

## INTRODUZIONE

Un moderno calcolatore è tipicamente formato da uno o più processori, memoria centrale, dischi, stampanti e altre periferiche di I/O. I dettagli di basso livello sono molto complessi, è impossibile comprendere nel dettaglio il funzionamento dei componenti. Gestire tutte queste componenti, dialogare con il processore o con la memoria richiede uno strato intermedio software: il Sistema Operativo.

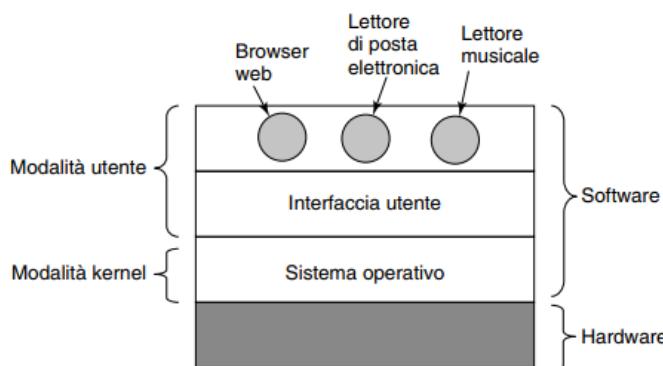
Dentro qualsiasi computer moderno troviamo:

Uno o più processori, la memoria principale, dischi o unità flash, interfacce di rete, dispositivi di I/O.

Alla base troviamo i componenti hardware sopra il software che si divide in:

modalità kernel (o supervisor): ha accesso a tutto l'hardware e può eseguire qualunque istruzione.

modalità utente: è disponibile solo un sottoinsieme delle istruzioni,



Al livello superiore troviamo delle applicazioni, processi che eseguono istruzioni.

La User Interface Program, (Interfaccia utente, shell o GUI) permette di interfacciarsi dal livello applicativo al livello del SO che ci mette a disposizione le chiamate di sistema che interagiscono con l'hardware. Dal livello applicativo non si potrà mai accedere alle risorse senza passare per il SO.

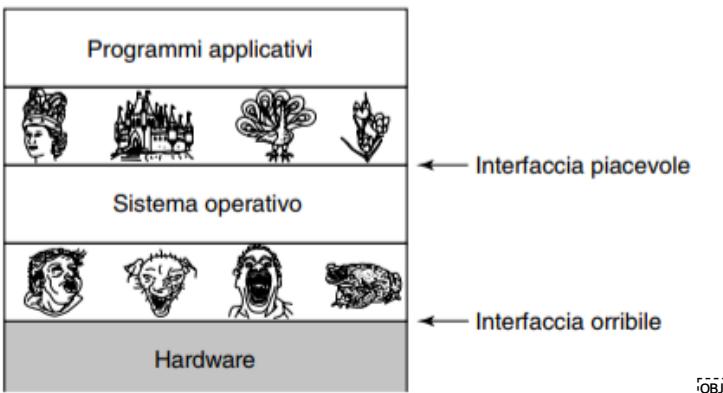
Un sistema operativo fornisce un insieme semplice e pulito di risorse astratte, dall'altra gestisce le risorse hardware in questione.

Idea di astrazione: Il SO si pone tra l'hardware e le applicazioni (obiettivo di SO astrarre l'hardware)

Visione top-down (interfaccia beautiful): Il SO fornisce astrazioni ai programmi applicativi.

Visione bottom-up (interfaccia awful): Il SO gestisce parti di un sistema complesso e fornisce un'allocazione ordinata (simula il parallelismo) e controllata (permette di dedicare le risorse

solo per coloro che hanno il diritto di accedervi) delle risorse.



Un SO gestisce: più programmi in esecuzione e più utenti (non persona fisica, applicazione, processo, etc..).

Le risorse non sono solo hardware: file, database,...

Multiplexing (mettere a disposizione dell'utente in maniera condivisa le risorse):

nel tempo: CPU, stampante,...

nello spazio: memoria centrale, disco,...

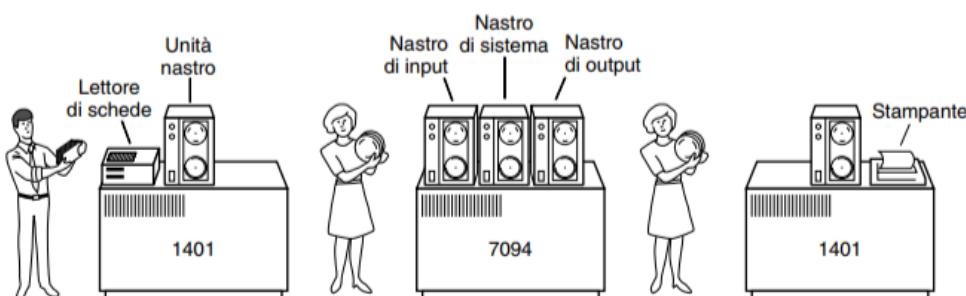
I sistemi operativi si sono evoluti negli anni:

The first generation (1945-55): Vacuum tubes

Erano macchine con tubi vuoti che emettevano luce per simulare gli 1 e 0. Nessun supporto per la programmazione, l'unico modo era spostare i connettori. Usata principalmente per calcoli balistici durante la seconda guerra mondiale e calcoli per come si sarebbe evoluta l'esplosione della bomba atomica. Colossus si occupava di decifrare il codice di Enigma. I calcoli potevano essere eseguiti uno alla volta (limitata ottimizzazione della macchina, quando scrivo o leggo non vengono eseguiti calcoli).

The second generation (1955-65): Transistors and batch systems

Scomposizione delle operazioni della macchina e del sistema operativo in blocchi (I/O operazioni). Componenti come il lettore di schede iniziavano a essere definiti (con programmi, compilatori e sistema operativo) e il lettore di nastri (da schede a nastri). La soluzione adottata fu il sistema bash: i programmi e i dati da elaborare venivano trascritti sui nastri per poi essere inviati all'esecuzione.

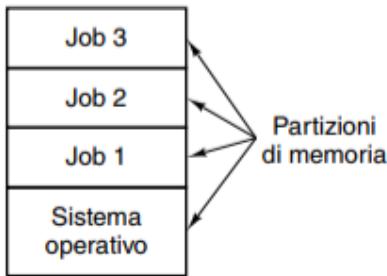


The third generation (1965-1980): ICs and multiprogramming

Era comunque presente il problema delle architetture batch: CPU inattiva in attesa di I/O

Grazie a memorie più grandi venivano caricate più job su un singolo nastro tramite la

partizione di esso, introducendo l'idea di Spooling: caricamento dei nuovi job senza interruzioni. Per risolvere lo spreco di tempo dall'inserimento di un job ad un altro (multiplexing tempo) si è definita l'idea di Time Sharing: mentre i risultati venivano prodotti la CPU veniva assegnata a un job diverso. Non è stata implementata fino agli anni '70 a causa della mancanza di protezione hardware per garantire che, in caso di errore, un processo non scrivesse in un'area destinata ad un altro. MULTICS era un sistema per multiplexing e servizi informatici non ebbe mai veramente successo perché troppo complesso ma introdusse caratteristiche che tutt'oggi ritroviamo nei sistemi operativi.



#### The fourth generation (1980-present): Personal computers

Con lo sviluppo dei circuiti LSI (Large Scale Integration, integrazione su larga scala), chip contenenti migliaia di transistor su un centimetro quadrato di silicio, iniziarono a svilupparsi i primi personal computer (inizialmente chiamati microcomputer).

Intel introduce l'8080 ma necessitò di un SO, Gary Kildall crea CP/M e fondò Digital Research che adattò CP/M per microcomputer, negli anni 80' IBM cerca un SO per il PC, incontra Digital Research ma Kildall rifiuta l'incontro con IBM. Microsoft di Bill Gates quindi acquisisce DOS da Seattle Computer Products e nasce MS-DOS che domina il mercato dei PC IBM.

La GUI fu introdotta negli anni 80' da Doug Engelbart e adottata da Xerox PARC, Steve Jobs ispirato da GUI, sviluppa Apple Macintosh e Microsoft crea Windows come ambiente grafico su MS-DOS.

#### The fifth generation (1990-present): Mobile computers

Il primo vero sistema operativo fu Unix è un sistema operativo multiutente, con multiprogrammazione e open-source. Ken Thompson (Bell Laboratories -1969) definisce UNICS (UNiplexed Information and Computing Service) e insieme a Dennis Ritchie (1970-1974) danno vita a UNIX. Ritchie sviluppa il linguaggio C (partendo dal linguaggio B) linguaggio facilmente compilabile in assembler per poter essere capito dalla macchina e decisero di sviluppare interamente in C. La terza versione di UNIX è scritta in C. Un articolo su UNIX viene pubblicato nel 1974 (ACM Turing Award 1984).

UNIX diventa popolare in ambito accademico e aziendale e si iniziò a creare diverse versioni di quest'ultimo con il problema che UNIX non aveva funzioni di sistema che erano multipiattaforma. Ci fu quindi lo sforzo di standardizzare queste funzioni così da poter utilizzare lo stesso software su tutte le architetture possibili, IEEE sviluppa lo standard POSIX.

Sebbene ci siano molte versioni di UNIX, le implementazioni sono diverse in base alle necessità o all'hardware utilizzato.

Tanenbaum negli anni 80 definì un sistema UNIX diviso in moduli e imposero un approccio modulare così che le chiamate di sistema non potessero interagire con le altre e compatibile con gli standard POSIX, chiamato MINIX.

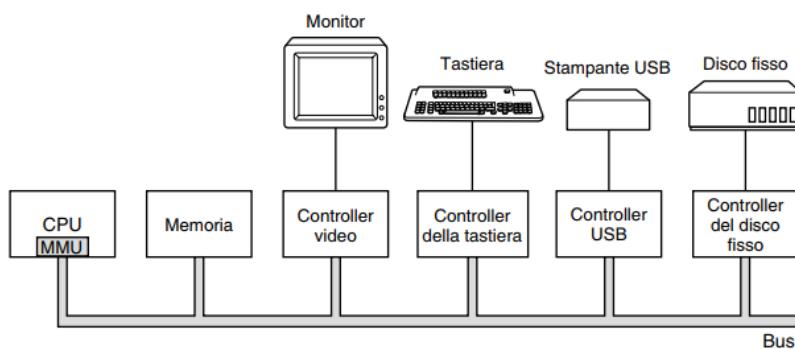
MINIX è un sistema sviluppato a scopo didattico basato sul modello a micro-kernel ([www.cs.vu.nl/~ast/minix.html](http://www.cs.vu.nl/~ast/minix.html)). Ancora usato nei processori moderni.

Da Minix, Torvalds sviluppa Linux cercando di renderlo eseguibile su quanti più processori possibili e sullo standard x86.

Da allora Linux (e android) hanno prosperato soprattutto in ambienti Server e Mobile.

MINIX 3 è stato adottato da Intel per il suo motore di gestione ed è ora presente in desktop, server e laptop.

#### Architettura (semplificata) di un calcolatore



#### Processore:

La CPU è il cervello del computer, esegue istruzioni dalla memoria. Il ciclo base della CPU: preleva (fetch), decodifica (decode), esegue (execute) istruzioni. Le CPU eseguono un set ristretto specifico di istruzioni, i registri interni memorizzano dati importanti e risultati. I set di istruzioni includono anche funzioni per il caricamento/salvataggio dati dalla memoria.

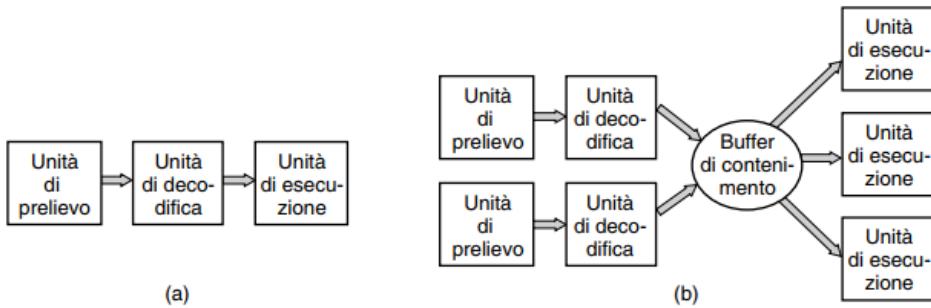
Sono presenti alcuni registri speciali come:

il Program Counter (PC) indica l'area di memoria dell'istruzione successiva,

lo Stack Pointer (SP) punta alla cima dello stack in memoria che contiene frame di procedure con parametri e variabili locali.

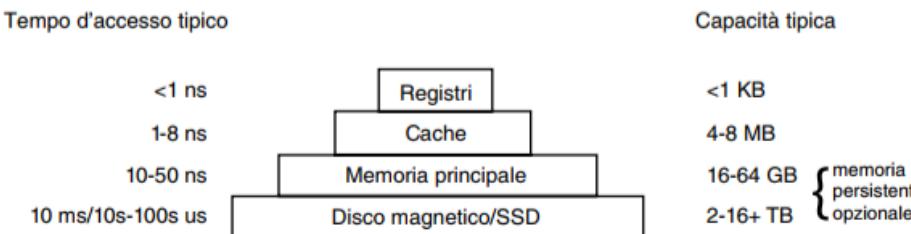
il Program Status Word (PSW) contiene informazioni sullo stato del programma ed è fondamentale per chiamate di sistema e I/O.

Il SO gestisce il multiplexing temporale della CPU. Durante il multiplexing, il SO salva e ripristina i registri costantemente permettendo di eseguire programmi in modo efficiente. L'istruzione successiva viene tradotta eseguita e si ottiene l'output ma per sfruttare tutti i transistor si sono inventate funzionalità avanzate come la Pipeline che mette in coda tutti quelle istruzioni che possano essere svolte in parallelo e vengono eseguite anche in caso di un if che non si verifica (comunque il SO calcola cosa può accadere in caso l'if sia un si o un no per non far sprecare cicli di clock alla CPU successivamente).



Al giorno d'oggi si hanno più di un processore che siano fisici o logici utilizzando Multithreading (o hyperthreading): tiene all'interno della CPU lo stato di due thread ma non c'è una esecuzione parallela vera e propria ma si alternano in millesimi di secondo. Multiprocessori, vantaggi: throughput; economia di scala; affidabilità, in un processore troviamo più core come se fossero dei minichip. La GPU invece sono dei processori utilizzati per determinati calcoli specifici, come il calcolo di poligoni (schede grafiche giochi) o il calcolo matriciale (schede grafiche AI).

Il secondo componente principale di ogni computer è la memoria, idealmente una memoria dovrebbe essere estremamente veloce (più veloce dell'esecuzione di un'istruzione in modo da non bloccare la CPU), molto capiente e a basso costo ma ciò non è possibile.



Più la memoria è veloce e più sarà piccola.

Come prima troviamo i registri che si trovano all'interno della CPU e servono per eseguire le varie istruzioni salvando i dati in essi.

La cache è così utile che le CPU moderne ne possiedono due o anche di più. Il primo livello L1, è sempre all'interno della CPU e fornisce di solito istruzioni codificate al motore di esecuzione della CPU. Molti chip dispongono di una seconda cache L2 per parole di dati usate frequentemente. Le cache L1 hanno 32 KB. La cache L2, contenente molti megabyte di parole di memoria usate recentemente. La differenza fra le cache L1 e quelle L2 sta nelle tempistiche: l'accesso alla L1 avviene senza ritardi, mentre l'accesso alla L2 comporta un ritardo di alcuni cicli di clock. La gestione delle cache è molto problematica: Quando inserire un nuovo elemento nella cache? In quale riga della cache inserire il nuovo elemento? Quale elemento rimuovere dalla cache quando è necessario uno slot. Dove mettere un elemento appena eliminato nella memoria più grande? Tutte queste cose vengono decise dal SO. La memoria principale o RAM (Random Access Memory) riceve tutte le richieste della CPU che non possono essere soddisfatte dalla cache.

Esistono anche ROM ed EEPROM.

Infine abbiamo le memorie non volatili, esistono due tipologie di dischi: HDD (disco) e SSD (flash) il secondo più veloce, costoso e bruciabile al lungo uso.

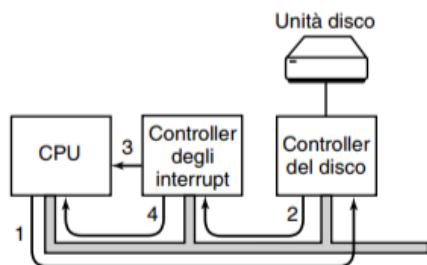
Per interfacciarsi con i dispositivi di I/O si utilizzano due componenti:

il controller: più semplice da usare per il SO;

il dispositivo in sé: interfaccia elementare ma complicata da pilotare.

Ogni controller ha bisogno di un driver per il SO, il driver interagisce con il controller attraverso le porte di I/O, istruzioni tipo IN / OUT e mappatura in memoria.

Per eseguire l'I/O il processo esegue la chiamata di sistema, Il kernel effettua una chiamata al driver e il driver avvia l'I/O interrogando di continuo il dispositivo per vedere se ha finito (busy waiting) o chiede al dispositivo di generare un interrupt che ne segnala il completamento.



Il driver fa uso di hardware speciale: DMA è l'acronimo di "Direct Memory Access" ed è un dispositivo hardware speciale, consente ai componenti di accedere direttamente alla memoria del computer senza coinvolgere la CPU migliorando l'efficienza ed aumentando le prestazioni nelle operazioni di input/output (I/O) ad alta velocità. Riduce il carico sulla CPU durante le operazioni di I/O, consentendole di concentrarsi su altri compiti critici.

**Buses:** L'evoluzione dei computer ha portato all'aggiunta di bus supplementari per gestire l'aumento del traffico dati. Un sistema x86 moderno ha diversi bus con funzioni e velocità di trasferimento diverse, la CPU comunica tramite un bus veloce DDR4 con la memoria, sul bus PCIe con una periferica grafica esterna (Peripheral Component Interconnect Express) e attraverso un hub su un bus DMI (Direct Media Interface) con tutti gli altri dispositivi. Il bus PCIe è il principale e più veloce bus di comunicazione nei computer attuali e utilizza un'architettura a connessioni punto a punto dedicate, migliorando l'efficienza rispetto ai bus condivisi.

Dispositivi legacy possono essere collegati a un processore hub separato. USB (Universal Serial Bus) è stato sviluppato per connettere dispositivi lenti al computer a seconda delle varie versioni cambia la velocità che possono raggiungere. USB utilizza un connettore a 4-11 conduttori per alimentazione e comunicazione e consente il collegamento immediato dei dispositivi senza necessità di riavvio del sistema.

**Avvio del sistema:** La memoria flash della scheda madre contiene il BIOS (Basic Input Output System). Dopo aver premuto il pulsante di accensione, la CPU esegue il BIOS che inizializza RAM e altre risorse, esegue la scansione dei bus PCI/PCIe e inizializza i dispositivi e imposta il firmware runtime per i servizi critici (ad esempio, I/O a basso livello) che il sistema deve utilizzare dopo l'avvio. Il BIOS cerca la posizione della tabella delle partizioni sul secondo settore del dispositivo di avvio ed è in grado di leggere semplici file system oltre ad essere in grado di avviare il primo programma di bootloader (dalla partizione indicata dal boot manager) caricando così il SO.

Un moderno calcolatore è tipicamente formato da: uno o più processori, memoria centrale, dischi, altre periferiche di I/O. I dettagli di basso livello sono molto complessi quindi gestire tutte queste componenti richiede uno strato intermedio software: il Sistema Operativo.

Esistono vari tipi di SO:

Sistemi operativi per mainframe: progettati per gestire sistemi informatici di grande scala, come mainframe aziendali.

Sistemi operativi per server: Ottimizzati per fornire servizi e risorse su reti e su Internet.

Sistemi operativi per personal computer: Utilizzati su computer desktop e laptop per l'uso quotidiano

Sistemi operativi per smartphone e computer palmari: Progettati per dispositivi mobili.

Sistemi operative per IOT e sistemi operativi embedded: Utilizzati in dispositivi embedded e nell'Internet delle Cose.

Sistemi operativi real-time.

Sistemi operativi per smart card (!?!).

Cos'hanno in comune

Extended Machine: Forniscono un'estensione della funzionalità hardware, un astrazione dell'hardware e nascondono i dettagli al programmatore.

Resource Manager: Protegge l'uso simultaneo/non sicuro delle risorse, Condivisione equa (a chi ne ha bisogno di più) delle risorse.

Resource accounting/limiting tenere traccia di quali e quante risorse un processo utilizza e limitarlo in caso.

Il sistema operativo offre funzionalità dell'hardware attraverso le chiamate di sistema (funzioni).

Gruppi di chiamate di sistema implementano servizi, ad esempio File System Service e Process Management Service.

I processi sono astrazioni a livello utente per eseguire un programma per conto dell'utente e ogni processo ha il proprio spazio di indirizzamento (degli indirizzi dove il processo è in grado di leggere e scrivere).

I dati coinvolti nell'elaborazione vengono recuperati e memorizzati in file, tutte le risorse in UNIX sono collegate in qualche maniera alla nozione di file (non un file fisico ma il SO lo tratta come tale). I file persistono rispetto ai processi.

Il Processo è un concetto chiave in tutti i sistemi operativi.

Definizione: Il processo è un programma in esecuzione.

Ad ogni processo è associato: uno spazio di indirizzi dove il progetto può leggere o scrivere, un insieme di risorse come registri, file aperti e allarmi.

Il processo può essere pensato come un contenitore che contiene tutte le informazioni necessarie per l'esecuzione del programma.

Layout dipende dall'architettura della macchina, dal SO e dal programma.

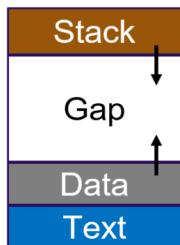
Very basic layout:

Stack: Puntatori ai vari dati utilizzati può aumentare o diminuire nel gap,

Data: Contiene le variabili del programma, tutte le informazioni che servono al programma per essere eseguito può aumentare o diminuire nel gap,

Text: Codice binario del programma.

Gap: Un'area della memoria che può essere espansa o ridotta fornisce spazio per l'allocazione e la deallocazione dinamica della memoria durante l'esecuzione del processo.



Il ciclo di vita di un Processo:

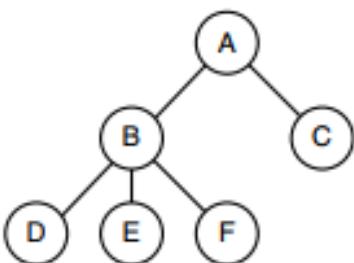
Le informazioni sui processi sono conservate nella tabella dei processi del SO (comando ps).

Avendo più core sui computer moderni si eseguono più processi in parallelo, ma il numero di risorse di calcolo su una macchina è sempre minore del numero dei processi, perciò per fare un multiplexing nello spazio e nel tempo il SO, in particolare lo scheduler, decide quali processi vanno in esecuzione e per quanti secondi così da dare l'illusione che siano eseguiti tutti in parallelo.

Un processo (sospeso) consiste in una voce della tabella dei processi (registri salvati e altre informazioni necessarie per riavviare il processo) e del suo spazio degli indirizzi.

Gestione dei processi: Un processo viene creato quindi allocato nella tabella dei processi, viene terminato quindi deallocato dalla tabella dei processi, e viene messo in pausa e ripreso all'occorrenza.

Un processo può creare (clonato) un altro processo (dal boot partono più processi), conosciuto come processo "figlio", creando una gerarchia (o "albero") di processi.



I processi sono "di proprietà" di un utente, identificato da un UID (User IDentification). Ogni processo ha tipicamente l'UID dell'utente che lo ha avviato. Su UNIX, un processo figlio ha lo stesso UID del suo processo padre. Gli utenti possono essere membri di gruppi, identificati da un GID. Uno specifico UID (superuser/root/administrator) ha più privilegi e può violare molte regole di protezione.

File:

File: astrazione di un dispositivo di memorizzazione (eventualmente) reale (ad esempio, un disco). È possibile leggere e scrivere dati da/su file fornendo una posizione e una quantità di dati da trasferire.

I file vengono collezionati in directory (o cartelle), una directory conserva un identificatore per ogni file che contiene. Una directory è un file a sé stante. La filosofia UNIX: "Everything is a file".

Microsoft sceglie di mettere più punti di accesso collegati ad una lettera dell'alfabeto (A: floppy disk 3.5 inch, B: floppy più grandi, C: disco fisico del sistema operativo, D: CD e altri hard disk, etc..). In Unix le directory e i file formano una gerarchia che inizia dalla "directory principale" o "directory radice" (root, /). In UNIX esistono due tipi di percorsi per identificare un file:

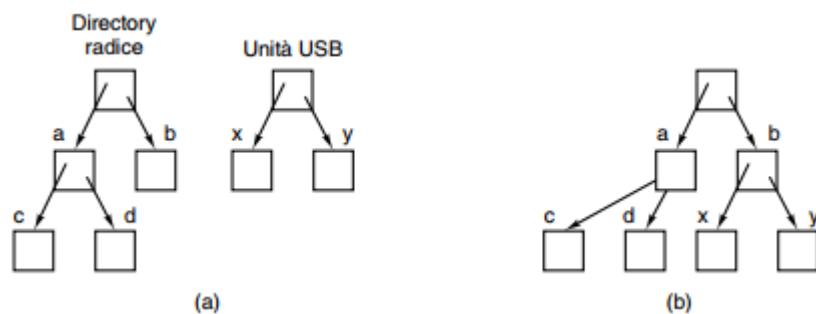
Percorso Assoluto: Inizia dalla radice ("/") e specifica la posizione completa del file, ad esempio: "/home/username/file.txt".

Percorso Relativo: è specifico alla directory corrente e si riferisce ai file relativamente alla directory in cui ci si trova, ad esempio: "../documents/report.pdf".

Altri filesystem possono essere montati (da mount) nella root: "/mnt/windows"

Diritti di accesso: I file sono "protetti" da tuple a tre bit per il proprietario (owner), il gruppo (group) e gli altri utenti (other users). Le tuple contengono un bit (r)ead, (w)rite e un bit e(x)ecute (ma sono disponibili più bit)

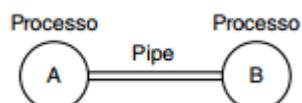
Esempio: -rwxr-x--x, il proprietario può leggere scrivere ed eseguire, il gruppo può leggere ed eseguire mentre gli altri possono solo eseguirlo.



a) Prima del mount, i file dell'unità USB non sono accessibili. b) Dopo aver eseguito l'operazione di mount, fanno parte della gerarchia dei file.

Dispositivi Hardware sono astratti come i file: file speciali a blocchi avranno "b" all'inizio come i dischi, i file speciali a caratteri "c" come le porte seriali, le cartelle "d" altri file speciali: (l) link, file etc..

Pipes: pseudo file che consentono ai processi di comunicare su un canale FIFO (First In First Out) e deve essere impostato in anticipo. Un file "normale" per leggere e scrivere da/sui processi in esecuzione.



Terminologia importante legata ai file include il percorso (path), la directory (folder), la directory di lavoro (working directory, .), il descrittore di file (file descriptor), i file speciali di blocco/carattere e le chiamate di sistema per la gestione dei file.

Chiamate di sistema:

Le chiamate di sistema sono l'interfaccia che il SO offre alle applicazioni per richiedere

servizi.

Problema: Il meccanismo delle chiamate di sistema è altamente specifico del SO e dell'hardware, e la necessità di efficienza aumenta questo problema.

Soluzione: Incapsulare le chiamate di sistema nella libreria C (libc), tipicamente si esporta una chiamata di libreria per ogni chiamata di sistema. UNIX libc si basa sulla libreria C POSIX, esistono molte librerie C UNIX.

Gli 11 passi per effettuare la chiamata di sistema:

Prendiamo una chiamata di sistema come read() che ha tre parametri: il file, puntatore al buffer, il numero di byte da leggere.

1, 2, 3) Preparazione dei parametri: il programma chiamante prepara i parametri solitamente memorizzandoli nei registri o nello stack (file RDI, buffer RSI, n bytes RDX).

4) Chiamata alla procedura di libreria: Il programma effettua la chiamata alla procedura di libreria.

5) Collocazione del numero di chiamata di sistema: colloca il numero della chiamata di sistema in un registro (RAX).

6) Passaggio a modalità kernel: si esegue un'istruzione "trap", un cambio di contesto, (ad esempio, SYSCALL in x86-64) per passare dalla modalità utente a quella kernel. L'istruzione "trap" è simile a una chiamata di procedura ma cambia la modalità in modalità kernel, può saltare solo a indirizzi specifici o indici di una tabella di memoria, a differenza della chiamata di procedura.

7) Il codice del kernel esamina il numero di chiamata di sistema e lo indirizza al correttore gestore di chiamate di sistema tramite una tabella di puntatori.

8) Esecuzione del gestore di chiamate di sistema: Il gestore di chiamate di sistema specifico viene eseguito.

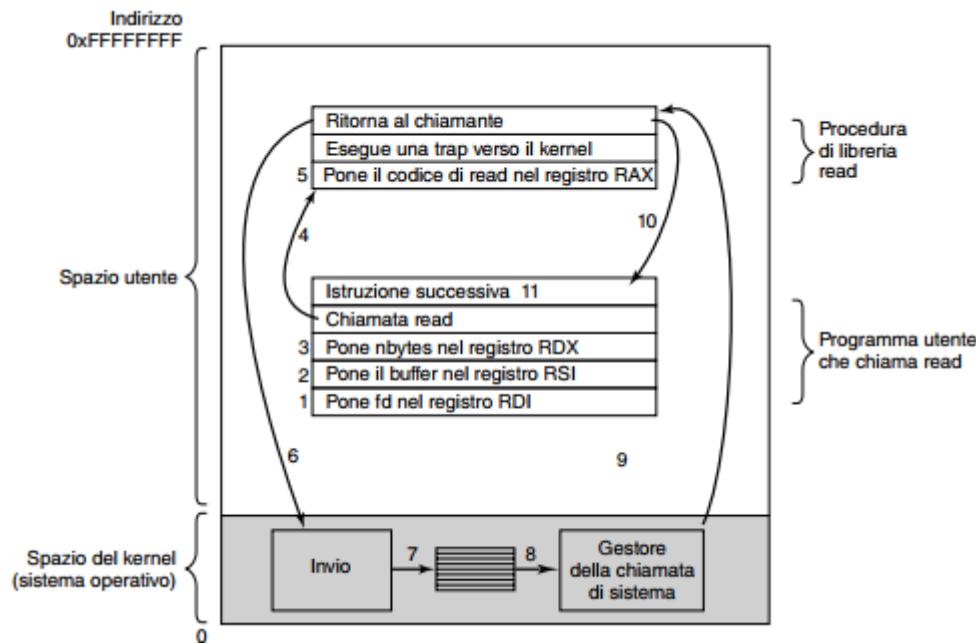
9) Ritorno alla libreria utente: il controllo può essere restituito alla procedura di libreria utente, all'istruzione che segue l'istruzione "trap".

Possibilità di blocco: La chiamata di sistema può bloccare il chiamante, ad esempio, se l'input desiderato non è disponibile, il SO può quindi eseguire altri processi.

Ripresa dopo il blocco: Quando l'input o le condizioni desiderate sono disponibili, il processo bloccato viene ripreso, tornando alla procedura di libreria utente e procedendo all'istruzione successiva.

10) la procedura ritorna al programma utente.

11) Il programma passa all'istruzione successiva.



Chiamate di Sistema per la Gestione dei Processi:

`pid fork():` Crea un processo figlio identico al processo genitore e restituisce il PID (Process Identifier) del processo figlio, restituisce 0 al figlio e il PID del figlio al padre.

`s = execve(name, argv, environp):` Sostituisce l'immagine centrale del processo con un nuovo programma specificato da name, passando gli argomenti in argv e l'ambiente in environp.

`pid waitpid(pid, &statloc, options):` Attende che un processo figlio specificato con il PID termini e restituisce lo stato di uscita del processo figlio.

`exit(status):` Termina l'esecuzione del processo corrente e restituisce uno stato specificato da status.

Chiamate di Sistema per la Gestione dei File:

`fd = open(file, how, ...):` Apre un file specificato da file in modalità lettura, scrittura o entrambe, restituendo un descrittore di file fd.

`s = close(fd):` Chiude un file aperto identificato dal descrittore di file fd.

`n = write(fd, &buffer, nbytes):` Scrive dati dal buffer in un file identificato dal descrittore di file fd, scrivendo nbytes.

`n = read(fd, &buffer, nbytes):` Legge dati da un file identificato dal descrittore di file fd nel buffer, leggendo un massimo di nbytes.

`p = lseek(fd, offset, whence):` Sposta il puntatore di lettura/scrittura in un file identificato dal descrittore di file fd in base all'offset specificato e alla modalità di spostamento whence.

`s = stat(name, &buf):` Ottiene informazioni sullo stato di un file specificato da name e le memorizza nella struttura buf.

Chiamate di Sistema per la Gestione del File System:

`s = mkdir(name, mode):` Crea una nuova directory con il nome specificato da name e con i diritti di accesso specificati da mode.

`s = rmdir(name):` Rimuove una directory vuota con il nome specificato daname.

s = link(name1, name2): Crea un riferimento a un file specificato da name1 con un nome alternativo specificato da name2.  
 s = unlink(name): Rimuove una voce dalla directory specificata da name, eliminando il file associato.  
 s = mount(special, name, flag): Monta un file system con una specifica opzione di montaggio identificata da flag sul punto di montaggio specificato da name.  
 s = umount(special): Smonta un file system identificato da special dal punto di montaggio.  
 s = chdir(dirname): Cambia la directory di lavoro corrente del processo in quella specificata da dirname.  
 s = chmod(name, mode): Modifica i bit di protezione di un file specificato da name in base ai diritti di accesso specificati da mode.  
 s = kill(pid, signal): Invia un segnale specificato da signal a un processo identificato dal PID specificato da pid (nota: questa chiamata di sistema non termina il processo, ma invia un segnale ad esso).  
 s = time(&seconds): Restituisce il tempo trascorso in secondi dal 1 gennaio 1970 e lo memorizza nella variabile seconds.

Esempio uso di fork, waitpid ed execve:

```

#define TRUE 1
while (TRUE) {
    type_prompt();
    read_command(comando, parametri);
    if (fork() != 0)
        /* Codice genitore */
        waitpid(-1, &status, 0);
    /* attende che il processo figlio esca */
} else{
    /* Codice figlio */
    execve(comando, parametri, 0); /* esegue il comando */
}
}
  
```

Per tutti questi comandi esistono i corrispettivi per le API Win32.

Due concetti fondamentali:

Link: è un riferimento ad un altro file, quando viene scaricato qualsiasi app, libreria etc.. si scarica uno specifico pacchetto con uno specifico nome, ma il SO deve fare in modo che si ignori il vero nome del file ma si faccia riferimento ad un nome canonico.

Mount: quando viene aggiunto una risorse (es. disco) viene eseguito il comando mount che monta il disco in maniera virtuale per metterlo a disposizione del SO.

Struttura di un SO:

Esistono diverse strutture di un sistema operativo:

Sistema Monolitico: Il programma principale invoca le chiamate di sistema richieste e il kernel è un blocco monolitico (unico contenitore di comandi) con:

Procedure di servizio che eseguono le chiamate di Sistema.

Procedure di utilità che aiutano a implementare le procedure di servizio.

Un approccio al design «tutto in uno», il kernel è un'unica unità grande e interconnessa, tutte le funzioni del sistema operativo, come la gestione dei processi, la gestione della memoria e la gestione dei dispositivi di I/O, sono strettamente integrate in un unico spazio di indirizzamento.

Flessibilità vs Complessità: offre una certa flessibilità in termini di prestazioni e design, tuttavia, dato che tutto è strettamente interconnesso, un bug o un errore in una parte del sistema può causare problemi in altre parti, potenzialmente portando a crash sistemici.

Compilazione e Collegamento: tutte le funzioni e procedure del sistema operativo devono essere compilate e collegate in un unico eseguibile del kernel.

Mancanza di occultamento: tutte le procedure possono, in teoria, accedere a qualsiasi altra procedura o variabile all'interno del kernel, non c'è un vero e proprio "occultamento" o isolamento tra le diverse parti del sistema.

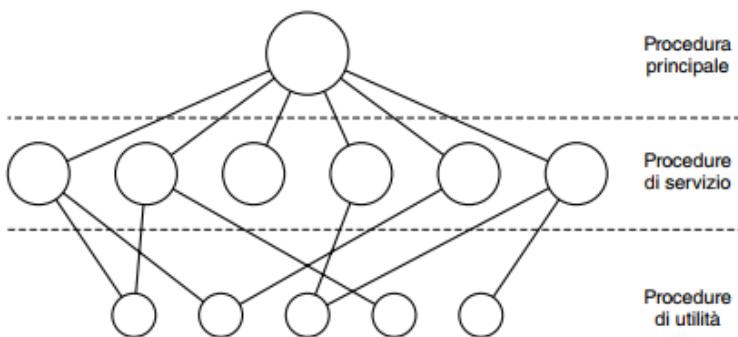
Utilizzo di "trap": meccanismo attraverso il quale un programma può richiedere i servizi del sistema operativo, avviene attraverso interruzioni software che trasferiscono il controllo al sistema operativo.

Composto da una "struttura a tre strati": una suddivisione del sistema in livelli, spesso user mode, kernel mode e hardware, con il "trap" che agisce come meccanismo di comunicazione tra questi livelli.

Estensioni caricabili: Molti sistemi operativi permettono di caricare dinamicamente componenti aggiuntivi, come driver di dispositivi o file system, senza dover riavviare o ricompilare l'intero sistema operativo, questi componenti possono essere caricati e scaricati a seconda delle necessità, offrendo una certa modularità anche in un sistema monolitico.

Librerie Condivise e DLL: Sia UNIX che Windows supportano l'idea di librerie di codice che possono essere condivise tra più programmi, in UNIX, queste sono chiamate "librerie condivise", In Windows sono chiamate "Dynamic Link Libraries" (DLL) e contengono codice che può essere eseguito da più programmi contemporaneamente, riducendo la necessità di avere copie multiple del medesimo codice in memoria.

La struttura base per il sistema operativo monolitico è formata così:  
Un programma principale che richiama la procedura di servizio richiesta, Un set di procedure di servizio che realizzano le chiamate di sistema, Un set di procedure di utilità che aiutano le procedure di servizio.



### Sistema a Livelli:

L'organizzazione stratificata dei sistemi operativi è una generalizzazione dell'approccio monolitico, il sistema THE fu uno dei primi a implementare questa idea, con sei livelli gerarchici. Questi livelli gestivano l'allocazione del processore, la memoria, la comunicazione, l'I/O, i dispositivi, e gli utenti.

Il livello 0 forniva processi sequenziali e una multiprogrammazione base della CPU. Il livello 1 gestiva la memoria e allocava spazio per i processi. Il livello 2 gestiva la comunicazione fra ogni processo e la console dell'operatore. Il livello 3 gestiva i dispositivi di I/O astratti. Il livello 4 ospitava i programmi utente. Il livello 5 gestiva l'operatore di sistema.

Livello	Funzione
5	L'operatore
4	Programmi utente
3	Gestione dell'input/output
2	Comunicazione operatore-processo
1	Gestione della memoria e del tamburo
0	Allocazione del processore e multiprogrammazione

Un altro modello di stratificazione era presente nel sistema MULTICS usava anelli concentrici per definire i privilegi, con livelli interni più privilegiati di quelli esterni.

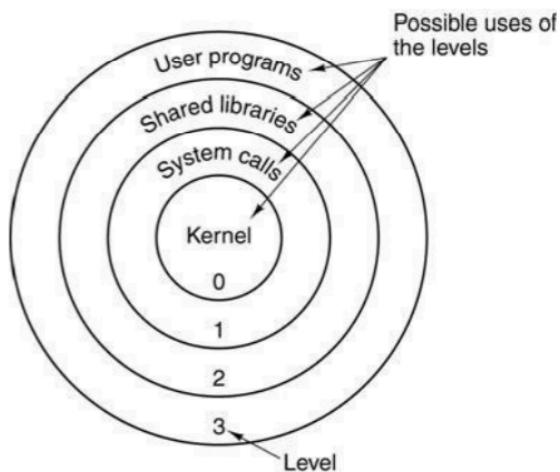
Vantaggi: Protezione delle risorse e dati critici, Separazione chiara dei compiti (es. valutazione degli studenti).

Così facendo si hanno tre proprietà:

Kernel Unificato: Tutte le funzionalità centralizzate in un unico kernel.

Interconnessione: Ogni componente ha la capacità di richiamare qualsiasi altro componente.

Scalabilità: Questa struttura può diventare complessa e meno gestibile con l'evoluzione del sistema, da UNIX ad oggi non è cambiato così tanto.

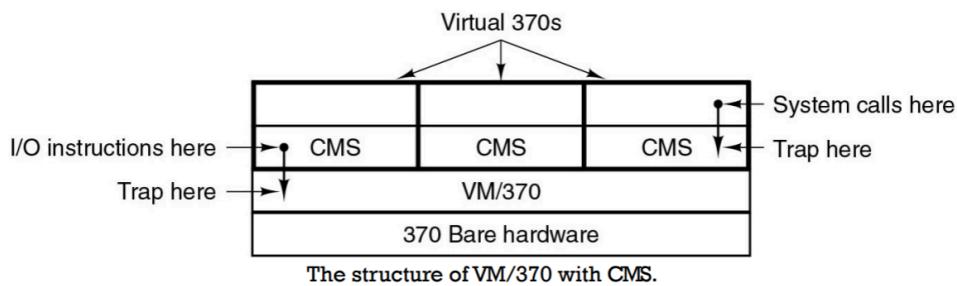


### Virtualizzazione:

Inventato negli anni '70 per separare la multiprogrammazione dalla macchina estesa, ad oggi di nuovo interesse in diversi ambiti, si hanno N interfacce di chiamata di sistema indipendenti dal sistema operativo.

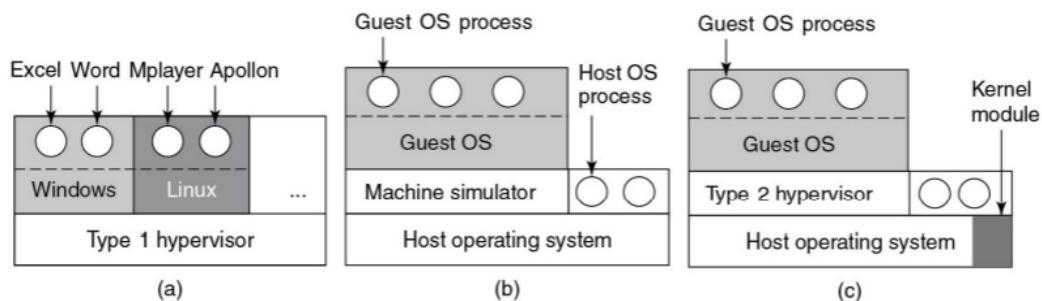
Il cuore del sistema, denominato monitor delle macchine virtuali, gira sul nudo hardware e realizza la multiprogrammazione fornendo non una, ma tante macchine virtuali al livello successivo. Queste macchine virtuali sono la copia esatta dell'hardware, inclusi modalità kernel/utente, I/O, interruzioni e tutto ciò di cui dispone la macchina reale. Ogni macchina virtuale può eseguire un suo sistema operativo che viene eseguito direttamente sull'hardware.

Riescono ad avere più macchine virtuali sullo stesso hardware fisico, condividendo le risorse per ognuna di esse.



Virtual machine monitor (VMM) o Hypervisor emula l'hardware, è il gestore delle virtual machine e ne esistono di tre tipi:

- Type 1: VMM viene eseguito sul "pezzo di ferro" (direttamente su HW, come Xen), non è presente un SO se non l'hypervisor su cui posso lanciare contemporaneamente più SO. (a)
- Type 2: VMM ospitato nel sistema operativo (applicazione sul SO) (esempio: QEMU) (b)
- Hybrid: VMM all'interno del sistema operativo (esempio: KVM) (c)



I container:

I container possono eseguire più istanze di un sistema operativo su una singola macchina. Ogni container condivide il kernel del sistema operativo host e i file binari e le librerie. Il container non contiene il sistema operativo completo e può quindi essere leggero.

I container sono isolati a livello di processo, se un container altera la stabilità del kernel sottostante, ciò può influire sugli altri container.

Vantaggi dei container: Evita la duplicazione delle stesse risorse e file di sistema e crea un ambiente protetto (Sand Box), aumentando la robustezza delle singole macchine.

Svantaggi: Non è possibile eseguire un sistema operativo completamente diverso dal sistema host, perché alla base le risorse sono le stesse, e a differenza delle macchine virtuali, non esiste un rigido partizionamento delle risorse.

### Exokernel:

Idea: Separare il controllo delle risorse dalla macchina estesa, simile a un VMM/Hypervisor, ma: Exokernel non emula l'hardware e fornisce solo una condivisione sicura delle risorse a basso livello mettendo a disposizione solo le chiamate di sistema che servono.

Ogni macchina virtuale a livello utente esegue il suo sistema operativo, ma è limitata a utilizzare solo le risorse assegnate. Rispetto ad altri approcci, l'exokernel elimina la necessità di mappature complesse, concentrando solo su quale macchina virtuale ha accesso a quali risorse.

Ho quindi tre principali vantaggi: sono meno le risorse che utilizzo, sono meno le risorse impegnate e c'è una maggiore sicurezza.

### Unikernel:

Gli unikernel sono sistemi minimi basati su LibOS, progettati per eseguire una singola applicazione su una macchina virtuale come ad esempio un WebServer. Questi sistemi contengono solo la funzionalità necessaria per supportare l'applicazione specifica, come un server web, su una macchina virtuale. Gli unikernel sono altamente efficienti poiché non richiedono protezione tra il sistema operativo (LibOS).

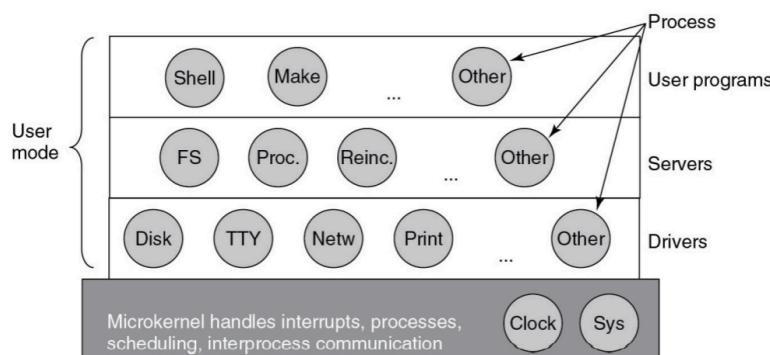
Esiste solo un'applicazione per macchina virtuale.

Il concetto degli unikernel è stato recentemente riscoperto, offrendo una soluzione leggera ed efficiente per eseguire applicazioni isolate su macchine virtuali.

### Microkernel Client-Server:

Organizza le service procedure che vengono eseguiti in modo separato, come se fossero dei moduli ben definiti, di cui solo uno è presente nel kernel e tutti gli altri a livello utente. I processi di sistema comunicano attraverso il passaggio di messaggi. Per passare quindi da un processo ad un altro si devono attraversare diversi moduli. Le chiamate di sistema si basano sullo stesso meccanismo di messaggistica.

Meccanismo di messaggistica implementato nel kernel minimale (Microkernel).



### Struttura semplificata del sistema MINIX.

Pro: Ogni processo del sistema operativo può fare solo ciò che è necessario per svolgere il proprio compito e la compromissione di un driver di stampa non influisce sul resto del sistema operativo.

Contro: Il passaggio di messaggi è più lento di una chiamata di funzione (come in un kernel monolitico).

Linux è un clone di UNIX e mette a disposizione una serie di microprogrammi tra cui:

awk: Pattern scanning and processing language (ex. awk '{print \$3 "\t" \$4}' marks.txt stampa terza e quarta colonna e li separa da un tab e molto altro).

cp: copia file o directory.  
cat: Legge il file e lo mostra a schermo (appoggia tutti i byte nello STDOUT, che all'interno del SO sono due, lo STDOUT e lo STDERR).  
cut: Extract selected fields of each line of a file.  
diff: Compare two files, Dati due STDOUT mostra le righe su cui differiscono.  
grep: Search text for a pattern, Ricerca tutte le linee che contengono una stringa.  
head: Prende un file (o STDOUT) e seleziona le prime n righe.  
less: Display files on a page-by-page basis, Prende STDOUT e lo legge paginato a schermate.  
od: Dump files in various formats.  
sed: Stream editor (esp. search and replace), cerca e sostituisce una stringa con un'altra.  
sort: Prende un file e lo ordina.  
split: Split files.  
tail: Prende un file (o STDOUT) e seleziona le ultime n righe.  
tr: Translate/delete characters.  
uniq: Filter out repeated lines in a file.  
wc: Line, word and character count, conta il numero di linee, parole e caratteri.  
tar: File archive (similar to zip).  
man: manuale che mostra la funzione dei vari comandi/applicazioni.  
pwd: mostra dove ti trovi nel file system.  
wget: dando un link permette di scaricare file dal web.  
chmod: serve per assegnare permessi di rwx per un file o directory.  
history: mostra tutti i comandi eseguiti in precedenza.  
1>: redirect STDOUT (reindirizza i byte all'interno di un file, lo STDOUT mandalo su un file, comando di base se il numero non è specificato).  
2>: redirect STDERR (reindirizza gli errori dello STDERR in un file, se non sono presenti errori il file sarà vuoto).

I comandi sono componibili, utilizzando la pipeline posso prendere l'output di un comando e metterlo in input al successivo (tramite | che sta a indicare la pipe).

La Shell è un programma che interpreta i comandi inseriti e li manda al SO, è un ambiente per lo scripting. La Shell più popolare è BASH.

BASH mette a disposizione variabili d'ambiente usate per leggere e scrivere, ad esempio la variabile \$PATH, queste variabili sono condivise con i programmi che sono eseguiti sulla shell.

printenv: comando che mostra tutte le variabili di sistema già definite.

echo: permette di leggere cos'è presente in una variabile.

Per creare una nuova variabile si usa l'operatore '=' (ex foo='ciao').

whoami: comando che mostra chi è l'utente.

hostname: comando che mostra il nome del computer.

date: mostra la data.

cal: stampa il calendario di questo mese.

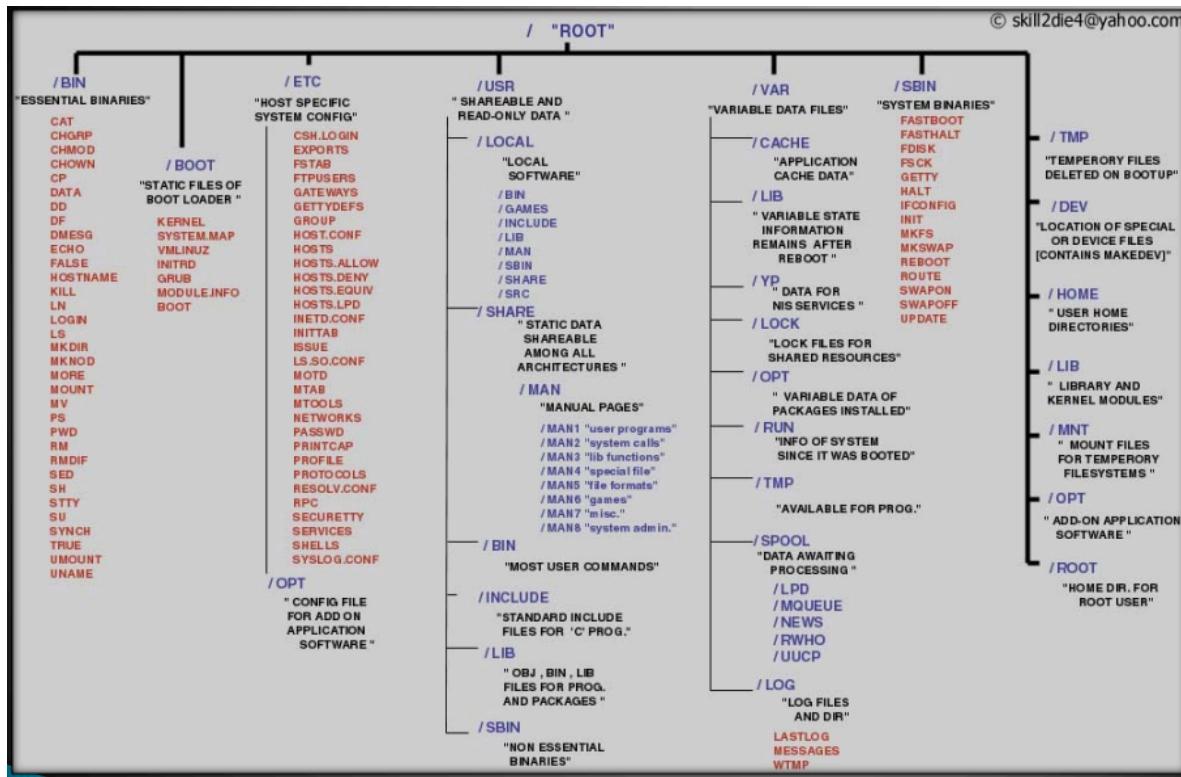
/: cerca tutte le ricorrenze in un file

h: display help

w: mostra chi è collegato

Ogni comando ha tre parti: comando, opzioni e parametri. (ex: cal -j 3 1999, cal comando, -j opzione e 3 e 1999 sono parametri). Le opzioni hanno una short form e una long form.

Un file system in Linux ha una struttura ad albero del tipo:



La cartella root contiene le seguenti cartelle:

Bin: sono presenti tutte le applicazioni di base come cat, echo, etc..

Boot: sono presenti tutte le informazioni che deve caricare in fase di avvio.

Etc: sono presenti tutti i file di configurazione del sistema.

Opt: è dove si consiglia di installare applicazioni.

Usr: tutte le informazioni legati agli utenti.

Var: sono presenti tutte le informazioni variabili, come il log.

Sbin: sono presenti applicazioni specifiche.

Tmp: cartella temporanea.

Dev: è presente la virtualizzazione dei devices, everything is a file.

Home: cartella assegnata agli utenti.

Lib: cartella contenente le librerie .SO del kernel.

Mnt: dove vengono montati hard disk ausiliari.

Root: cartella dedicata all'utente admin.

Per navigare nel file system troviamo diversi comandi:

pwd: print current directory

ls: list files ha diverse opzioni: -a tutti i file, anche i nascosti, -l mostra i dettagli, -h human readable, -s sort, -t sort by modification time.

cd: change directory.

cp: copia file.

mkdir: crea directory.

rmdir: rimuovi directory vuota.

mv: muovi/rinomina file.

rm: rimuovi file.

file: identifica il tipo di file.

find: aiuta a localizzare file in directory.

Interagendo con Linux vengono creati delle istanze di programmi chiamati processi

ps: mostra i processi attivi.

kill: uccide il processo (invia un segnale al processo).

Quando mandiamo un comando in esecuzione esso verrà mandato in foreground, con il carattere & esso verrà mandato in background permettendoci di inserire altri comandi nel frattempo senza aspettare che il comando inserito finisca.

Una volta eseguito un comando in foreground per farlo andare poi in background prima si sospende il programma con Ctrl+Z si esegue 'bg' così facendo il comando verrà continuato in background per portarlo in foreground bisogna inserire 'fg'.

## PROCESSI

Processi e Thread:

Un processo è un programma in esecuzione, per ogni programma può esistere anche più di un processo.

In un sistema multiprogrammato, ciascuna CPU passa rapidamente da un processo all'altro, eseguendo ognuno per decine o magari centinaia di millisecondi. La CPU in realtà esegue un solo processo alla volta (la CPU viene assegnata a turni a diversi processi), ma nel corso di 1 secondo può elaborarne parecchi e dare l'illusione del parallelismo (pseudoparallelismo). Mentre si parla multiprocessore quando ci sono 2 o più processori a livello hardware. Passare da un processo ad un altro richiede una serie di operazioni onerose, così come prendere un processo e cambiargli contesto quindi far sì che non esegua istruzioni nello spazio della memoria dell'applicazione ma in quelle del kernel.

I processi hanno diverse priorità in base alla loro funzione. Uno dei modi per comunicare con i processi che sono in esecuzione nel SO è tramite i segnali.

Eseguendo un processo la riga di comando sarà bloccata finché il processo non sarà terminato, nello stesso tempo ci sono molti altri processi che vengono eseguiti e prendono un pezzetto di tempo di utilizzo all'interno del processore in base anche alla priorità, se volessi mandarlo in parallelo otteniamo l'illusione che ci venga restituita la riga di comando e il processo viene eseguito in background (per la riga di comando aggiungere &).

Il processo è un'astrazione fondamentale del sistema operativo perché consente al sistema operativo di semplificare: Allocazione delle risorse, Accounting (o "Contabilizzazione") delle risorse e Limitazione delle risorse.

Il sistema operativo mantiene informazioni sulle risorse e sullo stato interno di ogni singolo processo del sistema.

In questo modello tutto il software eseguibile sul computer, incluso il sistema operativo, è organizzato in un certo numero di processi sequenziali.

Un processo, è un'istanza di un programma, include i valori attuali del contatore di programma, dei registri e delle variabili.

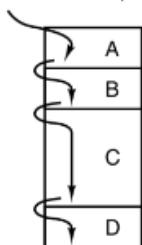
Come funziona il processo:

Consideriamo un computer che multiprogramma quattro programmi in memoria:

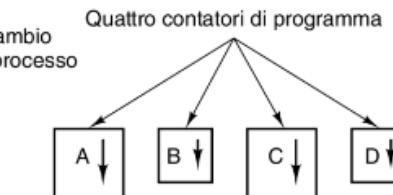
- a) è presente un unico program counter, ogni processo è in un'unica posizione, e la CPU, essendo una sola, passa da un processo all'altro in modo sequenziale.
- b) Ogni processo ha un proprio flusso di controllo e un proprio program counter logico che viene caricato nel program counter fisico quando il processo viene mandato in esecuzione, ogni volta che si passa da un processo all'altro, si salva in memoria il program counter fisico nel program counter logico archiviato del processo.

Nella figura c osserviamo che tutti i processi hanno avuto un avanzamento ma in un dato momento solo uno è realmente in esecuzione.

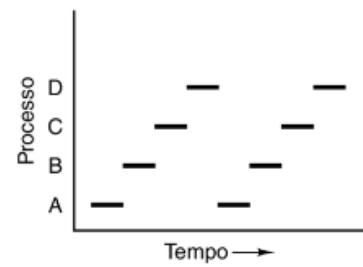
Un contatore di programma



(a)



(b)



(c)

All'interno del sistema operativo esiste un modulo (scheduler) che, in base alla priorità e ha dei criteri detti policy di scheduling, decide quale processo deve essere assegnato prima o dopo alla CPU. Tutto questo avviene perché un obiettivo del sistema operativo è di ottimizzare al massimo le risorse a disposizione, dando l'illusione di essere superveloce.

La CPU può essere assegnata a turno a diversi processi, quindi in linea di principio sono reciprocamente indipendenti e hanno bisogno di mezzi esplicativi per interagire tra loro, il sistema operativo normalmente non offre garanzie di tempistica o di ordine dell'esecuzione.

Gerarchia dei processi:

Generalmente, i processi sono legati da una stretta gerarchia, ognuno è responsabile degli altri processi che genera, così da assicurarsi che una volta che uno termina, terminano tutti gli altri generati da quel processo. Avremo sempre la gerarchia Parent-Child.

Il SO in genere crea solo un processo di init (pid 1), nei moderni sistemi init avvia kthreadd cioè un processo per la gestione dei thread.

I Sottoprocessi sono creati in modo indipendente: Un processo padre può creare un processo figlio, ne consegue una struttura ad albero e gruppi di processi.

Creazione dei Processi:

Quattro eventi principali che causano la creazione di processi:

1. Inizializzazione del sistema.
2. Esecuzione di una chiamata di sistema per la creazione di un processo da parte di un processo in esecuzione ( fork() ).
3. Richiesta dell'utente di creare un nuovo processo, esempio tramite bash.
4. Avvio di un lavoro in modalità batch ( o da bash ).

Termine di un processo:

1. Uscita normale (volontaria).
2. Uscita a causa di un errore (volontaria).
3. Errore "fatale" (involontario come il segmentation fault).
4. Ucciso da un altro processo (involontario come la kill).

## Process Management:

§ fork: crea un nuovo processo, il figlio è un clone "privato" del genitore e condivide alcune risorse con il genitore

§ exec: esegue di un nuovo processo, utilizzato in combinazione con fork

§ exit: causa la terminazione volontaria del processo, lo "stato di uscita" viene restituito al processo "genitore"

§ kill: invia un segnale a un processo (o a un gruppo), può causare la terminazione involontaria di un processo anche SIGTERM ma con un'aggressività diversa.

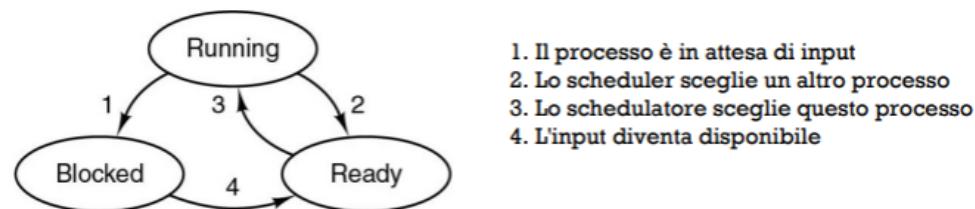
Per mettere in pausa un processo si può mandare il segnale Ctrl+Z, per riavviarlo scrivere sulla riga di comando fg.

## Stati di un processo:

Tre stati in cui può trovarsi un processo:

1. Running/In esecuzione (sta effettivamente utilizzando la CPU in quel momento).
2. Ready/Pronto (eseguibile; temporaneamente fermato per consentire l'esecuzione di un altro processo, e sta in fila per usare la CPU).
3. Blocked/Bloccato (non può essere eseguito fino a quando non si verifica un evento esterno).

Il sistema operativo alloca le risorse (ad esempio, la CPU) ai processi, per farlo deve tenere traccia degli stati dei processi stessi:



## Informazioni associate a un processo:

ID (PID), Utente (UID), Gruppo (GID), Spazio degli indirizzi di memoria, Registri hardware (ad esempio, il Program Counter), File aperti, Segnali (Signal) e Interrupt.

Queste informazioni sono memorizzate nella tabella dei processi del sistema operativo.

## "Signals" e "Interrupts":

Sono meccanismi utilizzati nei sistemi operativi e nelle applicazioni per gestire eventi asincroni.

### • Interrupts:

Origine: Dispositivi hardware (es. tastiera, disco rigido).

Gestione: Tramite routine di servizio di interrupt (ISR).

Uso: Comunicazione tra hardware e software; risposta pronta agli eventi hardware.

Asincronia: Si verificano in modo asincrono; gestiti immediatamente.

### • Signals:

Origine: Eventi software; generati da un processo o dal SO.

Gestione: Gestori di segnali personalizzati o comportamento predefinito.

Uso: Gestione condizioni eccezionali nelle applicazioni.

Asincronia: Inviati asincronamente; possono essere gestiti in modo sincrono, la gestione è del processo è lui che decide come ordinarli.

### Interrupt:

Idea: per deallocare la CPU a favore dello scheduler, ci si affida al supporto per la gestione degli interrupt fornito dall'hardware. Permette allo scheduler di ottenere periodicamente il controllo, cioè ogni volta che l'hardware genera un interrupt.

Interrupt vector: Associato a ciascun dispositivo di I/O e linea di interrupt. Parte della tabella dei descrittori di interrupt (IDT), contiene l'indirizzo iniziale di una procedura interna fornita dal sistema operativo (gestore degli interrupt).

Schema di ciò che fa il livello più basso del sistema operativo quando si verifica un'interruzione.

1. L'hardware impila il Program Counter e le altre informazioni del processo.
2. L'hardware carica il nuovo contatore di programma dal vettore di interrupt.
3. La procedura in linguaggio assembly salva i registri.
4. La procedura in linguaggio assembly imposta un nuovo stack.
5. Il servizio di interrupt C viene eseguito (tipicamente legge e esegue il buffer dell'input).
6. Lo scheduler decide quale processo deve essere eseguito successivamente.
7. La procedura C ritorna al codice assembly.
8. La procedura in linguaggio assembly avvia il nuovo processo (corrente).

Ogni volta che si verifica un'interruzione, lo scheduler ottiene il controllo agendo come mediatore. Un processo non può cedere la CPU a un altro processo (context switch) senza passare attraverso lo scheduler.

Esistono diversi tipi di Segnali:

Hardware-induced (e.g., SIGILL) o Software-induced (e.g., SIGQUIT or SIGPIPE).

Azioni possibili: Term, Ign, Core, Stop, Cont, sono azioni predefinite per ogni segnale, tipicamente sovrascrivibile, i segnali possono essere bloccati e le azioni ritardate.

I segnali devono essere gestiti:

Gestione (catching) dei segnali: Il processo registra il gestore del segnale, il kernel invia un segnale e consente al processo di eseguire l'handler, interrompe il codice in esecuzione e salva il contesto poi esegue il codice di gestione del segnale e ripristina il contesto originale.

### Thread:

Ci sono situazioni in cui è utile avere più processi in esecuzione nello stesso spazio di indirizzi, in parallelo. La Multithreaded execution implica che 1 processo può avere N thread in esecuzione. I thread sono Lightweight processes (Processi leggeri) e consentono un parallelismo efficiente in termini di spazio e di tempo, una comunicazione e una sincronizzazione (necessaria per non far accedere due thread in una regione critica) semplici. Essendo più leggeri (non hanno PID né spazio di indirizzi unico) sono più facili da creare e distruggere, inoltre quando c'è un'attività di elaborazione considerevole unita a I/O altrettanto considerevole, i thread permettono alle due attività di sovrapporsi e l'applicazione risulta più veloce. Non devono sottostare al controllo dello scheduler perché è il processo che li gestisce.

### Modello e Caratteristiche:

Thread: Parallelismo, Chiamate di sistema bloccanti.

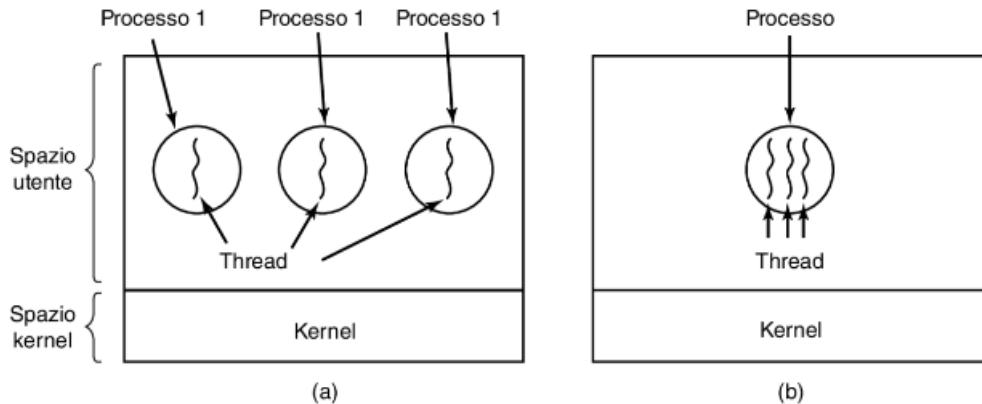
Processo a thread singolo: Nessun parallelismo, Chiamate di sistema bloccanti.

Finite-state machine: Parallelismo, Chiamate di sistema non bloccanti, Interrupt.

### Thread e Processi:

I thread risiedono nello stesso spazio degli indirizzi di un singolo processo. Tutti gli scambi di informazioni avvengono tramite dati condivisi tra i thread, i thread si sincronizzano tramite semplici primitive. Ogni thread ha il proprio program counter, i propri stack (che contiene la storia della sua esecuzione), i propri registri hardware e il proprio stato.

Esiste una Tabella/interrupt dei thread. Ciascun thread può chiamare qualsiasi chiamata di sistema supportata dal sistema operativo per conto del processo a cui appartiene.



(a) Tre processi con un thread ciascuno. (b) Un processo con tre thread.

Per processo abbiamo quindi: Spazio di indirizzi, Variabili globali, File aperti, Processi figli, Allarmi, Segnali e Gestore dei segnali e Informazioni.

Per thread invece abbiamo: Program counter, Registri, Stack, Stato.

Alcune chiamate di funzione di Pthreads:

`pthread_create`: Crea un nuovo thread.

`pthread_exit`: Termina il thread chiamante.

`pthread_join`: Attende l'“uscita” di uno specifico thread.

`pthread_yield`: Rilascia la CPU per consentire l'esecuzione di un altro thread.

`pthread_attr_init`: Crea e inizializza la struttura di attributi di un thread.

`pthread_attr_destroy`: Rimuove la struttura di attributi di un thread.

Esistono tre luoghi di implementazione dei thread:

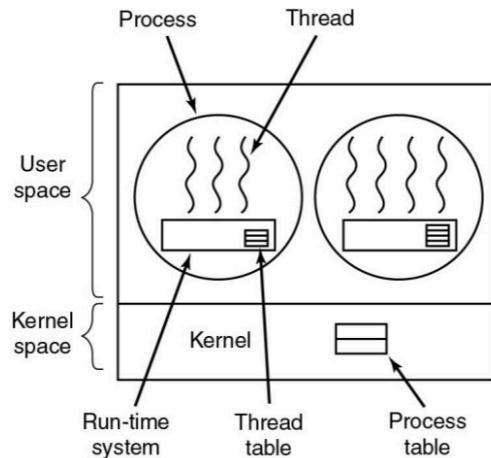
- Nello Spazio Utente:

sono gestiti dal kernel come processi ordinari a singolo thread, possono essere eseguiti su sistemi operativi che non supportano direttamente i thread, sono gestiti tramite una libreria. Ogni processo che usa thread a livello utente necessita di una propria tabella dei thread per tracciare lo stato e altre proprietà dei suoi thread. L'interruzione e il cambio tra thread a livello utente non richiedono un cambiamento di contesto completo. Quindi non ci sono trap e sono molto più veloci rispetto alle operazioni nel kernel. Offrono l'abilità di personalizzare l'algoritmo di scheduling per ogni processo e una maggiore scalabilità.

Tuttavia, ci sono problemi con le chiamate di sistema bloccanti, se un thread fa una chiamata che lo blocca, tutti gli altri thread nel processo vengono fermati. Gli errori di pagina, dove un programma accede a memoria non presente, possono bloccare l'intero processo quando sono causati da un thread a livello utente.

I thread nello spazio utente non hanno interrupt del clock, rendendo impossibile uno

scheduling di tipo round-robin. Sebbene i thread a livello utente siano più veloci e flessibili, sono meno adatti per applicazioni in cui i thread si bloccano frequentemente, come i web server multithread.

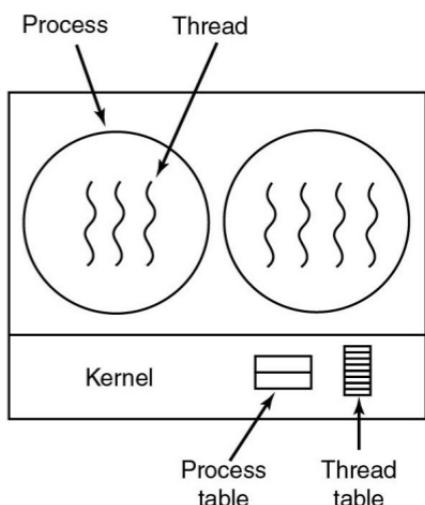


- Nel Kernel: Il kernel che gestisce i thread elimina la necessità di un sistema run-time per processo, la tabella dei thread del kernel conserva informazioni simili a quelle dei thread a livello utente. Le chiamate che potrebbero bloccare un thread vengono implementate come chiamate di sistema avendo costi più elevati rispetto alle chiamate di procedura dei sistemi run-time e se un thread si blocca, il kernel può eseguire un altro thread, sia dello stesso processo sia di un altro.

Alcuni sistemi "riciclano" i thread per ridurre i costi, invece che terminarli.

Se un thread causa un errore di pagina, il kernel verifica la disponibilità di altri thread eseguibili nel processo e può eseguire uno di essi.

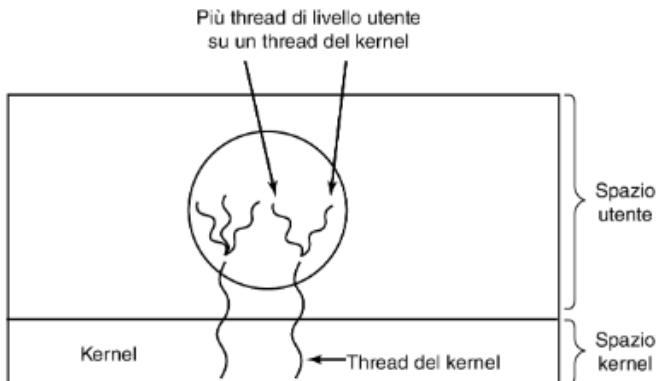
La programmazione con thread richiede cautela per evitare errori.



- Un'implementazione Ibrida: Alcuni sistemi effettuano il multiplexing dei thread utente sui thread del kernel, combinano i vantaggi dei due approcci.

I programmatore decidono quanti thread del kernel utilizzare e quanti thread utente multiplexare, avendo maggiore flessibilità.

Il kernel è consapevole solo dei thread del kernel ma ogni thread del kernel può gestire più thread a livello utente.



Molte procedure di libreria possono causare conflitti se un thread sovrascrive dati cruciali per un altro. L'implementazione di wrappers (impostare un bit per segnalare che la libreria è in uso) può evitare conflitti, ma limita il parallelismo. La gestione dei segnali è complicata, alcuni sono specifici per un thread, mentre altri no, decidere chi deve gestire questi segnali e come gestire conflitti tra thread può essere sfidante.

I processi hanno bisogno di un modo per comunicare (IPC) cioè condividere i dati durante l'esecuzione. I processi hanno bisogno di un modo per sincronizzarsi così da tenere conto delle dipendenze, evitare che si intralcino a vicenda applicando questa sincronizzazione anche all'esecuzione multithread.

Le Race conditions:

Esempio di “race conditions”: Il processo A legge in=7 e decide di aggiungere il suo file in quella posizione ma viene sospeso dal Sistema operativo (perché il suo slot è scaduto). Anche il processo B legge in=7 e inserisce il suo file in quella posizione impostando in=8 e venendo infine sospeso. A viene ripreso e scrive il suo file nella posizione 7 sovrascrivendo B.

Problema: la lettura/aggiornamento di un file dovrebbe essere un'azione atomica. Se non lo è, i processi possono “gareggiare” tra loro e giungere a conclusioni errate.

Bisogna avere dei requisiti per evitare le “race conditions”:

1. Due processi non possono trovarsi contemporaneamente all'interno delle rispettive regioni critiche.
  2. Non si possono fare ipotesi sulla velocità o sul numero di CPU.
  3. Nessun processo in esecuzione al di fuori della propria regione critica può bloccare altri processi.
  4. Nessun processo deve aspettare all'infinito per entrare nella propria regione critica.
- Esiste anche la situazione limite del deadlock, cioè k processi che vogliono fare qualcosa e si bloccano tutti tra loro.

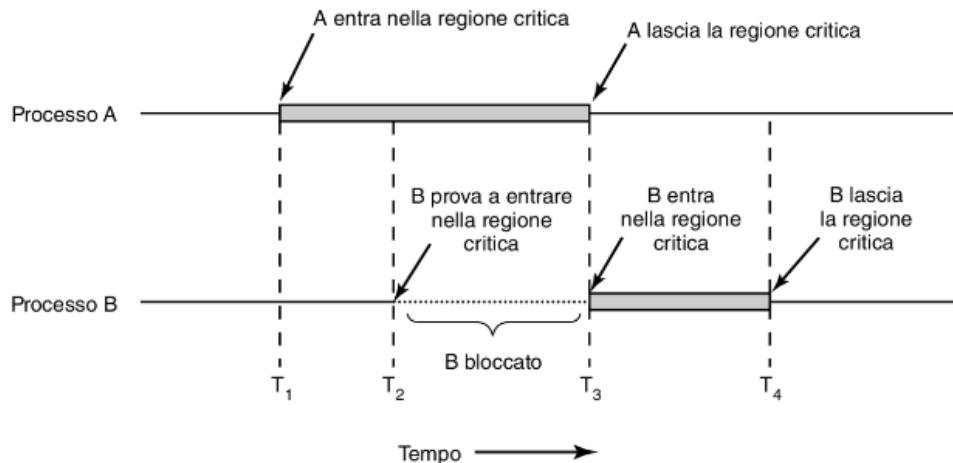
(NON) soluzioni:

Disabilitare gli interrupt: impedisce semplicemente che la CPU possa essere riallocata, funziona solo per sistemi a CPU singola (lascia che il processo finisca per se).

Bloccare le variabili: proteggere le regioni critiche con variabili 0/1. Le “corse” si verificano ora sulle variabili di blocco.

Mutua Esclusione:

Esclusione reciproca tramite regioni critiche:



Mutua Esclusione reciproca con busy waiting: Alternanza Rigorosa:

```
while (TRUE) {
    while (turn != 0) { }          /* ciclo */
    critical region( );
    turn = 1;
    noncritical region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1) { }          /* ciclo */
    critical region( );
    turn = 0;
    noncritical region( );
}
```

(b)

Abbiamo due processi a e b che competono per la regione critica.

Purtroppo, questa è un'altra (non) soluzione: Non permette ai processi di entrare nelle loro regioni critiche per due volte di seguito e un processo fuori dalla regione critica può effettivamente bloccarne un altro. Viene chiamato il busy waiting perché il processore continuamente, controlla in questo caso se il turn cambia valore, consumando cicli di clock.

## Mutua Esclusione reciproca con busy waiting: Peterson's Algorithm

```
#define FALSE      0
#define TRUE       1
#define N          2           /* numero di processi */
int turn;                  /* A chi tocca? */
int interested[N];         /* Tutti i valori inizialmente 0 (FALSE) */
void enter_region(int process);    /* process è 0 o 1 */
{
    int other;             /* numero dell'altro processo */
    other = 1 - process;   /* l'opposto del processo */
    interested[process] = TRUE;  /* mostra che si è interessati */
    turn = process;        /* imposta il flag */
    while (turn == process && interested[other] == TRUE) /* istruzione null */ ;
}
void leave_region(int process)    /* process: chi esce */
{
    interested[process] = FALSE; /* indica l'uscita dalla regione critica */
}
```

Alice e Bob vogliono usare un'unica postazione computer in un ufficio. Ma ci sono delle regole: Solo una persona può usare il computer alla volta, Se entrambi vogliono usarlo contemporaneamente, devono decidere chi va per primo.

Idea dell'algoritmo:

Alice o Bob devono segnalare il loro interesse a usare il computer.

Se l'altro non è interessato, la persona interessata può usarlo subito.

Se entrambi mostrano interesse, registrano il loro nome su un foglio. Ma se scrivono quasi allo stesso tempo, l'ultimo nome sul foglio ha la precedenza.

La persona che non ha la precedenza aspetta finché l'altra ha finito.

Una volta finito, la persona che ha usato il computer segnala che ha finito, e l'altra può iniziare.

Funzionamento TSL:

Quando lock è 0: Un processo può impostare lock a 1 con TSL e accedere alla memoria condivisa, al termine, il processo resetta lock a 0.

Metodo per gestire Regioni Critiche: Processi chiamano enter\_region prima di entrare nella regione critica e leave\_region dopo, se chiamati correttamente, garantisce la mutua esclusione. ma se usati in modo errato, la mutua esclusione fallisce.

Istruzione TSL (Test and Set Lock): Legge il contenuto della parola di memoria lock in un registro e salva un valore non zero in lock. L'operazione è indivisibile: nessun altro processore può accedere finché TSL non è completata, inoltre bloccare il bus della memoria impedisce altri accessi alla memoria da altre CPU.

Istruzione XCHG (Exchange): Scambia i contenuti di due posizioni (es. un registro e una parola di memoria) atomicamente, utilizzate dalle CPU x86 Intel per sincronizzazione di basso livello.

#### UTILIZZO DELL'ISTRUZIONE TSL:

```
enter_region:  
    TSL REGISTER,LOCK      | copia il lock nel registro e lo imposta a 1  
    CMP REGISTER,#0        | il lock era zero?  
    JNE enter_region       | se non era zero, il lock era stato impostato, per cui esegui il ciclo  
    RET                   | torna al chiamante; si è entrati nella regione critica  
  
leave_region:  
    MOVE LOCK,#0          | memorizza 0 in lock  
    RET                   | torna al chiamante
```

#### UTILIZZO DELL'ISTRUZIONE XCHG:

```
enter_region:  
    MOVE REGISTER,#1      | mette un 1 nel registro  
    XCHG REGISTER,LOCK     | scambia il contenuto del registro e della variabile lock  
    CMP REGISTER,#0        | il lock era zero?  
    JNE enter_region       | se non era zero il lock era stato impostato, per cui esegui il ciclo  
    RET                   | torna al chiamante; si è entrati nella regione critica  
  
leave_region:  
    MOVE LOCK,#0          | memorizza 0 in lock  
    RET                   | torna al chiamante
```

Entrambe le istruzioni sono essenziali per garantire la sicurezza nelle operazioni condivise tra più processori.

Le soluzioni finora adottate consentono a un processo di tenere occupata la CPU in attesa di poter entrare nella sua regione critica. (spin lock, cioè girare a vuoto) SPRECO DI RISORSE!!!

Soluzione: lasciare che un processo in attesa di entrare nella sua regione critica restituisca volontariamente la CPU allo scheduler.

```
void sleep() {  
    set own state to BLOCKED;  
    give CPU to scheduler;  
}  
  
void wakeup(process) {  
    set state of process to READY;  
    give CPU to scheduler;  
}
```

Programmazione concorrente nel problema produttore-consamatore:

Nel problema del produttore-consamatore, due processi condividono un buffer di dimensioni

fisse. Il produttore inserisce informazioni nel buffer, mentre il consumatore le preleva. Il produttore si addormenta (entra in modalità «sleep») se il buffer è pieno e viene risvegliato (wakeup, viene riattivato) quando il consumatore preleva dati. Analogamente, il consumatore dorme se il buffer è vuoto e viene risvegliato quando il produttore inserisce dati.

```
#define N 100      /* numero di posti nel buffer */

int count = 0;      /* numero di elementi nel buffer */

void producer(void)

{

    int item;

    while (TRUE) {      /* ripeti sempre */

        item = produce_item( );      /* genera l'elemento successivo */

        if (count == N) sleep( );      /* se il buffer è pieno, vai a dormire */

        insert_item(item);      /* metti l'elemento nel buffer */

        count = count + 1;      /* incrementa il numero di elementi nel buffer */

        if (count == 1) wakeup(consumer);      /* il buffer era vuoto? */

    }

}

void consumer(void)

{

    int item;

    while (TRUE) {      /* ripeti sempre */

        if (count == 0) sleep( );      /* se il buffer è vuoto, vai a dormire */

        item = remove_item( );      /* togli un elemento dal buffer */

        count = count - 1;      /* decrementa il numero di elementi nel buffer */

        if (count == N - 1) wakeup(producer); /* il buffer era pieno? */

        consume_item(item);      /* stampa l'elemento */

    }

}
```

Problema: Il consumatore potrebbe essere risvegliato un attimo prima di andare a dormire.  
Possibile soluzione:

Aggiungere al tutto un bit di attesa del wakeup, che viene impostato quando viene inviato un wakeup a un processo non ancora dormiente. Più tardi, quando il processo tenta di entrare in sleep, se il bit di attesa del wakeup è acceso, il processo lo spegne e rimane ancora sveglio. Il bit di attesa del wakeup è un salvadanaio per accumulare segnali di wakeup. Il

consumatore ripulisce il bit di attesa wakeup a ogni iterazione del ciclo. E' un workaround, non funziona sempre...

### Mutua Esclusione: Semafori

Creato da E. W. Dijkstra nel 1965 per contare e gestire i "wakeup".

Valori: Può essere 0 (nessun wakeup) o un numero positivo (wakeup in attesa).

Operazioni:

Down, se il valore del semaforo è maggiore di zero, questo valore viene decrementato, e il processo continua la sua esecuzione. Se il valore del semaforo è 0, il processo che ha invocato down viene bloccato e messo in una coda di attesa associata al semaforo, in altre parole, il processo "va a dormire".

Up, se il valore è 0, ci sono processi nella coda di attesa, vengono «svegliati» (eventualmente per entrare in competizione ed eseguire di nuovo down). In ogni caso, il valore viene incrementato e il processo continua la sua esecuzione.

Atomicità: Le operazioni sui semafori sono «indivisibili», evitando conflitti.

Problema Produttore-Consumatore: Uso dei semafori per gestire accesso e capacità di un buffer.

Tipi di Semafori: mutex (non un vero e proprio mutex ma un semaforo, mutual exclusion cioè accesso esclusivo), full (tutti posti occupati) empty (tutti posti liberi).

Uso: mutex previene accessi simultanei, full blocca il consumatore quando il buffer è vuoto e empty blocca il produttore quando il buffer è pieno, insieme coordinano attività.

## Semafori: Produttore - Consumatore

```
#define N 100      /* numero di posti nel buffer */

typedef int semaphore;      /* i semafori sono un tipo speciale di int */

semaphore mutex = 1;        /* controlla l'accesso alla regione critica */

semaphore empty = N;        /* conta i posti vuoti del buffer */

semaphore full = 0;         /* conta i posti pieni del buffer */

void producer(void)

{

    int item;

    while (TRUE) {      /* TRUE è la costante 1 */

        item = produce_item( );      /* genera qualcosa da mettere nel buffer */

        down(&empty);      /* decrementa il contatore empty */

        down(&mutex);      /* entra nella regione critica */

        insert_item(item);      /* mette un nuovo elemento nel buffer */

        up(&mutex);      /* lascia la regione critica */

        up(&full);       /* incrementa il contatore dei posti pieni */

    }

}

void consumer(void)

{

    int item;

    while (TRUE) {      /* ciclo infinito */

        down(&full);      /* decrementa il contatore full */

        down(&mutex);      /* entra nella regione critica */

        item = remove_item( );      /* prende l'elemento dal buffer */

        up(&mutex);      /* lascia la regione critica */

        up(&empty);       /* incrementa il contatore dei posti vuoti */

        consume_item(item);      /* fa qualcosa con l'elemento */

    }

}
```

## Problema Lettori-Scrittori:

Regola Base: In ogni momento, possono essere ammessi: R lettori e solo 1 scrittore.

Esempio: Si possono avere molteplici letture su un database, ma solo un singolo scrittore.

## Funzionamento Sintetico:

Il primo lettore blocca l'accesso al database, lettori successivi incrementano un contatore e soltanto l'ultimo lettore libera l'accesso al database così gli scrittori possono fare il loro lavoro. N processi accedono (cioè leggono o scrivono) ad alcuni dati condivisi.

Soluzione di base:

```
typedef int semaphore;      /* usate l'immaginazione */

semaphore mutex = 1;        /* controlla l'accesso a rc */

semaphore db = 1;          /* controlla l'accesso al database */

int rc = 0;                /* # di processi che leggono o vogliono leggere */

void reader(void)

{

    while (TRUE) {          /* ripeti per sempre */

        down(&mutex);       /* ottieni accesso esclusivo a rc */

        rc = rc + 1;         /* un lettore in più */

        if (rc == 1) down(&db); /* se questo è il primo lettore ... */

        up(&mutex);         /* rilascia accesso esclusivo a rc */

        read data base( );   /* accedi ai dati */

        down(&mutex);       /* ottieni accesso esclusivo a rc */

        rc = rc - 1;         /* un lettore in meno */

        if (rc == 0) up(&db); /* se questo è l'ultimo lettore ... */

        up(&mutex);         /* rilascia accesso esclusivo a rc */

        use data read( );    /* regione non critica */

    }

}

void writer(void)

{

    while (TRUE) {          /* ripeti per sempre */

        think up data( );    /* regione non critica */

        down(&db);           /* ottieni accesso esclusivo */

        write data base( );   /* aggiorna i dati */

        up(&db);              /* rilascia accesso esclusivo */

    }

}
```

Se nuovi lettori arrivano mentre uno scrittore è in attesa, lo scrittore potrebbe mai ottenere l'accesso, portando a un blocco perpetuo.

Soluzione Proposta: Nuovi lettori vengono posti in coda dietro gli scrittori in attesa, così facendo gli scrittori ottengono accesso dopo i lettori già attivi.

Implicazioni: Questo metodo riduce la concorrenza ma ha un potenziale impatto sulle prestazioni.

Alternative: Esistono soluzioni che danno priorità agli scrittori, ogni strategia ha i suoi vantaggi e svantaggi.

#### Mutua Esclusione: Mutex

Un "mutex" è una versione esplicita e semplificata dei semafori, usata per gestire la mutua esclusione di risorse o codice condiviso, quando non bisogna contare accessi o altri fenomeni. Può essere in due stati: locked (bloccato), unlocked (sbloccato). Un bit basta per rappresentarlo, ma spesso viene usato un intero (0 = unlocked, altri valori = locked).

Due sono le procedure principali: mutex\_lock e mutex\_unlock.

Quando un thread vuole accedere a una regione critica, chiama mutex\_lock. Se il mutex è unlocked, il thread può entrare; se è locked, il thread attende. Al termine dell'accesso, il thread chiama mutex\_unlock per liberare la risorsa. Questo metodo non utilizza il "busy waiting", se un thread non può acquisire un lock chiama thread\_yield per cedere la CPU ad un altro thread.

```
mutex_lock:  
    TSL REGISTER,MUTEX      | copia mutex nel registro e lo imposta a 1  
    CMP REGISTER,#0         | mutex era zero?  
    JZE ok                 | se era zero il mutex era unlocked, quindi ritorna  
    CALL thread_yield      | il mutex è occupato; schedula un altro thread  
    JMP mutex_lock         | prova di nuovo  
ok:     RET                | torna al chiamante; è entrato nella regione critica  
  
mutex_unlock:  
    MOVE MUTEX,#0          | inserisci 0 nel mutex  
    RET                     | torna al chiamante
```

I mutex possono essere implementati nello spazio utente con istruzioni come TSL o XCHG. Alcuni pacchetti di thread offrono mutex\_trylock che tenta di acquisire il lock o restituisce un errore, senza bloccare. I mutex sono efficaci quando i thread operano in uno spazio degli indirizzi comune, la condivisione di memoria tra processi può essere gestita tramite il kernel o con l'aiuto di sistemi operativi che permettono la condivisione di parti dello spazio degli indirizzi.

L'efficienza nella sincronizzazione diventa cruciale con l'aumento del parallelismo. Spin lock e mutex con busy waiting: efficaci per attese brevi, ma sprecano CPU per attese lunghe. Passaggio al kernel per bloccare processi e fare il context switch è oneroso. La soluzione sta nel Futex (Fast User Space Mutex) – combina il meglio di entrambi gli approcci.

## Mutua Esclusione: Futex

Caratteristica di Linux per implementare lock elementari evitando il kernel finché non è necessario, migliorando le prestazioni riducendo il passaggio al kernel ma non è uno standard (serve #include).

E' diviso in due parti: servizio kernel e libreria utente.

Operazione: La variabile condivisa nello spazio utente viene usata come lock e il passaggio al kernel avviene solo quando un thread è bloccato da un altro. Quando un lock è rilasciato, il kernel può essere chiamato per svegliare altri processi in attesa.

Pthread, mutex e varie funzioni:

Pthread: fornisce funzioni per sincronizzare i thread.

Mutex: variabile che può essere locked o unlocked, protegge le regioni critiche.

Funzionamento: Thread tenta di bloccare (lock) un mutex per accedere alla regione critica, se mutex è unlocked, l'accesso è immediato e atomico, se locked, il thread attende.

(pthread\_mutex\_init, \_unlock, ...)

Semafori o Mutex:

Finalità:

Mutex: È utilizzato principalmente per garantire l'esclusione mutua, è destinato a proteggere l'accesso a una risorsa condivisa, garantendo che un solo thread possa accedervi alla volta.

Semaforo: Può essere utilizzato per controllare l'accesso a una risorsa condivisa, ma è anche spesso usato per la sincronizzazione tra thread (vedi esempio produttore/consumatore).

Semantica:

Mutex: Di solito ha una semantica di "proprietà", il che significa che solo il thread che ha acquisito il mutex può rilasciarlo.

Semaforo: Non ha una semantica di proprietà, qualsiasi thread può aumentare o diminuire il conteggio del semaforo, indipendentemente da chi lo ha modificato l'ultima volta.

Casistica:

Per l'esclusione mutua: Un mutex è generalmente preferibile, è più semplice (di solito ha solo operazioni di lock e unlock) e spesso offre una semantica più rigorosa e un comportamento più prevedibile.

Per la sincronizzazione tra thread: Un semaforo può essere più adatto, specialmente quando si tratta di coordinare tra diversi thread o di gestire risorse con un numero limitato di istanze disponibili.

## Mutua Esclusione: Monitor

La comunicazione tra processi usando mutex e semafori non è semplice come potrebbe sembrare. Programmare con semafori richiede estrema attenzione: piccoli errori possono causare comportamenti imprevisti come race conditions o deadlock. Brinch Hansen e Hoare proposero un concetto di sincronizzazione ad alto livello chiamato "monitor" per semplificare la scrittura di programmi. Un monitor raggruppa procedure, variabili e strutture dati. I processi possono chiamare le procedure di un monitor ma non possono accedere direttamente alle sue strutture dati interne. Solo un processo può essere attivo in un monitor in un dato momento, garantendo la mutua esclusione. Il compilatore gestisce la mutua esclusione dei monitor, riducendo la probabilità di errori da parte del programmatore.

Per gestire situazioni in cui i processi devono attendere, i monitor utilizzano variabili condizionali e due operazioni su di esse: wait e signal. A differenza dei semafori, le variabili condizionali non accumulano segnali, se un segnale viene inviato e non c'è un processo in attesa, il segnale viene perso. Linguaggi come Java supportano i monitor (grazie alla JVM che agisce come se fosse un SO), permettendo una sincronizzazione e mutua esclusione più sicura e semplice in contesti multithreading. I metodi sono dichiarati synchronized in modo che solo un thread (in Java la programmazione concorrente è basata su thread perché il processo è uno ed è la JVM) può accedervi. In questo modo non saranno i singoli thread a gestire la mutua esclusione ma sarà il monitor.

```

monitor ProdCons{
    condition full, empty;
    int count=0;
    void enter(int item) {
        if(count==N) wait(full);
        insert_item(item);
        count++;
        if(count==1) signal(empty);
    }
    void remove(int *item) {
        if(count==0) wait(empty);
        *item = remove_item();
        count--;
        if(count==N-1) signal(full);
    }
}

void producer() {
    int item;
    while(TRUE) {
        item = produce_item();
        ProdCons.enter(item);
    }
}

void consumer() {
    int item;
    while(TRUE) {
        ProdCons.remove(&item);
        consume_item(item);
    }
}

```

Differenze tra sleep/wakeup e wait/signal:

sleep/wakeup: sono meccanismi più primitivi utilizzati per mettere un processo/thread in attesa (sleep) e poi svegliarlo (wakeup). Hanno un problema: possono portare a delle race condition, immagina che il processo A voglia svegliare il processo B, se il processo A chiama wakeup per il processo B proprio mentre B sta per chiamare sleep B potrebbe finire per dormire indeterminatamente perché ha perso il segnale di sveglia.

wait/signal (nei monitor): Differenza cruciale: wait e signal sono protetti dalla mutua esclusione all'interno del monitor. Una volta che un thread/processo entra in una procedura del monitor, ha l'esclusività completa di quella procedura fino a quando non termina o chiama wait. In JAVA la procedura ha il modificatore synchronized. Se un thread/processo chiama wait all'interno di un monitor, può essere certo che non verrà interrotto (ad esempio, dallo scheduler) finché non ha terminato di posizionarsi in uno stato di attesa. Questo elimina la possibilità di perdere un segnale come poteva accadere con sleep/wakeup.

I monitor sono costruiti di linguaggio, riconosciuti dal compilatore per garantire la mutua esclusione. Molti linguaggi, come C e Pascal, non hanno monitor o semafori, ma, si possono aggiungere semafori attraverso routine in assembly. I semafori sono pratici per risolvere la mutua esclusione in sistemi con memoria condivisa, ma non in sistemi distribuiti.

Conclusione: I semafori sono a basso livello; i monitor sono limitati ai linguaggi che li supportano.

Mutua Esclusione: Scambi di messaggi

Metodo di comunicazione tra processi usando due primitive: send e receive, può essere

utilizzato in diversi scenari, compresi sistemi distribuiti.

Problemi: i messaggi in rete possono essere persi quindi c'è necessità di acknowledgment per confermare la ricezione. C'è una gestione dei messaggi duplicati usando numeri sequenziali e un'autenticazione e denominazione dei processi.

Malgrado l'inaffidabilità, lo scambio di messaggi è cruciale nello studio delle reti.

#### Problema Produttore-Consumatore:

Soluzione senza memoria condivisa usando solo messaggi, utilizza un totale di N messaggi, simili ai N posti del buffer nella memoria condivisa. Il consumatore invia al produttore N messaggi vuoti, il produttore prende un messaggio vuoto, lo riempie e lo invia.

Numero totale di messaggi rimane costante, gestito dal sistema operativo. Questa soluzione garantisce efficienza e memoria predeterminata.

```
#define N 100           /* numero di posti nel buffer */

void producer(void)

{
    int item;

    message m;          /* buffer del messaggio */

    while (TRUE) {

        item = produce_item(); /* genera qualcosa da mettere nel buffer */

        receive(consumer, &m); /* aspetta che ne arrivi uno vuoto */

        build_message(&m, item); /* costruisce un messaggio da spedire */

        send(consumer, &m);     /* invia l'elemento al consumatore */

    }
}

void consumer(void)

{
    int item, i;

    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* invia N vuoti */

    while (TRUE) {

        receive(producer, &m); /* prende un messaggio che contiene un elemento */

        item = extract_item(&m); /* estrae l'elemento dal messaggio */

        send(producer, &m);     /* ne manda indietro uno vuoto */

        consume_item(item);    /* fa qualcosa con l'elemento */

    }
}
```

#### Problematiche:

Dinamica Produttore-Consumatore: Se il produttore è più veloce, tutti i messaggi saranno pieni, costringendo il produttore ad attendere. Se il consumatore è più veloce, tutti i

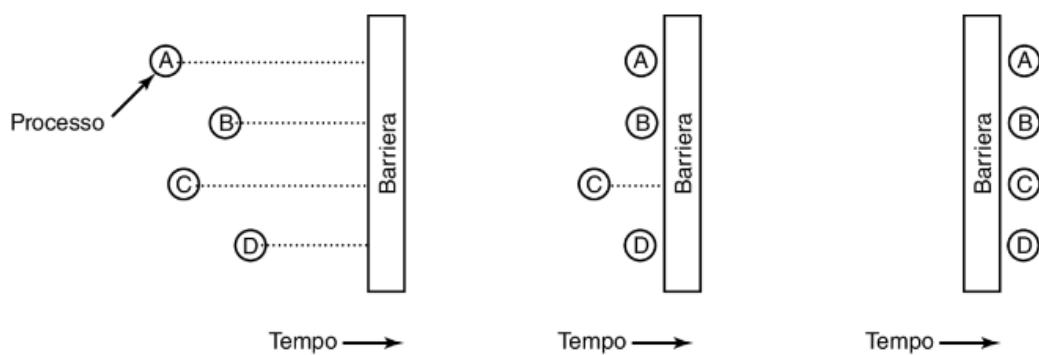
messaggi saranno vuoti, e il consumatore attende un messaggio pieno.

Indirizzamento dei Messaggi: Ogni processo può avere un indirizzo univoco. Introduzione di "mailbox" come buffer per i messaggi, send e receive fanno riferimento alle mailbox, non ai processi.

Lo scambio di messaggi è comunemente usato nei sistemi di programmazione parallela. Un sistema di scambio di messaggi ben conosciuto, per esempio, è MPI (message passing interface), usato diffusamente per elaborazioni scientifiche.

### Sincronizzazione: Barriere

Le barriere sono utilizzate per sincronizzare processi in fasi diverse, quando un processo raggiunge una barriera, attende fino a quando tutti gli altri processi la raggiungono (calcoli paralleli su matrici).



### Inversione delle priorità read-copy-update:

Si tratta di una situazione in cui un processo con priorità più bassa interferisce con l'esecuzione di un processo con priorità più alta. Nella sincronizzazione dei processi, l'inversione delle priorità può verificarsi quando un processo ad alta priorità attende il rilascio di una risorsa detenuta da un processo a priorità più bassa. In questa situazione, il processo ad alta priorità deve attendere che il processo a bassa priorità rilasci la risorsa.

### Possibili soluzioni:

Disattivazione degli interrupt: Semplice ma rischioso, gli interrupt potrebbero non essere riattivati.

Priority Ceiling (Limite della Priorità): Assegnare una priorità al mutex, la priorità viene assegnata al processo che detiene il mutex, fintanto che nessun processo che deve acquisire il mutex ha una priorità superiore al limite, l'inversione non è più possibile. (viene messo un limite basso di priorità così che i processi con una priorità più alta non potranno accedere al mutex e lasceranno il mutex a processi con più bassa priorità).

Priority Inheritance (Ereditarietà della Priorità): Task a bassa priorità che detiene il mutex eredita temporaneamente la priorità del task ad alta priorità.

Random Boosting (Potenziamento Casuale): Aumenta la priorità di thread casuali che detengono un mutex.

### Read-Copy-Update

I lock più rapidi sono quelli che non si verificano mai. E se non ci sono lock non si rischia nemmeno l'inversione delle priorità. L'obiettivo è quindi avere accessi concorrenti senza lock. Non possiamo però imporre accessi in lettura e scrittura concorrenti senza lock perché

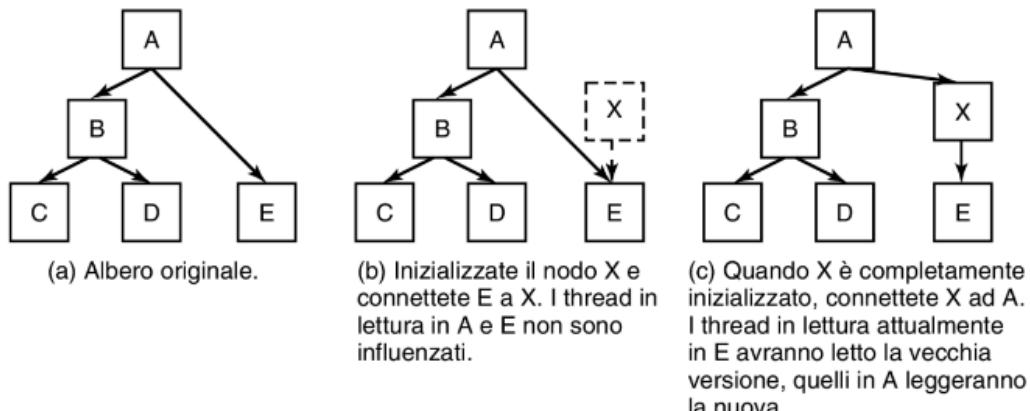
ci potrebbe essere un'inconsistenza dei dati (es. calcolo della media mentre si riordina un array).

**Principio Base di Read-Copy-Update:** Aggiornare strutture dati consentendo letture simultanee senza incappare in versioni inconsistenti dei dati, i lettori vedono: o la versione vecchia o la nuova, mai un mix delle due.

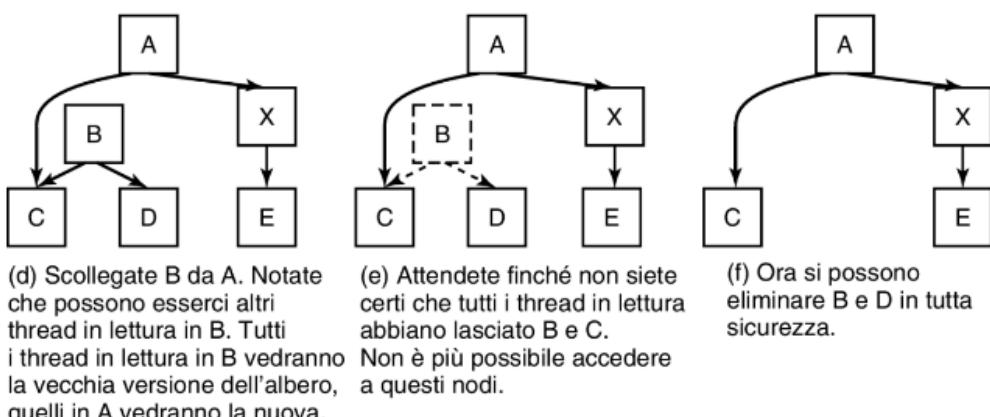
**Inserimento:** Nodo X preparato e reso visibile in modo atomico, nessuna versione non coerente letta.

**Rimozione:** Nodo B e D eliminati senza bisogno di lock, i Lettori vedono o la nuova o la vecchia struttura, mai entrambe.

#### Aggiunta di un nodo:



#### Rimozione di nodi:



#### Problema e Soluzione:

Quando liberare B e D? Finché ci sono lettori, non si possono liberare.

**Operazione RCU:** determina il tempo massimo per trattenere un riferimento.

**Grace Period:** tempo in cui ogni thread esce almeno una volta dalla sezione critica. Aspetta un periodo (grace period) prima di liberare la memoria. I thread nella sezione critica non si bloccano né vanno in sleep, quindi si aspetta un cambio di contesto.

#### Scheduler:

Non esiste uno scheduler ottimo, ma una politica di scheduling ottima per quello che si vuole fare. In un computer multiprogrammato, molteplici processi/thread possono competere per la CPU contemporaneamente, lo scheduler decide quale processo/thread eseguire successivamente seguendo un algoritmo di scheduling. Molti problemi di scheduling per

processi valgono anche per i thread. Lo scheduling al livello del kernel avviene per i thread indipendentemente dal processo di appartenenza.

Nei sistemi batch storici, lo scheduling era lineare: si eseguiva il lavoro successivo sul nastro.

Con la multiprogrammazione, lo scheduling è diventato complesso a causa della concorrenza tra utenti. Gli algoritmi di scheduling sono cruciali per la prestazione e la soddisfazione dell'utente nei mainframe.

Nei personal computer: Spesso un solo processo è attivo, la CPU raramente è una risorsa scarsa: la maggior parte dei programmi è limitata dalla velocità dell'input dell'utente, essendoci tanti tempi morti in cui l'utente non fa nulla la CPU non è mai utilizzata al massimo.

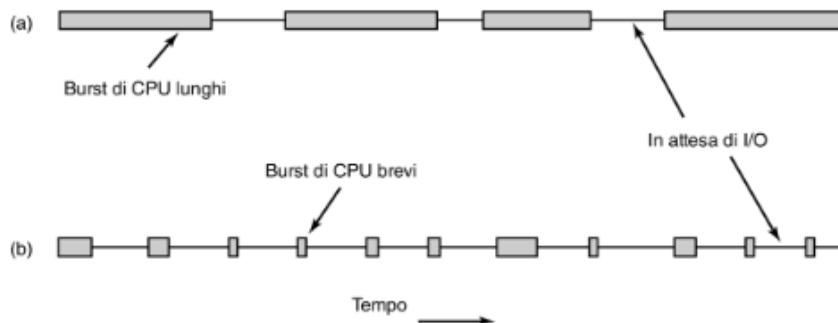
Lo scheduling costa in termini di tempo in quanto lo scambio di processi (o "context switch") è oneroso: Cambio da modalità utente a modalità kernel, Salvataggio dello stato del processo, Esecuzione dell'algoritmo di scheduling, Cambio della mappa della memoria, Invalidazione potenziale della memoria cache.

Troppe commutazioni possono consumare tempo di CPU.

I processi alternano fasi di elaborazione CPU-intense con richieste di I/O.

a) Compute-bound (CPU-bound): Burst di CPU lunghi, attese di I/O infrequenti.

b) I/O-bound: Burst di CPU brevi, attese di I/O frequenti. Sono tali a causa della bassa necessità di calcoli, non della durata delle richieste di I/O.



Con CPU più veloci, i processi tendono a essere più I/O-bound. CPU e dischi magnetici non stanno avanzando rapidamente in velocità. Gli SSD sostituiscono gli HDD nei PC, ma i data center utilizzano ancora HDD per il costo. Lo scheduling varia in base al contesto: ciò che funziona per un dispositivo potrebbe non essere efficace per un altro.

Se ci sono più processi pronti che CPU disponibili lo scheduler decide quale processo eseguire successivamente. L'algoritmo utilizzato dallo scheduler è chiamato algoritmo di scheduling.

Situazioni in cui è necessario lo scheduling:

Creazione Nuovo Processo: Decisione tra l'esecuzione del processo genitore o figlio, dato che sono entrambi pronti può essere scelto chiunque.

Uscita di un Processo: Se un processo esce, occorre scegliere un altro dai processi pronti. Se nessuno è pronto, occorre eseguire un processo inattivo del sistema.

Blocco del Processo: Se un processo si blocca (I/O, semaforo, etc.), occorre selezionarne un altro, a volte la causa del blocco può influire sulla decisione.

**Interrupt di I/O:** Alla conclusione di un I/O, un processo potrebbe diventare pronto, bisogna decidere se eseguire il processo appena pronto, il precedente o un altro.

**Tipi di Scheduling e Prelazione:**

**Non Preemptive (Senza Prelazione):** Seleziona un processo e lo lascia eseguire fino al blocco o al rilascio volontario, nessuna decisione durante gli interrupt del clock. Ripristina il processo precedente dopo l'interrupt, a meno che non ci sia una priorità superiore.

**Preemptive (Con Prelazione):** Sceglie un processo e lo lascia eseguire per un tempo massimo definito, se ancora in esecuzione dopo il tempo, è sospeso e viene scelto un altro, sospeso da un interrupt del clock per restituire controllo allo scheduler.

**Importanza della Prelazione:** Rilevante per le applicazioni e i kernel dei sistemi operativi. Necessaria per prevenire che un driver o una chiamata di sistema lenti blocchino la CPU. In un kernel con prelazione, lo scheduler può forzare un cambio di contesto.

**Diversi ambienti di scheduling:**

**Batch:** Ideale per attività aziendali periodiche, accetta algoritmi senza prelazione e ha priorità a prestazioni efficienti (un lavoro dopo l'altro).

**Interattivo:** Prelazione fondamentale, previene la monopolizzazione della CPU ed è adatto per server e utenti multipli (prevede l'interazione di un utente).

**Real-time:** Processi spesso si bloccano velocemente sapendo di non poter eseguire a lungo (hanno un tempo prefissato), la Prelazione non sempre necessaria e si eseguono programmi per specifiche applicazioni, a differenza dei sistemi interattivi che possono eseguire programmi arbitrari.

**Obiettivi generali degli algoritmi di scheduling:**

• **Sistemi Batch:**

Throughput: Massimizzare il numero di job completati in un tempo fissato.

Tempo di Turnaround: Minimizzare il tempo dallo start all'end di un job.

Utilizzo della CPU: Mantenere la CPU costantemente attiva.

• **Sistemi Interattivi:**

Tempo di risposta: Ottimizzare il tempo di risposta rapida alle richieste degli utenti.

Adeguatezza: Soddisfare le aspettative dell'utente in termini di tempi di risposta.

• **Sistemi Real-time:**

Rispetto delle scadenze: Assicurarsi che i dati vengano elaborati nei tempi previsti.

Prevedibilità: Assicurarsi che il funzionamento sia costante, specialmente in sistemi multimediali per evitare degradi della qualità.

• **Tutti i sistemi:**

Equità: Garantire un'equa condivisione (non in termini di 1/n) della CPU a tutti i processi.

Imposizione della policy: Garantire l'attuazione delle policy dichiarate.

Bilanciamento: Mantenere tutti i componenti del sistema attivi.

L'equità è fondamentale in ogni scenario, in un sistema batch, è ideale combinare processi CPU-bound e I/O-bound e nei sistemi real-time è cruciale rispettare le scadenze e garantire la prevedibilità.

**Scheduling nei sistemi batch:**

**First-Come First-Served:**

**Descrizione:** Algoritmo di scheduling senza prelazione, i processi vengono assegnati alla CPU nell'ordine in cui arrivano, quindi è presente una singola coda di processi in stato

pronto e il primo job viene eseguito immediatamente senza interruzioni, i processi bloccati ritornano in fondo alla coda.

Vantaggi: Facile da capire e programmare (Gestione semplice con una linked list), ed è equo in base all'ordine di arrivo.

Svantaggi: Prestazioni non ottimali in scenari misti (es. processi CPU-bound e I/O-bound).

Può risultare in tempi di attesa molto lunghi per processi I/O-bound in presenza di un processo CPU-bound.

Esempio:

Un processo CPU-bound esegue per 1 secondo e successivamente molti processi I/O-bound leggono dal disco, usando FCFS, i processi I/O-bound potrebbero impiegare 1000 secondi per terminare, invece che 10 secondi con un algoritmo di scheduling con prelazione.

Shortest Job First:

Descrizione: Algoritmo batch senza prelazione, richiede che i tempi di esecuzione siano noti in anticipo. Il job più breve viene eseguito per primo.

Esempio: 4 job (A, B, C, D) con tempi di 8, 4, 4 e 4 min.

Esecuzione in ordine: 8 min per A, 12 min per B, 16 min per C, 20 min per D (media 14 min).

Esecuzione con SJF: 4 min, 8 min, 12 min e 20 min (media 11 min).

Ottimalità: Shortest Job First è ottimale nel minimizzare il tempo di turnaround medio quando tutti i job sono disponibili contemporaneamente.

Limitazione: Se i job arrivano in momenti diversi, SJF potrebbe non essere ottimale.

Es. Job A-E con tempi 2, 4, 1, 1, 1 e arrivi a 0, 0, 3, 3, 3. Due sequenze diverse producono medie di 4,6 e 4,4.

Shortest Remaining Time Next:

Descrizione: Versione con prelazione di SJF, seleziona sempre il processo con il tempo rimanente più breve per completare. Il tempo di esecuzione deve essere noto in anticipo.

Funzionamento: Confronta il tempo totale del nuovo job con il tempo rimanente dei processi in esecuzione, se il nuovo job è più breve del processo corrente, sospende il processo corrente ed esegue il nuovo job. Assicura che i nuovi job brevi ricevano un servizio rapido.

Scheduling in sistemi interattivi:

Il tempo di risposta è fondamentale rispondendo rapidamente alle richieste.

Proporzionalità: occorre soddisfare le aspettative degli utenti.

Round-Robin Scheduling:

Concetto: Uno degli algoritmi di scheduling più vecchi, semplici, equi e ampiamente utilizzati. Ogni processo riceve un intervallo di tempo o "quanto" per l'esecuzione, se il processo non ha terminato al termine del quanto, la CPU viene prelazionata per un altro processo, se un processo termina o si blocca prima del quanto, il passaggio avviene automaticamente.

Implementazione: Mantenere una lista dei processi eseguibili, una volta esaurito il quanto, il processo viene spostato alla fine della lista.

Durata del Quanto: La scelta del quanto influenza sull'efficienza, supponendo 1 ms per il cambio di contesto e 4 ms per il quanto: il 20% del tempo CPU sprecato in overhead.

Trade-off:

Quanto lungo: riduce l'overhead, ma peggiora la reattività (es. 5 secondi di attesa per un breve comando in un server affollato).

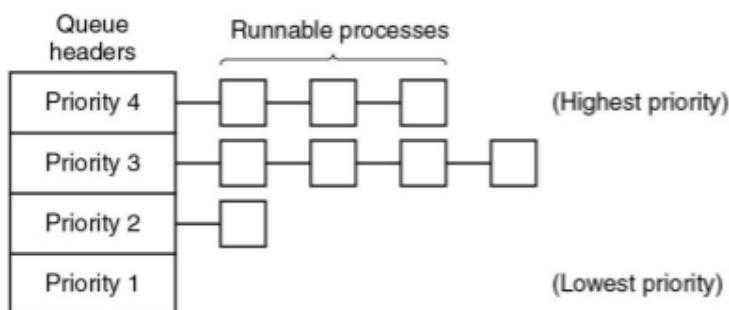
Quanto breve: maggiore overhead e riduzione dell'efficienza della CPU.

Ottimizzazione: Se il quanto è maggiore del tempo medio di burst di CPU, la prelazione potrebbe non avvenire spesso. Molti processi potrebbero bloccarsi prima.

Compromesso: un quanto tra 20 e 50 ms è spesso ragionevole per bilanciare efficienza e reattività.

Scheduling a priorità:

Premessa: Round-robin considera tutti i processi ugualmente importanti, ma alcuni contesti richiedono una gerarchia. Si ha il bisogno di scheduling a Priorità: Ogni processo ha una priorità assegnata, la CPU esegue il processo con la priorità più alta tra quelli pronti, è applicabile anche su singoli PC: ad es. un daemon (processo di utente artificiale) di posta elettronica avrebbe meno priorità di un video in tempo reale.



Gestione delle Priorità: Priorità del processo attualmente in esecuzione può diminuire con il tempo, se scende sotto quella del processo successivo, avviene un cambio. C'è quindi la possibilità di assegnare un quanto di tempo: al suo esaurirsi, si passa al processo con priorità appena inferiore, bisogna però evitare che processi rimangano inibiti indefinitamente, altrimenti potrebbero finire a priorità 0.

Priorità Statica vs Dinamica:

Statica: es. gerarchie militari (processi con più importanza) o basate sui costi nel data center (più pago più ho priorità).

Dinamica: es. basata sull'utilizzo della CPU o sul comportamento I/O bound.

Raggruppamento in Classi: I processi divisi in classi di priorità, c'è uno scheduling a priorità tra le classi e uno all'interno della stessa classe (in questo caso round-robin).

Esempio: Sistema con 4 classi di priorità., fintanto ci sono processi in priorità 4, si usano in round-robin, se vuota, si passa alla 3, poi alla 2, e così via.

Importante rivedere periodicamente le priorità per evitare che processi a bassa priorità non vengano mai eseguiti.

Shortest Process Next:

"Shortest Job First" ottimizza il tempo medio di risposta nei sistemi batch, l'obiettivo è applicarlo ai sistemi interattivi. Per identificare quale tra i processi eseguibili sia effettivamente il più breve si utilizza l'Aging: Fare stime basate sul comportamento passato dei processi.

Prima si stima del tempo per un comando: T0, poi la stima viene aggiornata dopo nuova esecuzione T1 diventa  $aT_0 + (1 - a)T_1$ , e così via.

La scelta di 'a' determina il peso delle esecuzioni precedenti nella nuova stima.

Esempio con  $a = 1/2$ :  $T_0, T_0/2 + T_1/2, T_0/4 + T_1/4 + T_2/2, \dots$  Dopo 3 esecuzioni, il peso di

T0 nella stima è 1/8.

Utilizzo dell'Aging: Prevista in molte situazioni dove si basa la previsione su valori passati, si dà molto più peso agli ultimi processi.

#### Guaranteed Scheduling:

Concetto Principale: Garantire equamente a tutti i processi l'accesso alle risorse (policy).

Promessa Base: Se ci sono n utenti o processi, ciascuno ottiene  $\sim 1/n$  della potenza della CPU.

Come Funziona: Il sistema tiene traccia di quanta CPU ha ricevuto ogni processo dal momento della sua creazione (esempio 100 secondi), calcola quanto tempo CPU ogni processo dovrebbe avere tempo da creazione /n (ad esempio 100 sec/ 10 processi, dovrebbe avere 10 secondi), poi valuta il rapporto tra il tempo CPU consumato e quello dovuto. Se il rapporto di 0,5: ha avuto metà di quanto dovuto, se il rapporto di 2,0: ha avuto il doppio di quanto dovuto. Esegue quindi il processo con il rapporto più basso finché non supera il suo concorrente più vicino.

#### Scheduling a lotteria:

Concetto di base: Assegnazione di biglietti della lotteria ai processi per le risorse del sistema, successivamente si fa un estrazione casuale di un biglietto per decidere quale processo ottiene la risorsa.

Distribuzione delle priorità: possono essere assegnati biglietti extra per processi più importanti avendo quindi maggiori probabilità di vincere.

Esempio: Se un processo ha il 20% dei biglietti, guadagnerà a lungo termine il 20% della CPU.

Reattività: Risponde velocemente ai nuovi processi grazie alla distribuzione dei biglietti.

Cooperazione tra processi: Possibilità di scambiarsi biglietti tra processi cooperanti.

Soluzione a problemi complessi: Adatto a situazioni dove altri metodi falliscono. Esempio: Server video con diverse necessità di frequenze di fotogrammi, assegnazione di biglietti in base alla velocità necessaria, così da avere una divisione automatica della CPU nelle proporzioni corrette.

#### Scheduling Fair-Share:

Premessa: Ogni processo è oggetto di scheduling in base all'utente che lo ha creato. Es. Se l'utente 1 ha 9 processi e l'utente 2 ne ha 1, con round-robin o priorità uguali, l'utente 1 avrà il 90% della CPU, l'utente 2 solo il 10%.

Approccio Fair-Share: Considera la proprietà di ogni processo prima di considerarlo, ogni utente riceve una frazione predefinita di CPU, lo scheduler si assicura che ogni utente riceva la sua frazione, indipendentemente dal numero di processi posseduti.

Esempio: Due utenti, ciascuno con il 50% della CPU. Utente 1 ha processi A, B, C, D;

Utente 2 ha processo E.

Sequenza con round-robin: A E B E C E D E A E...

Se l'utente 1 ha il doppio del tempo di CPU rispetto all'utente 2: A B E C D E A B E...

Versatilità: Molti modi di implementare, basati sulla definizione di "equità".

#### Scheduling Sistemi Real Time:

Usato nei sistemi operativi in applicazioni in cui il tempo di risposta è fondamentale come lettori cd, monitoraggio in terapia intensiva, piloti automatici, controllo robotico in fabbriche, missili. Ritardi o mancati tempi di risposta possono avere gravi implicazioni. Due categorie:

Hard Real-Time: Scadenze assolute da rispettare.

Soft Real-Time: Qualche scadenza mancata è tollerabile.

Comportamento: Processi prevedibili, brevi e noti in anticipo, ogni processo porta con sé due informazioni: il periodo di attivazione e il tempo di esecuzione.

Tipi di eventi:

Periodici: Avvengono a intervalli regolari.

Non Periodici: Avvengono in modo imprevedibile.

Condizione di «Schedulabilità»: La CPU deve essere in grado di gestire la somma totale del tempo richiesto dai processi.

Per esempio, se ci sono  $m$  eventi periodici, l'evento  $i$  avviene con un periodo  $P_i$  e richiede  $C_i$  secondi di tempo della CPU per gestire ogni evento, allora il carico può essere gestito solo se

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

Esempio: Eventi periodici: 100ms, 200ms, 500ms. Tempi richiesti: 50ms, 30ms, 100ms.

Condizione:  $0,5+0,15+0,2 < 1$ .

Gli algoritmi di Scheduling possono essere:

Statici: Decisioni prese prima dell'esecuzione.

Dinamici: Decisioni prese durante l'esecuzione.

Limitazioni: Lo scheduling statico richiede una perfetta conoscenza delle esigenze e delle scadenze.

Processi e scheduling:

Premessa: Abbiamo sempre considerato i processi come appartenenti a utenti differenti in competizione per la CPU, nel mondo reale un processo può avere molti processi figli sotto il suo controllo.

Problematica: Gli scheduler tradizionali non accettano input dai processi utente, spesso portando a decisioni sub-ottimali.

I processi vengono eseguiti allo spazio utente e i processi del sistema operativo cercano di essere protetti rispetto alle richieste dell'utente, così da non avere una priorità minore di quelli dell'utente. Se un processo crea un clone come fa ad assegnare una priorità? Nasce quindi una separazione tra le politiche e il meccanismo di scheduling.

Separazione tra meccanismo e politica di scheduling:

Vantaggio: L'algoritmo di scheduling può essere parametrizzato, ma i parametri sono forniti dai processi utente.

Esempio pratico: Kernel con algoritmo di scheduling a priorità, la chiamata di sistema permette a un processo di impostare le priorità dei suoi figli, il genitore può influenzare lo scheduling dei suoi figli senza controllarlo direttamente.

Conclusione: Il meccanismo sta nel kernel, la policy è determinata dal processo utente.

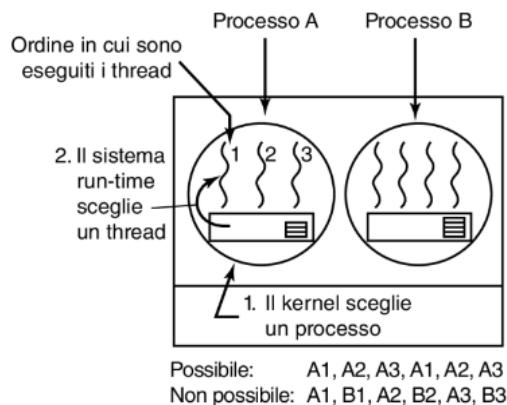
Due livelli di parallelismo: processi e thread.

Lo scheduling differisce in base al tipo di thread: livello utente vs. livello kernel.

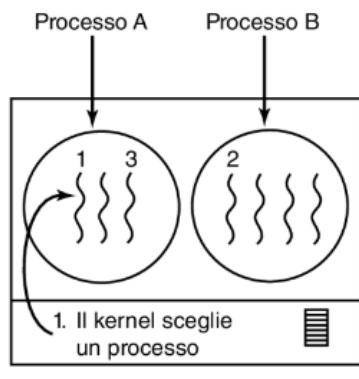
Thread a livello utente: Il kernel ignora l'esistenza dei thread; sceglie un processo per il suo quanto, il thread interno decide quale thread eseguire senza interruzione del clock.

Risultato: Un thread può consumare l'intero quanto del processo, influenzando solo il processo interno e non gli altri.

Possibile ordine di esecuzione: A1, A2, A3, A1, A2... Scheduling del sistema run-time può variare: spesso round-robin o a priorità. C'è una cooperazione tra thread; nessuna interruzione forzata.



Thread del kernel: Il kernel seleziona un thread specifico per l'esecuzione, se un thread eccede il quanto, viene sospeso.



Thread a livello utente vs. Thread del kernel:

Scambio thread utente: poche istruzioni.

Scambio thread kernel: scambio completo di contesto (context switch) - più lento.

Blocco su I/O: con thread utente, intero processo sospeso; con thread kernel, solo il thread specifico.

Decisioni del kernel: Considera i costi per passare da un thread a un altro e può dare preferenza ai thread dello stesso processo.

Scheduling specifico dell'applicazione: Permette maggiore controllo e ottimizzazione dell'applicazione rispetto allo scheduling del kernel.

## GESTIONE DELLA MEMORIA

I programmi diventano sempre più complessi e offrono sempre più funzioni occupando sempre più memoria.

Memoria Principale (RAM): I processi crescono rapidamente quindi l'ideale è avere una memoria privata (un processo non può andare a leggere es), grande, veloce, persistente e a basso costo, ma la realtà tecnologica è diversa.

**Gerarchia della Memoria:** Concetto sviluppato nel tempo, da memoria veloce, piccola e costosa a memoria lenta, economica e di grandi dimensioni. Es. Da pochi MB di memoria veloce a TB di memoria lenta e dispositivi di archiviazione come USB.

Il Sistema Operativo fornisce un'astrazione di questa gerarchia e si occupa della gestione di questa.

Il Gestore della Memoria gestisce la memoria e la sua gerarchia, traccia l'uso della memoria, alloca e libera memoria per i processi, controlla che un processo non vada a leggere o scrivere in memoria riservata ad altri.

### Astrazione della Memoria

#### Memoria Senza Astrazione:

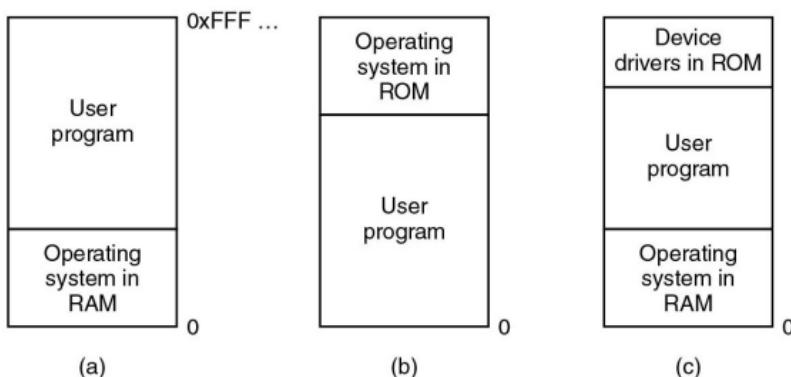
Modello più semplice: abbiamo un uso diretto dalla memoria fisica! Nei primi computer mainframe, minicomputer e PC, non esisteva astrazione della memoria e ogni programma vedeva solo la memoria fisica.

Ad esempio, con l'istruzione MOV REGISTER1,1000 la locazione fisica di memoria 1000 veniva trasferita in REGISTER1. Un programma poteva interferire con un altro, causando crash. Esempio/Rischio: un applicativo utente può cancellare il sistema operativo!!!

In queste condizioni non era possibile avere più di due programmi eseguiti in memoria nello stesso istante, il primo programma scriveva un valore il secondo sovrascriveva.

Si utilizzavano tre modelli principali di organizzazione della memoria:

- OS in RAM utilizzato sui mainframe e sui minicomputer (desueto)
- OS in ROM Es: sistemi embedded
- OS+drivers in ROM+RAM primi personal computer (es. MS-DOS: la ROM era chiamata Basic Input Output System - BIOS).

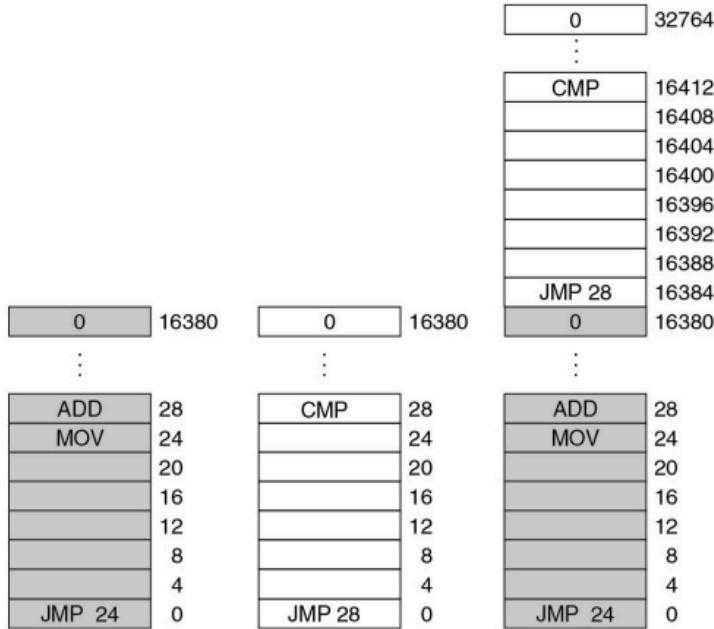


Esisteva la possibilità di eseguire più programmi contemporaneamente senza astrazione della memoria usando il "swapping".

**Swapping:** salvataggio del contenuto della memoria in un file su memoria non volatile e prelievo del programma successivo.

**Naive Approach:** Caricamento di più programmi in memoria fisica consecutivamente, senza astrazione dell'indirizzo.

- Un programma di 16 KB che inizia con l'istruzione JMP 24.
- Un altro programma di 16 KB che inizia con l'istruzione JMP 28.
- Entrambi i programmi caricati consecutivamente.



Quando il secondo programma viene eseguito, JMP 28 indirizza erroneamente all'istruzione del primo programma, causando errori.

**Problema Principale:** I programmi utilizzano indirizzi assoluti di memoria fisica, portando a conflitti durante l'esecuzione, la mancanza di astrazione dell'indirizzo può causare il crash dei programmi.

**Problema:** L'accesso diretto alla memoria fisica da parte dei programmi può causare problemi come la distruzione del sistema operativo e la difficoltà di esecuzione simultanea di più programmi.

La soluzione è astrarre la memoria per separare e proteggere i programmi in esecuzione. Viene introdotto il concetto di Spazio degli Indirizzi: Ogni programma ha un insieme unico di indirizzi (spazio degli indirizzi) che può usare per indirizzare la memoria, questo spazio è indipendente da altri processi e rappresenta una forma di memoria astratta.

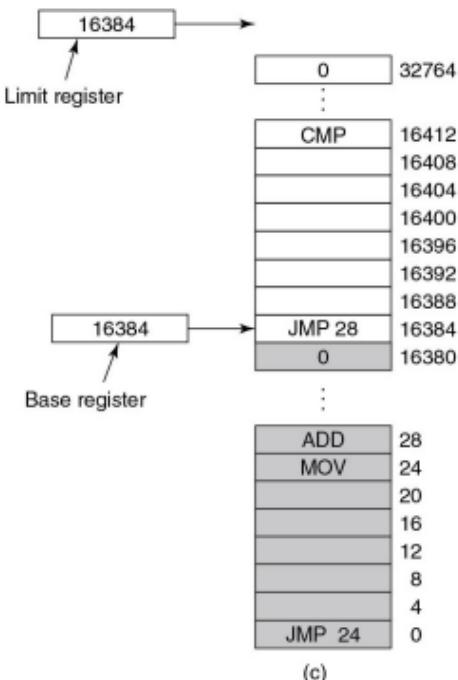
**Vecchia soluzione:** mappare lo spazio degli indirizzi di ogni processo in parti diverse della memoria fisica. Si hanno Registre Base e Limite: Due registri hardware speciali presenti in molte CPU.

**Registro Base:** contiene l'indirizzo fisico di inizio di un programma in memoria.

**Registro Limite:** contiene la lunghezza del programma.

Gli indirizzi generati dai programmi vengono aggiustati automaticamente aggiungendo il valore del registro base.

Il registro di base mette in atto la rilocazione dinamica, il registro limite applica la protezione.



(c)

Ogni riferimento alla memoria da parte di un programma aggiunge il valore del registro base all'indirizzo generato e confronta con il registro limite per assicurare che l'accesso sia entro i limiti consentiti.

Vantaggi: Offre a ogni processo uno spazio degli indirizzi separato e protetto.

Svantaggi: Necessità di eseguire somme e confronti ad ogni accesso alla memoria, il che può essere lento.

Un computer medio al suo avvio può avere 50-100 processi o più.

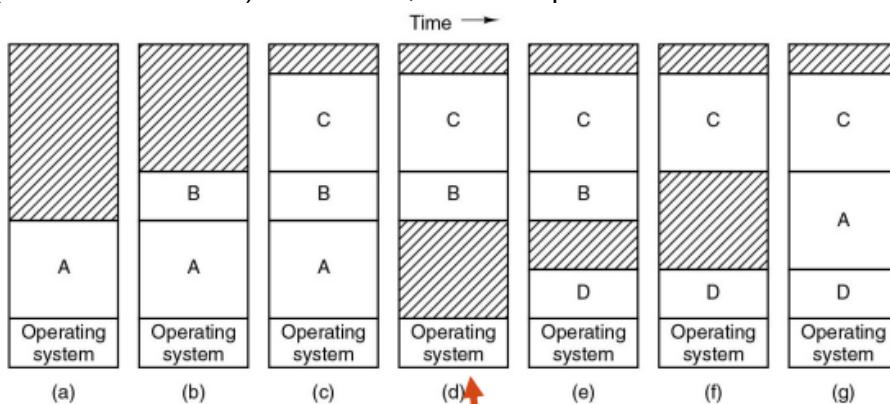
Le Applicazioni possono richiedere anche 1 GB solo per avviarsi, quindi la memoria fisica totale necessaria è spesso superiore a quella disponibile.

Si usano diverse strategie per gestire il sovraccarico di memoria:

1. Swapping (Scambio) dei processi: Sposta interi processi tra la memoria RAM e la memoria non volatile (disco) e i processi inattivi archiviati su memoria non volatile
2. Memoria Virtuale: Permette l'esecuzione dei programmi anche se solo parzialmente presenti nella memoria principale (pagine).

Swapping:

Lo scambio può portare alla frammentazione della memoria, è necessaria ricompattare (deframmentazione) la memoria, ma è un'operazione estremamente lenta.



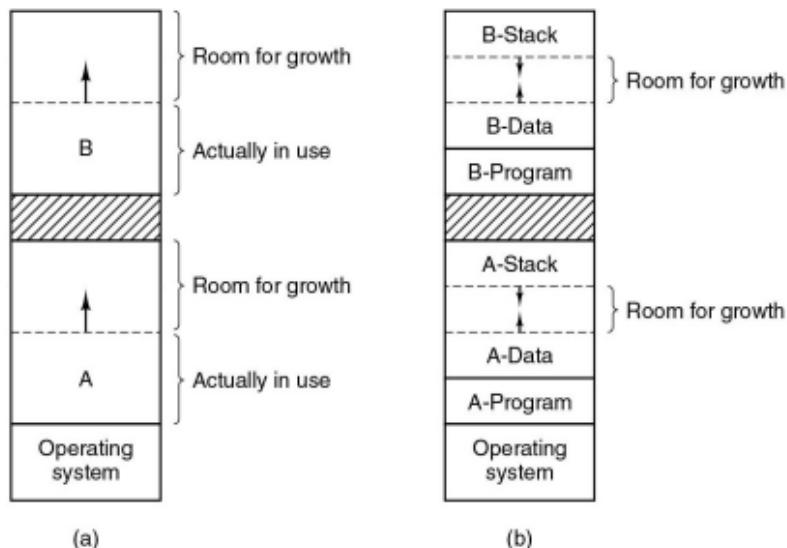
Gestione dello spazio e crescita dei processi:

Bisogna gestire processi con segmenti di dati in crescita, possiamo utilizzare la Memory Compaction che sposta processi per liberare spazio, ma richiede tempo.

Soluzione: allocare memoria extra durante lo swapping o durante lo spostamento dei processi. In caso un processo andrà Out of Memory si può ucciderlo, trasferirlo o fare lo Swapping.

(a) Spazio allocato per segmento dati in crescita.

(b) Spazio per stack e segmento dati che crescono.



Gestione della memoria libera

Gestione dinamica della memoria:

L'obiettivo è tenere traccia dell'utilizzo della memoria, ad esempio ogni blocco di 4 bytes.

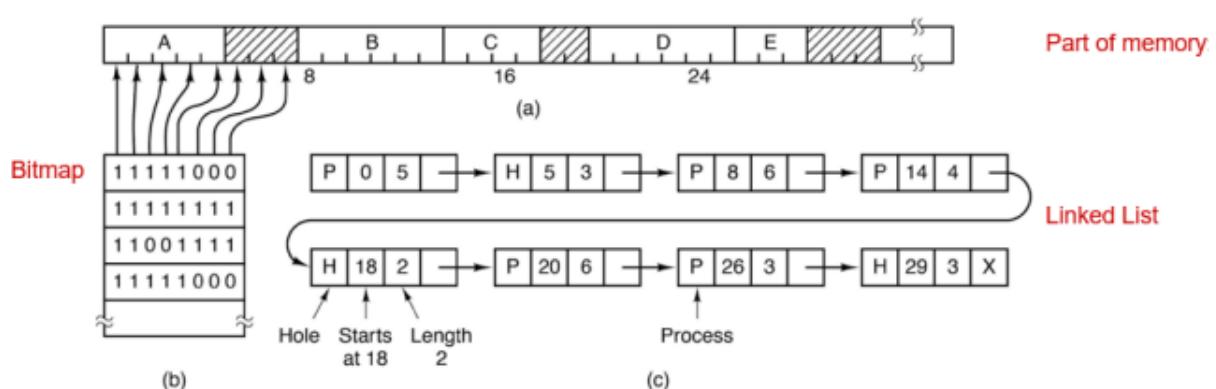
Per farlo utilizziamo o una Bitmap che tiene traccia di quali blocchi vengono allocati o una lista collegata che tiene traccia della memoria non allocata.

Importanza: Questo tracciamento non riguarda solo la memoria, ma anche risorse come i file system.

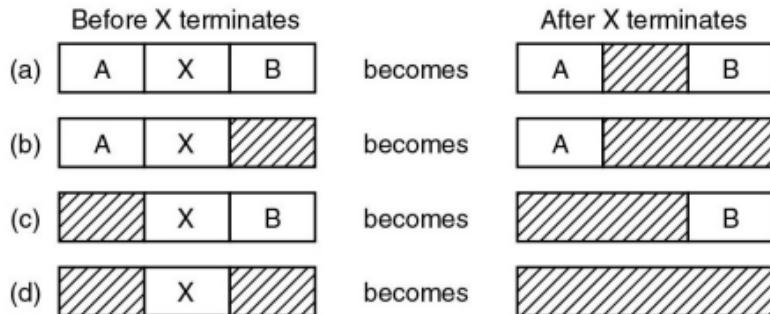
Bitmap vs LinkedList

Bitmap: trovare i fori richiede una scansione (lenta).

Lista: liste di processi/«buchi»: Allocazione lenta contro deallocazione lenta, buchi ordinati per indirizzo per una rapida coalescenza (come fusione tra gocce).



Nella pratica viene spesso usata una doppia linked list, rende più facile gestire lo spazio libero, può controllare facilmente se il precedente spazio è libero e può regolare facilmente i puntatori.



Sistemi di allocazione della memoria:

First Fit: Seleziona il primo spazio disponibile. Opzione più semplice (MINIX3).

Next Fit: Seleziona il successivo spazio disponibile. Più lento del First Fit in pratica.

Best Fit: Sceglie lo spazio più adeguato. Tende alla frammentazione (lascia piccoli spazietti).

Worst Fit: Sceglie lo spazio meno adeguato (così da lasciare più spazio a disposizione). Ha prestazioni scadenti in pratica.

Quick Fit: Mantiene liste divise per le dimensioni più richieste così da non scorrere tutta la linked list, ha una scarsa performance nella coalescenza.

Buddy Allocation (Linux): Migliora la performance di coalescenza del Quick Fit.

Allocazione memoria in Linux:

Linux utilizza vari meccanismi per l'allocazione della memoria, è un'allocazione delle pagine basata sull'algoritmo di Buddy Memory Allocation.

Funzionamento: La memoria inizia come un singolo pezzo contiguo, ad ogni richiesta, la memoria viene divisa con una potenza di 2, blocchi di memoria contigui vengono uniti quando rilasciati.

Inizialmente la memoria consiste di un singolo pezzo contiguo, per esempio 64 pagine (a). Arriva una richiesta da 8 otto pagine, arrotondata a potenza di 2. L'intero pezzo di memoria viene quindi diviso a metà, come mostrato in (b).

Poiché ciascuno dei due pezzi è ancora troppo grande, il pezzo più in basso viene diviso ancora a metà (c) e poi ancora (d).

Ora abbiamo un pezzo della dimensione corretta, che viene così allocato al chiamante, grigio in (d).

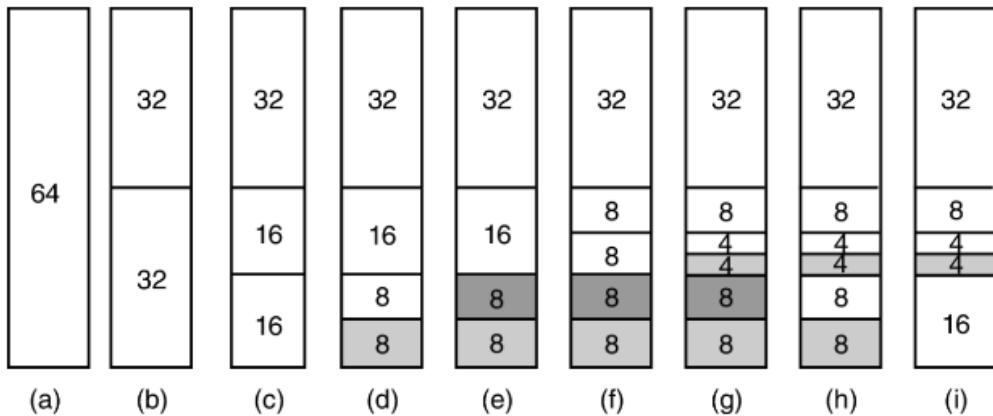
Arriva una seconda richiesta di otto pagine che viene soddisfatta immediatamente (e).

A questo punto arriva una terza richiesta di quattro pagine. Il pezzo disponibile più piccolo viene diviso (f) e ne viene assegnata la metà (g).

Successivamente, il secondo pezzo di otto pagine viene rilasciato (h).

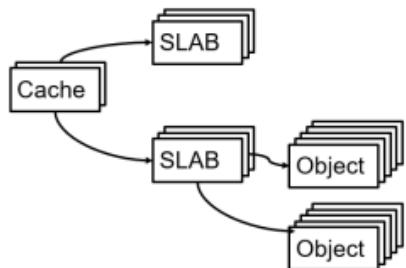
Infine, anche l'altro pezzo di otto pagine viene rilasciato. Poiché i due pezzi di otto pagine adiacenti appena liberati provengono dallo stesso pezzo di 16 pagine, vengono uniti per

riottenere il pezzo di 16 pagine (i).

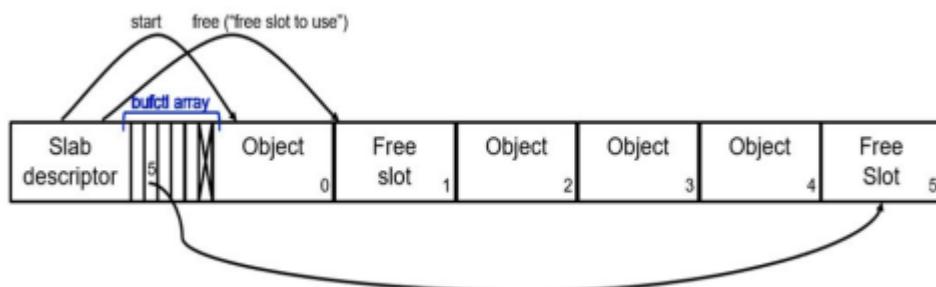


Il Buddy Algorithm può causare frammentazione interna: una richiesta di 65 pagine richiede l'allocazione di 128 pagine. Lo SLAB allocator in Linux risolve questo problema, prendendo blocchi tramite l'algoritmo buddy e ritagliando unità più piccole (slab) per gestirle separatamente.

Come Funziona: Nello slab allocation, la memoria è divisa in blocchi chiamati "slabs" ulteriormente suddivisi in chunk di dimensioni uniformi, e adeguati per ospitare un oggetto di un certo tipo. Un "slab" può essere in uno dei seguenti stati:  
 pieno (tutti i chunk sono utilizzati),  
 parzialmente pieno (alcuni chunk sono liberi).  
 o vuoto (tutti i chunk sono liberi).



Quando un oggetto viene deallocated, non viene immediatamente restituito al sistema come memoria libera, viene mantenuto nella cache in modo che, se viene nuovamente richiesto, possa essere rapidamente riallocata senza l'overhead di inizializzazione.



In questo esempio, uno Slab contiene: puntatore all'inizio della memoria con gli slot degli oggetti, indice del prossimo slot libero, bufctl: array di indici dei prossimi slot liberi.

I vecchi sistemi operativi che utilizzavano 32 bit per ogni processo dedicavano 4GB di memoria, i 64 bit di oggi dedicano potenzialmente 256TB di memoria per ogni processo. Di 64 bit si utilizzano 48 perché nessuno avrà mai necessità di 256TB di memoria massima. In futuro in caso sia necessario si useranno tutti e 64 bit.

### Memoria Virtuale

C'è una necessità di gestire programmi che superano la capacità della memoria disponibile. Negli anni '60 ci fu l'introduzione di tecniche per dividere programmi in parti gestibili.

Overlay: il programma è diviso in piccole parti o segmenti, solo l'overlay necessario viene caricato in memoria, quelli successivi sovrascrivono o coesistono con quelli precedenti avendo un scambio continuo tra memoria e disco.

Originariamente, i programmatore dovevano suddividere manualmente i programmi in overlay, questa soluzione era tediosa e soggetta a errori.

La memoria virtuale estende l'idea dei registri base e limite.

Ogni processo ha un proprio spazio degli indirizzi suddiviso in "pagine", che sono intervalli di indirizzi contigui.

Non tutte le pagine devono essere contemporaneamente nella memoria fisica: l'hardware crea una mappa di quelle direttamente in memoria, se una pagina manca, il sistema operativo interviene.

La maggior parte dei sistemi moderni usa il "paging" (paginazione) con una divisione dello spazio degli indirizzi in unità di dimensione fissa (può essere anche dinamica), es. 4 KB.

Alternativa: "segmentazione" con unità di dimensione variabile, ora meno utilizzata.

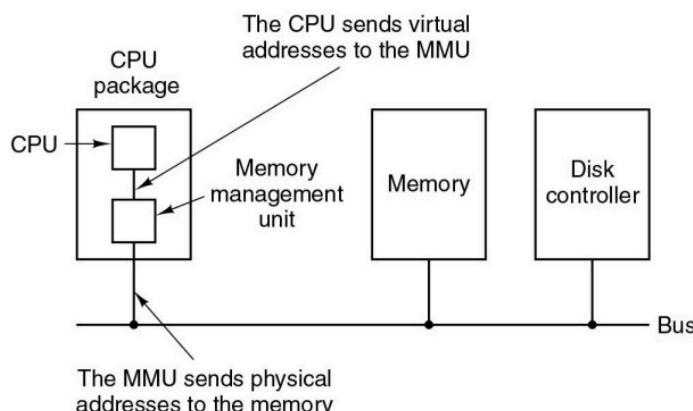
### Problema:

Finora la memoria può essere assegnata ai processi solo in blocchi contigui.

Soluzione (e vantaggio dell'uso della Memoria Virtuale):

Creare per il processo l'illusione di uno spazio di indirizzi ampio (ad esempio indicizzabile con 48 bit!!!). Questo spazio è noto come spazio di indirizzi virtuale (diverso dallo spazio di indirizzi fisico della macchina). La RAM (molto più limitata) è nota come memoria fisica.

La Memory Management Unit (MMU): traduce gli indirizzi virtuali (usati dal processo) in indirizzi fisici, tenendo così traccia di dove si trova ciascuna delle pagine. Richiede comunque tanti calcoli ogni volta che si accede ad un indirizzo, perciò viene messa a disposizione della CPU una piccola cache che permette di tenere gli ultimi indirizzi convertiti.



I sistemi moderni utilizzano la paginazione: dividendo la memoria fisica e virtuale in pagine di dimensioni fisse ad esempio 4096 byte o 4 KB e traducendo le pagine virtuali in pagine fisiche (frame).

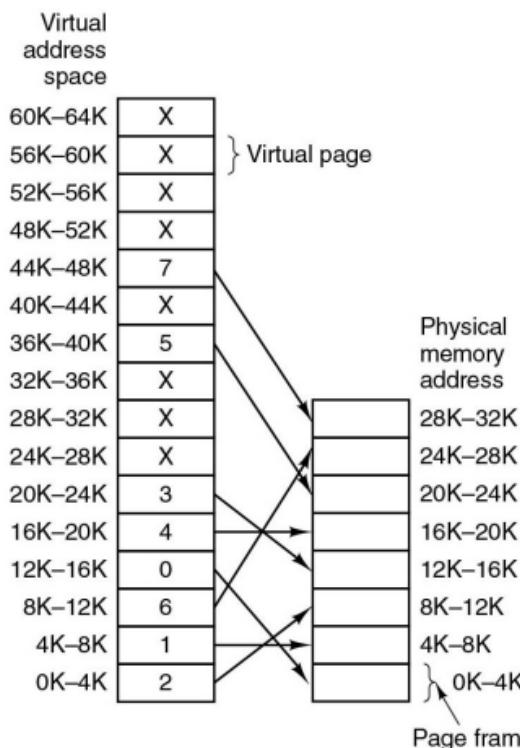
### Mappatura Memoria:

Nell'esempio 16 pagine virtuali possono essere mappate in 8 frame fisici usando la MMU. Tuttavia, non tutte le pagine virtuali sono mappate fisicamente (quelle NON mappate sono contrassegnate con una X). Quando un processo accede all'indirizzo 45K sarà il sistema operativo che tradurrà per la memoria fisica.

Se un programma fa riferimento a una pagina non mappata (non è mai stata richiesta e quindi mai allocata), si verifica un «Page fault». Il sistema operativo allora:

Carica la pagina richiesta nel frame libero o liberato e aggiorna la mappa della MMU per riflettere i cambiamenti. In caso la memoria fisica (page frame) è piena allora si sposta un frame raramente usato su disco, se serve con un algoritmo di scheduling.

La relazione tra gli indirizzi di memoria virtuale e fisica è data dalla Page Table.



Esempio: gestire istruzione MOV REG,32780 (immagine sopra)

Fa riferimento alla pagina virtuale 8,  $32780 - 2^{15} (32768) = 12$ , cioè indirizzo 12 della pagina. Se non è mappata, il sistema operativo potrebbe decidere di sostituire il frame 1. Spostando il precedente su disco, popolando il nuovo frame e puntando poi a  $4108 = 4096 + 12$ . Il page fault avviene nello spazio kernel durante il «trap» eseguito dal sistema operativo.

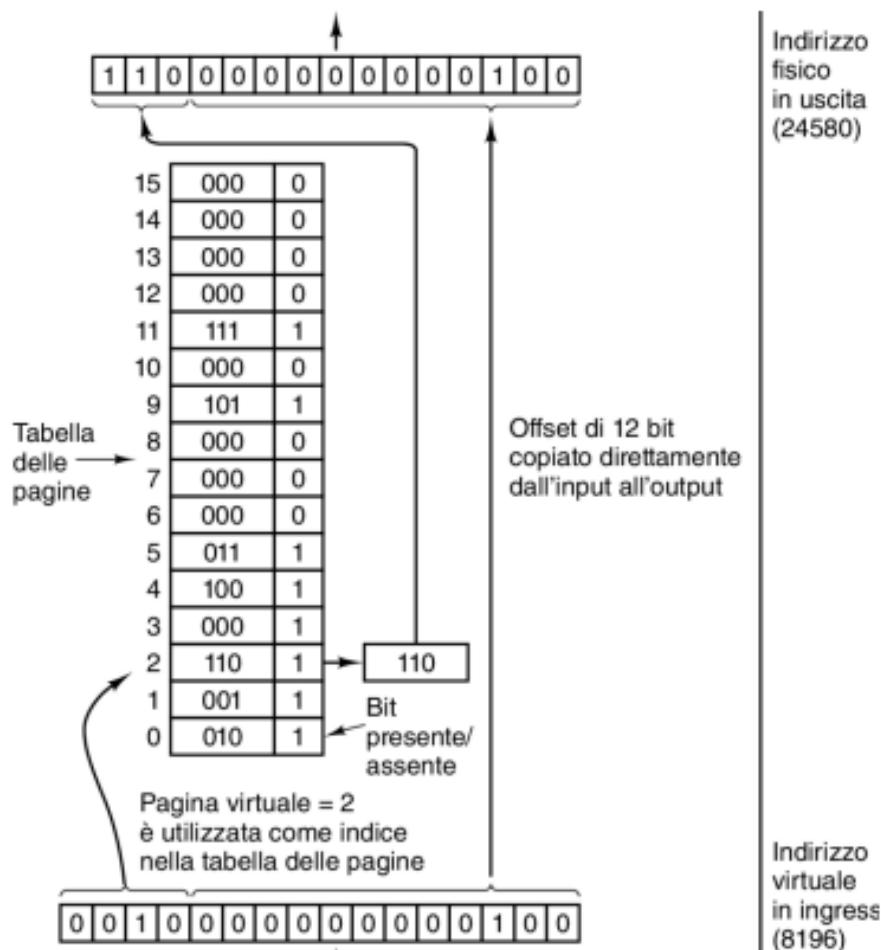
La MMU si occupa di tradurre questi indirizzi, funziona così:

Indirizzo Virtuale: 8196, Rappresentazione Binaria: 0010 00000000100

Abbiamo una suddivisione dell'Indirizzo Virtuale:

La prima parte indica il numero di pagina: 4 bit (permette di gestire 16 pagine). Seleziona la pagina corrispondente all'indice delle pagine (0010 = 2), la seconda parte l'offset: 12 bit

(indirizza 4096 byte per spostarmi nelle pagine che compongono ogni frame).



Indirizzi nei nostri esempi: 16 bit (per chiarezza nelle illustrazioni) ma nei moderni PC usano indirizzi a 32 o 64 bit. Con 32 bit e pagine da 4 KB: 12 bit per indirizzare 4096 byte per pagina la tabella delle pagine avrà  $2^{(32-12)} = 2^{20} = 1.048.576$  voci per le pagine!

Si avrà una taglia di 4GB ( $1.048.576$  voci \* 4KB per pagina) è «fattibile» anche per PC con «pochi» GB di RAM. Ma con gli Indirizzi a 64 bit e pagine da 4KB: Richiede  $2^{52}$  voci ( $\sim 4,5 \times 10^{15}$ ) nella tabella. In realtà nei sistemi a 64 bit si usano 48 bit 256TB bastano e avanzano (con  $2^{48}$  avremo una tabella degli indirizzi troppo grande che occupa tutta la memoria, si affronta con le tabelle multilivello).

Come è composta la Page Table?

Ogni voce nella PAGE TABLE ha informazioni cruciali, oltre al numero del frame (es 12 bit per 4KB), vengono utilizzati altri bit successivi come:

Bit Presente/Assente: Indica se la pagina virtuale è in memoria.

Bit Protezione: Specifica i tipi di accesso consentiti (lettura, scrittura, esecuzione).

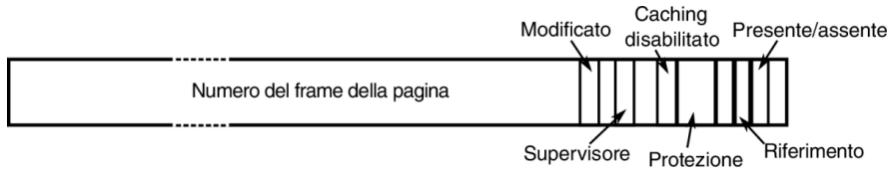
Bit Supervisor: Stabilisce se la pagina è accessibile solo al sistema operativo o anche ai programmi utente.

Bit Modificato (M) e Riferimento (R): Registrano l'uso della pagina:

Il bit Modificato si attiva quando la pagina viene scritta.

Il bit Riferimento viene impostato ogni volta che si accede alla pagina.I/O

Nota: per un processo l'indirizzo in memoria della «sua» tabella delle pagine è scritto nel registro Page Table Base Register (PTBR).



### Velocizzare la paginazione:

Ad ogni riferimento in memoria è necessaria una mappatura veloce (conversione da indirizzo virtuale a indirizzo fisico), inoltre ogni istruzione può richiedere più riferimenti alla tabella delle pagine. Se un'istruzione impiega 1 ns, la ricerca nella tabella delle pagine deve essere inferiore a 0,2 ns per evitare colli di bottiglia.

### Dimensione della Tabella delle Pagine:

Con 48 bit di indirizzamento e pagine di 4 KB, ci sono ~64 miliardi di pagine.

Usare centinaia di gigabyte solo per la tabella delle pagine è impraticabile dato che ogni processo richiede una propria tabella delle pagine.

Due possibili soluzioni consistono nel mettere la:

- Tabella delle Pagine in Registri Hardware:

Funzionamento: Un registro hardware per ogni pagina virtuale, caricato all'avvio del processo.

Vantaggi: Semplice, non richiede accessi alla memoria durante la mappatura.

Svantaggi: Costoso con tabelle delle pagine grandi, ricaricare l'intera tabella ad ogni cambio di contesto è inefficiente.

- Tabella delle Pagine in Memoria Principale:

Funzionamento: La tabella delle pagine è interamente in RAM, con un registro che punta al suo inizio.

Vantaggi: Facile da cambiare a ogni cambio di contesto, richiede solo il ricaricamento di un registro.

Svantaggi: Richiede accessi frequenti alla memoria, rendendo la mappatura più lenta.

### La soluzione sta nel mezzo:

Ogni istruzione richiede l'accesso alla memoria per prelevare l'istruzione stessa e un ulteriore accesso per la tabella delle pagine. I programmi tendono a fare molti riferimenti a un piccolo numero di pagine, solo una parte limitata delle voci della tabella delle pagine viene utilizzata frequentemente.

Viene quindi utilizzato il Translation Lookaside Buffer (TLB):

Dispositivo hardware che mappa indirizzi virtuali in fisici senza passare dalla tabella delle pagine, riducendo gli accessi alla memoria durante la paginazione.

### Struttura:

Piccolo numero di voci (es. 8-256), ciascuna con numero di pagina virtuale, bit modificato, codice di protezione, e frame fisico.

Funzionamento: Alla richiesta di un indirizzo virtuale, la MMU controlla prima nel TLB, se trovato e valido, il frame è prelevato direttamente dal TLB. Se non trovato (TLB miss),

avviene una ricerca normale nella tabella delle pagine e la voce trovata rimpiazza una voce nel TLB.

Gestione delle Modifiche: Le modifiche ai permessi di una pagina nella tabella delle pagine richiedono l'aggiornamento del TLB, la voce corrispondente viene quindi eliminata o aggiornata.

Valid	Virtual Page	Modified	Protection	Page Frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TLB in Architetture RISC:

Alcune macchine RISC come SPARC, MIPS, e HP PA gestiscono le voci del TLB tramite software.

Processo in Caso di TLB Miss: Un TLB miss non porta a una ricerca automatica nella tabella delle pagine da parte della MMU. Invece, si genera un errore di TLB e il sistema operativo deve intervenire che cerca la pagina, aggiorna il TLB, e riavvia l'istruzione.

Frequenza dei TLB Miss:

I TLB miss sono comuni a causa del numero limitato di voci nel TLB. Aumentare la dimensione del TLB è costoso e richiede compromessi nella progettazione dei chip.

Esistono due tipi di Miss:

Soft Miss: La pagina è in memoria ma non nel TLB. Richiede solo l'aggiornamento del TLB.

Hard Miss: La pagina non è in memoria e richiede un accesso alla memoria non volatile.

Page TableWalk e Diverse Tipologie di Miss:

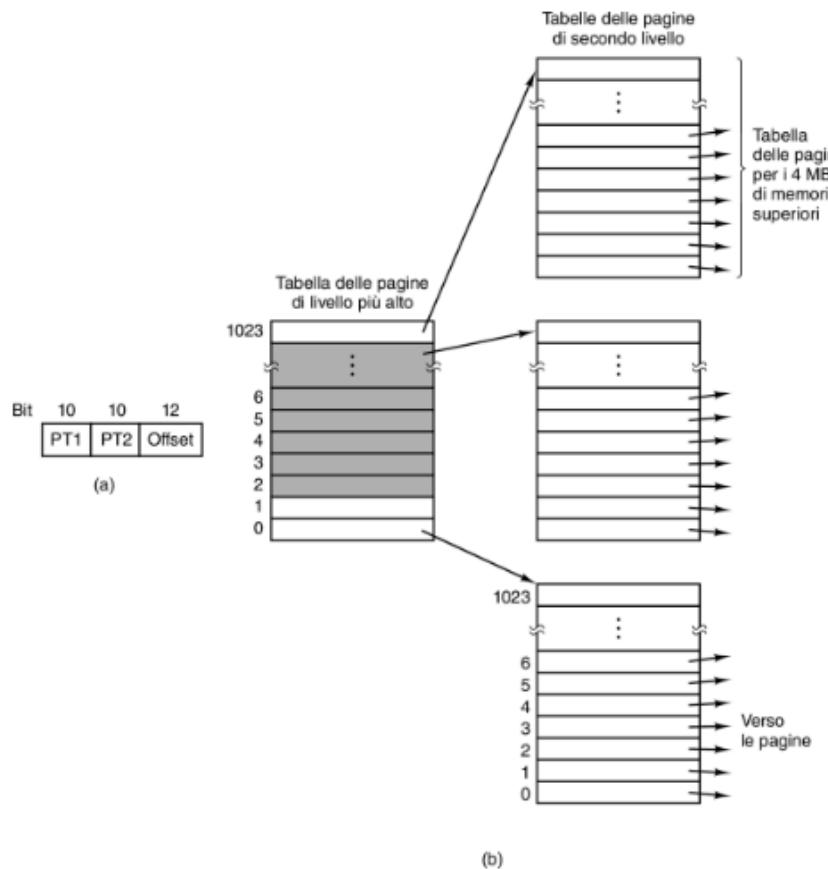
La ricerca nella gerarchia delle tabelle delle pagine è chiamata "page table walk"(va da TLB alla memoria al disco). I miss possono variare in «gravità» da minori (pagina in memoria ma non nella tabella delle pagine) a maggiori (pagina da caricare dalla memoria non volatile).

Un accesso a un indirizzo non valido può portare a un segmentation fault e alla terminazione del programma.

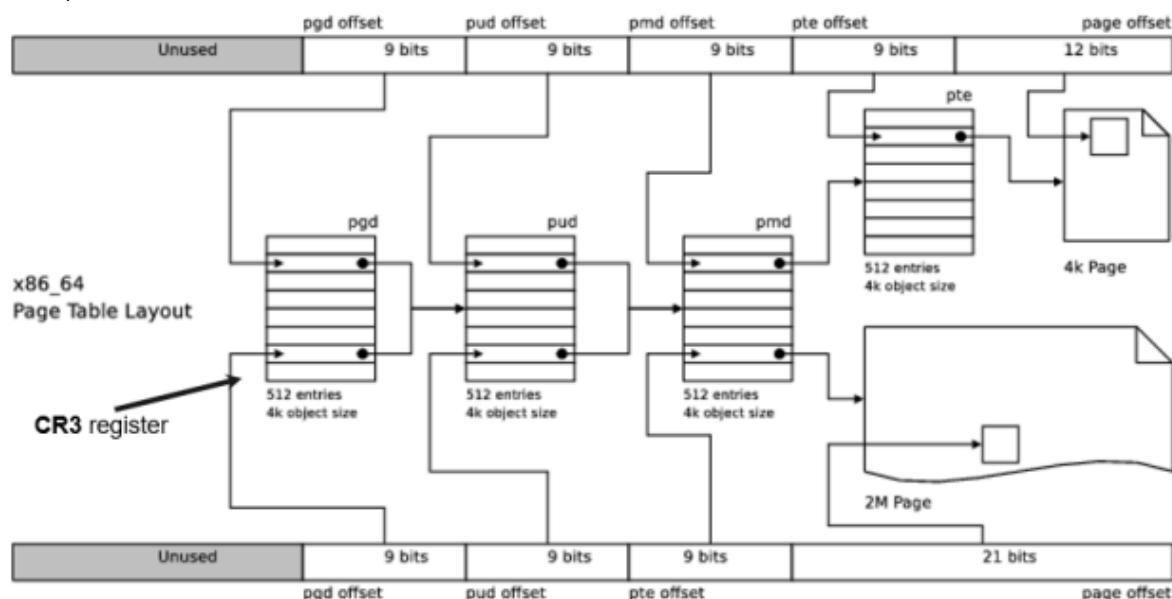
Come detto prima, uno spazio di indirizzi virtuali molto grande porterebbe a una tabella di pagine molto grande con conseguente spreco di memoria. Una possibile soluzione è la Tabella delle pagine Multilivello: cioè pagine in cui l'indirizzo viene diviso in più parti.

La tabella delle pagine è attraversata dalla MMU. Il CR3 register è un registro speciale che punta al vertice della gerarchia delle tabelle di pagina.

Esempio: a) Un indirizzo a 32-bit con due campi 10+10 bit, b) Una page table a due livelli



Per architetture a 64 bit l'indirizzo viene diviso in 4 livelli. (Se si vuole caricare in memoria una pagina più grande (un file mappato in memoria) si rinuncerà ad un livello per maggiore offset).



### Algoritmi di sostituzione delle pagine

Come detto il computer potrebbe utilizzare più memoria virtuale di quanta ne abbia di fisica. La paginazione quindi crea l'illusione di una memoria praticamente illimitata a disposizione dei processi utente. Quando una pagina logica non è in memoria si ha un page fault e il sistema operativo la deve caricare. Un'altra pagina logica potrebbe essere scambiata... Ma quale? Esistono diversi algoritmi di sostituzione delle pagine:

### Algoritmo ottimale

Concetto: Scegliere la pagina con il riferimento più distante nel futuro da rimuovere. Idealmente, si rimuove la pagina che non sarà usata per il maggior numero di istruzioni future. Esempio: «Se una pagina non sarà usata per 8 milioni di istruzioni e un'altra per 6 milioni, si rimuove la prima».

Problema: È impossibile per il sistema operativo prevedere il momento del prossimo riferimento per ciascuna pagina.

Il metodo ottimale non è quindi realizzabile in pratica perché richiede la previsione del futuro utilizzo delle pagine. Possibile implementazione su un simulatore per valutare le prestazioni rispetto agli algoritmi reali, quindi se un sistema ha prestazioni inferiori dell'1% rispetto all'ottimale, il miglioramento massimo teorico è dell'1%.

Gli algoritmi reali devono essere valutati per la loro applicabilità pratica, non per l'ottimalità teorica.

### NRU (Not Recently Used)

Obiettivo: Trovare le pagine non modificate che non sono state accedute «recentemente». Vengono usati i Bit di Stato R e M: R indica l'accesso alla pagina e M segnala le modifiche (le modifiche avvengono in memoria, non sul disco quindi se un programma prende un array dal disco e lo modifica, finché quel programma è ancora in esecuzione la modifica è avvenuta solo in memoria e sul disco è presente l'array originale).

Entrambi i bit vengono impostati dall'hardware a ogni accesso. Il bit R viene periodicamente ripulito per identificare pagine non recentemente usate, per esempio a ogni interrupt del clock. Si ha quindi una classificazione delle Pagine in base ai bit R e M. Le pagine sono divise in 4 classi (da 0 a 3) in funzione dell'uso e delle modifiche.

Classe 0: Non referenziata, non modificata.

Classe 1: Non referenziata, modificata.

Classe 2: Referenziata, non modificata.

Classe 3: Referenziata, modificata.

Le pagine di classe 1 sembrano a prima vista impossibili, ma appaiono quando un interrupt del clock azzerà il bit R di una pagina di classe 3. Gli interrupt del clock non azzerano il bit M perché questa informazione è necessaria per sapere se la pagina deve essere riscritta su disco o meno.

Selezione per Rimozione: NRU rimuove una pagina casuale dalla classe più bassa non vuota.

Vantaggi di NRU: Semplicità, efficienza implementativa e prestazioni accettabili.

### FIFO (First-In First-Out)

Obiettivo: FIFO è un algoritmo di paginazione che elimina la pagina più vecchia in memoria. Il sistema operativo rimuove la pagina in testa alla lista (la più vecchia) durante un page fault, aggiungendo la nuova pagina in coda (come se fosse una queue).

Problema di FIFO: Nel contesto informatico, la pagina più vecchia potrebbe ancora essere frequentemente utilizzata, rendendo FIFO poco efficace.

Conclusione: A causa di queste limitazioni, FIFO è raramente utilizzato nella sua forma più semplice.

### Second Chance (miglioramento del FIFO)

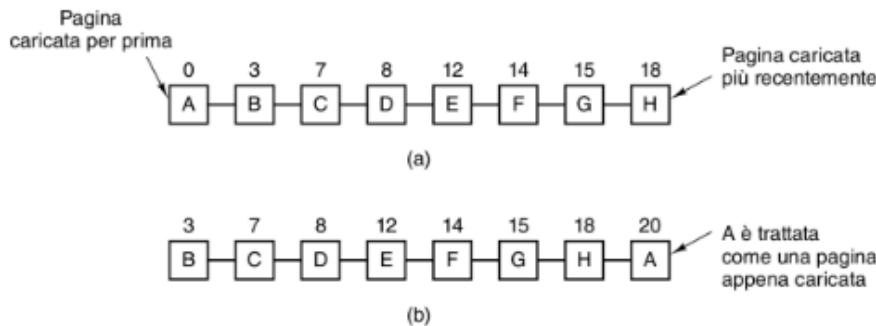
Obiettivo: Controllo del bit R (bit di lettura) della pagina più vecchia per decidere la rimozione.

Se  $R = 0$ : la pagina è vecchia e non usata di recente, quindi viene sostituita.

Se  $R = 1$ : il bit R viene azzerato, la pagina è reinserita in fondo alla lista e considerata come appena caricata.

a) Pagine ordinate in ordine FIFO.

b) Elenco delle pagine se si verifica un errore di pagina al tempo 20 e A ha il bit R impostato.  
I numeri sopra le pagine sono i loro tempi di caricamento.



Se la pagina A ha  $R = 0$ , viene rimossa.

Se A ha  $R = 1$ , viene messa in fondo alla lista.

Se trova una pagina non referenziata, la rimuove. Se tutte le pagine sono state referenziate, Seconda Chance opera come FIFO puro, con un ciclo completo di reset dei bit R prima di rimuovere la pagina iniziale.

Clock:

Obiettivo: Lista circolare dei frame di pagina con un puntatore simile a una lancetta d'orologio per identificare la pagina più vecchia.

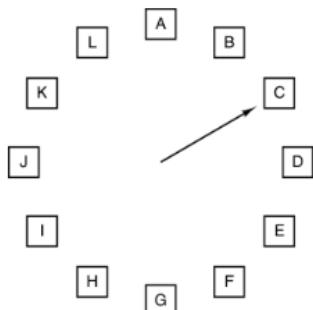
In caso di Page Fault:

Se  $R = 0$ , la pagina viene rimossa e sostituita con la nuova, poi il puntatore avanza.

Se  $R = 1$ , il bit viene azzerato e il puntatore si sposta alla pagina successiva.

Ripete il processo finché non trova una pagina con  $R = 0$ .

Vantaggio: Elimina l'inefficienza della continua riallocazione delle pagine lungo la lista rendendolo più efficiente e più performante rispetto a Seconda Chance e FIFO.



LRU (Least Recently Used)

Obiettivo: Pagine non usate di recente sono candidate alla sostituzione. C'è una lista delle pagine con quelle più usate in testa e quelle meno usate in coda. Ogni riferimento richiede l'aggiornamento della lista e copia di pagine intere, operazione costosa anche con hardware dedicato. Sebbene tendente all'ottimo, praticamente non è efficiente e non utilizzato.

Esistono però altri metodi per implementare l'LRU tramite un contatore a 64 bit (hardware)

per ogni riferimento a memoria.

Alla generazione di un page fault, si rimuove la pagina con il contatore più basso, indicando l'uso meno recente.

### NFU (Not Frequently Used)

Obiettivo: Associa un contatore a ogni pagina, incrementato con ogni interrupt del clock in base al bit R. Tanti accessi ad una pagina, significa alto valore di «frequenza» assegnato alla pagina, cioè minore possibilità di rimozione.

Svantaggi: NFU non dimentica l'uso passato, può portare a scelte subottimali in ambienti multi-pass o in fase di boot. Esempio: una pagina utilizzata con altissima frequenza in un determinato periodo e poi «abbandonata» potrebbe non venire sostituita.

Un miglioramento di NFU si ha con l'Aging:

Numero di bit fisso, esempio 8 bit, ad ogni interrupt del clock i bit vengono spostati a destra, prima dello shift dei contatori, il bit R viene aggiunto al lato sinistro.

Effetto dell'Aging: Emula LRU, dando meno peso agli usi passati e preferendo le pagine meno referenziate di recente.

Con NFU e aging, la pagina 3 viene rimossa poiché la pagina 5 ha avuto riferimenti in (a) prima e la pagina 3 no.

	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
Page	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00010000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	(a)	(b)	(c)	(d)	(e)

Differenze da LRU: Aging non distingue l'ordine esatto dei riferimenti recenti e ha un orizzonte temporale limitato. Non è necessariamente un male, anzi.

Fattibilità: 8 bit sono generalmente sufficienti per un buon compromesso tra accuratezza e uso di memoria.

### Working Set:

Il Working Set rappresenta l'insieme delle pagine (locazione di riferimento) a cui un processo fa riferimento durante una fase dell'esecuzione.

### Demand Paging:

Le pagine sono caricate in memoria "on demand", solo quando necessario. Inizialmente, molti page fault si verificano finché non vengono caricate tutte le pagine necessarie (esiste quindi l'idea di precaricamento: al posto di fare molti page fault uno per volta si fanno tutti

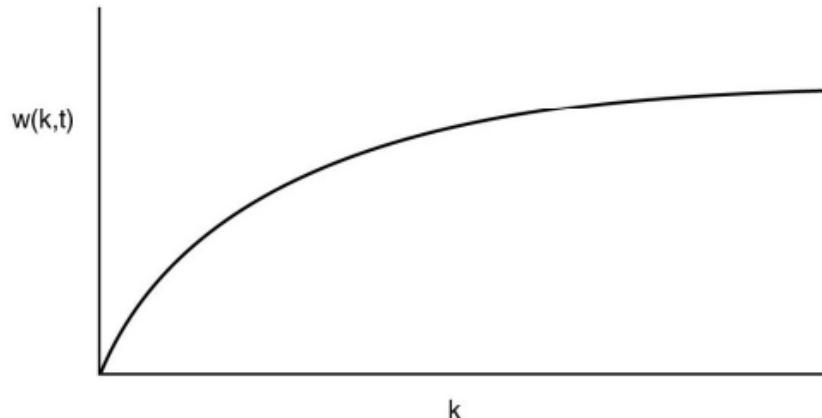
insieme precaricando le pagine che servono, sprecando meno tempo per accedere in memoria, mettere in pausa il processo etc.).

Definizione: Working set  $w(k,t)$  è l'insieme di pagine usate negli ultimi  $k$  riferimenti.

$w(k,t)$  è una funzione monotona non decrescente al crescere di  $k$ .

Asintoto: Il limite di  $w(k,t)$  è finito, correlato allo spazio degli indirizzi del programma.

Implicazione: C'è un ampio intervallo di  $k$  dove il working set resta invariato.



#### Gestione della Memoria e Page Fault:

Se il working set di un processo è completamente in memoria, si verificano pochi page fault.

Se il working set è più grande della memoria disponibile, si verificano frequenti page fault, rallentando significativamente il processo. Fenomeno noto come thrashing (OOM o fare costantemente swap tra memoria e disco).

Working Set Model: Molti sistemi operativi cercano di tracciare il working set di ogni processo e di mantenerlo in memoria per ridurre i page fault. La pre-paginazione carica in anticipo le pagine basandosi sul working set del processo.

In pratica: è spesso definito in termini di tempo, ad esempio, le pagine usate negli ultimi  $\tau$  secondi di tempo di esecuzione.

Algoritmo di Sostituzione Basato sul Working Set: Alla verifica di un page fault, si ricerca una pagina fuori dal working set per rimuoverla. Utilizza informazioni come il bit di riferimento e il tempo dell'ultimo utilizzo per determinare quali pagine rimuovere.

Impostazione dei Bit R e M: Un interrupt periodico azzerà il bit R a ogni ciclo di clock.

Durante un Page Fault: Scansione delle pagine alla ricerca di una pagina da rimuovere.

Controllo del bit R per ogni pagina:

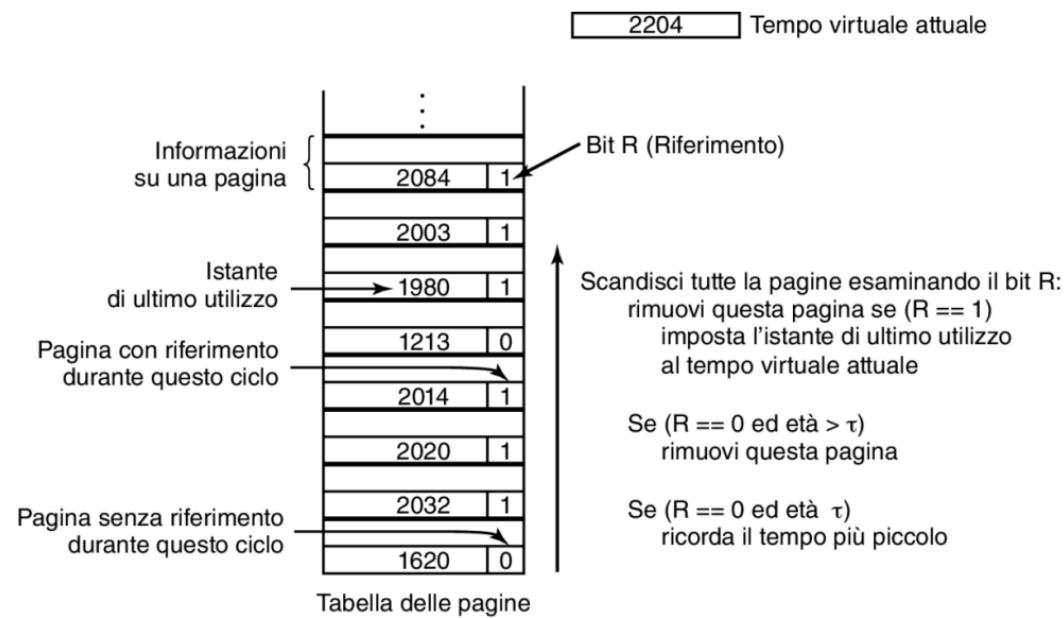
$R = 1$ : Aggiornamento del tempo dell'ultimo utilizzo, la pagina è nel working set.

$R = 0$  e  $Eta > tau$ : La pagina non è nel working set e viene rimossa.

$R = 0$  e  $Eta \leq tau$ : La pagina rimane, ma si contrassegna la più vecchia per possibile rimozione.

Se nessuna pagina è rimovibile, viene selezionata la più vecchia con  $R = 0$  in caso contrario, una pagina a caso. Il problema è però che devo andare a scansionare questa lista

costantemente, è un overhead.



#### Miglioramento dell'Algoritmo Working Set:

WSClock è un'evoluzione dell'algoritmo Clock che integra informazioni del working set.

Popolare per la sua semplicità e buone prestazioni.

Struttura Dati: Usa una lista circolare di frame, simile all'algoritmo Clock. Ogni frame nella lista contiene: il tempo dell'ultimo utilizzo, il bit R (Riferimento), il bit M (Modificato).

Ad ogni page fault è esaminata per prima la pagina indicata dalla lancetta dell'orologio.

Se il bit R = 1, la pagina NON è la candidata ideale alla rimozione è stata usata nel ciclo del clock, Il bit R viene quindi impostato a 0.

La lancetta avanza alla pagina successiva e l'algoritmo viene ripetuto per la nuova pagina.

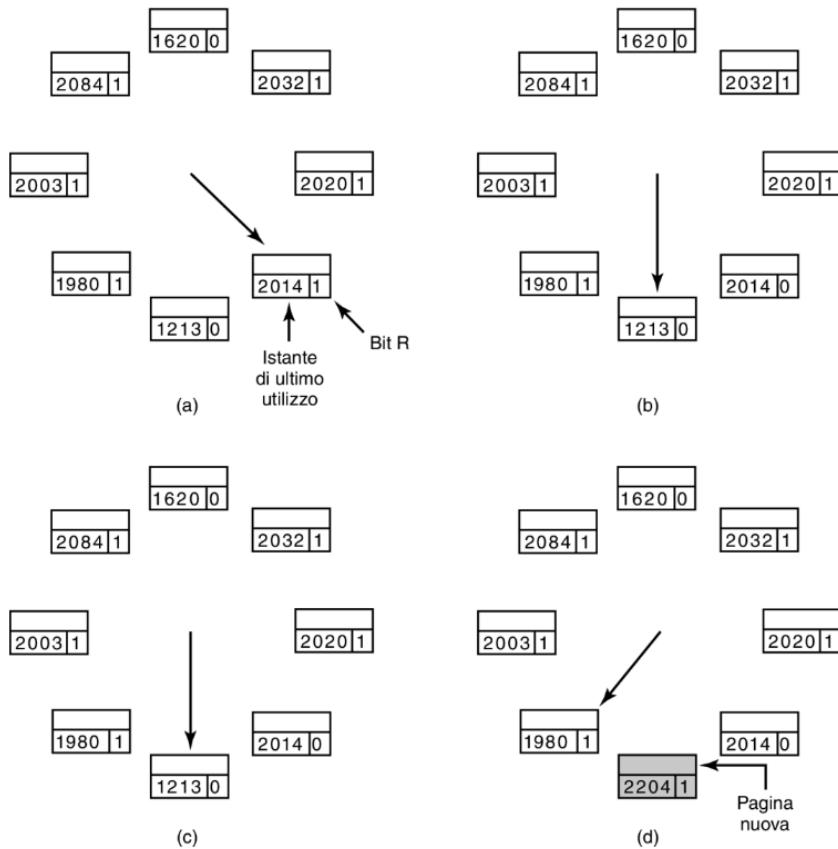
La situazione dopo questa sequenza è mostrata nella Figura (b).

Se la pagina indicata ha R = 0 (c) e se l'età è maggiore di tau:

Se M = 0 (pagina pulita) non è nel set di lavoro e ne esiste una copia valida su memoria non volatile, il frame viene semplicemente riciclato e vi viene posta la nuova pagina (d).

Se M = 1 invece la pagina è «sporca» (ovvero modificata) non ne esiste una copia valida in memoria non volatile, non può essere sfrattata immediatamente.

Per evitare «rallentamenti» (come un cambio di processo) la scrittura su memoria non volatile viene schedulata e rimandata, lungo la lista potrebbe esserci una pagina pulita e vecchia che può essere usata immediatamente quindi la lancetta avanza e l'algoritmo procede con la pagina successiva.



#### Limitazione Scritture su Memoria Non Volatile:

Esiste la possibilità di schedulare tutte le pagine per I/O su memoria non volatile in un ciclo di clock. Per ridurre il traffico su disco, si imposta quindi un limite massimo di scritture (n pagine). Una volta raggiunto il limite n, ulteriori scritture non vengono scheduled.

Comportamento al Completamento del Giro di Orologio:

- Quando ci Sono Scritture Pendenti:

La lancetta prosegue il suo giro cercando pagine "pulite" (non modificate). Non appena una scrittura pendente viene completata, la pagina associata diventa "pulita". La lancetta seleziona la prima pagina pulita che incontra e la rimuove dalla memoria.

- Quando NON ci Sono Scritture Pendenti:

Significa che tutte le pagine sono attivamente utilizzate ("nel set di lavoro"). La strategia diventa quella di scegliere e rimuovere una pagina pulita a caso. Se non ci sono pagine pulite disponibili, la pagina corrente viene scelta per la rimozione e la sua copia viene scritta su disco.

#### Resume:

Ottimale: Non implementabile, ma utile come termine di confronto e valutazione.

LRU (Last Recently Used): Eccellente, ma difficile da implementare con precisione.

NRU (Not Recently Used): Approssimazione molto rozza dell'LRU.

FIFO (First-In, First Out): Potrebbe eliminare pagine importanti.

Seconda chance: Deciso miglioramento rispetto al FIFO.

Clock: Realistico.

NFU (Non Frequently Used): Approssimazione abbastanza rozza dell'LRU.

Aging: Algoritmo efficiente che approssima bene l'LRU.

Working set: Piuttosto dispendioso da implementare.

WSClock: Algoritmo efficiente e buono.

Algoritmi Preferiti:

Aging e WSClock sono i «migliori» tra gli algoritmi analizzati. Entrambi basati rispettivamente su LRU e sull'idea di Working Set, con buone prestazioni e implementazione efficiente.

La nozione di «migliore» è il risultato del trade-off tra la complessità del metodo e i vincoli hardware che il sistema operativo deve comunque rispettare.

Implementazioni nei Sistemi Operativi: Sistemi come Windows e Linux adottano varianti di questi algoritmi, a volte combinando diversi elementi per ottimizzare le prestazioni in base a specifiche esigenze e al tipo di hardware.

### Problemi di Progettazione per Sistemi di Paginazione

La paginazione è un processo complesso che richiede una comprensione approfondita di molteplici aspetti per una progettazione efficace.

Aspetti Cruciali:

Allocazione della Memoria: Scelta tra allocazione globale vs locale oppure allocazione equa vs proporzionale e come questa influisce sulla gestione delle risorse e sulle prestazioni del sistema.

Gestione dei Page Fault: Monitoraggio della frequenza dei page fault per ottimizzare l'uso e allocazione della memoria, e ridurre i tempi di attesa.

Ottimizzazione delle Prestazioni: Valutare le prestazioni del sistema di paginazione per massimizzare l'efficienza. Esempio: «quando deve essere grande una pagina?», «come limitare l'uso della memoria per i processi?»

Decisioni di Progettazione: Considerare fattori come la dimensione del set di lavoro, il comportamento dei processi, e la località dei riferimenti alla memoria per scegliere l'algoritmo più adatto.

### Allocazione Globale VS Locale:

- Allocazione Locale: Ogni processo riceve una porzione fissa della memoria, semplice da implementare, ma può portare a inefficienze se è allocato tutto il set di lavoro del processo.
- Allocazione Globale: Distribuzione dinamica della memoria tra i processi, più efficace per adattarsi alle esigenze variabili dei processi, ma richiede una gestione più complessa.

Age
A0
10
A1
7
A2
5
A3
4
A4
6
A5
3
A6
B0
9
B1
4
B2
6
B3
2
B4
5
B5
6
B6
12
C1
3
C2
5
C3
6

(a)

Age
A0
A1
A2
A3
A4
A5
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

Age
A0
A1
A2
A3
A4
A5
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(c)

Esempio Pratico: In figura la differenza tra sostituzione locale (solo pagine del processo A) e globale (pagine di tutti i processi).

Allocazione Locale: è possibile rimuovere solo pagine del processo A.

Allocazione Globale: è possibile rimuovere pagine di qualsiasi processo Il processo.

Adattabilità degli Algoritmi Globali: Gli algoritmi globali di sostituzione delle pagine si adattano meglio alle esigenze variabili dei processi, aumentano l'efficienza quando la dimensione del set di lavoro varia nel tempo.

Limiti degli Algoritmi Locali: Il thrashing può verificarsi con algoritmi locali se il set di lavoro di un processo cresce oltre la memoria allocata, oppure la memoria può essere sprecata quando il set di lavoro di un processo si riduce e la memoria non viene riassegnata.

Gestione Dinamica della Memoria: Con l'allocazione globale, il sistema operativo deve dinamicamente assegnare e riassegnare frame ai processi, è possibile utilizzare i bit di aging per monitorare la frequenza di accesso delle pagine, anche se questo potrebbe non essere sufficiente per prevenire il thrashing.

Sfide del Monitoraggio del Set di Lavoro: I bit di aging forniscono una stima approssimativa, che potrebbe non riflettere cambiamenti rapidi nel set di lavoro, è fondamentale che il sistema di paginazione possa reagire in modo agile ai cambiamenti delle esigenze di memoria.

Allocazione Equa vs Proporzionale:

- Allocazione Equa: Distribuzione uniforme dei frame tra i processi. Esempio: 12.416 frame divisi equamente tra 10 processi risultano in 1.241 frame per processo.

Svantaggi: Non tiene conto delle diverse esigenze di memoria tra processi di dimensioni varie.

- Allocazione Proporzionale: Assegnazione di frame in base alla dimensione del processo. Rispecchia meglio le necessità di memoria, evitando allocazioni inadeguate.

Importanza del Limite Minimo di Pagine:

Bisogna assicurare che ogni processo abbia abbastanza pagine per eseguire le operazioni fondamentali e prevenire situazioni in cui, processi con istruzioni che attraversano i limiti delle pagine non possano eseguire.

Gestione Dinamica dei Frame:

Inizio con un'allocazione proporzionale alla dimensione del processo ma con aggiornamento dinamico in base all'evoluzione delle esigenze durante l'esecuzione (più page fault un processo farà più avrà bisogno di pagine).

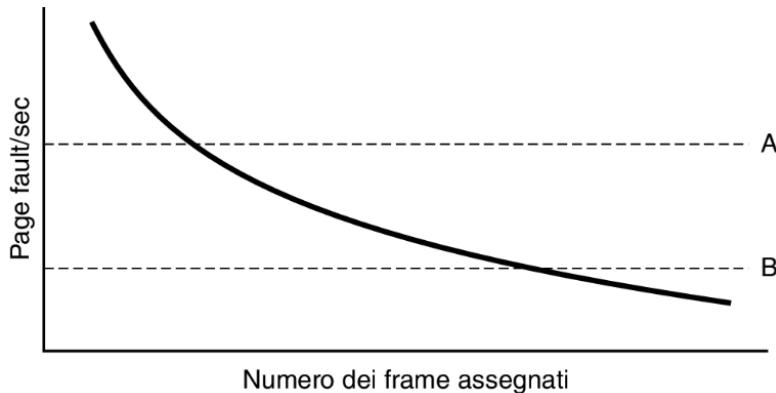
Page Fault Frequency (PFF):

Monitoraggio della frequenza dei page fault per regolare l'allocazione di memoria di un processo, aumenta i frame se i page fault sono troppo frequenti, diminuisce se sono rari. Non specifica quale pagina rimuovere, focalizzandosi sulla dimensione dell'allocazione.

Relazione tra Frame Assegnati e Page Fault:

Secondo algoritmi come LRU, più pagine vengono assegnate a un processo, meno frequenti saranno i page fault. La frequenza di page fault diminuisce man mano che aumenta il numero di frame assegnati.

Per monitorare la frequenza dei Page Fault si contano i page fault per secondo e si utilizza una media mobile per tenere traccia delle fluttuazioni.



- A. Alta frequenza di page fault indica necessità di più frame.  
 B. Bassa frequenza di page fault suggerisce che il processo ha più memoria del necessario.

#### Thrashing in Presenza di Allocazione Ottimale:

Anche con il miglior algoritmo, il thrashing purtroppo può sempre verificarsi se i set di lavoro di tutti i processi eccedono la memoria disponibile, il PFF può segnalare una richiesta collettiva di più memoria senza che nessun processo possa cedere frame.

#### Strategie di Mitigazione (Soluzioni):

- Out Of Memory Killer (OOM): è un processo di sistema che seleziona e termina i processi in base a un punteggio di "cattiveria" per liberare memoria, processi con elevato utilizzo di memoria o minor importanza sono tipicamente selezionati.
- Swapping (Scambio): Meno drastico dell'OOM Killer, sposta i processi su memoria non volatile, liberando le loro pagine per altri processi, può ridurre la richiesta di memoria senza interrompere l'esecuzione dei processi.

#### Tecniche di riduzione della memoria:

Scheduling a Due Livelli: alcuni processi sono in memoria non volatile e solo una parte è in memoria volatile, aiuta a gestire meglio il carico di memoria ed è utile per ridurre occupazione di memoria di processi in background in sistemi interattivi.

Gestione della Multiprogrammazione: La selezione dei processi da spostare considera caratteristiche come: sono processi CPU bound o I/O bound, qual è la dimensione e/o frequenza di paginazione di questi processi.

Altre Tecniche: Oltre a «uccidere» o spostare processi, si possono usare compattamento, compressione e deduplicazione (same page merging).

#### Policy di Pulizia:

La policy di pulizia è un aspetto critico nella gestione della memoria.

Aging e Frame Liberi: L'aging è più efficace con molti frame di pagina liberi disponibili, se i frame sono tutti occupati e modificati, occorre scrivere le vecchie pagine sul disco prima di caricarne di nuove, è preferibile quindi mantenere un buon numero di frame di pagina liberi piuttosto che occupare tutta la memoria e cercare frame liberi solo al bisogno, di ciò si occupa il paging daemon.

Paging Daemon: è un processo in background usato dai sistemi di paginazione, inattivo per la maggior parte del tempo, si attiva periodicamente per controllare lo stato della memoria, quando i frame liberi scarseggiano, inizia a selezionare pagine da rimpiazzare utilizzando un algoritmo di sostituzione delle pagine.

**Scrittura in Memoria Non Volatile:** Se le pagine sono state modificate, vengono scritte sul disco, i contenuti precedenti delle pagine vengono conservati, permettendo un eventuale rapido ripristino in caso la pagina venisse richiesta senza causare un page fault.

Implementazione è fatta con un «Clock a Due Lancette»: La lancetta anteriore (gestita dal paging daemon) avanza scrivendo le pagine sporche in memoria non volatile e procede senza azioni ulteriori sulle pagine pulite mentre La lancetta posteriore si occupa della sostituzione delle pagine con maggiore probabilità di trovare pagine pulite (grazie al lavoro del paging daemon).

#### Dimensione delle Pagine:

I sistemi operativi possono selezionare la dimensione delle pagine. Esempio: unendo due pagine da 4096 byte per formarne una da 8 KB.

**Fattori a Favore di Pagine Piccole:** Riducono la frammentazione interna (spazio sprecato nelle pagine parzialmente vuote) e l'utilizzo di memoria, un programma potrebbe richiedere meno memoria con pagine più piccole.

**Svantaggi delle Pagine Piccole:** Richiedono tabelle delle pagine più grandi (più voci) e possono aumentare il tempo e lo spazio necessario per il trasferimento di dati e la gestione della memoria.

**Dimensione Ottimale:** Determinata equilibrando frammentazione interna (favorevole a pagine più grandi) e overhead della tabella delle pagine (favorevole a pagine più piccole).

**Pagine di Diverse Dimensioni:** Alcuni sistemi operativi utilizzano pagine di diverse dimensioni per parti diverse del sistema (ad es., pagine grandi per il kernel).

**Transparent Huge Pages (THP):** Tecnica per utilizzare pagine di grandi dimensioni (anche 1GB) ottimizzando l'uso della memoria, spostando la memoria del processo per creare intervalli contigui.

#### Parametri Considerati:

Dimensione media del processo:  $s$  byte (esempio 1MB).

Dimensione della pagina:  $p$  byte (da calcolare).

Dimensione di ogni voce nella tabella delle pagine:  $e$  byte (esempio 4 o 8 byte).

#### Calcolo Overhead:

Numero di pagine per processo:  $\approx s/p$ .

Spazio occupato nella tabella delle pagine:  $e \cdot s / p$  byte.

Memoria sprecata per frammentazione interna nell'ultima pagina:  $p/2$ .

Questo perché è presente il Fenomeno dell'Ultima Pagina: Per qualsiasi processo, l'ultima pagina di memoria allocata potrebbe non essere completamente riempita. Esempio: un processo richiede 10.5 KB di memoria la pagina è di 4 KB, il sistema dovrà allocare 3 pagine lasciando 1.5 KB di spazio inutilizzato nell'ultima pagina.

Overhead totale:  $es/p + p/2$ :

Il primo termine (tabella delle pagine) aumenta con pagine più piccole.

Il secondo termine (frammentazione interna) aumenta con pagine più grandi.

L'ottimo si trova bilanciando questi due fattori.

Formula per la Dimensione Ottimale delle Pagine:

Derivata della funzione di overhead rispetto a  $p$  uguagliata a zero:  $-se/p^2 + 1/2 = 0$ .

Dimensione ottimale delle pagine:  $p = \text{rad}(2se)$ .

Esempio: Per  $s = 1$  MB e  $e = 8$  byte,  $p$  ottimale è 4 KB (dimensione comune attuale).

## Problemi di Progettazione:

La maggior parte dei computer ha un unico spazio di indirizzamento condiviso da programma e dati. In passato alcuni sistemi avevano uno spazio di indirizzamento separato per istruzioni e dati. Al giorno d'oggi si vedono ancora spazi Istruzioni e Dati separati nelle cache, nei TLB o dove lo spazio è poco.

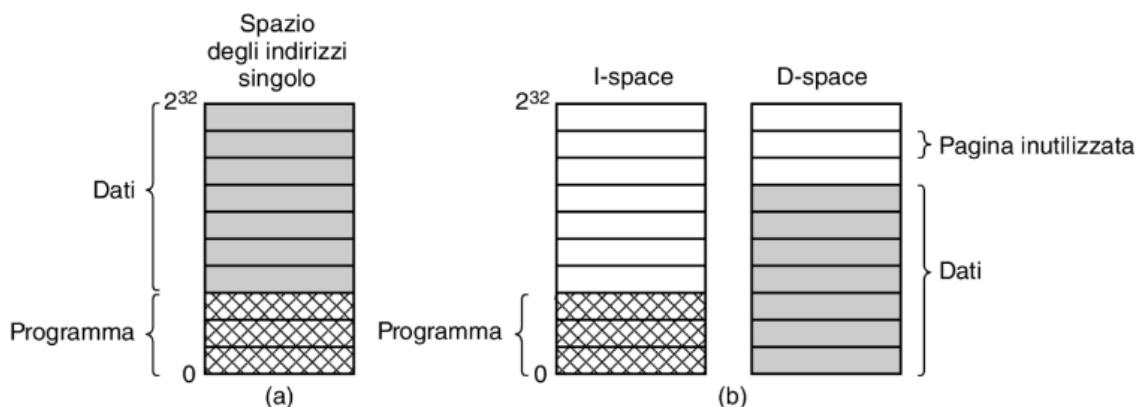
Motivazione della Condivisione: E' comune che molti utenti eseguano lo stesso programma o utilizzino le stesse librerie quindi condividere pagine di memoria tra questi processi è più efficiente che mantenere copie separate.

Il tipo di pagine condivise sono: le pagine di sola lettura, come il testo dei programmi ma non le pagine dei dati.

Per facilitare la condivisione è meglio separare spazi di indirizzo in:

I-space: Istruzioni.

D-space: Dati.



Processi diversi possono utilizzare la stessa tabella delle pagine per l'I-space ma tabelle diverse per il D-space.

Implementazione e Scheduling: ciascun processo ha puntatori sia all'I-space che al D-space, lo scheduler utilizza questi puntatori per impostare l'MMU, collegando a più processi lo stesso I-space.

## Gestione delle pagine:

Problemi con Pagine Condivise: La rimozione di un processo da memoria può causare numerosi page fault in un altro processo che condivide le stesse pagine. È cruciale sapere se le pagine sono ancora in uso per evitare la loro liberazione accidentale.

La Condivisione dei Dati è più complessa rispetto alla condivisione del codice! Esempio: in UNIX, dopo una fork, genitore e figlio condividono sia il testo che i dati, inizialmente come sola lettura.

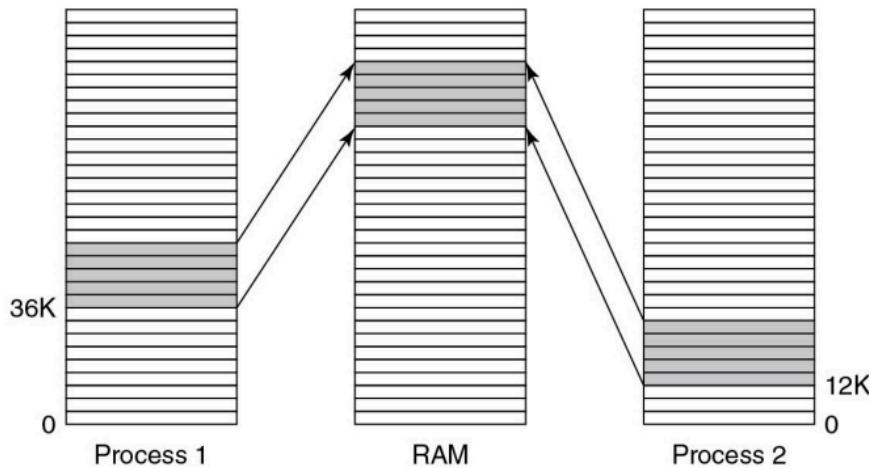
**Copy on Write (Copia in Caso di Scrittura):** Se un processo modifica i dati, si genera una trap, e viene creata una copia della pagina modificata, entrambe le copie diventano poi modificabili (READ/WRITE). Questo metodo evita la copia di pagine che non vengono mai modificate.

## Librerie Condivise:

Condivisione su Ampia Scala: I SO condividono automaticamente funzioni che più programmi utilizzano caricandole sulle pagine. Per evitare problemi, meglio pagine in sola lettura.

Librerie condivise - Dynamic Link Libraries (DLLs): Usate per ridurre l'ingombro di grandi librerie comuni (librerie che già sono compilate) risparmiando spazio.

**Problema di Indirizzamento:** Le librerie condivise possono essere posizionate a indirizzi diversi nei vari processi, questo impedisce l'uso di indirizzi assoluti nelle istruzioni. Perciò le librerie condivise vengono compilate con indirizzi relativi anziché assoluti, le istruzioni usano offset relativi piuttosto che puntare a indirizzi specifici.



#### File "mappati" in memoria:

**Concetto:** I file mappati consentono a un processo di mappare un file all'interno del proprio spazio di indirizzi virtuali (così da accederci come se fosse un array).

**Funzionamento:** Alla mappatura, nessuna pagina viene caricata immediatamente, sono paginate su richiesta dalla memoria non volatile, man mano che vengono riferite.

**Scrittura su File:** Quando il processo termina o la mappatura è eliminata, tutte le pagine modificate vengono riscritte sul file (in caso di kill del processo ci potrebbe essere un problema di scrittura).

**Modello I/O Alternativo:** Offre un modo diverso di eseguire I/O, permettendo di accedere al file come se fosse un grande array di caratteri in memoria.

**Utilizzato anche per la comunicazione tra Processi:** Se più processi mappano lo stesso file contemporaneamente, possono comunicare attraverso questa memoria condivisa. Le modifiche apportate da un processo sono immediatamente visibili agli altri.

#### Problemi di implementazione della memoria virtuale

**Sfide nell'Implementazione:** Bisogna scegliere tra algoritmi teorici (es. Seconda Chance, Aging) e pratiche operative (allocazione locale/globale, paginazione a richiesta/prepaginazione).

#### Attività del Sistema Operativo nella Paginazione:

- Creazione del Processo:

Determinare le dimensioni iniziali del programma e dei dati, creare e inizializzare la tabella delle pagine, allocare spazio nella memoria non volatile per lo scambio, inizializzare l'area di scambio e registrare informazioni nella tabella dei processi.

- Esecuzione del Processo:

Azzerrare la MMU (cambio contesto) e, se necessario, svuotare il TLB. Rendere attiva la tabella delle pagine del processo. Pre-paginazione: Facoltativamente, caricare alcune pagine in memoria per ridurre i page fault iniziali.

- Gestione dei Page Fault: Determinare l'indirizzo virtuale che ha causato il fault. Trovare la pagina necessaria nella memoria non volatile. Scegliere un frame disponibile,

eventualmente sfrattando pagine vecchie. Caricare la pagina nel frame e ripristinare il contatore del programma.

- Chiusura del Processo: Rilasciare la tabella delle pagine, le pagine in memoria e lo spazio su disco (per pagine non referenziate). Gestire le pagine condivise con altri processi, rilasciandole solo dopo l'ultimo utilizzo.

I 10 passi del Page Fault:

1. Trap nel Kernel da Parte dell'Hardware: (La pagina non sta in cache, la pagina non sta nella TLB, la pagina non sta in memoria, la pagina va presa dal disco) L'hardware esegue una trap nel kernel, salvando il contatore del programma nello stack e le informazioni sull'istruzione corrente salvate nei registri speciali della CPU.
2. Avvio Routine di Servizio Interrupt: Viene eseguita una routine in assembly per salvare i registri e altre informazioni volatili, si invoca il gestore dei page fault.
3. Identificazione della Pagina Virtuale Necessaria: Il sistema operativo determina quale pagina virtuale manca, se non disponibile dai registri hardware, recupero e analisi dell'istruzione dal contatore di programma.
4. Verifica Validità Indirizzo e Protezione: Controllo della validità dell'indirizzo e coerenza della protezione con l'accesso, se invalide, invio di un segnale di errore o terminazione del processo (ancor prima di caricare la pagina bisogna fare i controlli).
5. Rilascio di un Frame Libero: Se non ci sono frame liberi, esecuzione di un algoritmo di sostituzione delle pagine, se la pagina è "sporca", viene schedulata per la scrittura in memoria non volatile e il processo è sospeso.
6. Caricamento della Pagina Richiesta: Una volta liberato (o scritto in memoria non volatile), il frame viene usato per caricare la pagina necessaria da memoria non volatile. Durante il caricamento della pagina, il processo in page fault è ancora sospeso e viene eseguito, se disponibile, un altro processo utente.
7. Aggiornamento delle Tabelle delle Pagine: Al completamento del trasferimento dal supporto non volatile, le tabelle delle pagine vengono aggiornate per riflettere la nuova posizione della pagina (si manda un interrupt dal disco che dice di aver finito). Il frame viene contrassegnato come disponibile.
8. Ripristino dell'Istruzione in Errore: L'istruzione in errore è riportata allo stato che aveva all'inizio. Il contatore di programma è ripristinato in modo da puntare a quell'istruzione.
9. Ripresa del Processo in Errore: Il processo precedentemente in errore viene schedulato per l'esecuzione. Ritorno alla routine in assembly che lo aveva interrotto.
10. Ricarica dei Registri e Ritorno allo Spazio Utente: La routine di servizio ricarica i registri e le informazioni di stato. Il controllo ritorna allo spazio utente per continuare l'esecuzione da dove era stata interrotta.

Bloccare le pagine in memoria durante l'I/O:

Scenario: Un processo invia una richiesta di lettura da un file o dispositivo in un buffer nel suo spazio di indirizzi. Mentre attende il completamento dell'I/O, può essere sospeso per permettere l'esecuzione di un altro processo.

Problema con Page Fault: Se il secondo processo genera un page fault, esiste il rischio che la pagina contenente il buffer di I/O venga selezionata per essere rimossa. Se avviene un trasferimento DMA (Direct Memory Access) su quella pagina, la rimozione potrebbe causare scritture errate nei dati.

Soluzione: Le pagine utilizzate per l'I/O vengono "bloccate" o "pinned" (fissate) in memoria, prevenendo la loro rimozione. Questo approccio assicura che le operazioni di I/O possano

procedere senza interruzioni.

Alternativa: Un'altra strategia è gestire l'I/O nei buffer del kernel e poi copiare i dati nelle pagine utente. Questo metodo richiede una copia aggiuntiva dei dati, potenzialmente rallentando il processo.

Ma dove viene messa una pagina quando viene spostata nella memoria non volatile dopo essere stata «paginata fuori» dalla memoria?

Gestione dello Spazio di Scambio (file o partizione di swap):

Il sistema operativo prevede una partizione speciale o dispositivo separato per lo scambio, un'area della memoria non volatile strutturata in maniera differente dal file system, è una partizione di scambio con file system semplificato (non serve per memorizzare o recuperare informazioni, ma solo per leggere e scrivere), utilizza numeri di blocchi relativi.

Allocazione in Memoria di Scambio:

All'avvio del sistema questa partizione di scambio è vuota ed è rappresentata in memoria come una singola voce che ne indica l'inizio e la dimensione, nella partizione viene riservata una parte di dimensione pari a quella del processo, è gestita come una lista di parti libere (anche se esistono miglioramenti).

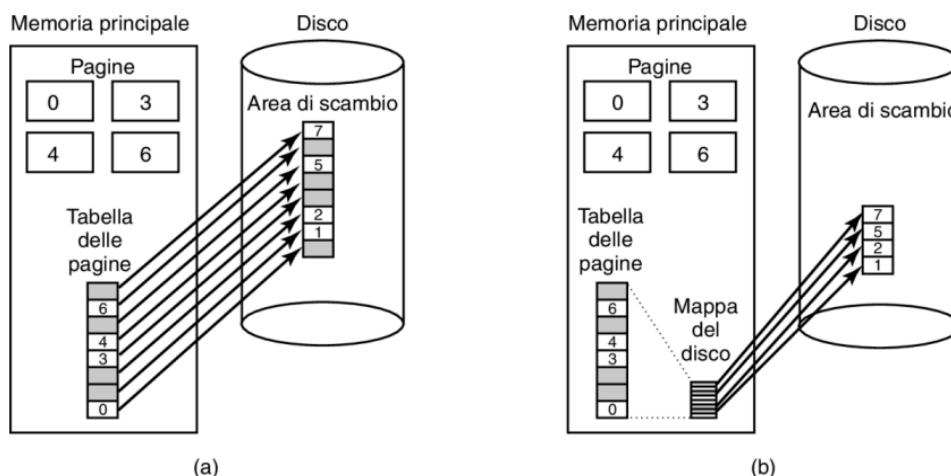
Ogni processo ha un'area di scambio in memoria non volatile, l'indirizzo in cui scrivere una pagina è calcolato sommando: l'offset della pagina nel suo spazio virtuale all'inizio dell'area di scambio.

Bisogna gestire anche il problema di quando i processi crescono, si riservano di aree separate per testo, dati e stack, per gestire l'espansione dei processi.

L'alternativa è allocare lo spazio su memoria non volatile al momento dello scambio di ogni pagina. Lo svantaggio è che per tener traccia di ogni pagina in memoria non volatile è necessario avere un indirizzo del disco in memoria.

Esempi di Gestione Paginazione:

- Paginazione in area di scambio statica. Ogni pagina ha una posizione fissa su disco.
- Salvataggio dinamico delle pagine. Indirizzo su disco scelto al momento dello scambio.



Non è sempre possibile avere partizioni di scambio, perciò si usano file pre-allocati in file system normali. (es. Windows)

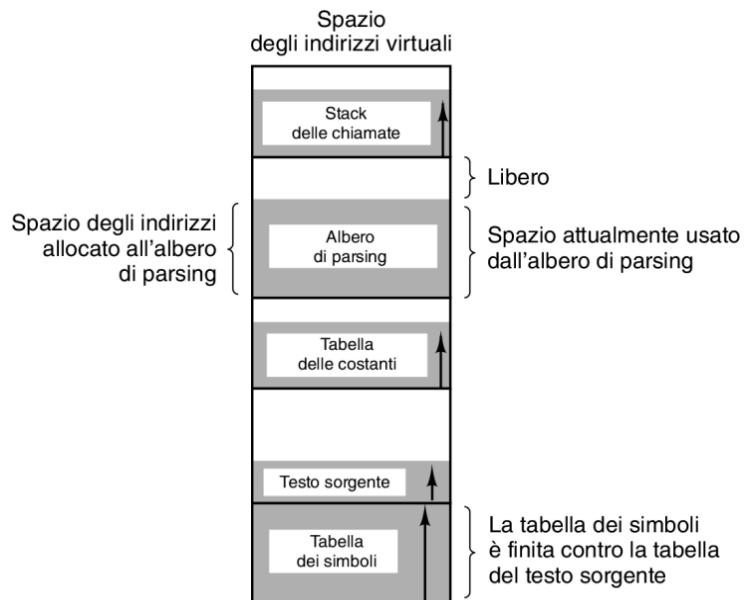
Segmentazione:

In un sistema di memoria monodimensionale, gli indirizzi virtuali vanno da 0 a un massimo e

disposti in modo lineare e contiguo.

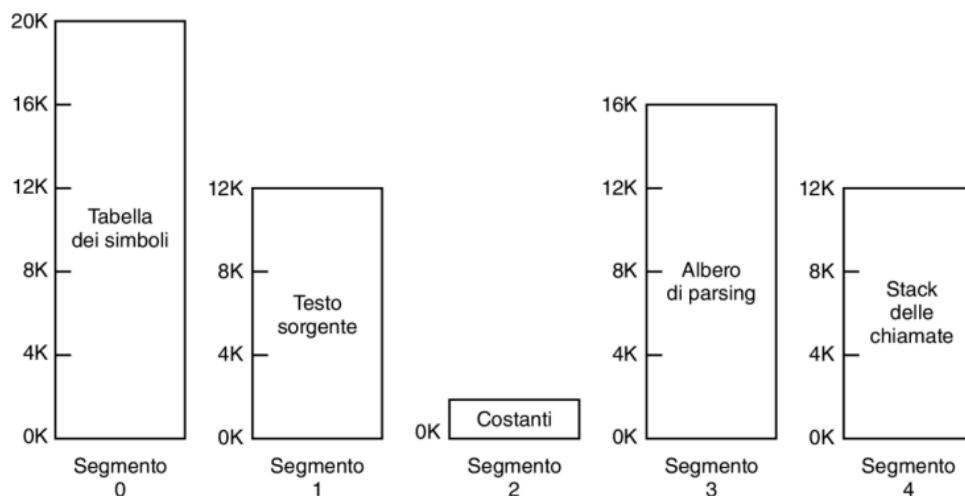
Può risultare problematica in alcuni scenari, come nella compilazione dove diverse tabelle (testo sorgente, tabella dei simboli, costanti, albero di parsing, stack) crescono dinamicamente e in modo imprevedibile.

La crescita di una tabella può causare sovrapposizioni con altre, creando difficoltà nella gestione della memoria richiedendo una riorganizzazione complessa.



La segmentazione introduce l'idea di spazi di indirizzi virtuali multipli e indipendenti, chiamati segmenti. Ciascun segmento ha una sequenza lineare di indirizzi, iniziando da 0 fino a un massimo variabile. I segmenti possono avere lunghezze diverse e la loro dimensione può cambiare durante l'esecuzione.

Questa struttura consente ai segmenti di crescere o ridursi senza interferire l'uno con l'altro. Per specificare un indirizzo in memoria segmentata, si usa un indirizzo a due parti: numero di segmento e indirizzo nel segmento.



Vantaggi della segmentazione:

Flessibilità: I segmenti possono crescere o ridursi in modo indipendente l'uno dall'altro. Ciò elimina il problema di collisione presente nella memoria monodimensionale.

**Semplificazione del Linking:** Se ogni procedura occupa un segmento separato, il linking di procedure diventa molto più semplice. In caso di modifiche, non è necessario aggiornare gli indirizzi di altre procedure non correlate.

**Condivisione e Protezione:** La segmentazione facilita la condivisione di risorse, come librerie condivise, tra processi diversi. Offre anche la possibilità di applicare vari livelli di protezione ai segmenti (es. solo lettura, solo esecuzione), migliorando la sicurezza e aiutando a identificare errori.

### Segmentazione vs Paginazione

**Confronto:** La segmentazione suddivide la memoria in segmenti con indirizzi lineari mentre la paginazione divide la memoria in pagine di dimensioni fisse.

La segmentazione offre maggiore flessibilità e gestione delle strutture dati rispetto alla paginazione, ma può essere più complessa da implementare.

**Esempi Pratici:**

- **Uso nel Compilatore:** Le varie tabelle utilizzate durante la compilazione possono essere allocate in segmenti separati, le tabelle possono crescere indipendentemente e sono gestite più efficacemente.

- **Librerie Condivise:** In un sistema segmentato, una libreria grafica può essere posta in un segmento e condivisa tra più processi, risparmiando spazio e migliorando l'efficienza.

### Paginazione VS Segmentazione:

Il programmatore deve sapere che questa tecnica è in uso? Nella paginazione NO, nella segmentazione SI.

Quanti spazi di indirizzi lineari ci sono? Nella paginazione 1, nella segmentazione Molti.

Lo spazio degli indirizzi totale può superare la dimensione della memoria fisica: Nella paginazione SI, nella segmentazione SI.

Le procedure e i dati possono essere distinti e protetti separatamente? Nella paginazione NO, nella segmentazione SI.

Le tabelle la cui dimensione varia possono essere disposte facilmente? Nella paginazione NO, nella segmentazione SI.

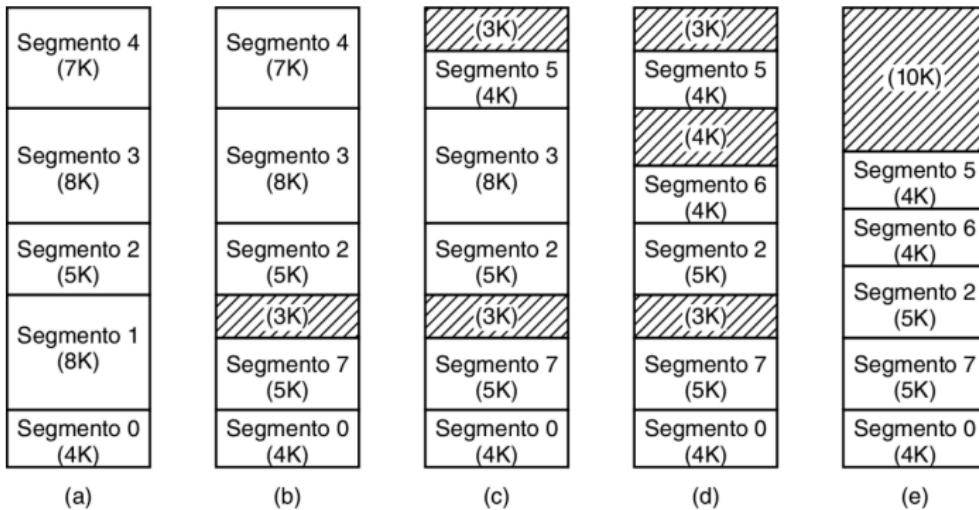
La condivisione delle procedure fra utenti è facilitata? Nella paginazione NO, nella segmentazione SI.

Perché fu inventata questa tecnica? Nella paginazione per avere uno spazio degli indirizzi lineare grande senza dover acquistare ulteriore memoria fisica, nella segmentazione Per consentire a programmi e dati di essere spezzati in spazi degli indirizzi logicamente indipendenti e per facilitare la condivisione e la protezione.

### Evoluzione della Configurazione della Memoria:

Le pagine hanno dimensioni fisse, i segmenti no.

- a) memoria fisica con cinque segmenti.
- b) il segmento 1 è rimosso e il segmento 7, che è più piccolo, viene messo al suo posto. Fra il segmento 7 e il segmento 2 c'è dello spazio inutilizzato, cioè vuoto.
- c) il segmento 4 è sostituito dal segmento 5.
- d) il segmento 3 è rimpiazzato dal segmento 6.
- e) **Frammentazione Esterna ("Checkerboarding"):** Dopo un po', la memoria sarà suddivisa in parti, qualcuna contenente segmenti e altre spazi vuoti. Può essere risolto tramite la compattazione.

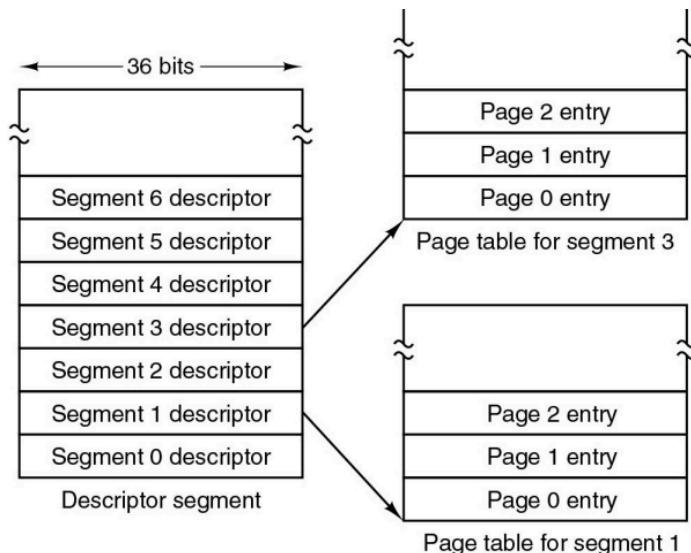


Il primo utilizzo della segmentazione e della paginazione:

Breve storia di MULTICS: Progetto di ricerca del M.I.T. diventato operativo nel 1969, influente fino al 2000, ha avuto un impatto su UNIX, architettura x86 e TLB.

Architettura della Memoria: MULTICS forniva fino a  $2^{18}$  segmenti per programma, con ogni segmento lungo fino a 65.536 parole ( $= 2^{16}$ ).

Approccio alla Memoria Virtuale: I segmenti venivano trattati come spazi di memoria virtuale indipendenti e paginati per gestire meglio lo spazio in memoria. Il Rilevanza storica:segmento dei descrittori punta alle tabelle delle pagine:

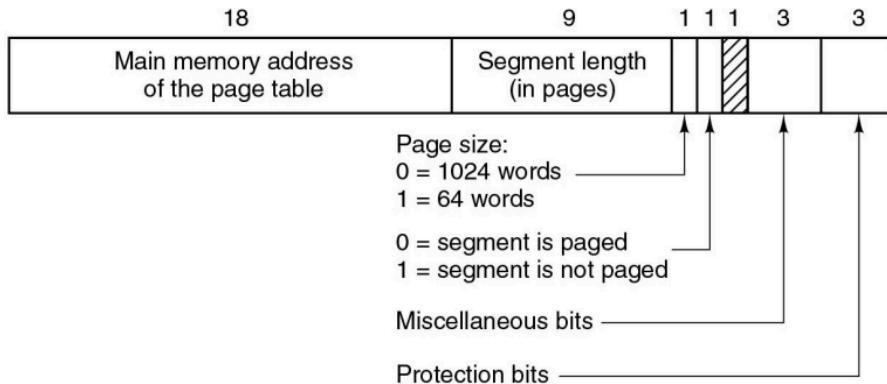


Gestione della memoria e dei segmenti:

Ogni segmento trattato come spazio virtuale indipendente e paginato.

Tabella dei Segmenti: Contenevano descrittori per ogni segmento, indicando se sono in memoria e collegamenti alle tabelle delle pagine.

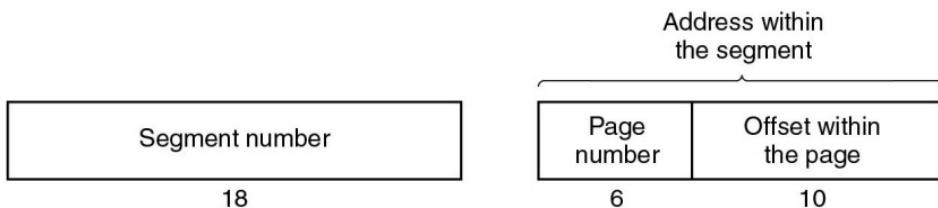
Funzionamento dei Descrittori: Descrittori con puntatori, dimensione del segmento, bit di protezione, e altre informazioni. (figura. Un descrittore di segmento).



Nota che la somma di tutti i campi è 36 (il numero di bit nel descriptor segment).

#### Gestione degli indirizzi:

Indirizzi costituiti da due parti - segmento e indirizzo nel segmento - con suddivisione in numero di pagina e parola nella pagina. (34 bit)



#### Conversione di un indirizzo in Multics:

1. Il numero del segmento è usato per trovare il descrittore del segmento (18 bit).
2. Si verifica se la tabella delle pagine del segmento è in memoria per prevenire eventuali errori.
3. Si esamina la voce della pagina virtuale: Se la pagina non memoria, page fault invece Se in memoria, dalla voce della tabella delle pagine viene estratto l'indirizzo dell'inizio della pagina nella memoria principale (6 bit).
4. Si ottiene l'indirizzo nella memoria principale in cui era localizzata la parola aggiungendo l'offset all'origine della pagina (10 bit).
5. Avviene la lettura o il salvataggio.

Ottimizzazione delle prestazioni del MULTICS tramite l'uso del TLB (Translation Lookaside Buffer): Fu il primo sistema ad utilizzare un TLB con 16 parole per velocizzare la ricerca degli indirizzi. I programmi con set di lavoro minori del TLB raggiungono una maggiore efficienza e gestione degli errori (perché tutto il programma era inserito in TLB).

#### Evoluzione e Declino:

Eredità di MULTICS nell'x86: fino all'x86-64, Intel x86 rifletteva il modello di MULTICS, combinando segmentazione e paginazione, ma presto la segmentazione diventa obsoleta e viene mantenuta solo per compatibilità (troppo complicata da gestire, troppo overhead).

Ragioni del Cambiamento: Sistemi operativi chiave come UNIX e Windows non adottano la segmentazione per questioni di portabilità, inoltre Intel elimina la segmentazione per ottimizzare lo spazio del chip nelle CPU a 64 bit.

## Comandi UNIX:

total: Quantità totale di memoria fisica disponibile.

used: Memoria attualmente in uso.

free: Memoria libera/non utilizzata.

shared: Memoria condivisa (obsoleta, presente solo per compatibilità).

buff/cache: Memoria utilizzata per buffer/cache/slub, recuperabile se necessario.

available: Stima della memoria disponibile per nuove applicazioni, considerando buffer e cache.

-h: human.

-b, --kilo, --mega, --giga per specificare l'unità di misura (byte, kilobyte, megabyte, gigabyte).

-t totale della memoria e dello swap.

-s per aggiornamenti continui ogni *tot* secondi, simile al comando watch.

## FILE SYSTEM

### Problemi di Memorizzazione:

- Limitazioni della RAM: La RAM fisica offre uno spazio limitato per salvare dati, insufficiente per molte applicazioni che richiedono molto più spazio, a volte fino a terabyte.
- Perdita di Dati: Le informazioni in RAM vanno perse al termine del processo o in caso di crash del computer o blackout.
- Accesso Concorrente: La necessità di accesso simultaneo alle informazioni da più processi rende inadeguata la memorizzazione in uno spazio di indirizzi di un unico processo.

Quindi bisogna avere diversi requisiti, tra cui:

Capacità di salvare grandi quantità di informazioni, persistenza delle informazioni oltre la vita del processo che le utilizza e accessibilità delle informazioni da più processi simultaneamente.

### Soluzioni di Memorizzazione:

Uso di Dischi Magnetici e SSD: Tradizionalmente, si utilizzano dischi magnetici e unità a stato solido (SSD) per memorizzazione a lungo termine.

Operazioni sui Dischi: I dischi e gli SSD supportano gli SSD supportano operazioni essenziali come la lettura e la scrittura di blocchi di dati.

File System sono un'astrazione del File: Il file come astrazione risolve il problema di memorizzazione, consentendo la persistenza, l'accesso multiplo, e la gestione di grandi volumi di dati.

Il ruolo dei Sistemi Operativi è quindi gestire i file attraverso il file system, che si occupa della struttura, denominazione, accesso, protezione, implementazione dei file.

Ci sono aspetti visibili all'utente (nomi dei file, operazioni consentite) e aspetti tecnici rilevanti per i progettisti del sistema (gestione della memoria, struttura interna del file system).

I file system sono:

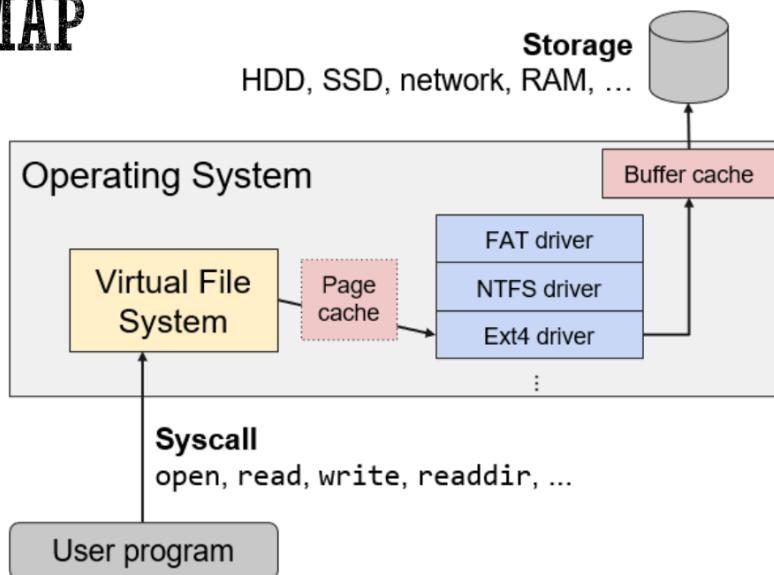
Un modo per organizzare e memorizzare (in modo persistente) le informazioni e un'astrazione sui dispositivi di memorizzazione: Disco rigido, SSD, rete, RAM, ...

Organizzati in file e (tipicamente) directory.

Esempi di file system:

FAT12/FAT16: MS-DOS, NTFS: Windows, Ext4: Linux, APFS: mac OS/iOS

# ROADMAP



Ad una chiamata di sistema del tipo read, verrà fatto riferimento ad un virtual file system (astrazione) e a dei driver, costrutti software che ci permettono di convertire l'astrazione in strutture dati che sono state utilizzate per organizzare le strutture all'interno dello Storage. In mezzo ci saranno diverse cache (circuiteria o virtualizzate).

**File e astrazione:** I file fungono da metodo di astrazione per salvare e leggere informazioni su disco, nascondendo i dettagli tecnici all'utente.

**Nomenclatura dei File:** I file vengono identificati tramite nomi, che possono variare in base al sistema operativo.

**Lunghezza e «Sensibilità» dei Nomi:** Alcuni sistemi operativi limitano la lunghezza dei nomi dei file (es. 8 lettere in MS-DOS) mentre altri supportano nomi più lunghi (255 lettere).

**Caratteri speciali nei nomi dei file:**

FAT12: No "\*" | "<>?!" e altro

Ext4: No 10' e ", o i nomi speciali "." e "..".

**Sono case sensitive:** Sistemi come UNIX distinguono tra maiuscole e minuscole, a differenza di MS-DOS.

**Estensioni di File:** Le estensioni sono parti di nomi di file che seguono un punto, indicano generalmente una caratteristica specifica del file. Esempio: .jpg per immagini JPEG, .mp3 per musica MPEG layer 3).

**Ruolo delle Estensioni:** In alcuni sistemi, le estensioni sono puramente convenzionali e non richieste dal sistema operativo (come in UNIX), mentre in altri (come Windows) hanno un significato specifico, infatti le estensioni dei file sono registrate nel sistema operativo e associate a programmi specifici che si avviano quando l'utente interagisce con il file. Esempio: apertura di un file .docx con Microsoft Word.

Diversi tipi di estensione:

.bak File di backup

.c Programma sorgente in linguaggio C

.gif Immagine in Compuserve Graphical Interchange Format

.html Documento HTML (world wide web hypertext markup language)  
 .jpg Immagine codificata con lo standard JPEG  
 .mp3 Musica codificata in formato audio MPEG layer 3  
 .mpg Filmato codificato in formato audio MPEG standard  
 .o File oggetto (output da compilatore, non ancora linkato)  
 .pdf Documento in formato Adobe PDF (portable document format)  
 .ps File PostScript  
 .tex Input per il programma di formattazione TEX  
 .txt File di testo generico  
 .zip Archivio compresso

Tipologie di struttura dei file:

a) Sequenza Non Strutturata di Byte:

I file sono visti dal sistema operativo come una serie non strutturata di byte. Il significato dei dati è determinato dai programmi a livello utente, non dal sistema operativo. Questo approccio è adottato da sistemi operativi come UNIX, Linux, macOS e Windows, offrendo massima flessibilità.

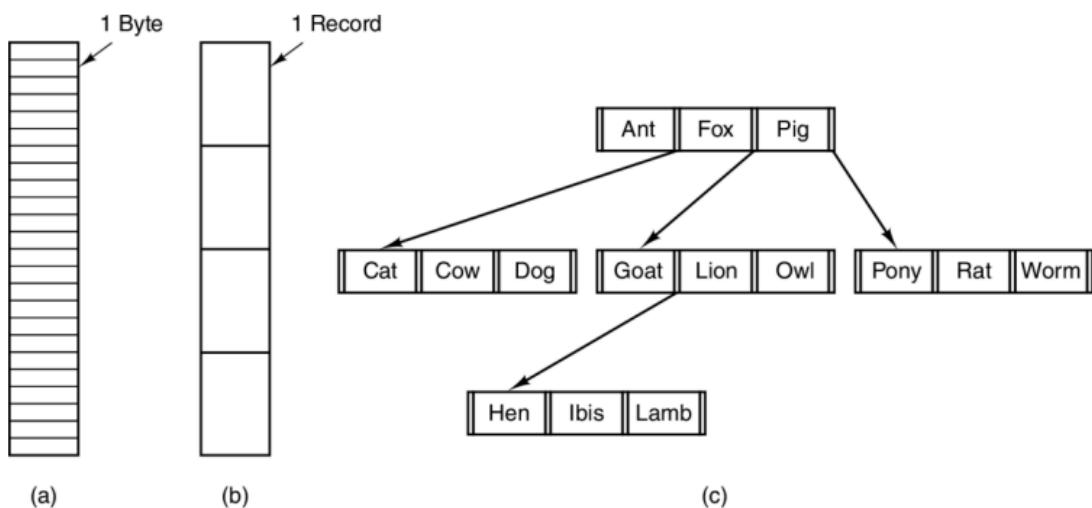
b) Sequenza di Record di Lunghezza Fissa:

Un file è una sequenza di record con una struttura interna definita e lunghezza fissa. Il modello storico basato su schede perforate a 80 colonne in mainframe. Letture e scritture avvengono a unità di record, meno comune nei sistemi operativi moderni ma era prevalente nei mainframe del passato.

c) File come Albero di Record:

Il file è organizzato come un albero di record, con lunghezze variabili e un campo chiave in posizione fissa. L'organizzazione consente ricerche rapide basate su chiavi specifiche.

Utilizzato principalmente in sistemi mainframe per elaborazioni dati di carattere commerciale, diverso dalle sequenze non strutturate di UNIX e Windows.



Tipi di file:

- File e Directory Normali:

Utilizzati in UNIX (inclusi macOS e Linux) e Windows.

File Normali: Contengono informazioni utente e sono la forma più comune.

Directory: File di sistema per mantenere la struttura del file system.

- File Speciali:

File Speciali a Caratteri: Usati per modellare dispositivi seriali di I/O come terminali e stampanti.

File Speciali a Blocchi: Usati per modellare dischi (usati nella memoria swap).

- Tipi di File Normali:

File ASCII: Composti da righe di testo, visualizzabili e stampabili.

File Binari: Non leggibili come testo; hanno una struttura interna conosciuta dai programmi che li utilizzano. Esempi includono file eseguibili e archivi.

File e strutture interne:

a) File Eseguibile:

Sequenza di byte con formato specifico per l'esecuzione.

Intestazione (Header): Contiene un 'numero magico' per identificare il file come eseguibile (così da non non eseguire file «non eseguibili»), dimensioni delle parti del file, indirizzo di esecuzione iniziale e flag.

Testo e Dati: Parti effettive del programma, caricate e rilocate in memoria.

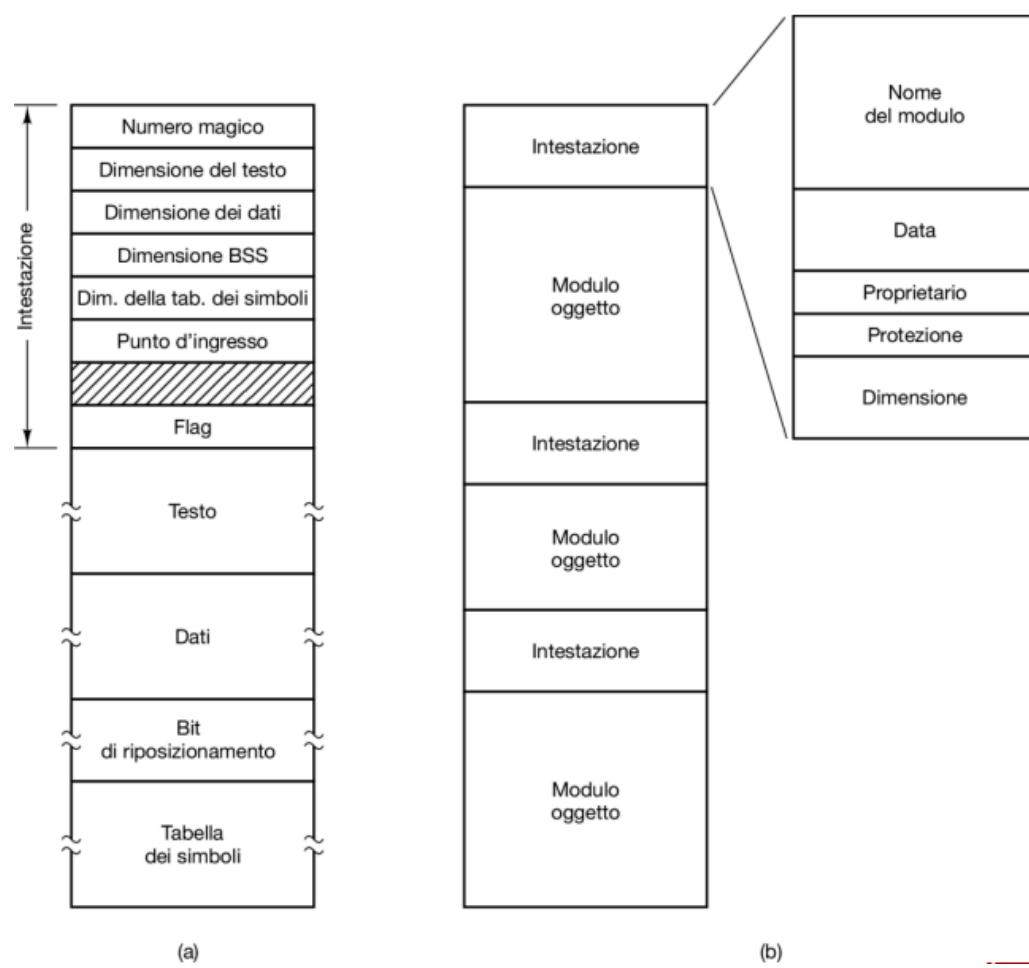
Tabella dei Simboli: Utilizzata per il debug.

b) File di Archivio

Descrizione: Raccolta di procedure di libreria (moduli) compilate ma non collegate.

Intestazioni dei Moduli: Indicano nome, data di creazione, proprietario, codice di protezione e dimensione.

Carattere Binario: Stampare questi file produrrebbe caratteri incomprensibili (come i file zip).



### Riconoscimento dei Tipi di File:

Esempi di File Binari: File eseguibili con intestazioni e dati specifici; archivi con moduli di libreria e intestazioni dettagliate.

Sistemi Operativi e File Tipizzati: Alcuni sistemi, come il vecchio TOPS-20, avevano meccanismi complessi per riconoscere i tipi di file, che potevano portare a limitazioni nell'uso dei file (ad oggi abbandonata questa tecnica).

Utility in UNIX: Il comando 'file' in UNIX usa euristiche per determinare il tipo di file.

### Metodi di accesso ai file:

Accesso Sequenziale: Nei primi sistemi operativi, era l'unico metodo disponibile. Consentiva la lettura dei file dall'inizio alla fine, adatto ai nastri magnetici.

Accesso Casuale: Introdotta con l'avvento dei dischi, permette la lettura di byte o record in qualsiasi ordine, senza seguire una sequenza, cruciale per applicazioni come i sistemi di database, dove è necessario accedere rapidamente a record specifici senza attraversare l'intero file. Per accedere in modo casuale si utilizza la syscall lseek.

### Metodi di Specificazione della Posizione di Lettura:

Lettura con Posizione Specifica: Ogni operazione di lettura inizia da una posizione definita all'interno del file.

Utilizzo di seek: Un'operazione speciale per impostare la posizione corrente nel file, dopo la quale il file può essere letto sequenzialmente dalla posizione impostata.

### Attributi dei file:

Definizione: Oltre a nome e dati, ogni file ha attributi (o metadati) che variano a seconda del sistema operativo.

#### Esempi di Attributi Comuni:

Protezione e Accesso: Indica chi può accedere al file e come (es. proprietario, creatore, password).

Flag Specifici: Diversi flag per controllo (es. sola lettura, file nascosto, file di sistema, file di backup).

Tipologia del File: Indica se il file è ASCII o binario, accesso casuale o sequenziale.

Attributi Temporali: Data e ora di creazione, ultimo accesso, ultima modifica.

Dimensione: Dimensione attuale e massima del file (quella massima serve soprattutto per far assegnare al sistema operativo il file in una porzione più grande di memoria perché potrebbe crescere).

Gestione dei Record (per file basati su record): Lunghezza del record, posizione e lunghezza della chiave.

Importanza: Gli attributi dei file sono cruciali per la protezione, il controllo dell'accesso, la gestione efficace dei file nei sistemi operativi.

Attributo	Significato	Attributo	Significato
Protezione	Chi può accedere al file e in che modalità	Flag temporaneo	0 per normale; 1 per cancellare il file al termine del processo
Password	Password necessaria per accedere al file	Flag di file bloccato	0 per non bloccato; non zero per bloccato
Creatore	ID della persona che ha creato il file	Lunghezza del record	Numero di byte nel record
Proprietario	Proprietario attuale	Posizione della chiave	Offset della chiave in ciascun record
Flag di sola lettura	0 per lettura/scrittura; 1 per sola lettura	Lunghezza della chiave	Numero di byte del campo chiave
Flag di file nascosto	0 per normale; 1 per non visualizzare negli elenchi	Data e ora di creazione del file	Data e ora di quando il file è stato creato
Flag di file di sistema	0 per file normali; 1 per file di sistema	Data e ora di ultimo accesso al file	Data e ora di quando è avvenuto l'ultimo accesso al file
Flag di file archivio	0 per già sottoposto a backup; 1 per file di cui fare il backup	Data e ora di ultima modifica al file	Data e ora di quando è avvenuta l'ultima modifica al file
Flag ASCII/binario	0 per file ASCII; 1 per file binari	Dimensione attuale	Numero di byte nel file
Flag di accesso	0 per accesso sequenziale; casuale 1 per accesso casuale	Dimensione massima	Numero di byte di cui può aumentare il file

### Operazioni su file:

1. Create: Creazione di un file senza dati, principalmente per «annunciare» la presenza del file e impostare alcuni attributi.
2. Delete: Eliminazione di un file per liberare spazio su disco, attraverso una specifica chiamata di sistema.
3. Open: Apertura di un file per consentire al sistema di caricare in memoria gli attributi e gli indirizzi del disco, facilita l'accesso rapido in seguito. L'apertura di un file restituisce un handle (descrittore di file) per le operazioni future. Errori: ENOENT o EBADF.
4. Close: Chiusura del file al termine degli accessi per liberare spazio nelle tabelle interne, forza anche la scrittura dell'ultimo blocco del file.
5. Read: Lettura dei dati da un file, generalmente dalla posizione corrente, con specificazione della quantità di dati richiesti e fornitura di un buffer per la loro memorizzazione.
6. Write: Scrittura di dati nel file, tipicamente alla posizione corrente, può comportare l'ampliamento del file o la sovrascrittura dei dati esistenti.
7. Append: Aggiunta di dati solo alla fine del file, usata in alcuni sistemi operativi come forma limitata di scrittura.
8. Seek: Riposizionamento del puntatore del file su una posizione specifica per file ad accesso casuale, permettendo la lettura o la scrittura da quella posizione. Sposta il puntatore del file di n byte in avanti dalla posizione corrente nel file.
9. Get Attributes: Lettura degli attributi di un file, necessaria per alcuni processi per svolgere le loro funzioni (es. il programma UNIX make per la gestione dei progetti software).
10. Set Attributes: Modifica degli attributi di un file da parte dell'utente, come la modalità di protezione o altri flag, dopo la creazione del file.
11. Rename: Ridenominazione di un file, utilizzata come alternativa al processo di copia ed eliminazione del file originale, specialmente utile per file di grandi dimensioni.

### Altre operazioni:

Rimuovere file: unlink("foo.txt");

Rinominare file: rename("foo.txt", "bar.txt");

Cambiare i file permission attribute: chmod("foo.txt", 0755);

Cambiare la proprietà del file: chown("foo.txt", uid, gid);

## Directory:

Le directory sono file che tengono traccia degli altri file all'interno di un file system.

### Sistemi di Directory a Livello Singolo:

Struttura Semplice: Una singola directory, talvolta chiamata root directory, contiene tutti i file.

Vantaggi: Semplicità e rapidità nella localizzazione dei file.

Utilizzati nei primi computer ed ad oggi nei dispositivi embedded grazie al loro basso costo: concetti semplici di file system sono ancora utili in dispositivi come fotocamere digitali o riproduttori MP3 e tecnologie RFID come chip RFID, carte di credito e tessere di trasporto.

### Limiti dei Sistemi a Singolo Livello:

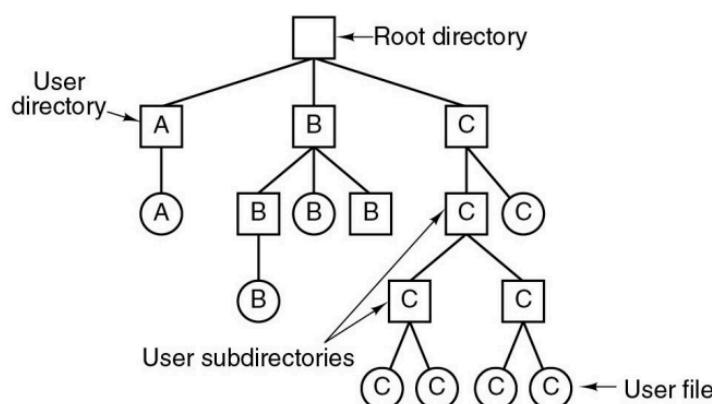
Non pratici per utenti con migliaia di file, c'è difficoltà nel rintracciare i file in un unico spazio.

### Introduzione della Gerarchia:

Organizzazione dei file in gruppi correlati mediante directory ramificate. Struttura ad albero per separare e organizzare logicamente i file. Ogni utente può avere una directory principale privata in ambienti condivisi come reti aziendali.

### Importanza nei File System Moderni:

Tutti i file system moderni utilizzano una struttura gerarchica per la loro flessibilità e potenziale organizzativo. Primo utilizzo di questi file system nei '60 con il MULTICS.



I nomi dei file diventano quindi il nome dei percorsi che possono essere:

Percorsi Assoluti: Percorsi che iniziano dalla directory principale e conducono al file, unici per ogni file (iniziano sempre con il separatore '/') (es. /usr/ast/mailbox).

Percorsi Relativi: Basati sulla directory di lavoro (directory corrente) dell'utente. (es. mailbox).

## Directory di Lavoro (Working Directory):

Cambia dinamicamente per ciascun processo. Non influisce sugli altri processi o sul file system dopo l'uscita del processo.

### Procedure di Libreria:

Evitano di cambiare la directory di lavoro o la ripristinano dopo il loro uso.

### Voci Speciali:

. (punto): Rappresenta la directory corrente, .. (punto punto): Rappresenta la directory genitore.

## Operazioni di Base:

create: Creazione di una directory vuota con le voci '.' e '..' predefinite.

**delete:** Eliminazione di una directory, possibile solo se la directory è vuota.

**opendir:** Apertura di una directory per la lettura del suo contenuto.

**closedir:** Chiusura di una directory dopo la lettura per liberare risorse.

**Lettura e Modifica:**

**readdir:** Restituisce la prossima voce in una directory aperta senza esporre la struttura interna.

**rename:** Rinomina di una directory, simile al rinomino di un file.

**Linking e Unlinking:**

**link:** Crea un hard link, collegando un file esistente a un nuovo percorso, condividendone l'i-node (punta direttamente alla risorsa).

**unlink:** Rimuove una voce di directory, cancellando il file se è l'unico link.

**Link Simbolici:**

Varianti dei hard link che possono puntare a file su dischi o computer diversi. Rappresentano un file tramite un riferimento indiretto che il file system risolve all'uso. (punta ad un riferimento che punta all'i-node). I link simbolici offrono flessibilità oltre i limiti dei dischi ma possono essere meno efficienti rispetto agli hard link.

**Creazione di archivi:**

**File TAR:** TAR (Tape Archive) è un formato usato per raccogliere più file e cartelle in un unico archivio, mantenendo la struttura e i permessi originali. Utilizzato comunemente per raggruppare file correlati per backup, trasferimento o archiviazione.

**Comprimere con GZ**

Dopo l'archiviazione con tar, l'archivio viene compresso con gzip per ridurre lo spazio su disco. gzip è un algoritmo di compressione che riduce efficacemente la dimensione del file senza perdita di dati.

**TAR e GZ:**

**Creazione di un Archivio tar.gz**

**Comando di Base:** tar -czvf nome-archivio.tar.gz /percorso/della/cartella

- c: crea un nuovo archivio.

- z: comprime l'archivio usando gzip.

- v: visualizza un output verboso.

- f: specifica il nome del file di archivio.

**Estrazione di un Archivio tar.gz**

**Comando di Base:** tar -xzvf nome-archivio.tar.gz

- x: estrae il contenuto dall'archivio.

- z: decomprime l'archivio usando gzip.

- v: visualizza un output verboso.

- f: specifica il nome del file di archivio.

In realtà viene creato un file con il comando tar e poi compresso con gzip. Nulla vieta di comprimere con gzip un qualsiasi file.

**ZIP/UNZIP**

**ZIP** è un formato di compressione che riduce la dimensione dei file singolarmente prima di archiviarli insieme. **UNZIP** è utilizzato per decomprimere e estrarre i file dagli archivi ZIP.

**Vantaggi:** Compatibilità ampia con diversi sistemi operativi, compressione individuale dei file.

**TAR.GZ:**

TAR raccoglie molti file in un unico archivio, poi GZ (gzip) comprime l'intero archivio.

Vantaggi: Elevata compressione, conservazione della struttura delle directory e dei permessi dei file.

#### Confronto tra ZIP e TAR.GZ

Efficacia di Compressione: TAR.GZ tende ad avere un tasso di compressione più alto, specialmente per archivi di grandi dimensioni.

Velocità: ZIP può essere più veloce nella compressione di file individuali e più comunemente supportato.

#### Come memorizzare i file?

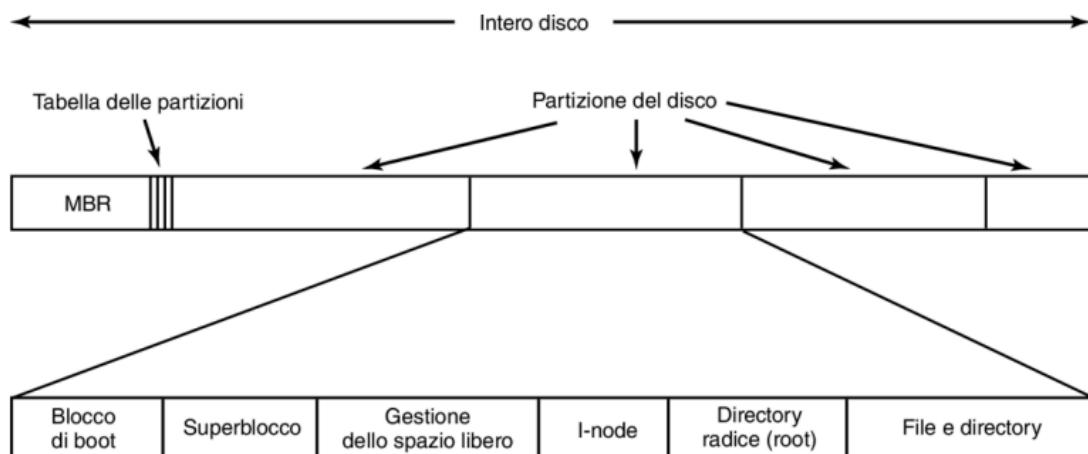
Definizione: Il file system è il metodo utilizzato per organizzare e memorizzare dati sui dispositivi di memoria non volatile (dischi/SSD). Fornisce un modo strutturato per gestire informazioni come file e directory su dispositivi di memoria. Un disco può essere suddiviso in più partizioni, ciascuna con un proprio file system indipendente. I metodi di strutturazione del file system variano a seconda dell'epoca del computer, influenzando come i dati vengono gestiti e acceduti.

#### Vecchia scuola - BIOS con MBR (Master Boot Record)

MBR nel BIOS: è situato nel settore 0 del disco, l'MBR è essenziale per l'avvio del computer, contiene la tabella delle partizioni con dettagli su inizio e fine di ciascuna partizione e identifica la partizione attiva da cui avviare il sistema.

Processo di Avvio: Il BIOS legge l'MBR per trovare la partizione attiva e carica il boot block della partizione attiva per avviare il sistema operativo.

Layout del File System: Ogni partizione inizia con un boot block, seguito da vari elementi di sistema: superblocco, gestione dello spazio libero, i-node, directory root e file e directory.

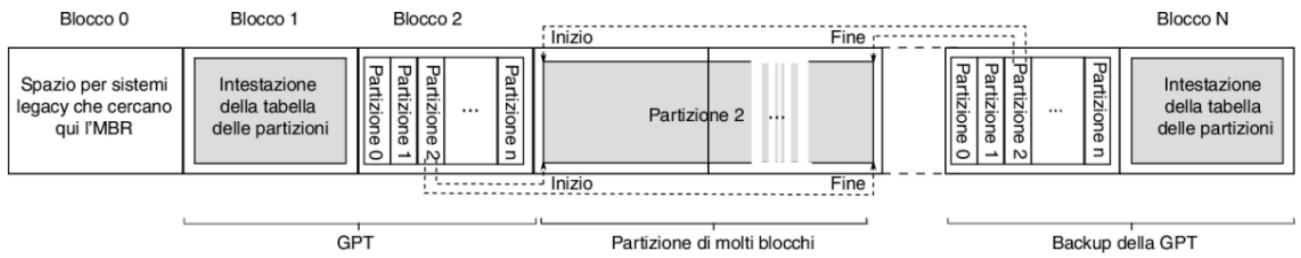


#### Nuova scuola - UEFI (Unified Extensible Firmware Interface)

UEFI: supera le limitazioni del BIOS tradizionale (max 4 partizioni primarie e 2TB per partizione), supportando dischi di dimensioni maggiori e avvio più veloce, introduce così una maggiore flessibilità e compatibilità con diverse architetture hardware.

GPT (GUID Partition Table): UEFI utilizza la GPT per gestire informazioni più dettagliate sulle partizioni, supporta dischi fino a 8 ZiB e contiene un backup nell'ultimo blocco del disco.

All'interno di UEFI troviamo l'EFI System Partition: Utilizza un file system FAT per memorizzare i programmi di avvio, il firmware UEFI legge la GPT e carica i file dalla partizione EFI per avviare il sistema.



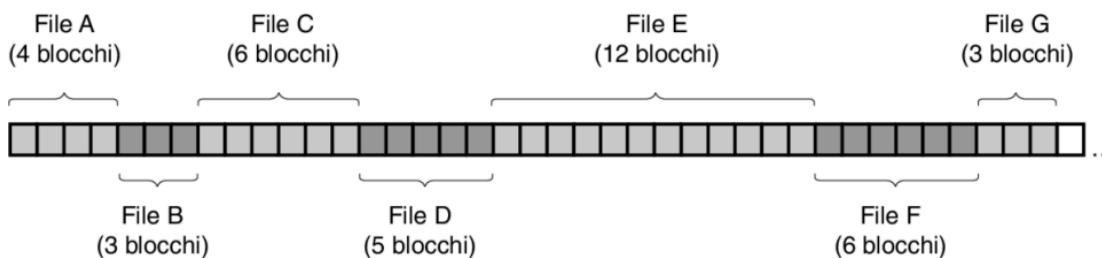
### Implementazione dei file nei file system:

**Obiettivo Principale:** Gestire l'associazione tra i file e i blocchi del disco su cui sono memorizzati, fondamentale per assicurare l'integrità, l'accesso efficiente e la gestione dello spazio su disco. Diversi sistemi operativi adottano approcci differenti per questa associazione: indici, liste concatenate, bitmap, strutture ad albero.

### Allocazione contigua nel file system:

**Concetto di Allocazione Contigua:** I file sono memorizzati come sequenze contigue di blocchi sul disco, es: un file di 50 KB su un disco con blocchi da 1 KB occupa 50 blocchi consecutivi.

**Implementazione e Vantaggi:** Semplice da implementare: richiede solo l'indirizzo del primo blocco e il numero totale di blocchi, ha un'alta efficienza di lettura: l'intero file può essere letto in una sola operazione, senza ritardi (la testina del disco una volta posizionata sul primo blocco legge n blocchi in sequenza andando avanti).



### Problemi e Limitazioni:

**Frammentazione del Disco:** Col passare del tempo, i dischi si frammentano a causa della rimozione di file.

**Gestione dello Spazio Libero:** Richiede una lista di spazi liberi e la conoscenza della dimensione finale dei nuovi file, ma si hanno problemi nella previsione della dimensione del file (dato che possono anche crescere nel tempo) e nella ricerca di spazi adeguati.

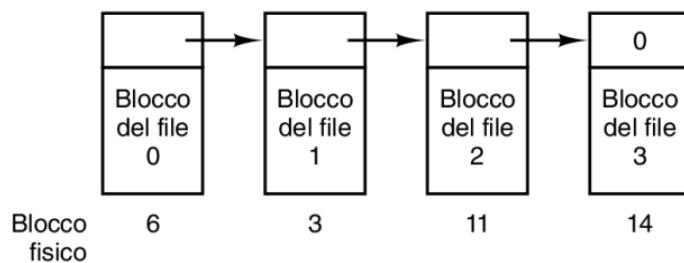
**Implicazioni Pratiche:** Difficoltà nell'aggiungere nuovi file in un disco frammentato, necessita di compattazione del disco (defrag per Windows) o di gestire in modo intelligente dello spazio libero.

### Allocazione con Liste concatenate:

**Principio di Allocazione:** I file sono organizzati come liste concatenate di blocchi su disco, ogni blocco contiene una parte di dati e un puntatore al blocco successivo.

**Gestione dello Spazio:** Efficiente utilizzo di tutti i blocchi disponibili sul disco e minima frammentazione esterna: riduce lo spreco di spazio non utilizzato (dato che utilizza i puntatori).

**Struttura delle Voci di Directory:** Ogni voce di directory traccia solo l'indirizzo del primo blocco di un file, il percorso completo di un file è costruito seguendo i puntatori da un blocco all'altro.



#### Problemi e Limitazioni:

##### Accesso ai Dati:

- Accesso Sequenziale: Leggere un file è efficiente, procedendo blocco per blocco.
- Accesso Casuale (seek): Estremamente lento, richiede comunque la lettura sequenziale di ogni blocco precedente.

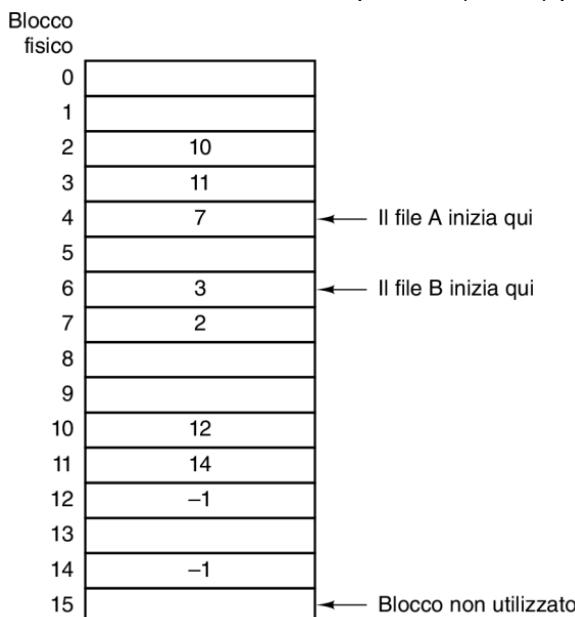
**Dimensione dei Blocchi e Efficienza:** Ogni blocco ha una dimensione effettiva ridotta a causa dello spazio occupato dai puntatori, inoltre le letture e scritture di dimensioni standard (potenze di due) possono essere meno efficienti.

**Implicazioni Pratiche:** Questo metodo offre un'elevata efficienza nello sfruttamento dello spazio su disco ma introduce complessità e rallentamenti nelle operazioni di accesso casuale, è adatto per file a cui si accede principalmente in modo sequenziale.

#### Allocazione con Liste Concatenate usando FAT:

**Ottimizzazione dell'Allocazione a Liste Concatenate:** Eliminazione degli svantaggi dell'allocazione a liste concatenate spostando i puntatori in una tabella di memoria (FAT - File Allocation Table), ogni blocco del disco è rappresentato come una voce nella FAT in memoria RAM. Contiene la sequenza dei blocchi di ciascun file.

es: File A utilizza i blocchi 4, 7, 2, 10, 12; File B i blocchi 6, 3, 11, 14, le sequenze sono terminate da un indicatore speciale (es. -1) per marcare la fine.



**Vantaggi della FAT:** L'intero blocco è disponibile per i dati, ottimizzando lo spazio, e l'accesso casuale è semplificato dato che la sequenza dei blocchi è interamente in memoria.

#### Problemi e Limitazioni:

**Gestione in Memoria:** La FAT deve essere mantenuta interamente in memoria principale e richiede una quantità significativa di memoria: per un disco da 1 TB con blocchi da 1 KB, la FAT richiederebbe fino a 3 GB di RAM.

**Implicazioni di Efficienza:** Lo spazio e la velocità influenzano la dimensione della voce della FAT (da 3 a 4 byte per voce), l'approccio non è ottimale per dischi di grandi dimensioni a causa dell'elevato consumo di memoria.

**Utilizzo Pratico:** Originariamente implementato in MS-DOS, ancora supportato da Windows e UEFI è comunemente usato in dispositivi portatili come schede SD in fotocamere, lettori musicali e altri dispositivi elettronici.

#### I-node (Index Node) nei File System UNIX-like:

Si è deciso di non caricare in memoria file non aperti, e che la nomenclatura del file e il file stesso sono indipendenti.

L'I-node è una struttura dati fondamentale nei file system come ext2/ext3/ext4 in Linux.

Contiene tutte le informazioni su un file, esclusi il nome e il contenuto, include metadati come permessi, proprietario, timestamp e indirizzi dei blocchi di dati. Ogni file e directory è rappresentato da un I-node univoco, indicizzato in una tabella di I-node.

#### Confronto con FAT (File Allocation Table):

**Gestione dei File:** FAT si basa su una tabella di allocazione per tracciare i file, mentre i sistemi basati su I-node utilizzano una tabella di I-node che separa le informazioni del file dalla sua posizione fisica sul disco.

**Informazioni sui File:** FAT fornisce meno dettagli sui file, concentrandosi principalmente sull'allocazione dello spazio, i sistemi I-node offrono una gestione più dettagliata dei metadati, inclusi permessi e proprietà.

**Efficienza e Performance:** I file system basati su I-node tendono a essere più efficienti e performanti, specialmente su dischi di grandi dimensioni, grazie alla loro struttura avanzata.

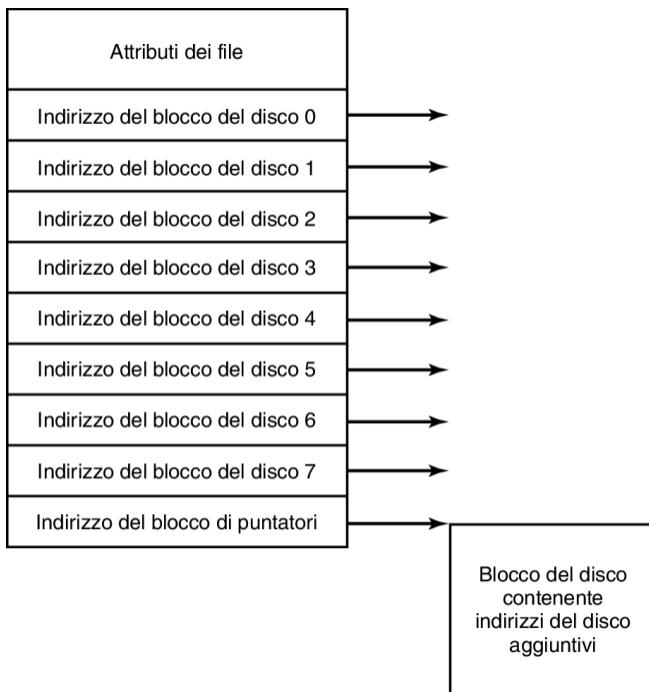
#### Allocazione con I-node (Index-Node):

Gli I-node sono strutture dati che elencano gli attributi e gli indirizzi dei blocchi dei file, ogni I-node rappresenta un file, fornendo un metodo efficiente per trovare tutti i suoi blocchi di dati.

**Efficienza della Memoria con I-node:** Solo gli I-node dei file aperti sono mantenuti in memoria, riducendo significativamente l'utilizzo della memoria e, l'array degli I-node in memoria è proporzionale al numero di file aperti, non alla dimensione del disco (se ho una cartella con 90mila file e uso il comando ls dovrà caricare in memoria tutti gli I-node riempiendo la RAM).

**Esempio e Struttura:** Ogni I-node ha una dimensione fissa e contiene informazioni quali dimensione del file, permessi, proprietario, e indirizzi dei blocchi di dati.

**Gestione File di Grandi Dimensioni:** Gli I-node hanno uno spazio limitato per gli indirizzi del disco, per file che superano il limite l'ultimo indirizzo nell'I-node punta a un blocco contenente ulteriori indirizzi di blocchi di dati dello stesso file, questo sistema permette di gestire file di dimensioni molto grandi con efficacia.



**I-node nei File System UNIX e Windows NTFS:** Gli I-node sono un concetto fondamentale in UNIX e nei suoi file system derivati. NTFS, il file system di Windows, utilizza una struttura simile con I-node più grandi che possono contenere file di piccole dimensioni all'interno dell'I-node stesso.

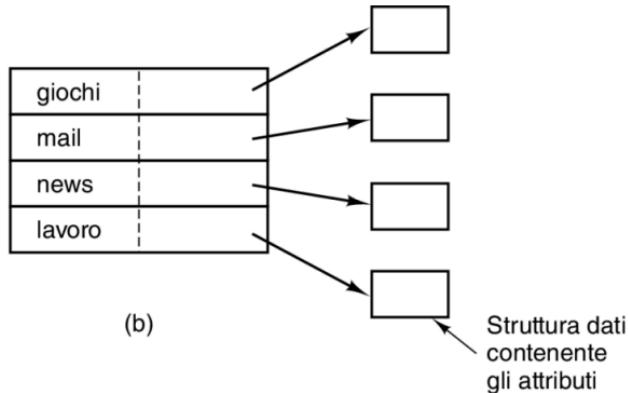
Come implementare le directory?

**Funzione Principale:** Le directory “mappano” i nomi ASCII dei file sulle informazioni necessarie per localizzare i dati su disco.

**Metodi di Allocazione:** Variano a seconda del sistema operativo, includendo indirizzi di blocchi contigui, il primo blocco nelle liste concatenate, o i numeri degli I-node.

giochi	attributi
mail	attributi
news	attributi
lavoro	attributi

(a)



a) Una semplice directory contenente voci a dimensione fissa con gli indirizzi del disco e gli attributi nella voce della directory.

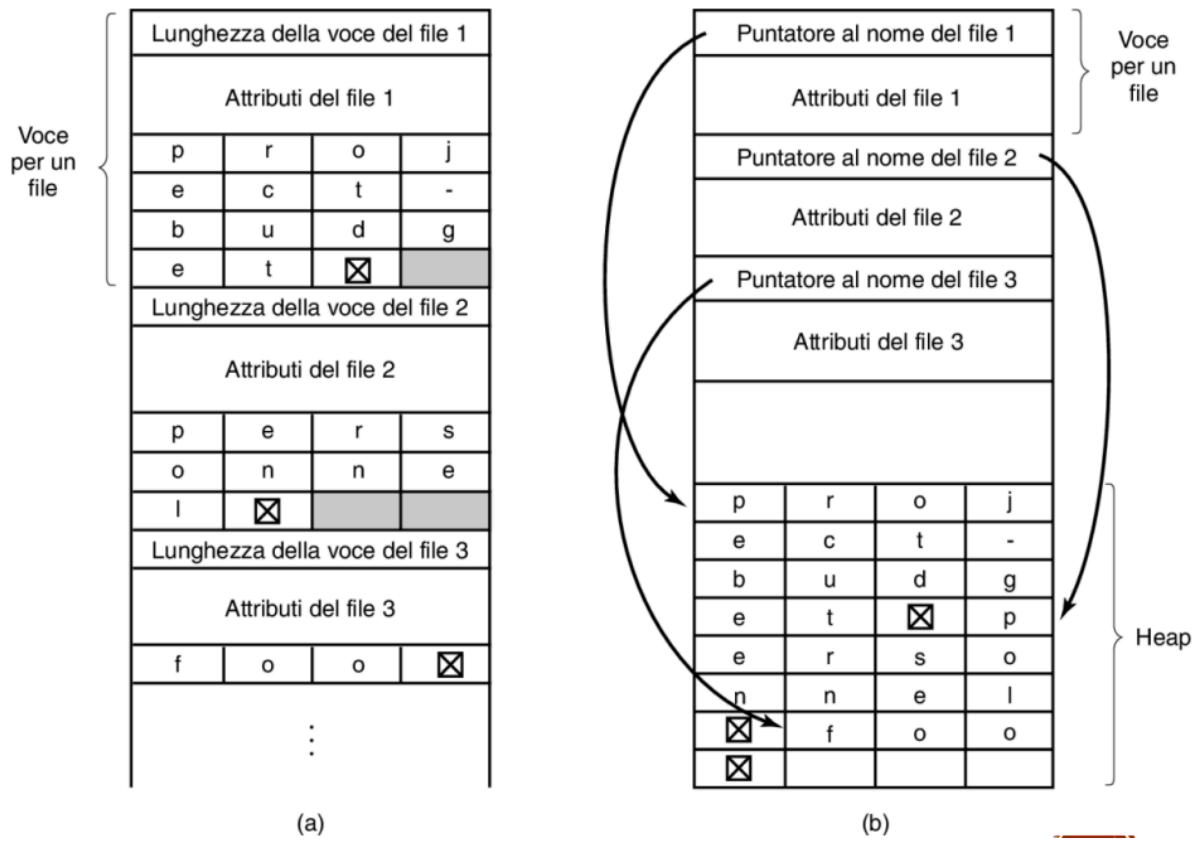
b) Una directory in cui ogni voce fa soltanto riferimento a un I-node.

**Nomi di File Variabili:** Supporto per nomi di file di lunghezza variabile, limite tipico (255).

**Strutture di Directory:**

a) Voci di Directory di Lunghezza Variabile: l'header di lunghezza fissa seguito dal nome del file.

b) Gestione degli Heap: le voci di directory di lunghezza fissa con nomi dei file gestiti in uno heap separato.



**Efficienza e Limitazioni:** gestiscono i nomi di lunghezza variabile ma presentano problemi, nella gestione degli spazi vuoti è sempre presente la frammentazione quando un file viene cancellato (quando verrà cancellata un directory con un nome lungo 7 non potrà rimpiazzarla con una cartella con un nome lungo 255).

Ottimizzazione della ricerca:

**Ricerca Lineare Tradizionale:** Inizialmente, i file in una directory venivano cercati linearmente dall'inizio alla fine, questo metodo può diventare lento in directory con un gran numero di file. Si usano quindi delle Tabelle di Hash: Introducendo le tabelle di hash in ogni directory per accelerare il processo di ricerca, il nome di un file è sottoposto a hashing per generare un indice nell'intervallo da 0 a n-1, la voce corrispondente nella tabella di hash indica il punto di partenza per la ricerca del file.

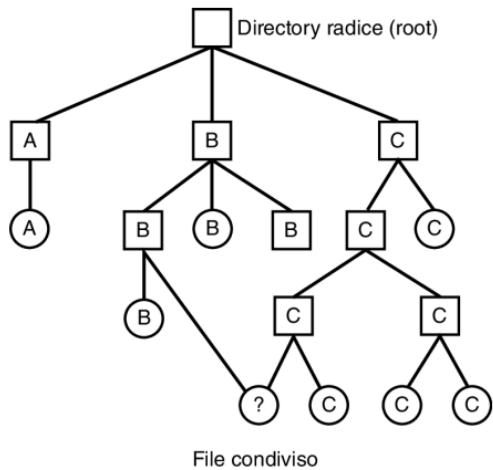
**Gestione delle Collisioni:** Si creano liste concatenate per gestire più file che condividono lo stesso valore hash. La ricerca verifica tutte le voci nella catena per trovare il file desiderato. È presente il Caching delle Ricerche: si salvano i risultati di ricerche comuni nella cache per un accesso rapido, prima di avviare una ricerca, si verifica se il file si trova nella cache.

**Vantaggi e Limitazioni:** La cache aumenta l'efficienza delle ricerche, specialmente per file frequentemente richiesti, efficace quando la maggior parte delle ricerche riguarda un numero limitato di file.

**Complessità Amministrativa:** L'uso di tabelle di hash e cache introduce una maggiore complessità nella gestione delle directory, è più adatto a sistemi con directory molto estese, dove si prevede un elevato numero di file.

## Link nei File System:

**File Condivisi:** Essenziali in ambienti collaborativi per permettere a più utenti di lavorare sugli stessi file.



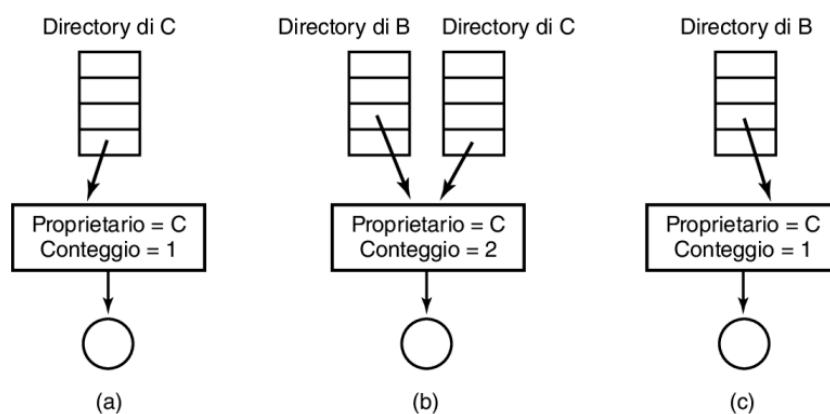
Tipi di Link (comando ln -s):

- Hard Link: Puntano direttamente all'I-node di un file condiviso.
  - Link Simbolico (Soft Link): Puntano al nome di un file piuttosto che all'I-node.

Gestione degli Hard Link: Un file con hard link viene rimosso solo quando non ci sono più riferimenti ad esso, è presente una notevole efficienza di spazio: una sola voce di directory per ciascun hard link e sono ideali per la gestione di file condivisi tra più proprietari.

Vantaggi degli Hard Link: usano un solo I-node indipendentemente dal numero di link, e il file rimane accessibile finché almeno un hard link è presente.

**Problemi e Limitazioni:** Il file permane fino all'eliminazione di tutti gli hard link, potenzialmente causando confusione sulla proprietà del file.



- a) Prima del link. b) Dopo il link. c) Dopo che il proprietario originale elimina il file.

Vantaggi dei Link Simbolici: possono riferirsi a nomi di file oltre i confini del file system e su macchine remote ma sono meno efficienti in termini di spazio: richiedono un I-node per ogni link simbolico.

**Problemi e Limitazioni:** Link simbolici diventano invalidi alla rimozione del file originale, overhead maggiore nella risoluzione del percorso rispetto agli hard link e gestione più complessa, ma con benefici in termini di flessibilità e organizzazione.

**Problemi Comuni:** I file con più percorsi possono essere processati più volte da programmi

di backup o di ricerca, con il rischio di duplicazione dei file su unità di backup. Software avanzati riescono a gestire correttamente i file condivisi e i loro link.

#### Come gestire lo spazio su disco?

**Memorizzazione dei File:** I file sono generalmente memorizzati su disco, e ci sono due modi principali per farlo:

- **Allocazione Contigua:** Richiede spostamenti di file se le loro dimensioni aumentano, simile alla gestione della memoria con segmentazione.
- **Blocchi Non Contigui:** I file vengono spezzettati in blocchi di dimensioni fisse, consentendo una maggiore flessibilità e un migliore utilizzo dello spazio su disco. La scelta della dimensione dei blocchi è un compromesso tra spazio ed efficienza, la dimensione comune di 4 KB è un compromesso tra lo spazio su disco e le prestazioni di trasferimento dei dati.

**Prestazioni di Trasferimento Dati:** I dischi magnetici con blocchi più grandi consentono il trasferimento di più dati per operazione di lettura/scrittura ma blocchi grandi portano a spreco di memoria.

**Efficienza dello Spazio:** Blocchi più piccoli minimizzano lo spreco di spazio con file piccoli ma significa distribuire la maggior parte dei file su più blocchi e incorrere in più ricerche e ritardi (addirittura di rotazione nei dischi) per leggerli.

L'efficienza dello spazio diminuisce con l'aumento della dimensione dei blocchi (oltre la dimensione media dei file). Le prestazioni migliori richiedono blocchi più grandi, ma ciò può comportare uno spreco maggiore di spazio su disco, la scelta ottimale della dimensione del blocco deve bilanciare il tempo di trasferimento e l'efficienza dello spazio. In genere 4 KB.

#### Dischi Magnetici vs Memoria Flash:

**Dischi Magnetici:** La scelta delle dimensioni dei blocchi è influenzata dal tempo di ricerca e dal ritardo di rotazione, con l'aumento della dimensione dei blocchi, si incrementa la velocità di trasferimento ma si riduce l'efficienza dello spazio.

**Memoria Flash:** Diversamente dai dischi magnetici, la memoria flash può avere sprechi di spazio sia con blocchi grandi che piccoli a causa delle dimensioni fisse delle pagine flash.

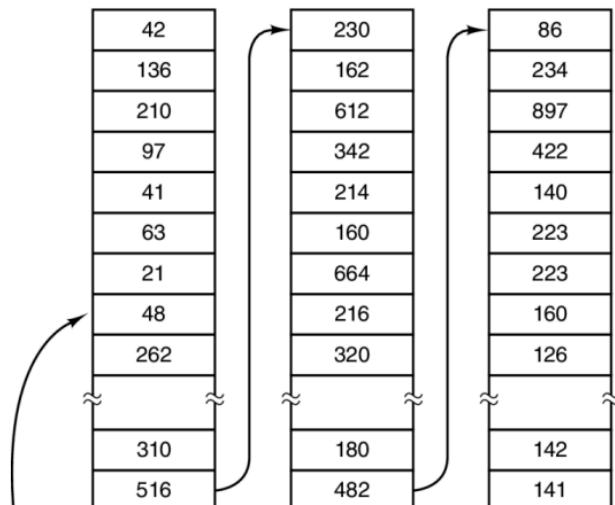
**Tendenze Attuali:** Con l'aumento della capacità dei dischi (TB), potrebbe essere vantaggioso considerare blocchi più grandi per accettare una minore efficienza dello spazio in cambio di prestazioni migliori.

#### Come tenere traccia dei blocchi liberi?

- **Metodo 1: Lista Concatenata:**

Utilizzo di una lista concatenata di blocchi del disco, nella lista si tengono conto solo dei blocchi liberi e si usano gli stessi blocchi del disco liberi per ospitare le liste. Ogni blocco contiene numeri di blocchi del disco liberi. Esempio: Con blocchi da 1 KB e numeri da 32 bit, ogni blocco lista può contenere numeri di 255 blocchi liberi (1 blocco riservato al puntatore del blocco successivo). Per 1 TB, servono 4 milioni di entrature, ma più cresce la memoria occupata sul disco più decresce quella utilizzata per memorizzare queste liste.

**Efficienza:** Richiede meno spazio solo se il disco è quasi pieno, perché ci sono meno liste.



Un blocco del disco da 1 KB può contenere 256 numeri di blocchi del disco da 32 bit

(a)

- Metodo 2: Bitmap:

Utilizzo di una bitmap per tracciare i blocchi liberi, un bit per ogni blocco del disco, 1 libero, 0 allocato. Esempio: Per un disco da 1 TB, serve una bitmap da 1 miliardo di bit.

Efficienza: Richiede meno spazio rispetto alla lista concatenata, tranne in dischi quasi pieni dove la lista concatenata occupa meno spazio.

1001101101101100
011011011110111
1010110110110110
0110110110111011
1110111011101111
1101101010001111
0000111011010111
1011101101101111
1100100011101111
~
0111011101110111
1101111011101111

Una bitmap

#### Ottimizzazione della lista concatenata:

Modifiche alla Lista dei Blocchi Liberi: Tracciamento di serie di blocchi consecutivi anziché blocchi singoli, a ciascun blocco può essere associato un conteggio a 8, 16 o 32 bit, che rappresenta il numero di blocchi liberi consecutivi. Nell'ipotesi migliore, un disco fondamentalmente vuoto è rappresentato da due numeri: l'indirizzo del primo blocco libero seguito dal conteggio dei blocchi liberi.

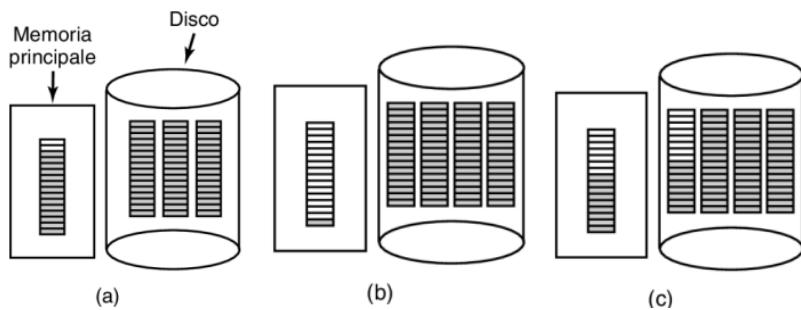
Efficienza: Migliore per dischi quasi vuoti; meno efficiente per dischi frammentati, se il conteggio viene memorizzato c'è overhead eccessivo dovuto a indice e conteggio.

Nella Progettazione di Sistemi Operativi bisogna scegliere la struttura dati ottimale senza dati anticipati sull'uso del sistema.

Gestione dei blocchi liberi con lista di puntatori:

La gestione dei blocchi liberi può utilizzare una lista concatenata di puntatori, nota come "free list", solo un blocco di puntatori è mantenuto in memoria contemporaneamente, ottimizzando così l'utilizzo della memoria. Quando si libera un file, i blocchi necessari vengono allocati dai puntatori disponibili nel blocco in memoria.

Questo metodo evita I/O su disco inutili mantenendo una lista di blocchi liberi direttamente accessibili in memoria, al riempimento del blocco di puntatori in memoria, un nuovo blocco viene letto da disco per proseguire con le operazioni.



- a) un blocco quasi pieno di puntatori a blocchi del disco liberi in memoria e tre blocchi di puntatori a blocchi del disco liberi in disco.
- b) situazione dopo aver liberato 3 blocchi: il blocco riempito viene caricato su disco e, quello contenente puntatori a blocchi liberi ma non riempito viene lasciato in memoria.
- c) La presenza di file temporanei può portare a frequenti operazioni di I/O su disco se il blocco di puntatori in memoria è quasi pieno quindi una strategia alternativa prevede di dividere il blocco pieno di puntatori per gestire meglio i blocchi liberi senza I/O su disco.

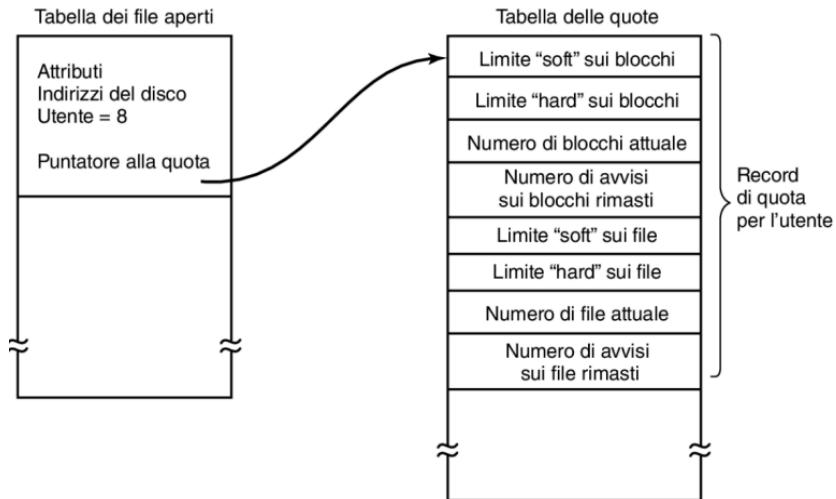
Meccanismo di quota del disco nei sistemi multiutente:

L'amministratore di sistema assegna a ogni utente un numero massimo di file e blocchi, il sistema operativo controlla che gli utenti non superino la loro quota.

Ogni apertura di file coinvolge il controllo degli attributi e degli indirizzi del disco, attributi includono l'identificazione del proprietario del file e incrementi della dimensione del file sono contabilizzati nella quota del proprietario.

Una tabella separata tiene traccia delle quote di ogni utente con file aperti, record delle quote aggiornati e riscritti sul file delle quote alla chiusura dei file.

Limiti delle quote: Limite "soft" può essere temporaneamente superato, "hard" mai, una violazione dei limiti "hard" o ignorare gli avvisi dei limiti "soft" possono portare alla restrizione dell'accesso, gli utenti possono superare temporaneamente i limiti "soft" durante una sessione di lavoro, ma devono rientrare nei limiti prima di scollegarsi.



### Organizzazione EXT2:

#### Componenti Chiave del File System Ext2:

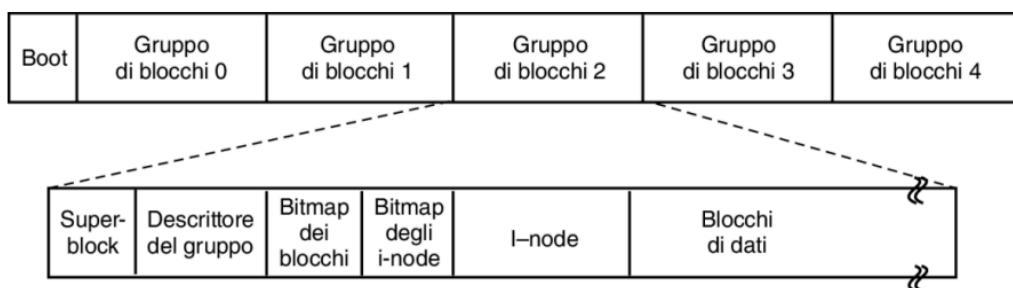
**Superblocco:** Informazioni sul layout, numero di I-node, e blocchi del disco.

**Descrittore del Gruppo:** Dettagli sui blocchi liberi, I-node, e posizione delle bitmap.

**Bitmap:** Traccia i blocchi liberi e gli I-node liberi, seguendo il design di MINIX 1.

**I-node:** Numerati, descrivono un solo file, contengono informazioni di contabilità e indirizzi dei blocchi di dati.

**Blocchi di Dati:** Archiviano i file e le directory, non necessariamente contigui sul disco.



#### Gestione degli I-node e dei file:

Gli I-node delle directory sono distribuiti tra i gruppi di blocchi del disco, Ext2 cerca di posizionare i file nella stessa area di blocchi della directory genitore per minimizzare la frammentazione, si utilizza la bitmap per determinare rapidamente aree libere dove allocare nuovi dati del file system.

**Preallocazione di Blocchi:** Ext2 prealloca otto blocchi aggiuntivi per un nuovo file per ridurre la frammentazione dovuta a scritture successive, questa strategia bilancia il carico del file system e migliora le prestazioni riducendo la frammentazione.

Per l'accesso ai file si utilizzano chiamate di sistema, come open (le applicazioni non possono accedere autonomamente ai file), l'analisi del percorso del file inizia dalla directory corrente del processo o dalla directory radice.

**Ricerca e Accesso ai File:** Le ricerche nelle directory sono lineari ma ottimizzate tramite una cache delle directory recentemente accedute, per aprire un file, il percorso viene analizzato per localizzare l'I-node della directory e, infine, l'I-node del file stesso, si usano Soft link e Hard link per fare riferimento ai file.

Come garantire le prestazioni del file system?

Velocità di Accesso: nella memoria abbiamo una velocità di accesso ultraveloce (es. 10 ns per leggere una parola a 32 bit), sul disco magnetico l'accesso è più lento a causa del tempo di ricerca della traccia (5-10 ms) e dell'attesa del posizionamento del settore sotto la testina di lettura (ridotto negli SSD, ma non sono usati nei server e sono più costosi).

Bisogna progettare il file system con diverse ottimizzazioni per migliorare le prestazioni, considerando le significative differenze nel tempo di accesso e riducendo al minimo, i numeri di accesso al disco/SSD, il tempo di ricerca (seek) del dato sul disco/SSD e l'utilizzo dello spazio.

Tecniche di ottimizzazione:

- Block Cache / Buffer Cache: Utilizzata per ridurre i tempi di accesso al disco, mantenendo i blocchi più usati in memoria, migliora le prestazioni conservando in memoria i blocchi logici del disco.

- Allocazione dei Blocchi e Read Ahead:

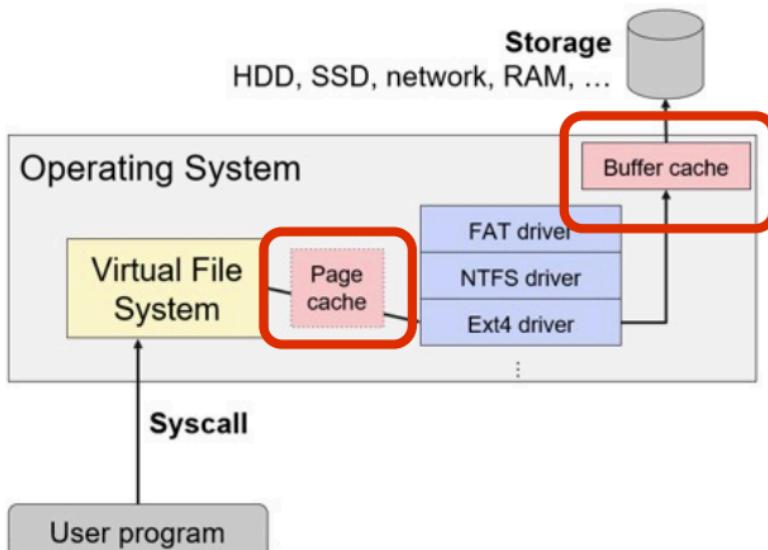
Tecniche di allocazione intelligente: blocchi vicini allocati nello stesso cilindro per minimizzare il movimento del braccio del disco.

Bitmap in memoria per allocare blocchi adiacenti e migliorare l'efficienza di scrittura sequenziale.

- Deframmentazione: Con il tempo, i dischi si frammentano; i file sparsi portano a prestazioni inferiori, con la deframmentazione si riorganizzano i file per essere contigui e si raggruppa lo spazio libero. (Windows strumento defrag, consigliato per HDD ma non per SSD che sono molto più sensibili a cancellazioni).

Concetti di Caching:

- Buffer Cache: Memorizza i blocchi del disco in RAM per ridurre gli accessi al disco.
- Page Cache: Memorizza le pagine (non le stesse pagine che vengono messe sul disco dalla memoria, ma strutture di intermediazione tra VFS e FS) del filesystem virtuale (VFS) in RAM prima di passare al driver del dispositivo.



Ottimizzazione del Caching:

Dati duplicati: Buffer cache e page cache spesso contengono gli stessi dati. Esempio: file «mappati» in memoria (legge il file, lo mette in cache e dopo lo mette in memoria).

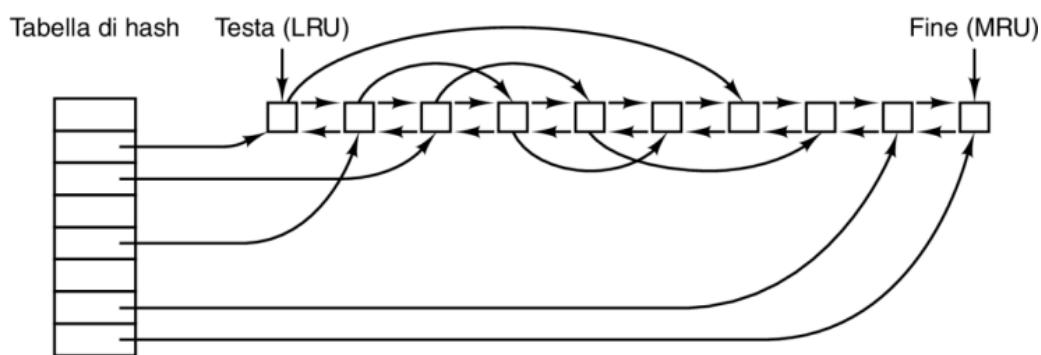
Fusione di cache: I sistemi operativi possono combinare buffer cache e page cache per un uso più efficiente della memoria e una riduzione degli accessi al disco.

#### Definizione e Scopo della Cache:

Cache (block cache o buffer cache): Raccolta di blocchi del disco tenuti in memoria per migliorare le prestazioni con lo scopo di ridurre i tempi di accesso al disco.

Gestione della Cache: Verifica delle richieste di lettura per determinare se il blocco richiesto è già in cache, se il blocco è in cache, viene soddisfatta la richiesta senza accedere al disco, se il blocco non è in cache, viene prima letto da disco, portato in cache e poi utilizzato.

Uso di hash per identificare rapidamente la presenza di un blocco in cache e lista concatenata bidirezionale LRU per gestire i blocchi con lo stesso valore hash (meno recenti in testa, più recenti in fondo per mantenere l'esatto ordine LRU).



#### Sostituzione dei Blocchi nella Cache:

Funzionamento: Quando la cache è piena, i nuovi blocchi sostituiscono quelli esistenti, che vengono riscritti su disco se modificati.

Algoritmi di Sostituzione: Uso di algoritmi come FIFO, seconda chance, e LRU.

#### Limitazioni dell'Algoritmo LRU:

LRU può portare a inconsistenze in caso di crash, specialmente per blocchi critici come i blocchi di I-node perciò si è deciso di utilizzare uno schema di LRU modificato basato sull'importanza e la necessità immediata dei blocchi.

#### Scrittura e Coerenza del File System:

Per i blocchi critici se modificati c'è una scrittura immediata su disco, per mantenere la coerenza del file system.

Unix: Utilizzo di chiamate di sistema come sync in UNIX per scrivere periodicamente i blocchi modificati su disco e ridurre la perdita di dati in caso di crash.

Windows: Strategia write-through (scrittura immediata) per i blocchi modificati, ora integrata anche con la chiamata FlushFileBuffers.

#### Integrazione tra Cache Buffer e Cache delle Pagine:

Obiettivo: Ottimizzare l'I/O e semplificare la gestione della memoria.

Alcuni sistemi operativi combinano cache buffer e cache delle pagine per una gestione efficiente dei dati così da avere un trattamento unificato di blocchi e pagine dei file in una singola cache. In questo modo un file interamente mappato in memoria non occupa un'area di memoria utile per altre pagine, ma sfrutta il Cache Buffer del disco.

Ottimizzazione della disposizione di dati:

Riduzione movimentazione del braccio: Minimizzare i movimenti del braccio del disco raggruppando i blocchi di dati utilizzati in sequenza, quindi uso la bitmap in memoria per allocare blocchi vicini riducendo i tempi di ricerca (al posto di ruotare e spostare il braccio ruoto solo il disco).

Strategie di Allocazione: Tracciamento dello spazio non per blocchi singoli, ma per gruppi consecutivi per migliorare la lettura sequenziale, posizionamento rotazionale quindi allocazione di blocchi consecutivi di un file nello stesso cilindro per ridurre i tempi di ricerca.

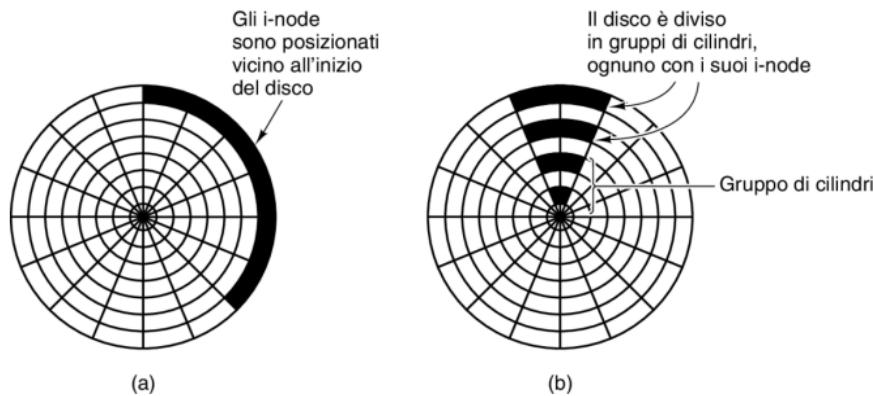
Posizionamento degli I-node:

a) Posizionamento Tradizionale: I-node vicino all'inizio del disco, risultando in ricerche più lunghe.

Miglioramenti nel Posizionamento:

b) Centrare gli I-node: Posizionare gli I-node nel mezzo del disco per diminuire il tempo di ricerca (muovendo solo il braccetto), quindi si ha una divisione del disco in gruppi di cilindri con I-node, blocchi e lista dei blocchi liberi per ciascun gruppo.

Considerazioni per SSD: Gli SSD non hanno parti mobili, quindi i tempi di accesso casuale sono simili a quelli sequenziali ("Seek time" costante).



Il comando free è uno strumento essenziale in Linux per monitorare l'utilizzo della memoria, fornisce una panoramica dettagliata della memoria RAM, inclusa la cache e lo spazio libero. La colonna "buff/cache" rivela quanto spazio è utilizzato per buffer e cache, inclusi i dati letti dal disco, "Free": mostra la memoria fisica non utilizzata attualmente, "Available": stima la memoria disponibile per nuovi processi, considerando anche la memoria facilmente liberabile come cache e offre una visione realistica della memoria effettivamente disponibile per applicazioni senza influire sulle prestazioni.

Impatto della Frammentazione: Con il tempo, i file e lo spazio libero si disperdonano sul disco, causando una diminuzione delle prestazioni. La deframmentazione consolida i file e lo spazio libero, migliorando l'efficienza di lettura/scrittura per HDD. In Windows defrag in Linux con ext3/ext4 si gestisce meglio la frammentazione, Ext4 introduce la preallocazione di blocchi, quando si scrive un file Ext4 prealloca un gruppo di blocchi contigui, piuttosto che uno alla volta, riducendo la frammentazione per i file in espansione.

### Compressione e duplicazione:

Compressione dei Dati: Riduce la dimensione dei file tramite algoritmi che identificano e sostituiscono sequenze di dati ripetuti, file system come NTFS (Windows), Btrfs (Linux) e ZFS (vari sistemi operativi) possono comprimere dati automaticamente.

Deduplicazione dei Dati: Rileva e rimuove i dati duplicati all'interno di un intero file system, conservando una sola copia di ciascun dato unico, applicata sia a livello di blocchi di disco che di porzioni di file, prevalentemente in ambienti con dati condivisi.

### Metodi di Deduplicazione:

- Inline: Controlla i dati per duplicati durante la scrittura, potenzialmente rallentando il processo.
- Post-Process: Scrittura immediata dei dati, con controllo di duplicati eseguito successivamente in background.

Sicurezza e Affidabilità tramite controllo degli Hash: Necessario per assicurare che i dati non siano falsamente identificati come duplicati a causa di collisioni hash.

### Come garantire l'affidabilità del file system?

#### Minacce e Guasti del Disco:

Blocchi Danneggiati: Settori illeggibili che possono corrompere i dati.

Errori su Intero Disco: Fallimenti hardware che rendono il disco completamente inutilizzabile.

Interruzioni di Energia: Portano a scritture inconsistenti quindi incongruenze nei dati o nei metadati sul disco.

Bug del Software / Corruzione dei (Meta)dati: Errori di programmazione portano alla scrittura di dati errati/corrotti.

Errori Umani/Comandi Errati: rm \*.o VS rm \* .o.

Perdita o furto del Computer/Accesso non Autorizzato.

Malware/Ransomware: Virus o altri software dannosi che possono infettare, criptare o distruggere i dati.

Il backup è quindi cruciale per salvaguardare informazioni importanti come documenti, database, piani aziendali, ecc.

#### Modalità di Backup:

- Backup Completo: Esegue una copia totale dei dati, solitamente su base settimanale o mensile.
- Backup Incrementale: Copia solo i file modificati dall'ultimo backup completo, riducendo il tempo e lo spazio richiesti.

#### Tipologie di Backup:

- Backup Fisico: Copia sequenziale di tutti i blocchi del disco, a partire dal blocco 0 fino all'ultimo.
- Backup Logico: Seleziona e copia solo i file e le directory specifici, ignorando i file di sistema e i blocchi danneggiati.

### Considerazioni Tecniche:

Comprimere i Dati: Riduce lo spazio necessario, ma aumenta il rischio di perdita di dati a causa di errori di compressione.

Backup di File System Attivi: Richiede l'uso di snapshot per garantire coerenza durante il backup di un sistema in uso (sennò i file modificati non saranno presenti in backup).

Sicurezza dei Backup: Importanza di tenere i backup in luoghi sicuri e separati per prevenire perdite o danni.

### Considerazioni sul Backup Fisico:

Efficienza: Semplice e veloce, può essere eseguito alla velocità del disco.

Gestione dei Blocchi Danneggiati: Necessità di evitare la copia di blocchi danneggiati per prevenire errori di lettura (fa il backup anche dei blocchi danneggiati).

File Non Necessari: Necessità di evitare la copia di file di sistema come i file di paginazione e ibernazione.

Vantaggi: Facile da implementare e utilizzare, ha minore probabilità di errori nel processo di backup.

Mancanza di Flessibilità: Difficoltà nel saltare directory specifiche o nel fare backup incrementali.

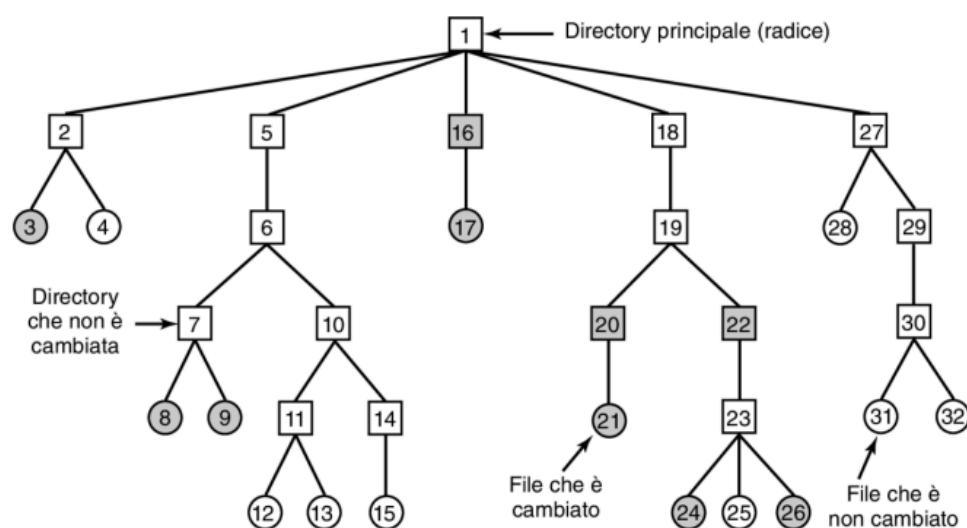
Ripristino di Singoli File: Non è possibile ripristinare file individuali senza un intero ripristino del sistema.

### Backup Logico:

Parte da specifiche directory e effettua il backup di tutti i file e directory modificati a partire da una data specifica, permette il ripristino o il trasferimento dell'intero file system su un nuovo computer, ideale per backup incrementali o completi.

Recupero Facilitato: Consente il ripristino semplice di file o directory specifici grazie alla precisa identificazione dei dati salvati.

Algoritmo di Backup in UNIX: Include file e directory modificati (in grigio nella figura), e tutte le directory lungo il percorso verso i file modificati.



I quadrati sono cartelle, i file sono cerchi, colore grigio soltanto se hanno subito modifiche rispetto ad un backup e il numero rappresenta i loro I-node.

Utilizzo di una Bitmap per il Backup: L'algoritmo per il backup logico utilizza una bitmap indicizzata per numero di I-node, con bit impostati per I-node di file e directory modificati.

Fase 1: Rilevamento delle Modifiche: Inizia dalla directory radice, esamina tutte le voci e contrassegna nella bitmap gli I-node di file e directory modificati, include tutte le directory nel percorso verso file modificati, indipendentemente dal loro stato di modifica. a) mostra gli oggetti selezionati nella bitmap.

Fase 2: Pulizia della Bitmap: Deseleziona le directory che non contengono né file né sottodirectory modificate. Lascia nella bitmap solo gli elementi che richiedono backup. b) mostra la bitmap dopo la pulizia.

Fase 3: Backup delle directory e file contrassegnati, con i loro attributi. c e d) indica le directory (c) e i file da salvare (d).

(a) 

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

(b) 

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

(c) 

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

(d) 

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Ripristino dal Backup: Creazione di un nuovo file system vuoto e ripristino del backup completo più recente, seguito dai backup incrementali.

Considerazioni Aggiuntive: Ricostruzione della lista dei blocchi liberi, gestione dei link multipli ai file, trattamento dei file con buchi (spazi non scritti tra i dati), esclusione di file speciali non necessari nel backup.

Definizione di rsync: è un comando utilizzato nei sistemi basati su UNIX per la sincronizzazione di file e cartelle tra due location diverse, sia su una stessa macchina sia tra macchine diverse, ottimizza il trasferimento dei dati trasmettendo solo le parti di file che sono state modificate, ideale per backup, ripristino e sincronizzazione di dati.

Funzionalità Principali:

Efficienza: Trasferisce solo le differenze tra le sorgenti e le destinazioni.

Versatilità: Supporta la copia di link, dispositivi, attributi, permessi, dati utente e gruppo.

Sicurezza: Può utilizzare SSH per trasferimenti criptati.

Esempio:

Sincronizzazione Locale: rsync -av /sorgente/cartella /destinazione/cartella

-a: modalità archivio, mantiene i permessi e la struttura dei file.

-v: modalità verbosa, mostra i dettagli del trasferimento.

Sincronizzazione Remota: rsync -av /sorgente/cartella utente@remoto:/destinazione/cartella, sincronizza la cartella dalla macchina locale a quella remota utilizzando l'account utente.

Importanza della Coerenza tra file system e backup: Cruciale per mantenere l'integrità dei dati, problemi di incoerenza possono sorgere a seguito di crash durante la scrittura dei blocchi.

Utility per la Verifica della Coerenza: UNIX (fsck) e Windows (sfc) hanno comandi per verificare la coerenza, eseguite all'avvio, specialmente dopo un crash.

File System con Journaling: Progettati per gestire autonomamente la maggior parte delle incoerenze, e non necessitano di controlli esterni dopo un crash.

Tipi di controllo di coerenza:

Controllo dei Blocchi: Costruzione di due tabelle con contatori per ogni blocco (per file e blocchi liberi) e analisi di tutti gli I-node con verifica della presenza dei blocchi in file e nella lista dei blocchi liberi.

Situazioni Riscontrabili:

- a) File system coerente.
- b) Il blocco 2 è segnalato come mancante, quindi viene aggiunto alla lista dei blocchi liberi.
- c) Il blocco 4 appare 2 volte come se fosse libero, quindi viene ricostruita la lista.

Numero del blocco															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0

0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(a)

Numero del blocco															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0

0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(b)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0

0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(c)

Verifica del sistema e Protezione dell'utente:

Controllo del Sistema delle Directory: Utilizzo di una tabella di contatori per file, e confronto dei contatori dei link negli I-node con le voci di directory.

Tipi di Errori: Contatori dei link troppo alti o bassi significa errori nel formato delle directory o I-node.

Protezione dell'Utente contro Errori Umani: Esempio: distinzione tra 'rm \*.o' e 'rm \* .o' in sistemi UNIX, in Windows invece, file cancellati vengono spostati nel cestino per un possibile recupero.

Funzionamento di Journaling:

Definizione e Scopo: File system con journaling registrano anticipatamente le operazioni da eseguire in un log per garantire la coerenza in caso di crash. Ampio uso in file system come NTFS (Microsoft), ext4 e ReiserFS (Linux), e opzione predefinita in macOS.

Esempio di Operazione: Eliminazione di un File:

Processo in UNIX: Rimozione dal file dalla directory, rilascio dell'I-node, restituzione dei blocchi al pool dei blocchi liberi.

In caso di crash si ha la perdita di accesso agli I-node e ai blocchi, o assegnazione errata di I-node e blocchi.

Un journal in un file system è come un registro che tiene traccia delle modifiche che verranno apportate al file system prima che esse avvengano effettivamente.

Come Funziona:

1. Fase di Registrazione: Prima di eseguire qualsiasi modifica (come la creazione o la cancellazione di un file), il file system scrive un record nel journal. Questo record descrive l'operazione che verrà eseguita.
2. Fase di Esecuzione: Dopo aver registrato l'operazione, il file system procede con la modifica effettiva dei dati sul disco.
3. Fase di Conferma: Una volta completata l'operazione, il file system aggiorna il journal per indicare che l'azione è stata completata con successo. Se si verifica un crash del sistema prima che una modifica sia completata, al riavvio successivo il file system consulta il journal, se trova operazioni registrate ma non confermate, procede a completare. Questo assicura che le modifiche parziali non lascino il file system in uno stato incoerente.

Vantaggi del Journaling:

Integrità dei Dati: Il journaling riduce la possibilità di corruzione del file system in caso di crash inaspettato, assicurando che tutte le operazioni siano completate o nessuna.

Recupero Rapido: Riduce significativamente il tempo di recupero dopo un crash, poiché il file system sa esattamente quali operazioni completare o annullare.

Eliminazione sicura dal disco:

La cancellazione standard non rimuove fisicamente i dati dal disco, lasciandoli vulnerabili agli attacchi, l'eliminazione sicura richiede la distruzione fisica o la sovrascrittura approfondita dei dati.

Dati Residui su Dischi Magnetici: Sovrascrivere con tutti 0 non è sempre sufficiente a causa dei residui magnetici che possono essere recuperati con tecniche avanzate. Gli SSD presentano sfide aggiuntive: la mappatura dei blocchi flash è gestita dalla FTL e non dal file system, rendendo la sovrascrittura meno prevedibile. È consigliato inserire sequenze di 0 e numeri casuali, ripetendo l'operazione almeno 3-7 volte, a differenza degli HDD molte scritture possono mettere a dura prova gli SSD.

Cifratura:

Cifratura del Disco: La soluzione più efficace per proteggere i dati è cifrare l'intero disco con algoritmi robusti come l'AES, sistemi operativi moderni, come Windows, offrono la cifratura del disco che lavora in background, spesso sconosciuta agli utenti.

Implementazione della Cifratura:

SED (Self-Encrypting Drives): Dispositivi con cifratura integrata, che tuttavia possono avere vulnerabilità di sicurezza.

Windows utilizza AES (Advanced Encryption Standard) per cifrare i dischi, con la chiave master del volume decifrata tramite password utente, chiave di ripristino o TPM.

Virtual File System:

File System in Sistemi Operativi: Sistemi operativi moderni gestiscono diversi file system simultaneamente:

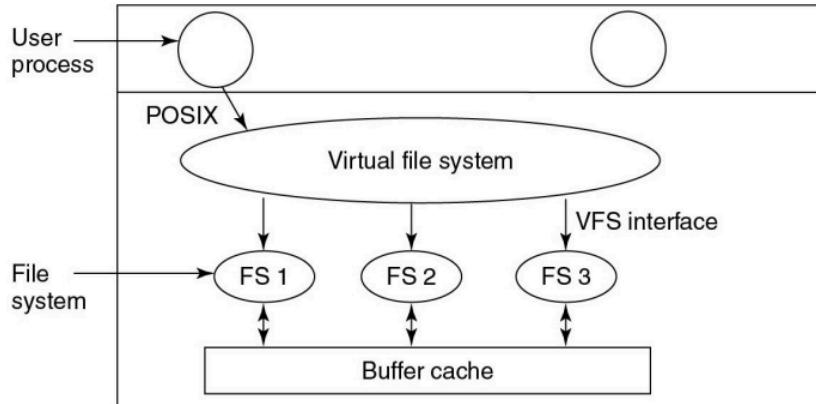
Windows utilizza lettere di unità (C:, D:, ecc.) per gestire file system differenti.

UNIX e Linux integrano più file system in un'unica struttura gerarchica.

Virtual File System (VFS): è un layer di astrazione nei sistemi operativi, una struttura che permette di integrare vari file system in una struttura unificata, si basa su un livello di codice comune che interagisce con i file system reali sottostanti.

Interfacce del VFS: ha un'interfaccia superiore dove interagisce con le chiamate di sistema

POSIX dei processi utente (es. open, read,) e interfaccia inferiore composta da funzioni che il VFS può inviare al file system sottostante.

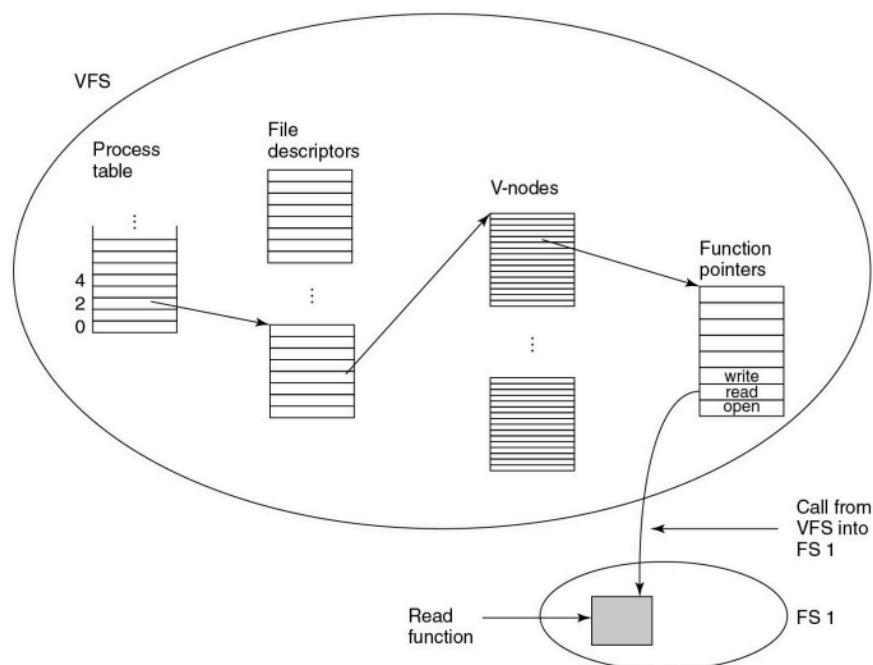


### Struttura del VFS:

**Superblock nel VFS:** Rappresenta il descrittore di alto livello di un file system specifico nel VFS, informazioni cruciali sul file system, come il tipo, la dimensione, viene usato per identificare e interagire con il file system sottostante, facilitando l'accesso e la gestione delle sue risorse.

**V-node (virtual-node) nel VFS:** Astrazione di un file individuale all'interno del VFS, rappresentando un nodo nel file system virtuale, contiene metadati come permessi, proprietà, dimensione del file e riferimenti ai dati effettivi sul disco. Il VFS sfrutta i v-node per fornire un accesso indipendente dal file system ai file, permettendo operazioni di lettura, scrittura e gestione dei file attraverso vari file system.

**Directory nel VFS:** Struttura che gestisce l'organizzazione e il mapping dei file e delle sottodirectory all'interno del VFS, permette al VFS di mappare i nomi dei file ai loro v-node corrispondenti, indipendentemente dal file system in cui si trovano. Facilita la navigazione e l'accesso ai file, consentendo agli utenti e ai processi di interagire con un'interfaccia unificata.



Una volta eseguito il mount del file system si deve:

Registrazione dei File System con il VFS: File system forniscono un vettore di funzioni (funzioni messe a disposizione dal VFS per colmare il gap tra la chiamata di sistema virtualizzata e la chiamata di sistema vera e propria) richieste dal VFS al momento della registrazione. Permette al VFS di sapere come eseguire specifiche operazioni su un file system registrato.

Montaggio e Uso del File System: Al montaggio, file system fornisce informazioni al VFS (es. superblock). Esempio: apertura di un file in /usr con un file system montato su /usr e creazione di un v-node e mappatura di operazioni specifiche del file system reale.

Gestione delle Richieste di I/O: Tracciamento dei file aperti nei processi utente tramite v-node e tabelle dei descrittori dei file. Chiamate come read seguono il puntatore dalla tabella dei descrittori ai v-node e alle funzioni del file system reale.

Aggiunta di Nuovi File System: Relativamente semplice aggiungere nuovi file system, i progettisti devono fornire funzioni che rispettino l'interfaccia VFS. Il VFS rende possibile la gestione trasparente di file system eterogenei.

Quale file system viene utilizzato?

Introduzione al file system v7 di UNIX (1979):

Origine e Influenza: Derivato dal MULTICS, il file system V7 è stato implementato nel PDP-11 e ha contribuito significativamente alla fama di UNIX.

Struttura: Formato da un albero gerarchico con la possibilità di formare un grafo aciclico orientato tramite link.

Nomi dei File: Lunghezza massima di 14 caratteri, escludendo i caratteri '/' e NUL.

Limitazione Numerica: Massimo di 64K file per file system, a causa del formato delle voci di directory.

Voce di Directory: Composta da un nome di file (14 byte) e un numero di i-node (2 byte).

I-node: Contiene attributi del file, inclusi dimensioni, timestamp, proprietario, gruppo, e informazioni di protezione.

Gestione dei Link: Contatore di link nell'i-node, che viene incrementato o decrementato con la creazione o rimozione di link.

Recupero di Blocchi: Utilizzo di indirizzi diretti e indiretti (singoli, doppi e tripli) per gestire file di varie dimensioni.

Ricerca di File: Processo per localizzare file tramite un percorso, partendo dalla directory radice o dalla directory corrente.

Percorsi Relativi: Gestiti allo stesso modo dei percorsi assoluti, iniziando dalla directory di lavoro corrente.

Gestione delle Directory Speciali: Uso di . e .. per indicare rispettivamente la directory corrente e la genitore.

Limitazioni del file system V7:

Limitazioni Temporali: Problemi legati alla rappresentazione di date e orari.

Semplicità ed Efficienza: Struttura del file system progettata per essere semplice ma efficace nell'accesso e nella gestione dei file.

Impatto su Sistemi Successivi: Il design e le caratteristiche del file system V7 hanno influenzato lo sviluppo di file system UNIX successivi, tra cui quelli utilizzati in sistemi Linux moderni.

Evoluzione dei file system in Linux:

- Ext (1992): Il Primo File System di Linux:

Creato specificatamente per il kernel di Linux da Rémy Card, superava i limiti del file system MINIX, con una capacità massima di 2 GB era il primo ad utilizzare il Virtual File System (VFS) nel kernel Linux.

- Ext2 (1993): Ext2 introdotto per risolvere problemi di Ext, come l'immutabilità degli i-node e la frammentazione.

Immutabilità: una volta creati, gli attributi principali di un i-node (come il suo numero identificativo) non cambiano per tutta la durata della vita del file a cui sono associati (se creati in sola lettura rimanevano in sola lettura).

Competizione con Xafs, ma Ext2 prevale per la sua maggiore affidabilità a lungo termine.

- Ext3: L'Introduzione del Journaling per una maggiore integrità dei dati.
- Ext4 (2006-2008): Inizialmente estensioni retrocompatibili di Ext3, Google passa da Ext2 a Ext4 per il suo storage nel 2010, adottata anche da Android.

Journaling e Affidabilità: Introduce il journaling per prevenire la perdita di dati in caso di crash, utilizza un Journaling Block Device (JBD) per le operazioni di log, configurabile solo per i metadati o per l'intero disco.

Supporta file di 16TB e file system fino a 1EB.

Miglioramenti rispetto a Ext2 e Ext3:

Aumento della dimensione massima dei file e dei file system.

Introduzione degli "extent" per una gestione efficiente dei blocchi di memoria contigui, gli "extent" in ext4 sono strutture che indicano un intervallo contiguo di blocchi su disco, specificando l'indirizzo di inizio e la quantità di blocchi consecutivi, questo approccio semplifica la gestione di blocchi contigui, riducendo il numero di voci necessarie per descrivere la locazione dei dati su disco, specialmente per file di grandi dimensioni.

Compatibile con ext2 ed ext3, ma con prestazioni e capacità superiori.

BTRFS:

Btrfs: Copy-on-Write: Quando un file è duplicato, Btrfs condivide il file originale invece di creare una copia, riducendo lo spazio occupato.

Prevenzione della Perdita di Dati: quando i dati vengono modificati, Btrfs non sovrascrive i dati esistenti, ma invece scrive le modifiche in una nuova posizione sul disco.

Caratteristiche Salienti di Btrfs:

Supporto per File Enormi: Gestisce file fino a 16 exbibyte (18446.7 petabyte).

Archiviazione Efficiente: Riduce il sovraccarico nei metadati dei file, ottimizzando così la gestione dello spazio e delle prestazioni.

Supporto RAID: Compatibile con RAID 0, 1 e 1+0 per striping e mirroring dei dati.

Deframmentazione e Ridimensionamento Facili: Operazioni eseguibili mentre il filesystem è attivo.

Allocazione Dinamica degli Inode: Evita l'esaurimento degli inode, salvandoli per un gran numero di piccoli file.

Supporto per Snapshot: Permette la creazione e il ripristino facili degli snapshot (backup) del filesystem.

Supporto Checksum: Riduce il rischio di corruzione dei dati attraverso blocchi di dati verificati costantemente.

Ottimizzazione per SSD: Migliora le prestazioni degli SSD, estendendone la durata.

Confronto tra i vari file system:

Btrfs:

Progettato per l'era dei moderni dispositivi di archiviazione, supporta snapshot e rollbacks, ottimale per backup e ripristini, gestione nativa del RAID e miglioramento nell'integrità dei dati con checksum, compressione dei dati e deduplicazione per un utilizzo efficiente dello spazio.

Ext3: Affidabile con supporto journaling, ma limitato in termini di funzionalità avanzate.

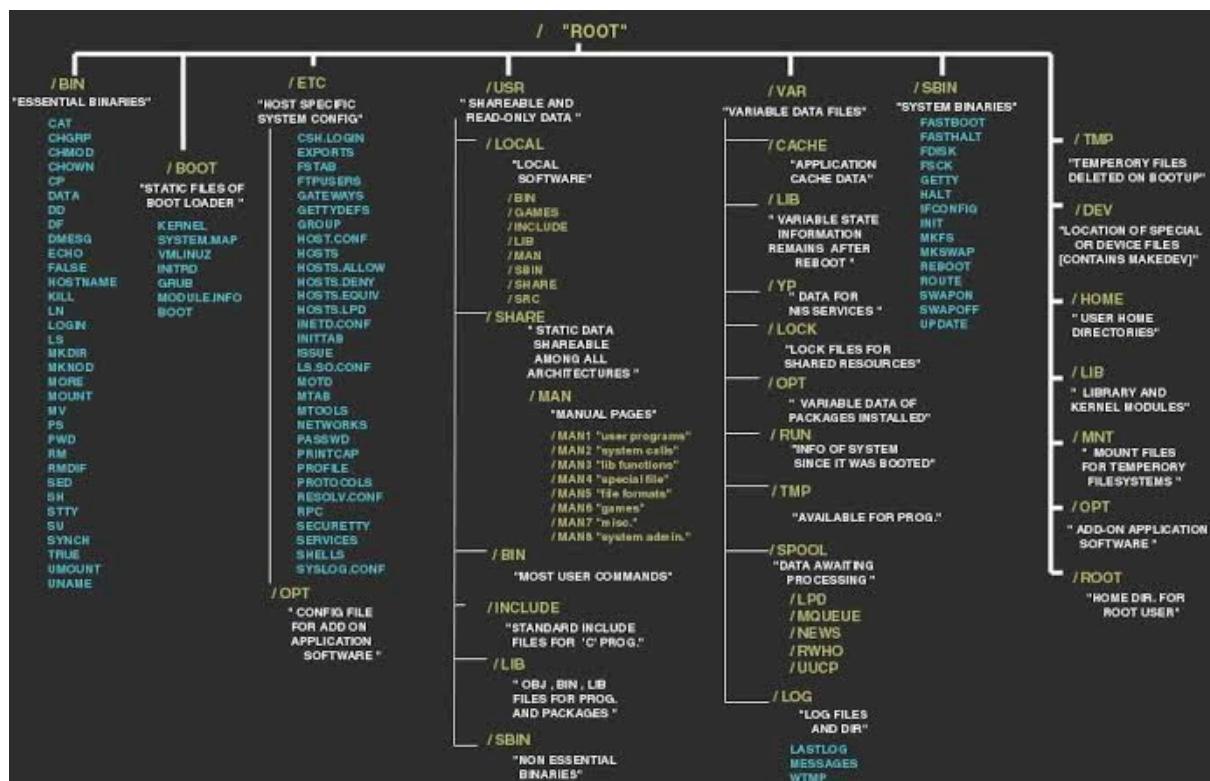
Ext4: Miglioramenti nell'efficienza, supporto per file di grandi dimensioni, riduzione della frammentazione.

Vantaggi e Svantaggi:

Btrfs offre funzionalità avanzate ma è meno maturo e testato rispetto a Ext4.

Ext3/Ext4 è noto per la sua stabilità e prestazioni, ma manca di alcune delle caratteristiche moderne di Btrfs.

Struttura cartelle in Linux:



/bin: contiene i binari dei principali comandi eseguibili dal sistema: cat, ls, pwd, ecc...

/boot: contiene tutti i file necessari al Boot Loader per il processo di avvio del sistema.

Inoltre, contiene il Kernel.

/etc: contiene tutti i file di configurazione del sistema e di controllo

/usr: contiene i pacchetti del sistema e le directory e le applicazioni dell'utente

/var: contiene file temporanei, log, ecc...

/sbin: ci sono i binari del sistema, quindi anche comandi come ifconfig, mkfs, fdisk , ecc...

/dev: i file descrittori dei device, ovvero dei dispositivi/periferiche come gli hard disk interni.

In linux anche i dispositivi vengono visti come file!

/home: cartella principale dell'utente dove vengono salvati i file locali, contenente le cartelle: Documenti, Immagini, ecc...

/lib: contiene le librerie essenziali, incluso il compilatore C e le relative librerie.

/media: contiene cartelle che servono per gestire media rimovibili «montati» automaticamente dal sistema, come cd, floppy, ecc...

/mnt: usata per eseguire il mount manuale dei filesystem temporanei dei dispositivi rimovibili come USB e cd. § Infatti, una volta attaccata una chiavetta usb, se non “montassimo” il suo filesystem per poterne leggere il contenuto, non saremmo in grado di visualizzarne le informazioni. § Solitamente c’è una cartella per ogni dispositivo montato.

/opt: abbreviazione di “Optional”, contiene sia gli add-ons di alcuni software, sia programmi che non sono necessari al sistema. Molte persone usano questa cartella per creare sotto-directory in cui compilare e installare programmi.

/root: home dell’utente root.

#### Partizioni:

Scoprire Partizioni e File System Disponibili, vari comandi.

lsblk: Mostra un elenco dei dispositivi di blocco, inclusi dischi e partizioni

fdisk -l: Elenca dettagliatamente tutte le partizioni sui dischi, comprese quelle non montate. § Richiede privilegi di root (sudo)

mount -l: Elenco dei file system attualmente montati con le loro opzioni di montaggio e etichette, utile per vedere rapidamente dove e come sono montate le partizioni e i file system.

Importanza di Conoscere le Partizioni: Fondamentale per gestire correttamente lo spazio su disco e l’organizzazione dei dati e per operazioni di backup, ripristino e manutenzione del sistema.

Montaggio di una Partizione/File System: mount [opzioni]. Esempio: sudo mount /dev/sda1 /mnt/mydisk per montare /dev/sda1 in /mnt/mydisk.

Creazione della Directory di Montaggio: La directory di destinazione deve esistere prima del montaggio (Esempio: mkdir /mnt/mydisk).

Opzioni di Montaggio Comuni: specifica del Tipo di File System: -t , es. -t ext4, opzioni aggiuntive: -o , es. -o ro per montaggio in sola lettura.

Smontaggio di un File System: umount, utilizzato per smontare in modo sicuro un file system o una partizione, necessario per prevenire la perdita di dati. Esempio: sudo umount /mnt/mydisk.

Note Finali: Queste operazioni richiedono privilegi di root.

#### INPUT/OUTPUT

Gestione dell’IO nel sistema operativo:

Funzione Principale: Controllo dei dispositivi di I/O: invio di comandi, intercettazione di interrupt, gestione degli errori e fornire un’interfaccia semplice e uniforme tra dispositivi e il sistema (indipendenza dai dispositivi). Abbiamo:

Principi dell’Hardware per l’I/O: Fondamenti dell’hardware specifico per l’I/O

Software per l’I/O: Strutturato in livelli, ciascuno con funzioni specifiche.

#### Principi dell’hardware di I/O:

Ingegneri Elettronici: Vedono l’hardware di I/O in termini di componenti fisici (chip, cavi, alimentatori, motori, etc.), mentre i Programmatori sono Interessati all’interfaccia software dell’hardware, inclusi i comandi accettati, le funzioni eseguibili e i possibili errori.

Connessione tra Programmazione e Operatività Interna: La programmazione di molti dispositivi di I/O è spesso legata all’operatività interna dei dispositivi stessi.

Categorie dei Dispositivi di I/O:

- Dispositivi a Blocchi: Archiviazione di informazioni in blocchi di dimensioni fisse (512 a 32.768 byte), come dischi fissi magnetici, unità SSD, unità a nastro magnetico. Caratteristica chiave: ogni blocco può essere letto o scritto indipendentemente.
- Dispositivi a Caratteri: Flusso di caratteri senza struttura a blocchi (un bit dopo l'altro), non indirizzabili, senza operazioni di ricerca. Esempi: stampanti, interfacce di rete, mouse.

Limitazioni della Classificazione: Alcuni dispositivi non si adattano perfettamente a questa classificazione (es. clock, schermi mappati in memoria, touch screen) ma rimane un modello utile per astrarre alcuni software di I/O nel sistema operativo.

Applicazioni nel Sistema Operativo: Il file system gestisce dispositivi a blocchi astratti (sa che deve leggere o scrivere blocchi) mentre il software di livello inferiore gestisce le specificità dei dispositivi.

Dispositivi di I/O possono variare notevolmente in termini di velocità di trasferimento, creando sfide per il software di gestione.

Dispositivo	Velocità di trasferimento
Tastiera	10 byte/s
Mouse	100 byte/s
Modem a 56 K	7 KB/s
Bluetooth 5 BLE	256 KB/s
Scanner a 300 dpi	1 MB/s
Videocamera digitale	3,5 MB/s
Wireless 802.11n	37,5 MB/s
USB 2.0	60 MB/s
Disco Blu-ray 16x	72 MB/s
Gigabit Ethernet	125 MB/s
Disco fisso SATA 3	600 MB/s
USB 3.0	625 MB/s
Bus PCIe 3.0 single lane	985 MB/s
Wireless 802.11ax	1,25 GB/s
SSD NVME PCIe Gen 3.0 (lettura)	3,5 GB/s
USB 4.0	5 GB/s
PCI Express 6.0	126 GB/s

Componenti dei Dispositivi di I/O: Composti da una parte meccanica (il dispositivo stesso) e una elettronica (controller del dispositivo o adattatore), il Controller spesso è integrato nella scheda madre o come scheda aggiuntiva su slot PCIe.

Connettività e Gestione Multi-dispositivo: Controller con connettori per il collegamento a dispositivi c'è la capacità di gestire più dispositivi identici.

Standardizzazione dell'Interfaccia: Con il tempo si sono imposti diversi standard, (ANSI, IEEE, ISO) o standard de facto (SATA, SCSI, USB, ThunderBolt) in modo da permettere la produzione di dispositivi e controller compatibili da diverse aziende.

Interfaccia è di Basso Livello: Esempio: flusso seriale di bit in dischi, con preambolo, dati e ECC, il compito del controller è convertire il flusso seriale in blocchi di byte, correggere errori, trasferire in memoria principale.

Impatto sulla Programmazione del Sistema Operativo: Senza il controller, il programmatore

dovrebbe gestire dettagli complessi come la modulazione di ciascun pixel, il controller inizializzato con parametri essenziali, gestisce autonomamente questi dettagli complessi.

Per comunicare CPU e dispositivo utilizzano:

Input/Output mappati in memoria utilizzano:

- Registri dei Controller: Ogni controller di dispositivo ha registri per comunicare con la CPU.  
Scrittura nei registri: invia comandi al dispositivo (trasferimento dati, accensione/spegnimento, altre azioni).

Lettura dai registri: verifica lo stato del dispositivo e legge.

- Buffer di Dati: Molti dispositivi includono un buffer di dati per scrittura/lettura del sistema operativo. Esempio: RAM video utilizzata per visualizzare punti sullo schermo.
- Comunicazione CPU-Dispositivo: La CPU comunica con i registri di controllo e i buffer dei dati tramite tre approcci principali:

1. Port-mapped I/O
2. Memory-mapped I/O + (bonus)
3. Approccio Irido (Port-mapped I/O + Memory-mapped I/O ).

Importanza nel Sistema Operativo: Cruciale per gestire efficientemente il trasferimento di dati e il controllo dei dispositivi, incide sulla progettazione e sulle prestazioni del sistema.

### Port-mapped I/O

Assegnazione delle Porte di I/O: Ogni registro di controllo ha un numero di porta di I/O associato (intero di 8 o 16 bit). Formano lo spazio delle porte di I/O, accessibile solo dal sistema operativo per motivi di protezione.

Istruzioni di I/O Speciali:

Lettura: IN REG, PORT: La CPU legge dal registro di controllo PORT e salva il risultato in REG.

Scrittura: OUT PORT, REG: La CPU scrive il contenuto di REG in un registro di controllo. Ampio uso in vecchi computer e mainframe (es. IBM 360).

Spazi di indirizzi della memoria e dell'I/O sono distinti e non correlati

Esempi di istruzioni:

IN R0,4 Legge dalla porta di I/O 4 e salva in R0.

MOV R0,4 Legge dalla parola di memoria 4 e salva in R0.

Numero identico (es. 4) ma si riferisce a spazi di indirizzi diversi (primo IO, secondo di memoria).

Impatto sul Design di Sistema: Questo approccio influisce sul design e sulla programmazione dei sistemi operativi, differenziando nettamente la gestione della memoria e dell'I/O.

### Memory-mapped I/O

Approccio I/O Mappato in Memoria: Introdotta con il PDP-11, questa metodologia assegna a ogni registro di controllo un indirizzo di memoria univoco, si hanno registri di controllo mappati nello spazio della memoria.

Vantaggi dell'I/O Mappato in Memoria:

Elimina la necessità di istruzioni speciali di I/O come IN o OUT, i registri di controllo possono essere trattati come variabili in C, consentendo la scrittura di driver completamente in C e c'è una protezione semplificata: i processi utente non accedono direttamente ai registri di controllo.

### Controllo Selettivo dei Dispositivi:

Attraverso la gestione delle pagine di memoria, è possibile dare controllo selettivo su dispositivi specifici. Consente l'esecuzione di driver di dispositivi in spazi di indirizzo separati, aumentando la sicurezza e riducendo le dimensioni del kernel.

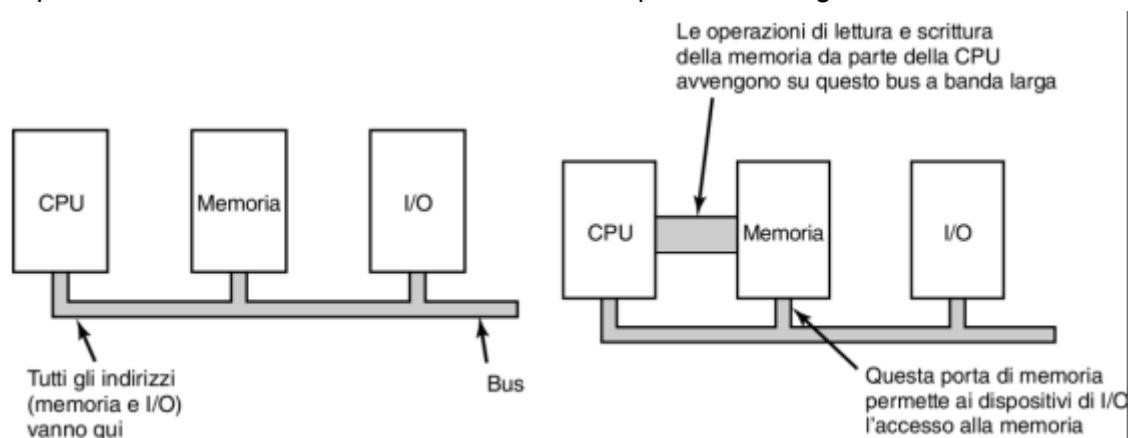
### Attenzione a combinare MMIO e cache:

Rischio di caching dei registri di controllo: Un registro finisce in cache, la CPU controlla solo la cache e non controlla se è stato modificato o no dal device, quindi se un ciclo while è in attesa che il contenuto del registro cambi, e quella variabile viene cambiata dal controller di un device si ha un ciclo infinito, c'è quindi la necessità di disabilitare selettivamente la cache per alcuni indirizzi (complessità aggiuntiva).

Gestione degli Indirizzi e Architetture del Bus (come comunicano): La CPU scrive sul bus e tutti leggono ma c'è la necessità per tutti i moduli di memoria e dispositivi di I/O di esaminare ogni riferimento alla memoria per sapere quali indirizzi finiscono in memoria e quali da un'altra parte, sorgono quindi problemi con bus della memoria separati in architetture come quelle x86 con bus multipli (memoria, PCIe, SCSI, USB).

Soluzioni per Architetture di Bus Multipli: Invio di riferimenti alla memoria prima alla memoria stessa; in caso di mancata risposta, tentativo con altri bus oppure si utilizza un dispositivo "spia" sul bus della memoria per intercettare indirizzi pertinenti ai dispositivi di I/O.

Equilibrio tra Prestazioni e Complessità: L'I/O mappato in memoria comporta compromessi tra prestazioni ottimizzate della memoria e la complessità nella gestione dell'I/O.



### PMIO-MMIO Ibrido

Approccio di «Filtro» degli Indirizzi: Implementato nel controller della memoria, filtrando gli indirizzi per distinguere tra memoria e dispositivi I/O.

Funzionamento: Controller della memoria con registri di indirizzi impostati all'avvio. Hanno un Intervallo specifico (es. 640K a 1M - 1) marcato come non di memoria e gli indirizzi in questo intervallo vengono inoltrati ai dispositivi anziché alla memoria.

Vantaggi e Svantaggi: Permette la coesistenza di memoria e dispositivi I/O in un unico spazio di indirizzi, richiede quindi la configurazione iniziale degli indirizzi di memoria non effettivi, lo svantaggio principale è la necessità di determinare all'avvio quali indirizzi non sono veramente indirizzi di memoria.



### MMIO / PMIO / PMIO-MMIO Ibrido

Cosa succede se l'operazione richiesta richiede tempo? La maggior parte dei dispositivi offre bit di stato nei propri registri per segnalare che una richiesta è stata completata (e il codice di errore). Il sistema operativo può interrogare questo bit di stato (polling), ma non è la soluzione ottimale, la soluzione ottimale utilizza il DMA.

#### DMA (Direct Memory Access)

**Definizione e Necessità del DMA:** DMA permette alla CPU di scambiare dati con i controller dei dispositivi bypassando il trasferimento manuale byte per byte riducendo lo spreco di tempo della CPU e migliorando l'efficienza del trasferimento dati.

**Configurazione Hardware:** Presenza di un controller DMA in molti sistemi, a volte integrato nei controller di dispositivi, il controller DMA può gestire trasferimenti a più dispositivi perché spesso situato proprio sulla scheda madre.

**Registri e Funzionamento del Controller DMA:** Contiene registri per indirizzi di memoria, conteggi di byte e controlli (direzione di trasferimento, unità di trasferimento, etc.).

#### Processo Tradizionale vs DMA:

- **Senza DMA:** Il controller del disco legge i dati e li memorizza nel suo buffer. Dopo aver controllato gli errori, provoca un interrupt e il SO copia i dati in memoria, considerando tutte le spese che il SO deve sostenere per ogni interrupt, quindi context switch e tutto non è la miglior soluzione.

- **Con DMA:**

Passo 1: La CPU imposta il controller DMA e invia un comando al controller del disco.

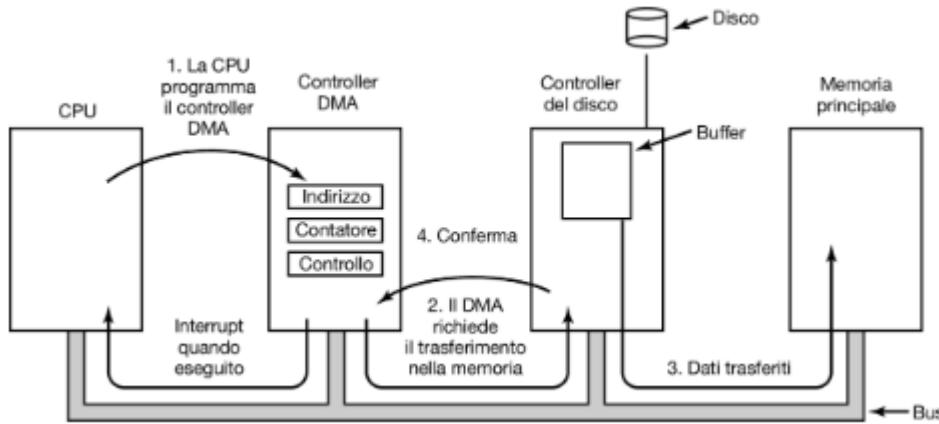
Passo 2: Il controller DMA richiede la lettura al controller del disco.

Passo 3: Scrittura in memoria da parte del controller del disco.

Passo 4: Conferma dal controller del disco al controller DMA.

Ripetizione dei passi 2-4 fino al completamento del trasferimento.

DMA invia un interrupt alla CPU al termine del trasferimento per avvertire che ha finito.



#### Variabilità nei Controller DMA:

Range: Da semplici (un trasferimento alla volta) a complessi (gestione di trasferimenti multipli simultanei).

Controller DMA Complessi: Multipli set di registri per diversi canali, ogni canale programmabile per trasferimenti specifici c'è la capacità di gestire contemporaneamente diversi controller di dispositivi.

Gestione dei Trasferimenti Multipli: Selezione del dispositivo successivo post-trasferimento tramite algoritmi come round-robin o prioritari, utilizza linee di conferma separate per ogni canale DMA sul bus.

Considerazioni sul Non-Uso del DMA: In alcune situazioni la CPU può essere più veloce del DMA nel gestire trasferimenti, nei computer embedded, l'eliminazione del controller DMA può essere una scelta per ridurre i costi.

#### Modalità DMA e Interazioni con il Bus:

- Cycle Stealing: DMA trasferisce una parola per volta, "rubando" cicli alla CPU. Questo rallenta leggermente la CPU ma permette la condivisione del bus.
- Burst: DMA ottiene controllo completo del bus, eseguendo trasferimenti multipli in una volta. Questa modalità è efficiente ma può bloccare la CPU per periodi prolungati perché può non avere accesso al bus.
- Fly-by Mode: DMA trasferisce dati direttamente alla memoria principale senza intermediari riducendo l'uso del bus ma richiedendo un ciclo extra per ogni trasferimento (bisogna verificare se va tutto a buon fine).

Gestione degli Indirizzi: DMA utilizza tipicamente indirizzi fisici, richiedendo conversione da parte del sistema operativo. Alcuni controller DMA supportano indirizzi virtuali tramite MMU o IOMMU.

Alcune periferiche (come il disco) hanno un Buffer Interno utilizzato per verificare i dati (checksum) e gestire l'afflusso costante di bit dal disco. Questo approccio semplifica il design del controller evitando problemi di temporizzazione e rischi di buffer overrun.

#### Interrupt:

##### Tipi di Interrupt:

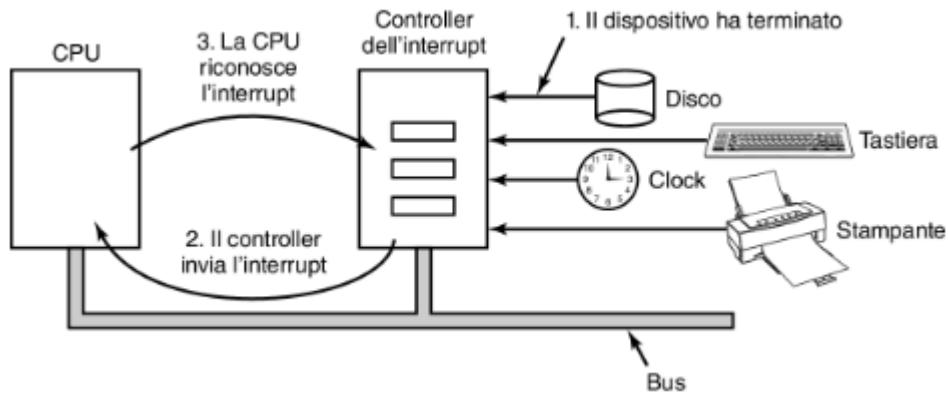
Trap: Azione deliberata del codice del programma, come una chiamata di sistema.

Fault o Eccezione: Azioni non deliberate, come errori di segmentazione o divisione per zero.

Interrupt Hardware: Segnali inviati da dispositivi come stampanti o reti alla CPU.

**Funzionamento Interrupt Hardware:** Un dispositivo di I/O invia un segnale di interrupt alla CPU tramite una linea del bus assegnata, è gestito dal interrupt controller.

**Gestione degli Interrupt da Parte del Controller:** Se non ci sono altri interrupt in corso, il controller gestisce immediatamente l'interrupt. In caso di interrupt simultanei o prioritari, il dispositivo è temporaneamente ignorato.



#### Gestione degli Interrupt:

**Segnalazione dell'Interrupt alla CPU:** Il controller assegna un numero alle linee degli indirizzi per specificare il dispositivo che richiede attenzione e invia un segnale di interruzione alla CPU.

**Interruzione e Gestione da Parte della CPU:** La CPU interrompe il suo attuale compito. Utilizza il numero sulle linee degli indirizzi come indice nella tabella del vettore degli interrupt per ottenere un nuovo contatore di programma.

**Vettore degli Interrupt:** Punta all'inizio della procedura di servizio degli interrupt corrispondente, può avere posizione fissa o variabile in memoria, con un registro della CPU che indica l'origine.

**Conferma e Gestione degli Interrupt:** La procedura di servizio conferma l'interrupt scrivendo su una porta del controller degli interrupt. Questo evita race condition tra interrupt quasi simultanei.

**Salvataggio dello Stato:** Il contatore di programma deve essere salvato per riavviare i processi interrotti. Alcune CPU salvano tutti i registri visibili e interni.

**Le Informazioni prelevate dai dispositivi di I/O** sono salvate nei registri interni o sullo stack.

**Problemi con i registri interni:** tempi morti lunghi e rischio di sovrascrittura, perciò si preferiscono gli stack corrente (processo utente) o stack del kernel.

**Problemi con lo stack corrente:** puntatori non leciti, rischio di errori di pagina.

**Uso dello stack del kernel:** cambiamento di contesto della MMU, invalidazione della cache e del TLB - overhead.

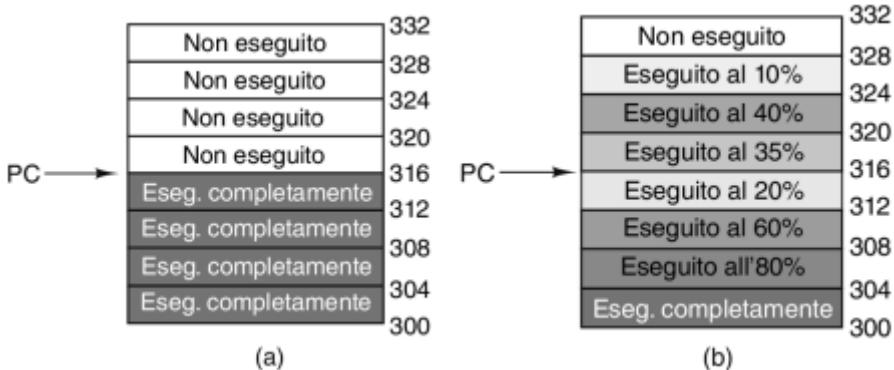
#### Il problema:

Le moderne CPU utilizzano architetture pipeline e superscalari, complicando la gestione degli interrupt. Questa complessità nasce dalla capacità delle CPU di iniziare l'esecuzione di istruzioni multiple prima del completamento delle precedenti.

#### Definizione di Interrupt Precisi e Imprecisi:

- Interrupt Precisi:** Situazione in cui il sistema può determinare con esattezza quali istruzioni sono state completate al momento dell'interrupt e quali no.
- Interrupt Imprecisi:** Condizione in cui diverse istruzioni vicino al contatore di programma

(PC) si trovano in vari stati di completamento al momento dell'interrupt, rendendo incerto lo stato esatto del programma.



**Interrupt Preciso:** Il Program Counter (PC) è salvato in un luogo noto. Tutte le istruzioni eseguite prima del PC sono completate. Nessuna istruzione dopo il PC è stata eseguita. Lo stato dell'istruzione puntata dal PC è noto.

**Gestione degli Interrupt Precisi:** La CPU cancella gli effetti di eventuali istruzioni transitorie eseguite dopo il PC. Questo approccio è utilizzato da architetture come x86 per garantire compatibilità e prevedibilità.

**Interrupt Imprecisi:** Occorre quando diverse istruzioni vicino al PC sono in vari stati di completamento. Richiede che la CPU "vomiti" una grande quantità di stato interno sullo stack.

**Problemi con gli Interrupt Imprecisi:** Rende il sistema operativo più complesso e lento. Il salvataggio di molte informazioni rallenta il processo di interrupt e ripristino.

**Impatto sull'Architettura della CPU e Sicurezza:** Gli interrupt precisi richiedono una logica interna complessa, ma semplificano la gestione del sistema operativo mentre gli interrupt imprecisi possono avere implicazioni di sicurezza poiché non tutti gli effetti sono completamente annullati.

#### Livelli del software di I/O

**Indipendenza dal Dispositivo:** Il software di I/O dovrebbe permettere l'accesso a diversi dispositivi senza specificare il tipo di dispositivo in anticipo.

**Esempio:** un programma che legge un file dovrebbe funzionare indifferentemente con dischi fissi, SSD o penne USB.

**Denominazione Uniforme:** I nomi di file o dispositivi dovrebbero essere stringhe o numeri indipendenti dal dispositivo.

**Esempio:** in UNIX, l'integrazione dei dispositivi nella gerarchia del file system consente un indirizzamento uniforme tramite nomi di percorso. Non vogliamo digitare ST6NM04 per indirizzare il primo disco rigido. /dev/sda è meglio, /mnt/movies ancora meglio.

**Gestione degli Errori:** Gli errori vanno gestiti il più vicino possibile all'hardware, idealmente dal controller stesso o dal driver del dispositivo. Errori transitori (come quelli di lettura) spesso scompaiono ripetendo l'operazione.

**Trasferimenti Sincroni vs Asincroni:** La maggior parte dell'I/O fisico è asincrono, ma per semplicità, molti programmi utente trattano l'I/O come se fosse sincrono (bloccante).

Il sistema operativo rende operazioni asincrone come se fossero bloccanti, ma fornisce

anche l'accesso all'I/O asincrono per applicazioni ad alte prestazioni. Il sistema operativo deve gestire questi trasferimenti con il DMA.

**Buffering:** Spesso i dati da un dispositivo non vanno direttamente alla destinazione finale, richiedendo un buffer temporaneo. Come ad esempio un pacchetto che arriva su un'interfaccia di rete che deve essere ricevuto e analizzato prima di capire quale applicazione (esempio browser) può usarlo. L'uso di buffer può influenzare le prestazioni, soprattutto per dispositivi con vincoli real-time.

**Dispositivi Condivisibili vs Dedicati:** Dispositivi come dischi e SSD possono essere condivisi da più utenti, mentre altri come stampanti e scanner sono tipicamente dedicati. Il sistema operativo deve gestire entrambe le categorie per evitare problemi come i deadlock (stallo).

#### Tipologie di Software per I/O:

I/O Programmato,

I/O Guidato dagli Interrupt.

I/O con DMA

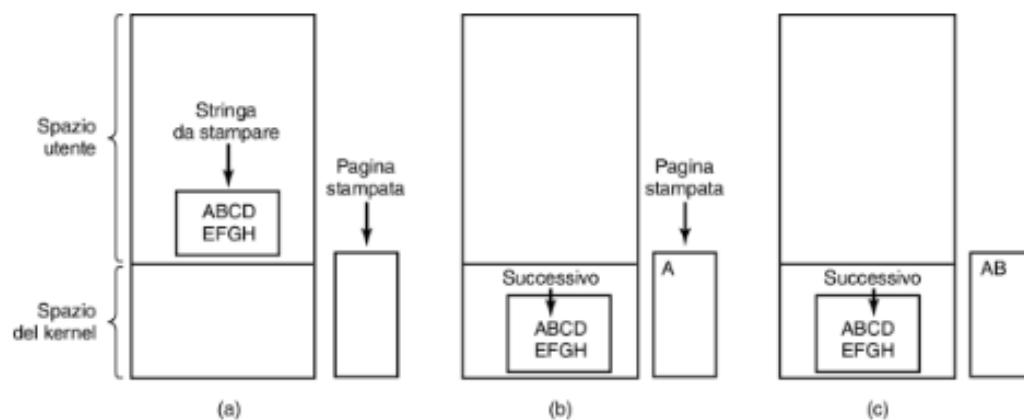
#### I/O Programmato

**Definizione di I/O Programmato:** la CPU gestisce direttamente tutto il processo di trasferimento dei dati.

**Esempio Pratico:** Un processo utente prepara una stringa "ABCDEFGH" in un buffer dello spazio utente. Il processo effettua una chiamata di sistema per stampare la stringa dopo aver ottenuto l'accesso alla stampante (a).

**Azione del Sistema Operativo:** copia il buffer in uno spazio del kernel. Invia i caratteri alla stampante uno alla volta, aspettando che questa sia pronta per ogni carattere (b e c).

**Polling o BusyWaiting:** Il sistema operativo quindi entra in un ciclo di polling, controllando il registro di stato della stampante e inviando un carattere alla volta perdendo tanto tempo.



Esempio di Codice:

```
copy_from_user(buffer, p, count);      /* p è il buffer del kernel */

for (i = 0; i < count; i++) {          /* ripeti per tutti i caratteri */

    while (*printer_status_reg != READY) ;    /* ripeti finché lo stato diventa READY */

    *printer_data_register = p[i];      /* fai l'output di un carattere */

}

return_to_user();
```

Utilizza `copy_from_user(buffer, p, count)` per copiare i dati dal buffer utente a quello del kernel. Un ciclo `for (i = 0; i < count; i++)` gestisce il trasferimento carattere per carattere alla stampante.

**Svantaggi dell'I/O Programmato:** Occupa la CPU a tempo pieno durante il processo di I/O, facendo continuamente polling sullo stato della stampante (finché non è pronta a stampare aspetta). Inefficiente in sistemi complessi dove la CPU ha altre attività importanti da gestire.  
**Applicazioni e Contesti Efficaci:** L'I/O programmato è efficace quando il tempo di elaborazione di un carattere è breve. Adatto a sistemi embedded dove la CPU non ha altre attività significative.

**Necessità di Metodi di I/O Alternativi:** Nei sistemi più complessi, il busy waiting diventa un approccio inefficiente. C'è quindi una ricerca di metodi di I/O che liberino la CPU da costanti attività di polling.

I/O Guidato dagli Interrupt

```
copy_from_user(buffer, p, count);

enable_interrupts();

while (*printer_status_reg != READY) ;

*printer_data_register = p[0];

scheduler();
```

**Scenario di Stampa senza Buffer:** Una stampante che stampa un carattere alla volta, con un ritardo di 10 ms per carattere, permettendo alla CPU di eseguire altri processi durante l'attesa.

**Utilizzo degli Interrupt:** Dopo la chiamata di sistema `copy_from_user` per stampare una stringa, il buffer viene copiato nello spazio del kernel e il primo carattere viene inviato alla stampante. La CPU poi passa l'esecuzione ad altri processi mentre attende che la stampante sia pronta per il carattere successivo.

**Cambio di Contesto e Blocco del Processo:** Il processo che ha richiesto la stampa viene bloccato fino a quando non è stampata l'intera stringa. La CPU attiva lo scheduler per eseguire altri processi durante l'attesa.

(copia dati dall'utente al kernel, abilita gli interrupt, attende che la stampante sia pronta a ricevere caratteri, invia il primo carattere alla stampante, passa il controllo a un altro processo).

Successivamente è eseguita la seconda parte di codice (la procedura di servizio di interrupt della stampante):

```

if (count == 0) {

    unblock_user();

} else {

    *printer_data_register = p[i];

    count = count - 1;

    i = i + 1;

}

acknowledge_interrupt();

return_from_interrupt();

```

**Generazione dell'Interrupt da Parte della Stampante:** La stampante genera un interrupt quando è pronta per il carattere successivo, interrompendo il processo corrente e salvandone lo stato.

**Esecuzione della Procedura di Servizio Interrupt:** Viene eseguita la procedura di servizio di interrupt per la stampante. Se ci sono altri caratteri da stampare, il gestore stampa il successivo.

**Sblocco del Processo e Ritorno dall'Interrupt:** Se tutti i caratteri sono stati stampati, il gestore degli interrupt esegue azioni per sbloccare il processo utente. Riconosce l'interrupt e ritorna al processo interrotto, che riprende l'esecuzione da dove era stato lasciato.

**Problema:** Interrupt ad ogni carattere non è la soluzione ottimale!

(controlla se tutti i caratteri sono stati stampati, se si sblocca il processo utente che ha richiesto la stampa, se no invia il carattere successivo alla stampante, decrementa il contatore, e passa al carattere successivo, riconosce l'interrupt ricevuto dalla stampante, ritorno dall'interrupt permettendo alla CPU di riprendere le altre operazioni).

## I/O con DMA

**Principio del DMA:** Il DMA riduce il numero di interrupt, passando da uno per ogni carattere a uno per buffer. Libera la CPU per eseguire altre attività durante il trasferimento di I/O.

**Setup e Inizio del Trasferimento (a):**

Preparazione dei Dati: `copy_from_user`.

Configurazione DMA: `set_up_DMA_controller`.

Ottimizzazione delle Risorse CPU: `scheduler`.

```

copy_from_user(buffer, p, count);

set_up_DMA_controller( );

scheduler( );

```

(Copia i dati dell'utente al kernel, Impostazione del controllore DMA per il trasferimento, la CPU esegue altri processi mentre il DMA gestisce il trasferimento).

**Gestione dell'Interrupt e Conclusione (b):**

Interrupt generato dal completamento del trasferimento DMA: `acknowledge_interrupt`.

Ripresa del Processo Utente: unlock\_user.

Ritorno dal Contesto dell'Interrupt: return\_from\_interrupt.

```
acknowledge_interrupt( );
unlock_user( );
return_from_interrupt( );
```

(Riconosce l'interrupt ricevuto dal DMA, sblocca il processo utente dopo che il trasferimento è completo, ritorna dall'interrupt consentendo alla CPU di proseguire con altre operazioni).

### Struttura del Software di I/O

Organizzazione a Quattro Livelli: Il software di I/O è strutturato in quattro livelli distinti, ogni livello ha funzioni e interfacce specifiche. Le funzionalità e le interfacce variano a seconda del sistema operativo, l'analisi dettagliata parte dal livello più basso.



### Gestione degli Interrupt

Blocco dei Driver: Durante l'I/O, i driver vengono bloccati (es. con semafori, anche se è più complicato) fino al completamento dell'I/O e all'arrivo dell'interrupt.

La gestione degli interrupt richiede diversi passaggi, inclusi salvataggio dei registri, impostazione di contesti e conferme al controller degli interrupt.

Su sistemi con memoria virtuale, la gestione degli interrupt richiede passaggi aggiuntivi per gestire MMU, TLB e cache, aumentando la complessità e i cicli macchina necessari.

L'elaborazione di un interrupt quindi richiede numerosi cicli CPU e varia notevolmente a seconda del sistema e dell'architettura.

Di seguito una serie di passaggi da eseguire nel software dopo il completamento dell'interrupt hardware, i dettagli dipendono molto dal sistema (in alcune macchine i seguenti passi potrebbero essere ordinati differentemente o non esserci):

1. Salvataggio dei Registri: Salvataggio di tutti i registri, inclusi quelli non salvati dall'interrupt hardware.
2. Impostazione del Contesto: Impostazione di un contesto per la procedura di servizio dell'interrupt, incluso il setup di TLB, MMU e una tabella delle pagine.
3. Impostazione dello Stack: Configurazione di uno stack per la procedura di servizio dell'interrupt.
4. Conferma al Controller degli Interrupt: Conferma al controller degli interrupt e riabilitazione degli interrupt, se necessario.
5. Copia dei Registri nella tabella dei Processi: Copia dei registri salvati nella tabella dei

processi.

6. Esecuzione della Procedura di Servizio dell'Interrupt: Estrazione delle informazioni dai registri del controller del dispositivo che ha generato l'interrupt.
7. Scelta del Processo Successivo: Determinazione di quale processo eseguire come successivo, potenzialmente uno con priorità alta sbloccato dall'interrupt.
8. Impostazione del Contesto per il Nuovo Processo: Impostazione del contesto della MMU e potenzialmente del TLB per il processo successivo.
9. Caricamento dei Nuovi Registri del Processo: Caricamento dei registri, inclusi PSW, del processo successivo.
10. Avvio del Nuovo Processo: Inizio dell'esecuzione del processo selezionato.

#### Driver dei dispositivi

Ruolo dei Driver di Dispositivo: Gestiscono i dispositivi di I/O attraverso registri di dispositivi specifici, i driver sono diversi per ciascun tipo di dispositivo (es. mouse vs disco rigido), al più gestiscono un tipo o una classe di dispositivi correlati (ma spesso un unico dispositivo), e ogni dispositivo necessita di un codice specifico, noto come driver di dispositivo, solitamente fornito dal produttore.

Esempi di Tecnologie Basate su Driver Comuni: Tecnologie come USB utilizzano una pila di driver per gestire una vasta gamma di dispositivi. Sono composti da livelli diversi per gestire aspetti specifici dei dispositivi USB, ogni livello «parla» con il livello inferiore.

Livello di Base: Gestione dell'I/O seriale e delle questioni hardware.

Livelli Superiori: Trattano pacchetti dati e funzionalità comuni condivise dalla maggior parte dei dispositivi USB.

API di Alto Livello: Forniscono interfacce specifiche per diverse categorie di dispositivi.

Posizionamento nel Kernel: I driver di solito fanno parte del kernel del sistema operativo per poter accedere ai registri del controller del dispositivo (l'installazione richiede il riavvio).

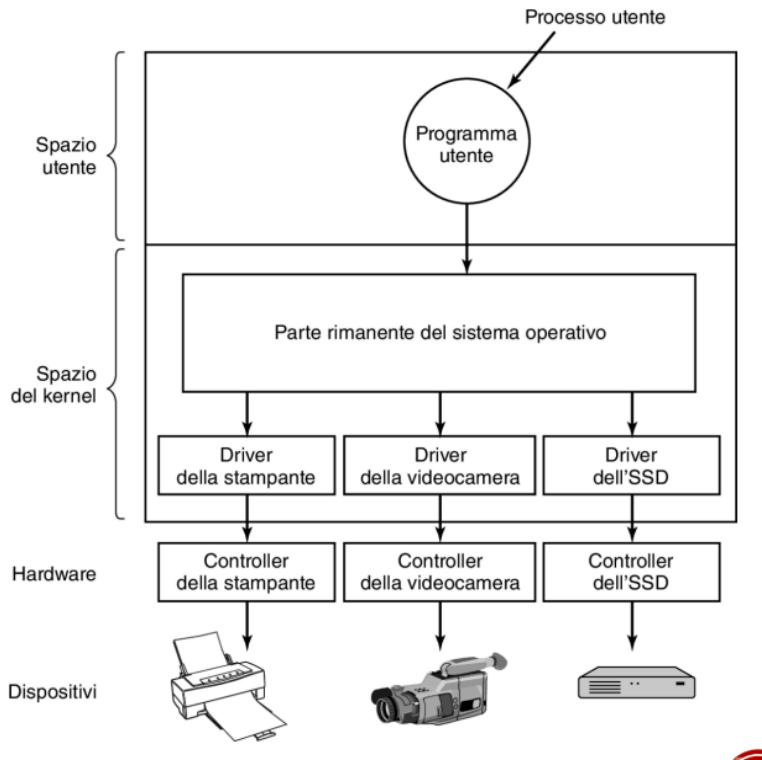
Se usati nello spazio utente sono più facili da installare, mettono meno «a rischio» il Sistema Operativo ma sono più lenti (occorre passare allo spazio kernel per ogni operazione).

Interfaccia con il Sistema Operativo: Il sistema operativo deve permettere l'installazione di codice scritto da terze parti (driver). I driver si posizionano sotto il resto del sistema operativo. Ogni categoria ha un'interfaccia standard che i driver devono supportare.

Classificazione dei Driver: I sistemi operativi classificano i driver in categorie come dispositivi:

- a blocchi: come i dischi, contenenti molteplici blocchi di dati indirizzabili indipendentemente.
- a caratteri: come stampanti e tastiere, che generano o accettano un flusso di caratteri.

Caricamento dei Driver: In alcuni sistemi, i driver sono inclusi nel programma binario del sistema operativo, quindi aggiunto un dispositivo nuovo il kernel andava ricompilato!!! Nei sistemi moderni, i driver vengono caricati dinamicamente.



I driver hanno diverse funzioni, tra cui: Gestione di letture e scritture, inizializzazione del dispositivo, gestione dell'alimentazione e del registro degli eventi.

Processo Generale di un Driver: Verifica della validità dei parametri di input, traduzione dei parametri in comandi specifici per il dispositivo, e gestione dell'uso del dispositivo.

I driver gestiscono l'I/O e controllano eventuali errori. In alcuni casi, un driver deve aspettare l'interrupt per completare l'operazione.

Complessità e Rientranza dei Driver: I driver devono essere rientranti per gestire più richieste simultaneamente. Esempio: mentre sta gestendo un pacchetto di informazioni il driver viene richiamato anche per un altro pacchetto. Devono gestire le situazioni complesse come l'aggiunta o la rimozione di dispositivi in sistemi "hot pluggable". Esempio: Se viene disconnesso un dispositivo mentre si sta leggendo/scrivendo il sistema operativo deve «ripulire» tutte le operazioni in corso e impedire nuove richieste al dispositivo assente.

#### Software del sistema operativo indipendente dal dispositivo

Il software di I/O indipendente dal dispositivo funge da intermediario tra i driver specifici dei dispositivi e le applicazioni utente, mira a semplificare l'interazione con i dispositivi hardware offrendo un'interfaccia uniforme e gestendo operazioni comuni.

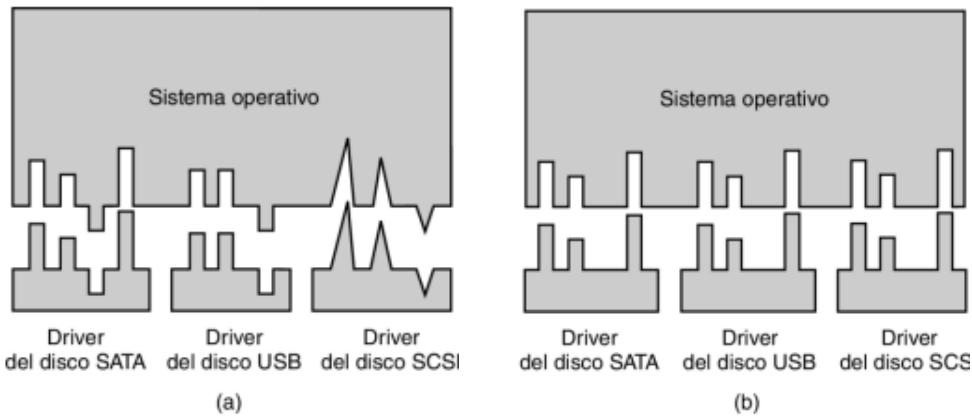
#### Funzioni Chiave:

1. Interfaccia Uniforme dei Driver dei Dispositivi: Fornisce un'interfaccia standard per diversi tipi di driver di dispositivo.
2. Buffering: Gestisce i buffer per l'efficienza del trasferimento dei dati tra i dispositivi e il sistema.
3. Segnalazione degli Errori: Identifica e comunica gli errori provenienti dai dispositivi all'utente o ad altri sistemi.
4. Allocazione e Rilascio dei Dispositivi Dedicati: Gestisce l'assegnazione e la liberazione di dispositivi dedicati a specifici compiti o utenti.
5. Dimensione dei Blocchi Indipendente dal Dispositivo: Assicura che la dimensione dei

blocchi di dati sia gestita in modo uniforme, indipendentemente dalla specificità del dispositivo.

Una questione fondamentale di un sistema operativo è come rendere tutti i dispositivi e i driver più o meno simili tra loro. Evitando la necessità di modificare il sistema operativo ogni volta che viene introdotto un nuovo dispositivo.

Interfacce Diverse (a) vs Interfaccia Standard (b): alcuni driver hanno interfacce uniche verso il sistema operativo ciò significa che le funzioni dei driver chiamabili sono diverse a seconda del driver, nel secondo caso tutti i modelli hanno la stessa interfaccia comune, significa che le funzioni dei driver chiamabili sono le stesse per tutti questi modelli.



**Definizione di Funzioni per Classe di Dispositivi:** Ogni classe di dispositivi ha un insieme definito di funzioni che i driver devono supportare (es. operazioni di lettura/scrittura per dischi).

**Tabella di Puntatori a Funzioni nel Driver:** I driver includono una tabella con puntatori a funzioni richieste, utilizzata dal sistema operativo per facilitare chiamate indirette.

C'è una mappatura dei nomi simbolici dei dispositivi ai driver corrispondenti (es. /dev/disk0 in UNIX), gestione dei permessi e protezione dei dispositivi simile a quella dei file, consentendo un controllo amministrativo appropriato.

Questa struttura fornisce un'interfaccia coesa fra i driver e il resto del sistema operativo, semplificando l'integrazione di nuovi dispositivi.

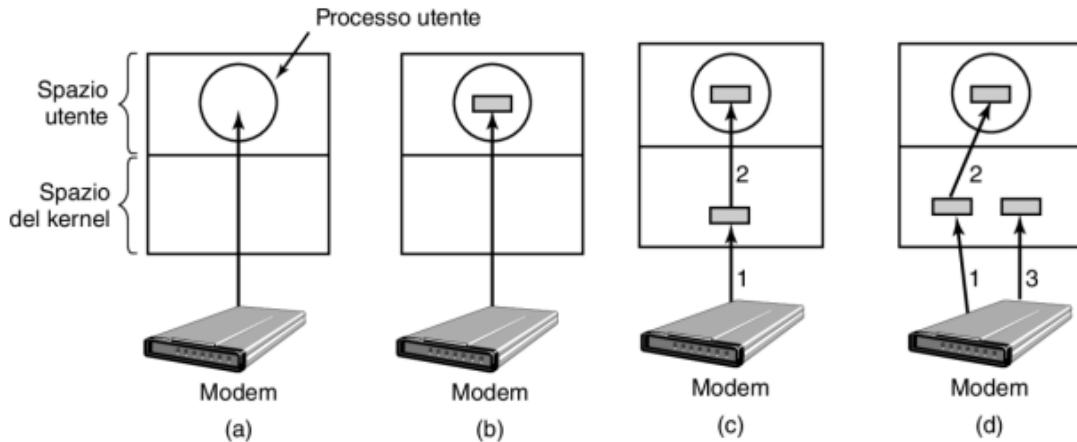
**Scenari di Buffering in Input (da kernel a utente):**

(a) Esempio di lettura dati da un modem VDSL: l'input senza uso di buffer è inefficiente poiché richiede un riavvio del processo utente per ogni carattere ricevuto.

(b) Miglioramento con buffer nello spazio utente: il processo fornisce un buffer e si blocca solo quando è pieno.

(c) Problemi di paginazione e soluzione con buffer nel kernel: il buffer nel kernel accumula i caratteri, riducendo il riavvio del processo utente.

(d) Doppio Buffer nel Kernel (così che il driver sia rientrante): Soluzione per gestire i caratteri in arrivo durante la lettura del buffer utente dal disco. Utilizzo di due buffer nel kernel che si alternano: uno accumula nuovi input mentre l'altro è in copia nello spazio utente.



### Buffering in Output:

Esempio di output su un modem: senza buffering (b), il processo utente può restare bloccato per lungo tempo, la soluzione utilizza un buffer nel kernel: copia dei dati in un buffer del kernel e sblocco immediato del processo utente.

**Problemi e Complessità del Buffering:** Il buffering multiplo può influire negativamente sulle prestazioni, come illustrato nella rete. (copia multi-stadio) Dal buffer utente al kernel, poi al controller, successivamente sulla rete, e infine al buffer del kernel e processo ricevente.

**Impatto del Buffering sulla Velocità di Trasmissione:** Le molteplici copie richieste per la trasmissione di pacchetti rallentano la velocità effettiva di trasmissione, la sequenzialità delle operazioni di copia aumenta il tempo complessivo di trasmissione.

**Frequenza e Tipi di Errori di I/O:** Gli errori di I/O sono comuni e variano da errori di programmazione a veri errori di I/O, errori di programmazione includono azioni come la scrittura su un dispositivo di input o la lettura da un dispositivo di output.

**Risposta ai Diversi Errori:** Errori di programmazione: Ritornano un codice d'errore al processo chiamante mentre veri errori di I/O (es. scrittura su un blocco danneggiato) sono gestiti dal driver o, se non risolvibili, passati al software indipendente dal dispositivo.

Le azioni possono dipendere dal contesto:

In presenza di un utente interattivo c'è la possibilità di dialogo per scegliere come gestire l'errore (riprova, ignora, termina processo), senza utente interattivo la chiamata di sistema fallisce restituendo un codice d'errore.

In caso di danneggiamento di strutture dati critiche c'è la visualizzazione di un messaggio d'errore e possibile terminazione del sistema.

**Uso Esclusivo di Alcuni Dispositivi:** Dispositivi come stampanti richiedono l'uso esclusivo da un singolo processo alla volta.

Il sistema operativo valuta le richieste per l'uso del dispositivo, accettandole o rifiutandole a seconda della disponibilità del dispositivo.

**Metodi di Allocazione e Rilascio:**

- **Approccio Tradizionale:** I processi eseguono una 'open' su file speciali per i dispositivi e la 'close' del dispositivo rilascia il file.
- **Approccio Alternativo:** Si utilizzano meccanismi speciali per richiedere e rilasciare dispositivi dove un tentativo di acquisizione non riuscito causa il blocco del processo richiedente, i processi bloccati sono inseriti in una coda (coda di spooling) e acquisiscono il dispositivo quando diventa disponibile.

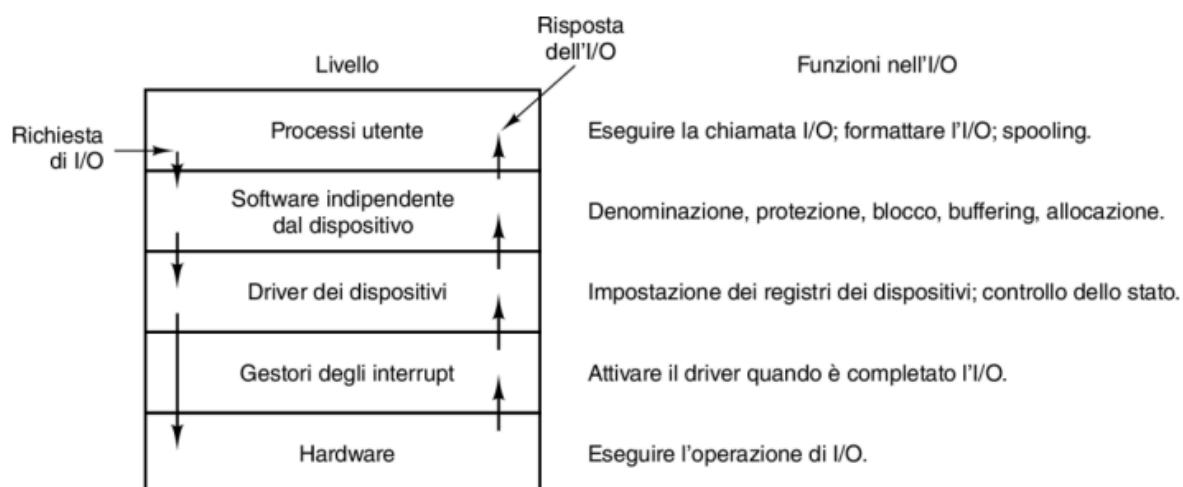
**Definizione di Spooling:** Tecnica per gestire dispositivi dedicati in ambienti multiprogrammati, evitando il blocco prolungato da parte di un unico processo.

**Implementazione Pratica:** Utilizzo di un processo daemon e una directory di spooling per ordinare e gestire i lavori di stampa.

Con lo spooling si incrementa l'efficienza nell'uso dei dispositivi dedicati e si migliora la gestione delle risorse, permettendo a più utenti o processi di accedere ai dispositivi in modo controllato e sequenziale.

**Interazione e Flusso di Controllo:** Descrive come una richiesta di I/O, ad esempio la lettura di un blocco da un file, attraversa diversi livelli (hardware, gestori degli interrupt, driver dei dispositivi).

**Gestione delle Richieste di I/O:** Spiega il processo dalla richiesta iniziale all'intervento degli interrupt e al successivo risveglio del processo utente, enfatizzando il ruolo cruciale di ogni livello nel trattamento efficiente delle operazioni di I/O.



**Variabilità nelle Dimensioni Fisiche:** Dispositivi come SSD e dischi rigidi presentano dimensioni fisiche di blocchi o settori variabili, anche i dispositivi a caratteri possono differire nella quantità di dati che trasmettono per volta.

Il Software Indipendente dal Dispositivo nasconde le differenze fisiche nelle dimensioni dei blocchi o settori tra diversi dispositivi e fornisce una dimensione di blocco logico uniforme ai livelli superiori del sistema. Trasforma più settori o pagine flash in un unico blocco logico e permette ai livelli superiori di interagire con dispositivi "astratti" che utilizzano una dimensione di blocco logico standard, indipendentemente dalle dimensioni fisiche.

Gestisce così la varianza nella quantità di dati trasmessi dai dispositivi a caratteri (es. mouse vs interfacce di rete), rendendo queste differenze trasparenti ai livelli superiori.

**Ruolo delle Librerie di I/O:** Facilitano le chiamate di sistema per l'I/O, esemplificato dal metodo `write(fd, buffer, nbytes)` in C.

**Funzioni di Libreria per Formattazione:** `printf()` e `scanf()` trasformano e gestiscono i dati prima di invocare funzioni di sistema, facilitando operazioni di input e output.

Queste librerie semplificano la programmazione di I/O, permettendo ai programmati di concentrarsi sulla logica dell'applicazione piuttosto che sui dettagli di basso livello delle operazioni di I/O.