

**DISPENSA SISTEMI OPERATIVI**

**(2024/2025)**

## LEZ2

Il sistema operativo è un software che agisce da strato intermedio tra le applicazioni e le componenti di basso livello del computer. Il SO deve essere in grado di gestire più programmi in esecuzione e più utenti: ciò necessita allocazione ordinata e controllata di processi, memoria, unità di I/O, multiplexing (nel tempo e nello spazio).

Storia di sistemi operativi e macchine:

1. Prima generazione (1945-55): Vacuum tubes
2. Seconda generazione (1955-65): Transistors and batch systems (utilizzo di schede)
3. Terza generazione (1965-1980): ICs and multiprogramming

Introduzione di MULTICS : uno dei primi sistemi operativi ad introdurre l'idea di Timesharing, ha introdotto molte delle caratteristiche che ritroviamo oggi; non ha avuto successo perché estremamente complesso.

A seguito di Multics si arriva a UNIX: un sistema operativo multiutente e con multiprogrammazione diventato fin da subito molto popolare in ambito accademico e aziendale.

A seguito di un' emergenza, nata per via delle diverse versioni di UNIX che portavano ad una mancanza di standardizzazione, nascono diversi progetti che mirano a ottenere un'interfaccia standard per programmare in UNIX: tra i più importanti troviamo POSIX (Portable Operating System).

4. Quarta generazione (1980-present): Personal computers

Nel 1987 Tanenbaum sviluppa MINIX, un piccolo sistema UNIX (11800 righe di codice C e 800 righe di codice Assembler) compatibile con gli standard POSIX.

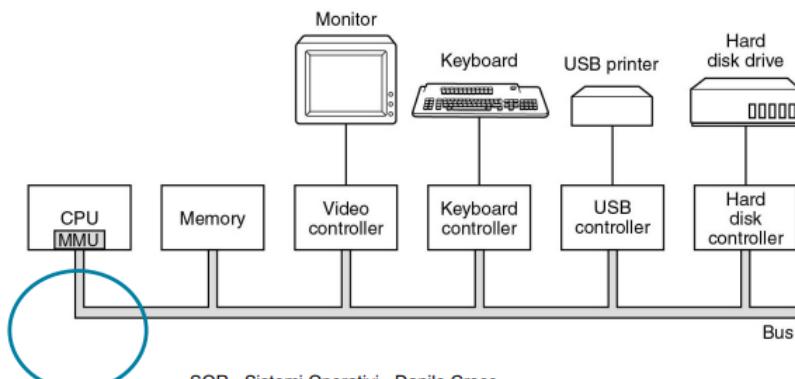
5. Quinta generazione (1990-present): Mobile computers

Da MINIX A LINUX (1990S)

Windows 2000 e successori consolidano la posizione di Microsoft.

Apple consolida il suo sistema operativo basato su UNIX ma architecture-specific

Uno sguardo all'hardware:



### PROCESSORE

È il cervello del computer, esegue istruzioni dalla memoria.

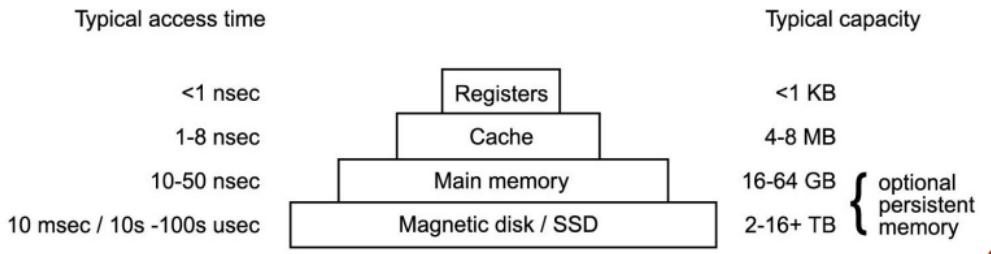
- Il ciclo base della CPU: preleva (fetch), decodifica (decode), esegue (execute) istruzioni. I programmi vengono eseguiti in questo ciclo.
- Le CPU eseguono un set specifico di istruzioni: Registri interni memorizzano dati importanti e risultati. I set di istruzioni includono funzione per il caricamento/salvataggio dati dalla memoria.
- Registri speciali come il program counter (PC) indicano l'istruzione successiva.
- Lo stack pointer punta alla cima dello stack in memoria.
- Il registro Program Status Word (PSW) contiene informazioni sullo stato del programma. Il PSW è fondamentale per chiamate di sistema e I/O.

Il sistema operativo gestisce il multiplexing temporale della CPU: Durante il multiplexing, il sistema operativo salva e ripristina i registri e ciò permette al sistema operativo di eseguire programmi in modo efficiente. Progettazioni avanzate: pipeline.

Possono esserci più processori, si parla così di:

- Multithreading (o hyperthreading): tiene all'interno della CPU lo stato di due thread ma non c'è una esecuzione parallela vera e propria
- Multiprocessori: hanno tra i vantaggi: throughput; economia di scala; affidabilità.

## MEMORIA



## DISPOSITIVI DI I/O

Si individuano due componenti:

- il controller: più semplice da usare per il SO, ogni controller ha bisogno di un driver per il S.O. che interagisce con il controller attraverso le porte di I/O tramite istruzioni di tipo IN/OUT e mappatura in memoria.
- il dispositivo in sé: interfaccia elementare ma complicata da pilotare

Per eseguire l'I/O:

1. Il processo esegue la chiamata di sistema
2. Il kernel effettua una chiamata al driver
3. Il driver avvia l'I/O e interroga il dispositivo per vedere se ha finito (è occupato in attesa) o chiede al dispositivo di generare un interrupt quando ha finito (e restituisce qualche informazion)
4. più avanzato: fa uso di hardware speciale (DMA). DMA è l'acronimo di "Direct Memory Access" ed è un dispositivo hardware speciale che consente ai componenti di accedere direttamente alla memoria del computer senza coinvolgere la CPU.

## DMA

DMA è l'acronimo di "Direct Memory Access", è un dispositivo hardware speciale che consente ai componenti di accedere direttamente alla memoria del computer senza coinvolgere la CPU.

Migliora l'efficienza ed aumenta 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

Il bus PCIe è il principale e più veloce bus di comunicazione nei computer attuali.

La CPU comunica con la memoria attraverso un bus veloce DDR4.

## BOOT

La memoria flash della scheda madre contiene il firmware (il BIOS). Dopo aver premuto il pulsante di accensione, la CPU esegue il BIOS che inizializza RAM e altre risorse, poi esegue la scansione dei bus PCI/PCIe e inizializza i dispositivi. A questo punto 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. Il BIOS è in grado di leggere semplici file system (ad esempio, FAT-32) e avvia il primo programma di bootloader (dalla partizione indicata dal boot manager). Il bootloader può caricare altri programmi di bootloading. Alla fine viene caricato l'OS.

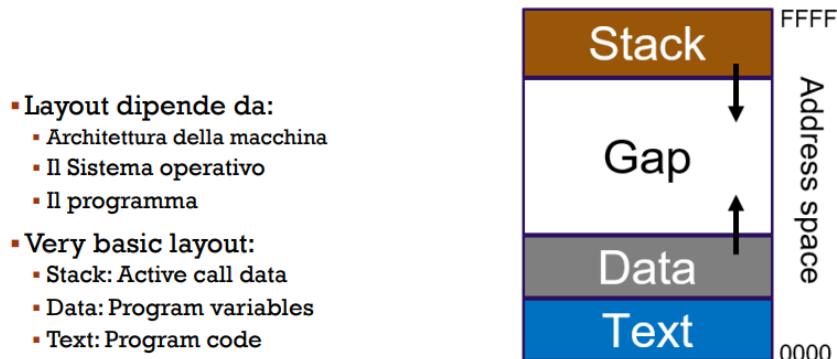
Ci sono diversi tipi di SO, nati per esigenze diverse:

1. SO per Mainframe (funzioni principali: esecuzione senza interfaccia utente, gestione di numerose richieste simultanee, esecuzione di operazioni da parti di più utenti)
2. SO per Server (funzioni principali: fornitura servizi come file sharing, database o hosting web)
3. SO per Personal Computer (caratteristiche: facilità d'uso, utilizzo per compiti quotidiani)
4. SO per Smartphone e Tablet (supportano GPS, fotocamere e numerose app di terze parti)
5. SO per IOT (tipicamente leggeri con funzioni specifiche e limitate)
6. SO per Sistemi Real Time (devono rispettare scadenze rigide)
7. SO per Smart Card (utilizzati in smart card per pagamenti e autenticazioni)

## PROCESSO

È un programma in esecuzione. I processi sono astrazioni a livello utente per eseguire un programma per conto dell'utente, ogni processo ha il proprio spazio di indirizzamento e i dati coinvolti nell'elaborazione vengono recuperati e memorizzati in file. I file persistono rispetto ai processi.

Visivamente un processo può essere pensato come un contenitore contenente tutte le informazioni necessarie per l'esecuzione del programma. Layout di un processo:



Le informazioni sui processi sono conservate nella tabella dei processi del sistema operativo. Un processo sospeso consiste in una voce della tabella dei processi (registri salvati e altre informazioni necessarie per riavviare il processo) e nel suo spazio degli indirizzi. La gestione dei processi si occupa di operazioni come la creazione, la terminazione, la pausa e la ripresa di un processo. Inoltre, un processo può creare un altro processo, e quest'ultimo è conosciuto come processo "figlio" creando così una gerarchia (o "albero") di processi.

I processi sono "di proprietà" di un utente, identificato da un UID. 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. Un processo (superuser/root/administrator) è speciale perché ha più privilegi degli altri.

## FILE ("EVERYTHING IS A FILE")

È l'astrazione di un dispositivo di memorizzazione. 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. Le directory e i file formano una gerarchia che inizia dalla directory principale detta root, è possibile accedere ai file tramite percorsi assoluti (absolute path: /home/ast/todo-list) o percorsi relativi a partire dalla directory di lavoro corrente ( ../courses/slides1.pdf).

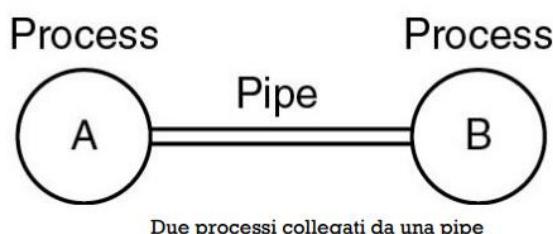
I file sono "protetti" da tuple a tre bit per il proprietario (owner), il gruppo (group) e gli altri utenti (other users). Le tuple contengono un bit (r)ead, (w)rite e un bit e(x)ecute (ma sono disponibili più bit). Esempio:

```
-rwxr-x--x myuser my group 14492 Dec 4 18:04 my file
```

Owner ha il permesso di execute, modify, read del file;  
 Group ha il permesso di read e execute del file;  
 Other users hanno il solo permesso di execute del file.

## PIPE

Per pipe intendiamo uno pseudo file che consentono ai processi di comunicare su un canale FIFO. Deve essere impostato in anticipo e sembra un "file" normale per leggere e scrivere dai/sui processi in esecuzione.



## SYSTEM CALL / CHIAMATE DI SISTEMA

Il sistema operativo offre funzionalità attraverso le chiamate di sistema (system call), gruppi di chiamate di sistema implementano servizi. Le chiamate di sistema, quindi, sono l'interfaccia che il sistema operativo offre alle applicazioni per richiedere servizi. Il meccanismo delle chiamate di sistema è altamente specifico del sistema operativo e dell'hardware, e la necessità di efficienza esaspera questo problema risolvibile incapsulando le chiamate di sistema nella libreria C.

I passi per effettuare una chiamata di sistema sono:

1. Preparazione dei parametri memorizzandoli nei registri e nello stack
2. Effettuazione della chiamata alla procedura di libreria
3. Collocazione del numero della chiamata di sistema in un registro
4. Passaggio a modalità kernel eseguendo una istruzione trap(istruzione trappola per generare interruzioni)
5. Esecuzione del gestore di chiamate di sistema che prende in carico la chiamata
6. Restituzione controllo della chiamata alla procedura di libreria utente
7. Possibilità di blocco per eseguire altri processi (ad esempio per via di risorse non disponibili per eseguirla)
8. Ripresa dopo il blocco

Alcuni esempi di chiamate di sistema (POSIX) sono le seguenti

System Call per gestione dei processi:

Call	Description
pid = fork()	Creare un processo figlio identico al genitore
pid = waitpid(pid, &statloc, options)	Attendere che un processo figlio termini
s = execve(name, argv, environp)	Sostituire l'immagine centrale di un processo
exit(status)	Terminare l'esecuzione del processo e restituire lo stato

Call	Description
fd = open(file, how, ...)	Apre un file per la lettura, la scrittura o entrambe le operazioni.
s = close(fd)	Chiude un file aperto
n = read(fd, buffer, nbytes)	Legge dati da un file in un buffer
n = write(fd, buffer, nbytes)	Scrive dati da un buffer in un file
Position = lseek(fd, offset, whence)	Sposta il puntatore del file
s = stat(name, &buf)	Ottiene informazioni sullo stato di un file

- Il codice di ritorno s è -1 se si è verificato un errore. I codici di ritorno sono i seguenti: fd è un descrittore di file, n è un conteggio di byte, position è un offset all'interno del file.

System Call per gestione del file system:

Call	Description
s = mkdir(name, mode)	Crea una nuova directory
s = rmdir(name)	Rimuove una directory vuota
s = link(name1, name2)	Crea una nuova voce, nome2, che punta a nome1
s = unlink(name)	Rimuove una voce della directory
s = mount(special, name, flag)	Monta un file system
s = umount(special)	Smonta un file system

Altre chiamate di sistema:

Call	Description
s = chdir(dirname)	Cambia la directory di lavoro
s = chmod(name, mode)	Modifica i bit di protezione di un file
s = kill(pid, signal)	Invia un segnale a un processo (NON UCCIDE)
s = time(&seconds)	Ottiene il tempo trascorso dal 1° gennaio 1970

## LEZ3

### SO MONOLITICO

Utilizza un approccio al design «tutto in uno»

- il kernel è un'unica unità grande e interconnessa, si parla di kernel unificato in quanto tutte le funzionalità sono centralizzate in un unico kernel
- 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

Questo offre grande 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 sistematici.

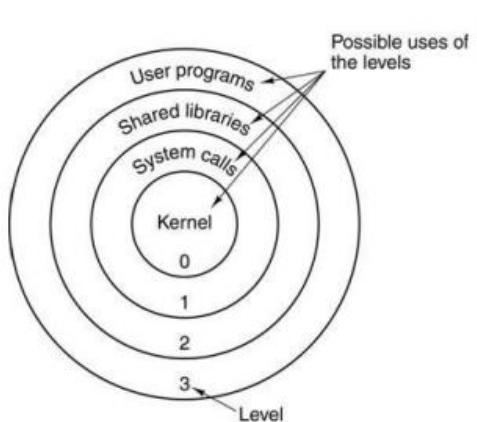
Per quanto riguarda compilazione e collegamento, tutte le funzioni e procedure del sistema operativo devono essere compilate e collegate in un unico eseguibile del kernel.

In questo modello tutte le procedure possono teoricamente accedere a qualsiasi altra procedura o variabile all'interno del kernel e non c'è un vero e proprio "occultamento" o isolamento tra le diverse parti del sistema.

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

Un sistema operativo monolitico ha 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.

La struttura di un sistema operativo monolitico si basa su una organizzazione stratificata, come da immagine

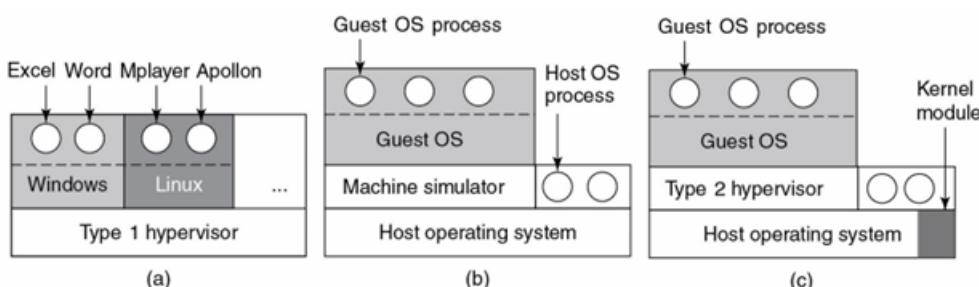


### VIRTUALIZZAZIONE

Per virtualizzazione si intende aggiungere software in mezzo che tramuta le istruzioni del sistema operativo in istruzioni macchina.

Virtual machine monitor (VMM) o Hypervisor emula l'hardware:

- Type 1: viene eseguito sul "pezzo di ferro" (direttamente su HW, come Xen) ("parte la macchina che fa partire hypervisor senza bisogno di sistema operativo")
- Type 2: VMM ospitato nel sistema operativo (esempio: VirtualBox)
- Hybrid: VMM all'interno del sistema operativo (esempio: KVM)



(a) A type 1 hypervisor. (b) A pure type 2 hypervisor. (c) A practical type 2 hypervisor

## 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.

Gli svantaggi dei container sono che

1. Non è possibile eseguire un container con un sistema operativo completamente diverso da quello dell'host
2. A differenza delle macchine virtuali, non esiste un rigido partizionamento delle risorse.
3. Se un container altera la stabilità del kernel sottostante, ciò può influire sugli altri container (perché i container sono isolati a livello di processo)

Esempio: docker, ha vantaggi di isolamento ("scatole di sabbia") e permette di non avere numerose copie di file di sistema; inoltre è leggero (intendo nel senso più piccolo come dimensione, meglio docker che iso del sistema operativo)

## EXOKERNEL

Informalmente, è una versione semplificata dell'idea di container, non vuole emulare l'hw ma fornisce un kernel con meno chiamate (solo le fondamentali), questo porta ad una maggiore sicurezza perché meno cose puoi fare e meno sei vulnerabile verso l'esterno.

Idea: Separare il controllo delle risorse dalla macchina estesa

È simile a un VMM/Hypervisor, ma non emula l'hardware e fornisce solo una condivisione sicura delle risorse a basso livello. 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, concentrandosi solo su quale macchina virtuale ha accesso a quali risorse.

## UNIKERNEL

Gli unikernel sono sistemi minimi progettati per eseguire una singola applicazione su una macchina virtuale. Esempio: 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 (esiste solo un'applicazione per macchina virtuale)

## MICROKERNEL-BASED CLIENT/SERVER

Utilizzato in sistemi ad alta affidabilità.

I processi di sistema comunicano attraverso il passaggio di messaggi, le chiamate di sistema si basano sullo stesso meccanismo di messaggistica. Il meccanismo di messaggistica è implementato nel kernel minimale.

Pro: Ogni processo del sistema operativo può fare solo ciò che è necessario per svolgere il proprio compito, inoltre la compromissione di un driver 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)

---

Alcuni comandi importanti da conoscere:

- pwd (abbreviazione dalla lingua inglese di print working directory, stampa la directory corrente)
- mkdir è un comando che crea una o più directory
- chmod è utilizzato per modificare i permessi di accesso di file e directory
- ls è usato per elencare file o cartelle
- ps aux serve a vedere tutti i processi in esecuzione (a mostra i processi per tutti gli utenti, u = mostra l'utente/proprietario del processo, x = mostra anche i processi non collegati a un terminale)
- ln è usato per creare un collegamento a un file o a una directory (abbreviazione di link)

LEZ3\_1

## Utilities linux per selected text processing:

- awk per eseguire piccoli script (modello di scansione ed elaborazione di pattern)
  - cat legge i file che gli sono specificati come parametri (o lo standard input) e produce sullo standard output la concatenazione del loro contenuto
  - cut permette di selezionare alcune colonne in un file (estrai campi selezionati di ogni riga)
  - diff confronta due file
  - grep cerca una specifica stringa (cerca il testo in un pattern)
  - head visualizza la prima parte dei file (generalmente le prime 10 righe)
  - less visualizzazione dei file pagina per pagina
  - od permette di convertire in diversi formati (dump di file in vari formati)
  - sed permette di sostituire delle lettere in un file (stream editor)
  - sort ordina i file di testo (scrive concatenazioni ordinate di file sullo standard output)
  - split divide file (divide grandi file in blocchi)
  - tail visualizza l'ultima parte di un file
  - tr traduce/elimina caratteri (permette di sostituire un singolo carattere)
  - uniq permette di rimuovere linee duplicate (filtrà le righe ripetute in un file)
  - wc conteggio di righe, parole e caratteri
  - tar archivio file (simile a zip)

Per sapere come funziona comando: "man nomeComando" (sarebbe il manuale) (da wlc).  
pipe = file speciale che permette di comunicare tra i processi. Simbolo: | (Nota: tutto è un file).

SHELL

Interpretatori di script  
È un programma che interpreta i comandi digitati e li invia al sistema operativo. Sui sistemi Linux (e altri, come DOS/Windows), fornisce anche una serie di comandi integrati e strutture di controllo della programmazione, variabili di ambiente, ecc.

La maggior parte dei sistemi Linux supporta almeno due shell: TCSH e BASH. La shell predefinita è BASH.

La variabile d'ambiente Bash utilizzata per il valore del path corrente è \$PATH.

Per vedere tutte le variabili d'ambiente si può usare il comando `printenv`.

Per creare variabili in Bash utilizzare l'operatore "=", esempio: `foo="this is foo's value"` (le variabili sono molto utili negli scripts).

I comandi della shell hanno tre parti: comando , opzione , parametri.

Esempio: `cal -j 3 1999` cal è il comando , -j è l'opzione , 3 e 1999 sono i parametri

Per eseguire uno script o un programma nella directory corrente tramite shell basta scrivere `./exec name`

## REDIRECT

Ogni programma sputa verso l'esterno informazioni, utilizzando la pipe possiamo concatenare più comandi .

esempio: `w | grep 'danilo'` (|=pipe)

Alcuni esempi sono:

w   wc	conta linee, parole e caratteri
w   awk -F" " '{print \$1}'   less	estrae la prima colonna, pagina con 'less'
w   awk -F" " '{print \$1}'   sort	riordina users (con duplicati)
w   awk -F" " '{print \$1}'   sort   uniq	elimina i duplicati

Nota: puoi fare redirect dell'output in un file utilizzando >

```
w | awk -F" " '{print $1}' | sort | uniq > users    (output inviato in file "users" tramite >)
```

## ESTENSIONE

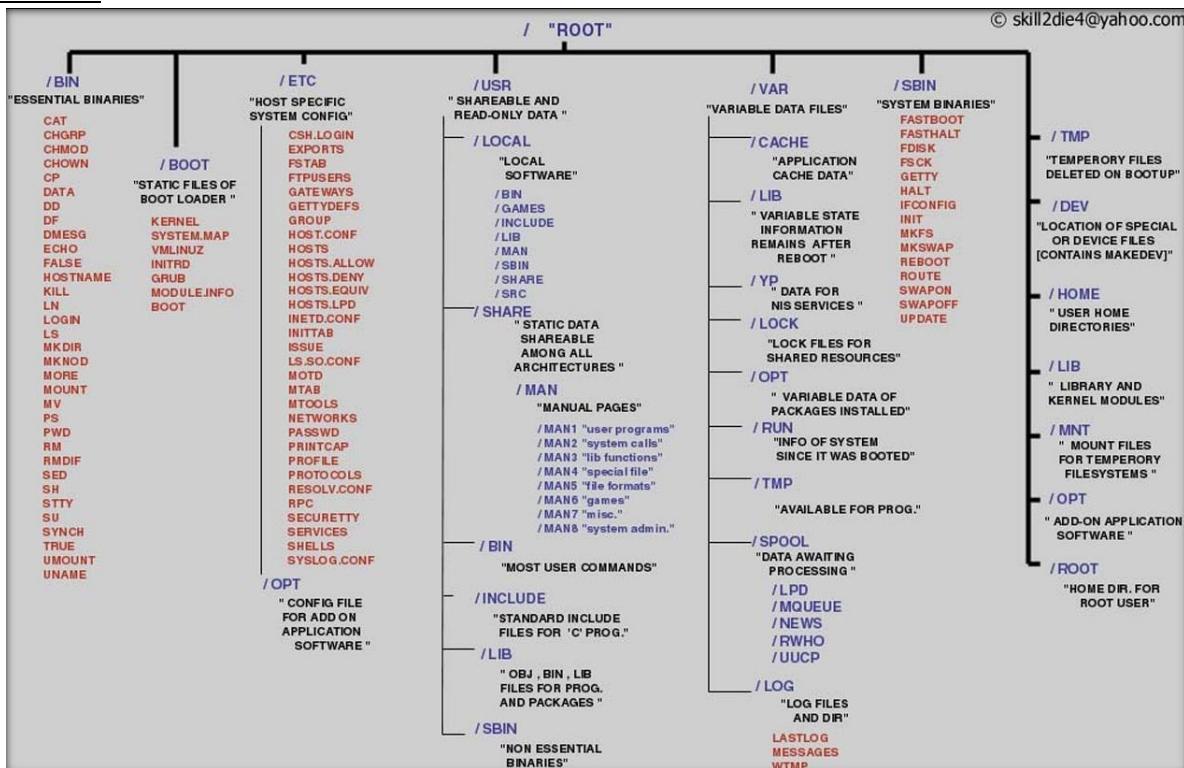
In Linux l'estensione del file non è importante, dobbiamo conoscere noi il programma da applicare a un file (quando lo usiamo da terminale, da GUI se facciamo doppio click ce lo apre in base all'estensione), la riga di comando se ne frega del formato.

Per convenzione mettiamo le estensioni ai file, per convenzioni quelli senza estensione dovrebbero essere programmi.

FIND : il comando find serve a trovare specifici file. Esempi:

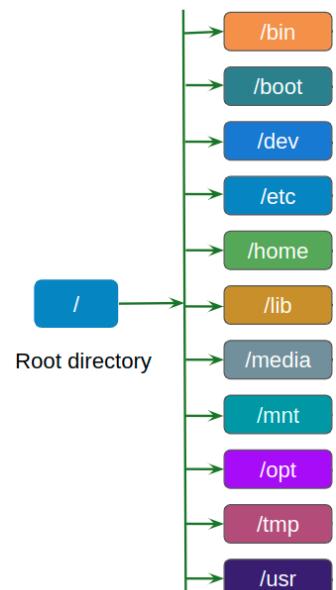
find . -name my-file.txt	cerca file specifico in .
find ~ -name bu -type d	cerca "bu" directory in ~

## FILE SYSTEM LINUX



Linux ha una struttura di cartelle ad albero (che per convenzione è sempre fatta nello stesso modo da parecchi anni), le cartelle di primo livello (come root) tendenzialmente sono accedibili da solo l'amministratore (tranne tmp). Nel dettaglio:

- La cartella Home è quella che contiene le cartelle degli utenti
- La cartella /BIN è per la gestione del sistema operativo (“essential binaries”)
- La cartella /BOOT contiene le informazioni necessarie per far partire il sistema operativo
- La cartella /ETC contiene i file di configurazione
- La cartella /DEV contiene le astrazioni (file) dei dispositivi come stampanti etc
- La cartella /LIB contiene le librerie
- La cartella /MNT è generalmente vuota, quando aggiungiamo una pennetta viene aggiunta in mnt come una cartella
- La cartella /ROOT è la home dell'utente amministratore
- La cartella /OPT è dove a volte vengono messe applicazioni aggiuntive disponibili per ogni utente
- La cartella /TMP è dei file temporanei, generalmente tutti hanno diritto di r e w in essa.
- La cartella /USR contiene file leggibili da tutti
- La cartella /VAR contiene variabili utili per programmi, per variabili si intendono dati che nel tempo sono variabili.



Nelle creazioni dei file/directory il gruppo viene assegnato automaticamente ed il gruppo che viene assegnato è un gruppo formato dal solo utente (è stata introdotta questa prassi in ubuntu per sicurezza)

Caratteri speciali interpretati dalla shell:

~	home directory
.	directory corrente
..	directory superiore (padre)
*	carattere jolly che corrisponde a qualsiasi nome di file
?	carattere jolly che corrisponde a qualsiasi carattere
TAB	prova a completare il nome del file

Navigare tra le cartelle:

- pwd stampa la cartella corrente
- cd (tilde) ci porta alla nostra home
- cd . cartella corrente
- cd .. cartella precedente

Il comando ls ha tantissime opzioni da usare con esso:

- ls mostra la lista dei file
- ls -l ci mostra i file in una forma tabellare con più info (consigliatissimo, "elle esse meno elle")
- ls -lt ce li ordina temporalmente
- ls -lh li mostra in modo più leggibile (la grandezza dei dati è scritta in modo leggibile)

Esempio: ls -lrth:

- ✓ ls: Elenca i file e le directory nel percorso corrente.
- ✓ -l: Mostra l'elenco in formato dettagliato (long format), che include permessi, numero di link, proprietario, gruppo, dimensione e data di modifica.
- ✓ -r: Ordina l'elenco in ordine inverso (dall'ultimo al primo).
- ✓ -t: Ordina per data di modifica, mostrando prima i file più recenti.
- ✓ -h: Mostra le dimensioni dei file in un formato leggibile (es. KB, MB).
- ✓ Risultato finale: Questo comando fornisce un elenco dettagliato dei file e delle directory, ordinato per data di modifica in ordine inverso e con dimensioni leggibili

Comandi Utili per i file:

- cp [file1] [file2] copia un file
- mkdir crea cartella
- rmdir elimina cartella (la cartella deve essere vuota)
- mv [file] [destination] spostare un file
- file [file] identifica il tipo di file
- rm [file] elimina un file, (-r per farlo ricorsivo)
- touch [filename] crea un file vuoto

Comandi sui diritti di accesso:

- chmod serve a cambiare i diritti di un file, per utilizzarlo usiamo le key:
  - u = owner      g = group      o = other      a = all
  - r = read        w = write      x = execute

⇒ Quindi se facciamo chmod ug+x foo rendo il file denominato foo eseguibile al proprietario e al gruppo  
(si utilizza + per aggiungere e – per togliere!)

Per cambiare i diritti di accesso con chmod si può usare anche la tabella seguente:

#	Permission	rwx
0	none	000
1	execute only	001
2	write only	010
3	write and execute	011
4	read only	100
5	read and execute	101
6	read and write	110
7	read, write, and execute	111

### GESTIONE DEI PROCESSI (ps command & kill)

Ogni processo quando ne esegue un altro crea un processo figlio e il sistema operativo gli associa un ID (PID process ID). Se voglio informazioni sul mio processo dovrei conoscere il PID del mio processo, per conoscere il PID del mio processo usiamo il comando ps.

- ps mostra tutti i processi attivi con relativi PID
- ps aux mi stampa la completa alberatura del processo, cioè tutti i processi che sono attivati rispetto al mio utente facendoci vedere la "concatenazione" (anche quelli che non sono esecuzione sulla mia bash cioè quelli in background)
- ps faux mi fa vedere TUTTI i processi in esecuzione anche quelli del sistema operativo

Il comando kill: se si chiama kill su processo figlio il processo padre rimane (anche se uccido padre il figlio rimane).

Ricorda: comando kill è un segnale, non uccide direttamente il processo!

CTRL+C: utilizzato per terminare un processo, manda al processo che voglio interrompere un segnale per terminarne l'esecuzione. Il suo funzionamento si basa sul fatto che in ogni applicazione esiste un handler (funzione) che tramite ctrl+c può essere attivata con conseguente terminazione del processo stesso.

(funzionamento: CTRL+C manda segnale di terminazione al processo, dentro il processo c'è un handler che gli è iniettato in fase di compilazione e linking che si attiva per terminare).

## FOREGROUND/BACKGROUND

Un processo attivo (anche detto in foreground) è un processo che posso vedere perchè eseguito dalla riga di comando che sto utilizzando. Tecnicamente potrei “staccare” il processo dalla riga di comando mandandolo in background per poter riprendere così il controllo della bash per poter fare altro (utile, ad esempio, per mandare in background un processo che impiega giorni in modo da liberarmi la riga di comando per eseguire qualcos’altro).

- ⇒ Background = In parallelo
- ⇒ Foreground = Interagire con command line

Per eseguire comandi in background si utilizza la “e commerciale”: command\_name & fregatura: nel momento in cui lo sto eseguendo lo standard output da qualche parte deve finire quindi finisce nella mia bash anche se voglio usarla per fare altro.

Come possiamo mettere un job in background/foreground?

- Si utilizza CTRL+Z per stoppare un programma (sospendere)
- Per capire quanti processi ci sono e in che stato sono si utilizza il comando “jobs”
- comando per mandare in foreground il job 2: fg 2
- comando per mandare in background il job 2: bg 2

## ESEMPIO COMANDO COMPOSTO UTILIZZANDO PIPE

printenv | grep ^PATH | sed 's/^PATH=///g' | tr ":" "\n" | sort | uniq | wc

- ✓ printenv: Stampa tutte le variabili d’ambiente.
- ✓ grep ^PATH: Filtra l’output per mostrare solo la riga che inizia con "PATH".
- ✓ sed 's/^PATH=///g': Rimuove "PATH=" dall’inizio della riga, lasciando solo i percorsi.
- ✓ tr ":" "\n": Sostituisce i due punti (":") con un newline, separando i vari percorsi su righe diverse.
- ✓ sort: Ordina alfabeticamente i percorsi.
- ✓ uniq: Rimuove i duplicati dall’elenco.
- ✓ wc: Conta le righe, le parole e i caratteri dell’output finale.
- ✓ Risultato finale: Questo comando fornisce un conteggio dei percorsi unici presenti nella variabile d’ambiente PATH.

## ESECUZIONE SCRIPT UTILIZZANDO BASH + SPECIFICARE UN INTERPRETE PER UNO SCRIPT

Comando = bash nomeScript.sh

Nota: se scrivo #!/path\Interprete posso specificare quale interprete utilizzare

- esempio: #!/usr/bin/python3
- esempio: #!/bin/bash

## STANDARD OUTPUT E STANDARD ERROR

Standard Output (stdout) e Standard Error (stderr) sono due flussi di output utilizzati nei sistemi Unix/Linux:

- Standard Output (stdout): È il flusso predefinito in cui vengono scritti i risultati normali dei comandi. Di solito, viene visualizzato sul terminale.
- Standard Error (stderr): È il flusso predefinito in cui vengono scritti i messaggi di errore. Anche questo viene visualizzato sul terminale, ma è separato dallo standard output.

Esempio comando: bash countdown.sh 30 1>tmp.txt 2>error.txt &

- ✓ bash countdown.sh 30: Esegue lo script countdown.sh con argomento 30.
- ✓ 1>tmp.txt: Reindirizza l’output standard (stdout) verso il file tmp.txt
- ✓ 2>error.txt: Reindirizza l’output di errore (stderr) verso il file error.txt
- ✓ &: Esegue il comando in background, permettendo di continuare a usare il terminale mentre lo script è in esecuzione.

Nota: i numeri 1 e 2 in 1> e 2> hanno significati specifici nel contesto dei flussi di input/output in Unix/Linux:

- 1: Rappresenta il Standard Output (stdout). È il flusso predefinito per l’output normale dei comandi.
- 2: Rappresenta il Standard Error (stderr). È il flusso predefinito per l’output di errore.

Questa numerazione è standard e permette di reindirizzare ciascun flusso in modo specifico. Ad esempio, quando usi 1> stai dicendo che vuoi reindirizzare l’output normale, mentre con 2> stai reindirizzando gli errori.

## IL COMANDO NOHUP

Il comando nohup serve per "staccare" l'esecuzione della bash gestendola come root in modo che se va via la corrente o problemi simili non perdo la sua esecuzione. Questo comando, quindi, permette al processo di continuare anche se il terminale viene chiuso.

esempio: nohup bash countdown.sh 30 &

## L' APPLICATIVO SCREEN

Comando: screen

Funzionamento: bash apre screen e poi screen apre un'altra bash

Per uscire dalla screen si utilizza CTRL+D

Comando per vedere le screen aperte: screen-ls

Per chiudere la screen basta usare comando exit come per ogni altra bash

## CREAZIONE USER ACCOUNT UTILIZZANDO COMANDO USERADD

Si può utilizzare il comando useradd per creare un nuovo utente, la sintassi è la seguente:

```
sudo useradd-s /path/to/shell -d /home/{dirname} -m -G {secondary-group} {username}
```

per assegnare una password si utilizza: sudo passwd {username}

Esempio creazione account chiamato vivek:

```
sudo useradd-s /bin/bash -d /home/vivek/ -m -G sudo vivek
```

```
sudo passwd vivek
```

- -s /bin/bash imposta /bin/bash come login shell del nuovo account
- -d /home/vivek/ imposta /home/vivek/ come home directory del nuovo Ubuntu account
- -m crea la cartella home del nuovo utente
- -G sudo - [Optional] si assicura che l'utente vivek può eseguire comandi sudo

## FILE EDITORS IN LINUX

1. Emacs
2. Vim
3. Gedit
4. nano

## EXTRA LEZIONE

tail -f tmp.txt ci mostra la fine del file tmp.txt ogni volta che viene modificato avendo specificato -f

## LEZ4 (PROCESSI E THREAD)

PROCESSO = programma in esecuzione.

È un'astrazione fondamentale del sistema operativo, che gli consente di semplificare l'allocazione delle risorse, la "contabilizzazione" delle risorse e la limitazione delle risorse.

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

### MODELLO DI UN PROCESSO:

Un processo è solo un'istanza di un programma in esecuzione, che contiene i valori correnti del contatore del programma, dei registri e delle variabili.

Ogni processo si trova in un'unica posizione, la CPU durante l'esecuzione di più processi passa da un processo all'altro (questo rapido passaggio da vita alla multiprogrammazione). Ogni processo quindi ha un proprio flusso di controllo (proprio contatore logico di programma), ogni volta che si passa da un processo all'altro, si salva il contatore di programma del primo processo e si ripristina il contatore di programma del secondo.

Nota: tutti i processi progrediscono, ma solo uno è attivo in un dato momento (è attivo un programma alla volta).

Nel caso di processi concorrenti, in linea di principio, i processi sono reciprocamente indipendenti tra loro e per interagire tra loro hanno bisogno di mezzi esplicativi. La CPU può essere assegnata a turno a diversi processi, e il SO normalmente non offre garanzie di tempistica o di ordine.

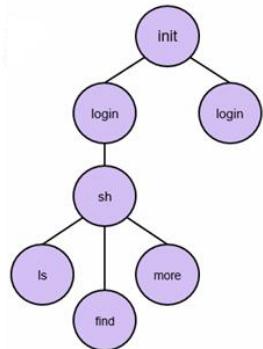
### GERARCHIE DI PROCESSI

Il SO crea solo un processo *init* (nei moderni sistemi operativi *init* avvia *kthreadd*, 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).

Ad esempio, la shell esegue i comandi:

```
$ find /tmp & > t.log &  
$ ls | more
```



Spiegazione comandi:

1) \$ find /tmp & > t.log &

- o `find /tmp` : Questo comando cerca all'interno della directory `/tmp` per trovare file e directory. Se non specificato diversamente, `find` elencherà tutti i file e le directory contenuti in `/tmp` .
- o `&` : In una shell Unix/Linux, l'operatore `&` alla fine di un comando esegue il comando in background. Questo significa che il terminale non attenderà il completamento di `find` prima di restituire il controllo all'utente, permettendo di eseguire altri comandi nel frattempo.
- o `> t.log` : Questo ridireziona l'output standard del comando in un file chiamato `t.log` . Tuttavia, la sintassi corretta per redirigere l'output di un comando eseguito in background dovrebbe essere `find /tmp > t.log &` .

In sintesi, il comando corretto sarebbe `find /tmp > t.log &` , che cerca file in `/tmp` , scrive l'output in `t.log` , e lo esegue in background.

2) \$ ls | more

- o `ls` : Questo comando elenca i file e le directory nella directory corrente.
- o `|` : L'operatore pipe (`|`) prende l'output del comando a sinistra (`ls`) e lo passa come input al comando a destra (`more` ).
- o `more` : Questo comando visualizza l'output in modo paginato. Se l'output di `ls` è lungo e non può essere visualizzato tutto in una sola schermata, `more` permette di scorrere l'output premendo il tasto spazio per andare alla pagina successiva.

In sintesi, il comando `ls | more` elenca i file e le directory e mostra l'output in modo paginato, consentendo di visualizzare lunghe liste senza sovraccaricare il terminale.

## CREAZIONE DI UN PROCESSO

I quattro eventi principali che causano la creazione di processi sono:

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 (ad esempio tramite bash)
4. Avvio di un lavoro in modalità batch (all'avvio di un SO in genere vengono creati numerosi processi)

In UNIX, c'è solo una chiamata di sistema per creare un nuovo processo: *fork*. Questa chiamata crea un clone esatto del processo chiamante. Dopo il *fork*, i due processi, il genitore e il figlio, hanno la stessa immagine di memoria, le stesse stringhe d'ambiente e gli stessi file aperti.

## TERMINE DI UN PROCESSO

Condizioni tipiche che terminano un processo:

1. Uscita normale (volontaria).
2. Uscita a causa di un errore (volontaria).
3. Errore "fatale" (involontario).
4. Ucciso da un altro processo (involontario).

## GESTIONE DI UN PROCESSO

*fork*: crea un nuovo processo

- Il figlio è un clone "privato" del genitore
- 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

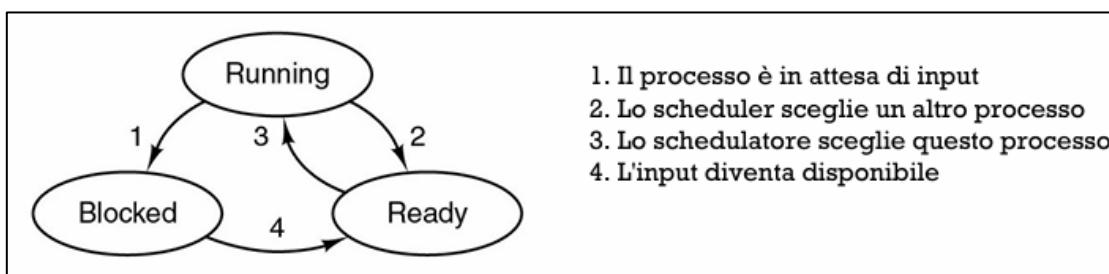
## GLI STATI DI UN PROCESSO

Un processo può trovarsi in tre stati:

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).
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 allocare la CPU, il sistema operativo deve tenere traccia degli stati dei processi (Running/Blocked/Ready).

Lo scheduler (de)assegna la CPU (vedi transizioni 2&3).



## INFORMAZIONI ASSOCIATE AD UN PROCESSO

Le informazioni seguenti associate ad un processo sono memorizzate nella tabella dei processi del SO:

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

"Signals" e "Interrupts" sono meccanismi utilizzati nei sistemi operativi e nelle applicazioni per gestire eventi asincroni.

## Interrupts:

- Idea: per deallocare la CPU a favore dello scheduler, ci si affida al supporto per la gestione degli interrupt fornito dall'hardware
- 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

## VETTORE DI INTERRUPT

Il vettore di interrupt è una struttura che contiene gli indirizzi di tutte le routine di servizio per gli interrupt.

Ogni tipo di interrupt (ad esempio, tastiera o mouse) ha un proprio indirizzo in questa tabella, permettendo al sistema di rispondere rapidamente in base alla priorità di ciascun interrupt.

## IMPLEMENTAZIONE DEI PROCESSI

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 e agisce come mediatore.

Un processo non può cedere la CPU a un altro processo (context switch) senza passare attraverso lo scheduler.

## GESTIONE DEI SEGNALI

Tipi di Segnali:

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

Azioni possibili:

- Term, Ign, Core, Stop, Cont
- Azione predefinita per ogni segnale, tipicamente sovrascrivibile
- I segnali possono essere tipicamente bloccati e le azioni ritardate.

Gestione (catching) dei segnali:

- Il processo registra il gestore del segnale
- Il sistema operativo invia il segnale e consente al processo di eseguire l'handler
- Il contesto di esecuzione corrente deve essere salvato/ripristinato

=>Gestione dei segnali:

1. il kernel invia un segnale
2. interrompe il codice in esecuzione
3. salva il contesto
4. esegue il codice di gestione del segnale
5. ripristina il contesto originale

## THREAD

Assunzione implicita finora: 1 processo => 1 thread in esecuzione

Multithreaded execution: 1 processo => N thread in esecuzione

Perché consentire più thread per processo?

1. Invece di pensare a interrupt, timer e cambi di contesto, possiamo pensare a processi paralleli. Solo che ora con i thread aggiungiamo un nuovo elemento: la possibilità per le entità parallele di condividere uno spazio di indirizzamento e tutti i suoi dati tra di loro
  2. Poiché sono più leggeri dei processi, sono più facili (cioè più veloci) da creare e distruggere rispetto ai processi. In molti sistemi, la creazione di un thread è 10-100 volte più veloce rispetto alla creazione di un processo
  3. Per le prestazioni. I thread non producono alcun miglioramento delle prestazioni quando tutti sono legati alla CPU, ma quando c'è un calcolo sostanziale e anche un I/O sostanziale, la presenza di thread consente a queste attività di sovrapporsi, velocizzando così l'applicazione

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 stack, i propri registri hardware e il proprio stato.

Tabella/interrupt dei thread: una tabella/ interrupt di processo più leggera.

Ciascun thread può chiamare qualsiasi chiamata di sistema supportata dal sistema operativo per conto del processo a cui appartiene.

Il modello di thread classico:

Per processo	Per thread
<ul style="list-style-type: none"><li>• Address space</li><li>• Global variables</li><li>• Open files</li><li>• Child processes</li><li>• Pending alarms</li><li>• Signals and signal handlers</li><li>• Accounting information</li></ul>	<ul style="list-style-type: none"><li>• Program counter</li><li>• Registers</li><li>• Stack</li><li>• State</li></ul>

I thread in posix:

<i>pthread_create</i>	Crea un nuovo thread
<i>pthread_exit</i>	Termina il thread chiamante
<i>pthread_join</i>	Attende l' "uscita" di uno specifico thread
<i>pthread_yield</i>	Rilascia la CPU per consentire l'esecuzione di un altro thread
<i>pthread_attr_init</i>	Crea e inizializza la struttura di attributi di un thread
<i>pthread_attr_destroy</i>	Rimuove la struttura di attributi di un thread

## LUOGHI DI IMPLEMENTAZIONE DEI THREAD

---

Esistono tre luoghi di implementazione dei thread:

- Nello spazio utente
  - Nel kernel
  - Un'implementazione ibrida

## IMPLEMENTAZIONE DEI THREAD NELLO SPAZIO UTENTE

Pro dell'implementazione dei thread nello spazio utente:

1. I thread nello spazio utente sono gestiti dal kernel come processi ordinari a singolo thread; quindi, possono essere eseguiti su sistemi operativi che non supportano direttamente i thread.  
(sono gestiti tramite una libreria)
  2. 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.
  3. L'interruzione e il cambio tra thread a livello utente non richiedono un cambiamento di contesto completo. Non si fa utilizzo di trap e sono molto più veloci rispetto alle operazioni nel kernel.
  4. Offrono l'abilità di personalizzare l'algoritmo di scheduling per ogni processo e una maggiore scalabilità.

Contro dell'implementazione dei thread nello spazio utente:

1. Ci sono alcuni 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.
2. I thread nello spazio utente non hanno interrupt del clock, rendendo impossibile uno scheduling di tipo round-robin.
3. 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 (i thread a livello utente possono fermarsi completamente se un singolo thread effettua una chiamata di sistema bloccante, influenzando tutti gli altri thread nel processo).

#### IMPLEMENTAZIONE DEI THREAD NELLO SPAZIO KERNEL

- i. 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).
- ii. Le chiamate che potrebbero bloccare un thread vengono implementate come chiamate di sistema (hanno costi più elevati rispetto alle chiamate di procedura dei sistemi run-time) (se un thread si blocca, il kernel può eseguire un altro thread, sia dello stesso processo sia di un altro).
- iii. Alcuni sistemi "riciclano" i thread per ridurre i costi, invece che terminarli.
- iv. 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.
- v. La programmazione con thread richiede cautela per evitare errori.

#### IMPLEMENTAZIONI IBRIDE DEI THREAD

- i. Alcuni sistemi effettuano il multiplexing dei thread utente sui thread del kernel, queste implementazioni ibride combinano i vantaggi dei due approcci.
- ii. I programmatore decidono quanti thread del kernel utilizzare e quanti thread utente multiplexare (questo porta a maggiore flessibilità).
- iii. Il kernel è consapevole solo dei thread del kernel (ma ogni thread del kernel può gestire più thread a livello utente).

#### PROBLEMI APERTI RIGUARDANTI I THREADS

Molte procedure di libreria possono causare conflitti se un thread sovrascrive dati cruciali per un altro, esempio:

- l'invio di un messaggio sulla rete potrebbe essere programmato assemblando il messaggio in un buffer fisso nella libreria e poi eseguendo una trap nel kernel per spedirlo
- che cosa accade se un thread ha preparato il suo messaggio nel buffer e poi un interrupt del clock forza uno scambio con un secondo thread, che sovrascrive immediatamente il buffer con un suo messaggio?

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.

## LEZ5 (ELEMENTI DI PROGRAMMAZIONE NEI SO)

### STDIN, STDOUT, STDERR VISTI COME FILE (TUTTO È UN FILE)

NOME	SHORT_NAME	NUMERO_FILE	DESCRIZIONE
Standard In	stdin	0	Input da tastiera
Standard Out	stdout	1	Output alla console
Standard Error	stderr	2	Output di errore

Per impostazione predefinita, ogni processo inizia con questi 3 "file"... "Aperti".

### 3 MODI PER STAMPARE SU CONSOLE (OUTPUT STANDARD)

```
1) #include <stdio.h>
int main(int argc, char **argv)
{
    printf("Hello World!\n");
    return 0;
}
```

```
2) #include <unistd.h>
#define STDOUT 1
int main(int argc, char **argv)
{
    char msg[] = "Hello World!\n";
    write(STDOUT, msg, sizeof(msg));
    return 0;
}
```

```
3) #include <unistd.h>
#include <stdio.h>
#include <sys/syscall.h>
#define STDOUT 1
int main(int argc, char **argv)
{
    char msg[] = "Hello World!\n";
    int nr = SYS_write;
    syscall(nr, STDOUT, msg, sizeof(msg));
    return 0;
}
```

### LIBRERIE STANDARD E SYSCALL

Libc fornisce wrapper utili intorno alle syscall, come ad esempio write, read, exit.

È necessario chiamare la syscall oppure l'istruzione int 0x80 (fatto in assembly).

Tabella con alcune syscall:

%rax (return value)	Name	Manual	Entry point
0	read	<a href="#">read(2)</a>	<a href="#">sys_read</a>
1	write	<a href="#">write(2)</a>	<a href="#">sys_write</a>
2	open	<a href="#">open(2)</a>	<a href="#">sys_open</a>
3	close	<a href="#">close(2)</a>	<a href="#">sys_close</a>
22	pipe	<a href="#">pipe(2)</a>	<a href="#">sys_pipe</a>
32	dup	<a href="#">dup(2)</a>	<a href="#">sys_dup</a>
57	fork	<a href="#">fork(2)</a>	<a href="#">sys_fork</a>
59	execve	<a href="#">execve(2)</a>	<a href="#">sys_execve</a>
60	exit	<a href="#">exit(2)</a>	<a href="#">sys_exit</a>
62	kill	<a href="#">kill(2)</a>	<a href="#">sys_kill</a>

### COSA SUCCIDE QUANDO VIENE INVOCATA LA FUNZIONE read()?

Di seguito viene riportata la sequenza approssimativa dei passaggi:

1. Funzione read() in glibc
  - la funzione read() è implementata in glibc e funge da wrapper per la chiamata di sistema read.
2. Macro SYSCALL\_CANCEL
  - la macro SYSCALL\_CANCEL viene utilizzata in read.c per effettuare la chiamata di sistema effettiva in modo sicuro.
3. Macro INTERNAL\_SYSCALL
  - SYSCALL\_CANCEL si basa su INTERNAL\_SYSCALL per configurare i registri e invocare l'istruzione syscall, che fa passare il controllo dal contesto utente a quello kernel.
4. Punto di ingresso nel kernel (entry\_SYSCALL\_64)
  - l'istruzione syscall provoca il passaggio alla modalità kernel e indirizza l'esecuzione a entry\_SYSCALL\_64, definito in assembly.
5. Funzione do\_syscall\_64
  - entry\_SYSCALL\_64 chiama do\_syscall\_64, una funzione C che determina quale funzione kernel chiamare basandosi sul numero della chiamata di sistema.
6. Tabella delle chiamate di sistema (sys\_call\_table)
  - do\_syscall\_64 consulta sys\_call\_table per mappare il numero di system call alla funzione kernel appropriata. Per read, il numero è 0 e punta a \_\_x64\_sys\_read.
7. Funzione \_\_x64\_sys\_read
  - la funzione \_\_x64\_sys\_read è il wrapper per read nel kernel, definita con SYSCALL\_DEFINE3, che richiama ksys\_read per la logica di lettura.

## CREAZIONE DI PROCESSI

La fork esegue un clone del processo copiando tutto tranne alcune informazioni quali ad esempio l'area di memoria a cui punta. Quindi per duplicare processo corrente usiamo pid\_t fork() che restituisce un numero. Vendono quindi eseguite istruzioni, poi chiamata fork, poi si va sulla fork che farà una copia dell'area di memoria e a questo punto crea un nuovo processo.

Differenza: nel processo che è padre viene dato il numero del figlio, al figlio viene dato 0 (altrimenti non si sa chi è l'originale e chi è la copia, la realtà è che sono due cloni in quanto uguali)

- pid\_t fork()
  - o Duplica il processo corrente,
  - o Restituisce il pid del figlio nel chiamante (genitore)
  - o Restituisce 0 nel nuovo processo (figlio)
- pid\_t wait(int \*wstatus)
  - o Attende che i processi figlio cambino stato,
  - o Scrive lo stato in wstatus
- int execv(const char \*path, char \*const argv[])
  - o Carica un nuovo path nel processo corrente, rimuovendo tutte le altre mappature di memoria
  - o Constargv contiene gli argomenti del programma
  - o L'ultimo argomento è NULL
  - o Esempio: constargv = {"./bin/ls", "-a", NULL}

execv permette di dire "io sono un processo e sto facendo delle cose, nel momento in cui viene chiamata execv gli passo il path di un altro binario: viene cancellato tutto quello che è del processo e viene sostituito tutto con quello che è nel binario". Quindi viene sostituito interamente e ho la possibilità da un processo di chiamarne un altro (è quello che fa bash quando eseguiamo un comando). Ho bisogno di sapere qual è il path e quali parametri gli devo fornire.

Esempio di Fork,Wait,Execv:

```
void main(void)
{
    int pid, child_status;
    char *args[] = {"./bin/ls", "-l", NULL};
    if (fork() == 0) { // fork creates child process
        execv(args[0], args); // in child: load+execute program
    } else {
        wait(&child_status); // Wait for child
    }
}
```

Esempio di shell minimale:

```
while (1) {
    char cmd[256], *args[256];
    int status;
    pid_t pid;
    read_command(cmd, args); /* reads command and arguments from command line */

    pid = fork();

    if (pid == 0) {
        execv(cmd, args);
        exit(1);
    } else {
        wait(&status);
    }
}
```

## GESTIONE DEI SEGNALI

Si possono terminare programmi tramite Ctrl+C, questo accade grazie ai segnali (Ctrl+C = SIGINT). A volte i processi devono essere interrotti durante la loro esecuzione, per fare ciò viene inviato un segnale al processo che deve essere interrotto. Il processo interrotto può catturare il segnale installando un gestore di segnali (signal handler).

Per gestire i segnali si usano i comandi seguenti: signal, alarm, kill. Più nel dettaglio:

- Sighandler\_t signal(int signum, sighandler\_t handler)
  - o registra un gestore di segnali per il segnale signum
- unsigned int alarm(unsigned int seconds)
  - o consegna SIGALARM in un numero di secondi specificato
- int kill(pid\_t pid, int sig)
  - o consegna il segnale sig al processo pid (non uccide!)

## Esempio di alarm:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

void alarm_handler(int signal)
{
    printf("In signal handler: caught signal %d!\n", signal);
    exit(0);
}

int main(int argc, char **argv)
{
    signal(SIGALRM, alarm_handler);
    alarm(1); // alarm will send signal after 1 sec

    while (1) {
        printf("I am running!\n");
    }
    return 0;
}
```

## COMUNICAZIONE TRA PROCESSI ATTRAVERSO PIPE

Per far comunicare due processi abbiamo due modi: i segnali e le pipe.

### OPEN, CLOSE, PIPE, DUP:

- int open(const char \*pathname, int flags)
  - o Apre il file specificato dal nome del percorso (pathname)
- int close(int fd)
  - o Chiude il descrittore di file specificato fd
- int pipe(int pipefd[2])
  - o Crea una pipe con due fd per le sue estremità
- int dup(int oldfd)
  - o Crea una copia del descrittore di file oldfd utilizzando il descrittore di file inutilizzato con il numero più basso per la copia.

### PIU' NEL DETTAGLIO DI DUP E DUP2:

dup e dup2 sono chiamate di sistema su Unix/Linux per duplicare descrittori di file.

#### Funzione:

- dup(int oldfd): Crea un duplicato del descrittore oldfd.
- dup2(int oldfd, int newfd): Duplica oldfd su un descrittore specifico newfd.

Perché? è comune voler reindirizzare l'output di un programma verso un file o un altro programma. dup e dup2 permettono di «agganciare» il descrittore di output standard (STDOUT\_FILENO, ovvero 1) o di input standard (STDIN\_FILENO, ovvero 0) a un file.

Utilità: Permettono la gestione avanzata di input/output, come il reindirizzamento di file, la creazione di pipeline tra processi, e la gestione di input/output in processi figli.

### Esempio1 - reindirizzamento di output:

**Scenario:** Salvare l'output di un programma in un file.

#### Esempio:

```
int file_fd = open("output.txt", O_WRONLY | O_CREAT, 0644);
dup2(file_fd, STDOUT_FILENO); // Reindirizza stdout su file_fd
printf("Questo va in output.txt\n");
close(file_fd);
```

**Risultato:** L'output viene scritto in output.txt invece che in console.

**Utilizzo tipico:** Registrazione di log, salvataggio di output in file.

### Esempio2 - creazione di pipeline tra processi

**Scenario:** Collegare l'output di un programma all'input di un altro

#### Esempio: ps aux | grep httpd

```
int fd[2];
pipe(fd);
if (fork() == 0) {
    close(fd[0]);
    dup2(fd[1], STDOUT_FILENO); // Output di `ps aux` nella pipe
    execvp("ps", "ps", "aux", NULL);
} else {
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO); // Input di `grep` dalla pipe
    execvp("grep", "grep", "httpd", NULL);
}
```

**Risultato:** ps aux manda l'output a grep, simulando una pipeline.

**Utilizzo tipico:** Shell scripting, automazione di comandi.

### Esempio3 – gestione dei processi figli

**Scenario:** Reindirizzare l'output di un processo figlio a un file di log.

```
if (fork() == 0) {
    int log_fd = open("logfile.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
    dup2(log_fd, STDOUT_FILENO); // Output va in logfile.txt
    close(log_fd);
    execlp("ls", "ls", "-l", NULL); // Comando eseguito dal figlio
}
```

**Risultato:** L'output del processo figlio ls -l va in logfile.txt

**Nota:** execlp sostituisce il processo corrente con il processo specificata dal file

**Utilizzo tipico:** Logging centralizzato, configurazione di ambienti di processo.

### ESEMPIO DI PIPE (LA SECONDA PARTE IN C LA CHIEDE SEMPRE ALL'ESAME):

Cosa succede se si esegue il seguente comando?

```
$ cat names.txt | sort
```

E i seguenti comandi?

```
$ mkfifo named.pipe
$ echo "Hello World!" > named.pipe
$ cat named.pipe
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define STDIN 0
#define STDOUT 1

#define PIPE_RD 0
#define PIPE_WR 1

int main(int argc, char** argv)
{
    pid_t cat_pid, sort_pid;
    int fd[2];
    pipe(fd);

    cat_pid = fork();
    if (cat_pid == 0) {
        close(fd[PIPE_RD]);●
        close(STDOUT);
        dup(fd[PIPE_WR]);
        execl("/bin/cat", "cat", "names.txt", NULL);
    }

    sort_pid = fork();
    if (sort_pid == 0) {
        close(fd[PIPE_WR]);●
        close(STDIN);
        dup(fd[PIPE_RD]);
        execl("/usr/bin/sort", "sort", NULL);
    }

    /* wait for children to finish */
    waitpid(cat_pid, NULL, 0);
    waitpid(sort_pid, NULL, 0);

    return 0;
}
```

Si possono saltare le "chiusure" contrassegnate dal pallino arancione? Risposta:

1. Evitare Blocchi: Chiudi la fine di lettura di una pipe per impedire al processo di scrittura di rimanere bloccato.
2. Ricezione EOF: sort aspetta un EOF per terminare la lettura. Chiudi la fine di scrittura per inviare un EOF a sort.
3. Evitare Letture Accidentali: Nel processo cat, chiudi la fine di lettura per evitare letture inaspettate dalla pipe.
4. Reindirizzamento di STDOUT in cat: Dopo la duplicazione, chiudi il descriptor originale per garantire che cat scriva solo nella pipe.
5. Reindirizzamento di STDIN in sort: Dopo la duplicazione, chiudi il descriptor originale per assicurarti che sort legga solo dalla pipe.
6. Descriptor nel Processo Padre: Dopo la fork, il processo padre dovrebbe chiudere entrambe le estremità della pipe

## CODICI LEZIONE 6

### 5.2\_my\_first\_fork\_1.c

- un semplice esempio di invocazione del metodo fork

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define STDIN 0
#define STDOUT 1
#define PIPE_RD 0
#define PIPE_WR 1

int main(){
    int pid, child_status;

    if ((pid = fork()) == 0) {
        printf("I am the child and I see the PID %d\n", pid);
    } else {
        wait(&child_status); // Wait for child
        printf("I am the parent, I see the child's PID (%d) and the status (%d)\n", pid, child_status);
    }
}
```

### 5.2\_my\_first\_fork\_2.c

- creazione dei processi e avvio di applicazioni da codice C

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// Definizione di costanti per leggibilità per i descrittori di file
#define STDIN 0
#define STDOUT 1
#define PIPE_RD 0 // Lato di lettura della pipe
#define PIPE_WR 1 // Lato di scrittura della pipe

int main(int argc, char** argv) {

    pid_t cat_pid, sort_pid;
    int fd[2]; // Descrittori di file per la pipe: fd[0] per leggere, fd[1] per scrivere

    pipe(fd); // Crea una pipe. fd[PIPE_RD] sarà l'estremità di lettura, fd[PIPE_WR] sarà l'estremità di scrittura

    cat_pid = fork(); // Fork del processo corrente

    if (cat_pid == 0) {
        // Questo è il processo figlio per 'cat'
        close(fd[PIPE_RD]); // Chiude l'estremità di lettura della pipe, non necessaria qui
        close(STDOUT); // Chiude l'output standard

        /*
         La funzione dup viene utilizzata nei programmi Unix/Linux per duplicare un
         file descriptor esistente, ottenendo un secondo file descriptor che punta alla
         stessa risorsa interna (ad esempio, uno stesso file o lato di una pipe).
         Questo nuovo file descriptor è il più basso disponibile: dup sceglie automaticamente
         il numero di file descriptor più basso che è disponibile nel processo corrente.
         Per esempio, se i file descriptor 0, 1, e 2 sono chiusi,
         dup utilizzerà il 0 per la sua duplicazione.
        */

        dup(fd[PIPE_WR]); // Duplica l'estremità di scrittura della pipe al descrittore di
        // file dell'output standard. Ora, 'cat' scriverà sulla pipe invece che sul terminale
        execl("/bin/cat", "cat", "names.txt", NULL); // Esegui il comando 'cat'
    }

    sort_pid = fork(); // Fork del processo corrente ancora per 'sort'

    if (sort_pid == 0) {
        // Questo è il processo figlio per 'sort'
        close(fd[PIPE_WR]); // Chiude l'estremità di scrittura della pipe, non necessaria qui
        close(STDIN); // Chiude l'input standard
        dup(fd[PIPE_RD]); // Duplica l'estremità di lettura della pipe al descrittore di file dell'input standard
        // Ora, 'sort' leggerà dalla pipe invece che dalla tastiera
        execl("/usr/bin/sort", "sort", NULL); // Esegui il comando 'sort'
    }

    // Questo è il processo genitore
    close(fd[PIPE_RD]); // Chiude l'estremità di lettura della pipe, il genitore non la usa
    close(fd[PIPE_WR]); // Chiude l'estremità di scrittura della pipe, il genitore non la usa

    // Aspetta che i processi figli terminino
    waitpid(cat_pid, NULL, 0); // Aspetta il processo 'cat'
    waitpid(sort_pid, NULL, 0); // Aspetta il processo 'sort'

    return 0;
}
```

### 5.3\_my\_signal\_1.c

#### - Gestione dei segnali

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h> // Per strsignal()

void signalHandler(int signum) {
    printf("Interrupt signal %d received which is %s\n", signum, strsignal(signum));
    // cleanup and terminate program
    exit(signum);
}

int main() {
    // register signal SIGINT and signal handler
    signal(SIGINT, signalHandler); // CTRL+C

    while(1) {
        printf("Going to sleep...\n");
        sleep(1);
    }

    return 0; // questa riga non verrà mai raggiunta a causa del ciclo while(1)
}
```

### compile.sh

```
#!/bin/bash

## Il #!/bin/bash all'inizio di un file è chiamato "shebang".
# Ha lo scopo di indicare al sistema con quale interprete eseguire lo script che segue.
#
# Ecco cosa fa:
#
# - Indica l'Interprete: Dice al sistema che lo script dovrebbe essere eseguito usando
#   l'interprete specificato dopo #!. Nel caso di #!/bin/bash, indica che lo script
#   dovrebbe essere eseguito con l'interprete Bash.
# - Esecuzione Diretta: Permette di eseguire lo script come un programma direttamente
#   dalla linea di comando (ad es. ./myscript.sh), senza dover chiamare esplicitamente
#   l'interprete (bash myscript.sh).

CC=clang # o gcc?

# -O0 Means "no optimization": this level compiles the fastest and generates the most debuggable code.
# -O1 Somewhere between -O0 and -O2.
# -O2 Moderate level of optimization which enables most optimizations.
# -O3 Like -O2, except that it enables optimizations that take longer to perform or that may generate larger code
#   (in an attempt to make the program run faster).
OPTIONS="-O2"

# Differenti modi per scrivere su standard out
${CC} ${OPTIONS} 5.1_hello_world_1.c -o 5.1_hello_world_1
${CC} ${OPTIONS} 5.1_hello_world_2.c -o 5.1_hello_world_2
${CC} ${OPTIONS} 5.1_hello_world_3.c -o 5.1_hello_world_3

# Diverse modi di generare processi
${CC} ${OPTIONS} 5.2_my_first_fork_1.c -o 5.2_my_first_fork
${CC} ${OPTIONS} 5.2_my_first_fork_2.c -o 5.2_my_first_fork_2

# Gestione dei segnali
${CC} ${OPTIONS} 5.3_my_signal_1.c -o 5.3_my_signal_1
${CC} ${OPTIONS} 5.3_my_signal_2.c -o 5.3_my_signal_2

# Interazione tra processi e PIPE (vedi esempio su slide)
${CC} ${OPTIONS} 5.4_fork_pipe.c -o 5.4_fork_pipe

# La mia prima bash
${CC} ${OPTIONS} 5.5_my_first_bash.c -o 5.5_my_first_bash
```

### 5.3\_my\_signal\_2.c

- Gestione degli allarmi

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h> // Per strsignal()

void alarm_handler(int signal) {
    printf("In signal handler: caught signal %d which is %s!\n", signal, strsignal(signal));
    exit(0);
}

int main(int argc, char **argv) {
    signal(SIGALRM, alarm_handler);
    alarm(1); // alarm will send signal after 1 sec

    while (1) {
        printf("I am running!\n");
    }

    return 0;
}
```

### 5.4\_fork\_pipe.c

- Comunicazione tra processi attraverso PIPE

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define STDIN 0
#define STDOUT 1
#define PIPE_RD 0
#define PIPE_WR 1

int main(int argc, char** argv) {
    pid_t cat_pid, sort_pid;
    int fd[2];

    pipe(fd);

    cat_pid = fork();

    if (cat_pid == 0) {
        close(fd[PIPE_RD]);
        close (STDOUT);
        dup(fd[PIPE_WR]);
        execl("/bin/cat", "cat", "names.txt", NULL);
    }

    sort_pid = fork();

    if (sort_pid == 0) {
        close(fd[PIPE_WR]);
        close (STDIN);
        dup(fd[PIPE_RD]);
        execl("/usr/bin/sort", "sort", NULL);
    }

    close(fd[PIPE_RD]);
    close(fd[PIPE_WR]);

    /* wait for children to finish */
    waitpid(cat_pid, NULL, 0);
    waitpid(sort_pid, NULL, 0);

    return 0;
}
```

## 5.5\_my\_first\_bash.c

- Sviluppo di una BASH minimale in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

void free_args(char** args) {
    int i = 0;
    while (args[i]) {
        free(args[i]);
        i++;
    }
}

void read_command(char* cmd, char** args) {
    char* input = NULL;
    size_t len = 0;
    printf("myshell> ");
    ssize_t read_len = getline(&input, &len, stdin);

    if (read_len == -1) { // L'utente ha premuto CTRL+D
        //Quando l'utente preme CTRL+D, la funzione getline restituirà un valore negativo.
        //Puoi utilizzare questo per rilevare un EOF e agire di conseguenza.
        printf("\n"); // Per andare a capo dopo CTRL+D
        exit(0); // Chiudi il shell
    }

    char* token = strtok(input, " \n"); // Dividiamo input su spazi e eventuali "\n" (alla fine della stringa)

    int i = 0;
    while (token) {
        args[i] = strdup(token); // Usa strdup per duplicare e assegnare memoria al token
        if (i == 0) {
            strcpy(cmd, token);
        }
        token = strtok(NULL, " \n");
        i++;
    }
    args[i] = NULL;
    free(input);
}

int main() {
    while (1) {
        char cmd[256], *args[256];
        int status;
        pid_t pid;

        read_command(cmd, args);

        if (strcmp(cmd, "exit") == 0) {
            exit(0);
        }

        pid = fork();

        if (pid < 0) {
            perror("fork failed");
            exit(1);
        }

        if (pid == 0) {
            if (execvp(cmd, args) == -1) {
                perror("execvp failed");
                exit(1);
            }
        } else {
            if (wait(&status) == -1) {
                perror("wait failed");
                exit(1);
            }
        }

        //gestisce lo "spreco di memoria" introdotto da strdup
        free_args(args);
    }
    return 0;
}
```

## LEZ6

Può accadere che due o più processi devono eseguire operazioni in parallelo, e quindi hanno bisogno di essere orchestrati...

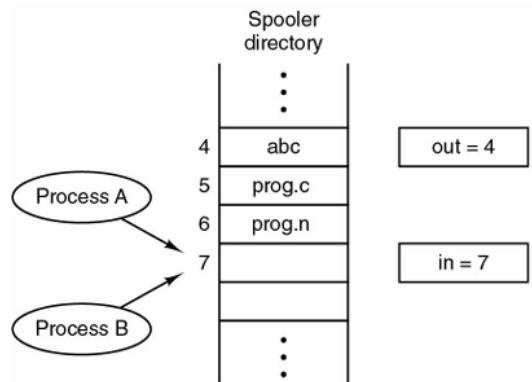
Si riscontrano problemi nel caso in cui due processi hanno bisogno di leggere/scrivere gli stessi dati. Si parla di race condition quando abbiamo due processi o due thread che agiscono sulla stessa risorsa condivisa.

### RACE CONDITION

Si parla di Race Condition quando:

- ✓ Il processo A legge  $in=7$  e decide di aggiungere il suo file in quella posizione.
- ✓ A 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.
- ✓ B imposta  $in=8$  e alla fine viene sospeso.
- ✓ A scrive il suo file nella posizione 7

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.



Nota: una "race condition" è una situazione in cui l'output dipende dall'ordine di esecuzione dei processi.

### CRITICAL REGION

Requisiti per evitare "race conditions":

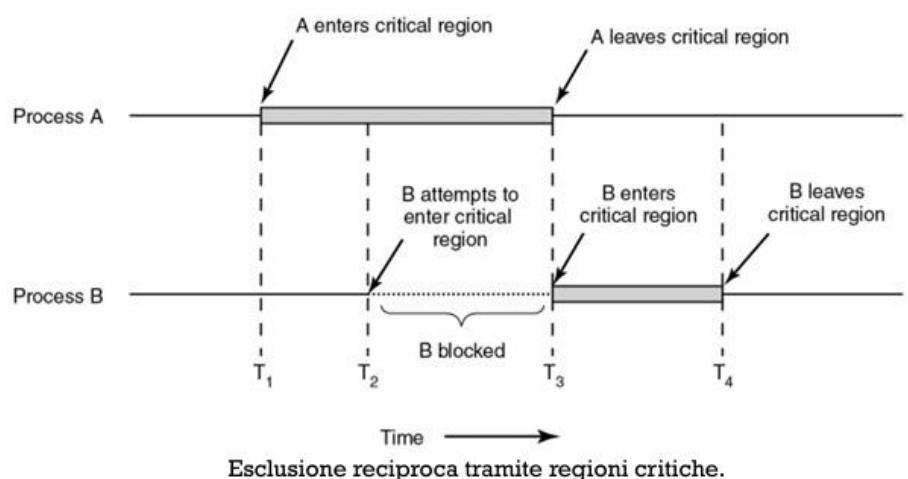
1. Due processi non possono trovarsi contemporaneamente all'interno delle rispettive regioni critiche.
2. Non si possono fare ipotesi sulla velocità o sul numero di CPU.
3. Nessun processo in esecuzione al di fuori della propria regione critica può bloccare altri processi.
4. Nessun processo deve aspettare all'infinito per entrare nella propria regione critica.

Sono soluzioni errate (non soluzioni) le seguenti idee:

- Disabilitare gli interrupt: impedisce semplicemente che la CPU possa essere riallocata. Funziona solo per sistemi a CPU singola.
- Bloccare le variabili: proteggere le regioni critiche con variabili 0/1. Le "corse" si verificano ora sulle variabili di blocco.

### MUTUA ESCLUSIONE

La regola di mutua esclusione impone che le operazioni con le quali i processi accedono alle variabili comuni non si sovrappongano nel tempo. La mutua esclusione prova a risolvere le criticità che si presentano quando ci sono risorse condivise.



Esclusione reciproca tramite regioni critiche.

### BUSY WAITING

Si parla di busy waiting quando si spendono risorse nel processore per cose inutili come richiedere continuativamente accesso alla risorsa quando è occupata.

## PETERSON'S ALGORITHM (Esclusione reciproca con busy waiting)

Questo algoritmo non risolve il problema del busy waiting ma soltanto quello dell'attesa rigorosa. Si tratta di un algoritmo di mutua esclusione per due processi, non generalizzabile a più di due processi.

Funzionamento:

- Alice e Bob vogliono usare un'unica postazione computer in un ufficio. Ma ci sono delle regole:
  - o Solo una persona può usare il computer alla volta.
  - o Se entrambi vogliono usarlo contemporaneamente, devono decidere chi va per primo.
- Idea dell'algoritmo
  - o Alice o Bob devono segnalare il loro interesse a usare il computer.
  - o Se l'altro non è interessato, la persona interessata può usarlo subito.
  - o 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.
  - o La persona che non ha la precedenza aspetta finché l'altra ha finito.
  - o Una volta finito, la persona che ha usato il computer segnala che ha finito, e l'altra può iniziare

```
#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 */
}
```

Questa soluzione usa busy waiting, quindi consente a un processo di tenere occupata la CPU in attesa di poter entrare nella sua regione critica (spin lock). La soluzione a questo problema può essere lasciare che un processo in attesa di entrare nella sua regione critica restituisca volontariamente la CPU allo scheduler.

<pre>void sleep(){     set own state to BLOCKED;     give CPU to scheduler; }</pre>	<pre>void wakeup(process){     set state of process to READY;     give CPU to scheduler; }</pre>
---	--

## PROGRAMMAZIONE CONCORRENTE NEL PROBLEMA PRODUTTORE-CONSUMATORE

Nel problema del produttore-consumatore, 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 (viene riattivato) quando il consumatore preleva dati. Analogamente, il consumatore dorme se il buffer è vuoto e viene risvegliato quando il produttore inserisce dati.

Qual è il problema? Il problema è che il consumatore potrebbe essere risvegliato un attimo prima di andare a dormire e nessuno lo sveglierebbe più.

<pre>#define N 100 int count=0;  void producer(void){     int item;     while(TRUE){         item = produce_item();         if(count==N) sleep();         insert_item(item);         count++;         if(count==1) wakeup(cons);     } }</pre>	<pre>void consumer(void){     int item;     while(TRUE){         if(count==0) sleep();         item = remove_item();         count--;         if(count==N-1) wakeup(prod);         consume_item(item);     } }</pre>
--	--

Si può aggiungere un bit di attesa come rimedio per segnali di risveglio persi.

Possibile soluzione:

1. Si attiva quando un processo non dormiente riceve un 'wakeup'.
2. Se acceso, previene l'entrata in 'sleep' del processo e viene poi spento.
3. Funziona come accumulatore per segnali di risveglio.
4. Viene resettato dal consumatore ad ogni ciclo.

È un workaround, non funziona sempre... ("Porkaround", non farlo!)

## I SEMAFORI

Creati da Dijkstra per contare e gestire i "wakeup".

Valori: 0 se nessun wakeup  
positivo se wakeup in attesa

Operazioni:

- Down
  - o Se il valore del semaforo è maggiore di zero, questo valore viene decrementato, e il processo continua la sua esecuzione.
  - o 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
  - o Se il valore è 0, ci sono processi nella coda di attesa, vengono «svegliati» (eventualmente per entrare in competizione ed eseguire di nuovo down).
  - o In ogni caso, il valore viene incrementato e il processo continua la sua esecuzione.

Atomicità: le operazioni sui semafori sono indivisibili per evitare conflitti.

- Nel problema produttore-consumatore: uso dei semafori per gestire accesso e capacità di buffer.  
Tipi di semafori:

mutex (mutua esclusione, accesso esclusivo)  
full (tutti i posti occupati)  
empty (tutti i posti liberi)

Uso: mutex previene accessi simultanei  
full e empty coordinano attività



## SCENARIO LETTORI E SCRITTORI: REGOLE E PROBLEMI

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

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

Funzionamento Sintetico:

5. Il primo lettore blocca l'accesso al database.
6. Lettori successivi incrementano un contatore.
7. L'ultimo lettore libera l'accesso al database così gli scrittori possono fare il loro lavoro

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

Soluzione Proposta:

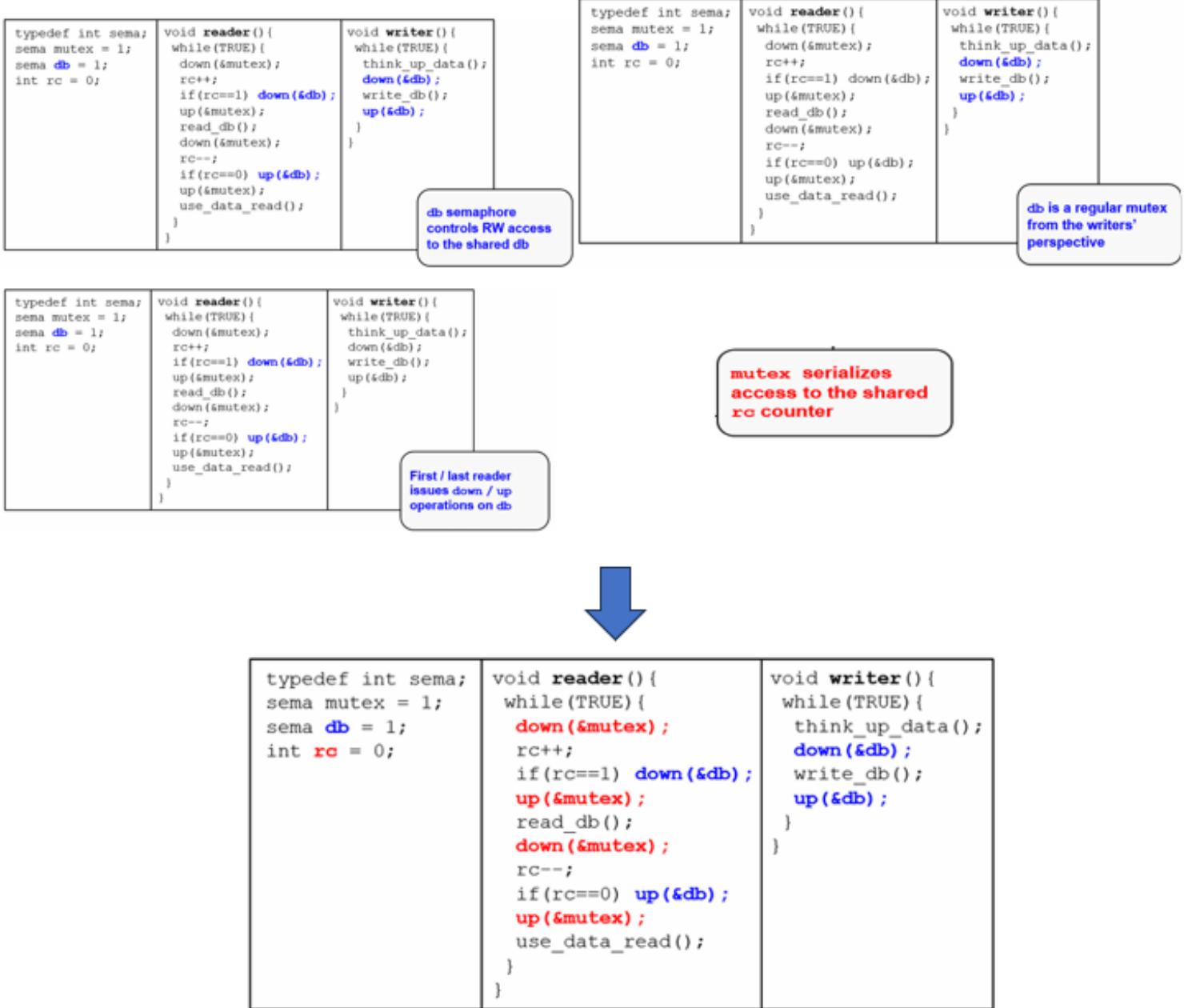
8. Nuovi lettori vengono posti in coda dietro gli scrittori in attesa.
9. Gli scrittori ottengono accesso dopo i lettori già attivi.

Implicazioni:

10. Questo metodo riduce la concorrenza.
11. Potenziale impatto sulle prestazioni.

Alternative:

12. Esistono soluzioni che danno priorità agli scrittori.
13. Ogni strategia ha i suoi vantaggi e svantaggi



## MUTEX E PTHREADS (MUTUA ESCLUSIONE)

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.

Un mutex può essere in due stati: locked (bloccato) e unlocked (sbloccato).

Un bit basta per rappresentarlo, ma spesso viene usato un intero (0 = unlocked, altri valori=locked).

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

Come funziona un mutex?

1. Quando un thread vuole accedere a una regione critica chiama mutex\_lock.
2. Se il mutex è unlocked allora il thread può entrare, invece se è locked il thread attende.
3. Al termine dell'accesso, il thread chiama mutex\_unlock per liberare la risorsa.

Importante: non si utilizza il "Busy waiting" (se un thread non può acquisire un lock, chiama thread\_yield per cedere la CPU ad un altro thread).

Alcune considerazioni:

- I mutex possono essere implementati nello spazio utente con istruzioni come TSL o XCHG.
- Alcuni pacchetti di thread offrono mutex\_trylock (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

Pthread = fornisce funzioni per sincronizzare i thread

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

Funzionamento:

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

Mutex in Phtreads (tabella thread call e description):

- pthread\_mutex\_init: inizializza un oggetto mutex per l'uso, configurando le risorse necessarie.
- pthread\_mutex\_destroy: distrugge un oggetto mutex, liberandole risorse associate. Deve essere chiamato solo se il mutex non è detenuto da alcuni thread.
- pthread\_mutex\_lock: blocca un mutex, sospendendo l'esecuzione del thread chiamante se il mutex è già occupato da un altro thread
- pthread\_mutex\_trylock: tenta di bloccare un mutex senza sospendere l'esecuzione. Se il mutex è già bloccato, la funzione restituisce immediatamente con un codice di errore specifico.
- pthread\_mutex\_unlock: sblocca un mutex, permettendo ad altri thread di acquisirlo. Deve essere chiamato solo dal thread che detiene il lock.
- pthread\_cond\_init: inizializza una variabile di condizione (pthread\_cond\_t) per consentire la sincronizzazione tra thread. La variabile di condizione viene associata a un mutex per coordinare l'accesso a risorse condivise.
- pthread\_cond\_destroy: distrugge una variabile di condizione, liberando le risorse associate. Deve essere chiamata solo quando non ci sono thread in attesa sulla condizione.
- pthread\_cond\_wait: blocca il thread chiamante in attesa della segnalazione di una condizione. Il thread deve detenere un lock sul mutex associato, che viene rilasciato automaticamente durante l'attesa e riacquisito al termine.
- pthread\_cond\_signal: risveglia uno dei thread in attesa sulla variabile di condizione. Se nessun thread è in attesa, la segnalazione viene persa. Questo è utile per notificare il cambiamento di stato di una risorsa condivisa.
- pthread\_cond\_broadcast: risveglia tutti i thread in attesa sulla variabile di condizione. Questo è utile quando un evento deve notificare più thread contemporaneamente, ad esempio, quando una risorsa diventa disponibile per tutti.

## Quando usare lock e quando usare trylock?

- ✓ usare pthread\_mutex\_lock quando l'accesso esclusivo è necessario e possiamo attendere se il lock non è disponibile (esempio: protezione di risorse condivise in operazioni critiche, dove i thread possono aspettare in coda)
- ✓ usare pthread\_mutex\_trylock quando vogliamo solo tentare di acquisire il lock senza aspettare, proseguendo con altre operazioni se il lock è già in uso (esempio: evitare deadlock in scenari complessi o verificare la disponibilità della risorsa senza bloccare il thread)

```
if (pthread_mutex_trylock(&mutex) == 0) {  
    // Se otteniamo il lock, eseguiamo le operazioni protette  
    printf("Ho ottenuto il lock e sto eseguendo operazioni.\n");  
    sleep(2); // Simuliamo qualche operazione lunga  
    printf("Ho terminato e rilascio il lock.\n");  
    // Rilasciamo il lock  
    pthread_mutex_unlock(&mutex);  
} else {  
    // Se il lock non è disponibile, continuiamo con altre operazioni  
    printf("Non sono riuscito a ottenere il lock, continuo con altre operazioni.\n");  
}
```

## 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 una sola 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.

## MUTEX vs VARIABILI CONDIZIONALI

Mutex(pthread\_mutex): Blocca l'accesso a una risorsa, garantisce che solo un thread alla volta possa accedere a una risorsa condivisa, evitando conflitti di accesso. Tuttavia, i mutex da soli non consentono di sincronizzare l'attesa su condizioni specifiche.

Possibili problemi: Se due thread vogliono accedere a una variabile condivisa (ad esempio, un contatore), il mutex può bloccare uno dei thread, garantendo che solo uno acceda alla variabile per volta.

Ma se uno dei thread ha bisogno di aspettare che il contatore raggiunga un certo valore (ad esempio > 10), il mutex da solo non è sufficiente per far attendere il thread in modo efficiente.

## VARIABILI CONDIZIONALI (pthread\_cond)

Le variabili condizionali (pthread\_cond) aggiungono la possibilità di sincronizzare i thread basandosi su condizioni specifiche, piuttosto che solo su accesso esclusivo.

- Perché usare pthread\_cond?
- o Le variabili condizionali permettono ai thread di mettersi in attesa di un evento specifico e di essere notificati solo quando questo evento si verifica.
- o Ad esempio, in uno scenario produttore-consumatore, un thread produttore può notificare al consumatore che è stato prodotto un nuovo elemento e che ora è possibile consumarlo.
- Come funziona:
- o Un thread può attendere con pthread\_cond\_wait su una variabile condizionale finché una condizione specifica non è soddisfatta. Durante l'attesa, il thread rilascia temporaneamente il mutex, permettendo ad altri thread di accedere alla risorsa e di soddisfare la condizione.
- o Usando una variabile condizionale, il consumatore può sospendersi in attesa finché il buffer non è pieno, senza consumare risorse CPU. Quando il produttore aggiunge un elemento al buffer, invia un segnale (pthread\_cond\_signal) che risveglia il consumatore. Questo sistema è molto più efficiente e coordinato.

## CONFRONTO TRA MUTEX E VAR.CONDIZIONALI NELLO SCENARIO PRODUTTORE CONSUMATORE

Senza pthread\_cond\_wait: Se hai solo un mutex, un thread (qui il consumatore) deve usare il busy-waiting per controllare continuamente una condizione, come in questo esempio:

```

pthread_mutex_lock(&mutex);
while (buffer == 0) { // Controllo continuo della condizione (inefficiente)
    pthread_mutex_unlock(&mutex);
    usleep(1000); // Ritardo per ridurre il busy-waiting (non ottimale)
    pthread_mutex_lock(&mutex);
}
consume(buffer);
pthread_mutex_unlock(&mutex);

```

Problema: Il consumatore controlla continuamente buffer == 0, sprecando risorse CPU.

Con le variabili condizionali, il consumatore può sospendersi in modo efficiente:

```

pthread_mutex_lock(&mutex);
while (buffer == 0) {
    // Attesa passiva finché il buffer non è pieno
    pthread_cond_wait(&cond, &mutex);
}
consume(buffer);
pthread_mutex_unlock(&mutex);

```

Soluzione: Il thread consumatore si sospende senza consumare CPU finché il produttore non segnala (pthread\_cond\_signal) che il buffer non è più vuoto.

Vedi esempio 6.4\_producer\_consumer\_pthread.c, nota di questo esempio:

```

...
for (i = 1; i <= MAX; i++) {
    pthread_mutex_lock(&the_mutex);
    while (buffer != 0) {
        pthread_cond_wait(&condp, &the_mutex);
    }
...

```

Protezione della risorsa condivisa: pthread\_mutex\_lock assicura che solo un thread alla volta possa accedere e modificare la risorsa condivisa (in questo caso, il buffer).

Attesa condizionale: Quando un thread chiama pthread\_cond\_wait, due operazioni avvengono atomicamente. Il thread:

1. Rilascia il mutex.
2. Mette il thread in uno stato di attesa sulla variabile condizionale.

Quindi, anche se il produttore ha acquisito il mutex, non lo detiene mentre è in attesa sulla variabile condizionale. Questo permette al consumatore (o a un altro thread) di acquisire il mutex, fare le sue operazioni, e poi mandare un segnale alla variabile condizionale usando pthread\_cond\_signal.

## I MONITOR (MUTUA ESCLUSIONE)

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.

NOTA: 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, permettendo una sincronizzazione e mutua esclusione più sicura e semplice in contesti multithreading. I metodi sono dichiarati synchronized in modo che solo un thread può accedervi (in Java la programmazione concorrente è basata su thread).

```

monitor example
integer i;
condition c;

procedure producer();
|
end;

procedure consumer();
|
end;

end monitor;

```

## MONITOR NELLO SCENARIO PRODUTTORE-CONSUMATORE

```

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);
    }
}

```

Access to enter and remove is serialized by the monitor

wait suspends caller on a condition variable.  
→ Monitor state?

signal wakes up one waiter on a condition variable.  
→ Monitor state?  
→ Lost wakeups?

## 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).
- Problema: possono portare a delle race condition (visto prima...)
  - o Immagina che il processo A voglia svegliare il processo B.
  - o Se il processo A chiama wakeup per il processo B proprio mentre B sta per chiamare sleep...
  - o ... 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.
  - o 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.
  - o 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.

## MONITOR E SEMAFORI

I monitor sono costrutti 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.

## SCAMBI DI MESSAGGI (MUTUA ESCLUSIONE)

È un metodo di comunicazione tra processi che utilizza due primitive: send e receive.

Può essere utilizzato in diversi scenari, compresi sistemi distribuiti.

Problemi:

- Messaggi persi dalla rete.
- Necessità di acknowledgment per confermare la ricezione.
- Gestione dei messaggi duplicati usando numeri sequenziali.
- Autenticazione e denominazione dei processi.

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

Meccanismo di scambio di messaggi e problematiche:

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).

Applicazioni:

- Scambio di messaggi usato in programmazione parallela.
- MPI(Message Passing Interface) è un esempio ben conosciuto usato in elaborazioni scientifiche.

## 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.

## SCAMBI DI MESSAGGI NEL 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:

- i) Il consumatore invia al produttore N messaggi vuoti.
- ii) 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.

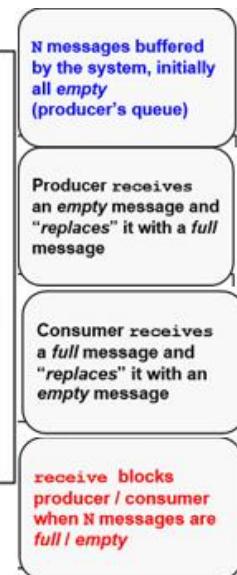
Codice message passing producer-consumer:

```
#define N 100

void producer() {
    int item;
    message msg;

    while(TRUE) {
        item = produce_item();
        receive(consumer, &msg);
        build_message(&msg, item);
        send(consumer, &msg);
    }
}
```

```
void consumer() {
    int item, i;
    message msg;
    for(i=0; i<N; i++)
        send(producer, &msg);
    while(TRUE) {
        receive(producer, &msg);
        item = extract_item();
        send(producer, &msg);
        consume_item(item);
    }
}
```



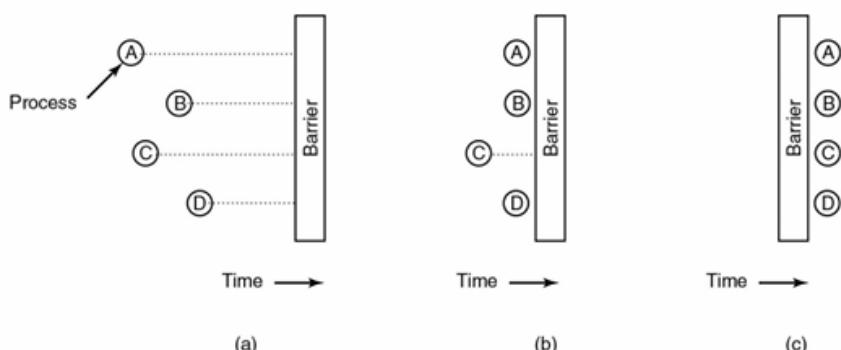
## BARRIERE

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 (ad esempio usando pthread\_join).

Esempio: in calcoli paralleli su matrici, i processi non possono avanzare a un'iterazione successiva finché tutti non hanno terminato l'iterazione attuale.

- (a) Processi che si avvicinano a una barriera.
- (b) Tutti i processi tranne uno vengono bloccati alla barriera.
- (c) Quando l'ultimo processo arriva alla barriera, tutti vengono lasciati passare.



## PROBLEMA DEL MARS PATHFINDER

Il problema si è verificato quando:

- Un thread di bassa priorità stava usando una risorsa condivisa (bloccata da un mutex).
- Un thread di alta priorità ha dovuto aspettare perché la risorsa era bloccata.
- Nel frattempo, un thread di priorità media continuava a lavorare e impediva al thread di bassa priorità di liberare la risorsa.

Questo ha causato un blocco del thread di alta priorità, che non poteva lavorare e quindi attivava un riavvio del sistema.

## VERSIONE SEMPLIFICATA DEL PROBLEMA PRIORITY INVERSION (INVERSIONE DELLE PRIORITA')

Immagina tre persone:

1. Anna(alta priorità) vuole usare una fotocopiatrice.
2. Luca(bassa priorità) sta usando la fotocopiatrice.
3. Marco(priorità media) sta facendo altro, ma sta continuamente interrompendo Luca con domande.

Risultato: Luca non riesce a finire il lavoro e liberare la fotocopiatrice.

Anna, che avrebbe la precedenza, deve aspettare inutilmente.

Per risolvere il problema, è stato applicato il Priority Inheritance Protocol:

- Quando Anna (alta priorità) aspetta Luca (bassa priorità), Luca eredita temporaneamente la priorità di Anna.
  - Questo gli permette di completare velocemente il suo lavoro, ignorando Marco (priorità media).
- Una volta liberata la risorsa, Luca torna alla sua priorità normale.

## READ-COPY-UPDATE

I migliori lock sono quelli che non si usano.

Obiettivo: accessi concorrenti senza lock.

Problema: possibile 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.
- ⇒ Lettori vedono: o la versione vecchia o la nuova, mai un mix delle due

## NOTE LEZIONE:

- mutex "è proprietà mia" mentre il semaforo "non è di mia proprietà, può essere modificata da altri" se dobbiamo sincronizzare processi tra loro meglio semafori, altrimenti meglio mutex
- Ricorda: mutua esclusione è una cosa, sincronizzazione è un'altra

### SCHEDULING DI PROCESSI/THREAD

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 thread, indipendentemente dal processo di appartenenza.

### STORIA E COSTO IN TERMINI DI TEMPO

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

Con l'arrivo della multiprogrammazione, lo scheduling è diventato complesso a causa della concorrenza tra utenti: gli algoritmi di scheduling diventano così cruciali per la prestazione e la soddisfazione dell'utente nei mainframe.

Nei personal computer invece spesso un solo processo è attivo, e la CPU raramente è una risorsa scarsa (la maggior parte dei programmi è limitata dalla velocità dell'input dell'utente).

Per quanto riguarda i server in rete, la CPU spesso è contesa: lo scheduling torna ad essere vitale.

Per i dispositivi IoT, gli smartphone e i nodi di sensori, la durata della batteria è un vincolo cruciale: lo scheduling in questi casi può cercare di ottimizzare il consumo energetico.

Lo scambio di processi (o "context switch") è oneroso per via di:

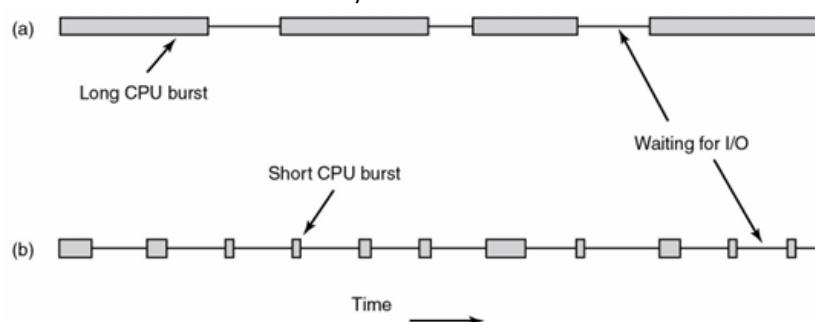
- i. Cambio da modalità utente a modalità kernel
- ii. Salvataggio dello stato del processo
- iii. Esecuzione dell'algoritmo di scheduling
- iv. Cambio della mappa della memoria
- v. Invalidazione potenziale della memoria cache (prossime lezioni...)
- vi. Troppe commutazioni possono consumare tempo di CPU

La prudenza è quindi essenziale.

### INTRODUZIONE AL PROBLEMA DI SCHEDULING DEI PROCESSI

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.

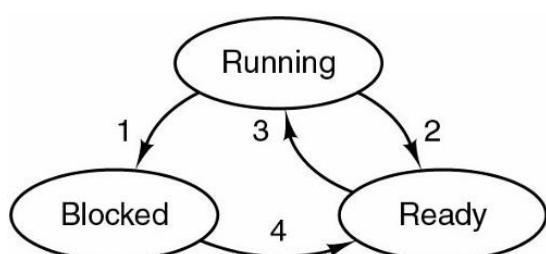


Con CPU più veloci, i processi tendono a essere più I/O-bound (CPU e dischi magnetici non stanno avanzando rapidamente in velocità).

SSD sostituiscono gli hard disk nei PC, ma i data center utilizzano ancora HDD per il costo per bit.

Lo scheduling varia in base al contesto, ciò che funziona per un dispositivo potrebbe non essere efficace per un altro.

### PROCESS STATE REVISITED (STATO DEL PROCESSO RIVISITATO)



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.

Nota: tra running e ready interviene lo scheduler.

## SITUAZIONI IN CUI È NECESSARIO LO SCHEDULING

### A. Creazione Nuovo Processo

- Decisione tra l'esecuzione del processo genitore o figlio.
- Entrambi pronti: può essere scelto chiunque.

### B. 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.

### C. 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.

### D. Interrupt di I/O

- Alla conclusione di un I/O, un processo potrebbe diventare pronto.
- Decidere se eseguire il processo appena pronto, il precedente o un altro.

## TIPOLOGIE DI SCHEDULING E PRELAZIONE

Tipologie di scheduling:

### 1) Non Preemptive (Senza Prelazione)

- Seleziona un processo e lo lascia eseguire fino al blocco o alla 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.

### 2) 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.
- Richiede 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  
(se prelazione => interruzione forzata, ossia il processore dice al processo che non lo esegue più)

## DIVERSITÀ NEGLI AMBITI DI SCHEDULING E LORO OBIETTIVI DEGLI ALGORITMI DI SCHEDULING

Ci sono in base agli ambiti 3 tipologie di esigenze, per ognuna delle quali abbiamo bisogno di schemi di scheduling diversi. Queste sono:

### 1) Sistemi Batch (necessità di massimizzare throughput e di minimizzare tempo di turnaround)

- Caratteristiche:
  - Ideale per attività aziendali periodiche.
  - Accetta algoritmi senza prelazione.
  - Priorità a prestazioni efficienti.
- Obb. alg. di scheduling:
  - Throughput: 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.

### 2) Sistemi Interattivi (necessità di minimizzare tempo di risposta e di ottimizzare adeguatezza)

- Caratteristiche
  - Prelazione fondamentale.
  - Previeni monopolizzazione della CPU.
  - Adatto per server e utenti multipli.
- Obb. alg. di scheduling:
  - Tempo di risposta: Risposta rapida alle richieste degli utenti.
  - Adeguatezza: Soddisfare le aspettative dell'utente in termini di tempi di risposta

### 3) Sistemi Real-time

- Caratteristiche:
  - Processi spesso si bloccano velocemente sapendo di non poter eseguire a lungo.
  - Prelazione non sempre necessaria.
  - Eseguono programmi per specifiche applicazioni, a differenza dei sistemi interattivi che possono eseguire programmi arbitrari.
- Obb. alg. di scheduling:
  - 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

## OBIETTIVI GENERALI DEGLI ALGORITMI DI SCHEDULING

Per tutti i sistemi gli obiettivi che gli algoritmi di scheduling devono rispettare sono:

- ❖ Equità: Garantire un'equa condivisione 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.
- Nei sistemi real-time è cruciale rispettare le scadenze e garantire la prevedibilità

## SCHEDULING IN SISTEMI BATCH

Alcuni possibili metodi per sistemi batch sono:

### 1) FIRST-COME FIRST-SERVED

- Descrizione:
  - Algoritmo di scheduling senza prelazione.
  - Processi assegnati alla CPU nell'ordine in cui arrivano.
  - Una singola coda di processi in stato pronto. Il primo job esegue immediatamente senza interruzioni.
  - Processi bloccati ritornano in fondo alla coda.
- Vantaggi:
  - Facile da capire e programmare.
  - Gestione semplice con una linked list.
  - 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.

### 2) 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 (il tempo dallo start all'end di un job) quando tutti i job sono disponibili contemporaneamente.
- Limitazione:
  - Se i job arrivano in momenti diversi, SJF potrebbe non essere ottimale.
  - Esempio: 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.

### 3) 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.

Importante per sistemi batch: "la prelazione viene imposta dall'esterno".

Importante: quando si parla di lavori in batch i job necessitano di molto tempo per essere eseguiti (si parla almeno di minuti e si può arrivare a settimane), inoltre quando si parla di processi batch di solito io ho bisogno del risultato finale quindi ho bisogno che terminino per avere il risultato.

## SCHEDULING IN SISTEMI INTERATTIVI

Nei sistemi interattivi il tempo di risposta è fondamentale per poter rispondere rapidamente alle richieste. Inoltre, in questi sistemi, si ricerca la proporzionalità in quanto occorre soddisfare le aspettative degli utenti. Metodi utilizzati in questi sistemi sono:

### 1) ROUND-ROBIN SCHEDULING

- Concetto:
  - o Uno degli algoritmi di scheduling più vecchi, semplici, equi e ampiamente utilizzati.
  - o Ogni processo riceve un intervallo di tempo o "quanto" per l'esecuzione.
  - o Se il processo non ha terminato al termine del quanto, la CPU è oggetto di prelazione per un altro processo.
  - o Se un processo termina o si blocca prima del quanto, il passaggio avviene automaticamente
- Implementazione:
  - o Mantenere una lista dei processi eseguibili.
  - o Una volta esaurito il quanto, il processo viene spostato alla fine della lista
  - o Tips: vedilo come una lista circolare!



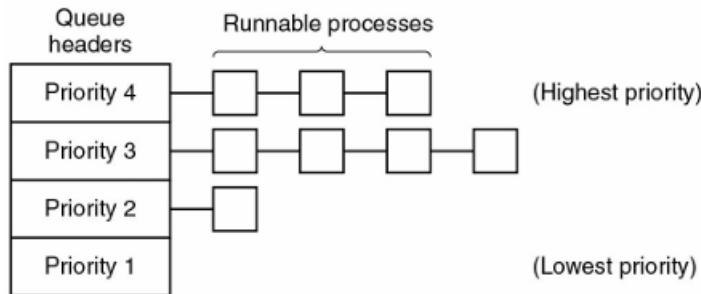
- Durata del Quanto:
  - o La scelta del quanto influenza sull'efficienza.
  - o Supponendo 1 ms per il cambio di contesto e 4 ms per il quanto: il 20% del tempo CPU sprecato in overhead.
- Trade-off:
  - o Quanto lungo: riduce l'overhead, ma peggiora la reattività (es. 5 secondi di attesa per un breve comando in un server affollato).
  - o Quanto breve: maggiore overhead e riduzione dell'efficienza della CPU
- Ottimizzazione:
  - o Se il quanto è maggiore del tempo medio di burst di CPU, la prelazione potrebbe non avvenire spesso. Molti processi potrebbero bloccarsi prima.
  - o Compromesso: un quanto tra 20 e 50 ms è spesso ragionevole per bilanciare efficienza e reattività.

### 2) PRIORITY SCHEDULING (SCHEDULING A PRIORITA')

- Premessa:
  - o Round-robin considera tutti i processi ugualmente importanti.
  - o Alcuni contesti richiedono una gerarchia (es. università con differenti ruoli).
- Concetti:
  - o Ogni processo ha una priorità assegnata.
  - o La CPU esegue il processo con la priorità più alta tra quelli pronti.
  - o Applicabile anche su singoli PC: (ad es. un daemon di posta elettronica avrebbe meno priorità di un video in tempo reale).
- Funzionamento gestione delle priorità:
  - o Priorità del processo attualmente in esecuzione può diminuire con il tempo.
  - o Se scende sotto quella del processo successivo, avviene uno cambio.
  - o Possibilità di assegnare un quanto di tempo: al suo esaurirsi, si passa al processo con priorità appena inferiore.
  - o Evitare che processi rimangano inibiti indefinitamente, altrimenti potrebbero finire a priorità 0.
- Funzionamento priorità statica vs dinamica:
  - o Statica: es. gerarchie militari o basate sui costi nel data center.
  - o Dinamica: es. basata sull'utilizzo della CPU o sul comportamento I/O bound.
- Da notare:
  - o Nasce per tenere conto dell'importanza dei processi, ad esempio processi del S.O. sono più importanti degli altri e quindi devono avere priorità maggiore.
  - o (extra slide) l'importanza più alta in Linux corrisponde a numeri piccoli (quindi maggiore è la priorità minore sarà il valore associato alla priorità)
  - o La priorità può essere statica o dinamica

## STRATEGIE COMBINATE E CLASSI DI PRIORITÀ

- Raggruppamento in classi:
  - o Processi divisi in classi di priorità.
  - o Scheduling a priorità tra le classi, ma round-robin all'interno della stessa classe.
- Esempio:
  - o Sistema con 4 classi di priorità.
  - o Fintanto ci sono processi in priorità 4, si usano in round-robin. Se vuota, si passa alla 3, poi alla 2, e così via.
  - o Importante rivedere periodicamente le priorità per evitare che processi a bassa priorità non vengano mai eseguiti (morendo di inedia)



### 3) SHORTEST PROCESS NEXT (CON AGING)

- Idea: "Shortest Job First" ottimizza il tempo medio di risposta nei sistemi batch.
- Obiettivo: l'obiettivo è applicarlo ai sistemi interattivi. Ricordare che processi lunghi potrebbero aspettare e morire di inedia (starvation).
- Sfida: identificare quale tra i processi eseguibili sia effettivamente il più breve.
- Soluzione-Aging: fare stime basate sul comportamento passato.
- Utilizzo dell'Aging: Prevista situazioni dove si basa la previsione su valori passati.
- Da note lezione:
  - o si cerca di modificare la priorità nel tempo per favorire i processi con priorità minore degli altri (si cerca di evitare starvation)
  - o Scenario: necessità di bilanciare un processo a cui manca poco tempo per essere finire l'esecuzione ma che ha priorità molto bassa.
  - o Soluzione in questo schema: se a un processo basso mancano 55 secondi, facciamo in modo che dichiari gliene manchino 57.
  - o "aging" -> "invecchiamento"

### 4) GUARANTEED SCHEDULING (difficile da realizzare nel concreto)

- Concetto Principale: Fare promesse concrete sugli standard di prestazione e rispettarle.
- Promessa Base: Se ci sono n utenti o processi, ciascuno ottiene  $\sim 1/n$  della potenza della CPU.
- Come Funziona:
  - o Il sistema tiene traccia di quanta CPU ha ricevuto ogni processo dal momento della sua creazione (esempio 100 secondi)
  - o Calcola quanto tempo CPU ogni processo dovrebbe avere (tempo da creazione  $\div n$ , ad esempio 100 sec / 10 processi, un processo dovrebbe avere 10 secondi).
  - o Valuta il rapporto tra il tempo CPU consumato e quello dovuto.  
(Rapporto di 0,5: ha avuto metà di quanto dovuto).  
(Rapporto di 2,0: ha avuto doppio di quanto dovuto).
  - o Esegue il processo con il rapporto più basso finché non supera il suo concorrente più vicino.

### 5) LOTTERY SCHEDULING (abbastanza semplice da realizzare)

- Concetto di base:
  - o Assegnazione di biglietti della lotteria ai processi per le risorse del sistema, come il tempo della CPU.
  - o Estrazione casuale di un biglietto per decidere quale processo ottiene la risorsa.
  - o Esempio: Estrazione 50 volte al secondo => vincitore riceve 20ms di tempo della CPU.
- Distribuzione delle probabilità:
  - o Biglietti extra per processi più importanti => maggiori probabilità di vincere.
  - o Esempio: Se un processo ha il 20% dei biglietti, guadagnerà a lungo termine il 20% della CPU.

- Proprietà e applicazione dello scheduling a lotteria – Reattività: risponde velocemente ai nuovi processi grazie alla distribuzione dei biglietti
- Proprietà e applicazione dello scheduling a lotteria – Cooperazione tra processi:
  - o Possibilità di scambiarsi biglietti tra processi cooperanti.
  - o Esempio: Un processo client dona i suoi biglietti a un processo server per farlo eseguire più rapidamente.
- Proprietà e applicazione dello scheduling a lotteria – Soluzione a problemi complessi:
  - o Adatto a situazioni dove altri metodi falliscono.
  - o Esempio: Server video con diverse necessità di frequenze di fotogrammi. (Assegnazione di biglietti in base alla velocità necessaria => divisione automatica della CPU nelle proporzioni corrette).

## 6) FAIR-SHARE SCHEDULING

- Premessa:
  - o Tradizionalmente, ogni processo è oggetto di scheduling individualmente.
  - o 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:
  - o Considera la proprietà di ogni processo prima di considerarlo.
  - o Ogni utente riceve una frazione predefinita di CPU.
  - o Lo scheduler si assicura che ogni utente riceva la sua frazione, indipendentemente dal numero di processi posseduti.
- Esempio:
  - o Due utenti, ciascuno con il 50% della CPU (Utente 1 ha processi A, B, C, D) (Utente 2 ha processo E)
  - o Sequenza con round-robin: A E B E C E D E A E ...
  - o 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 IN SISTEMI REALTIME

Usato nei sistemi operativi in applicazioni in cui il tempo di risposta è fondamentale (ad esempio per lettori di compact disc, monitoraggio in terapia intensiva, piloti automatici, controllo robotico in fabbriche).

In questi sistemi ritardi o mancati tempi di risposta possono avere gravi implicazioni.

Lo scheduling in sistemi real-time si divide in 2 categorie:

- 1) Hard real-time: scadenze assolute da rispettare
- 2) Soft real-time: qualche scadenza mancata è tollerabile

Comportamento dei processi in sistemi real time: processi prevedibili, brevi e noti in anticipo.

Tipi di eventi che lo scheduling in sistemi real-time deve gestire:

- 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 Pi e richiede Ci secondi di tempo della CPU per gestire ogni evento, allora il carico può essere gestito solo se

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

Esempi e tipologie di algoritmi real-time:

- Eventi periodici di 100ms, 200ms, 500ms
- tempi richiesti di 50ms, 30ms, 100ms
- condizione  $0,5+0,15+0,2 < 1$

	Periodo	Tempi Richiesti
P1	100 ms	50 ms
P2	200 ms	30 ms
P3	500 ms	100 ms

Gli algoritmi di scheduling possono essere:

- 1) statici -> se decisioni prese prima dell'esecuzione
- 2) dinamici -> se 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.
- Scenario reale: Un processo può avere molti processi figli sotto il suo controllo.
- Esempio: Un sistema di gestione di una base di dati con molti figli, ognuno con funzioni specifiche.
- Problematica: Gli scheduler tradizionali non accettano input dai processi utente, spesso portando a decisioni sub-ottimali. Quindi un processo utente non può mai far passare prima i suoi processi parenti.

## SEPARAZIONE TRA MECCANISMO E POLITICA DI SCHEDULING

- Principio: Separare il meccanismo di scheduling dalla politica di scheduling (Levin et al., 1975).
- 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.
  - 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 (concetto fondamentale).

## PARALLELISMO: PROCESSI E THREAD

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 E SCHEDULING DEI THREAD A LIVELLO UTENTE

Per quanto riguarda i thread a livello utente il kernel ignora l'esistenza dei thread; sceglie infatti un processo per il suo quanto. È poi il thread interno a decidere quale thread eseguire senza interruzione del clock. Il risultato è quindi che un thread può consumare l'intero quanto del processo, influenzando solo il processo interno e non gli altri.

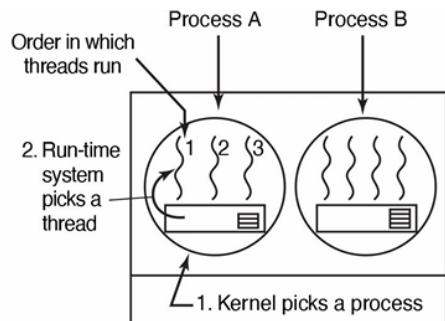
(da appunti lezione:)

SCHEDULING DEI THREAD A LIVELLO UTENTE: cosa succede quando ha a che fare coi thread? se thread gestito dal processo (quindi kernel non sa nulla) allora lo scheduler lo vede come un processo uguale a tutti gli altri. Se il kernel ha invece visibilità dei thread è il kernel ad occuparsi dell'ordine in cui i thread vengono eseguiti, questo porta a maggiore flessibilità ma bisogna considerare che devo pagare overhead per cambi di contesto.

Scheduling dei thread a livello utente:

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

Possibile scheduling di thread a livello utente con un quanto per un processo di 50 msec e thread che eseguono 5 msec per burst della CPU.

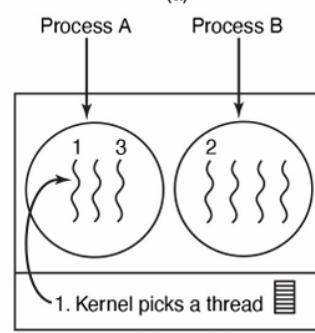


Possible: A1, A2, A3, A1, A2, A3  
Not possible: A1, B1, A2, B2, A3, B3

(a)

- **Thread del kernel:** Il kernel seleziona un thread specifico per l'esecuzione.
  - Ordine potenziale: A1, B1, A2, B2... (come in Figura (b)).
  - Se un thread eccede il quanto, viene sospeso.

Possibile schedulazione di thread a livello kernel con le stesse caratteristiche della diapositiva precedente



Possible: A1, A2, A3, A1, A2, A3  
Also possible: A1, B1, A2, B2, A3, B3

(b)

## DIFFERENZE THREAD LIVELLO UTENTE vs THREAD DEL KERNEL E ALCUNE CONSIDERAZIONI

Thread a livello utente vs. Thread del kernel:

- Scambio thread utente: poche istruzioni.
- Scambio thread kernel: scambio completo di contesto => più lento.
- Blocco su I/O:
  - o con thread utente, intero processo sospeso;
  - o con thread kernel, solo il thread specifico.

Decisioni del kernel:

- Considera i costi per passare da un thread a un altro.
- Può dare preferenza ai thread dello stesso processo.

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

## BREVE RECAP UTILE

### MODELLO DI ESECUZIONE E GESTIONE DEI PROCESSI

- Concetti base: Processi sequenziali eseguiti in parallelo per nascondere gli effetti degli interrupt.
- Caratteristiche dei Processi:
  - o Creati e terminati dinamicamente.
  - o Spazio di indirizzi unico per ogni processo.
- Thread:
  - o Multipli thread di controllo in un singolo processo.
  - o «Schedulati» indipendentemente con stack propri.
  - o Condivisione dello stesso spazio di indirizzi.
  - o Possono essere implementati nello spazio utente o nel kernel.

### SINCRONIZZAZIONE, STATI E SCHEDULING

- Sincronizzazione e comunicazione:
  - o Uso di semafori, monitor, messaggi.
  - o Prevengono l'accesso simultaneo a regioni critiche per evitare caos.
  - o Processi possono essere: in esecuzione, eseguibili o bloccati.
  - o Cambiamenti di stato causati da primitive di comunicazione.
- Algoritmi di Scheduling:
  - o Varietà studiata: shortest-job-first, round robin, scheduling a priorità, code multilivello, e molti altri.
  - o Distinzione in alcuni sistemi tra meccanismo di scheduling e policy, permettendo agli utenti di influenzare l'algoritmo di scheduling.

### NOTA EXTRA DA LEZIONE SEGUITA IN PRESENZA:

Scrivendo "time" prima di un comando in bash otteniamo informazioni su tempo e risorse usate dal comando.

TERMINE PRIMO MACRO BLOCCO DI LEZIONI IN CUI ABBIAMO VISTO LA GESTIONE DI PROCESSI, LE LORO CARATTERISTICHE, I THREAD RISPETTO AL PROCESSO. ABBIAMO POI VISTO SINCRONIZZAZIONE E COMUNICAZIONE, E ANCHE GLI ALGORITMI DI SCHEDULING.  
DALLA PROSSIMA LEZIONE IN POI VEDREMO COME S.O. ASTRAE LA MEMORIA.

## **LEZ8 (19/11/2024)**

### GESTIONE DELLA MEMORIA NEI SISTEMI OPERATIVI

Memoria Principale (RAM): cresce rapidamente, ma i programmi crescono più velocemente. Nasce dal desiderio di avere una memoria privata, grande, veloce, persistente e a basso costo (la realtà tecnologica è diversa).

Gerarchia della Memoria va da memoria veloce e costosa a memoria lenta, economica e di grandi dimensioni. Il compito del Sistema Operativo è quello di astrazione di questa gerarchia in un modello utile gestendone anche l'astrazione.

Il gestore della Memoria gestisce la memoria e la sua gerarchia, inoltre traccia l'uso della memoria e alloca e libera memoria per i processi.

Focus di questa lezione: memoria principale (storage di massa/dischi in lezioni successive)

Per trattare il tema della gestione della memoria parleremo in questa e nelle prossime lezioni di:

- A) MEMORY ABSTRACTION (i processi pensano la memoria sia organizzata in un modo ma in realtà no),
- B) VIRTUAL MEMORY,
- C) ALGORITMI DI SOSTITUZIONE DELLE PAGINE,
- D) PROBLEMI DI PROGETTAZIONE PER SISTEMI DI PAGING.

-----Inizio A)Memory Abstraction-----

### MEMORIA SENZA ASTRAZIONE

Il modello più semplice è l'uso diretto della memoria fisica (es. in mainframe ogni programma vedeva solo memoria fisica)(ad esempio con l'istruzione MOV REGISTER,1000 la locazione fisica di memoria 1000 veniva trasferita in REGISTER1).

C'è un problema però: cosa succede se un programma interferisce con un altro?...

Monoprogrammazione vs multiprogrammazione con memoria senza astrazione:

#### → Monoprogrammazione

Tre modelli principali di organizzazione della memoria:

- b) OS in RAM (utilizzato sui mainframe e sui minicomputer)
- c) OS in ROM (sistemi embedded)
- d) OS+drivers in ROM+RAM (primi personal computer)

#### → Multiprogrammazione

Possibilità di eseguire più programmi contemporaneamente senza astrazione della memoria usando "swapping", per swapping si intende il salvataggio del contenuto della memoria in un file su memoria non volatile e prelievo del programma successivo.

Approccio ingenuo: caricamento di più programmi in memoria fisica consecutivamente, senza astrazione dell'indirizzo.

- a) Un programma di 16 KB che inizia con l'istruzione JMP 24.
- b) Un altro programma di 16 KB che inizia con l'istruzione JMP 28.
- c) 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.

### ASTRAZIONE DELLA MEMORIA

- 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.
- Soluzione: Astrazione della memoria per separare e proteggere i programmi in esecuzione.
- Concetto di Spazio degli Indirizzi:
  - o Ogni programma ha un insieme unico di indirizzi (spazio degli indirizzi) che può usare per indirizzare la memoria.
  - o Questo spazio è indipendente da altri processi e rappresenta una forma di memoria astratta.

Nota: Passando da memoria senza astrazione a memoria con astrazione passiamo dal concetto di indirizzi a quello di spazi di indirizzi.

## REGISTRO BASE E REGISTRO LIMITE

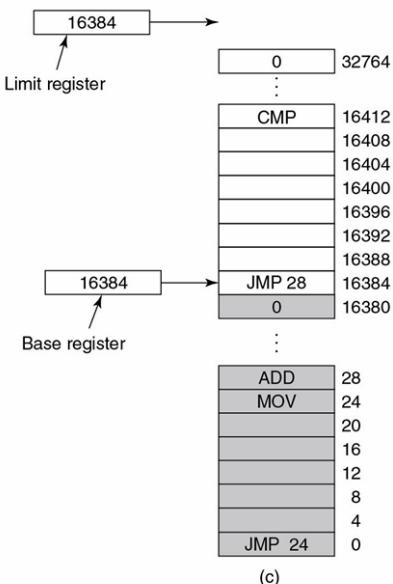
Vecchia soluzione: mappare lo spazio degli indirizzi di ogni processo in parti diverse della memoria fisica.

I registri 'Base' e 'Limite' sono due registri hardware speciali presenti in molte CPU:

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

2. 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

**Checks for MOV Reg1, Addr:**

**IF(Addr > LIMIT) → NOT OKAY**  
**IF(BASE + Addr < BASE) → NOT OKAY**

Nota: ogni processo ha la sua base

## **Base and limit registers**

Funzionamento dei registri = ogni riferimento alla memoria da parte di un programma:

1. Aggiunge il valore del registro base all'indirizzo generato.
2. 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.

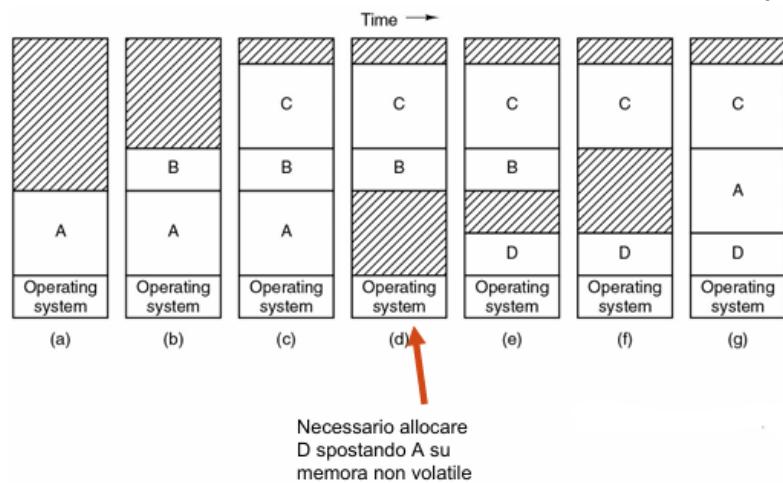
Svantaggi: Necessità di eseguire somme e confronti ad ogni accesso alla memoria, il che può essere lento.

## COSA SUCCIDE SE ESCO FUORI DALLA MEMORIA?

Le strategie per gestire il sovraccarico di memoria sono le seguenti

1. Swapping (Scambio) dei processi:
  - 1.1 Sposta interi processi tra la memoria RAM e la memoria non volatile (disco/SSD).
  - 1.2 Processi inattivi archiviati su memoria non volatile (vedi slide successiva)
2. Memoria Virtuale:
  - 2.1 Permette l'esecuzione dei programmi anche se solo parzialmente presenti nella memoria principale (prossime lezioni).

## MIGLIORAMENTO: DYNAMIC PARTITIONS E SWAPPING (PARTIZIONI DIMANICHE E SCAMBI)



Lo scambio può portare alla frammentazione della memoria.

Diventa quindi necessaria la compattazione della memoria che è però estremamente lenta.  
(stiamo sempre parlando di memoria principale)

## GESTIONE DELLO SPAZIO E CRESCITA DEI PROCESSI

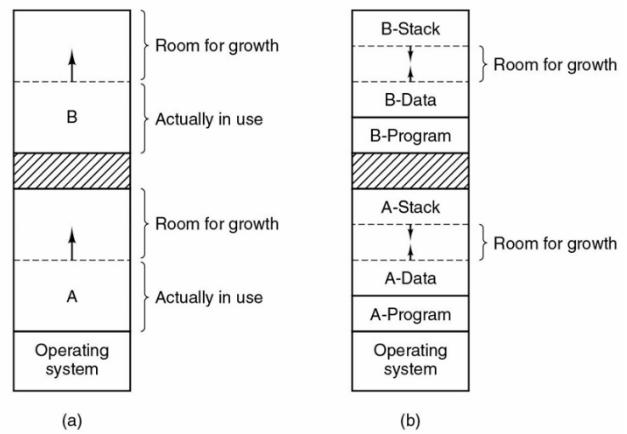
Sfida: gestire processi con segmenti di dati in crescita.

Memory Compaction: sposta processi per liberare spazio, ma richiede tempo.

Soluzione: allocare memoria extra durante lo swapping o lo spostamento dei processi.

Cosa fare in caso di Out of Memory?

1. «Uccidere» il processo
2. Trasferire il processo
3. Swapping



(a) Spazio allocato per segmento dati in crescita.  
(b) Spazio per stack e segmento dati che crescono.

## GESTIONE DELLA MEMORIA LIBERA (LIBERA=NON OCCUPATA)

### → GESTIONE DINAMICA DELLA MEMORIA

Obiettivo: tenere traccia dell'utilizzo della memoria (ad esempio ogni blocco di 4 bytes).

Metodi principali:

1. Soluzione1 = Bitmap tiene traccia di quali blocchi vengono allocati.
2. Soluzione2 = Una lista collegata tiene traccia della memoria non allocata.

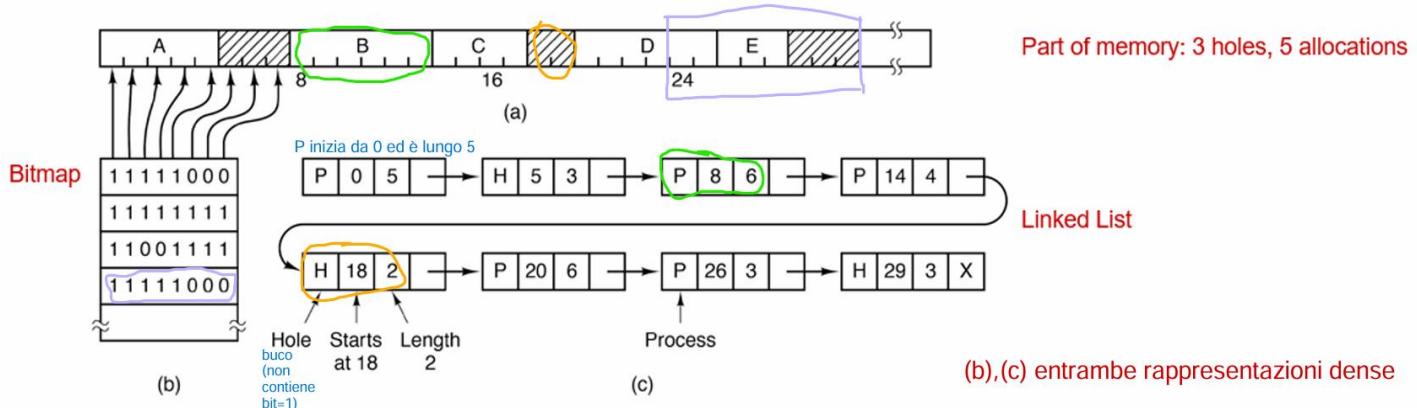
Importanza: questo tracciamento non riguarda solo la memoria, ma anche risorse come i file system

### → MEMORY MANAGEMENT: BITMAP VS LINKED LISTS

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 (coalescenza = quando due processi si fondono e formano un processo più grande)

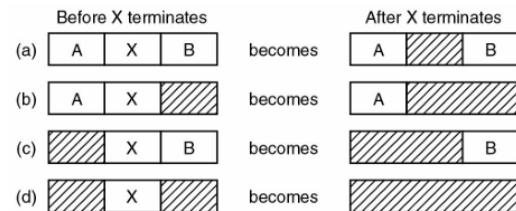


Definizioni matrice densa e matrice sparsa:

- Una matrice è considerata "densa" quando la maggior parte delle sue celle contiene valori significativi (sia 0 che 1). La rappresentazione densa memorizza tutti gli elementi della matrice, indipendentemente dal loro valore. Ogni cella è memorizzata esplicitamente in memoria.  
Vantaggio: semplice da implementare e da manipolare.  
Svantaggio: spreco di memoria per celle vuote e inefficiente per matrici grandi con molti valori irrilevanti.
- Una matrice è "sparsa" quando la maggior parte delle sue celle contiene un valore predefinito o trascurabile (spesso 0). La rappresentazione sparsa memorizza solo i valori non nulli (di solito gli 1 in una matrice binaria) insieme alle loro posizioni nella matrice.  
Vantaggio: risparmio di memoria.  
Svantaggio: richiede un overhead aggiuntivo per memorizzare le posizioni, inoltre accesso più lento agli elementi.

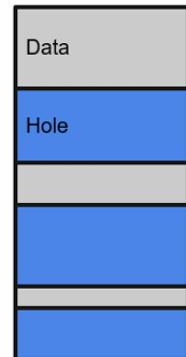
## → MEMORY MANAGEMENT E LINKED LISTS

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



## → SCHEMI DI ALLOCAZIONE DELLA MEMORIA

1. FIRST FIT: Seleziona il primo spazio disponibile (Opzione più semplice).
2. NEXT FIT: Seleziona il successivo spazio disponibile (Più lento del First Fit in pratica).
3. BEST FIT: Sceglie lo spazio più adeguato (Tende alla frammentazione).
4. WORST FIT: Sceglie lo spazio meno adeguato (Prestazioni scadenti in pratica).
5. QUICK FIT: Mantiene spazi di dimensioni diverse (le più richieste)(scarsa performance nella coalescenza).
6. BUDDY ALLOCATION (Linux): Migliora la performance di coalescenza del Quick Fit.



## → ALLOCAZIONE DELLA MEMORIA IN LINUX

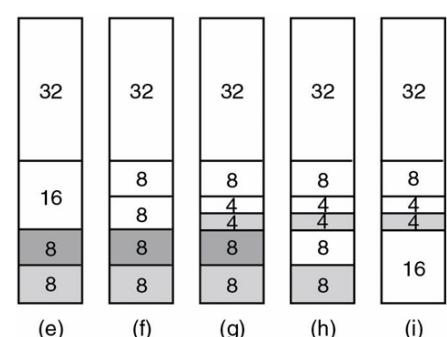
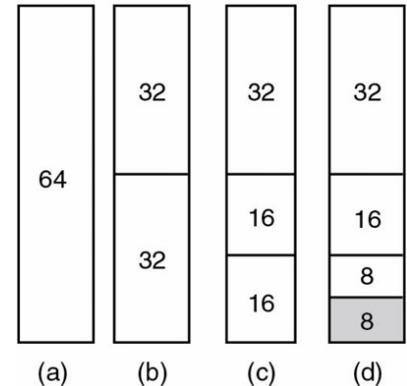
Linux utilizza vari meccanismi per l'allocazione della memoria. Il principale è una allocazione delle pagine basato sull'algoritmo di Buddy Memory Allocation.

Funzionamento:

1. La memoria inizia come un singolo pezzo contiguo.
2. Ad ogni richiesta, la memoria viene divisa secondo una potenza di 2.
3. Blocchi di memoria contigui vengono uniti quando rilasciati

## → ESEMPIO DI BUDDY MEMORY ALLOCATION

- Inizialmente la memoria consiste di un singolo pezzo contiguo (64 pagine nell'esempio (a)).
- Arriva una richiesta da 8 otto pagine (Si approssima alla prossima potenza di 2 più grande, se arriva 7 lo approssimo a 8, se arriva tre lo approssimo a 4 etc..).
- 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; questa può essere 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).

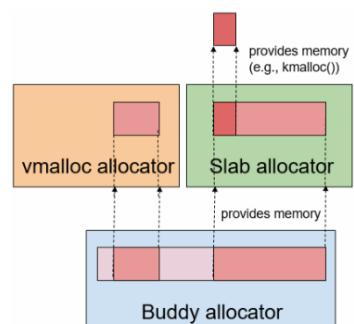


Oss: pensa alle pagine come dei blocchetti di memoria

## → FRAMMENTAZIONE E SLAB ALLOCATOR

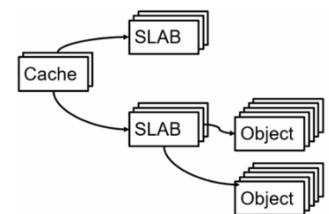
Il Buddy Algorithm può causare frammentazione interna: una richiesta di 65 pagine richiede l'allocazione di 128 pagine.

Lo SLAB allocator in Linux risolve questo problema prendendo blocchi tramite l'algoritmo buddy e ritagliando unità più piccole (slab) per gestirle separatamente.



## SLAB ALLOCATOR

- Principio di base:
  - o Il kernel spesso ha bisogno di creare e distruggere piccoli oggetti di dimensioni e tipi specifici.
  - o Senza ottimizzazione, questa operazione potrebbe portare a una significativa frammentazione della memoria
- Come Funziona:
  - o Nello slab allocation, la memoria è divisa in blocchi chiamati "slabs"
    - ulteriormente suddivisi in chunk di dimensioni uniformi,
    - adeguati per ospitare un oggetto di un certo tipo.
  - o 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).



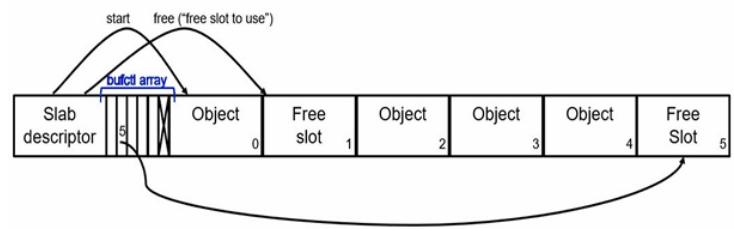
Slab = pagine richieste dal S.O a se stesso.

## SLAB E CACHING

Quando un oggetto viene deallocated, non viene immediatamente restituito al sistema come memoria libera ma viene mantenuto nella cache in modo che, se viene richiesta un'altra istanza dello stesso tipo di oggetto, può essere rapidamente riallocata senza l'overhead di inizializzazione.

Nell'esempio seguente, uno Slab contiene:

- puntatore all'inizio della memoria con gli slot degli oggetti
- indice del prossimo slot libero
- bufctl: array di indici dei prossimi oggetti liberi
- Slot per gli oggetti



## DOMANDE FONDAMENTALI A CUI DEVI SAPER RISPONDERE PER PASSARE SISTEMI OPERATIVI:

1) Quanti numeri servono per fare riferimento a un numero?

Per sapere quanti bit sono necessari per rappresentare un intervallo di numeri, si usa il calcolo del logaritmo in base 2:

- Esempio: se devi fare riferimento a numeri da 0 a 32764, ci sono  $32765 = 2^{15} - 1$  numeri in totale.
- La formula è:  $n \geq 2^b$ , dove  $b$  è il numero di bit. In questo caso, servono 15 bit per rappresentare l'intervallo.

2) Come si calcola il logaritmo in base 2?

Il logaritmo in base 2 ( $\log_2(x)$ ) si calcola con la formula:

$$\log_2(x) = \frac{\log_{10}(x)}{\log_{10}(2)}$$

oppure usando una calcolatrice che supporta logaritmi in base 2 direttamente.

- **Passaggio di base tra logaritmi:** Se conosci il logaritmo in una base diversa ( $\log_b(x)$ ), puoi passare a base 2:

$$\log_2(x) = \frac{\log_b(x)}{\log_b(2)}$$

3) Quanti bit servono per numeri da 0 a  $2^{16}$ ?

Numeri da 0 a  $(2^{16}) - 1$  richiedono 16 bit, perché  $2^{16}$  rappresenta l'intervallo massimo per un valore a 16 bit.

4) Quanti bit per rappresentare un numero in una base diversa?

Per rappresentare un numero in una base  $b$ , usa la formula:

$$n = \lceil \log_b(N) \rceil$$

dove  $N$  è il valore massimo che vuoi rappresentare e il risultato si arrotonda all'intero superiore.

- Ad esempio: Per rappresentare numeri fino a 1000 in base 10, calcola:

$$\log_{10}(1000) = 3$$

Quindi, servono almeno 3 cifre in base 10. In binario, calcola  $\log_2(1000)$  per sapere i bit necessari.

## LEZ9

### IL PROBLEMA DEL BLOATWARE E LA CRESCITA DELLA MEMORIA

Potrebbe esserci necessità di gestire programmi che superano la capacità della memoria disponibile, il problema dei programmi più grandi della memoria esiste da sempre.

Negli anni 60' c'è stata l'introduzione di tecniche per dividere programmi in parti gestibili:

- Overlay: sono piccole parti o segmenti di un programma.
- Solo l'overlay necessario viene caricato in memoria.
- Overlay successivi sovrascrivono o coesistono con quelli precedenti.
- Gli overlay vengono scambiati tra memoria e disco.

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

### MEMORIA VIRTUALE

La memoria virtuale estende l'idea dei registri base e limite. Quindi, ogni programma 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), ossia una divisione dello spazio degli indirizzi in unità di dimensione fissa, es. 4 KB. Una sua alternativa potrebbe essere la "segmentazione" con unità di dimensione variabile che è ora meno comune.

Problema: finora la memoria può essere assegnata ai processi solo in blocchi contigui.

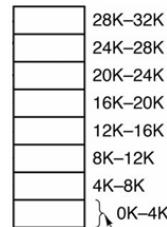
Soluzione (e vantaggi dell'uso della Memoria Virtuale):

1. Creare per il processo l'illusione di uno spazio di indirizzi ampio (ad esempio indicizzabile con 48 bit!!!).
2. Questo spazio è noto come spazio di indirizzi virtuale
3. La RAM(molto più limitata) è nota come memoria fisica.
4. Memory Management Unit (MMU): traduce gli indirizzi virtuali (come usati dal processo) in indirizzi fisici.

### MEMORIA VIRTUALE E PAGINAZIONE

I sistemi moderni utilizzano la paginazione (o paging) 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).

Pagina fisica (frame) graficamente:



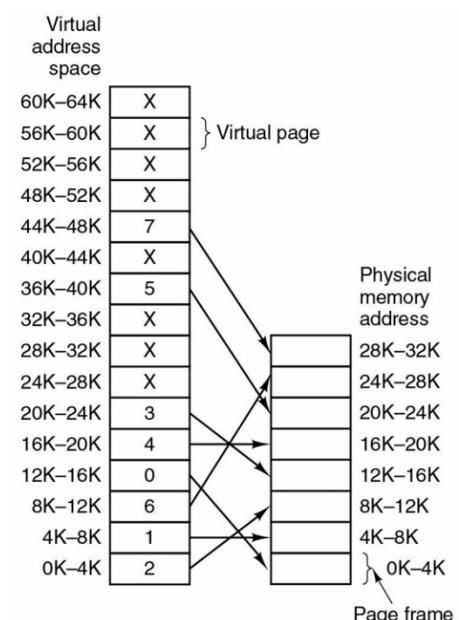
### SPAZIO DI INDIRIZZAMENTO VIRTUALE VS SPAZIO DEGLI INDIRIZZI FISICI + PAGE TABLE

Mappatura Memoria: 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).

Se un programma fa riferimento a una pagina non mappata, si verifica un «Page fault». Il sistema operativo allora:

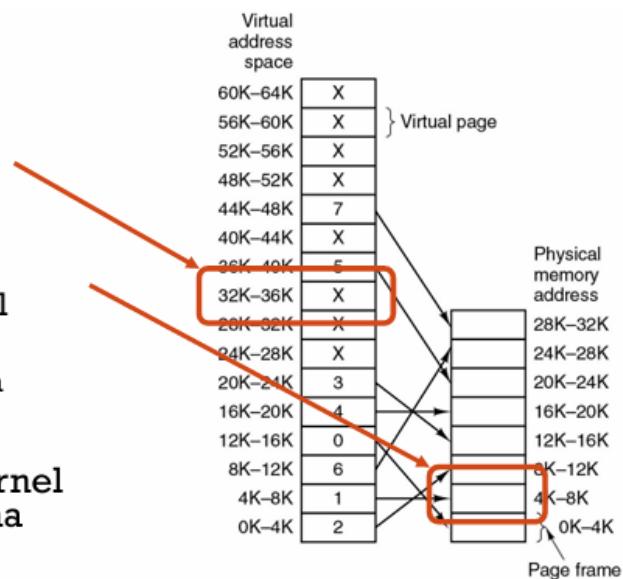
1. Sposta un frame raramente usato su disco, se serve
2. Carica la pagina richiesta nel frame libero o liberato.
3. Aggiorna la mappa della MMU per riflettere i cambiamenti.

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



Esempio:

- **Esempio: gestire istruzione**
  - MOV REG, 32780
- Fa riferimento alla pagina virtuale 8.
  - Indirizzo 12 della pagina
  - $32780 - 2^{15}$  (32768) = 12
- 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



## FUNZIONAMENTO INTERNO DELLA MMU

### ▪ Indirizzo Virtuale: 8196

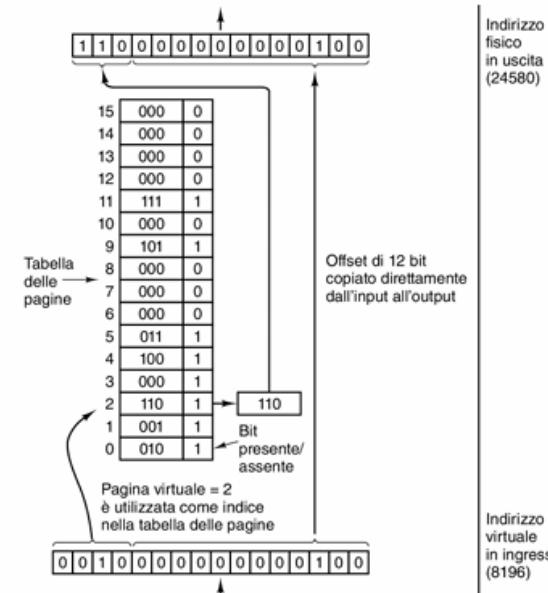
- Rappresentazione Binaria:
  - 0010 00000000100      [ 0010=>seconda pagina ]  
blocco2      indirizzo 100

### ▪ Suddivisione dell'Indirizzo Virtuale:

- Numero di pagina: 4 bit (permette di gestire 16 pagine)
- Offset: 12 bit (indirizza 4096 byte per pagina che compongono ogni frame)

### ▪ Mappatura tramite la Tabella delle Pagine:

- Numero di pagina →
  - Indice nella tabella delle pagine →
    - Numero di frame

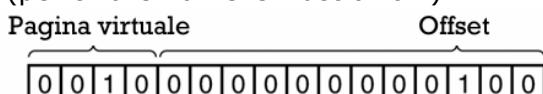


Funzionamento interno della MMU con 16 pagine da 4 kB.

Suggerimento da lezione seguente: ricorda l'esempio dello spostamento tra le mattonelle(pagine) quando si supera un certo valore, ad esempio da 1 a 999 pagina 1, con 1000 passo a pagina 2, da 1999 andando a 2000 passo a pagina 3 e così via... => 0010 (in decimale 2) (000000000100 in decimale 4) sto in mattonella 2 e nella posizione 4 di questa mattonella.

## EVOLUZIONE DEGLI INDIRIZZI E TABELLA DELLE PAGINE

Indirizzi nei nostri esempi: 16 bit (per chiarezza nelle illustrazioni)



Ma i PC moderni usano indirizzi a 32 o 64 bit.

Con 32 bit e pagine da 4 KB:

- 12 bit per indirizzare 4096 byte per pagina
- Tabella delle pagine di  $2^{(32-12)} = 2^{20} = 1.048.576$  voci! Una taglia di 4GB è «fattibile» anche per PC con «pochi» GB di RAM.

Indirizzi a 64 bit e pagine da 4 KB:

- Richiede  $2^{52}$  voci ( $\sim 4,5 \times 10^{15}$ ) nella tabella.
- In realtà nei sistemi a 64 bit si usano 48 bit (256 Terabyte bastano e avanzano... gli altri bit sono riservati per il futuro).

"Abbiamo visto l'astrazione della memoria negli ambienti unix e come fa il sistema operativo a virtualizzarla (divisione in pagine e assegnazione ai processi), questo avviene attraverso una unità attaccata al processore che fornisce l'indirizzo fisico.

- Questa mappatura della memoria è presente in memoria (tabelle/liste), nelle pagine precedenti abbiamo anche visto come avviene la mappatura degli indirizzi fisici in indirizzi virtuali."

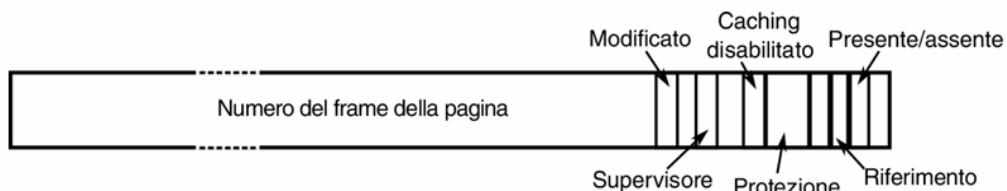
Ogni processo in una architettura a 32bit ha l'illusione di avere 4gb di ram disponibili.

In una page table i bit che non sono della pagina vengono usati per dare informazioni riguardo il la pagina, come ad esempio il bit che serve a indicare se la pagina virtuale è presente/assente in memoria, il bit di protezione etc...

### COME È COMPOSTA UNA VOCE DELLA PAGE TABLE?

Ogni voce ha informazioni cruciali come il numero del frame (es 12 bit per 4KB), 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.



Un bit importante è quello modificato e di riferimento che registrano l'uso della pagina, cioè va a indicare se la pagina è stata modificata e se non è stata modificata posso evitare di andare a riscrivere sul disco se già c'è.

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 – PROBLEMI CHIAVE

Mappatura Veloce: Necessaria a ogni riferimento alla memoria. Ogni istruzione può richiedere più riferimenti alla tabella delle pagine.

- Sfida: Se un'istruzione impiega 1 ns, la ricerca nella tabella delle pagine deve essere inferiore a 0,2ns per evitare colli di bottiglia.

Dimensione della Tabella delle Pagine:

- Contesto: Con 48 bit di indirizzamento e pagine di 4 KB, ci sono 64 miliardi di pagine. Una tabella delle pagine per questo spazio indirizzi richiederebbe voci enormi.
- Problema: Usare centinaia di gigabyte solo per la tabella delle pagine è impraticabile. Ogni processo richiede una propria tabella delle pagine.

Approcci alla soluzione:

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.

## PROBLEMI DELLA PAGINAZIONE

Problema di Prestazioni nella Paginazione:

- i. Ogni istruzione richiede l'accesso alla memoria per prelevare l'istruzione stessa e un ulteriore accesso per la tabella delle pagine.
- ii. Raddoppio degli accessi alla memoria riduce le prestazioni di metà.

Ma i programmi tendono a fare molti riferimenti a un piccolo numero di pagine, e inoltre solo una parte limitata delle voci della tabella delle pagine viene utilizzata frequentemente.

"La mappatura ha due problemi, un problema legato a dove mettiamo l'informazione e l'altro è la dimensione delle tabelle delle pagine che ci andiamo a mettere dentro (ne mettiamo tantissime). Una soluzione è mettere le tabelle delle pagine nei registri hardware oppure metterli in memoria, nessuna delle due è efficiente, il problema viene risolto parzialmente dal TLB (è un dispositivo hardware) che permette di mappare gli indirizzi virtuali in fisici senza passare dalla tabella delle pagine."

## TLB (INTRO, FUNZIONAMENTO E GESTIONE DEL TLB)

Translation Lookaside Buffer (TLB) è un dispositivo hardware che mappa indirizzi virtuali in fisici senza passare dalla tabella delle pagine. Esso riduce gli accessi alla memoria durante la paginazione.

Struttura: piccolo numero di voci (es. 8-256), ciascuno con numero di pagina virtuale, bit modificato, codice di protezione, e frame fisico.

Funzionamento:

1. Alla richiesta di un indirizzo virtuale, la MMU controlla prima nel TLB.
2. Se trovato e valido, il frame è prelevato direttamente dal TLB.
3. 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:

1. Le modifiche ai permessi di una pagina nella tabella delle pagine richiedono l'aggiornamento del TLB.
2. Per garantire la coerenza, la voce corrispondente nel TLB viene eliminata o aggiornata.

Per quanto riguarda la gestione software del TLB:

- TLB in architetture RISC: alcune macchine RISC gestiscono le voci del TLB tramite software
  1. Processi in caso di TLB Miss: un TLB miss non porta a una ricerca automatica nella tabella delle pagine da parte della MMU, si genera invece un errore di TLB e il sistema operativo deve intervenire. Il sistema operativo cerca la pagina, aggiorna il TLB, e riavvia l'istruzione.

Ricapitolando quindi: "quando viene chiesto un indirizzo virtuale la MMU come prima cosa controlla nel TLB, se viene trovato il frame della pagina allora viene prelevato direttamente dal TLB, se invece non viene trovato avviene una ricerca nella tabella delle pagine e la voce trovata prende posto di una voce nel TLB."

## TIPOLOGIE DI TBL-MISS E IMPLICAZIONI

Frequenza dei TLB Miss: i TLB miss sono comuni a causa del numero limitato di voci nel TLB (es. 64 voci), aumentare la dimensione del TLB è costoso e richiede compromessi nella progettazione dei chip.

Soft Miss vs Hard Miss:

- o Soft Miss: La pagina è in memoria ma non nel TLB. Richiede solo l'aggiornamento del TLB.
  - o Hard Miss: La pagina non è in memoria e richiede un accesso alla memoria non volatile (disco o SSD).
- Nota: Un hard miss è significativamente più lento di un soft miss.

Page TableWalk e Diverse Tipologie di Miss:

La ricerca nella gerarchia delle tabelle delle pagine è chiamata "page table walk".

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.

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

## PAGE TABLE SIZES

Abbiamo visto poco fa che «Con 32 bit si hanno pagine da 4 KB»:

- 12 bit per indirizzare 4096 byte per pagina
- tabella delle pagine di  $2^{(32-12)} = 2^{20} = 1.048.576$  voci! Fattibile per PC con GB di RAM.

Uno spazio di indirizzi virtuali molto grande porterebbe a una tabella di pagine molto grande, questo porta a spreco di memoria (senza contare cosa succederebbe per 64 bit!). Una possibile soluzione si trova nel concetto di Multi-level page table.

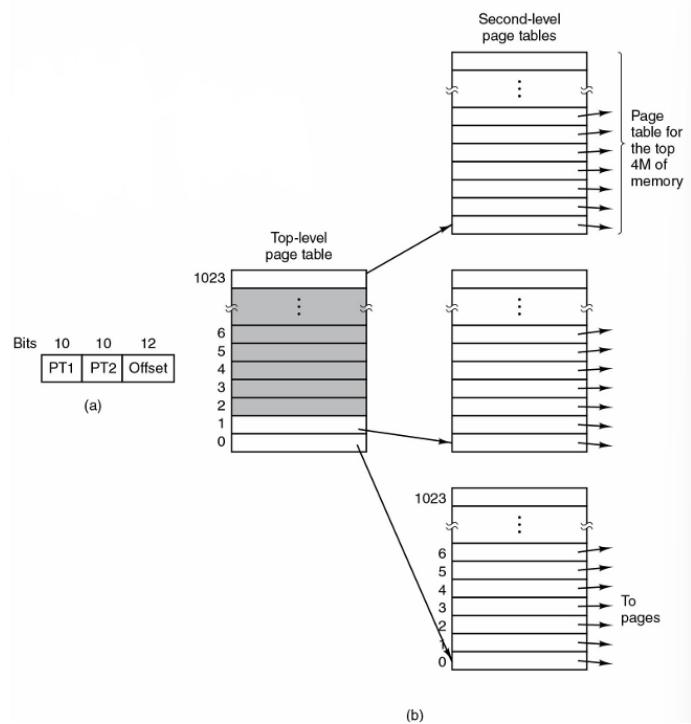
## PAGE TABLE A DUE LIVELLI (x86)

Le Page tables sono “attraversate” (“walked”) dal Memory Management Unit.

Il CR3 register è un registro speciale per puntare 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



## PAGE TABLE A 4 LIVELLI (64 BIT)

PGD: Page Global Directory

PUD: Page Upper Directory

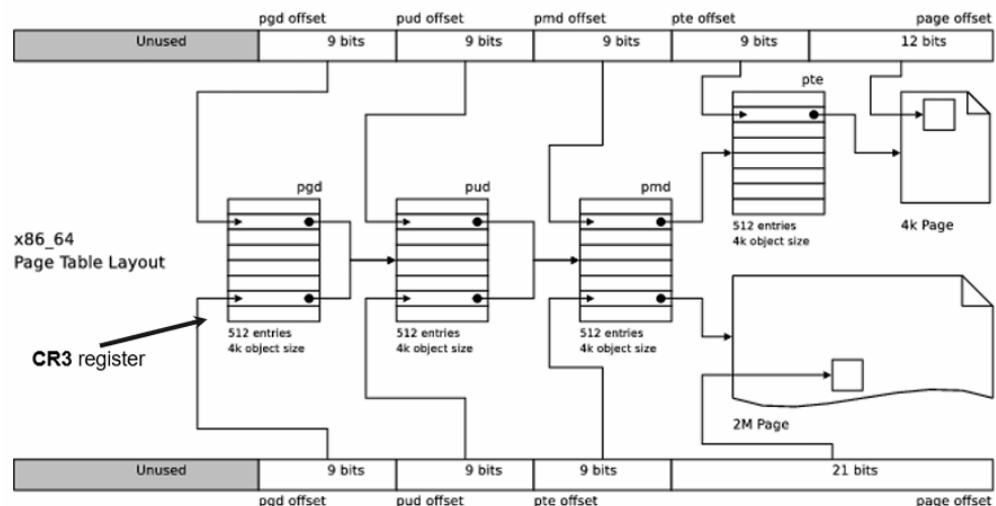
PMD: Page Mid-level Directory

PTE: page table entry

Nota:  $29 \times 29 \times 29 \times 29 \times 212 =$

248 byte. Ricordate i 48 bit?

Permettono di puntare per il momento, 256 TB di memoria...



## ALGORITMI DI SOSTITUZIONE DELLE PAGINE

Page replacement: Il computer potrebbe utilizzare più memoria virtuale di quanta ne abbia di fisica. La paginazione crea l'illusione di una memoria praticamente illimitata a disposizione dei processi utente.

- Quando una pagina logica non è in memoria (scambiata o swapped con un file/partizione), il sistema operativo deve caricarla in memoria in caso di page fault.
- Un'altra pagina logica potrebbe essere scambiata... Ma quale?

Alcuni algoritmi di sostituzione delle pagine sono:

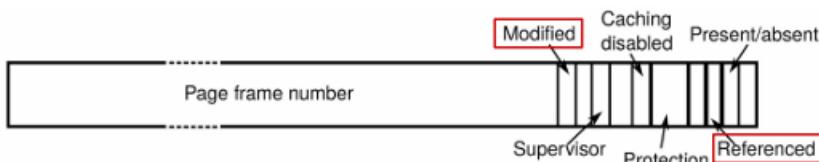
### 1) ALGORITMO OTTIMALE

- 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 realizzabile in pratica perché richiede la previsione del futuro utilizzo delle pagine. È però possibile l'implementazione su un simulatore per valutare le prestazioni rispetto agli algoritmi reali. Valutazione: Se un sistema ha prestazioni inferiori dell'1% rispetto all'ottimale, il miglioramento massimo teorico è dell'1%.

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

### BREVE RECAP UTILE:



Bit della Page Table Entry utili per gli algoritmi di sostituzione delle pagine:

- Modified (M): Impostato quando una pagina viene modificata (conosciuto anche come "dirty" bit)
- Referenced (R): Impostato quando la pagina viene acceduta (conosciuto anche come "accessed" bit)

### 2) NOT RECENTLY USED (NRU)

- 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, M segnala le modifiche.
- Aggiornamento Hardware: I bit vengono impostati dall'hardware a ogni accesso.
- Reset Periodico: Il bit R viene periodicamente ripulito per identificare pagine non recentemente usate (per esempio a ogni interrupt del clock)
- 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).
  - o Classe 0: Non referenziata, non modificata.
  - o Classe 1: Non referenziata, modificata.
  - o Classe 2: Referenziata, non modificata.
  - o 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:
  - o NRU rimuove una pagina casuale dalla classe più bassa non vuota.
  - o Azzerare R ma non M produce una pagina di classe 1: una pagina di classe 1 è stata modificata molto tempo fa e da allora non è stata più toccata.
- Vantaggi di NRU: Semplicità, efficienza implementativa e prestazioni accettabili

### 3) ALGORITMO FIFO(FIRST-IN, FIRST-OUT)

"FIFO è un algoritmo che elimina la pagina più vecchia in memoria, la pagina più vecchia potrebbe essere ancora una delle più utilizzate quindi non è così efficiente, ma ne esiste una variante migliore". Più nel dettaglio:

- Descrizione: FIFO è un algoritmo di paginazione che elimina la pagina più vecchia in memoria.
- Implementazione: Il sistema operativo rimuove la pagina in testa alla lista (la più vecchia) durante un page fault, aggiungendo la nuova pagina in coda.
- 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

### 4) SECOND-CHANCE ALGORITHM

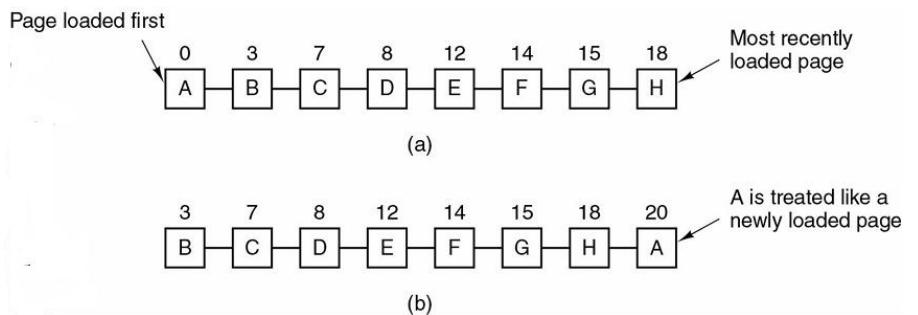
"Variante del FIFO, Controlla il bit R della pagina più vecchia per decidere la rimozione, se R=0 viene sostituita, se R=1 imposto R a 0 e la metto in fondo alla lista". Più nel dettaglio:

- Principio: Controllo del bit R (bit di lettura) della pagina più vecchia per decidere la rimozione.
- Funzionamento:
  - o Se R= 0: la pagina è vecchia e non usata di recente, quindi viene sostituita.
  - o Se R= 1: il bit R viene azzerato, la pagina è reinserita in fondo alla lista e considerata come appena caricata.

Nell'immagine abbiamo che:

- 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.

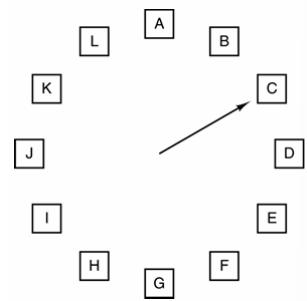


- Azioni:
  - o Se la pagina A ha R = 0, viene rimossa (scritta su memoria non volatile se modificata, altrimenti scartata).
  - o Se A ha R = 1, viene messa in fondo alla lista e il suo timestamp di caricamento aggiornato.
- Scenari Possibili:
  - o Se trova una pagina non referenziata, la rimuove.
  - o 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.

### 5) CLOCK ALGORITHM (algoritmo di clock per la sostituzione delle pagine)

"È una lista circolare (la pensiamo come un orologio, in realtà c'è un puntatore che scorre e poi l'ultimo puntatore punta al primo frame), se il bit R della pagina che punta è 0 viene rimossa e sostituita con la nuova, se R=1 il bit viene azzerato e il puntatore va alla successiva". Più nel dettaglio:

- Funzionamento: Lista circolare dei frame di pagina con un puntatore simile a una lancetta d'orologio per identificare la pagina più vecchia.
- Page Fault:
  - o Se il bit R della pagina puntata è 0, la pagina viene rimossa e sostituita con la nuova, poi il puntatore avanza.
  - o Se R= 1, il bit viene azzerato e il puntatore si sposta alla pagina successiva.
- Concetto: Ripete il processo finché non trova una pagina con R= 0
- Vantaggio: Elimina l'inefficienza della continua riallocazione delle pagine lungo la lista.
  - o Efficiente e più performante rispetto a Seconda Chance e FIFO.



## 6) LEAST RECENTLY USED (LRU) ALGORITHM

“Dal punto di vista teorico le pagine non usate di recente sono quelle che vogliamo sostituire, in pratica abbiamo una lista con le pagine più usate in cima e quelle meno usate alla fine, il riordinamento della lista però è inefficiente. Non usiamo quindi LRU ma utilizziamo NFU (not frequently used) che associa un contatore a ogni pagina, incrementato con ogni interrupt del clock in base al bit R. Per migliorare NFU si usa un algoritmo di nome Aging, quello che fa è emulare l’LRU dando meno peso agli usi passati (siccome NFU lo fa) e preferendo le pagine meno referenziate di recente”. Più nel dettaglio:

- Teoria:
  - o Fondamento LRU: Pagine non usate di recente sono candidate alla sostituzione.
  - o Possibile Implementazione: Lista delle pagine con quelle più usate in testa e quelle meno usate in coda.
  - o Aggiornamenti: Ogni riferimento richiede l’aggiornamento della lista (uno stack) e copia di pagine intere, operazione costosa anche con hardware dedicato.
- Sebbene tendente all’ottimo, praticamente non efficiente e non utilizzato. Esistono però altri metodi per implementare l’LRU con hardware speciale:
  - o Uso di un contatore a 64 bit per ogni riferimento a memoria.
  - o Selezione LRU: Alla generazione di un page fault, si rimuove la pagina con il contatore più basso, indicando l’uso meno recente

Simulazione software di LRU tramite algoritmo not frequently used:

- NFU(Not Frequently Used): Associa un contatore a ogni pagina, incrementato con ogni interrupt del clock in base al bit R.  
Tanti accessi ad una pagina => Alto valore di «frequenza» assegnato alla pagina => Minore possibilità di rimozione.
- Limite di 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.

Miglioramento di NFU tramite Aging:

- Si ha un numero di bit fisso (ad 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.

NFU e Aging in azione:

- Simula l’LRU via software  
es: Pagina 1
  - (a) NON è modificata ed ha valore 00000000
  - (b) viene modificata e diventa 10000000
  - (c) viene modificata e diventa 11000000
  - (d) NON è modificata e diventa 01100000
- Consideriamo le Pagine 3 e 5:
  - (c) Entrambe hanno avuto accesso
  - (d) e (e) Nessuna delle due ha avuto riferimenti
    - o Registrando un solo bit per intervallo di tempo non potremmo distinguere fra riferimenti in tempi recenti o meno
    - o 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)

Limiti e praticità dell’aging:

- 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.

## 7) WORKING SET ALGORITHM

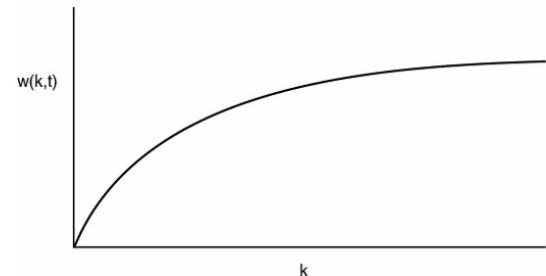
"Per working set intendiamo l'insieme delle pagine usate attualmente da un processo, più riferimenti faccio e più vado a toccare diverse pagine, una volta riferita deve essere inclusa nelle pagine che sto utilizzando (working set). I SO cercano di avere un modello basato sul Working set. L'algoritmo working set fa in modo che quando si verifica un page fault venga ricercata una pagina all'infuori del working set per rimuoverla". Più nel dettaglio:

Il concetto di working set

- Definizione di Working Set: insieme delle pagine attualmente usate da un processo. Rappresenta la località di riferimento, ovvero le pagine 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.

Concetto e dinamica del working set

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



Per quanto riguarda working set e performance

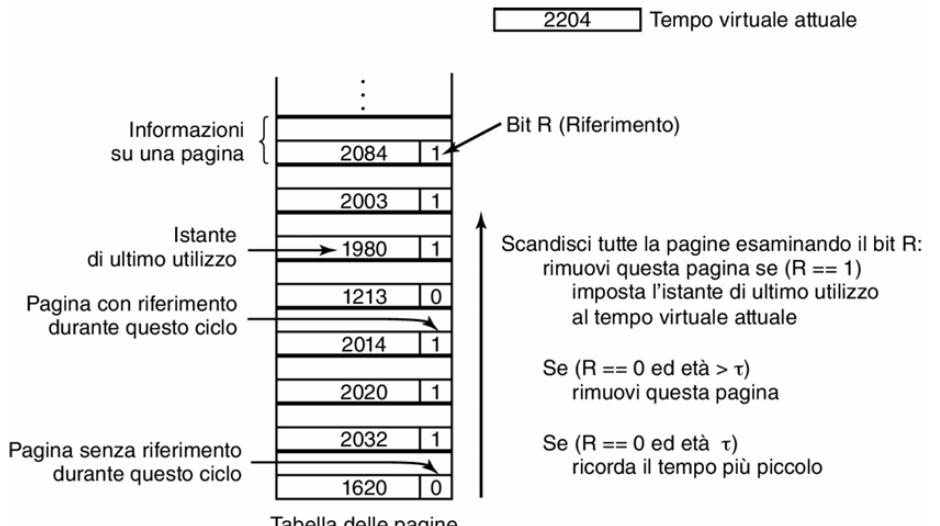
- Gestione della Memoria e Page Fault: se il working set di un processo è completamente in memoria, si verificano pochi page fault. Se il working set è più grande della memoria disponibile, si verificano frequenti page fault, rallentando significativamente il processo (fenomeno noto come thrashing).
- 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.

Implementazione e algoritmi di sostituzione

- Tracciamento del Working Set: il working set è definito come l'insieme delle pagine usate negli ultimi  $k$  riferimenti alla memoria. 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. Vengono utilizzate informazioni come il bit di riferimento e il tempo dell'ultimo utilizzo per determinare quali pagine rimuovere.

Esempio algoritmo working set

- Impostazione dei Bit R e M: un interrupt periodico azzera 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:
  - o  $R = 1$ : Aggiornamento del tempo dell'ultimo utilizzo, la pagina è nel working set.
  - o  $R = 0$  e  $\text{Età} > \tau$ : La pagina non è nel working set e viene rimossa.
  - o  $R = 0$  e  $\text{Età} \leq \tau$ : 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.

## 8) WS CLOCK ALGORITHM

"WS LOCK è un'evoluzione dell'algoritmo clock con alcuni miglioramenti dati da working set, in pratica ha una lista "circolare" (come quella del clock) e ogni frame ha il bit R, il bit M e il tempo dell'ultimo utilizzo". Più nel dettaglio:

Introduzione all'algoritmo wsclock

- Miglioramento dell'Algoritmo Working Set: WSClock è un'evoluzione dell'algoritmo Clock che integra informazioni del working set. È 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).

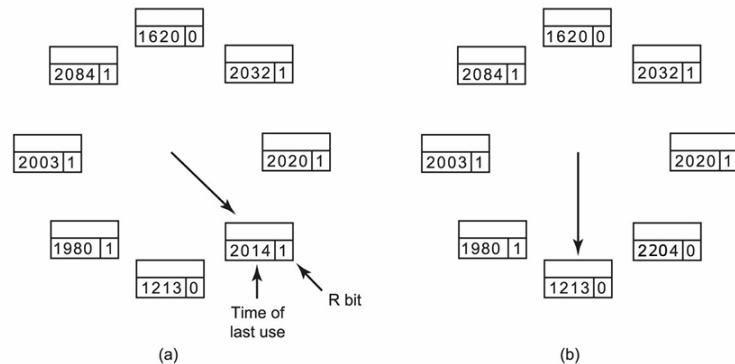
Esempio di wsclock:

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)

2204 Current virtual time



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

- o 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).
- o Se M = 1 invece la pagina è «sporca» (ovvero modificata) quindi 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. La lancetta avanza e l'algoritmo procede con la pagina successiva.

Gestione scritture e selezione pagina in wsclock

- Limitazione Scritture su Memoria Non Volatile:
  - o Possibilità di schedulare tutte le pagine per I/O su memoria non volatile in un ciclo di clock.
  - o Per ridurre il traffico su disco/SSD, si imposta un limite massimo di scritture (n pagine).
  - o Una volta raggiunto il limite n, ulteriori scritture non vengono schedulate.
- Comportamento al Completamento del Giro di Orologio:
  - o 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.
  - o 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.

## RIEPILOGO ALGORITMI DI SOSTITUZIONE DELLE PAGINE

Algoritmo	Commento
<b>Ottimale</b>	Non implementabile, ma utile come termine di confronto e valutazione
<b>LRU (Last Recently Used)</b>	Eccellente, ma difficile da implementare con precisione
<b>NRU (Not Recently Used)</b>	Approssimazione molto rozza dell'LRU
<b>FIFO (First-In, First Out)</b>	Potrebbe eliminare pagine importanti
<b>Seconda chance</b>	Deciso miglioramento rispetto al FIFO
<b>Clock</b>	Realistico
<b>NFU (Non Frequently Used)</b>	Approssimazione abbastanza rozza dell'LRU
<b>Aging</b>	Algoritmo efficiente che approssima bene l'LRU
<b>Working set</b>	Piuttosto dispendioso da implementare
<b>WSClock</b>	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.

-----Fine C)Algoritmi di sostituzione delle pagine-----

## LEZ10

### CONSIDERAZIONI NELLA PROGETTAZIONE DI SISTEMI DI PAGINAZIONE

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

Aspetti Cruciali:

1. 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.
2. Gestione dei Page Fault  
Monitoraggio della frequenza dei page fault per ottimizzare l'uso e allocazione della memoria e ridurre i tempi di attesa.
3. 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?»
4. 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.

### PROBLEMI DI PROGETTAZIONE

I problemi di progettazione più comuni sono:

#### 1) ALLOCAZIONE DI MEMORIA IN SISTEMI DI PAGINAZIONE: GLOBALE VS LOCALE

- Allocazione Locale:
  - o Ogni processo riceve una porzione fissa della memoria.
  - o Semplice da implementare, ma può portare a inefficienze se il set di lavoro del processo.
- Allocazione Globale
  - o Distribuzione dinamica della memoria tra i processi.
  - o 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
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

A0	A1	A2	A3	A4	A5	A6
B0	B1	B2	B3	B4	B5	B6
C0	C1	C2	C3			

(b)

A0	A1	A2	A3	A4	A5	A6
B0	B1	B2	B3	B4	B5	B6
C0	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)

Il processo A ha bisogno di allocare una pagina per A6

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

**Allocazione Globale:** è possibile rimuovere pagine di qualsiasi processo

### VANTAGGI DELL'ALLOCAZIONE GLOBALE DELLA MEMORIA

- a) 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.
- b) Limiti degli Algoritmi Locali: il thrashing può verificarsi con algoritmi locali se il set di lavoro di un processo cresce oltre la memoria allocata. La memoria può essere sprecata quando il set di lavoro di un processo si riduce e la memoria non viene riassegnata.
- c) Gestione Dinamica della Memoria: con l'allocazione globale, il sistema operativo deve dinamicamente assegnare e riassegnare frame ai processi. E' 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.
- d) 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.

## 2) STRATEGIE ALLOCAZIONE MEMORIA NEI SISTEMI DI PAGINAZIONE: EQUA VS PROPORZIONALE

- Allocazione Equa
  - o Distribuzione uniforme dei frame tra i processi.
  - o Esempio: 12.416 frame divisi equamente tra 10 processi risultano in 1.241 frame per processo.
  - o Svantaggi: Non tiene conto delle diverse esigenze di memoria tra processi di dimensioni varie.
- Allocazione Proporzionale
  - o Assegnazione di frame in base alla dimensione del processo.
  - o Rispecchia meglio le necessità di memoria, evitando allocazioni inadeguate.

Importanza del Limite Massimo di Pagine: assicurare che ogni processo abbia abbastanza pagine per eseguire le operazioni fondamentali, MA prevenire situazioni in cui processi con istruzioni che attraversano i limiti delle pagine non possano eseguire.

## 3) DINAMICA DI ALLOCAZIONE DELLE PAGINE

Gestione dinamica dei frame: inizio con un'allocazione proporzionale alla dimensione del processo e aggiornamento dinamico dell'allocazione in base all'evoluzione delle esigenze durante l'esecuzione.

### PAGE FAULT FREQUENCY (PFF)

È un algoritmo che si occupa del monitoraggio della frequenza dei page fault per regolare l'allocazione di memoria di un processo.

Come? Aumenta i frame se i page fault sono troppo frequenti, diminuisce invece se sono rari. Inoltre, non specifica quale pagina rimuovere, ma si focalizza sulla dimensione dell'allocazione.

### RELAZIONE TRA ALLOCAZIONE DI MEMORIA E PAGE FAULT

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.

Monitoraggio della Frequenza dei Page Fault: si contano i page fault per secondo e si utilizza una media mobile per tenere traccia delle fluttuazioni.

Nota: Alta frequenza di page fault indica necessità di più frame

Bassa frequenza di page fault suggerisce che il processo ha più memoria del necessario

### GESTIONE DEL TRASHING E CONTROLLO DEL CARICO DI MEMORIA

- 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:
  - o Out Of Memory Killer (OOM): 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.
  - o 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.

### SCHEDULING A DUE LIVELLI E TECNICHE DI RIDUZIONE DELLA MEMORIA

Scheduling a due livelli: alcuni processi sono in memoria non volatile e solo una parte è schedulata attivamente. Lo scheduling a due livelli aiuta a gestire 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 anche caratteristiche come la dimensione e/o frequenza di paginazione di processi e se si tratta di processi CPU bound oppure I/O bound.

Altre tecniche: oltre a uccidere o spostare processi si possono usare compattamento, compressione e deduplicazione(stesso page merging).

#### 4) POLICY DI PULIZIA

Contesto: 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 in memoria non volatile prima di caricarne di nuove.

È preferibile mantenere un buon numero di frame di pagina liberi piuttosto che occupare tutta la memoria e cercare frame liberi solo al bisogno.

PAGING DAEMON = è un processo in background usato dai sistemi di paginazione. È inattivo per la maggior parte del tempo e 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 in memoria non volatile (i contenuti delle pagine precedenti vengono conservati permettendo un eventuale rapido ripristino).

Implementazione con <<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;
- La lancetta posteriore si occupa della sostituzione delle pagine (maggiore probabilità di trovare pagine pulite grazie al lavoro del paging daemon).

#### 5) DIMENSIONE DELLE PAGINE E BILANCIO DEI FATTORI

Selezione Dimensione Pagine: i sistemi operativi possono selezionare la dimensione delle pagine (ad esempio, unendo due pagine da 4096 byte per formare una da 8 KB).

Vantaggi pagine piccole: riducono la frammentazione interna (spazio sprecato nelle pagine parzialmente vuote) e l'utilizzo di memoria.

Nota: un programma potrebbe richiedere meno memoria con pagine più piccole.

Svantaggi 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 DELLE PAGINE E THP

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

Pagine di diverse dimensioni: alcuni SO utilizzano pagine di diverse dimensioni per parti diverse del sistema (ad esempio pagine grandi per il kernel).

TRANSPARENT HUGE PAGES (THP) = Tecnica per utilizzare pagine di grandi dimensioni ottimizzando l'uso della memoria, spostando la memoria del processo per creare intervalli continui.

#### (\*)CALCOLO DELLA DIMENSIONE OTTIMALE DELLE PAGINE

Parametri considerati sono:  $s$  = dimensione media del processo in byte (esempio 1MB)

$p$  = dimensione della pagina in byte (da calcolare)

$e$  = dimensione di ogni voce nella tabella delle pagine in byte (esempio 4 o 8 byte)

Calcolo overhead:

- Numero di pagine per processo pari a circa  $s/p$ .
- Spazio occupato nella tabella delle pagine pari ad  $s \cdot e / p$  bytes.
- Memoria sprecata per frammentazione interna nell'ultima pagina pari a  $p/2$ .
  - o Fenomeno dell'ultima pagina: per qualsiasi processo, l'ultima pagina di memoria allocata potrebbe non essere completamente riempita.

Overhead totale =  $se/p + p/2$

- 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 Dimensione Ottimale delle Pagine:

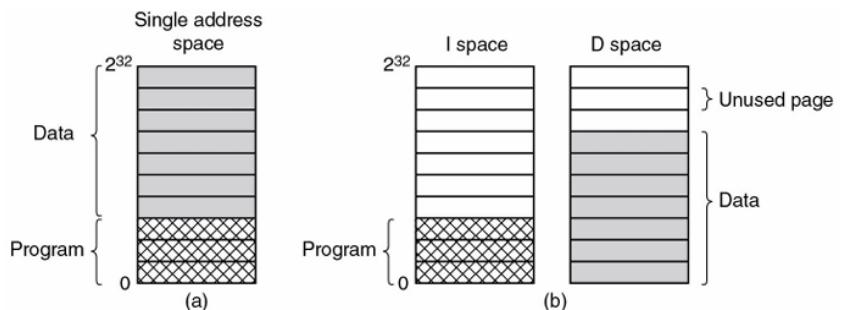
- Derivata della funzione di overhead rispetto a  $p$  uguagliata a zero:  $-se/p^{2+1/2} = 0$
- Dimensione ottimale delle pagine:  $p = \text{radiceQuadrata}(2se)$

## 6) ISTRUZIONI SEPARATE E SPAZI DEI DATI

Problema di progettazione: spazi separati per istruzioni e dati.

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 spazi Dati separati nelle cache, nei TLB, Cache L1 (dove lo spazio è poco si tende a separare le istruzioni più importanti dei dati)



## 7) PAGINE E LIBRERIE CONDIVISE

È comune che molti utenti eseguano lo stesso programma o utilizzino le stesse librerie, condividere pagine di memoria tra questi processi è più efficiente che mantenere copie separate.

Le pagine di sola lettura (come il testo dei programmi) possono essere condivise, le pagine dei dati invece generalmente no.

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

- I-space: istruzioni
- D-space: Dati

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

- Implementazione e scheduling: con ciascun processo che ha puntatori sia all'I-space che al D-space
- Lo scheduler utilizza questi puntatori per impostare l'MMU

### DIVERSI PROBLEMI: GESTIONE DELLE PAGINE CONDIVISE E COPY ON WRITE

Un problema con le pagine condivise è che la rimozione di un processo dalla memoria può causare numerosi page fault in un altro processo che condivide le stesse pagine. Inoltre un problema ulteriore è che è cruciale sapere se le pagine sono ancora in uso per evitare la loro liberazione accidentale.

Condivisione dei dati: più complessa rispetto alla condivisione del codice!

Ad 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 ed è estremamente efficiente per evitare la proliferazione di pagine.

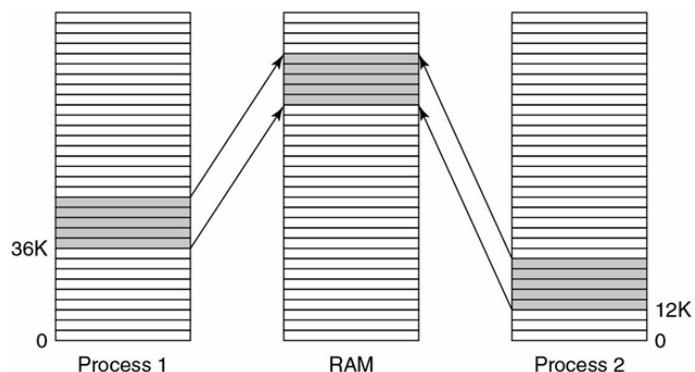
### LIBRERIE CONDIVISE – PRINCIPI E FUNZIONAMENTO

Condivisione su Ampia Scala: I SO condividono automaticamente tutte le pagine di testo di un programma avviato più volte. Per evitare problemi, meglio condividere le pagine in sola lettura.

Copy on Write per Dati: Se un processo modifica una pagina di dati condivisa, occorre applicare "copy on write".

Librerie condivise - Dynamic Link Libraries (DLLs): Usate per ridurre l'ingombro di grandi librerie comuni. (Vantaggi: Risparmio di 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.



Soluzione Compilativa: Le librerie condivise vengono compilate con indirizzi relativi anziché assoluti (le istruzioni usano offset relativi piuttosto che puntare a indirizzi specifici).

## 8) FILE MAPPATI IN MEMORIA

Concetto: I file mappati consentono a un processo di mappare un file all'interno del proprio spazio di indirizzi virtuali.

- Funzionamento: Alla mappatura, nessuna pagina viene caricata immediatamente (sono paginate su richiesta dalla memoria non volatile, man mano che vengono "toccate").
- Scrittura su File: Quando il processo termina o la mappatura è eliminata, tutte le pagine modificate vengono riscritte sul file.

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.

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

## DETtagli IMPLEMENTATIVI

### **→ PROBLEMI DI IMPLEMENTAZIONE DELLA MEMORIA VIRTUALE**

Sfide nell'implementazione:

- scelta tra algoritmi teorici (esempio Second Chance, Aging etc...) e pratiche operative (allocazione locale/globale, paginazione a richiesta/prepaginazione)
- gestione di problemi pratici di implementazione della memoria virtuale

Attività del Sistema Operativo nella Paginazione:

- Creazione del Processo:
  - o Determinare le dimensioni iniziali del programma e dei dati.
  - o Creare e inizializzare la tabella delle pagine.
  - o Allocare spazio nella memoria non volatile per lo scambio.
  - o Inizializzare l'area di scambio e registrare informazioni nella tabella dei processi.
- Esecuzione del Processo:
  - o Azzerrare la MMU e, se necessario, svuotare il TLB.
  - o Rendere attiva la tabella delle pagine del processo.
  - o Pre-paginazione: Facoltativamente, caricare alcune pagine in memoria per ridurre i page fault iniziali.

### **→ GESTIONE DEI PAGE FAULT E CHIUSURA DEL PROCESSO**

- Gestione dei Page Fault:
  - o Determinare l'indirizzo virtuale che ha causato il fault.
  - o Trovare la pagina necessaria nella memoria non volatile.
  - o Scegliere un frame disponibile, eventualmente sfrattando pagine vecchie.
  - o Caricare la pagina nel frame e ripristinare il contatore del programma.
- Chiusura del Processo:
  - o Rilasciare la tabella delle pagine, le pagine in memoria e lo spazio su disco/SSD.
  - o Gestire le pagine condivise con altri processi, rilasciandole solo dopo l'ultimo utilizzo

## → PAGE FAULT IN 10 PASSI

### A. Inizio della sequenza

1. Trap nel Kernel da parte dell'Hardware:
  - 1.1 L'hardware esegue una trap nel kernel, salvando il contatore del programma nello stack.
  - 1.2 Informazioni sull'istruzione corrente salvate nei registri speciali della CPU.
2. Avvio Routine di Servizio Interrupt:
  - 2.1 Viene eseguita una routine in assembly per salvare i registri e altre informazioni
  - 2.2 Invocazione del gestore dei page fault.
3. Identificazione della Pagina Virtuale Necessaria:
  - 3.1 Il sistema operativo determina quale pagina virtuale manca.
  - 3.2 Se non disponibile dai registri hardware, recupero e analisi dell'istruzione dal contatore di programma

### B. Gestione e risoluzione

4. Verifica Validità Indirizzo e Protezione:
  - 4.1 Controllo della validità dell'indirizzo e coerenza della protezione con l'accesso.
  - 4.2 Se invalide, invio di un segnale di errore o terminazione del processo.
5. Rilascio di un Frame Libero:
  - 5.1 Se non ci sono frame liberi, esecuzione di un algoritmo di sostituzione delle pagine.
  - 5.2 Se la pagina è "sporca", viene schedulata per la scrittura in memoria non volatile e il processo è sospeso.
6. Caricamento della Pagina Richiesta:
  - 6.1 Una volta liberato (o scritto in memoria non volatile), il frame viene usato per caricare la pagina necessaria da disco o SSD.
  - 6.2 Durante il caricamento della pagina, il processo in page fault è ancora sospeso e viene eseguito, se disponibile, un altro processo utente.

### C. Conclusione e ripresa

7. Aggiornamento delle Tabelle delle Pagine:
  - 7.1 Al completamento del trasferimento dal supporto non volatile, le tabelle delle pagine vengono aggiornate per riflettere la nuova posizione della pagina.
  - 7.2 Il frame viene contrassegnato come disponibile.
8. Ripristino dell'Istruzione in Errore:
  - 8.1 L'istruzione in errore è riportata allo stato che aveva all'inizio
  - 8.2 Il contatore di programma è ripristinato in modo da puntare a quell'istruzione.
9. Ripresa del Processo in Errore:
  - 9.1 Il processo precedentemente in errore viene schedulato per l'esecuzione.
  - 9.2 Ritorno alla routine in assembly che lo aveva interrotto.
10. Ricarica dei Registri e Ritorno allo Spazio Utente:
  - 10.1 La routine di servizio ricarica i registri e le informazioni di stato.
  - 10.2 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 = Pinning delle Pagine: 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 = Gestione I/O nei Buffer del Kernel: 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.

## → MEMORIA SECONDARIA E GESTIONE DELLO SCAMBIO

"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)
  - o Il sistema operativo prevede una partizione speciale o dispositivo separato per lo scambio, come nei sistemi UNIX.
  - o Un'area del disco/SSD strutturata in maniera differente dal file system usato per memorizzare file e cartelle (vedi lezioni successive)
  - o Partizione di scambio con file system semplificato, utilizzando numeri di blocchi relativi
- Allocazione in Memoria di Scambio:
  - o All'avvio, allocazione di spazio in partizione di scambio pari alla dimensione del processo.
  - o Gestione come lista di parti libere (anche se esistono miglioramenti, vedi Capitolo 10 del libro).
- Associazione Processo-Area di Scambio:
  - o Ogni processo ha un'area di scambio in memoria non volatile.
  - o L'indirizzo in cui scrivere una pagina è calcolato sommando l'offset della pagina al suo spazio virtuale all'inizio dell'area di scambio.

## → STRATEGIE DI PAGINAZIONE E OTTIMIZZAZIONI

Gestione di Crescita dei Processi: Riserva di aree separate per testo, dati e stack, per gestire l'espansione dei processi.

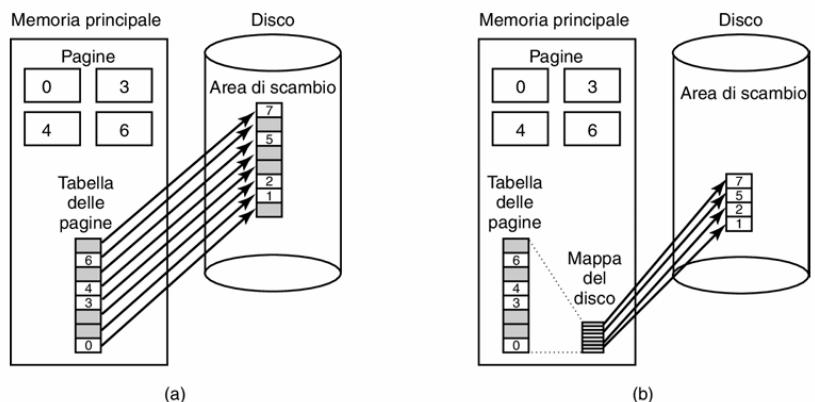
Alternativa di Allocazione Dinamica:

- Allocazione dello spazio su disco/SSD al momento dello scambio di ogni pagina.
- Tavola per ogni processo che indica la posizione di ogni pagina in memoria non volatile.

Esempi di Gestione Paginazione:

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

Ottimizzazioni su File System: Uso di file pre-allocati in file system normale (es. Windows).

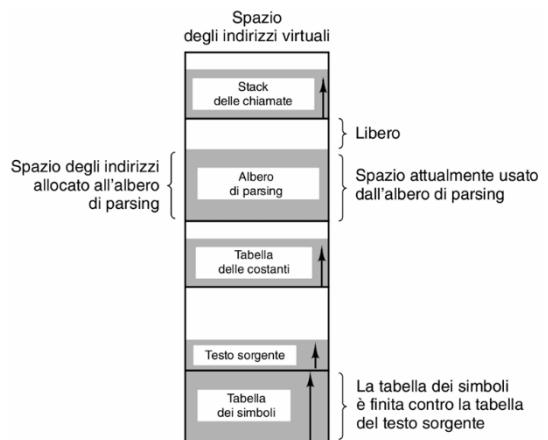


## LA SEGMENTAZIONE

### → MEMORIA MONODIMENSIONALE VS. SEGMENTAZIONE

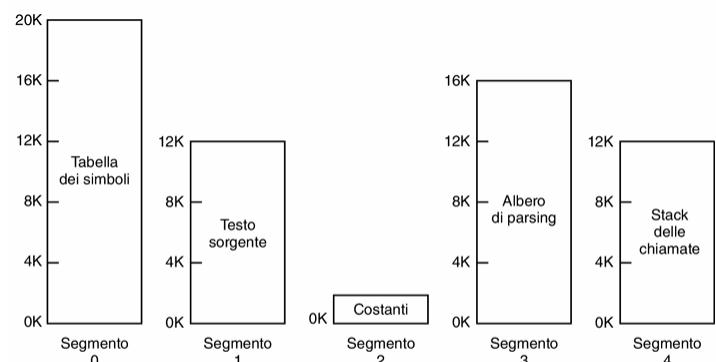
#### Memoria monodimensionale:

- In un sistema di memoria monodimensionale, gli indirizzi virtuali vanno da 0 a un massimo e sono disposti in modo lineare e contiguo.
- Può risultare problematica in alcuni scenari, come nella compilazione perché diverse tabelle 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).



#### Segmentazione:

- 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
  - indirizzo nel segmento.



Una memoria segmentata consente a ogni tabella di crescere o di ridursi indipendentemente dalle altre tabelle.

### → VANTAGGI DELLA SEGMENTAZIONE

#### I vantaggi della segmentazione sono:

1. Flessibilità: I segmenti possono crescere o ridursi in modo indipendente l'uno dall'altro. Esempio, lo stack del compilatore può espandersi o contrarsi senza influenzare le altre tabelle. (Ciò elimina il problema di collisione presente nella memoria monodimensionale).
2. 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.
3. 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.

### → MEMORIA SEGMENTATA VS PAGINATA

Confronto: la segmentazione suddivide la memoria in segmenti con indirizzi pienari, mentre la paginazione divide la memoria in pagine di dimensioni fisse.

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

#### Esempi Pratici

1. Uso nel Compilatore: Le varie tabelle utilizzate durante la compilazione possono essere allocate in segmenti separati. Le tabelle possono crescere indipendentemente e di essere gestite più efficacemente.
2. 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

Considerazione	Paginazione	Segmentazione
Il programmatore deve sapere che questa tecnica è in uso?	NO	SI
Quanti spazi di indirizzi lineari ci sono?	1	Molti
Lo spazio degli indirizzi totale può superare la dimensione della memoria fisica?	SI	SI
Le procedure e i dati possono essere distinti e protetti separatamente?	NO	SI
Le tabelle la cui dimensione varia possono essere disposte facilmente?	NO	SI
La condivisione delle procedure fra utenti è facilitata?	NO	SI
Perché fu inventata questa tecnica?	Per avere uno spazio degli indirizzi lineare grande senza dover acquistare ulteriore memoria fisica	Per consentire a programmi e dati di essere spezzati in spazi degli indirizzi logicamente indipendenti e per facilitare la condivisione e la protezione

## → IMPLEMENTAZIONE DELLA SEGMENTAZIONE PURA: LE SFIDE

Paginazione VS Segmentazione: Le pagine hanno dimensioni fisse, i segmenti no.

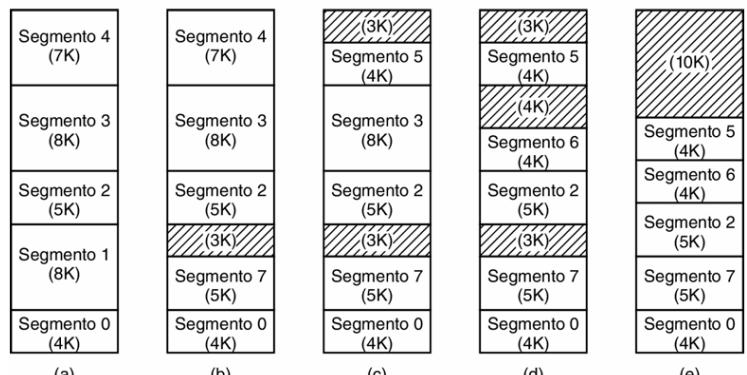
Evoluzione della Configurazione della Memoria:

- memoria fisica con cinque segmenti.
- 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].
- il segmento 4 è sostituito dal segmento 5
- il segmento 3 è rimpiazzato dal segmento 6

Frammentazione Esterna

("Checkerboarding"): Dopo un po', la memoria sarà suddivisa in parti, qualcuna contenente segmenti e altri spazi vuoti.

- può essere risolto tramite la compattazione



## IL COMANDO FREE IN LINUX PER MONITORARE LA MEMORIA

Funzione del Comando free: Fornisce dettagli sull'utilizzo della memoria fisica e dello swap nel sistema Linux.

Output del Comando:

- 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/slab, recuperabile se necessario.
- available: Stima della memoria disponibile per nuove applicazioni, considerando buffer e cache.

Per quanto riguarda opzioni e interpretazione dell'output di free:

- Formato Leggibile: Utilizzo dell'opzione -h per visualizzare i dati in formato megabyte o gigabyte.
- Specifica delle Unità di Misura: Opzioni come -b, --kilo, --mega, --giga per specificare l'unità di misura (byte, kilobyte, megabyte, gigabyte).
- Visualizzazione dei Totali: Opzione -t per mostrare il totale della memoria e dello swap.
- Aggiornamento Continuo: Opzione -s per aggiornamenti continui ogni tot secondi, simile al comando watch.

Nota: Il comando free è essenziale per comprendere come la memoria viene utilizzata nel sistema, identificando potenziali spazi liberi per nuove applicazioni e monitorando l'efficienza del sistema in termini di gestione della memoria.

## LEZ11

### TEMA LEZIONE = "IL FILE SYSTEM: GESTIONE FILE"

#### PROBLEMI E REQUISITI DI MEMORIZZAZIONE NEL COMPUTING

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.
- Requisiti Essenziali per la Memorizzazione a Lungo Termine:
  1. Capacità di salvare grandi quantità di informazioni.
  2. Persistenza delle informazioni oltre la vita del processo che le utilizza.
  3. Accessibilità delle informazioni da più processi simultaneamente.

#### FILE SYSTEMS

Pensate a un disco come a una sequenza lineare di blocchi di dimensioni fisse che supporta due operazioni: Leggere il blocco k , Scrivere il blocco k.

Domande che sorgono rapidamente: "Come si trovano le informazioni?" "Come si impedisce a un utente di leggere i dati di un altro utente?" "Come si fa a sapere quali blocchi sono liberi?" ...

#### SOLUZIONI DI MEMORIZZAZIONE E RUOLO DEI FILE SYSTEMS

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 operazioni essenziali come la lettura e la scrittura di blocchi di dati.

File System e Gestione delle Informazioni:

- 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.

Ruolo dei Sistemi Operativi: I sistemi operativi gestiscono i file attraverso il file system, che si occupa di:

- struttura , denominazione , accesso , protezione , implementazione dei file.

Interfaccia Utente VS Implementazione Tecnica:

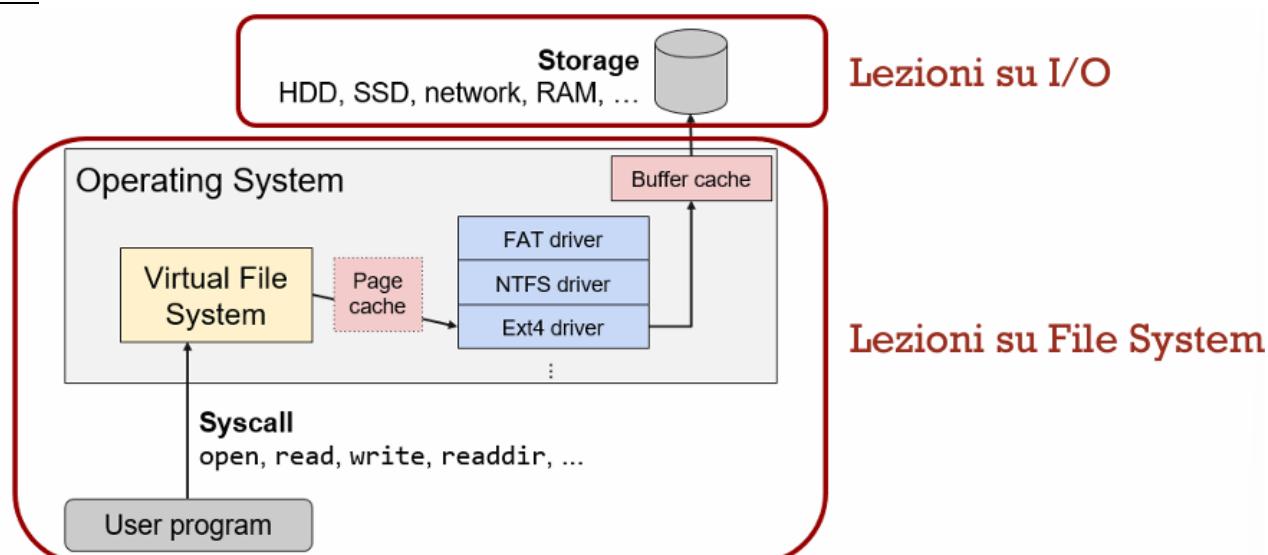
- Aspetti visibili all'utente (nomi dei file, operazioni consentite)
- Aspetti tecnici rilevanti per i progettisti del sistema (gestione della memoria, struttura interna del file system).

#### COSA SONO I FILE SYSTEM?

I file system sono un modo per organizzare e memorizzare (in modo persistente) le informazioni.

I file system sono anche un'astrazione sui dispositivi di memorizzazione: Disco rigido, SSD, rete, RAM, ... organizzati in file e (tipicamente) directory.

#### ROADMAP



## I FILE

### → CARATTERISTICHE DEI NOMI DEI FILE

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 (Esempi comuni includono nomi con lettere, numeri e caratteri speciali).

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.

Evoluzione dei File System: Vari file system sono stati sviluppati nel corso del tempo, tra cui FAT-16, FAT-32 e NTFS. Questi sistemi variano in termini di proprietà come la costruzione dei nomi dei file e il supporto per Unicode.

Caratteri speciali nei nomi dei file:

- FAT12: Non puoi usare nomi con "\*|<>?\\" e altro
- Ext4: No 10' e ", o i nomi speciali "." e "..".

Case sensitivity: Sistemi come UNIX distinguono tra maiuscole e minuscole, a differenza di MS-DOS.

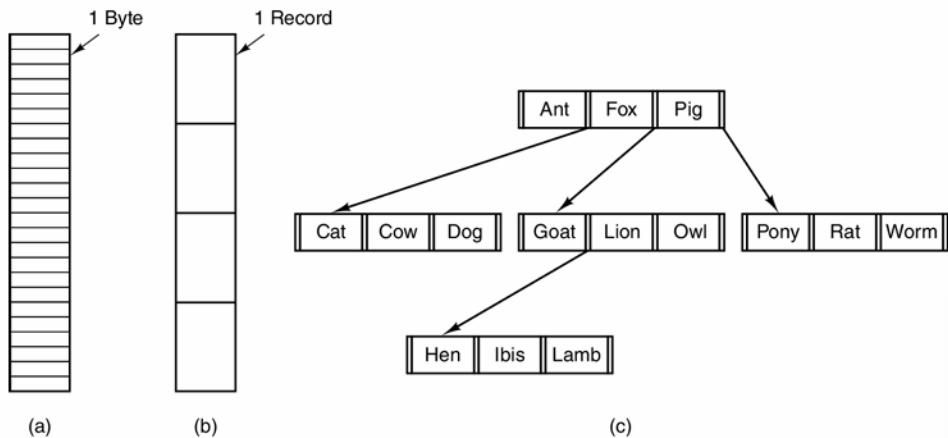
### → ESTENSIONE DEI FILE E LORO SIGNIFICATO

- Estensioni di File: Le estensioni sono parti di nomi di file che seguono un punto, indicando 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 e sono associati a programmi specifici.
- Gestione delle Estensioni in Windows: In Windows, 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

### → ALCUNE ESTENSIONI TIPICHE DI FILE

.bak	File di backup
.c	Programma sorgente in linguaggio C
.gif	Immagine in CompuserveGraphicalInterchangeFormat
.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 E LORO STRUTTURE

- **File e Directory Normali:**
  - Sistemi Operativi: 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.
- **Tipi di File Normali:**
  - File ASCII: Composti da righe di testo, visualizzabili e stampabili; variano nella terminazione delle righe (carattere di "a capo" o "nuova riga").
  - File Binari: Non leggibili come testo; hanno una struttura interna conosciuta dai programmi che li utilizzano. Esempi includono file eseguibili e archivi.

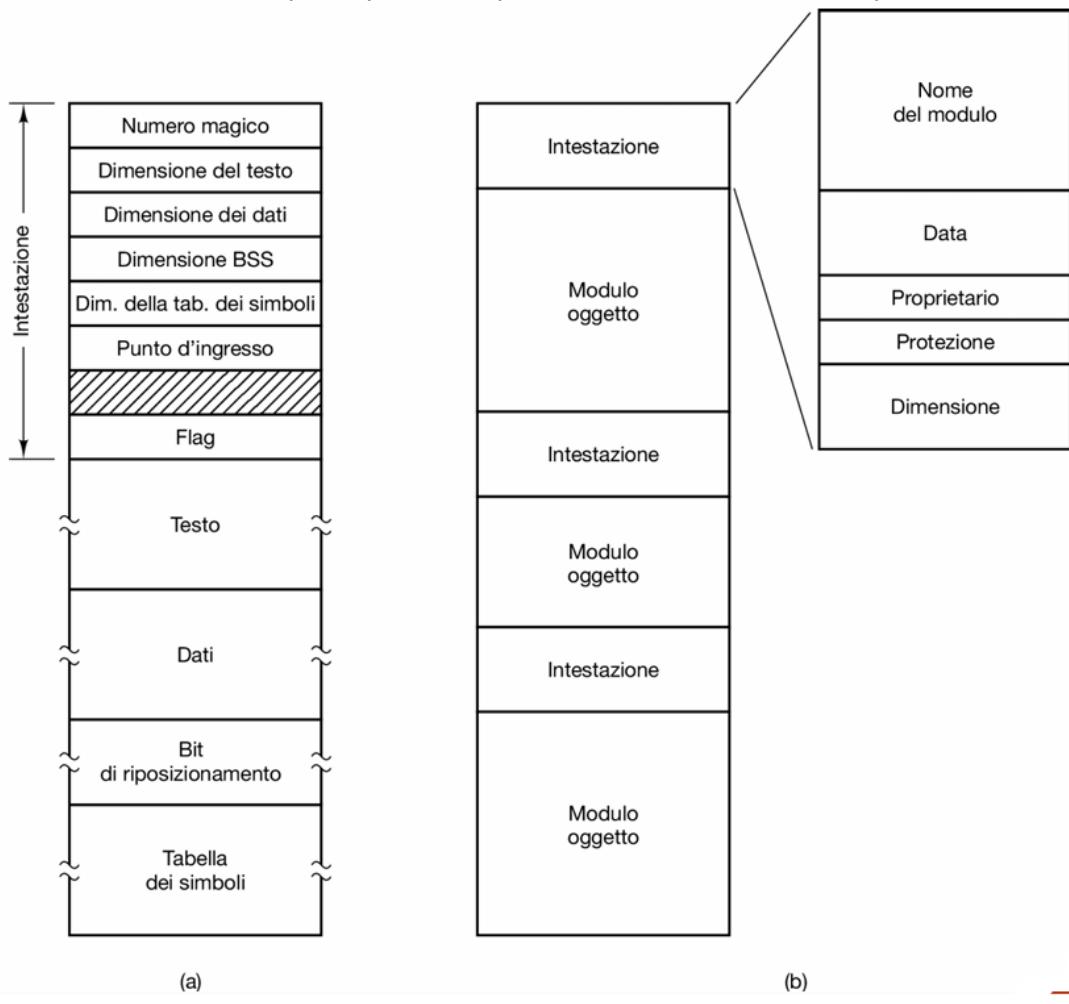
## → FILE E STRUTTURE INTERNE

### a) File Eseguibile :

- Esempio da una delle prime versioni di UNIX.
- Sequenza di byte con formato specifico per l'esecuzione.
- Componenti:
  - Intestazione (Header): Contiene un 'numero magico' per identificare il file come eseguibile (e 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.



"(b) ha a che fare con tar.gz che vedremo più avanti"

## → RICONOSCIMENTO E GESTIONE DEI TIPI DI FILE

### 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.
- Strumenti di Esame dei File:
  - Utility in UNIX: L'utility *file* in UNIX usa euristiche per determinare il tipo di file (testo, directory, eseguibile, ecc.)

## → METODI DI ACCESSO AI FILE

Evoluzione dell'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.

Importanza dell'Accesso Casuale: Cruciale per applicazioni come i sistemi di database, dove è necessario accedere rapidamente a record specifici senza attraversare l'intero file.

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.
- Implementazione: Questo metodo di accesso è adottato sia in UNIX sia in Windows

## → 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.
  - 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, per il controllo dell'accesso, e per 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 casuale	0 per accesso sequenziale; 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.
  - o 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.
  - o facilita l'accesso rapido in seguito
4. *Close*: Chiusura del file al termine degli accessi per liberare spazio nelle tabelle interne .
  - o forza anche la scrittura dell'ultimo blocco del file
5. *Read*: Lettura dei dati da un file, generalmente dalla posizione corrente.
  - o 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
  - o può comportare l'ampliamento del file o la sovrascrittura dei dati esistenti.
7. *Append*: Aggiunta di dati solo alla fine del file
  - o 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.
9. *GetAttributes*: Lettura degli attributi di un file
  - o 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
  - o come la modalità di protezione o altri flag, dopo la creazione del file.
11. *Rename*: Ridenominazione di un file
  - o utilizzata come alternativa al processo di copia ed eliminazione del file originale, specialmente utile per file di grandi dimensioni.

## → SCRIVERE E LEGGERE DATI IN FILE UTILIZZANDO FUNZIONI POSIX [11\_write\_and\_read\_POSIX.c]

```
/*
 * Questo programma dimostra come scrivere e leggere dati in un file utilizzando funzioni POSIX.
 * Utilizza sia formati binari che leggibili (testuali) per illustrare le differenze tra i due metodi.
 *
 * ***Sezioni principali:**
 * 1. **Scrittura nel file (`foo.txt`):**
 *   - Scrive un intero in formato binario.
 *   - Scrive un intero in formato leggibile (testuale).
 *   - Scrive un float in formato binario.
 *   - Scrive un float in formato leggibile.
 *   - Scrive una stringa.
 *   - Scrive una stringa combinata con un numero formattato leggibile.
 * 2. **Lettura dal file (`foo.txt`):**
 *   - Legge un intero scritto in formato binario.
 *   - Legge un intero scritto in formato leggibile.
 *   - Legge un float scritto in formato binario.
 *   - Legge un float scritto in formato leggibile.
 *   - Legge il resto del file, una riga alla volta, fino alla fine.
 * 3. **Funzione `read_line`:**
 *   - Implementa una lettura di una riga alla volta da un file in stile POSIX.
 *   - Questa funzione non è nativamente disponibile in POSIX, ma permette di leggere
 *     una riga fino al carattere '\n' o EOF.
 *
 * ***Note:**
 * - Il file è denominato "foo.txt". Viene creato se non esiste e viene sovrascritto se già esiste.
 * - Per le letture leggibili, il codice tiene traccia del numero di byte scritti per interpretare correttamente i dati.
 * - La lettura in formato binario richiede che il programma conosca il tipo e la dimensione dei dati.
 *
 * ***Vantaggi e svantaggi:**
 * - **Binario:** 
 *   - Pro: Più efficiente in termini di spazio.
 *   - Contro: Non leggibile dall'utente e dipendente dalla piattaforma (endianness, dimensioni dei tipi).
 * - **Leggibile:** 
 *   - Pro: Portatile e facile da leggere e modificare.
 *   - Contro: Richiede più spazio e operazioni di conversione.
 *
 * ***Utilizzo:**
 * - Compilare con un compilatore compatibile con POSIX, ad esempio `gcc`.
 * - Eseguire il programma, che creerà il file 'foo.txt' e stamperà i risultati della lettura sulla console.
 */
```

```

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FILENAME "foo.txt" // Nome del file da utilizzare
#define BUFFER_SIZE 256

// Funzione per leggere una riga intera da un file. In POSIX "non c'è" :-(

ssize_t read_line(int file_fd, char *buffer, size_t max_length) {
    char ch;
    size_t count = 0;

    // Leggi un carattere alla volta fino a trovare '\n' o EOF
    while (count < max_length - 1) {
        ssize_t bytes_read = read(file_fd, &ch, 1); // Legge un carattere
        if (bytes_read == 0) {
            // Fine del file
            break;
        } else if (bytes_read < 0) {
            // Errore nella lettura
            perror("Errore nella lettura del file");
            return -1;
        }

        // Aggiungi il carattere al buffer
        buffer[count++] = ch;

        // Termina la lettura se troviamo '\n'
        if (ch == '\n') {
            break;
        }
    }

    // Null-terminate la stringa
    buffer[count] = '\0';

    // Restituisce il numero di byte letti
    return count;
}

int main() {
    int file_fd;
    char buffer[BUFFER_SIZE];

    //-----HOW TO WRITE-----
    // 1. Aprire il file foo.txt e, se non esiste, crearlo.
    file_fd = open(FILENAME, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (file_fd < 0) {
        perror("Errore nell'aprire il file");
        return 1;
    }

    // 2. Scrive un intero in formato binario.
    int my_int_bin = 42; // Interi da scrivere in formato binario
    write(file_fd, &my_int_bin, sizeof(my_int_bin)); // Scrittura binaria

    // 3. Scrive un intero in formato leggibile.
    int my_int_text = 43; // Interi da scrivere in formato leggibile
    sprintf(buffer, "%d\n", my_int_text);
    int my_int_string_len = strlen(buffer);
    write(file_fd, buffer, my_int_string_len); // Scrittura testuale

    // 4. Scrive un float in formato binario.
    float my_float_bin = 3.14f; // Float da scrivere in formato binario
    write(file_fd, &my_float_bin, sizeof(my_float_bin)); // Scrittura binaria

    // 5. Scrive un float in formato leggibile.
    float my_float_text = 2.71f; // Float da scrivere in formato leggibile
    sprintf(buffer, "%.2f\n", my_float_text);
    int my_float_string_len = strlen(buffer);
    write(file_fd, buffer, my_float_string_len); // Scrittura testuale

    // 6. Scrive una stringa in formato leggibile.
    const char *my_string_text = "Hello Text"; // Stringa da scrivere
    sprintf(buffer, "%s\n", my_string_text);
    write(file_fd, buffer, strlen(buffer)); // Scrittura testuale

    // 7. Scrive una stringa più un numero in formato leggibile.
    const char *message = "My favorite number is"; // Messaggio da scrivere
    int favorite_number = 7; // Numero associato al messaggio
    // Formatta il messaggio e il numero in una stringa leggibile
    sprintf(buffer, "%s %d\n", message, favorite_number);
    // Scrive il contenuto formattato nel file
    write(file_fd, buffer, strlen(buffer));
}

```

```

// 8. Chiude il file dopo la scrittura.
close(file_fd);

-----HOW TO READ-----

// 9. Riapre il file per la lettura.
file_fd = open(FILENAME, O_RDONLY);
if (file_fd < 0) {
    perror("Errore nell'aprire il file per lettura");
    return 1;
}

// 10. Legge un intero in formato binario.
int read_int_bin;
read(file_fd, &read_int_bin, sizeof(read_int_bin)); // Lettura binaria
printf("Intero letto (binario): %d\n", read_int_bin);

// 11. Legge un intero in formato leggibile. Ma devo ricordare quanti byte ho scritto prima!
read(file_fd, buffer, my_int_string_len);
int read_int_text = atoi(buffer); // Conversione da stringa a intero
printf("Intero letto (testuale): %d\n", read_int_text);

// 12. Legge un float in formato binario.
float read_float_bin;
read(file_fd, &read_float_bin, sizeof(read_float_bin)); // Lettura binaria
printf("Float letto (binario): %.2f\n", read_float_bin);

// 13. Legge un float in formato leggibile. Ancora, devo ricordare quanti byte ho scritto prima!
read(file_fd, buffer, my_float_string_len);
float read_float_text = atof(buffer); // Conversione da stringa a float
printf("Float letto (testuale): %.2f\n", read_float_text);

// 14. Legge il resto del file, una riga alla volta fino alla fine del file
while (read_line(file_fd, buffer, BUFFER_SIZE) > 0) {
    printf("Resto del file. Riga letta: %s", buffer);
}

// 15. Chiude il file.
close(file_fd);

return 0;
}

```

## → OPERAZIONI SU FILE IN UNIX

### 1) Aprire e leggere un file:

```

int fd = open("foo.txt", O_RDONLY); // O_RDONLY=modalità con cui apro file
char buf[512];
ssize_t bytes_read = read(fd, buf, 512);
close(fd);
printf("read %zd: %s\n", bytes_read, buf);

```

NOTA: La funzione read restituisce il numero di byte che sono stati letti!

L'apertura di un file restituisce un handle (descrittore di file) per le operazioni future.

Qualsiasi funzione può restituire un errore, ad esempio ENOENT(File does not exist) oppure EBADF(Bad file descriptor).

### 2) Posizionamento (seek) nei file :

```

int fd = open("foo.txt", O_RDONLY);
lseek(fd, 128, SEEK_CUR);
char buf[8];
read(fd, buf, 8);
close(fd);

```

Sposta la posizione corrente nel file in avanti o indietro:

⇒ Sposta il puntatore del file di 128 byte in avanti dalla posizione corrente nel file.

L'uso di SEEK\_CUR indica che il movimento è relativo alla posizione corrente del puntatore nel file.

⇒ Dichiara un buffer buf di dimensione 8 byte e poi legge 8 byte dal file, a partire dalla nuova posizione (che è 128 byte oltre la posizione corrente a causa del precedente lseek), e li mette nel buffer buf.

3) Apre un file e scrive:

```
int fd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC);
char buf[] = "Hi there";
write(fd, buf, strlen(buf));
close(fd);
```

O\_CREAT e O\_TRUNC sono flag che controllano il comportamento dell'apertura del file:

- ❖ O\_WRONLY: Apre il file in modalità scrittura.
- ❖ O\_CREAT: Crea il file se non esiste.
- ❖ O\_TRUNC: Se il file esiste, ne tronca il contenuto a dimensione 0. sovrascriverà il contenuto esistente di foo.txt con "Hi there", o creerà un nuovo file con questo contenuto se foo.txt non esiste.

4) Rimuovere file:

```
unlink("foo.txt")
```

5) Rinominare file:

```
rename("foo.txt","bar.txt");
```

6) Cambiare i file permission attribute:

```
chmod("foo.txt",0755);
```

7) Cambiare la proprietà del file:

```
chown("foo.txt",uid,gid);
```

#### → PROGRAMMA PER COPIARE FILE [11\_COPYFILE.C]

```
/* Programma di copia di file con controllo errori minimale e reportistica. */
#include <sys/types.h> // Include i file header necessari
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#define BUF_SIZE 4096 // Dimensione del buffer: 4096 byte
#define OUTPUT_MODE 0700 // Bit di protezione per file di output
#define TRUE 1

// Prototipo della funzione main secondo lo standard ANSI
int main(int argc, char *argv[]);

int main(int argc, char *argv[]) {
    int in_fd, out_fd; // File descriptor per i file di input e output
    int rd_count, wt_count; // Contatori per la lettura e scrittura
    char buffer[BUF_SIZE]; // Buffer per la lettura e scrittura dei dati

    // Controllo del numero di argomenti
    if (argc != 3) {
        // Stampa un messaggio di errore se il numero di argomenti non è corretto
        fprintf(stderr, "Errore di sintassi. Uso: %s input_file_path output_file_path\n", argv[0]);
        exit(1);
    }

    // Apertura del file di input
    in_fd = open(argv[1], O_RDONLY); // Apre il file di origine
    if (in_fd < 0) exit(2); // Se non può aprirlo, esce

    // Creazione del file di output
    // Nota: equivalenza tra
    // - creat(path, mode);
    // - open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
    // YES: Ken Thompson, the creator of Unix, once joked that the missing letter was his largest regret in the design of Unix.
    out_fd = creat(argv[2], OUTPUT_MODE); // Crea il file di destinazione
    if (out_fd < 0) exit(3); // Se non può crearlo, esce

    // Ciclo di copia
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); // Legge un blocco di dati
        if (rd_count <= 0) break; // Se fine del file o errore, esce dal ciclo

        wt_count = write(out_fd, buffer, rd_count); // Scrive i dati
        if (wt_count <= 0) exit(4); // wt_count <= 0 è un errore
    }

    // Chiusura dei file
    close(in_fd);
    close(out_fd);

    if (rd_count == 0) exit(0); // Nessun errore sull'ultima lettura
    else exit(5); // Errore sull'ultima lettura
}
```

## Sintesi del programma:

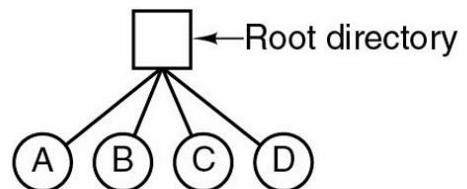
- Funzionalità: Programma in C per copiare il contenuto di un file di input in un file di output.
- Controlli: Verifica la correttezza del numero di parametri forniti all'applicazione.
- Operazioni di File:
  - o Apertura del file sorgente per la lettura.
  - o Creazione/apertura del file destinazione con permessi specifici.
  - o Copia del contenuto tramite un buffer di 4096 byte.
- Gestione degli Errori: Esce con un codice di errore specifico se incontra problemi nell'apertura, nella lettura, nella scrittura o nella chiusura dei file.
- Chiusura: I file vengono chiusi e il programma termina con un codice di stato appropriato a seconda dell'esito delle operazioni.

## DIRECTORY

Le directory, o cartelle, 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.
- Esempi Storici: Comune nei primi PC e nel supercomputer CDC 6600.
- Vantaggi: Semplicità e rapidità nella localizzazione dei file.



Esempio con quattro file in un sistema a directory singola.

### → EVOLUZIONE E APPLICABILITÀ DEI SISTEMI DI DIRECTORY A LIVELLO SINGOLO

- Evoluzione dei Concetti di File System: Molti concetti, come la directory singola, sono ciclici: emergono, cadono in disuso, e riemergono in nuovi contesti.
- Applicabilità Moderna:
  - o Dispositivi Embedded: Concetti semplici di file system sono ancora utili in dispositivi come fotocamere digitali o riproduttori MP3.
  - o Tecnologie RFID: Sistemi di directory semplici possono essere adatti per chip RFID o carte di credito e tessere di trasporto.
  - o Riflessione: Idee apparentemente obsolete possono essere rilevanti in contesti moderni e dispositivi a basso costo.

Nota: Le cartelle utilizzano overhead!

Nota: Non tutti i SO utilizzano cartelle!

### → SISTEMI DI DIRECTORY GERARCHICI

Limiti dei Sistemi a Singolo Livello:

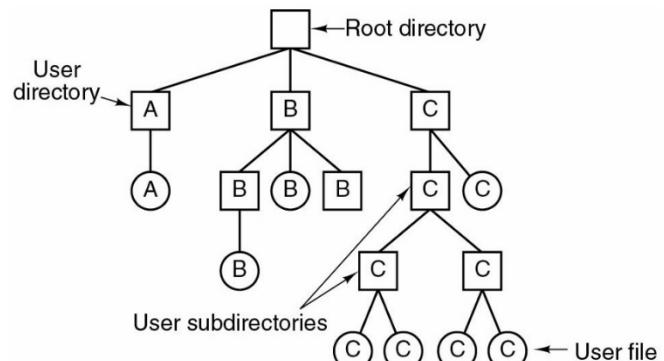
1. Non pratici per utenti con migliaia di file.
2. Difficoltà nel rintracciare i file in un unico spazio.

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.
- Storicamente, il file system gerarchico è stato sperimentato inizialmente in Multics negli anni '60.



La directory principale (Root) divisa in directory A, B e C, ognuna appartenente a utenti diversi.

- Possibilità di creare sottodirectory per progetti specifici o categorie di file.

## → NOMI DI PERCORSO NEI FILE SYSTEM GERARCHICI

Specificare i Nomi dei File: Necessità di definire i percorsi dei file in un sistema di directory ad albero.

Nomi di Percorso Assoluti:

- Percorsi che iniziano dalla directory principale e conducono al file.
- Unici per ogni file (es. /usr/ast/mailbox).
- Separatore di percorso: / per UNIX, \ per Windows, > per MULTICS.

Nomi di Percorso Relativi:

- Basati sulla directory di lavoro (directory corrente) dell'utente.
- Percorsi non iniziano con il separatore sono considerati relativi (es. mailbox).
- Esempi di comandi equivalenti in una data directory di lavoro:
  - o cp /usr/hjb/mailbox /usr/hjb/mailbox.bak
  - o cp mailbox mailbox.bak

## → UTILIZZO PRATICO E IMPLICAZIONI

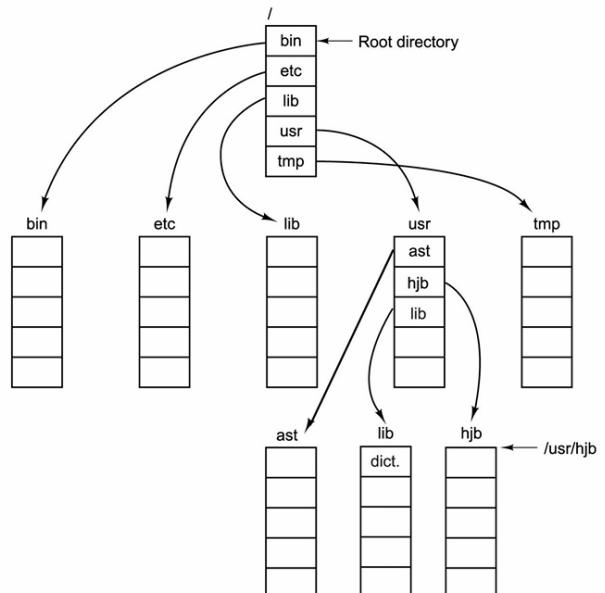
Directory di Lavoro (Working Directory):

- 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.
- Usati per navigare nell'albero dei file.  
Esempio = cp ..//lib/dictionary .



Un esempio di albero di directory UNIX.

## → OPERAZIONI SULLE DIRECTORY

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.

## → GESTIONE DEI LINK E ACCESSI AVANZATI

Linking e Unlinking:

- *link*: Crea un hard link, collegando un file esistente a un nuovo percorso, condividendone l'i-node.
- *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 (meglio nella prossima lezione)

Considerazioni Aggiuntive:

- Esistono altre chiamate per gestire dettagli come le informazioni di protezione di una directory.
- I link simbolici offrono flessibilità oltre i limiti dei dischi ma possono essere meno efficienti rispetto agli hard link.

## ➔ PROGRAMMA OPERAZIONI SU DIRECORY [11\_SHOW\_DIR\_CONTENT.C]

```
/*
 * Questo programma elenca il contenuto di una directory specificata come argomento,
 * mostrando informazioni dettagliate su ogni file e directory, simile al comando `ls -l` di UNIX.
 *
 * **Uso:**
 *   ./program_name <directory>
 *
 * **Descrizione delle funzioni principali:**
 * - `printf`: Stampa le informazioni di ciascun file in un formato leggibile dall'utente.
 *   Esempi in questo codice:
 *     - `printf("%llu ", fileInfo.st_size);` : Stampa la dimensione del file.
 *     - `printf((S_ISDIR(fileInfo.st_mode)) ? "d" : "-");` : Usa una ternaria per stampare "d" se è una directory o "-" altrimenti.
 *     - `printf("%s %s", pw->pw_name, gr->gr_name);` : Stampa i nomi del proprietario e del gruppo del file.
 *
 * - `snprintf`: Costruisce una stringa sicura di lunghezza limitata. Utilizzato per concatenare il percorso del file:
 *   ````c
 *   snprintf(filePath, sizeof(filePath), "%s/%s", dirName, fileName);
 *
 * Questo metodo protegge da buffer overflow, specificando la dimensione massima del buffer.
 *
 * - `strftime`: Formatta e stampa una data in una stringa leggibile:
 *   ````c
 *   strftime(dateStr, sizeof(dateStr), "%b %d %H:%M", localtime(&fileInfo.st_mtime));
 *   ```
 *
 * In questo caso, la data di ultima modifica del file viene convertita in formato leggibile
 * (es., "Oct 05 15:30").
 *
 * **Differenze principali tra `printf` e `snprintf`:**
 * - `printf`: Stampa direttamente su `stdout` (console).
 * - `snprintf`: Scrive in un buffer specificato dall'utente, permettendo un controllo sicuro sulla dimensione del buffer.
 *
 * **Esempi nel programma:**
 * 1. `snprintf` è usato per costruire il percorso completo del file (`filePath`), evitando di superare i limiti di memoria.
 * 2. `printf` è usato per stampare informazioni direttamente sulla console in formato leggibile.
 * 3. `strftime` è usato per formattare la data in modo comprensibile per l'utente.
 *
 * Questo approccio consente di combinare efficienza (costruzione sicura delle stringhe) e leggibilità
 * (output dettagliato e ben formattato).
 */
#include <stdio.h>
#include <dirent.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
void printFileInfo(const char* dirName, const char* fileName) {
    struct stat fileInfo;
    char filePath[1024];
    snprintf(filePath, sizeof(filePath), "%s/%s", dirName, fileName);

    if (stat(filePath, &fileInfo) < 0) {
        perror("Errore nel recupero delle informazioni del file");
        return;
    }

    printf("%llu ", fileInfo.st_size); // Dimensione del file

    // Permessi del file
    printf((S_ISDIR(fileInfo.st_mode)) ? "d" : "-");
    printf((fileInfo.st_mode & S_IRUSR) ? "r" : "-");
    printf((fileInfo.st_mode & S_IWUSR) ? "w" : "-");
    printf((fileInfo.st_mode & S_IXUSR) ? "x" : "-");
    printf((fileInfo.st_mode & S_IRGRP) ? "r" : "-");
    printf((fileInfo.st_mode & S_IWGRP) ? "w" : "-");
    printf((fileInfo.st_mode & S_IXGRP) ? "x" : "-");
    printf((fileInfo.st_mode & S_IROTH) ? "r" : "-");
    printf((fileInfo.st_mode & S_IWOTH) ? "w" : "-");
    printf((fileInfo.st_mode & S_IXOTH) ? "x" : "-");

    // Proprietario e gruppo
    struct passwd *pw = getpwuid(fileInfo.st_uid);
    struct group *gr = getgrgid(fileInfo.st_gid);
    printf(" %s %s", pw->pw_name, gr->gr_name);

    // Data di ultima modifica
    char dateStr[128];
    strftime(dateStr, sizeof(dateStr), "%b %d %H:%M", localtime(&fileInfo.st_mtime));
    printf(" %s", dateStr);

    // Nome del file
    printf(" %s\n", fileName);
}
```

```

int main(int argc, char *argv[]) {
    DIR *dirp;
    struct dirent *dirent;

    if (argc != 2) {
        fprintf(stderr, "Errore: manca inputfile. Uso: %s <directory>\n", argv[0]);
        return 1;
    }

    dirp = opendir(argv[1]);
    if (dirp == NULL) {
        perror("Errore nell'apertura della directory");
        return 1;
    }

    while ((dirent = readdir(dirp)) != NULL) {
        printFileInfo(argv[1], dirent->d_name);
    }

    closedir(dirp);
    return 0;
}

```

Funzionalità Aggiunta: Oltre a stampare i nomi, mostra informazioni dettagliate sui file nella directory, simili al comando ls-lh.

Informazioni Incluse:

- Dimensione del file.
- Permessi di accesso (lettura, scrittura, esecuzione).
- Proprietario e gruppo del file.
- Data e ora dell'ultima modifica.

Implementazione:

- Uso della funzione stat() per ottenere i metadati dei file.
- Formattazione e stampa delle informazioni in modo chiaro e leggibile.

## CREAZIONE DI ARCHIVI

File TAR: Cosa Sono e Perché si Usano

- 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.

Utilizzo di TAR e GZ:

- Creazione di un Archivio tar.gz
  - o Comando di base = `tar -czvf nome-archivio.tar.gz /percorso/della/cartella`
  - o In czvf:
    - 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
  - o 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 e UNZIP: Caratteristiche

- 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.
- Comandi Comuni:
  - o `zip nome-archivio.zip file1 file2`
  - o `unzip nome-archivio.zip`.
- Vantaggi: Compatibilità ampia con diversi sistemi operativi, compressione individuale dei file.

TAR.GZ: Caratteristiche

- 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.
- Conservazione dei Metadati: TAR.GZ mantiene meglio la struttura originale e i permessi dei file.
- Compatibilità Universale: ZIP è più comunemente supportato su diverse piattaforme, incluse Windows e macOS.

Consiglio professore: non utilizzare RAR

Nota: "La compressione è più lenta della decompressione"

## POSSIBILE DOMANDA D'ESAME

Possibile domanda d'esame: Quanti byte servono per rappresentare in testuale 43 (la stringa "43")?

Risposta: In memoria ho bisogno di 3 byte, perché occorre 1byte per il carattere '4', 1byte per il carattere '3' ed 1 byte per il carattere separatore.

## LEZ12

### TEMA LEZIONE = IMPLEMENTAZIONE DEL FILE SYSTEM

Alcune domande e risposte relative ai File System:

#### 1) COME MEMORIZZARE I FILE?

##### → INTRO

Definizione: Il file system è il metodo utilizzato per organizzare e memorizzare dati sui dispositivi di memoria non volatile (dischi/SSD).

Importanza: Fornisce un modo strutturato per gestire informazioni come file e directory su dispositivi di memoria.

Partizioni del Disco: Un disco può essere suddiviso in più partizioni, ciascuna con un proprio file system indipendente.

Evoluzione: I metodi di strutturazione del file system variano a seconda dell'epoca del computer, influenzando come i dati vengono gestiti e acceduti.

#### → VECCHIO STILE – BIOS CON MBR (MASTER BOOT RECORD)

MBR nel BIOS:

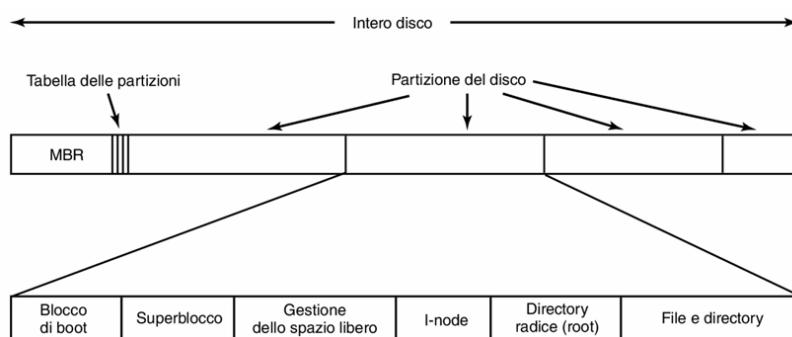
- Situato nel settore 0 del disco, l'MBR è essenziale per l'avvio del computer.
- Contiene la tabella delle partizioni con dettagli su inizio e fine di ciascuna partizione.
- Identifica la partizione attiva da cui avviare il sistema.

Processo di Avvio:

- Il BIOS legge l'MBR per trovare la partizione attiva.
- 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.
- Immagine seguente è un esempio di layout, includendo superblocco, bitmap, I-node e directory radice



#### → NUOVA SCUOLA – UEFI (UNIFIED EXTENSIBLE FIRMWARE INTERFACE)

UEFI è un sistema moderno che sostituisce il vecchio BIOS tradizionale.

Vantaggi:

1. Avvio più veloce: Ottimizza il processo di inizializzazione dell'hardware.
2. Compatibilità migliorata: Supporta architetture hardware a 32 e 64 bit.
3. Interfaccia utente avanzata: Può includere grafica e supporto per mouse.
4. Sicurezza: Aggiunge funzionalità come il Secure Boot.
5. Introduce una maggiore flessibilità e compatibilità con diverse architetture hardware

Supporto per dischi moderni: UEFI funziona con GPT, consentendo di superare i limiti di 2,2 TB dei dischi gestiti con il vecchio schema MBR.



## → GPT E EFI SYSTEM PARTITION (ESP)

GPT = Guid Partition Table, è un sistema avanzato di gestione delle partizioni che ha le seguenti caratteristiche:

- a) Supporta dischi fino a 8ZiB
- b) Consente un numero illimitato di partizioni (limite imposto solo dal sistema operativo).
- c) Include backup della tabella delle partizioni per maggiore sicurezza.
- d) Utilizza un controllo di integrità (CRC) per prevenire corruzione dei dati.

EFI = è una partizione speciale sui dischi GPT. Funzione:

- a) Archiviare i file di avvio come bootloader, driver e utility di diagnostica
- b) È essenziale per avviare il SO
- c) Usa il file system FAT32, garantendo compatibilità con il firmware UEFI
- d) Il firmware UEFI legge la GPT e carica i file dalla partizione EFI per avviare il sistema

## → SECURE BOOT

È una funzionalità UEFI progettata per impedire l'avvio di software non autorizzato.

Funzionamento:

- Controlla le firme digitali di bootloader, driver e sistema operativo.
- Avvia solo software autorizzato e firmato.
- Blocca malware e rootkit durante l'avvio.

Vantaggi:

- Protegge contro attacchi all'avvio (es. rootkit, malware).
- Mantiene l'integrità del sistema operativo.
- Aumenta la sicurezza per utenti domestici e aziende.

Considerazioni:

- Alcuni sistemi operativi (es. alcune distribuzioni Linux) potrebbero richiedere la disattivazione del Secure Boot per funzionare.
- Disattivabile nelle impostazioni UEFI, ma ciò può esporre il sistema a rischi.

## → IMPLEMENTAZIONE DEI FILE NEI FILE SYSTEM

Obiettivo Principale: Gestire l'associazione tra i file e i blocchi del disco su cui sono memorizzati.

Importanza: Fondamentale per assicurare integrità, accesso efficiente e gestione dello spazio su disco.

Varietà di Metodi: Diversi sistemi operativi adottano approcci differenti per questa associazione.

Basato su: Metodi basati su indici , Liste concatenate , Bitmap , Strutture ad albero.

Focus: Analisi dei vari metodi e delle loro caratteristiche specifiche nel contesto dei diversi file system.

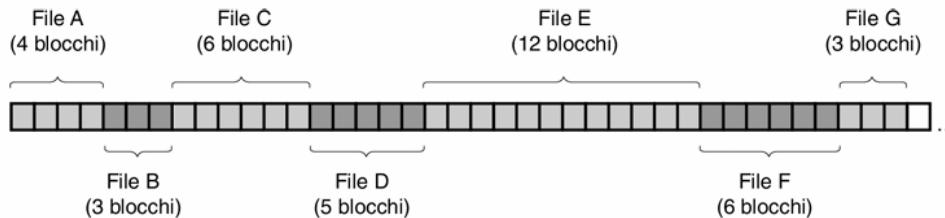
## → ALLOCAZIONE CONTIGUA NEL FILE SYSTEM+PROBLEMI E LIMITAZIONI DELL'ALLOCAZIONE CONTIGUA

Nota: Allocazione contigua utilizzata intorno agli anni 70/80!

Concetto di Allocazione Contigua: I file sono memorizzati come sequenze contigue di blocchi sul disco.

Esempio: un file di 50 KB su un disco con blocchi da 1 KB occupa 50 blocchi consecutivi.

(In basso: l'allocazione contigua di sette file su disco).



Implementazione e Vantaggi:

- Semplice da implementare: richiede solo l'indirizzo del primo blocco e il numero totale di blocchi.
- Alta efficienza di lettura: l'intero file può essere letto in una sola operazione, senza ritardi.

Problemi e limitazioni dell'allocazione contigua:

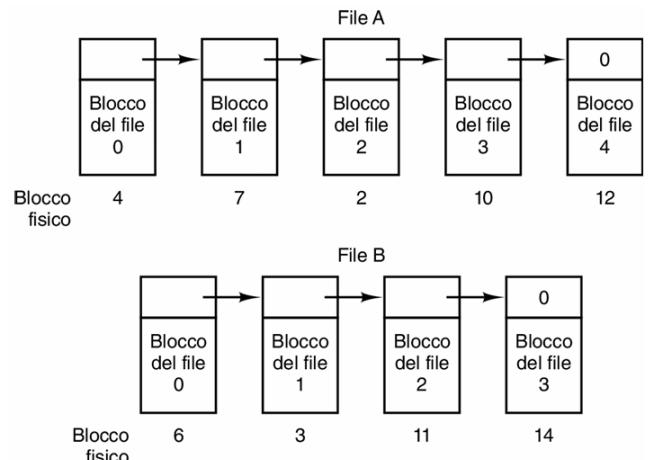
- Frammentazione del Disco:
  - Col passare del tempo, i dischi si frammentano a causa della rimozione di file (la frammentazione lascia intervalli di blocchi liberi).
  - Problema di allocazione di nuovi file in spazi liberi frammentati.
- Gestione dello Spazio Libero:
  - Richiede una lista di spazi liberi e la conoscenza della dimensione finale dei nuovi file.
  - Problemi nella previsione della dimensione del file e nella ricerca di spazi adeguati.
- Implicazioni Pratiche:
  - Difficoltà nell'aggiungere nuovi file in un disco frammentato.
  - Necessità di compattazione del disco o di gestione intelligente dello spazio libero.

## → ALLOCAZIONE A 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, minima frammentazione esterna: riduce lo spreco di spazio non utilizzato.

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.



Prestazioni e limitazioni dell'allocazione a liste concatenate:

- Accesso ai Dati:
  - Accesso Sequentiale: Leggere un file è efficiente, procedendo blocco per blocco.
  - Accesso Casuale (seek): Estremamente lento, richiede 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.
  - Letture e scritture di dimensioni standard (potenze di due) possono essere meno efficienti.
- Implicazioni Pratiche:
  - Il metodo offre un'elevata efficienza nello sfruttamento dello spazio su disco ...
  - MA introduce complessità e rallentamenti nelle operazioni di accesso casuale.
  - Adatto per file a cui si accede principalmente in modo sequenziale.

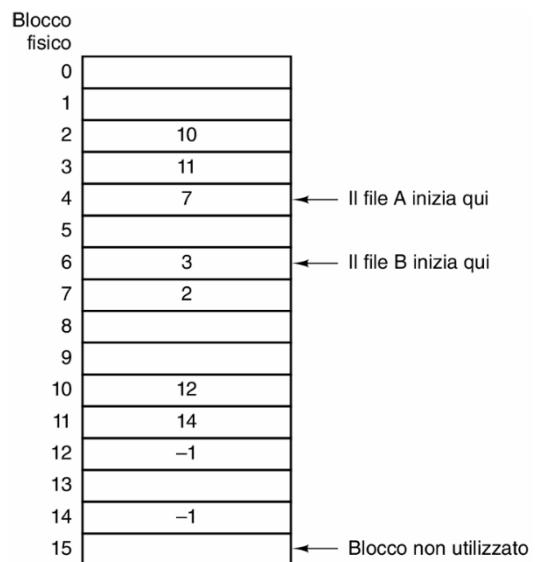
## → ALLOCAZIONE A LISTE CONCATENATE CON FAT

Ottimizzazione dell'Allocazione a Liste Concatenate:

- Eliminazione degli svantaggi dell'allocazione a liste concatenate spostando i puntatori in una tabella di memoria (*FAT - File AllocationTable*).
- Ogni blocco del disco è rappresentato come una voce nella FAT... IN MEMORIA RAM

Struttura della FAT: Contiene la sequenza dei blocchi di ciascun file.

- Esempio:
  - o File A utilizza i blocchi 4, 7, 2, 10, 12;
  - o File B i blocchi 6, 3, 11, 14.
  - o Sequenze terminate da un indicatore speciale (es. -1) per marcare la fine.
  - o In memoria principale



Vantaggi della FAT:

- L'intero blocco è disponibile per i dati, ottimizzando lo spazio.
- Accesso casuale semplificato: la sequenza dei blocchi è interamente in memoria.

Limitazioni della FAT:

- In termini di gestione di memoria, in quanto la FAT deve essere mantenuta internamente in memoria principale. Inoltre, richiede una quantità significativa di memoria (per un disco da 1TB con blocchi da 1KB, la FAT richiederebbe fino a 3GB di RAM).
- L'approccio non è ottimale per dischi di grandi dimensioni a causa dell'elevato consumo di memoria (lo spazio e la velocità influenzano la dimensione della voce della FAT (da 3 a 4 byte per ogni voce)).

Utilizzo pratico della FAT: Originariamente implementato in MS-DOS, ancora supportato da Windows e UEFI. Attualmente invece, viene comunemente usato in dispositivi portatili come schede SD in fotocamere, lettori musicali e altri dispositivi elettronici.

## → I-NODE

I-node (Index Node) nei File System UNIX-like:

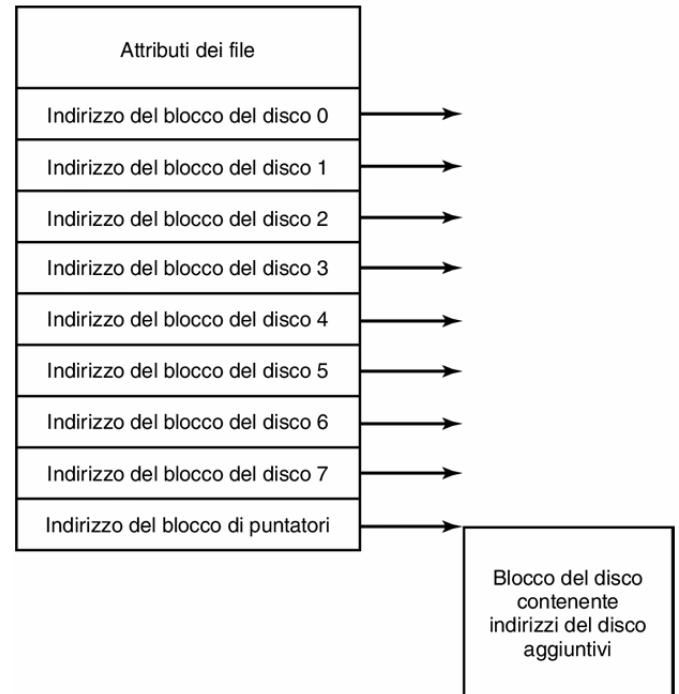
- Definizione: Struttura dati fondamentale nei file system come ext2/ext3/ext4 in Linux.
- Contenuto: Contiene tutte le informazioni su un file, esclusi il nome e il contenuto. Include metadati come permessi, proprietario, timestamp e indirizzi dei blocchi di dati.
- Funzione: Ogni file e directory è rappresentato da un I-node univoco, indicizzato in una tabella di I-node.

Confronto con FAT (File AllocationTable):

- Gestione dei File:
  - o 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.
  - o I sistemi I-node separano le informazioni sul file dalla sua posizione fisica sul disco.
- Informazioni sui File:
  - o FAT fornisce meno dettagli sui file, concentrandosi principalmente sull'allocazione dello spazio.
  - o 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.

## Funzionamento e vantaggi degli I-node:

- I-node (Index-Node) nei File System
  - o Gli I-node sono strutture dati che elencano gli attributi e gli indirizzi dei blocchi dei file.
  - o Ogni I-node rappresenta un file, fornendo un metodo efficiente per trovare tutti i suoi blocchi di dati.
- Efficienza della Memoria con I-node
  - o Solo gli I-node dei file aperti sono mantenuti in memoria, riducendo significativamente l'utilizzo della memoria.
  - o L'array degli I-node in memoria è proporzionale al numero di file aperti, non alla dimensione del disco.
- 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 dei file di grandi dimensioni e confronto con NTFS:

- Gestione File di Grandi Dimensioni
  - o Gli I-node hanno uno spazio limitato per gli indirizzi del disco.
  - o Per file che superano il limite, uno degli indirizzi nell'I-node punta a un blocco contenente ulteriori indirizzi di blocchi di dati.
  - o Questo sistema permette di gestire file di dimensioni molto grandi con efficacia.
- I-node nei File System UNIX e Windows NTFS
  - o Gli I-node sono un concetto fondamentale in UNIX e nei suoi file system derivati.
  - o 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.

## 2) COME IMPLEMENTARE LE DIRECTORY?

### → CONCETTI BASE DELL'IMPLEMENTAZIONE DELLE DIRECTORY

Funzione Principale: Le directory mappano i nomi ASCII dei file sulle informazioni necessarie per localizzare i dati su disco.

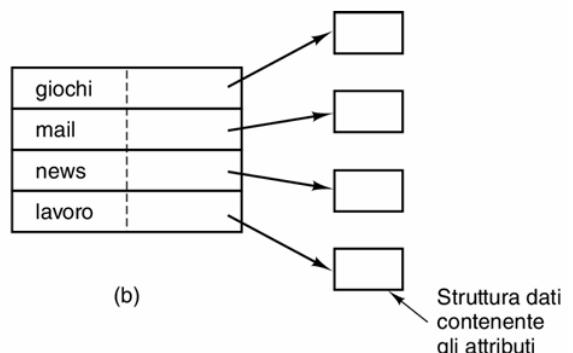
Metodi di Allocazione: Variano a seconda del sistema operativo, includendo indirizzi di blocchi contigui, il primo blocco nelle liste concatenate, o i numeri degli I-node.

- a) Una semplice directory contenente voci a dimensione fissa con gli indirizzi del disco e gli attributi nella voce della directory.  
(fatto così negli anni 80)

giochi	attributi
mail	attributi
news	attributi
lavoro	attributi

(a)

- b) Una directory in cui ogni voce fa soltanto riferimento a un I-node.  
(fatto così in UNIX)



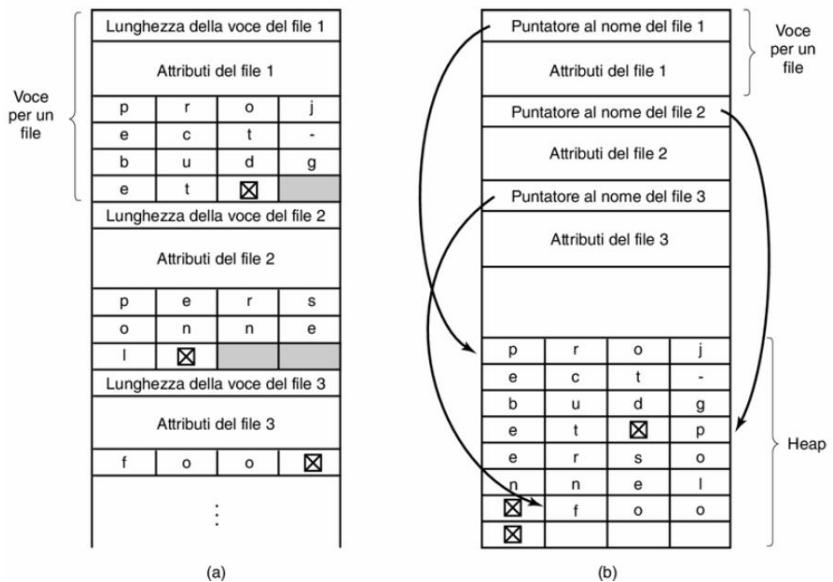
## → GESTIONE DEI NOMI DEI FILE PER L'IMPLEMENTAZIONE DELLE DIRECTORY

Nomi di File Variabili: Supporto per nomi di file di lunghezza variabile, con un limite tipico di 255 caratteri.

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 sfide (nella gestione degli spazi vuoti (un file viene cancellato)).



## → OTTIMIZZAZIONE DELLA RICERCA NELLE DIRECTORY CON TABELLE DI HASH

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.

Uso delle Tabelle di Hash:

- Introduzione di 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: Creazione di 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.

## → GESTIONE DELLA CACHE PER RICERCHE EFFICIENTI

Caching delle Ricerche: Salvataggio dei risultati di ricerche comuni nella cache per 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 richiesti frequentemente. È 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.
- Più adatto a sistemi con directory molto estese, dove si prevede un elevato numero di file.

## → FILE CONDIVISI E LINK NEI FILE SYSTEM

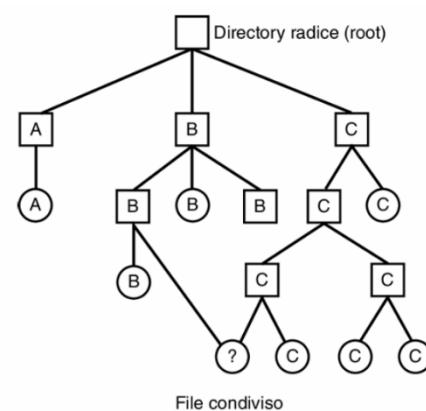
File Condivisi: Essenziali in ambienti collaborativi per permettere a più utenti di lavorare sugli stessi file.

Tipi di Link:

- 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.
- Efficienza di spazio: una sola voce di directory per ciascun hard link.
- Ideali per la gestione di file condivisi tra più proprietari.



Un file system con un file condiviso tra due utenti.

- Esempio: Un file di un utente può essere presente anche nella directory di un altro.

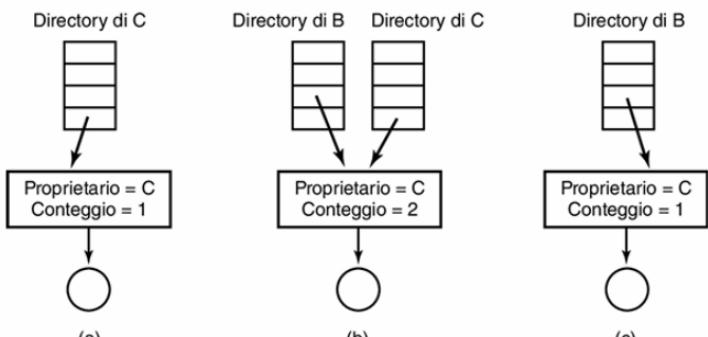
## → GESTIONE E PROBLEMI DEI FILE CONDIVISI: HARD LINKS

Vantaggi degli Hard Link:

- Spazio-efficienti: usano un solo I-node indipendentemente dal numero di link.
- Gestione trasversale degli utenti: 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.

Nell'immagine a destra: Illustrano come i file condivisi sono gestiti nel file system e le implicazioni dell'eliminazione di hard link.



a) Situazione prima del link.

b) Dopo la creazione del link.

c) Dopo che il proprietario originale elimina il file.

## → GESTIONE E PROBLEMI DEI FILE CONDIVISI: LINK SIMBOLICI

Vantaggi dei Link Simbolici:

- Maggiore flessibilità: possono riferirsi a nomi di file oltre i confini del file system e su macchine remote.
- 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.
- 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 (Rischio di duplicazione dei file su unità di backup).
- Necessità di software avanzato per gestire correttamente i file condivisi e i loro link.

## 3) COME GESTIRE LO SPAZIO SU DISCO?

### → CONSIDERAZIONI GENERALI SULLA GESTIONE DELLO SPAZIO SU DISCO

Memorizzazione dei File: I file sono generalmente memorizzati su disco, e ci sono due modi principali per farlo:

1. Allocazione contigua
2. Suddivisione in blocchi non contigui.

Allocazione Contigua vs Blocchi:

- 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.

Dimensione dei Blocchi: 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.

## → EFFICIENZA E PRESTAZIONI (ANALISI DEL COMPROMESSO)

Prestazioni di Trasferimento Dati:

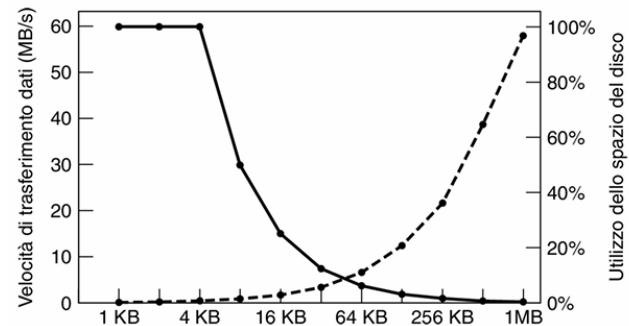
- I dischi magnetici con blocchi più grandi consentono il trasferimento di più dati per operazione di lettura/scrittura.
- MA: blocchi grandi portano a spreco di memoria

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).

Conflitto tra Prestazioni ed Efficienza:

- 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.



La curva tratteggiata (con scala a sinistra) indica la velocità di trasferimento dati di un disco. La curva continua (scala a destra) esprime l'efficienza nell'utilizzo dello spazio del disco. Tutti i file sono di 4 KB.

## → IMPLICAZIONI PER DISCHI MAGNETICI E 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'incremento della capacità dei dischi (TB), potrebbe essere vantaggioso considerare blocchi più grandi per accettare una minore efficienza dello spazio in cambio di prestazioni migliori.

## → GESTIONE DEI BLOCCHI LIBERI NEL DISCO

Come tenere traccia dei blocchi liberi?

Metodo 1: Lista Concatenata

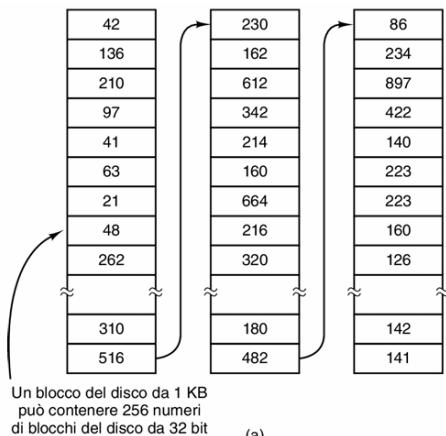
Utilizzo di una lista concatenata di blocchi del disco:

- Nella lista solo i blocchi liberi
- 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 entrate.



Efficienza: Richiede meno spazio solo se il disco è quasi pieno.

Metodo 2: Bitmap

Utilizzo di una bitmap per tracciare i blocchi liberi:

- Un bit per ogni blocco del disco, 1 indica libero mentre 0 indica 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 (La lista concatenata deve considerare meno blocchi liberi e quindi ha meno blocchi occupati).

1001101101101100
0110110111110111
1010110110110110
0110110110110101
1110111011101111
1101101010001111
0000111011010111
1011101101101111
1100100011101111
~
0111011101110111
1101111011101111

Una bitmap

(b)

## → OTTIMIZZAZIONE E PROBLEMI NELL'USO 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:
  - o l'indirizzo del primo blocco libero
  - o 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).

Sfide nella Progettazione di Sistemi Operativi: Scelta della struttura dati ottimale senza dati anticipati sull'uso del sistema.

Esempi: Differenze nella gestione dei file e nell'utilizzo del disco tra diversi dispositivi e ambienti.

## → GESTIONE DEI BLOCCHI LIBERI CON LISTA DI PUNTATORI

Funzionamento della Free List:

- 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 crea un file, i blocchi necessari vengono allocati dai puntatori disponibili nel blocco in memoria.

Ottimizzazione del flusso di I/O:

- 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.

Efficienza e File Temporanei:

- 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 (Vedi Figura a VS Figura b).
- Una strategia alternativa prevede di dividere il blocco pieno di puntatori per gestire meglio i blocchi liberi senza I/O su disco (Vedi Figura a VS Figura c).

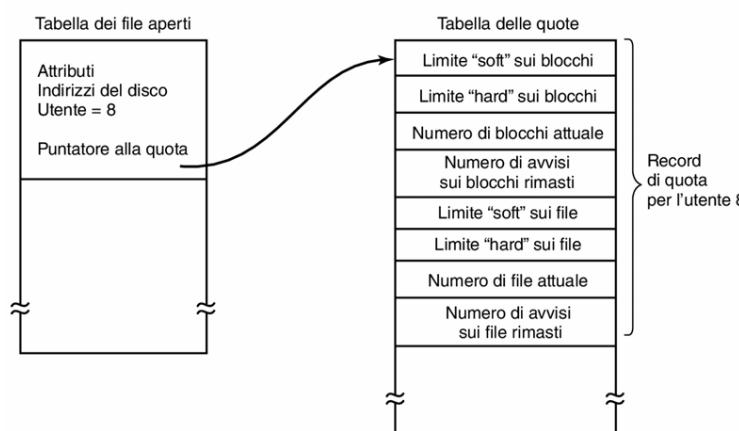
## → MECCANISMO DI QUOTA DEL DISCO NEI SO MULTIUTENTE

Assegnazione delle Quote:

- 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.

Gestione e Controllo delle Quote:

- Ogni apertura di file coinvolge il controllo degli attributi e degli indirizzi del disco.
- Attributi includono l'identificazione del proprietario del file.
- Incrementi della dimensione del file sono contabilizzati nella quota del proprietario.



## → CONTROLLI E LIMITI DELLE QUOTE

Tabelle delle Quote:

- 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 e Consequenze:

- Limiti "soft" e "hard" per le quote: "soft" può essere temporaneamente superato, "hard" mai.
- Violazioni 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.

## INTERMEZZO (ORGANIZZAZIONE EXT2)

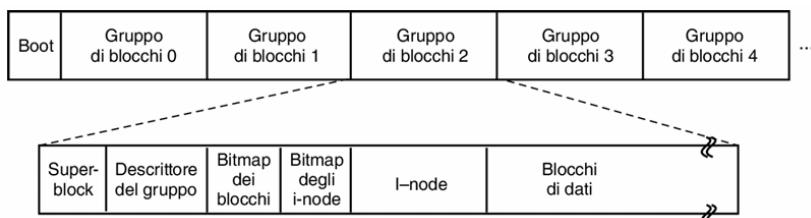
### ORGANIZZAZIONE E GESTIONE NEL FILE SYSTEM EXT2

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 e Blocchi di Dati:

- 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.



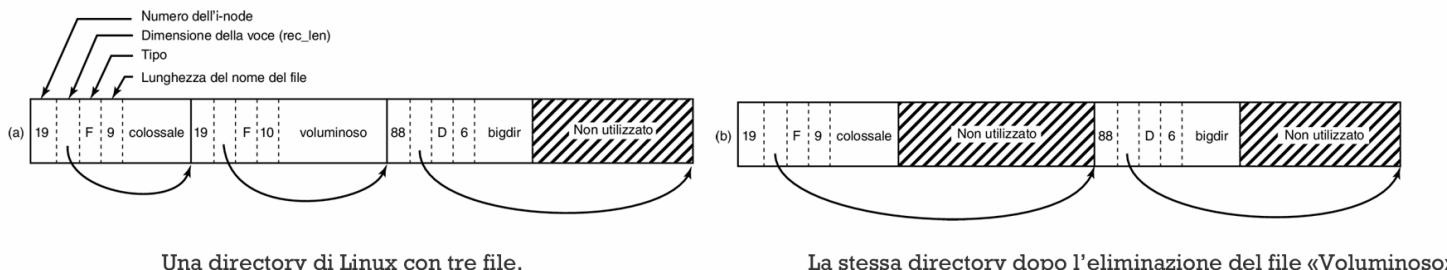
### GESTIONE DEGLI I-NODE E DEI FILE NEL FILE SYSTEM EXT2 DI LINUX

Collocazione 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.
- Utilizzo di bitmap per determinare rapidamente aree libere dove allocare nuovi dati del file system.

Preallocazione di Blocchi:

- Ext2 preallocata 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.



Una directory di Linux con tre file.

La stessa directory dopo l'eliminazione del file «Voluminoso».

### STRUTTURA E GESTIONE DELLE DIRECTORY IN EXT2

Accesso ai File: Utilizzo di chiamate di sistema, come open, per accedere 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.

Uso di Soft link e Hard link per fare riferimento ai file

#### 4) COME GARANTIRE LE PRESTAZIONI DEL FILE SYSTEM?

##### → COMPARAZIONE DELLE PRESTAZIONI DEL FILE SYSTEM: MEMORIA VS DISCO MAGNETICO

Velocità di Accesso:

- Memoria: Accesso ultraveloce (es. 10 ns per leggere una parola a 32 bit).
- Disco Magnetico: Accesso 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.

Differenza nelle Prestazioni: L'accesso a un disco magnetico può essere milioni di volte più lento dell'accesso alla memoria.

Ottimizzazioni nei File System:

- Progettazione di 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
- l'utilizzo dello spazio

##### → OTTIMIZZAZIONE DELLE PRESTAZIONI DEL FILE SYSTEM

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 intelligenti: 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.

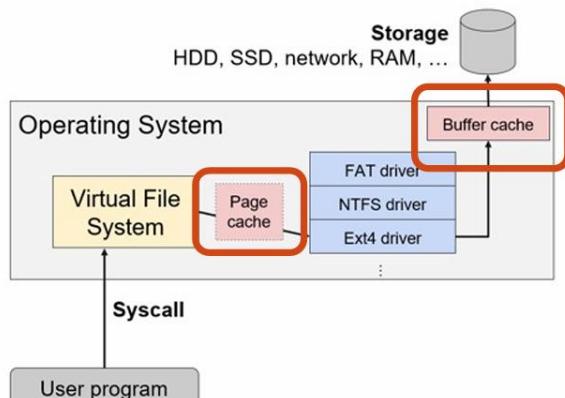
Deframmentazione:

- Con il tempo, i dischi si frammentano; i file sparsi portano a prestazioni inferiori.
- Deframmentazione: Riorganizza i file per essere contigui e raggruppa lo spazio libero.
- Windows fornisce lo strumento defrag per questa operazione (consigliato regolarmente per HDD, ma non per SSD).

##### → MINIMIZZAZIONE DELL'ACCESSO AL DISCO TRAMITE CACHING

Concetti di caching

- Buffer Cache: Memorizza i blocchi del disco in RAM per ridurre gli accessi al disco.
- Page Cache: Memorizza le pagine del filesystem virtuale (VFS, più avanti) 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)
- 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.

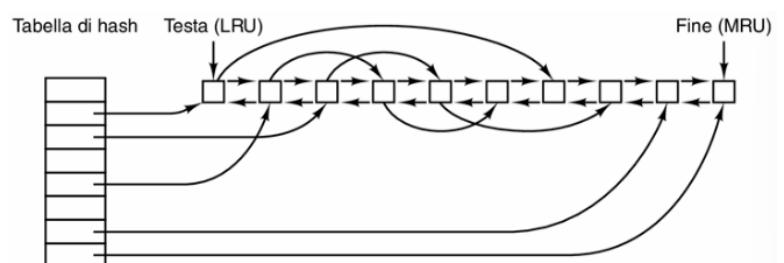
##### → IMPLEMENTAZIONE E FUNZIONAMENTO DELLA CACHE NEI FILE SYSTEM

Definizione e Scopo della Cache:

- Cache (block cache o buffer cache): Raccolta di blocchi del disco tenuti in memoria per migliorare le prestazioni.
- Scopo: 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.



Ottimizzazione della Ricerca nella Cache: Uso di hash per identificare rapidamente la presenza di un blocco in cache (Lista concatenata per gestire i blocchi con lo stesso valore hash).

## → GESTIONE DELLE CACHE E ALGORITMI DI SOSTITUZIONE

### Sostituzione dei Blocchi nella Cache

- Processo: 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 (Least Recently Used), simili alla paginazione.
  - Lista Bidirezionale LRU: Ordine d'uso dei blocchi (meno recenti in testa, più recenti in fondo) per mantenere l'esatto ordine LRU.

### Limitazioni dell'Algoritmo LRU

- Problema: LRU può portare a inconsistenze in caso di crash, specialmente per blocchi critici come i blocchi di I-node.
- Soluzione: Schema di LRU modificato basato sull'importanza e la necessità immediata dei blocchi.

### Scrittura e Coerenza del File System

- Blocchi critici: Scrittura immediata su disco se modificati per mantenere la coerenza del file system.
- Prevenzione della Perdita di Dati: 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.

## → STRATEGIE DI SCRITTURA IN UNIX E WINDOWS E INTEGRAZIONE DELLA CACHE

### Strategie di Scrittura nei Sistemi Operativi:

- UNIX: Uso della chiamata di sistema sync per scrivere periodicamente i blocchi modificati su disco.
- 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.
- Integrazione: Alcuni sistemi operativi combinano cache buffer e cache delle pagine per una gestione efficiente dei dati.

Supporto per File Mappati in Memoria: 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

## → COMPRENDERE L'USO DELLA MEMORIA IN LINUX CON IL COMANDO FREE

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.

### Visualizzazione della Cache di Memoria:

- Usa free -h per un output leggibile che include la dimensione e l'utilizzo della cache.
- La colonna "buff/cache" rivela quanto spazio è utilizzato per buffer e cache, inclusi i dati letti dal disco.

### Memoria Disponibile vs. Memoria Libera:

- Colonna "Free": mostra la memoria fisica non utilizzata attualmente.
- Colonna "Available": stima la memoria disponibile per nuovi processi, considerando anche la memoria facilmente liberabile come cache.
  - "Available" offre una visione realistica della memoria effettivamente disponibile per applicazioni senza influire sulle prestazioni.

Importanza per le Prestazioni del Sistema: La cache aiuta a velocizzare l'accesso ai file frequentemente usati, riducendo la necessità di leggere ripetutamente dal disco più lento.

## → DEFRAMMENTAZIONE DEL DISCO E LA SUA IMPORTANZA NEI DIVERSI SISTEMI

- Impatto della Frammentazione: Con il tempo, i file e lo spazio libero si disperdonano sul disco, causando una diminuzione delle prestazioni.
- Riorganizzazione dei Dati: La deframmentazione consolida i file e lo spazio libero, migliorando l'efficienza di lettura/scrittura per HDD.
- Strumenti e Pratiche: defrag in Windows riorganizza i file; raccomandato per HDD, sconsigliato per SSD per evitare usura inutile.
- Considerazioni sui File System: Linux con ext3/ext4 gestisce meglio la frammentazione, riducendo la necessità di deframmentazione manuale.
  - Preallocazione di Blocchi: 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.

## → OTTIMIZZAZIONE DELLO SPAZIO SU DISCO: COMPRESSESIONE E DEDUPPLICAZIONE

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.

Sicurezza e Affidabilità.

- Controllo degli Hash: Necessario per assicurare che i dati non siano falsamente identificati come duplicati a causa di collisioni hash.

## 5) COME GARANTIRE L'AFFIDABILITÀ DEL FILE SYSTEM?

### → MINACCIE ALLA AFFIDABILITÀ DEI FILE SYSTEM

Alcune minacce all'affidabilità del file system sono: Guasti del Disco(Blocchi Danneggiati, Errori su Intero Disco), Interruzioni di Energia(Scritture Inconsistenti), Bug del Software/Corruzione dei (Meta)dati, Errori Umani/Comandi Errati(`rm *.o` VS `rm * .o`), Perdita o Furto del Computer/Accesso Non Autorizzato, Malware/Ransomware.

### → BACKUP

Il backup è cruciale per salvaguardare informazioni importanti come documenti, database, piani aziendali, ecc.

Ci sono 2 modalità di backup:

1. il backup completo = Esegue una copia totale dei dati, solitamente su base settimanale o mensile
2. il backup incrementale = Copia solo i file modificati dall'ultimo backup completo, riducendo il tempo e lo spazio richiesti.

Ci sono 2 tipologie di backup, sono:

1. backup fisico = Copia sequenziale di tutti i blocchi del disco, a partire dal blocco 0 fino all'ultimo.
2. backup logico = Seleziona e copia solo i file e le directory specifici, ignorando i file di sistema e i blocchi danneggiati.

Considerazioni tecniche sul backup:

- a) Comprimere i Dati: Riduce lo spazio necessario, ma aumenta il rischio di perdita di dati a causa di errori di compressione.
- b) Backup di File System Attivi: Richiede l'uso di snapshot per garantire coerenza durante il backup di un sistema in uso.
- c) Sicurezza dei Backup: Importanza di tenere i backup in luoghi sicuri e separati per prevenire perdite o danni.

### → BACKUP FISICO

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 (blocchi danneggiati ci sono sempre).
- File Non Necessari: Necessità di evitare la copia di file di sistema come i file di paginazione e ibernazione

Vantaggi:

- Semplicità: Facile da implementare e utilizzare.
- Affidabilità: Minore probabilità di errori nel processo di backup.

Sfide e Limitazioni:

- Mancanza di Flessibilità: Difficoltà nel saltare directory specifiche o nel fare backup incrementalni.
- Ripristino di Singoli File: Non è possibile ripristinare file individuali senza un intero ripristino del sistema.

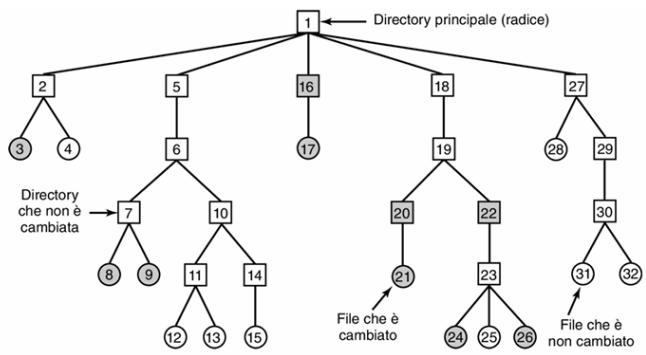
## → BACKUP LOGICO E IL SUO ALGORITMO

Funzionamento del Backup Logico:

- Parte da specifiche directory e effettua il backup di tutti i file e directory modificati a partire da una data specifica. Questo 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.



Un file system di cui eseguire il backup. I quadrati sono directory, i cerchi sono i file. Gli oggetti in grigio hanno subito modifiche dopo il backup precedente. Directory e file sono contrassegnati dal numero del loro I-node.

## → RSYNC ED ESEMPIO D'USO (SLIDES BONUS MA IMPORTANTI)

Definizione di rsync:

- *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 in ambienti di rete.

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 pratico:

- Sincronizzazione Locale: *rsync -av /sorgente/cartella /destinazione/cartella*
  - o *-a*: modalità archivio, mantiene i permessi e la struttura dei file.
  - o *-v*: modalità verbosa, mostra i dettagli del trasferimento.
- Sincronizzazione Remota: *rsync -av /sorgente/cartella utente@remoto:/destinazione/cartella*
  - o Sincronizza la cartella dalla macchina locale a quella remota utilizzando l'account utente.

## → INTRODUZIONE ALLA COERENZA DEL FILE SYSTEM

Importanza della Coerenza: è cruciale per mantenere l'integrità dei dati, problemi di incoerenza possono sorgere a seguito di crash durante la scrittura dei blocchi.

Utilità per la Verifica della Coerenza: Sistemi come UNIX (*fsck*) e Windows (*sfc*) hanno utility per verificare la coerenza, che vengono eseguite all'avvio, specialmente dopo un crash.

File System con Journaling (vedi dopo): Progettati per gestire autonomamente la maggior parte delle incoerenze. Non necessitano di controlli esterni dopo un crash.

## → PRINCIPI E FUNZIONAMNETO DEI FILE SYSTEM CON JOURNALING

Definizione e Scopo:

- File system con journaling: Registra anticipatamente le operazioni da eseguire in un log, per garantire la coerenza in caso di crash.
- Utilizzo: 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.
- Problemi in caso di crash: Perdita di accesso agli I-node e ai blocchi, o assegnazione errata di I-node e blocchi.

Cos'è il Journal? 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.

## → STRUTTURA E FUNZIONAMENTO DEL JOURNAL NEI FILE SYSTEM

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 completarle. Questo assicura che le modifiche parziali non lascino il file system in uno stato incoerente.

Vantaggi del Journaling:

- a) 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.
- b) Recupero Rapido: Riduce significativamente il tempo di recupero dopo un crash, poiché il file system sa esattamente quali operazioni completare o annullare.

## → SICUREZZA DEI DATI: ELIMINAZIONE SICURA E CIFRATURA DEL DISCO

Cancellazione Convenzionale vs Eliminazione Sicura:

- 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 zero 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  
(Attenzione: molte scritture possono mettere a dura prova gli SSD)

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.

## INTRODUZIONE AI FILE SYSTEM VIRTUALI

Diversità dei File System:

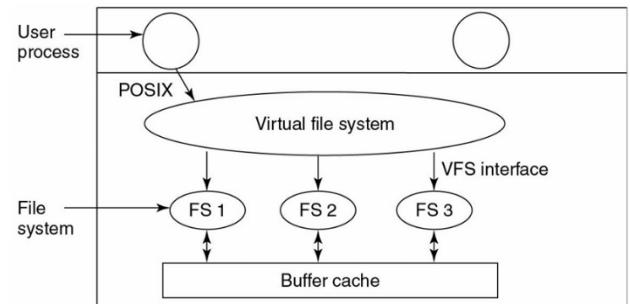
- Sistemi operativi moderni gestiscono diversi file system (NTFS, FAT-32, FAT-16, ecc.) simultaneamente.
- Windows utilizza lettere di unità (C:, D:, ecc.) per gestire file system differenti.
- Sistemi UNIX tentano di integrare più file system in una singola struttura gerarchica.

VFS (Virtual File System):

- 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.
- Gestisce file system locali e remoti, come quelli in NFS (Network File System).

Interfacce del VFS:

- Interfaccia superiore: Interagisce con le chiamate di sistema POSIX dei processi utente (es. *open*, *read*, *write*).
- Interfaccia inferiore: Composta da decine di funzioni che il VFS può inviare ai file system sottostanti.



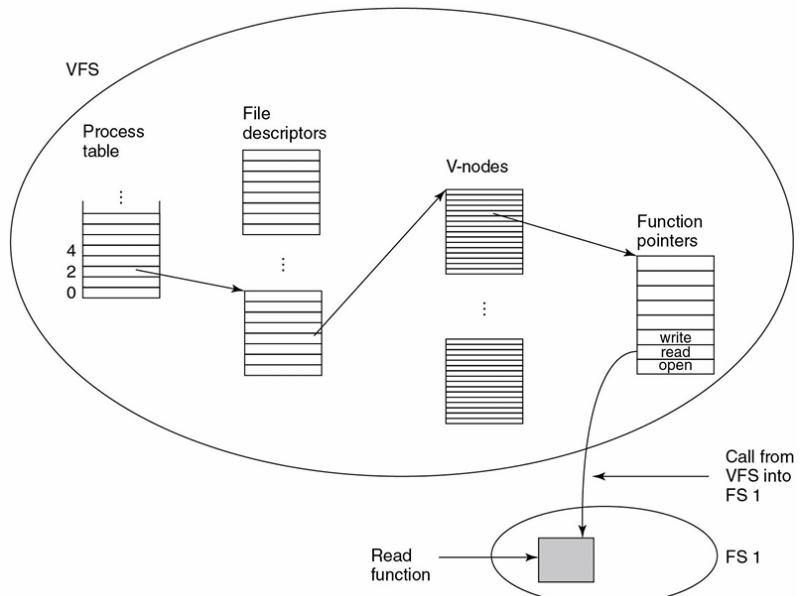
## → FUNZIONAMENTO E STRUTTURA DEL VFS: CONCETTI CHIAVE

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
- usato per identificare e interagire con il file system sottostante, facilitando l'accesso e la gestione delle sue risorse.

V-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

## → AGGIUNTA DI NUOVI FILE SYSTEM E VANTAGGI DEL VFS

Registrazione dei File System con il VFS: File system forniscono un vettore di funzioni 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.

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.

Progettisti devono fornire funzioni che rispettino l'interfaccia VFS. Il VFS rende possibile la gestione trasparente di file system eterogenei.

## RAID (SLIDE BONUS)

Non solo il SO garantisce l'affidabilità dei dati, ma anche l'hardware.

Storia e Sviluppo:

- RAID nato per migliorare le prestazioni dei sistemi di memorizzazione su dischi magnetici.
- Prima degli SSD, crescono esponenzialmente le prestazioni della CPU, ma non quelle dei dischi magnetici.

Definizione:

- RAID = Redundant Array of Inexpensive (poi Independent) Disks.
- Inizialmente contrastato dal concetto di SLED (Single Large Expensive Disk).

Funzionamento Base:

- Installazione di un contenitore di dischi accanto al computer.
- Utilizzo del controller RAID per migliorare prestazioni e affidabilità.
- Supporto sia per unità SCSI, SATA che SSD.

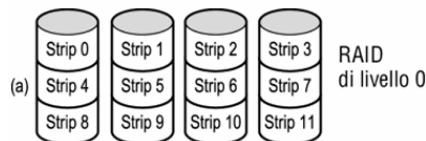
## LIVELLO RAID E LORO FUNZIONI

### a) RAID Level 0: + storage

Distribuzione dei dati in strip (strisce) su dischi multipli.

Migliora le prestazioni con richieste grandi.

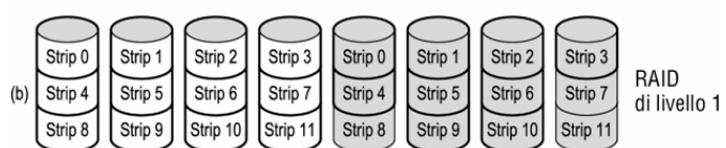
Nessuna ridondanza, quindi potenzialmente meno affidabile.



### b) RAID Level 1: + redundancy

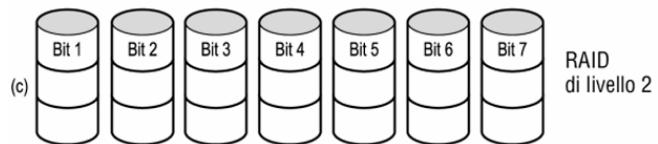
Duplicazione dei dischi per tolleranza agli errori.

Prestazioni di lettura migliorate, scrittura simile a un'unità singola.



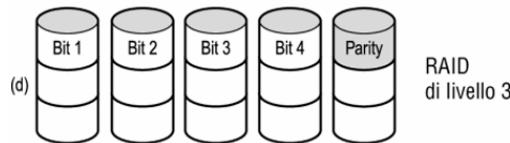
### c) RAID Levels2: + storage

Level 2 basato su parole o byte con codice di Hamming.



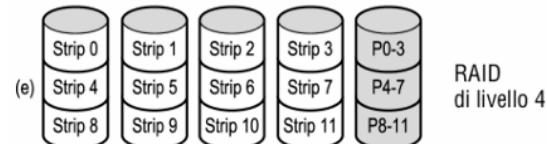
### d) RAID Levels3: + redundancy, +storage

Level 3 usa un singolo bit di parità per parola, richiede sincronizzazione delle unità.



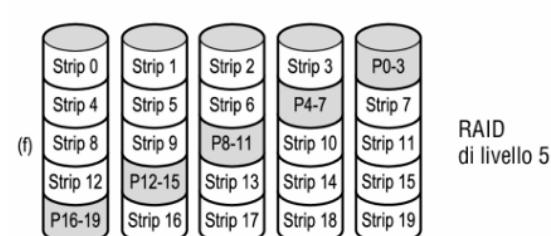
### e) RAID Level 4: +storage, +redundancy

Utilizzo di strip con un'unità extra per la parità.



### f) RAID Level 5: +storage, +redundancy

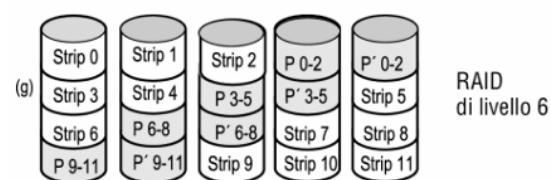
distribuisce bit di parità in modo uniforme su tutte le unità.



### g) RAID Level 6: +storage, +redundancy

Simile a Level 5 ma con un blocco di parità aggiuntivo.

Maggiore affidabilità e tolleranza agli errori.



Nota: è possibile combinare più schemi RAID

Esempio RAID 0+1: duplicazione ridondante secondo lo schema di RAID1 di dischi associati in schema RAID0

## MA ALLA FINE... QUALE FILE SYSTEM E' USATO?

Un po' di storia nascita e adozione dei file system:

1. File system V7 nel 1979,
2. Ext nel 1992,
3. Ext2 nel 1993,
4. Ext3 tra il 2003 e il 2006,
5. Ext4 nel 2008 (adottato da Google),
6. Uno sguardo al futuro: BTRFS.

Confronto tra BTRFS e EXT3/EXT4:

- Btrfs: File System Avanzato
  - 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/Ext4: Affidabilità e Stabilità
  - 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.

## VISUALIZZAZIONE DELLE PARTIZIONI E FILE SYSTEM CON MOUNT E ALTRI COMANDI

Scoprire Partizioni e File System Disponibili:

- Comando *lsblk*
  - o Mostra un elenco dei dispositivi di blocco, inclusi dischi e partizioni.
  - o Esempio: *lsblk* per visualizzare tutte le partizioni e i dispositivi.
- Comando *fdisk -l*
  - o Elenca dettagliatamente tutte le partizioni sui dischi, comprese quelle non montate.
  - o Richiede privilegi di root: *sudo fdisk -l*

Visualizzazione dei File System Montati:

- Comando *mount -l*
  - o Elenco dei file system attualmente montati con le loro opzioni di montaggio e etichette.
  - o 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.
- Cruciale per operazioni di backup, ripristino e manutenzione del sistema.

## MONTAGGIO DI PARTIZIONI E FILE SYSTEM CON IL COMANDO MOUNT

Montaggio di una Partizione/File System:

- Sintassi Base:
  - o *mount [opzioni] <dispositivo> <directory>*.
  - o 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 <tipo>*, es. *-t ext4*.
- Opzioni Aggiuntive: *-o <opzioni>*, es. *-o ro* per montaggio in sola lettura.

Smontaggio di un File System:

- Comando *umount*:
  - o Utilizzato per smontare in modo sicuro un file system o una partizione.
  - o Esempio: *sudo umount /mnt/mydisk*.

Note Finali:

- Queste operazioni richiedono privilegi di root.
- Smontaggio corretto è essenziale per prevenire la perdita di dati.

## **LEZ13**

### **TEMA LEZIONE = PRINCIPI DELL'HARDWARE DI I/O**

#### GESTIONE DELL'I/O NEL SISTEMA OPERATIVO

Funzione Principale:

1. Controllo dei dispositivi di I/O: invio di comandi, intercettazione di interrupt, gestione degli errori.
2. Fornire un'interfaccia semplice e uniforme tra dispositivi e il sistema (indipendenza dai dispositivi).

Importanza nel Sistema Operativo: Il codice per l'I/O costituisce una parte significativa del sistema operativo.

Argomenti fondamentali da sapere:

- 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

Diverse Prospettive:

- Ingegneri Elettronici: Vedono l'hardware di I/O in termini di componenti fisici (chip, cavi, alimentatori, motori, etc.).
- Programmatori: Interessati all'interfaccia software dell'hardware, inclusi i comandi accettati, le funzioni eseguibili e i possibili errori.
- Focus del Corso:
  - o Concentrazione sulla programmazione dei dispositivi di I/O, non sulla loro progettazione, costruzione o manutenzione.
  - o Interesse specifico nella interazione con l'hardware, piuttosto che nel suo funzionamento interno.

Connessione tra Programmazione e Operatività Interna: § La programmazione di molti dispositivi di I/O è spesso legata all'operatività interna dei dispositivi stessi.

### → DISPOSITIVI DI I/O

Categorie dei Dispositivi di I/O:

- Dispositivi a Blocchi: Archiviazione di informazioni in blocchi di dimensioni fisse (512 a 32.768 byte).
  - o Esempi: dischi fissi magnetici, unità SSD, unità a nastro magnetico.
  - o Caratteristica chiave: ogni blocco può essere letto o scritto indipendentemente.
- Dispositivi a Caratteri: Flusso di caratteri senza struttura a blocchi.
  - o Non indirizzabili, senza operazioni di ricerca.
  - o 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 il modello comunque utile per astrarre alcuni software di I/O nel sistema operativo.

Applicazioni nel Sistema Operativo:

- File system gestisce dispositivi a blocchi astratti.
- Software di livello inferiore gestisce le specificità dei dispositivi.

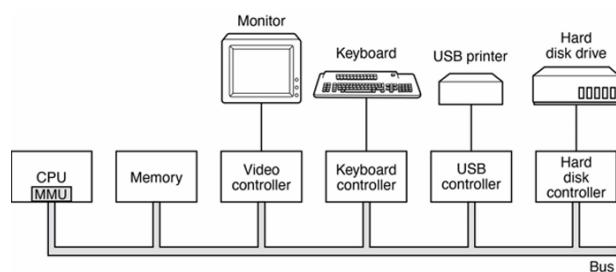
Nota: Dispositivi di I/O possono variare notevolmente in termini di velocità di trasferimento, creando sfide per il software di gestione.

Dispositivo	Velocità di trasferimento
Tastiera	10 byte/s
Mouse	100 byte/s
Modem a 56 K	7 KB/s

### → CONTROLLER DEI DISPOSITIVI

Panoramica generale:

- Componenti dei Dispositivi di I/O:
  - o Composti da una parte meccanica (il dispositivo stesso) e una elettronica (controller del dispositivo o adattatore).
  - o Controller spesso integrato nella scheda madre o come scheda aggiuntiva su slot PCIe.
- Connettività e Gestione Multi-dispositivo:
  - o Controller con connettori per il collegamento a dispositivi.
  - o Capacità di gestire più dispositivi identici.



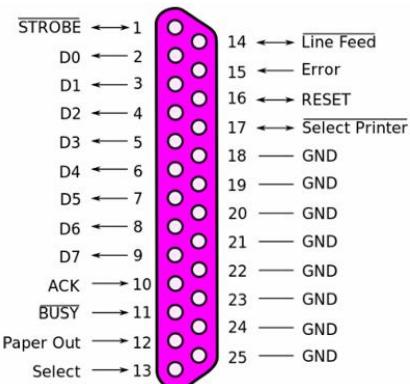
Funzionamento ed evoluzione:

- Standardizzazione dell'Interfaccia:
  - o Interfacce conformi a standard (ANSI, IEEE, ISO) o de facto (SATA, USB, ThunderBolt).
  - o Permette la produzione di dispositivi e controller compatibili da diverse aziende.
- Interfaccia di Basso Livello:
  - o Esempio: flusso seriale di bit in dischi, con preambolo, dati e ECC.
  - o Compito del controller: convertire il flusso seriale in blocchi di byte, correggere errori, trasferire in memoria principale.
- Impatto sulla Programmazione del Sistema Operativo:
  - o Senza il controller, il programmatore dovrebbe gestire dettagli complessi come la modulazione di ciascun pixel.
  - o Controller inizializzato con parametri essenziali, gestisce autonomamente dettagli complessi.

## → DALLA "VECCHIA" PORTA PARALLELA ALLA PORTA USB

Vecchia porta parallela:

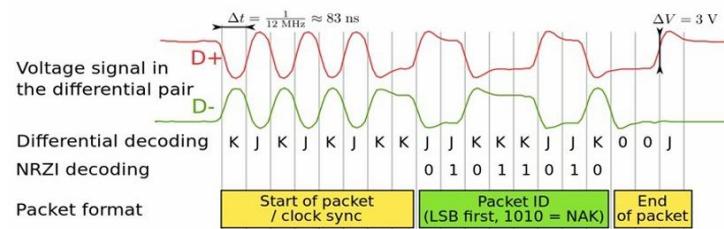
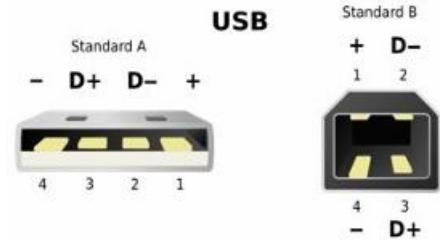
- Definizione e Utilizzo:
  - o Tipo di interfaccia di comunicazione tra computer e dispositivi (es. stampanti).
  - o Trasmette dati multi-bit simultaneamente su più canali (pin).
- Comunemente dotata di una connessione a 25 pin (standard DB-25) o 36 pin (Centronics).
- Distribuzione e Funzione dei Pin (Standard DB 25):
  - o Pin 1-8: Dati (D0-D7) - Trasmettono i dati effettivi.
  - o Pin 9-16: Controlli e Status - Includono pin di selezione della stampante, inizializzazione, linefeed, errori, occupato, ack, etc.
  - o Pin 17-25: Masse e alimentazione - Forniscono il ritorno elettrico e la connessione di terra.
- Caratteristiche Tecniche: Velocità variabile a seconda della modalità e del dispositivo collegato.



Porta USB:

- Definizione e Utilizzo:
  - o L'USB (Universal Serial Bus) è un'interfaccia standardizzata per la connessione e comunicazione tra dispositivi e computer.
  - o Utilizzata per trasferire dati e fornire alimentazione elettrica.
- Struttura dei Pin (Tipico USB 2.0):
  - o Pin 1 (Vcc): Alimentazione (+5V).
  - o Pin 2 (D-): Dati negativi.
  - o Pin 3 (D+): Dati positivi.
  - o Pin 4 (GND): Terra.

Pin	Name	Cable color	Description
1	VCC	Red	+5 VDC
2	D-	White	Data -
3	D+	Green	Data +
4	GND	Black	Ground



- Caratteristiche Tecniche:
  - o Supporta versioni USB 1.x, 2.0, 3.x, con velocità di trasferimento da 1,5 Mbps a 10 Gbps.
  - o USB 2.0 comunemente utilizzato per periferiche come tastiere, mouse, e dispositivi di archiviazione.
- Applicazioni e Vantaggi:
  - o Ampia applicazione in vari dispositivi come smartphone, periferiche di computer, e dispositivi di archiviazione.
  - o Facilità d'uso, connessione plug-and-play, supporto per trasmissione dati e alimentazione. § Compatibilità:
- Retrocompatibile con versioni precedenti, assicurando ampio supporto per vari dispositivi.

## → UNO SGUARDO AI CIRCUITI

- Il PCB(Printed Circuit Board) è realizzato in vetro intrecciato verde e rame, dotato di connettori SATA e di alimentazione.
  - o Supporta e collega i componenti elettronici dell'HDD.
  - o Comunica con il controller SATA e il BUS
- Al centro del PCB si trova il MCU (Micro Controller Unit), il chip più grande
  - o essenziale per le operazioni dell'HDD.
  - o Il MCU include una CPU (Central Processor Unit), un canale di lettura/scrittura per convertire i segnali analogici in digitali.
- Cache: La memoria è un chip di tipo DDR SDRAM.
  - o La sua capacità definisce la cache dell'HDD.
  - o Esempio: un chip da 32MB indica una cache teorica di 32MB.
- Il controller VCM (Voice Coil Motor) controlla la rotazione del motore del disco e i movimenti delle testine (consumando la maggior parte dell'energia elettrica sul PCB).
- Il chip flash memorizza parte del firmware dell'HDD, essenziale per l'avvio.

- Sensori di shock e diodi TVS (Transient Voltage Suppression) proteggono l'HDD da urti e sovratensioni. Il sensore di shock rileva urti eccessivi, inviando segnali al controller VCM per proteggere l'HDD. I diodi TVS proteggono da sovratensioni, sacrificandosi in caso di picchi di tensione.

## → COME FANNO CPU E DISPOSITIVO A COMUNICARE?

Input/output mappato in memoria

- Registri dei Controller
  - o Ogni controller di dispositivo ha registri per comunicare con la CPU.
  - o Scrittura nei registri: invia comandi al dispositivo (trasferimento dati, accensione/spegnimento, altre azioni).
  - o Lettura dai registri: verifica lo stato del dispositivo e la prontezza per nuovi comandi.
- Buffer di Dati:
  - o Molti dispositivi includono un buffer di dati per scrittura/lettura del sistema operativo.
  - o Esempio: RAM video utilizzata per visualizzare punti sullo schermo.
- Comunicazione CPU-Dispositivo: Come la CPU comunica con i registri di controllo e i buffer dei dati?

Due approcci principali:

### 1. PORT-MAPPED I/O (PMIO)

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).

Separazione: Spazi di indirizzi della memoria e dell'I/O sono distinti e non correlati (a).

- 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) riferisce a spazi di indirizzi diversi (primo I/O, secondo di memoria)



### 2. MEMORY-MAPPED I/O (MMIO)

Approccio I/O Mappato in Memoria:

- Introdotta con il PDP-11, questa metodologia assegna a ogni registro di controllo un indirizzo di memoria univoco.
- Registri di controllo mappati nello spazio della memoria ( b ).

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.
- Protezione semplificata: i processi utente non accedono direttamente ai registri di controllo. Il sistema operativo utilizza la gestione della memoria per proteggere gli indirizzi dei registri hardware, rendendoli accessibili solo al kernel.

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.

Spazio degli indirizzi singolo



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 dal device
- Caso limite: se un ciclo while è in attesa che il contenuto del registro cambi: ciclo infinito

(b)

- Necessità di disabilitare selettivamente la cache per alcuni indirizzi (complessità aggiuntiva). Bisogna identificare con precisione quali pagine sono dedicate ai dispositivi hardware. Gli accessi a queste aree non saranno ottimizzati dalla cache, quindi possono essere più lenti.

#### Gestione degli Indirizzi e Architetture del Bus:

- Necessità per tutti i moduli di memoria e dispositivi di I/O di esaminare ogni riferimento alla memoria.
- Problemi con bus della memoria separati in architetture come quelle x86 con bus multipli (memoria, PCIe, SCSI, USB).

#### Soluzioni:

- Tentativi sequenziali (Memory-first): La richiesta è indirizzata prima alla memoria principale. Se fallisce, viene inoltrata ad altri bus.  
Pro: Semplice da implementare.  
Contro: Maggiore latenza per gli accessi I/O.
- Bus Snooping ("Spia"): Un dispositivo sul bus monitora gli indirizzi e reindirizza quelli destinati ai dispositivi I/O.  
Pro: Accessi I/O più rapidi.  
Contro: Maggiore complessità hardware.

#### Prestazioni della Memoria:

- Il bus della memoria è ottimizzato per velocità, ma deve gestire sia gli accessi alla memoria principale che ai dispositivi.
- Tentativi sequenziali preservano le prestazioni della memoria (dando priorità agli accessi alla memoria principale).

#### Complessità nella Gestione dell'I/O:

- Dispositivi come "bus snooping" riducono la latenza I/O, ma introducono complessità hardware e software.
- La gestione avanzata richiede coordinamento tra memoria e dispositivi su bus separati.

Conclusione: L'I/O mappato in memoria richiede un bilanciamento tra:

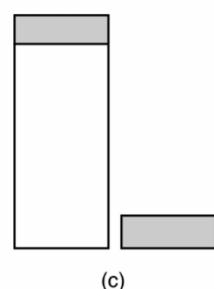
- Prestazioni: Minimizzare i ritardi negli accessi a memoria e dispositivi.
- Complessità: Mantenere un design efficiente senza overhead eccessivo.

### 3. APPROCCIO IBRIDO (Port-mapped I/O + Memory-mapped I/O) (bonus)

Cos'è un approccio ibrido? Combina Port-Mapped I/O (PMIO) e Memory-Mapped I/O (MMIO):

- PMIO: Utilizza un indirizzamento separato per i dispositivi I/O con istruzioni dedicate (IN e OUT).
- MMIO: I registri dei dispositivi sono mappati nello spazio della memoria, accessibili con normali istruzioni di memoria (LOAD, STORE).

Due spazi degli indirizzi:



#### Come funziona?

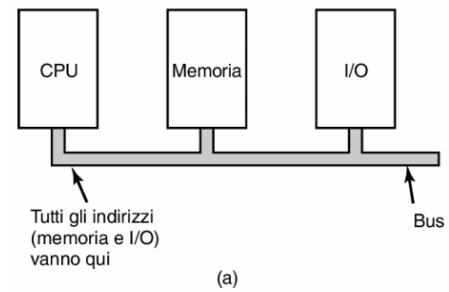
- Configurazione tramite PMIO: Configurazione iniziale di dispositivi hardware (es. attivazione o setup di registri di base). Usa lo spazio di indirizzi delle porte e istruzioni IN/OUT.
- Accesso ai dati tramite MMIO: Operazioni ad alta velocità come trasferimenti di dati (es. schede grafiche, controller di rete). I dispositivi sono mappati nello spazio di memoria principale.

#### Vantaggi:

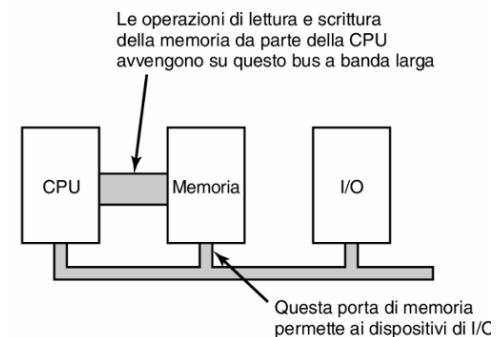
- Flessibilità: PMIO per configurazioni semplici, MMIO per operazioni rapide. Accesso diretto tramite istruzioni di memoria, Nessun overhead dovuto al cambio di modalità, Cache e parallelismo
- Ottimizzazione delle prestazioni: MMIO è più veloce per trasferimenti dati continui.
- Compatibilità legacy: PMIO consente di supportare dispositivi più vecchi.
- Separazione logica: Configurazione e utilizzo sono gestiti in modi distinti.

#### Svantaggi:

- Aumento della complessità: Due modalità richiedono logiche hardware e software più complesse.



(a)



(b)

- ❖ Overhead iniziale: PMIO può rallentare la configurazione rispetto a un approccio puramente MMIO.
- ❖ Limitazioni architettoniche: Alcune CPU moderne (es. ARM) non supportano PMIO.

Esempio pratico:

- x86 § PMIO per configurare i dispositivi PCIe (es. usando porte 0xCF8/0xCFC).
- MMIO per accedere ai registri del dispositivo PCIe dopo la configurazione.

- 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.

### IN ATTESA DI RICHIESTE DI I/O...

Ora che possiamo inviare comandi ai dispositivi, ma 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).
- È una buona soluzione?

### → DIRECT MEMORY ACCESS (DMA)

Definizione e Necessità del DMA:

- DMA permette alla CPU di scambiare dati con i controller dei dispositivi bypassando il trasferimento manuale byte per byte.
- Riduce lo spreco di tempo della CPU, migliorando l'efficienza del trasferimento dati.

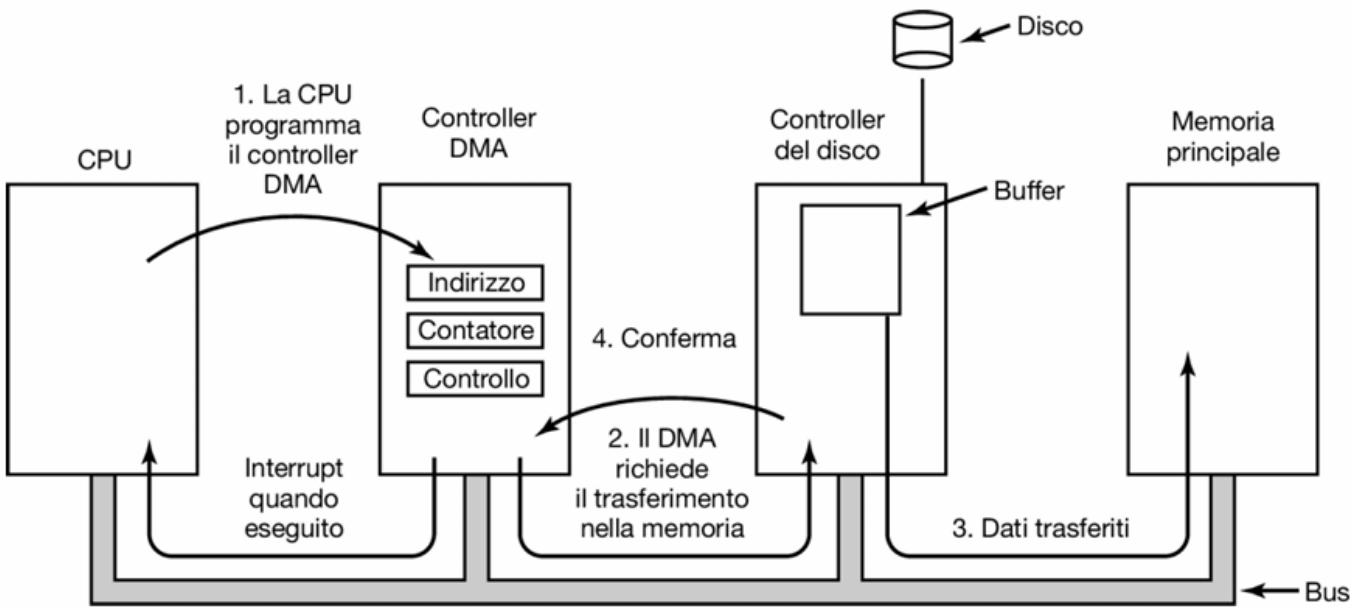
Configurazione Hardware:

- Presenza di un controller DMA in molti sistemi, a volte integrato nei controller di dispositivi.
- Il controller DMA può gestire trasferimenti a più dispositivi, spesso situato 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.
- Con DMA: La CPU impone il controller DMA e invia un comando al controller del disco (Passo 1).



Funzionamento dettagliato del DMA:

- 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.

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.
- 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. 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.
- Modalità 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.
- Fly-by Mode: DMA trasferisce dati direttamente alla memoria principale senza intermediari §
  - o Riducendo l'uso del bus ma richiedendo un ciclo extra per ogni trasferimento.
  - o Richiede un ciclo aggiuntivo per ogni trasferimento diretto, poiché il DMA deve gestire sia il dispositivo sorgente sia la memoria di destinazione in tempo reale.
- Gestione degli Indirizzi: DMA utilizza tipicamente indirizzi fisici, richiedendo conversione da parte del sistema operativo.
- Alcune periferiche (come il disco) hanno un Buffer Interno; perché? 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.

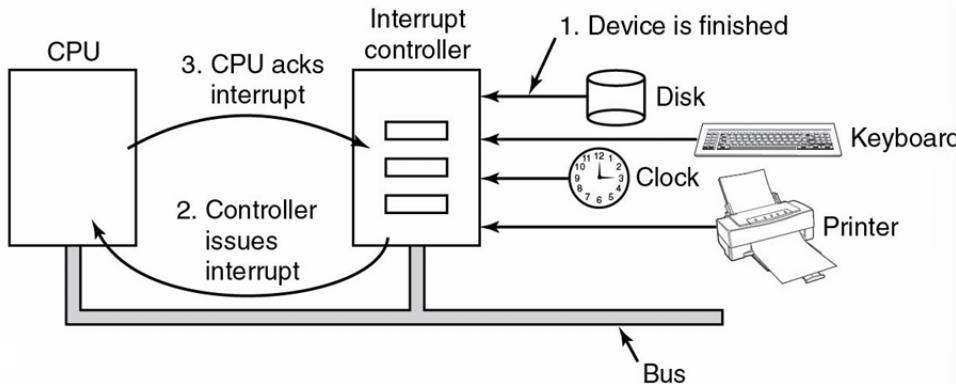
## → INTERRUPT NEL SISTEMA OPERATIVO

### 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 degli Interrupt Hardware: Un dispositivo di I/O invia un segnale di interrupt alla CPU tramite una linea del bus assegnata. Gestito dal chip del controller degli interrupt sulla scheda madre.

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.



## → PROCESSO DI 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: Punti all'inizio della procedura di servizio degli interrupt corrispondente. 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 E PROBLEMI RELATIVI

Salvataggio dello Stato: Al minimo, il contatore di programma deve essere salvato per riavviare i processi interrotti. Alcune CPU salvano tutti i registri visibili e interni.

Dove Salvare le Informazioni prelevate dai dispositivi di I/O: Salvataggio nei registri interni o sullo stack. Problemi con i registri interni: tempi morti lunghi e rischio di sovrascrittura.

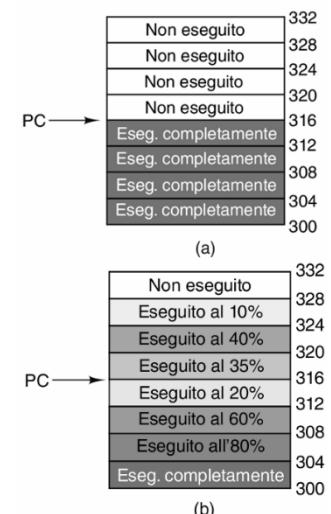
Uso dello Stack: Uso dello stack corrente (processo utente) o dello stack del kernel.

- 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.

## → INTERRUPT PRECISI VS IMPRECISI – CONTESTO E DIFFERENZE FONDAMENTALI

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 PRECISI – DEFINIZIONE E CARATTERISTICHE

Caratteristiche di un 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 – COMPLESSITÀ E IMPLICAZIONI

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.
- Gli interrupt imprecisi possono avere implicazioni di sicurezza poiché non tutti gli effetti sono completamente annullati.

## LEZ14

### TEMA LEZIONE = LIVELLI DEL SOFTWARE DI I/O

#### OBIETTIVI 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.
  - o /dev/sda è meglio
  - o /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 sembranti bloccanti, ma fornisce anche l'accesso all'I/O asincrono per applicazioni ad alte prestazioni.
- Il sistema operativo deve gestire DMA

Buffering:

- Spesso i dati da un dispositivo non vanno direttamente alla destinazione finale, richiedendo un buffer temporaneo.
  - o Un pacchetto che arriva su un'interfaccia di rete deve essere ricevuto e analizzato prima di capire quale applicazione (esempio browser) può usarlo
  - o Un segnale audio deve essere posizionato preventivamente in un buffer per evitare interruzioni
- 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.

#### TIPOLOGIE DI SW PER I/O

Le tipologie di software per I/O sono le seguenti:

##### 1) I/O PROGRAMMATO

Definizione di I/O Programmato: la CPU gestisce direttamente tutto il processo di trasferimento dati.

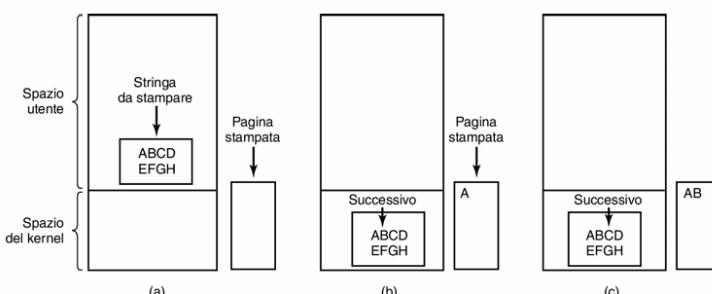
Esempio Pratico con Riferimento:

- 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 entra in un ciclo di polling, controllando il registro di stato della stampante e inviando un carattere alla volta.



Esempi di codice dell'I/O programmato:

```
copy_from_user(buffer, p, count);
for (i = 0; i < count; i++) {
    while (*printer_status_reg != READY) ;
    *printer_data_register = p[i];
}
return_to_user();
```

/\* p è il buffer del kernel \*/
/\* ripeti per tutti i caratteri \*/
/\* ripeti finché lo stato diventa READY \*/
/\* invia in output ogni carattere \*/

L'esempio di codice 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.
- 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.
- Ricerca di metodi di I/O che liberino la CPU da costanti attività di polling.

## 2) I/O GUIDATA DAGLI INTERRUPT (PROCESSO DI STAMPA)

Vediamolo in un processo di stampa:

- 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.
- 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!

```
/* copia dati dall'utente al kernel.*/
copy_from_user(buffer, p, count);

/*abilita gli interrupt*/
enable_interrupts();

/*attende che la stampante sia pronta
 a ricevere caratteri */
while (*printer_status_reg != READY) ;

/*invia il primo carattere alla stampante.*/
*printer_data_register = p[0];

/*passa il controllo a un altro processo.*/
scheduler();

Codice eseguito al momento della chiamata
di sistema per la stampa.
```

```
/* Controlla se tutti i caratteri sono stati stampati. */
if (count == 0) {
    /* Sblocca il processo utente che ha richiesto la stampa */
    unblock_user();
} else {
    /* Invia il carattere successivo alla stampante. */
    *printer_data_register = p[i];
    /* Decrementa il conteggio dei caratteri rimanenti. */
    count = count - 1;
    /* Passa al carattere successivo nel buffer. */
    i = i + 1;
}

/* Riconosce l'interrupt ricevuto dalla stampante. */
acknowledge_interrupt();

/* Ritorna dall'interrupt, permettendo alla CPU di riprendere altre
operazioni. */
return_from_interrupt();
```

Procedura di servizio interrupt per la stampante.

### 3) I/O CON DMA (EFFICIENZA E GESTIONE DEI PROCESSI)

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):
  - o Preparazione dei Dati: `copy_from_user`.
  - o Configurazione DMA: `set_up_DMA_controller`.
  - o Ottimizzazione delle Risorse CPU: `scheduler`.
- Gestione dell'Interrupt e Conclusione (b):
  - o Gestione dell'interrupt generato dal completamento del trasferimento DMA: `acknowledge_interrupt`.
  - o Ripresa del Processo Utente: `unblock_user`.
  - o Ritorno dal Contesto dell'Interrupt: `return_from_interrupt`.

```

/* Copia i dati dall'utente al kernel. */
copy_from_user(buffer, p, count);

/* Impostazione del controller DMA per il
trasferimento */
set_up_DMA_controller();

/* La CPU esegue altri processi mentre il DMA
gestisce il trasferimento. */
scheduler();
```

(a)

```

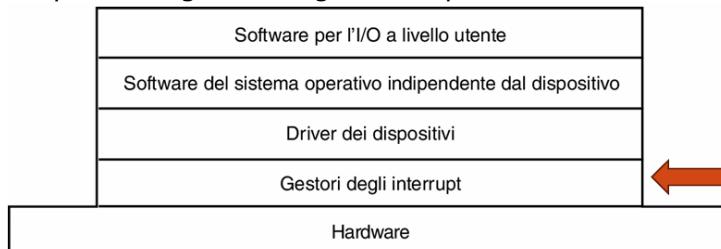
/* Riconosce l'interrupt ricevuto dal DMA. */
acknowledge_interrupt();

/* Sblocca il processo utente dopo che il
trasferimento è completo. */
unblock_user();

/* Ritorna dall'interrupt, consentendo alla CPU di
proseguire con altre operazioni. */
return_from_interrupt();
```

(b)

Dove mi trovo? Mi trovo nella parte sui gestori degli interrupt:



### GESTIONE DEGLI INTERRUPT NEL SISTEMA OPERATIVO

**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.

**Gestione Complessa:** La gestione degli interrupt richiede diversi passaggi, inclusi salvataggio dei registri, impostazione di contesti e conferme al controller degli interrupt.

**Impatto della Memoria Virtuale:** 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.

**Elaborazione Non Banale:** L'elaborazione di un interrupt 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 Tavola 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.

Dove mi trovo? Mi trovo nella parte sui gestori degli interrupt:



## DRIVER DI DISPOSITIVO

Ruolo dei Driver di Dispositivo: Gestiscono i dispositivi di I/O attraverso registri di dispositivi specifici.

- 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).
  - 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.

- 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.

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).

## → FUNZIONALITÀ E INTERFACCIA DEI DRIVER DI DISPOSITIVO

**Interfaccia con il Sistema Operativo:** Il sistema operativo deve permettere l'installazione di codice scritto da terze parti (driver).

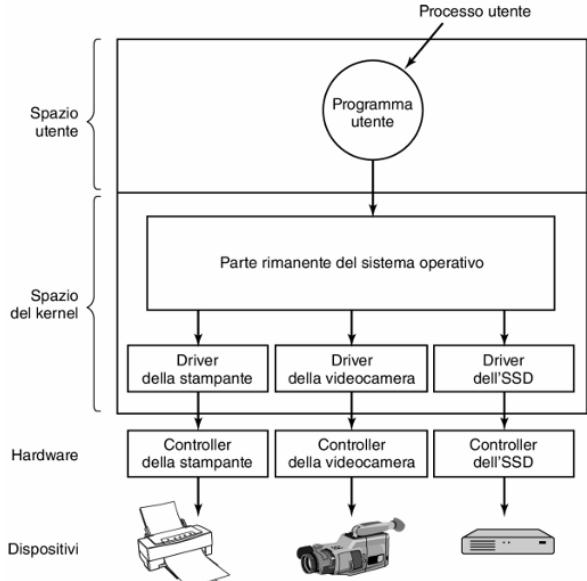
- I driver si posizionano sotto il resto del sistema operativo.
  - Ogni categoria ha un'interfaccia standard che i driver devono supportare.

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. Aggiunto un dispositivo nuovo il kernel andava ricompilato!!!
  - Nei sistemi moderni, i driver vengono caricati dinamicamente.



#### → IMPLEMENTAZIONE E COMPLESSITÀ' DEI DRIVER DI DISPOSITIVO

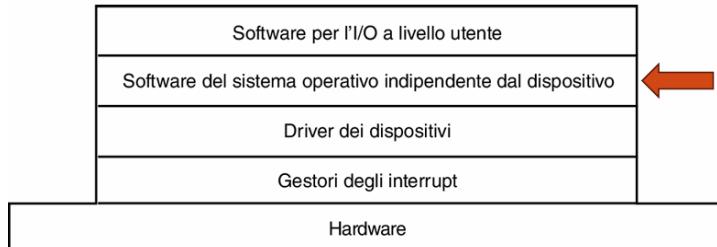
Funzioni dei Driver: Gestione di letture e scritture, inizializzazione del dispositivo, gestione dell'alimentazione e del registro degli eventi.

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.

Gestione dell'I/O e Errori: 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).  
**Gestione delle 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).

Dove mi trovo? Mi trovo nella parte sui gestori degli interrupt:



## SOFTWARE DI I/O INDIPENDENTE DAL DISPOSITIVO

Ma il software per I/O dipende sempre 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.

### → UNIFORMITA' NELL'INTERFACCIA DEI DRIVER DEI DISPOSITIVI

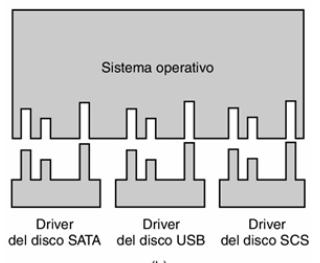
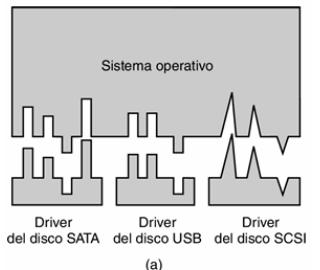
Necessità di Uniformità:

- Evita la necessità di modificare il sistema operativo ogni volta che viene introdotto un nuovo dispositivo.
- Importante per mantenere la consistenza e l'efficienza nel sistema.

Interfacce Diverse (a) vs Interfaccia Standard (b):

- Problema con diversi driver aventi interfacce uniche verso il sistema operativo.
- Soluzione: un modello uniforme dove tutti i driver condividono la stessa interfaccia.

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).



### → IMPLEMENTAZIONE E GESTIONE DELL'INTERFACCIA DEI DRIVER

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.

Uniformità nella Denominazione e Protezione dei Dispositivi:

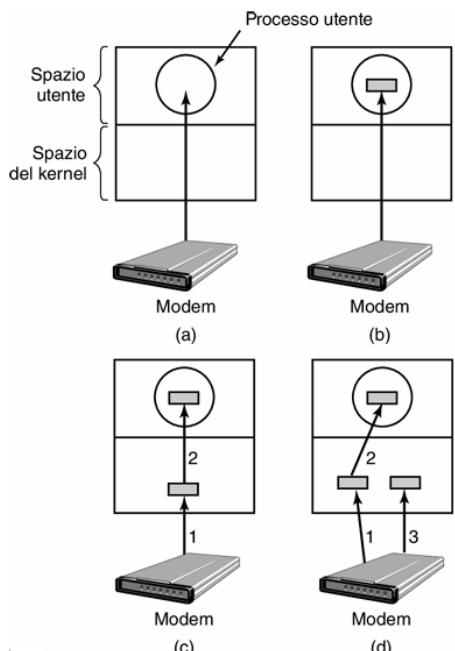
- Mappatura dei nomi simbolici dei dispositivi ai driver corrispondenti (es. /dev/disk0 in UNIX).
- Gestione dei permessi e protezione dei dispositivi simile a quella dei file, consentendo un controllo amministrativo appropriato.

Interfaccia e Integrazione nel Sistema: Questa struttura fornisce un'interfaccia coesa fra i driver e il resto del sistema operativo, semplificando l'integrazione di nuovi dispositivi.

## → BUFFERING E LA SUA NECESSITA' NEI DISPOSITIVI DI I/O

Scenari di Buffering in Input:

- Esempio di lettura dati da un modem VDSL: l'input senza uso di buffer (a) è inefficiente poiché richiede un riavvio del processo utente per ogni carattere ricevuto.
- Miglioramento con buffer nello spazio utente (b): il processo fornisce un buffer e si blocca solo quando è pieno.
- Problemi di paginazione e soluzione con buffer nel kernel (c): il buffer nel kernel accumula i caratteri, riducendo il riavvio del processo utente.



Doppio Buffer nel Kernel (d):

- 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 E LA SUA COMPLESSITA'

Buffering in Output:

- Esempio di output su un modem: senza buffering (analogamente a b), il processo utente può restare bloccato per lungo tempo.
- Soluzione con 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
- Processo di 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.
- Sequentialità delle operazioni di copia aumenta il tempo complessivo di trasmissione.

## → GESTIONE DEGLI ERRORI DI I/O NEL SISTEMA OPERATIVO

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.
- Veri errori di I/O (es. scrittura su un blocco danneggiato): Gestiti dal driver o, se non risolvibili, passati al software indipendente dal dispositivo.

Azioni Dipendenti dal Contesto:

- In presenza di un utente interattivo: 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.

Gestione degli Errori Critici: In caso di danneggiamento di strutture dati critiche: Visualizzazione di un messaggio d'errore e possibile terminazione del sistema.

## → GESTIONE DEI DISPOSITIVI DEDICATI NEL SISTEMA OPERATIVO

Uso Esclusivo di Alcuni Dispositivi: Dispositivi come stampanti richiedono l'uso esclusivo da un singolo processo alla volta.

Gestione delle Richieste di Uso: 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: Meccanismi speciali per richiedere e rilasciare dispositivi: un tentativo di acquisizione non riuscito causa il blocco del processo richiedente. I processi bloccati sono inseriti in una coda e acquisiscono il dispositivo quando diventa disponibile.

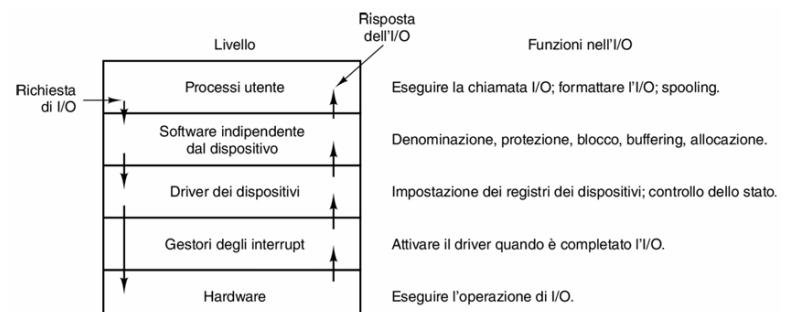
## → SISTEMA DI SPOOLING PER DISPOSITIVI DEDICATI

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, come mostrato in Figura, per ordinare e gestire i lavori di stampa.

Benefici del Spooling:

- Incrementa l'efficienza nell'uso dei dispositivi dedicati
- migliora la gestione delle risorse
- permettendo a più utenti o processi di accedere ai dispositivi in modo controllato e sequenziale.



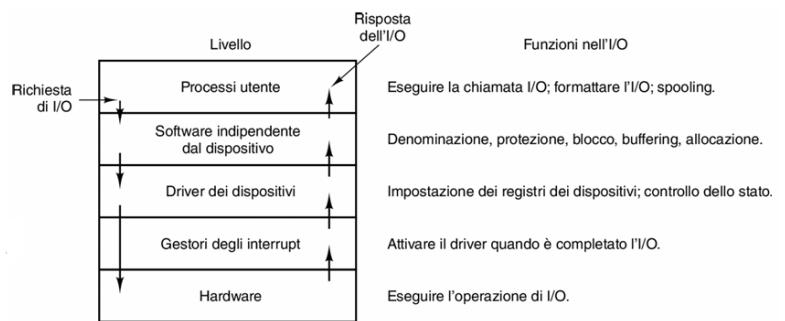
## → FLUSSO DEL CONTROLLO NEL SISTEMA DI I/O

Livelli del Sistema di I/O: Dall'hardware ai processi utente, come rappresentato in Figura.

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.



## → UNIFORMITA' NELLA DIMENSIONE DEI BLOCCHI NEI DISPOSITIVI 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.

Ruolo del Software Indipendente dal Dispositivo:

- Nasconde le differenze fisiche nelle dimensioni dei blocchi o settori tra diversi dispositivi.
- Fornisce una dimensione di blocco logico uniforme ai livelli superiori del sistema.

Creazione di Dispositivi Astratti:

- Trasforma più settori o pagine flash in un unico blocco logico.
- Permette ai livelli superiori di interagire con dispositivi "astratti" che utilizzano una dimensione di blocco logico standard, indipendentemente dalle dimensioni fisiche.

Occultamento delle Differenze nei Dispositivi a Caratteri: Gestisce la varianza nella quantità di dati trasmessi dai dispositivi a caratteri (es. mouse vs interfacce di rete), rendendo queste differenze trasparenti ai livelli superiori.

## → LIBRERIE DI I/O NELL'AMBITO UTENTE

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.

Importanza nelle Applicazioni: Queste librerie semplificano la programmazione di I/O, permettendo ai programmatore di concentrarsi sulla logica dell'applicazione piuttosto che sui dettagli di basso livello delle operazioni di I/O.

PER ESAME:

**Appunti del 17/12/2024 sull'esame (condivisa prova d'esame)**

- 1) Risposta multipla (10 domande in 30 minuti)
  - Quasi sempre solo 1 risposta esatta, a volte due (nella singola domanda)
- 2) Esercizio programmazione
  - Argomenti: Programmazione concorrente, mutex, creazione processi, scrittura su file, gestione di più thread, gestione di processi padre/figli etc...
- 3) Domanda aperta  
Porta pc e credenziali teams

**FONDAMENTALE SPECIFICATO DAL PROFESSORE AL TERMINE DELLA LEZIONE DEL 12/12/2024:**

- ESERCIZIO PROGRAMMAZIONE CONCORRENTE ALL'ESAME!
- POTREBBERO ESSERCHI THREAD
- PTREBBERO ESSERCI FILE

⇒ **IN C:**

IMPARA COME DISABILITARE SIGINT

IMPARA COME FAR COMUNICARE PROCESSI

SCRITTURA SU FILE