZIONE DELLA MEMORIA:

La memoria è vista come delle buche *hole*, buca = memoria libera, il SO ha informazioni su:

- Partizioni allocate
- Hole

Quando arriva un processo gli viene allocata la memoria usando la buca che lo può contenere. Quando un processo esce dalla memoria lascia un "buco" che mi genera **frammentazione esterna**.

Quale buco scegliere?

First-Fit: alloca la prima buca grande a sufficienza

Best-Fit: alloca la più piccola buca grande a sufficienza (richiede scansione, ma minimo spreco)

Worst-Fit: alloca la buca più grande (richiede scansione, lascia la buca di dimensioni più grandi)
(first fit sembra la migliore)

SUPPORTO PER LA RILOCAZIONE:

LA MMU è uguale come per le partizioni fisse. (2 registri, uno limite)

VANTAGGI	SVANTAGGI
Non ho frammentazione interna per la costruzione	Ho frammentazione esterna. Con first-fit, dati n blocchi allocati, ho $0,5^*N$ blocchi persi

Per ridurre la frammentazione esterna si può compattare il contenuto della memoria in modo da rendere contigui i blocchi di un processo

- È possibile solo se la rilocazione è dinamica
- E molto costosa

1-b) TECNICA DEL BUDDY SYSTEM

è un compromesso tra le partizioni fisse e variabili. La memoria è vista e disponibile come un insieme di blocchi di **dimensione 2^k** con $L < K < U$

- 2^L = blocco più piccolo allocato
- 2^U = blocco più grande allocato (es. tutta la memoria)

All'inizio tutta la memoria è disponibile, la lista del blocco 2^U contiene un solo blocco che rappresenta tutta la memoria, le altre sono vuote.

- Quando arriva una richiesta, si cerca un blocco libero con dimensione "adatta" purchè sia pari a una potenza di 2
- Quando un processo rilascia la memoria il suo blocco tona a fare parte della lista dei blocchi corrispondente

VANTAGGI	SVANTAGGI
La compattazione richiede solo di scorrere la lista 2^k quindi è veloce	Frammentazione interna solo ai blocchi di dimensione 2^L

2) PAGINAZIONE

La paginazione è una tecnica che garantisce l'uso della memoria efficace e una elevata multiprogrammazione.

La paginazione è una tecnica per eliminare la **frammentazione esterna**:

- Lo spazio logico è contiguo, quello fisico non contiguo
- Alloca memoria fisica dove è possibile

La memoria **fisica** è divisa in blocchi, chiamati **frame**.

La memoria **logica** (vista dalla CPU) è divisa in blocchi della stessa dimensione, chiamati **pagine**.

Per eseguire un programma, avente dimensione N pagine, bisogna trovare N frame liberi: si utilizza una **tabella delle pagine** per mantenere traccia di quale frame corrisponde a quale pagina

- Ho una tabella delle pagine per ogni processo
- Viene usata per tradurre ind. logico in ind. fisico

TRADUZIONE DEGLI INDIRIZZI:

L'indirizzo generato dalla CPU viene diviso in due parti:

- Numero di pagina (P): *usato come indice nella tabella delle pagine che contiene l'indirizzo di base di ogni frame*
- Offset (D): *combinato con l'indirizzo di base definisce l'indirizzo fisico che viene inviato alla memoria*

L'implementazione della tabella delle pagine può avvenire tramite registri e tramite memoria:

IMPLEMENTAZIONE TRAMITE REGISTRI:

Le entry (righe) della tabella delle pagine sono mantenute nei registri:

- Soluz. Efficiente ma solo se le entry sono poche
- Allunga il tempo di context switch (bisogna salvare lo stato dei registri)

IMPLEMENTAZIONE TRAMITE LA MEMORIA:

La tabella risiede in memoria, e vengono utilizzati due registri:

- Page-table base register (PTBR): *punta alla tabella delle pagine*
- Page-table lenght register (PTLR): *contiene la dimensione della tabella delle pagine* (opzionale)
- Il context switch è più breve.

Problema:

Quando la CPU genera un indirizzo logico, deve fare un accesso in memoria per leggere quello fisico, e quando l'ha trovato deve andare a prendere il dato (fa 2 accessi)
(accesso alla tabella delle pagine + dato/istruzione)

Per risolvere il problema del doppio accesso viene introdotto un supporto HW, che è una cache veloce, chiamata **translation look-aside buffers** (TLB)

- È molto costosa e più piccola della RAM
- Confronta l'elemento fornito con il campo chiave di tutte le entry

Nel TLB viene memorizzato un solo piccolo sottoinsieme di entry delle pagine. Ad ogni context switch il TLB viene ripulito per evitare il mapping di indirizzi errati.

Durante un accesso alla memoria:

Se la pagina è cercata nel TLB, esso restituisce il numero del frame con un singolo accesso, altrimenti è necessario accedere alla tabella delle pagine in memoria. La hit ratio *alpha* è la % delle volte in cui una pagina si trova nel TLB.

TEMPO DI ACCESSO EFFETTIVO:

$$EAT = \alpha * (T_{mem} + T_{tlb}) + (1 - \alpha) * (T_{mem} * (n-1) + T_{tlb})$$

PROTEZIONE della tabella delle pagine:

si aggiunge un bit ad ogni frame: bit di validità per ogni entry della tabella delle pagine (utile per la memoria virtuale)

→ 0 (valid) = la pagina è nello spazio di indirizzamento logico del processo

→ 1 (invalid) = la pagina associata non è nello spazio (il SO può sostituire il processo)

PAGINE CONDIVISE:

Posso condividere il codice di un processo: c'è un'unica copia fisica, ma più copie logiche (una per processo)

Quanto occupa la tabella delle pagine? Tanto. Si usano due meccanismi per gestire la dimensione della tabella delle pagine:

- Pagine multilivello
- Pagine invertite

PAGINE MULTILIVELLO

- Equivalente a impaginare la tabella delle pagine
- Posso avere più livelli delle pagine
- L'indirizzo logico è suddiviso in più offset dei vari livelli

PAGINE INVERTITE

- Ho un'unica tabella per tutti i processi, ho una entry per ogni frame
 - Indirizzo virtuale; informazioni sul processo che usa quella pagina
- Problema: più di un indirizzo virtuale può corrispondere a un frame
- Conseguenza: è necessario cercare il valore desiderato e aumenta il tempo per la ricerca di un riferimento nella pagina

2)SEGMENTAZIONE

Il programma è una collezione di segmenti, il quale segmento è un'unità logica main, procedure, funzioni, variabili ecc..

I **segmenti** non hanno tutti la stessa dimensione, sono indicati da nome e lunghezza;

- L'indirizzo logico è <numero segmento, offset>
- La tabella dei segmenti mappa i segmenti e non le pagine. Mappa indirizzi logici bidimensionali in indirizzi fisici unidimensionali. Ogni entry ha:
 - Base (indirizzo fido di partenza del segmento di memoria)
 - Offset (lunghezza del segmento)

TABELLA DEI SEGMENTI:

Simile alla tabella delle pagine:

in memoria ho STBR (base) e STLR (numero segmenti usati da un programma)

Il TLB è usato per memorizzare le entry maggiormente usate.

PROTEZIONE:

La segmentazione consente la protezione e condivisione di porzioni di codice. Ogni segmento ha r/w/e e il bit di validità.

Per la condivisione è accessibile a livello di segmento (es. si può condividere una libreria)

MEMORIA VIRTUALE

Con i precedenti schemi il programma doveva essere interamente caricato in memoria → questo non è necessario.

- Lo spazio degli indirizzi logici può essere molto più grande dello spazio degli indirizzi fisici
- Più processi possono essere mantenuti in memoria
- Possibilità di *swappare* pagine da e verso la memoria e non l'intero processo
- La memoria virtuale permette la separazione della memoria logica dalla memoria fisica

Memoria virtuale = memoria fisica + disco

Si può implementare con paginazione su domanda e segmentazione su domanda.

PAGINAZIONE SU DOMANDA

Una pagina viene caricato solo quando necessario, uso quindi meno I/O e meno memoria. Ma mi serve sapere lo stato della pagina (è in memoria?). Ad ogni entry della page table è associato un bit di validità (1 = memoria, 0 = non in memoria).

- All'inizio sono tutti 0
- Se durante la traduzione una entry ha il bit a 0, si ha un *page fault*.

GESTIONE DEI PAGE FAULT

Il page fault causa un *interrupt* al SO.

- 1) Il SO verifica una tabella associata al processo
 - a. Riferimento valido = attiva il caricamento della pagina
- 2) Cerca un frame vuoto
- 3) Swap della pagina nel frame (da disco)
- 4) Modifica le tabelle
 - a. Valid bit a 1
- 5) Ripristina l'istruzione che ha causato il page fault

NB: Il primo accesso in memoria causa sempre un page fault

RIMPIAZZAMENTO DELLE PAGINE

Cosa succede se non ci sono pagine libere? Quando cerco il frame libero?

Rimpiazzamento delle pagine:

- Richiede un opportuno algoritmo, che ottimizza e minimizza il # di page fault
- 1) Il SO verifica una tabella (associata al processo) per capire se si tratta di page fault o violazione di accesso
 - 2) Cerca un frame vuoto
 - a. Se esiste salta a 4
 - b. Se non ce usa un algoritmo per scegliere un frame vittima
 - 3) Swap della vittima su disco
 - 4) Swap della pagina del frame su disco
 - 5) Modifica delle tabelle
 - 6) Ripristina l'istruzione che ha causato page fault

In assenza di frame liberi sono necessari due accessi alla memoria: swap vittima e swap frame da caricare.

Page fault raddoppiato.

Per ottimizzare uso un bit nella page table:

- 1 = pagina modificata dal momento in cui è stata caricata
- solo le pagine che risultano modificate (1) vengono scritte sul disco quando diventano "vittime"

ALGORITMO DI RIMPIAZZAMENTO DELLE PAGINE:

L'obiettivo è il minimo tasso di page fault. Come vengono valutati?

- Esecuzione di una reference string
- Calcolo dei # p.f.
- Necessario sapere il di frame disponibili per il processo

ALG || FIFO

- La prima pagina introdotta è la prima ad essere rimossa
- Non viene valutata l'importanza della pagina rimossa
- Tende ad aumentare il tasso di page fault
- Soffre dell'anomia di belady

ALG || IDEALE

- Garantisce il numero minimo di page fault
- Idea: rimpiazza le pagine che non saranno usate per il periodo più lungo
- Bisogna sapere la reference string. È possibile approssimare ma l'implementazione è difficile
- Utile come riferimento agli altri algoritmi

ALG || LRU

Least Recently Used

- Usa il passato recente per prevedere il futuro
- Si rimpiazza la pagina che non viene usata da più tempo
- Migliore di FIFO
- Non banale ricavare il tempo dell'ultimo utilizzo
- Può richiedere hw addizionale
- O Viene creato tramite un contatore
 - Ha un contatore associato ad ogni pagina
 - Ogni volta che la pagina viene referenziata, il clock del sistema viene copiato nel contatore
 - Rimpiazza la pagina con il valore più piccolo del contatore
- O Viene creato tramite stack

ALG || LRU – Approssimato

Uso un bit reference

- Associato ad ogni pagina, inizialmente 0
- Quando la pagina è referenziata viene messo a 1 dall'HW

ALLOCAZIONE DEI FRAME

Data una memoria con N frame e M processi, è importante scegliere quanti frame allocare per processo.

- Ogni processo necessita di un numero di pagine per poter essere eseguito
- #minimo di pagine = massimo numero di indirizzi specificabile in una istruzione

L'allocazione può essere **fissa** (sempre stesso numero di frame) oppure **variabile** (il numero di frame può cambiare runtime)

Allocazione variabile:

Permette di modificare dinamicamente le allocazioni ai vari processi: in base a cosa cambio lo spazio?

- Calcolo il working set
- Oppure calcolo il page fault frequency

CALCOLO DEL WORKING SET:

Un criterio per rimodulare l'allocazione dei frame consiste nel calcolare in qualche modo quali sono le richieste effettive di ogni processo

- ➔ In base alla località degli indirizzi durante la sua esecuzione

Idealmente un processo necessita un numero di frame pari alla sua località. Ma come si misura?

MEMORIA SECONDARIA

Scheduling del Disco

Una responsabilità del Sistema Operativo è quella di fare un uso efficiente dell'hardware. Nel caso dei dischi magnetici il tempo d'accesso è dato da due principali componenti:

- **Tempo di Ricerca (Seek Time)**: è il tempo necessario affinchè il braccio dell'unità del disco sposti le testine fino al cilindro contenente il settore desiderato
- **Latenza di Rotazione** : è il tempo aggiuntivo perché il disco ruoti finchè il settore desiderato si trovi sotto la testina

Gestendo l'ordine delle richieste di operazioni di I/O verso un'unità disco si può migliorare il tempo di accesso.

Ogni volta che un processo P deve compiere delle operazioni di I/O con un'unità disco, un processo effettua una chiamata a sistema che contiene diverse informazioni.

Scheduling con FCFS: La forma più semplice di scheduling è FCFS.

- Non garantisce la massima velocità
- Non ha costo per decisione

Scheduling con SSTF: Seleziona la richiesta con il minimo spostamento rispetto alla posizione attuale della testina.

- Può condurre a *starvation*

Scheduling con SCAN: Il braccio del disco parte da un estremo e va fino all'altro eseguendo tutte le richieste che trova per strada. A volte è chiamato *algoritmo dell'ascensore*.

Assumendo una distribuzione uniforme delle richieste sui dischi, quando la testina arriva ad un estremo è molto probabile che la maggior quantità di richieste si situì sulla zona opposta.

- Può condurre a *starvation*

Scheduling con C-SCAN: aka SCAN Circolare, è una variante dello SCAN, concepita per garantire un tempo di attesa meno variabile: anziché fare avanti e indietro come lo SCAN, il C-SCAN una volta che arriva a fine estremità torna subito all'altra senza servire le richieste. Il disco è quindi visto come una lista circolare. i.e.: SCAN fa da A a B=>da B a A. Il C-SCAN Farà da A a B=>Torna ad A senza eseguire richieste => Va da A a B.

- Può condurre a *starvation* (< rispetto a SCAN)

Scheduling con LOOK (C-LOOK): L'algoritmo dello SCAN sposta la testina per tutta l'ampiezza del disco, il LOOK si sposta solo se ci sono altre richieste da servire in quella direzione.

Scheduling con N-STEP-SCAN: La coda delle richieste viene partizionata in N code: quando una coda viene processata per il servizio gli accessi riempiono altre code. (Vengono riempite diverse code, e poi viene eseguito l'algoritmo dello SCAN sulle singole code)

- Rimuove il problema della starvation

Scheduling con LIFO: Schedula in base all'ordine inverso di arrivo.

- Può condurre a *starvation*
- Velocizza i processi con accessi vicini

Gestione del Disco – Formattazione

La formattazione di **basso livello** o **formattazione fisica** riempie il disco con una speciale struttura dati per ogni settore, tipicamente consiste di

- Suddividere il disco in settori
- Identificare i settori
- Aggiungere uno spazio per la correzione di errori (*ECC*)

Il Sistema Operativo per poter utilizzare il disco deve prima **partizionarlo** e poi eseguire una **formattazione logica**, cioè la creazione di un *file system*: il sistema operativo registra le struttura iniziali relative al file system (i.e. Spazio libero, spazio allocato, e directory).

Gestione del Disco – Blocchi difettosi

Se si ha un **bad block**, ossia un blocco danneggiato che non si riesce più a gestire, può essere gestito in due modalità:

Gestione Off-Line: Durante una formattazione logica, ove il S.O. individua i *bad block* e gli marca come danneggiati inserendoli in una lista, dopodichè si può usare una utility per “isolarli” (i.e. CHDISK).

Gestione On-Line: Il controllore del disco (e non il S.O.) mappa i *bad block* su un blocco buono (azione trasparente per il S.O.) per poterlo fare è necessario che il disco abbia dei blocchi di scorta opportunamente riservati (chiamati *sector sparing*). Questo sistema ha come problema che le ottimizzazioni fornite dagli scheduler siano inutili (soluz: vengono allocati *sector sparing* su ogni cilindro)

FILE SYSTEM

Il file system consiste in due parti distinte: un insieme di *file*, ciascuno dei quali contiene dei dati, e una *struttura delle directory*, che organizza tutti i file nel sistema e fornisce le informazioni relative.

File

Un **file** è un insieme di informazioni correlate e identificate da un nome, salvate in uno spazio di indirizzamento logico e continuo. Esiste il file *dati* (Numerici, Binari, ...) e i file *programmi* (Istruzioni).

Cosa viene memorizzato di un file?

Gli **attributi di un file** vengono memorizzati su un disco nella struttura della directory (nella cartella).

- N.B.: Il **file** ha solo il contenuto associato al nome (risiedente nella directory).
- Gli **attributi** non sono dentro al file ma nella struttura che gestisce le directory

Gli **attributi salvati** sono quindi SEI [6]:

- Nome
- Tipo
- Posizione
- Dimensione
- Protezione
- Tempo, Data e identificazione dell'utente

TIPS:

- Se modifco il contenuto del file, modifco solo gli attributi di dimensione e il tempo
- Se cambio il nome del file, impatta sulla cartella, perché è salvato lì il suo nome

Operazioni sui file

Un file, è un tipo di dato *astratto* e il S.O. può offrire chiamate di sistema per *creare, scrivere, leggere, spostare, cancellare e troncare* un file.

Creazione:

- Bisogna cercare lo spazio nel file system
- Bisogna creare un nuovo elemento nella directory per gli attributi

Scrittura:

- Avviene mediante una *syscall* specificando nome file + contenuto
- Necessario un puntatore di scrittura da aggiornare ogni volta (da sapere dove scrivere la prox.)

Lettura:

- Avviene mediante una *syscall* specificando nome file + posizione di memoria
- Necessario un puntatore di lettura da aggiornare ogni volta

TIPS: La posizione è contenuta in un solo puntatore sia per la lettura che per la scrittura (un processo o legge o scrive un file) risparmiando così spazio.

Spostamento:

- Aggiorna il puntatore del file (non richiede operazioni di I/O)

Cancellazione:

- Viene liberato lo spazio associato al file e l'elemento nella directory

Per semplificare il problema della ricerca in ogni operazione sul file, con una primitiva *open* il file entra in una tabella dei file in cui viene salvato il puntatore finché non viene *chiuso (close)*. In un ambiente multiutente ci sono due livelli di tabelle interne:

- Una **tabella per ciascun processo** che contiene i riferimenti ai file aperti da quel processo (i.e. puntatore alla posizione corrente, info sui diritti di accesso).
- Ciascun elemento della tabella associata a ciascun processo punta a sua volta a una **tabella di sistema** dei file aperti, contenente informazioni indipendenti dai processi come la posizione della mem. secondaria, dimensione ecc..

Metodi di accesso al file

Esistono molti metodi per accedere ai file, alcuni S.O. ne offrono solo uno, altri di più:

Accesso Sequenziale:

Le informazioni dei file si elaborano record per record. È il metodo più comune.

- Usato dagli editor e compilatori
- *Permessi: ReadN, WriteN, Rewind. Non permessi: Rewrite!*

Accesso Diretto:

i file ad accesso diretto sono molto utili quando è necessario accedere immediatamente a grandi quantità di informazioni (es. *Database*)

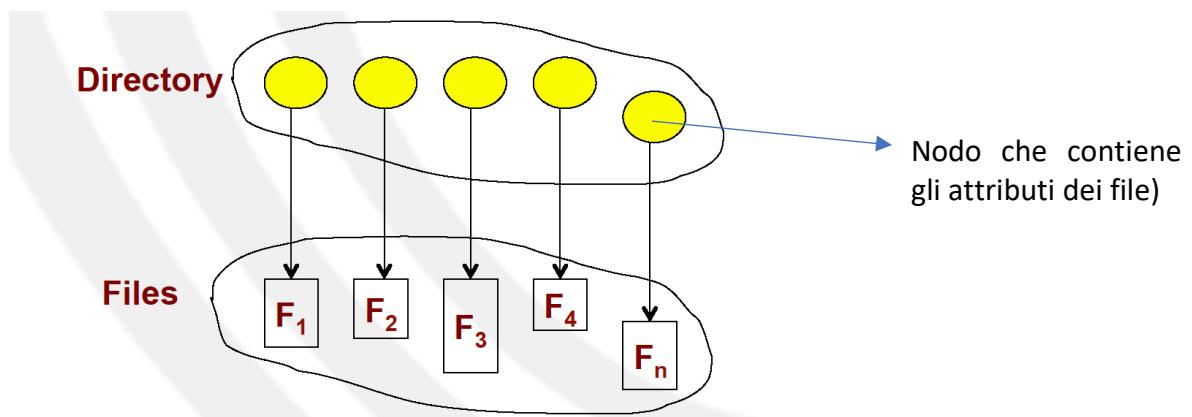
- *Permessi: Read(n), write(n), position (n), [...], rewrite (n)*

Struttura delle directory

Un disco, può essere *partizionato* e contenere un file system diverso per ogni partizione. Ogni volume contenente un file system deve anche avere le informazioni sui file presenti del sistema: esse risiedono in una **directory del dispositivo** (abbreviato *directory*).

La **directory** è una collezione di nodi ed elementi che contengono le *informazioni / attributi* sui file.

- La directory determina la struttura del file system



- In una **directory** sono presenti, per ogni file, le seguenti informazioni: *Nome, tipo, indirizzo, lungh. Attuale, massima lunghezza, data ultimo accesso, data ultima modifica, possessore, info protezione.*

Operazioni su directory

Aggiunta di un file, cancellazione di un file, visualizzare il contenuto della directory, rinominare un file, cercare un file, attraversare il file system: sono tutte operazioni che vengono fatte su una directory.

Gli **obiettivi** sono l'*efficienza* (ossia il rapido accesso a un file), *nomenclatura* e *raggruppamento*.

Struttura (liv.) delle directory

|| Directory a un livello

- Non esistono sottocartelle (*singola directory per tutti gli utenti*)
- Problema di nomenclatura
- Problema di raggruppamento

|| Directory a due livelli

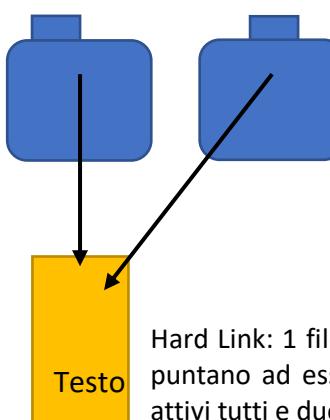
- Nasce il concetto di directory separata per ogni utente
- Nasce il contetto di *path*
- Ricerca efficiente tramite la *path*
- Si crea una variabile d'ambiente 'path' per contenere i programmi condivisi agli utenti

|| Directory ad albero

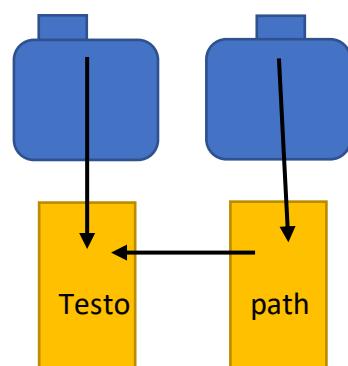
- Non vincolano più all'utente
- Ricerca efficiente
- Nasce il concetto di directory corrente (*pwd*)
- Nomi di percorso assoluti / relativi

|| Directory a grafo aciclico

- Permette la condivisione di file e directory (link simbolici e hard link)



Hard Link: 1 file ha 2 nodi che puntano ad esso. Finchè sono attivi tutti e due i link il file non viene sovrascritto



Link Simbolico: se elimino il primo link perdo il secondo collegamento. Il collegamento punta a un file contenente il path del file linkato.

|| Directory a grafo generico

- Problema dei cicli (gli algoritmi di ricerca possono non funzionare)
- Troppo costoso per impedire i cicli

Mount del file system

Il file system deve essere *montato* e ciò permette di montare più file system sulla stessa macchina.

- Possibilità di attaccare e staccare file system interi

Protezione

Il possessore di un file deve poter controllare chi fa e cosa fa. Le operazioni controllabili sono:

- Lettura, scrittura, esecuzione, append (aggiunta in coda), cancellazione [...]

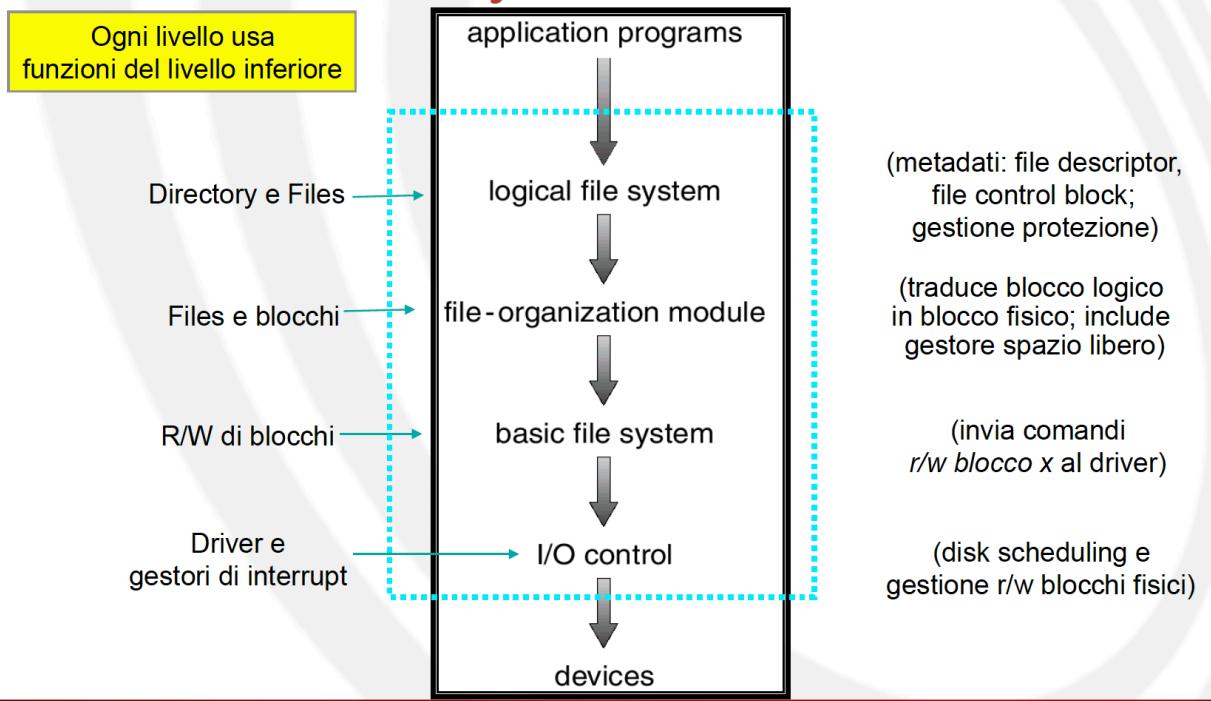
Si può attuare la **protezione** in **due modi**:

- 1) Lista d'accesso per file / directory (elenco di chi può fare cosa per ogni file / directory; problema: lungo)
- 2) Utenti raggruppati in 3 classi (Proprietario, Gruppo, Altri; ad ogni classe si gestiscono i permessi rwx)

REALIZZAZIONE FILE SYSTEM

Il file system ha anche il problema di creare gli algoritmi e strutture dati che permettano di far corrispondere il file system logico ai dispositivi fisici di memoria secondaria. Lo stesso file system è suddiviso da molti livelli, dove ogni livello si serve delle funzioni dei livelli inferiori per crearne nuove per i livelli superiori.

File System a livelli



Per gestire il file system si usano **diverse strutture dati**:

- Alcune risiedono su **disco**
- Alcune risiedono in **memoria** (vantaggio: efficienza)

Quali stanno dove, è deciso dal sistema operativo e dal tipo di file system.

Le strutture presenti **nel disco** sono le seguenti:

- **Blocco di boot**: ogni volume ha un blocco di boot contenente le informazioni necessarie per l'avviamento di un S.O. da quel volume.
- **Blocco di controllo delle partizioni (o volume)**: ogni partizione contiene un blocco che contiene i dettagli riguardanti la partizione (numero e dimensione dei blocchi, lista blocchi neri, lista blocchi liberi ecc..)
- **Strutture di directory**: Usata per organizzare i file
- **Descrittori di file**: contengono i dettagli sui file e i puntatori ai blocchi di dati

Le strutture **in memoria** sono le seguenti:

- **Tabella delle partizioni**: contiene informazioni sulle partizioni montante (quelle visibili)
- **Strutture di directory**: Copia in memoria delle directory a cui si è fatto accesso di recente (caching)
- **Tabella globale dei file aperti**: contiene una copia dei descrittori dei file
- **Tabella dei file aperti per processo**: contiene un puntatore alla tabella dei file aperti insieme ad altre informazioni di accesso

Allocazione su disco

Per minimizzare i tempi di accesso e massimizzare l'utilizzo dello spazio, esistono tre modi per l'allocazione del file system:

- **Allocazione contigua**
- **Allocazione a lista concatenata (linked)**
- **Allocazione indicizzata**

ALLOCAZIONE CONTIGUA

Ogni file occupa un insieme di blocchi contigui su disco

- ➔ Lo spazio che occupa la directory è poco (es. F1-9-5 : file1-9blocchi-a partire dal blocco 5)

Vantaggi	Svantaggi
<ul style="list-style-type: none"> -Accesso semplice: l'accesso al blocco $b+1$ non richiede lo spostamento della testina -Accesso sequenziale e causale: per accedere al blocco successivo basta fare $b+1$ 	<ul style="list-style-type: none"> -Spreco di spazio (frammentazione esterna) -Algoritmi best/first/worst -fit

NB: Se un file deve crescere e non c'è spazio viene terminato il programma oppure bisogna cercare un buco maggiore da inserire il file nuovo.

ALLOCAZIONE A LISTA

Il file può essere frammentato in diversi pezzi: ogni file è una lista di blocchi, ed essi possono essere sparsi nel disco.

- ➔ La directory contiene i puntatori al primo e all'ultimo blocco
- ➔ Ogni blocco contiene il puntatore al blocco successivo ($b+1$)

Vantaggi	Svantaggi
<ul style="list-style-type: none"> -Facile da creare i file -Facile da estendere un file -Non ho sprechi (eccetto lo spazio aggiuntivo del puntatore) aka posso usare qualsiasi blocco senza causare frammentazione esterna 	<ul style="list-style-type: none"> -Non è possibile fare accesso causale (non conosco nella lista la posizione i-esima. Essa è data solo dal blocco nella posizione $i-1$) -Richiede tanti rispostamenti sparsi alla testina(poco efficiente) -Scarsa affidabilità / protezione: se perdo un puntatore perdo tutta la lista. (Si può risolvere con una lista doppiamente puntata ma crea overhead)

ALLOCAZIONE CONTIGUA – VARIANTE

Alcuni SO moderni usano uno schema modificato di allocazione contigua, basato sul **concepto di extent** (serie di blocchi contigui su disco), cioè sono un insieme di blocchi contigui -> Creo una lista di extent.

- Il file system alloca gli extent anziche i blocchi
- File = serie di extent
- I vari extent non sono contigui

ALLOCAZIONE INDICIZZATA (assomiglia alla paginazione)

Ogni file ha un blocco indice contenente la tabella degli indirizzi dei blocchi fisici. La directory contiene l'indirizzo del blocco indice.

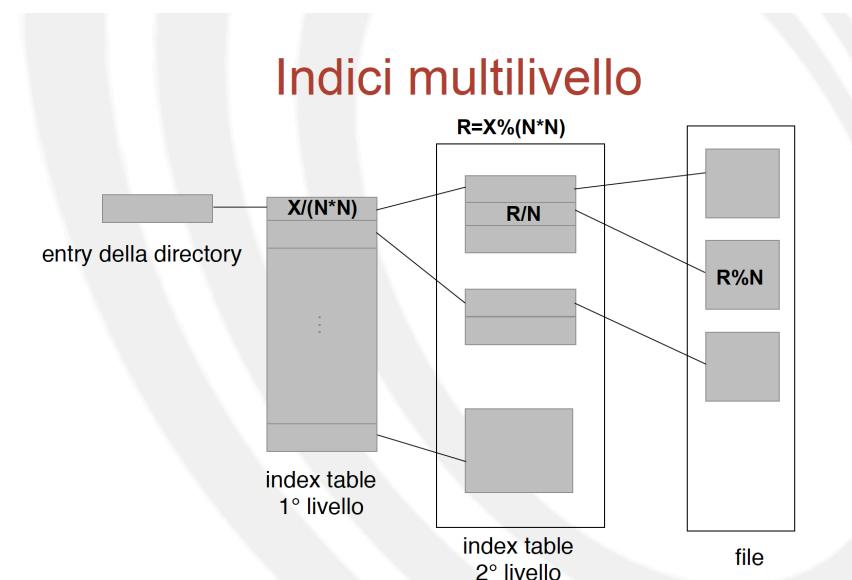
Vantaggi	Svantaggi
<ul style="list-style-type: none"> - Accesso casuale efficiente - Accesso dinamico senza frammentazione esterna (spreco un blocco per tenere gli indici dei blocchi) 	<ul style="list-style-type: none"> -la dimensione del blocco limita la dimensione del file.

Per ovviare al problema dei file con dimensioni senza limiti si usano degli schemi a più livelli:

- **Indici multilivello**
- **Schema concatenato**
- **Schema combinato**

1-sol) Indici Multilivello:

Una tabella esterna contiene i puntatori alle *index-table*. Gli indici multilivello contengono metadati, i quali occupano spazio!

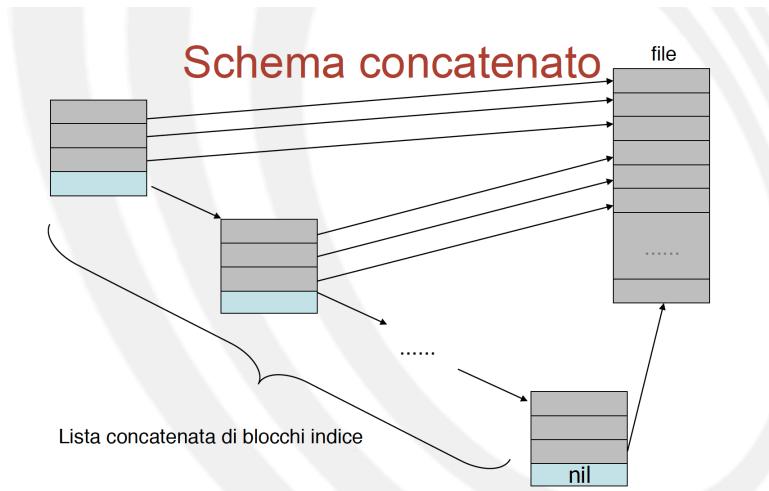


Per risolvere il problema dello spazio occupato dai metadati, si fa un mix tra indici e lista concatenata:

2-sol) Schema Concatenato:

Ho una lista di blocchi indice

- L'ultimo indice di un blocco punta a un altro blocco indice

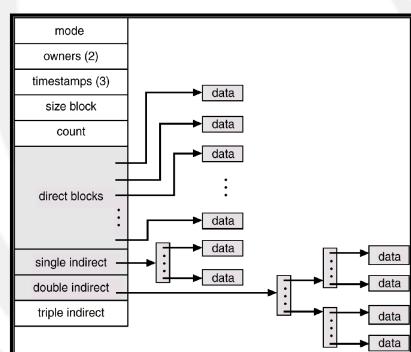


3-sol) Schema Combinato:

Ogni file viene assegnato a un blocco (iNode) che contiene gli attributi del file.

Allocazione indicizzata – esempio

- Schema combinato (Unix)
– blocco di 4KB



In sintesi, per implementare il file system devo gestire l'**allocazione dello spazio su disco**. L'obiettivo è quello di *minimizzare i tempi di accesso e massimizzare l'utilizzo dello spazio*. Esistono 3 alternative per fare ciò:

1) Allocazione Contigua

- a. Ogni file occupa un insieme di blocchi contiguo su disco

2) Allocazione a Lista

- a. Il file può essere frammentato sul disco
- b. Ogni file è quindi una lista di puntatori di blocchi (head-puntatore/body-dati)

3) Allocazione Indicizzata

- a. Ogni file ha un blocco indice che contiene gli indirizzi dei blocchi fisici

Prob. Questa allocazione non permette tutte le dimensioni di file. Per fare ciò bisogna adottare delle varianti:

- Indici multilivello
- Schema concatenato
- Schema combinato

GESTIONE DELLO SPAZIO LIBERO

Per tenere traccia dello spazio libero sul disco si mantiene una lista dei blocchi liberi.

- Per **creare** un file si cercano blocchi liberi nella lista
- Per **rimuovere** un file si aggiungono i suoi blocchi alla lista di quelli liberi

Ci sono diversi metodi per gestire lo spazio libero del disco:

- Vettore di bit
- Lista concatenata (*lista di puntatori ai blocchi liberi*)
- Raggruppamento
- Conteggio

VETTORE DI BIT

Ho un bit per blocco (*1 = libero, 0 = occupato*). Come funziona: Se nel vettore ho un 1 in una parola (i.e. 00001000) vuol dire che è libera.

- La mappa dei bit richiede troppo spazio!
- Facile ottenere file contigui

LISTA CONCATENATA

- Spreco minimo, solo per la testa della lista
- Spazio contiguo non ottenibile

RAGGRUPPAMENTO

- Lista concatenata ma raggruppata

CONTEGGIO

SISTEMI RAID

Il sistema RAID (Redundant array of independent disk) è per gestire un insieme di dischi con le caratteristiche di:

- Migliorare l'affidabilità
- Incrementare le prestazioni

Le strutture RAID **si basano su**:

- Copiatura speculare dei dati (*mirroring*): Affidabilità e rindondanza (aka dischi uguali)
- Sezionamento dei dati (*data striping*)

Posso avere RAID basati su **struttura software**:

- Più dischi indipendenti collegati al bus
- RAID implementato dal SO

Posso avere RAID basati su **struttura hardware**:

- Controllore intelligente gestisce i dischi

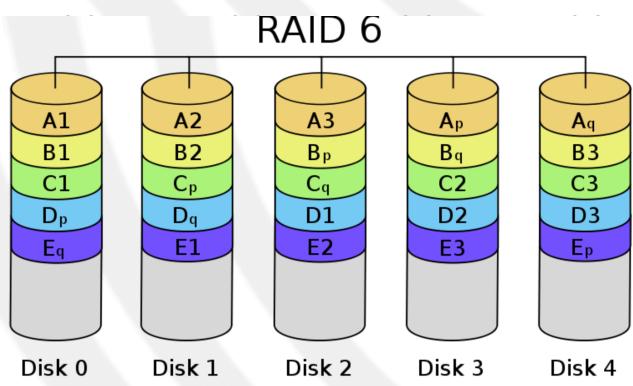
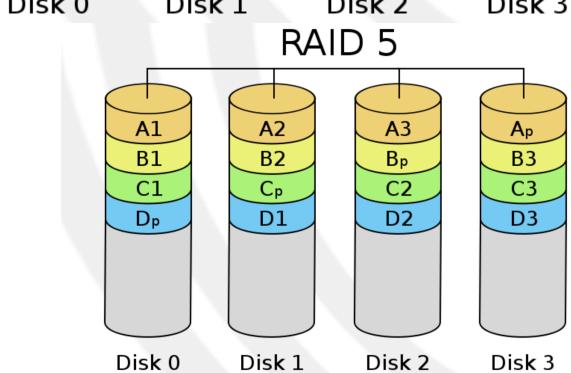
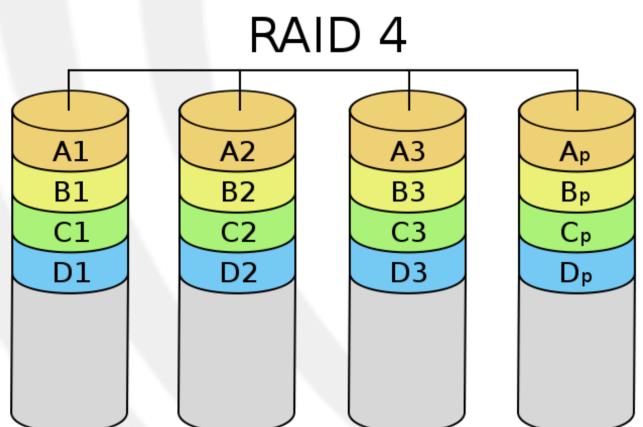
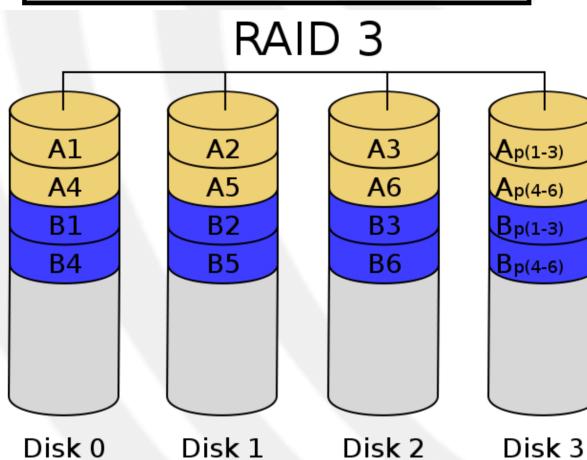
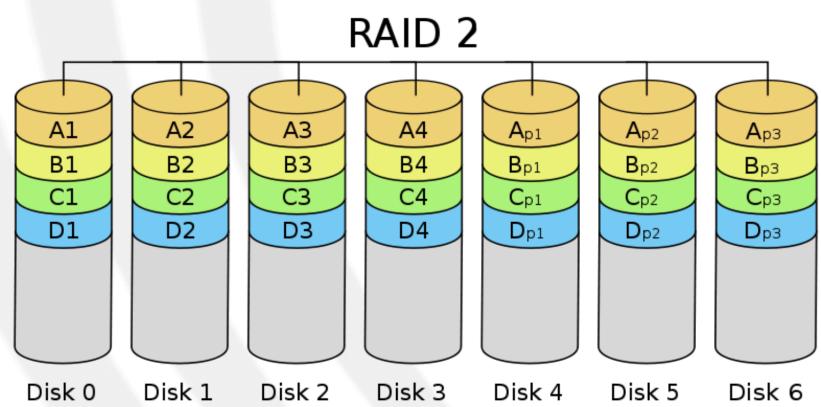
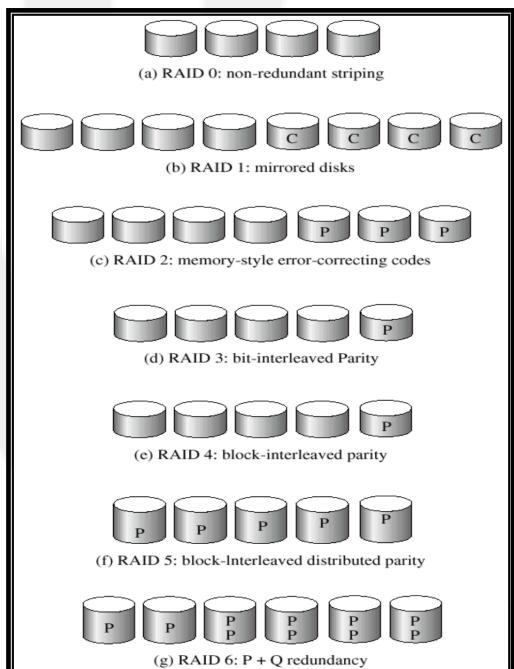
Posso avere RAID basati su **batteria RAID**:

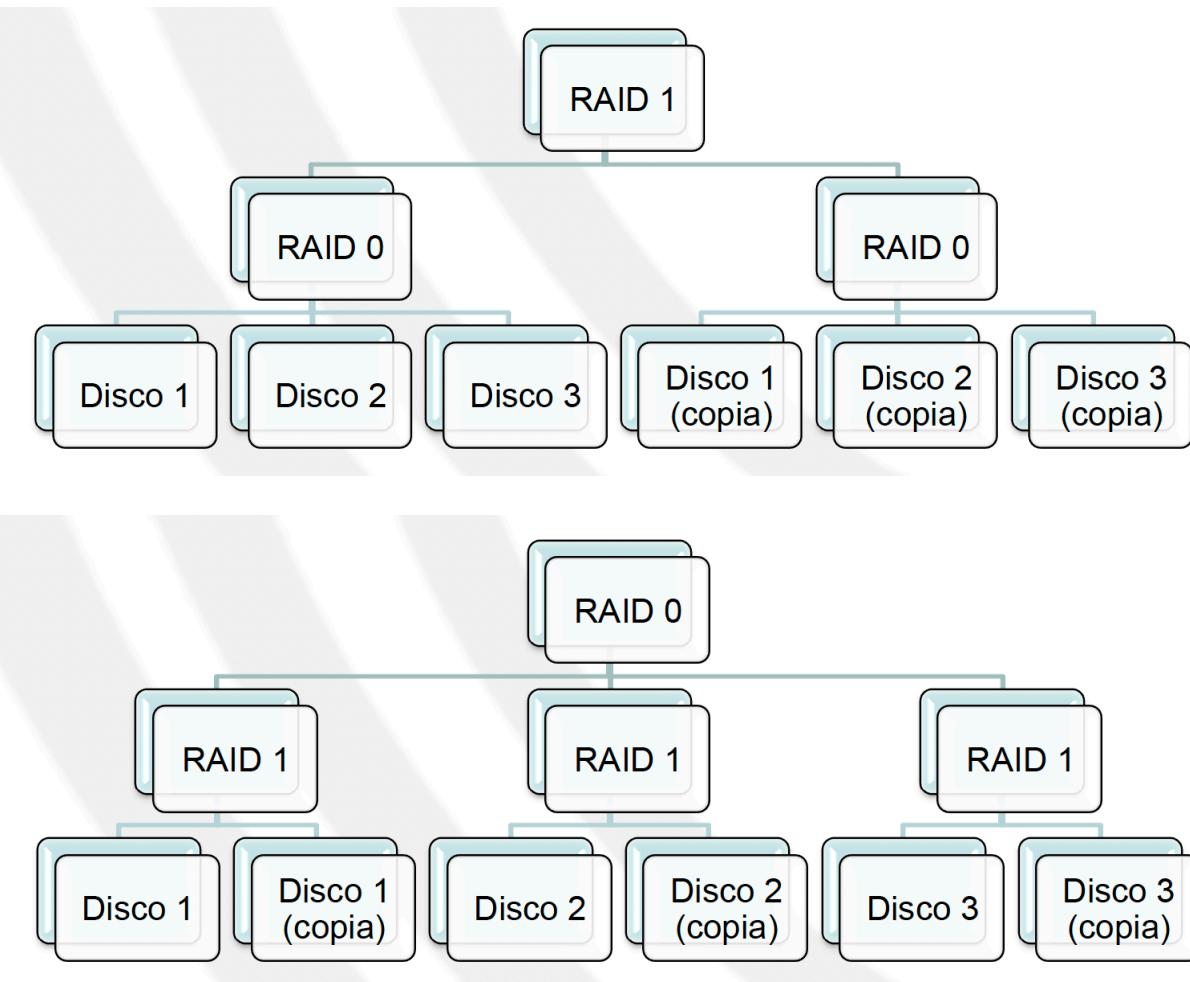
- Unità a se stante con cache, batteria, dischi autonomi

- Si possono sezionare i dati a livello di bit (ogni byte viene separato, 4 bit da una parte e 4 altrove)
- Si possono sezionare i dati a livello di blocco (un insieme di settori su un disco, e un insieme su un altro disco)

Questo (il sezionamento) mi crea il parallelismo:

- Aumento, tramite il bilanciamento del carico, della produttività per accessi multipli a piccole porzioni di dati
- Riduzione del tempo di risposta agli accessi a grandi quantità di dati





- RAID 0: aumento delle prestazioni in lettura e scrittura
- RAID 1: tollera il guasto di un disco, con ricostruzione automatica
- RAID 2: prevede codici a correzione d' errore
- RAID 3: oltre ad avere codici a correzione d' errore, comporta minor sovraccarico
- RAID 4: consente l' esecuzione di letture contemporanee
- RAID 5: letture e scritture contemporanee, parità distribuita tra i dischi
- RAID 6: doppia parità, tollera il guasto di 2 dischi
- RAID 0+1: sfrutta velocità del livello 0, implementando la sicurezza come nel livello 1
- RAID 1+0: Tollera il guasto di 2 dischi

SISTEMA DI I/O

Obiettivo: fornire ai processi utente un'interfaccia efficiente e indipendente dai dispositivi. (standardizzazione)

1) HARDWARE I/O

Il controllore (device driver) è connesso tramite bus al resto del sistema ed è associato ad un indirizzo. Contiene i registri per comandare il dispositivo:

- Registro di stato
- Registro di controllo
- Buffer

Le tecniche di accesso sono:

- **Pooling:**
è il SO che ogni tot guarda i registri del controllore. Determina lo stato del dispositivo mediante la lettura ripetuta del busy-bit
- **Interrupt:**
Il dispositivo "avverte" la CPU tramite un segnale fisico. Risolve il busy-waiting ma crea altre problematiche
- **DMA:**
Pensato per gli grandi spostamenti: aggiunge un controllore hw che non disturba la cpu, e la avvisa tramite interrupt solo al termine del trasferimento.

2) INTERFACCIA I/O

Si utilizzano, astrazione, encapsulamento e sw layering per nascondere le differenze al kernel del SO:
Si crea un'interfaccia comune con un insieme di funzioni standard (syscall).

3) SOFTWARE DI I/O

Fa da tramite tra le periferiche e il SO. Gli obiettivi sono: Gestione degli errori, notazione uniforme, Gestione delle operazioni di trasferimento, prestazioni ec...

Organizzazione per i livelli di astrazione:

1. Gestori degli interrupt
2. Device Driver
3. SW del SO indipendente dal dispositivo
4. Programmi utente

