Sistemi Operativi

Sicurezza dei Sistemi e delle Reti Informatiche
Anno Accademico 2024/2025

Nicola Iantomasi
nicola.iantomasi2020@gmail.com

Abstract

Questa dispensa, basata sul libro consigliato e sulle lezioni del Prof. Andrea Lanzi per il corso di Sistemi Operativi nel CdL in Sicurezza dei Sistemi e delle Reti Informatiche (SSRI), tratta la gestione dei processi, la virtualizzazione, la concorrenza, la sicurezza e la persistenza dei dati. Approfondisce lo scheduling della CPU, la gestione della memoria, la sincronizzazione tra processi e thread, i file system e le strategie di protezione dei sistemi operativi, con particolare attenzione alle vulnerabilità e alle mitigazioni di sicurezza. Pensata come supporto didattico, mira a fornire una comprensione chiara e applicata delle tematiche affrontate.

Dichiarazione sulla fonte

Questa dispensa si basa principalmente sul libro **Operating Systems: Three Easy Pieces**, scritto da Remzi H. Arpaci-Dusseau e Andrea C. Arpaci-Dusseau, pubblicato da Arpaci-Dusseau Books, LLC, nel 2018 [1]. Il materiale presentato in questa dispensa, inclusi frammenti di testo, diagrammi, grafici e tabelle, è tratto da questo libro, salvo diversa indicazione.

Per maggiori informazioni sul libro, si consiglia di consultare la versione ufficiale del testo: «*Arpaci-Dusseau, Remzi H., and Andrea C. Arpaci-Dusseau.* **Operating Systems: Three Easy Pieces**. Arpaci-Dusseau Books, LLC, 2018.»

Il materiale, inclusi frammenti di codice, grafici e tabelle, è tratto dal libro sopra citato e viene utilizzato a scopo educativo, per scopi di apprendimento e studio. Si ritiene che l'uso di questo materiale rientri nell'ambito delle normative sul Fair Use, in quanto non ha scopi commerciali e viene utilizzato in una quantità limitata per fini didattici. Si invita comunque a consultare il libro per una comprensione completa e dettagliata degli argomenti trattati.

Se desiderate ottenere una copia completa del libro o ulteriori dettagli, si invita a visitare il sito ufficiale: http://pages.cs.wisc.edu/~remzi/OSTEP/.

Contents

Introduzione ai sistemi operativi

Operative System (OS):
- Responsabile dell'esecuzione dei programmi.
- Consente ai programmi di condividere memoria.
- Garantisce il corretto e efficiente funzionamento del sistema, rendendo l'uso facile per l'utente.

Processi

Un processo, informalmente, altro non è che un programma in esecuzione. Un programma, a sua volta, è una sequenza finita di istruzioni scritte in un linguaggio comprensibile all'esecutore (nel nostro caso la CPU). L'esecuzione di un programma da parte del processore è concettualmente semplice:

1. **Fetch**: viene prelevata l'istruzione dalla memoria.
2. **Decode**: viene decodificata l'istruzione per renderla comprensibile alla cpu.
3. **Execute**: viene eseguita l'istruzione, ora in linguaggio macchina.

Virtualizzazione

Per svolgere i suoi compiti, l'OS utilizza la **virtualizzazione**, che astrae le risorse fisiche in risorse più generali. L'OS è anche chiamato macchina virtuale, e l'utente può interagire con esso tramite API o **System Calls** disponibili alle applicazioni. Fornisce librerie standard per gestire le risorse. La virtualizzazione permette l'esecuzione di molti programmi, e l'OS si occupa di gestire le risorse condivise tra tutti i programmi.

Tipi di Virtualizzazione:

- **Virtualizzazione della CPU**: Consiste nel trasformare una CPU fisica in un numero apparentemente infinito di CPU virtuali, consentendo ai programmi di funzionare in parallelo, apparentemente in contemporanea.

- **Virtualizzazione della memoria**: La memoria nelle macchine moderne è vista come un array di byte. Per accedere alla memoria, è necessario specificare un indirizzo. Inoltre, per scrivere in memoria, oltre all'indirizzo, è necessario specificare anche i dati. Ogni processo ha il proprio spazio privato di indirizzi virtuali che l'OS mappa sulla memoria fisica.

Policies: Un insieme di regole che governano il comportamento dell'OS e delle applicazioni che lo eseguono.

Mechanisms: Procedure o strumenti specifici tramite i quali le politiche vengono attuate o implementate.

Concurrency

La concorrenza è la capacità di eseguire più programmi contemporaneamente, utile per migliorare le prestazioni eseguendo più processi o thread in parallelo.

Thread: Una funzione in esecuzione all'interno dello stesso spazio di memoria di altre funzioni, che permette più attività contemporaneamente.

Persistence

La persistenza è la capacità di mantenere le informazioni anche dopo che il computer è stato spento. Questo viene fatto salvando le informazioni su dispositivi di archiviazione persistenti, come un disco rigido o un'unità flash.

Sicurezza

Un modello di protezione implementato dal sistema operativo è quello ad anelli. Vengono creati 5 livelli differenti e tre anelli differenti. A ciascun anello corrisponde un relativo livello di sicurezza. Solo chi ha i permessi per operare in un certo anello può eseguire codice in esso.

- **Level 1, hardware level**: qui vengono eseguiti, ad esempio, i device drivers visto che essi richiedono accesso diretto all'hardware dei dispositivi. Questo dispositivo è quindi il microcontroller (es: motherboard chipset, disk drivers, SATA, IDE, ard drives, processor, ecc… tutti questi hanno controller che risiedono quindi al livello 1) che controlla fisicamente un device.
- **Level 2, firmware level**: il firmware sta in cima al livello elettronico. Contiene il software necessario dal dispositivo hardware e dal micocontroller. Questo livello contiene quindi firmware sviluppato in microcode. Il microcode viene generalmente memorizzato in una memoria ROM.
- **Level 3, ring 0 o kernel level**: è il livello dove opera il kernel, dopo quindi la fase di bootload siamo qui.
- **Level 4, ring 1 e 2 o device drivers**: i device drivers passano attraverso il kernel per accedere all'hardware. Queste porzioni di codice per funzionare bene hanno bisogno di molta libertà senza però che essi possano andare a modificare componenti che causerebbero un crash (come ad esempio la GDT), questo è il motivo per cui sono eseguiti al ring 1 e 2 e non al ring 0.
- **Level 5, ring 3 o application level**. Qui è dove viene eseguito normalmente il codice utente, attraverso l'uso delle API del sistema o le interfacce driver.

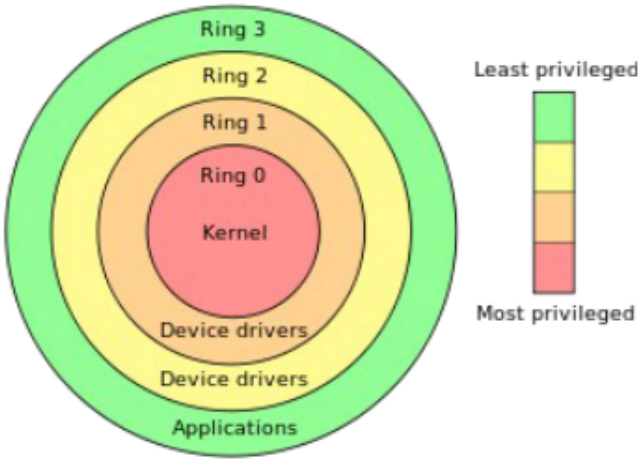


Figura 1: Livelli di sicurezza.

I Virtualizzazione

1 Processi e gestione dei processi

Multiprogrammazione: migliora l'utilizzo della CPU e delle risorse del sistema, consentendo l'esecuzione simultanea di più processi. In un ambiente multiprogrammato, quando un processo è in attesa di risorse (ad esempio, I/O), la CPU viene assegnata ad un altro processo pronto per l'esecuzione. Questo approccio massimizza l'efficienza, riduce i tempi di inattività e aumenta la produttività complessiva del sistema.

Introduciamo alcuni concetti fondamentali della multiprogrammazione:
- **Time sharing**: prevede che il tempo di CPU sia "equamente" diviso fra i programmi in memoria. In questo modo sono in grado di risolvere i lunghi tempi di risposta del modello batch.
- **Real time sharing**: nei sistemi che adottano questa soluzione, la politica di scheduling (scelta del processo da eseguire) è differente. Alcuni processi vanno serviti prima di altri, basti pensare al programma che consente di controllare il volo di un aereo il quale avrà priorità massima (hard real time sharing). Il programma che regola il ritiro al bancomat ha priorità più bassa (soft real time sharing).

Processo: Un'istanza in esecuzione di un programma. Ogni processo ha il proprio spazio di indirizzo virtuale, il proprio set di registri e le proprie risorse di sistema. I processi sono gestiti dall'OS, che assegna loro le risorse di sistema e ne aggiorna lo stato.

Process state: Indica la fase corrente in cui si trova il processo in esecuzione. Comprende tutto ciò che può leggere e aggiornare un programma, inclusa la memoria del processo, lo spazio di indirizzamento, i registri, l'heap, lo stack e il codice.

1.1 Virtualizzazione CPU

L'OS virtualizza la CPU, in modo che il processo creda di avere più CPU a sua disposizione. Questo consente di dare l'impressione che più programmi vengano eseguiti contemporaneamente, mentre in realtà l'OS sta eseguendo e interrompendo i processi in sequenza.

Policies: Regole definite per prendere decisioni. Definiscono "cosa è necessario fare".

Mechanisms: Metodi o protocolli di basso livello che implementano una funzionalità. Determinano "come fare qualcosa".

Esempi:
- **Context switch** (meccanismo di time-sharing): Offre all'OS la possibilità di interrompere l'esecuzione di un programma e iniziare a eseguirne un altro sulla stessa CPU.
- **Scheduling policy**: dato un numero n di programmi, definisce quale verrà eseguito per primo, determinando l'ordine di esecuzione.

Meccanismi vs Policies: La differenza principale è che i meccanismi sono i mezzi per implementare una politica, mentre le politiche sono le regole che determinano come i meccanismi vengono utilizzati.

Process Control Block (PCB): Una struttura dati che memorizza informazioni dettagliate per singolo processo. Include:
- Process ID (PID)
- Stato del processo (running, ready, waiting, etc.)
- Program Counter
- Registri della CPU
- Puntatore allo stack kernel
- Informazioni di scheduling
- Priorità del processo
- Puntatori alla memoria
- Informazioni sui file aperti
- Statistiche di utilizzo delle risorse

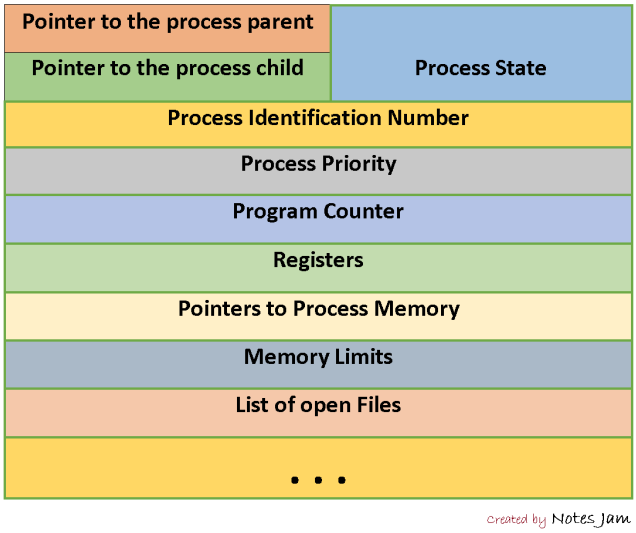


Figura 2: Contenuto della PCB. [2]

1.2 API di processo comuni:

- `Create`: Crea nuovi processi.
- `Destroy`: Distrugge forzatamente i processi.
- `Wait`: Attende la fine del processo.
- `Miscellaneous Control`: Metodi per sospendere un processo e poi riprenderlo.
- `Status`: Restituisce informazioni sullo stato (tempo di esecuzione, ecc.).

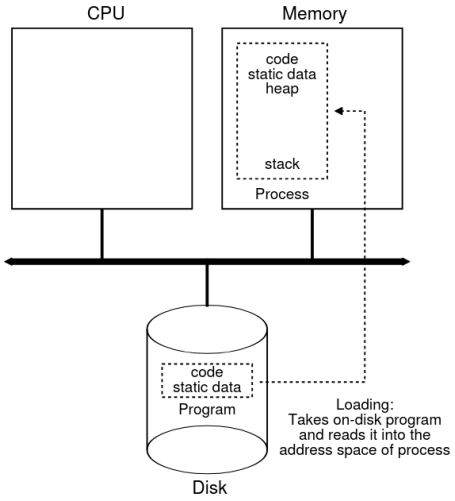


Figura 3: Loading from program to process.

1.3 Creazione dei processi

Quando viene creato un processo, l'OS esegue le seguenti fasi:
1. Carica i dati statici nell'AS del processo.
2. Alloca memoria nello stack per i dati statici (ad esempio, parametri del `main()` e le variabili da inizializzare) e la fornisce al processo.
3. Inizializza gli I/O con tre descrittori di file aperti (input, output, errore).
4. Avvia il programma dal `main()`.
5. Trasferisce il controllo dalla CPU al processo creato.
6. Il processo inizia l'esecuzione.

1.4 Stati di esecuzione dei processi

- **Running**: Il processo è in esecuzione su un processore.
- **Ready**: Il processo è pronto per l'esecuzione, ma non è ancora in esecuzione.
- **Blocked**: Il processo è sospeso in attesa di un evento, come l'input utente o l'accesso a un file.
- **Zombie**: Il processo padre termina prima del processo figlio (il processo ha ancora un PID e un PCB). Per evitare ciò, il processo padre deve chiamare `wait()` per terminare insieme al figlio e pulire le strutture dati.

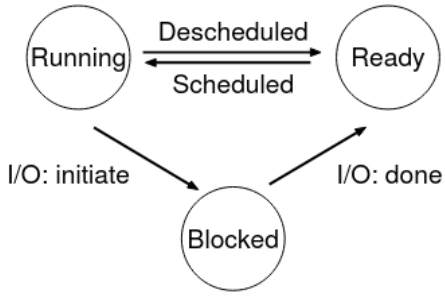


Figura 4: Process state transaction.

1.5 Task List o Process Table

La **task list** (o **process table**) è una struttura dati che tiene traccia di tutti i processi in esecuzione nel sistema.

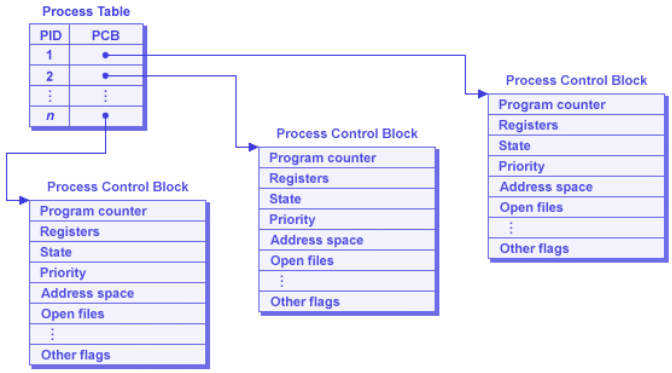


Figura 5: Relazione tra PCB e process table. [2]

1.6 Kernel Stack

Il **kernel stack** è un buffer di memoria allocato dal kernel. Viene utilizzato per memorizzare le informazioni di stato del processo corrente. Spazio di memoria dedicato a ciascun processo quando si trova in modalità kernel, ovvero quando esegue chiamate di sistema o viene gestito direttamente dal sistema operativo.

2 Process API

Process IDentification (PID): Identificatore del processo.

Scheduler: determina quale processo verrà eseguito in un determinato momento. Di solito non è deterministico il figlio potrebbe essere eseguito prima del padre (porta problematiche per i programmi multi thread).

2.1 System call `fork()`

`fork()`: una system call che l'OS fornisce come metodo per creare un nuovo processo. Il processo figlio è quasi identico al padre, cambia solamente l'ID. La `fork` al processo padre restituisce l'ID del processo figlio. Al processo figlio restituisce 0.

Rimangono gli stessi:
- L'indirizzo di memoria del processo padre.
- I registri del processo padre.
- I file aperti dal processo padre.
- I segnali inviati al processo padre.
- Il codice e i dati del processo padre.

Il processo figlio non ha il main come entry point ma la stringa successiva alla sua creazione.

```c
process = fork();
if (process === 0) {
  // Processo creato (codice eseguito solamente dal figlio)
} else if (process < 0) {
  // Errore (processo non creato)
} else {
  // Processo padre (codice eseguito solamente dal padre)
}
// Codice eseguito da padre e figlio
```

Codice 1: Utilizzo della `fork()`.

2.2 System call `exec()`

`exec()`: sostituisce il processo corrente con uno che eseguirà un altro programma. Carica il codice (e i dati statici) del programma che si vuole eseguire e li sovrascrive al segmento di codice corrente. Non viene creato un nuovo processo ma viene trasformato quello preesistente.

```c
int exec(const char *filename, char *const argv[], char *const envp[]);
```

L'istruzione `exec()` sostituisce il processo corrente con un nuovo processo che esegue il programma specificato in `filename`. Gli argomenti del programma sono specificati in `argv`. L'array di ambiente (contiene le variabili di ambiente) è specificato in `envp`.

2.3 System call `wait()`

`wait()`: la chiamata ritarda l'esecuzione del padre fino al termine dell'esecuzione del figlio. Quando il figlio ha finito, `wait()` ritorna al genitore. Rende l'output deterministico.

Processo zombie: processo che ha completato la sua esecuzione e ha chiamato `exit()`, ma il suo processo padre non ha ancora eseguito la chiamata di sistema `wait()` per leggere il suo stato di uscita.

Dettagli chiave:
- Il processo zombie non consuma risorse CPU.
- Mantiene una voce minima nella process table.
- Esiste solo per permettere al padre di recuperare il suo codice di uscita (exit status).
- Se il padre non chiama mai `wait()`, il processo zombie rimarrà nella process table.
- In casi estremi, troppi processi zombie possono esaurire le risorse della process table.

Motivi per cui si verificano i processi zombie:
- Il processo padre non si aspetta la terminazione del processo figlio.
- Il processo padre non è in grado di chiamare la funzione `wait()` a causa di un errore.
- Il processo padre è stato terminato.

3 CPU Mechanism

Interrupt: meccanismo che consiste in un tipo di segnale hardware o software che interrompe il normale flusso di esecuzione di un programma per gestire un evento o una richiesta prioritaria.

Direct execution: permette di eseguire il programma direttamente sulla CPU senza l'intervento di un interprete o di un layer intermedio.

Quando l'OS desidera avviare un programma in Direct Execution:
1. l'OS crea una voce di processo nella Process Table.
2. Alloca della memoria per il programma.
3. Carica il codice del programma in memoria.
4. Imposta lo stack con `argc` e `argv`.
5. Pulisce i registri.
6. Chiama `main()`.
7. il programma esegue `main()`.
8. Ritorna dal `main()`.
9. L'OS libera la memoria del processo terminato.
10. Rimuove l'entry dalla process table.

Per il controllare le azioni indesiderate si ricorre a LDE.

Limited Direct Execution (LDE): tecnica utilizzata per eseguire programmi utente in modo efficiente, mantenendo al contempo il controllo sulla gestione delle risorse e la sicurezza del sistema. Consiste nel controllare e limitare l'accesso diretto dei processi all'hardware sottostante. L'obiettivo principale della LDE è combinare i vantaggi dell'esecuzione diretta del codice da parte dell'hardware (per massimizzare le prestazioni) con i meccanismi di controllo necessari per prevenire comportamenti errati o malevoli da parte dei programmi utente.

3.1 Problema 1 (operazioni limitate)

Quando un processo decide di eseguire un tipo di esecuzione limitata ci sono due modalità di esecuzione:

- **User mode**: il codice è limitato in ciò che può fare in quanto le applicazioni non hanno accesso completo alle risorse hardware.
- **Kernel mode**: il codice eseguito può fare ciò che vuole, l'OS ha accesso a tutte le risorse della macchina.

Quando un processo utente vuole eseguire un operazione privilegiata lo fa attraverso le system call che consentono al kernel di esporre alcune funzionalità fondamentali al processo utente a seguito di una istruzione **trap**.

Per eseguire una system call il programma utente dovrà chiamare un interrupt facendo saltare l'esecuzione nel kernel e quindi cambiare i privilegi in kernel mode. Al termine l'OS richiama una **return-from-trap** e l'esecuzione ritorna al programma utente abbassando i privilegi alla user mode.

Flusso di esecuzione dell'interrupt:
1. L'hardware genera un segnale di interrupt.
2. La CPU interrompe l'esecuzione del processo corrente e salva lo stato del processo corrente sullo stack kernel del processo.
3. La CPU passa il controllo al trap handler. La routine di interrupt handler è responsabile di gestire l'interrupt.
4. La trap handler legge i dati dall'hardware che ha generato l'interruzione e gestisce l'interruzione.
5. La trap handler ripristina lo stato del processo corrente dallo stack kernel del processo.
6. La CPU passa il controllo nuovamente al processo corrente.

Funzionamento della trap: il processo chiamante non può selezionare direttamente dove saltare altrimenti i programmi potrebbero accedere ovunque nel kernel. Quindi, il kernel gestisce quale codice viene eseguito quando si verifica una trap.

1. All'avvio della macchina (in kernel mode) viene creata una **trap table** nella quale l'OS informa l'hardware della posizione dei trap handlers. La trap table contiene il puntatore alla prima cella della **system call table**.
2. Viene assegnato un codice ad ogni system call per essere identificata e poi chiamata.
3. Il codice utente specifica quale chiamata occorre effettuare e l'OS verifica la validità del codice che in nel caso affermativo effettuera la determinata system call.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of... syscall handler	

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to main	
		Run main()
		...
		Call system call **trap** into OS
	save regs (to kernel stack)	
	move to kernel mode	
	jump to trap handler	
Handle trap		
Do work of syscall		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to PC after trap	
		...
		return from main
		trap (via `exit()`)
Free memory of process		
Remove from process list		

Figura 6: Limited Direct Execution protocol.

3.2 Problema 2 (Context Switching)

Immaginiamo di avere un processo in esecuzione sulla CPU fisica che abbia appena esaurito il lasso di tempo a disposizione. A questo punto il sistema operativo dovrebbe eseguire il codice che consente di bloccare l'esecuzione del processo corrente e schedulare il prossimo da eseguire. Il problema è che ciò non può avvenire visto che il processore è attualmente occupato. Soluzioni

- **Approccio cooperativo (via software)**: dato che quando il processo è in esecuzione, non lo è anche l'OS. Si presume che i processi lunghi rinuncino periodicamente alla CPU per far riprendere l'attività all'OS. La maggior parte dei processi trasferisce il controllo all'OS dopo una system call attraverso `yeld()`. I processi che fanno qualcosa di illegale generano una trap. L'OS riprende il controllo della CPU ogni volta che si effettua un system call o un operazione illegale.
- **Approccio non cooperativo (via hardware)**: per restituire il controllo all'OS è il **timer di interrupt** (timer che allo scadere di x millisecondi, genera un interrupt). Quando l'interrupt viene generato, il processo in esecuzione viene interrotto e viene eseguito un **trap handler** (preconfigurato nell'OS). L'OS deve informare l'hardware di quale codice eseguire quando si verifica l'interruzione da parte del timer (impostato al momento del boot del sistema). Anche durante il boot l'OS deve avviare il timer. Il timer può essere anche disattivato. L'hardware, al momento dell'interruzione, deve salvare lo stato del programma in esecuzione per consentire, dopo la **return-from-trap**, di riprendere la sua corretta esecuzione.

3.3 Salvataggio/ripristino e context-switch

Dopo un interrupt è necessario stabilire come e se cambiare processo in esecuzione (decisione presa dallo **scheduler**). Se lo scheduler decide che occorre cambiare processo in esecuzione, l'OS esegue un pezzo di codice a basso livello chiamato

context-switch. L'OS per salvare il contesto del processo in esecuzione:

1. Il processo A è in esecuzione.
2. L'hardware:
   1. Genera l'interrupt allo scadere del timer interrupt.
   2. Salva i registri del processo A nello stack kernel.
   3. Eleva i privilegi passando in kernel mode.
   4. Passa al trap handler.
3. Il trap handler:
   1. Chiama la routine `switch()`.
      1. Salva i registri del processo A nella PCB di A.
      2. Ripristina i registri del processo B dalla PCB di B.
      3. Passa allo stack kernel del processo B.
   2. L'OS fa return-from-trap.
4. L'hardware:
   1. Ripristina dei registri di B dallo stack kernel.
   2. Cambia il livello di privilegio in user mode.
   3. Saltare all'indirizzo del PC del processo B.

Siccome no tutti gli interrupt portano a un context-switch, l'hardware salva i registri essenziali nel kernel per gestire lp'interrupt. Tipicamente salva il Program Counter (PC), Program Status Word (PSW) e lo Stack Pointer (SP). Successivamente, in caso di context-switch, l'OS salva tutti i registri per poi essere in grado di ripristinarli in seguito.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember addresses of...	
	syscall handler	
	timer handler	
start interrupt timer		
	start timer	
	interrupt CPU in X ms	

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) → proc_t(A)		
restore regs(B) ← proc_t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Figura 7: Limited Direct Execution with interrupt.

Per fare context-switching, l'OS salva alcuni valori dei registri per il processo in corso di esecuzione (generalmente li salva nel kernel stack) e ripristinarne altri per il processo scelto. Facendo ciò è in grado di assicurare che quando si esegue l'istruzione per ritornare dall'interrupt (**IRET**), anziché ritornare al processo che era in esecuzione, il sistema riprenderà l'esecuzione del processo schedulato. L'OS dovrà quindi salvare i registri di general purpose, program counter, kernel stack pointer del currently-running-process e successivamente ripristinare i rispettivi valori del chosen-process. Quando viene eseguita la return-from-trap il processo che andrà in esecuzione sarà quello scelto, completando così il context switch. Cambiando puntatori al kernel stack (di fatto cambiando stack), il kernel è in grado di cambiare "contesto", passando dal processo in corso di esecuzione a quello schedulato.

3.4 Problemi di concorrenza e locking

Se un altro interrupt si verifica durante il processo di gestione di un interrupt, il kernel deve decidere come gestire la situazione. È possibile disabilitare gli interrupt durante la gestione dell'interrupt impedendo che altri interrupt vengano inviati alla CPU fino a quando l'interrupt corrente non è stato gestito. Disabilitare gli interrupt per un periodo di tempo prolungato può causare la perdita di interrupt. Gli OS hanno sviluppato schemi di **locking** per proteggere l'accesso concorrente alle strutture dati interne. Ciò consente a più attività di essere in esecuzione all'interno del kernel contemporaneamente.

3.5 CPU scheduling

Burst: intervallo di tempo in cui viene usata intensamente una risorsa. Il CPU Burst, ad esempio, è un'intervallo di tempo t in cui si ha un elevato utilizzo del processore. Per l'I/O Burst vale la stessa definizione in termini di dispositivi di input/output. Da ciò possiamo estrapolare altre due definizioni di classi funzionali di processi, che sono:
- **CPU Bound**: processi con CPU Burst lunghi, ad esempio compilatori, simulatori, calcolo del tempo, ecc...
- **I/O Bound**: con l'avvento dei microcomputer si introduce anche questa necessità. Sono processi con I/O Burst lunghi, ciò comporta maggiore interattività con l'utente.

Un processo in esecuzione si trova o in CPU Burst o in I/O Burst. Mentre un processo viene eseguito sulla CPU, non fa I/O e mentre fa I/O non può essere eseguito dal processore. Lo scheduler deve fare in modo che CPU e dispositivi I/O siano sempre impegnati. Vogliamo evitare che una risorsa entri in uno stato chiamato di IDLE, in cui è accesa e funzionante ma non utilizzata. In questo modo siamo in grado di aumentare l'efficienza, riducendo gli sprechi di tempo.

Workload: processi in esecuzione nel sistema.

Metriche di pianificazione usata per misurare l'efficienza:
- **Tempi di consegna (prestazioni)**: il momento in cui il lavoro viene completato meno il momento in cui il lavoro è arrivato nel sistema.

$$T_{\text{turn around}} = T_{\text{completion}} - T_{\text{arrival}}$$

- **Equità**: il principio di trattare in modo giusto e bilanciato i processi in esecuzione sulla CPU. Uno scheduler può ottimizzare le prestazioni a costo di impedire l'esecuzione di alcuni jobs (diminuzione dell'equità).

Supponiamo che (assunzioni):
1. Ogni lavoro viene eseguito per la stessa quantità di tempo.
2. Tutti i lavori arrivano contemporaneamente ($T_{\text{arrival}} = 0$).
3. Una volta avviato, ogni lavoro viene eseguito fino al completamento.
4. Tutti i lavori utilizzeranno solo la CPU (non eseguono I/O).
5. Il tempo di esecuzione di ogni lavoro è noto.

3.6 First In, First Out (FIFO)

Politica anche chiamata First Come, First Served (FCFS).

Esegue i job seguendo l'ordine di arrivo.

Ordine di arrivo: 1° A(10s) , 2° B(10s), 3° C(10s).

Tempi di consegna:
- $T_{\text{turn around}}(A) = 10 - 0 = 10$
- $T_{\text{turn around}}(B) = 20 - 0 = 20$
- $T_{\text{turn around}}(C) = 30 - 0 = 30$
- Tempo medio di consegna: $\frac{10+20+30}{3} = 20$

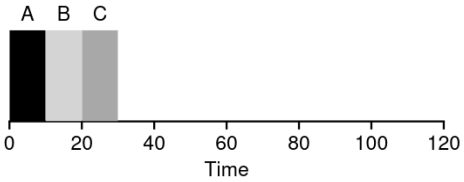


Figura 8: FIFO simple example

Vantaggi: semplice e facile da implementare.

Problema: supponiamo che il job A abbia un tempo di esecuzione maggiore di quello di B e C (negazione assunzione 1).

Ordine di arrivo: 1° A(100s) , 2° B(10s), 3° C(10s).

Tempo medio di consegna: $\frac{100+110+120}{3} = 110$

Convoy effect (effetto convoglio): un numero di processi relativamente brevi capitano in coda dietro un processo più lungo.

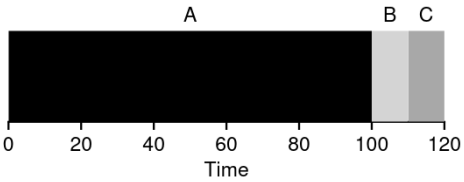


Figura 9: Convoy Effect example.

3.7 Shortest Job First (SJF)

Ottimizza i tempi di risposta ed evita il convoy effect. Esegue il workload con CPU burst minore, poi il successivo più breve e così via.

Ordine di arrivo: 1° A (100s) , 2° B (10s), 3° C (10s).

Tempi di consegna:
- $T_{\text{turn around}}(A) = 10 - 0 = 10$
- $T_{\text{turn around}}(B) = 20 - 0 = 20$
- $T_{\text{turn around}}(C) = 120 - 0 = 30$
- Tempo medio di consegna: $\frac{10+20+120}{3} = 50$

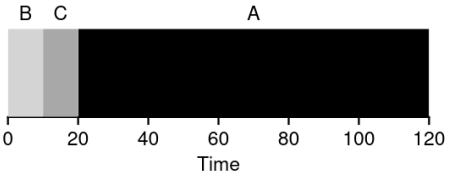


Figura 10: SJF simple example.

Problema: i lavori non arrivano tutti inseme (negazione dell'assunzione 2). Se un workload con burst CPU corto arriva quando la CPU è occupata con un workload con CPU più lungo, il convoy effect è ancora presente.

Ordine di arrivo: 1° A (t=0, 100s) , 2° B (t=10, 10s), 3° C (t=10, 10s).

B e C anche essendo arrivati leggermente dopo di A sono comunque costretti ad aspettare A.

Tempo medio di consegna: $\frac{100+(110-10)+(120-10)}{3} = 103,33$

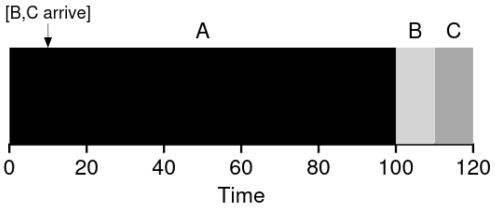


Figura 11: SJF with late arrivals from B and C.

Nuova metrica: **tempo di risposta**

$$T_{\text{response}} = T_{\text{first turn}} - T_{\text{arrival}}$$

Tempi di risposta:
- $T_{\text{response}}(A) = 0 - 0 = 0$
- $T_{\text{response}}(B) = 100 - 10 = 90$
- $T_{\text{response}}(C) = 110 - 10 = 100$
- $T_{\text{response AVG}} = \frac{0+90+100}{3} = 63,33$

È anche uno scheduler **non preventivo** e in quanto tale porta a termine i processi fino al completamento.

Non-preemptive scheduler: non interrompe un processo in esecuzione per pianificare un altro processo.

Preemptive scheduler: capace di interrompere un processo in esecuzione per assegnare la CPU a un altro processo con priorità maggiore o più urgente.

3.8 Shortest Time to Completation First (STCF)

Come SJF ma ottimizza i tempi di risposta. Ogni volta che un job entra nel sistema, lo scheduler determina quale job ha il minor tempo rimanente e lo schedula. Quindi anticipa l'esecuzione di B e C.

Ordine di arrivo: 1° A (t=0, 100s) , 2° B (t=10, 10s), 3° C (t=10, 10s).

Tempi di consegna:
- $T_{\text{turn around}}(A) = 120 - 0 = 120$
- $T_{\text{turn around}}(B) = 20 - 10 = 10$
- $T_{\text{turn around}}(C) = 30 - 10 = 20$
- Tempo medio di consegna: $\frac{120+10+20}{3} = 50$

Tempi di risposta:
- $T_{\text{response}}(A) = 0 - 0 = 0$
- $T_{\text{response}}(B) = 10 - 10 = 0$
- $T_{\text{response}}(C) = 20 - 10 = 10$
- $T_{\text{response AVG}} = \frac{0+0+10}{3} = 3,33$

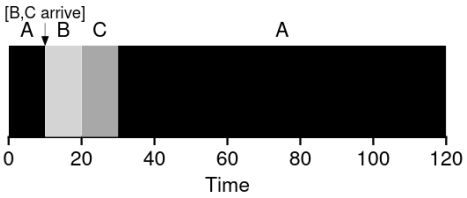


Figura 12: SJF simple example.

Problema: la policy non è efficace perché i job effettuano anche operazioni di I/O e con si conosce quanto sarà effettivamente il loro tempo di consegna.

3.9 Round Robin (RR)

Risolve il problema dell'equità, il convoy effect e il problema della responsività.

È uno scheduler **preventivo**: esegue i job per intervallo di tempo (**quanto di pianificazione**) grazie all'interrupt time. La durata di un quanto di pianificazione deve essere un multiplo del time di interrupt.

Problema: rendere T_{response} troppo corto perché sarebbe troppo oneroso per le risorse dato il context switch.

Ordine di arrivo: 1° A (t=0, 5s) , 2° B (t=0, 5s), 3° C (t=0, 5s).

Tempi di consegna:
- $T_{\text{turn around}}(A) = 13 - 0 = 13$
- $T_{\text{turn around}}(B) = 14 - 0 = 14$
- $T_{\text{turn around}}(C) = 15 - 0 = 15$
- Tempo medio di consegna: $\frac{13+14+15}{3} = 14$ (troppo)

Tempi di risposta:
- $T_{\text{response}}(A) = 0 - 0 = 0$
- $T_{\text{response}}(B) = 10 - 0 = 10$
- $T_{\text{response}}(C) = 20 - 0 = 20$
- $T_{\text{response AVG}} = \frac{0+10+20}{3} = 10$

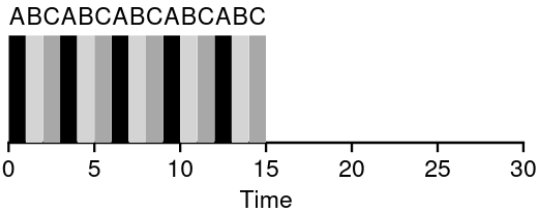


Figura 13: RR simple example.

3.10 Integrazione dell'I/O

Un processo necessariamente ha bisogno di I/O (negazione assunzione 4).

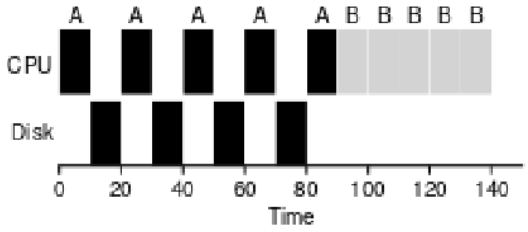


Figura 14: RR senza overlap.

- Job A = 50 ms (ogni 10 ms effettua una richiesta I/O con una durata di 10 ms).
- Job B = 50 ms (non effettua I/O).

Tempi di consegna
- $T_{\text{turn around}}(A) = 90 - 0 = 90$
- $T_{\text{turn around}}(B) = 140 - 0 = 140$
- $T_{\text{turn around AVG}} = \frac{90+140}{2} = 115$

Tempi di risposta
- $T_{\text{response}}(A) = 0 + 0 = 0$
- $T_{\text{response}}(B) = 90 + 0 = 90$
- $T_{\text{response AVG}} = \frac{0+90}{2} = 45$

Quando un job effettua operazioni di I/O non utilizzerà la CPU e quindi lo scheduler dovrebbe pianificare un altro job in attesa che quello precedente si sblocchi.

Quando l'operazione di I/O viene completata sarà generato un interrupt che metterà quel job nello stato "ready".

Funzionamento delle **overlap**:

Uno scheduler STCF tratta ogni lavoro secondario (10 ms) di A come un lavoro indipendente (tiene conto delle operazioni I/O e viene suddiviso in più jobs).

Parte prima un sub-job di A (10 ms), quando A fa I/O può partire B (per un quanto di pianificazione) contemporaneamente alle operazioni I/O di A (che non utilizza la CPU per fare I/O).

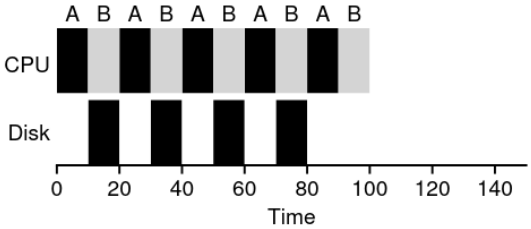


Figura 15: Overlap allows better use of resources.

Tempi di consegna
- $T_{\text{turn around}}(B) = 100 - 0 = 100$
- $T_{\text{turn around AVG}} = \frac{90+100}{2} = 80$
- $T_{\text{turn around}}(A) = 90 - 0 = 90$

Tempi di risposta
- $T_{\text{response}}(A) = 0 + 0 = 0$
- $T_{\text{response}}(B) = 10 + 0 = 10$
- $T_{\text{response AVG}} = \frac{0+10}{2} = 5$

3.11 Cheatsheet

- $T_{\text{turn around}} = T_{\text{completion}} - T_{\text{arrival}}$
- $T_{\text{response}} = T_{\text{first turn}} - T_{\text{arrival}}$
- $T_{\text{wait}} = T_{\text{turn around}} - T_{\text{run time}}$

4 Multi-Level Feedback Queue

Obbiettivi:
- Ottimizzare i tempi di consegna.
- Ridurre al minimo il tempo di risposta (rendere un sistema responsive).

MLFQ ha un numero di code distinte e ciascuna di queste ha un livello di priorità (priorità usata per stabilire il job da eseguire in un determinato momento). Quando più lavori si trovano nella stessa coda viene applicato il RR.

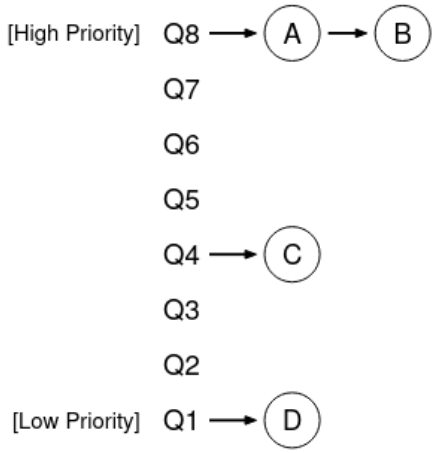


Figura 16: Esempio MLFQ.

MLFQ assegna una priorità variabile (in base al comportamento del processo).
- Se un job effettua ripetutamente I/O manterrà la sua priorità (per renderlo interattivo).
- Se un job utilizza in modo intensivo la CPU verrà ridotto di priorità.

Starvation: se ci sono troppi lavori interattivi questi monopolizzano la CPU e quindi i processi più lunghi non riceveranno più tempo dalla CPU (non vengono eseguiti quindi «moriranno di fame").

Come va impostato s (indica la frequenza di cambiamento delle priorità):
- Se troppo alto: i job più lunghi soffrirebbero di starvation.
- Se troppo basso: i job interattivi potrebbero non ottenere una quota adeguata della CPU.

Gaming: prima che un quanto di pianificazione sia trascorso il programma va a fare I/O e quindi non viene abbassato di priorità.

Siccome un programma può cambiare il suo comportamento nel tempo, un programma che era precedentemente vincolato dalla CPU e attualmente diventa interattivo sarà penalizzato perché si troverà nella fascia di priorità più bassa.

Per impedire il gaming, lo scheduler tiene traccia del tempo utilizzato da un processo (**accounting**), una volta che il suo quanto verrà esaurito seguirà il suo abbassamento di priorità.

Con il comando `nice` è possibile aumentare e diminuire la priorità di lavoro.

4.1 Funzionamento

1. Se `Priorità(A) > Priorità(B)`: va in esecuzione prima `A` e poi `B`.
2. Se `Priorità(A) = Priorità(B)`: viene applicato RR.
3. Quando job entra nel sistema gli viene assegnata la priorità più alta.
4. Decisione:
   - Se il processo completa la sua esecuzione entro il quanto di tempo, esso termina o viene messo in attesa.
   - Se il processo non completa l'esecuzione entro il quanto di tempo, viene spostato nella coda a priorità inferiore (**accounting** del tempo impiegato dal processo: soluzione al gaming).
5. Dopo un tempo s, sposta tutti i job nel sistema nella coda con massima priorità (risolve la starvation e la dinamicità dei programmi).

5 Address Space (AS)

I primi OS non offrivano molte astrazioni dalla memoria. L'OS era una serie di routine (una libreria, in realtà) che si trovava in memoria (a partire dall'indirizzo fisico 0 in questo esempio). C'era un solo processo in esecuzione che occupava il resto della memoria.

Nascita della multiprogrammazione: le macchine erano costose, le persone iniziavano a condividerle. I processi erano pronti per essere eseguiti in un momento e l'OS passava da un programma all'altro.

I sistemi multiprogrammazione e time sharing sono stati sviluppati per migliorare l'utilizzo del processore e per consentire a più utenti di utilizzare una macchina contemporaneamente.

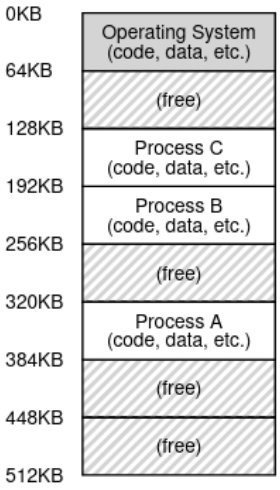


Figura 17: Sharing memory.

5.1 Address space moderno

In un sistema **multiprogrammazione**, più processi sono pronti per l'esecuzione e l'OS passa da un processo all'altro, ad esempio quando uno decide di eseguire un'operazione di I/O.

Interattività: capacità di un sistema di rispondere alle azioni dell'utente.

In un sistema time sharing, i processi vengono eseguiti per un breve periodo di tempo, prima di essere sospesi e sostituiti da un altro processo. Questo consente a più utenti di utilizzare una macchina contemporaneamente e di ricevere un feedback tempestivo dalle loro attività.

La protezione della memoria è un problema importante nei sistemi multiprogrammazione e time sharing, poiché è necessario impedire che un processo acceda alla memoria di un altro processo.

L'interattività diventa importante. Un modo per implementare il time-sharing (molto lento):
1. Eseguire il processo A per un breve periodo (dandogli accesso a tutta la memoria).
2. Interrompere il processo A.
3. Salvare lo stato del processo A su un disco.
4. Caricare stato del processo B.
5. Eseguire il processo B per il suo quanto di pianificazione ecc.

Approccio ottimale: si lasciano i processi in memoria facendo time-sharing (salvataggio e ripristino dello stato a livello di registro).

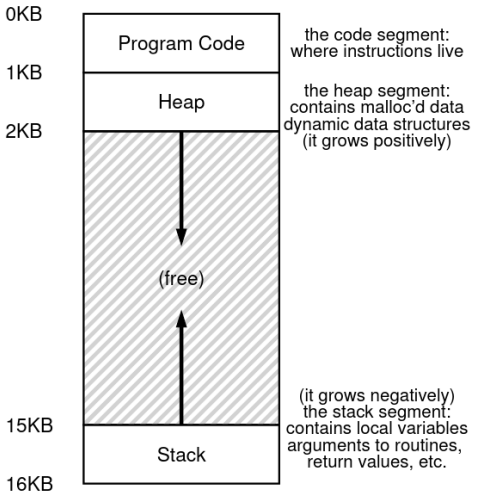


Figura 18: Address space example.

Address space: astrazione del programma in esecuzione fornita dall'OS (range di indirizzi di memoria sui quali il programma è in esecuzione).

Contiene:
- **Stato** della memoria del processo in esecuzione.
- **Codice** del programma.
- **Heap** utilizzato per la memoria allocata dinamicamente. Esso viene gestito dall'utente e cresce verso il basso.
- **Stack** che tiene traccia di dove si trova il processo nella catena di chiamate a funzione, utilizzato per allocare variabili locali, passare parametri, per restituire valori da/verso le routine. Cresce verso l'alto.

Virtualizzazione della memoria: il processo non dispone la conoscenza degli indirizzi fisici sui quali è in esecuzione e ha l'illusione di avere a disposizione una memoria potenzialmente molto grande.

Il programma non risiede realmente in memoria dall'indirizzo 0 a 16 KB, sarà invece caricato a indirizzi fisici arbitrari.

5.2 Obbiettivi dell'OS

- **Trasparenza:** l'OS deve virtualizzare la memoria in modo che il programma in esecuzione non sia consapevole del fatto che la memoria è virtualizzata.
- **Efficienza:** l'OS deve rendere la virtualizzazione efficiente, sia in termini di
  - Tempo: la virtualizzazione non deve rallentare il programma in esecuzione.
  - Spazio: l'OS non deve allocare più memoria di quella necessaria per virtualizzare la memoria.
- **Protezione:** l'OS deve proteggere i processi l'uno dall'altro e l'OS stesso dai processi. Ciò significa che un processo non deve essere in grado di accedere alla memoria o ai dati di un altro processo. l'OS deve proteggere l'OS stesso da processi che potrebbero essere malevoli.

6 Memory API

In un programma C, ci sono due tipi di memoria che vengono allocati:
- **Stack**: le cui allocazioni e deallocazioni vengono gestite implicitamente dal compilatore per il programmatore.
- **Heap**: dove tutte le allocazioni e le deallocazioni sono gestite esplicitamente dal programmatore.

6.1 `malloc()`

L'heap è necessaria per memorizzare informazioni che devono vivere oltre l'invocazione di una funzione. Per allocare memoria nello heap si usa `malloc()`. `malloc()` richiede come parametro un blocco di memoria di una determinata dimensione e restituisce un puntatore a tale blocco. Il programmatore è responsabile della deallocazione della memoria allocata con `malloc()` utilizzando la funzione `free()`.

Funzionamento di `malloc()`: si passa un valore di dimensione che indica la quantità di memoria richiesta e viene restituito un puntatore alla memoria appena allocata, oppure `NULL` se l'allocazione non è riuscita.

La sintassi di `malloc()` è la seguente:

```c
void *malloc(size_t size);
```

La funzione `malloc()` alloca un blocco di memoria di dimensioni `size` e restituisce un puntatore al blocco di memoria allocato. Il tipo del puntatore restituito è `void *` quindi è necessario castare il puntatore restituito a un tipo specifico prima di utilizzarlo.

Il tipo `size_t`, rappresenta il numero di byte da allocare. In genere si utilizza l'operatore `sizeof()` per ottenere il valore corretto. Ad esempio, per allocare spazio per un valore di tipo `double`, si può scrivere:

```c
double *d = (double *) malloc(sizeof(double));
```

In questo caso, l'operatore `sizeof()` viene utilizzato per ottenere il valore di 8, che è la dimensione di un valore di tipo `double`.

La chiamata `malloc()` può anche essere utilizzata per allocare memoria per un array di elementi. Ad esempio, per allocare memoria per un array di 10 elementi di tipo `int`, si può scrivere:

```c
int *x = malloc(10 * sizeof(int));
```

In questo caso, la chiamata `malloc()` richiede $10 * 8 = 80$ byte di memoria.

6.2 `free()`

Utile liberare la memoria heap che non è più in uso, i programmatori chiamano semplicemente `free()` .

La funzione `free()` prende un puntatore alla memoria allocata da `malloc()` e la dealloca.

```c
void free(void *ptr);
```

7 Address Translation

L'indirizzo logico 0, ad esempio, non corrisponderà all'indirizzo fisico 0 (generalmente occupato dal sistema operativo) ma dovrà essere mappato in un'altra locazione di memoria fisica.

Mapping: trovare una corrispondenza fra indirizzo logico e indirizzo fisico (PA).

Tecniche di mapping
1. **Mapping Diretto**: ogni pagina virtuale è mappata a un frame fisico fisso, senza flessibilità. È una tecnica semplice, ma non adatta per gestire efficientemente sistemi complessi o grandi.
2. **Base e Bound**: tecnica semplice per la protezione della memoria, ma limitata in termini di flessibilità e scalabilità. Utilizza due registri
   - **Base**: per l'indirizzo base della memoria allocata al processo.
   - **Bound**: per il limite che definisce la dimensione della memoria.
3. **Segmentazione**: divide la memoria in segmenti logici di dimensioni variabili (ad esempio, codice, dati, stack). Ogni segmento ha un indirizzo base e una lunghezza, ma può causare frammentazione esterna, che la rende più complessa da gestire rispetto al mapping diretto.
4. **Paginazione**: memoria viene suddivisa in blocchi fissi chiamati «pagine». Ogni pagina è mappata a un frame fisico attraverso una tabella delle pagine. Riduce la frammentazione esterna ma introduce la possibilità di frammentazione interna e overhead di gestione.
5. **Paginazione con TLB (Translation Lookaside Buffer)**: l'uso di una cache ad alta velocità (TLB) per memorizzare le traduzioni più recenti degli indirizzi virtuali migliora le performance della paginazione. Sebbene il TLB ottimizzi la velocità, introduce la complessità della gestione della cache.
6. **Paginazione Multilivello**: la tabella delle pagine è suddivisa in più livelli, riducendo il consumo di memoria per la gestione delle tabelle. Ogni livello di tabella punta a un altro livello fino ad arrivare alla traduzione finale dell'indirizzo. Aumenta la complessità di gestione, ma è più efficiente in termini di memoria.

7.1 Tipi di Address Translation

Assunzioni:
1. Address space contiguo.
2. Address space non troppo grande e inferiore alla memoria fisica.
3. Tutti gli address space hanno la stessa dimensione.

Hardware-based address translation: l'hardware trasforma ogni accesso in memoria modificando l'indirizzo virtuale (VA) fornito dall'istruzione in un indirizzo fisico (dove è effettivamente memorizzata l'informazione). L'hardware da solo non può virtualizzare la memoria perché fornisce soltanto i meccanismi a basso livello per farlo in modo efficiente. L'OS deve gestire la memoria tenendo traccia delle posizioni libere e utilizzate controllando come viene utilizzata la memoria.

Dynamic (Hardware-based) Relocation: permette di spostare un processo in diverse posizioni della memoria fisica durante l'esecuzione senza che il processo stesso debba essere modificato. Il processore mantiene due registri hardware, il **base register** e il **bounds register**, che vengono utilizzati per la traduzione dell'indirizzo.

7.2 Registri per l'address translation

Registri base-and-bounds: ci permettono di posizionare l'address space ovunque nella memoria fisica assicurando che il processo possa accedere solamente al suo address space.

base register: contiene l'indirizzo fisico di base dell'area di memoria del processo.

bounds register: contiene la dimensione dell'area di memoria del processo.

Quando un processo genera un riferimento alla memoria, il processore aggiunge il valore del base register all'indirizzo virtuale fornito dal processo e ottiene un indirizzo fisico.

$$\text{indirizzo fisico} = \text{indirizzo virtuale} + \text{base}$$

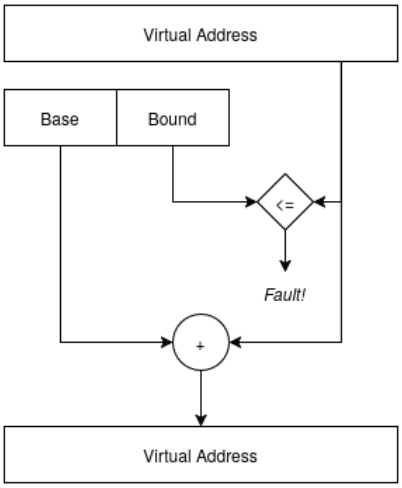


Figura 19: Base and bound register. Schema creato dall'autore.

Traduzione di indirizzi: trasformare un indirizzo virtuale in uno fisico. L'hardware che trasforma un indirizzo virtuale, a cui il processo pensa faccia riferimento, in uno fisico.

Se un processo genera un indirizzo virtuale esternamente ai limiti, la CPU solleverà un'eccezione e il processo verrà killato. I registri base-and-bounds sono strutture hardware mantenute dai chip (due per CPU).

Rilocazione statica: tecnica di rilocazione software-based eseguita dal loader. Il loader modifica gli indirizzi all'interno di un file eseguibile prima che il programma venga eseguito. Il processo di rilocazione consiste nel riscrivere gli indirizzi del programma, aggiungendo un offset che corrisponde alla posizione in memoria fisica in cui il programma verrà caricato. Ad esempio, se un programma è compilato per essere caricato a partire dall'indirizzo 0, ma viene caricato in memoria a partire dall'indirizzo 3000, tutti gli indirizzi del programma vengono aumentati di 3000. La rilocazione statica viene eseguita una sola volta, al momento del caricamento del programma in memoria. Una volta che un processo è stato caricato con rilocazione statica, è difficile spostarlo in un'altra posizione in memoria.Questa tecnica non offre protezione della memoria, poiché i processi possono generare indirizzi errati e accedere a memoria non autorizzata.

Rilocazione dinamica: tecnica di rilocazione hardware-based eseguita a runtime. Utilizza i registri base e bounds. Il base register contiene l'indirizzo fisico di partenza del processo in memoria. Ogni indirizzo virtuale generato dal programma viene sommato al valore del base register per ottenere l'indirizzo fisico corrispondente. Il bounds register definisce i limiti dell'area di memoria che il processo può utilizzare. L'hardware verifica che ogni indirizzo generato dal processo sia entro questi limiti. La traduzione degli indirizzi virtuali in indirizzi fisici viene eseguita dall'hardware ad ogni accesso alla memoria. La rilocazione dinamica permette di spostare un processo in una diversa posizione della memoria fisica anche dopo che è iniziato.Questa tecnica consente di implementare la virtualizzazione della memoria in modo efficiente e protetto, garantendo che ogni processo acceda solo alla propria area di memoria.

In sintesi, la **rilocazione statica** è un processo che modifica gli indirizzi di un programma prima della sua esecuzione, mentre la **rilocazione dinamica** avviene durante l'esecuzione del programma e utilizza registri hardware per la traduzione degli indirizzi e la protezione della memoria.

7.2.1 Esempio di traduzione

Supponiamo un processo con uno spazio di indirizzi di dimensioni 4 Kb che è stato caricato all'indirizzo fisico 16 KB.

Virtual Address		Physical Address
0	→	16 KB
1 KB	→	17 KB
3000	→	19384
4400	→	*Fault (out of bounds)*

Figura 20: Esempio di traduzione.

7.3 Gestione dello spazio

MMU (Memory Management Unit): gruppo di componenti hardware che gestisce gli accessi in memoria e si occupa di tradurre gli indirizzi virtuali in indirizzi fisici.

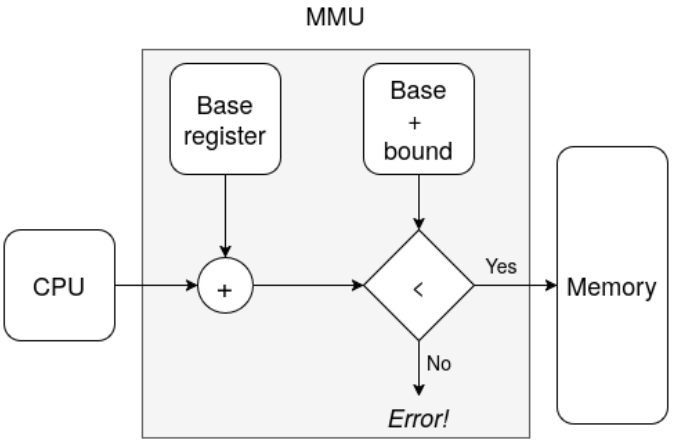


Figura 21: Come calcola il PA l'MMU. Schema creato dall'autore.

L'hardware dovrebbe fornire istruzioni speciali per modificare il base-and-bounds register accessibili solamente in modalità kernel.

Free list: elenco degli intervalli della memoria fisica non ancora utilizzati (l'OS deve tenere traccia della memoria per capire se è possibile allocare i processi).

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

Figura 22: Requisiti per la dynamic relocation.

La CPU deve essere in grado di generare eccezioni in situazioni in cui un processo utente tenta di accedere alla memoria illegalmente. in caso di out of bounds, il processore deve interrompere l'esecuzione del processo utente e organizzare l'esecuzione dell'handler di eccezione out-of-bounds dell'OS. L'handler dell OS decidere come reagire. Allo stesso modo, se un processo utente tenta di modificare i valori dei registri base e limiti (privilegiati), il processore deve sollevare un'eccezione e eseguire l'handler «ha tentato di eseguire un'operazione privilegiata in modalità utente».

OS Requirements	Notes
Memory management	*Need to allocate memory for new processes; Reclaim memory from terminated processes; Generally manage memory via* ***free list***
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise; likely action is to terminate offending process*

Figura 23: OS responsibilities.

OS @ boot (kernel mode)	Hardware	(No Program Yet)
initialize trap table		
	remember addresses of...	
	system call handler	
	timer handler	
	illegal mem-access handler	
	illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

Figura 24: Limited Direct Execution (Dynamic Relocation) @ Boot.

Per supportare la dynamic relocation, l'OS:
1. Quando viene creato un nuovo processo l'OS deve reagire trovando spazio per l'address space dell'processo attraverso la **free list**.
2. Quando il processo viene terminato l'OS ripulisce le strutture dati allocate per il processo (liberando memoria).
3. Quando avviene un context-switch l'OS deve salvare le coppie di base-and-bounds.
4. Quando un processo viene interrotto, l'OS deve salvare in memoria i registri base-and-bounds nel PCB.
5. Quando un processo viene fermato, il S.O. può spostare il suo address space in un'altra locazione di memoria.
6. Quando il processo riprende l'esecuzione, deve ripristinare base e bound.
7. L'OS deve fornire gestori di eccezioni, o funzioni da chiamare, installa questi gestori all'avvio (tramite istruzioni privilegiate). Se un processo tenta di accedere a una memoria al di fuori dei suoi confini, il processore genererà un'eccezione.

OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A:		
allocate entry		
in process table		
alloc memory for process		
set base/bound registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
	translate virtual address	
	perform fetch	
		Execute instruction
	if explicit load/store:	
	ensure address is legal	
	translate virtual address	
	perform load/store	
		(A runs...)
	Timer interrupt	
	move to **kernel mode**	
	jump to handler	
Handle timer		
decide: stop A, run B		
call `switch()` routine		
save regs(A)		
to proc-struct(A)		
(including base/bounds)		
restore regs(B)		
from proc-struct(B)		
(including base/bounds)		
return-from-trap (into B)		
	restore registers of B	
	move to **user mode**	
	jump to B's PC	
		Process B runs
		Execute bad load
	Load is out-of-bounds;	
	move to **kernel mode**	
	jump to trap handler	
Handle the trap		
decide to kill process B		
deallocate B's memory		
free B's entry		
in process table		

Figura 25: Limited Direct Execution (Dynamic Relocation) @ Runtime.

8 Segmentation

Binding: durante il processo rilocazione vengono cambiati tutti gli indirizzi del programma (anche le jump) per evitare che vadano fuori dallo spazio di indirizzamento previsto. Il binding è l'operazione che viene fatta per modificare gli indirizzi. Può essere:

- **Early binding**: riallocazione degli indirizzi fatta a compile time. Il compilatore deve conoscere la posizione di partenza del programma in memoria (informazione che, generalmente, non possiamo conoscere a priori). Questo meccanismo funziona solamente nei sistemi embedded, dove il compilatore genera direttamente il codice assoluto, oppure nei sistemi monoprogrammati.

- **Delayed binding**: la riallocazione degli indirizzi viene fatta durante il trasferimento (loading) del programma da disco a memoria. Quest'operazione viene quindi svolta dal sistema operativo ed è quella che veniva adottata prima dell'introduzione dell'MMU.

- **Late binding**: la riallocazione degli indirizzi viene fatta immediatamente prima di eseguire l'istruzione corrente, quindi a runtime. Occorre il dovuto supporto hardware (MMU) per poter implementare questa tecnica. Durante la fase di decodifica del programma, gli indirizzi relativi vengono cambiati in indirizzi assoluti. Questo meccanismo è alla base della virtualizzazione della memoria.

Problema: lo spazio allocato per la crescita di stack e heap, se non utilizzato e se si adotta la tecnica del base-and-bounds, costituisce uno spreco di memoria.

Prima di proseguire è bene introdurre alcune assunzioni:
- Il programma viene caricato in locazioni di memoria contigue.
- L'address space è minore della memoria fisica.
- L'address space ha dimensione fissata (assunzione già superata con base e bound).

Nel meccanismo base e bound, quando un processo viene caricato in memoria, il sistema operativo mette il punto di partenza all'interno del registro base. Durante la fase di decodifica delle istruzioni viene fatto il binding:

$$\text{Indirizzo fisico (PA)} = \text{Indirizzo logico (VA)} + \text{Base}$$

8.1 Segmantazione

Segmantazione: suddivisione della memoria in blocchi chiamati segmenti con il fine di ottimizzare la gestione degli spazi vuoti (tra stack e heap) di un processo nel suo address space. La segmentazione è un approccio **dinamico** di allocazione della memoria. Questo significa che i processi possono richiedere e rilasciare memoria durante l'esecuzione. Invece di una sola coppia base-and-bounds per MMU, si utilizza una coppia base-and-bounds per ogni singolo segmento. La segmentazione consente all'OS di posizionare i segmenti in parti diverse della memoria fisica, evitando di riempire la memoria fisica con l'address space virtuale inutilizzato. Solo la memoria utilizzata è allocata in memoria fisica.

Il meccanismo funziona come segue:

Input: indirizzo logico B.
1. Individua il segmento s di appartenenza dell'indirizzo B.
2. Calcola l'offset k sottraendo all'indirizzo virtuale l'indirizzo di partenza (logico) del segmento ($k = B -$ indirizzo iniziale di s).
3. Viene calcolato l'indirizzo fisico sommando k e il base register ($\text{PA} = \text{Base(s)} + k$).

L'offset è la distanza l'inizio del segmento e l'indirizzo logico

8.1.1 Esempio di traduzione di un indirizzo

Segment	Base	Size
Code	32K	2K
Heap	34K	3K
Stack	28K	2K

Figura 26: Esercizio: Segment table.

Esercizio 2 Input:
- Indirizzo virtuale (offset): 100

Calcoli:
- Indirizzo base: 32 kb
- indirizzo fisico: $100 + 32 \text{ kb} = 32 * 1024 + 100 = 32868$
- Verifica che rientri nei limiti: $100 < 2$ kb.

Esercizio 2 Input:
- Indirizzo virtuale: 4200

Calcoli:
- Inizio heap (virtuale): 4kb
- Estrazione offset: $4\text{kb} * 1024 - 4200 = 104$
- Indirizzo fisico: $34\text{kb} * 1024 + 104 = 34920$

Steps per la conversione di Indirizzi Virtuali

Input: VA, segment table e virtual start.

1. Identificazione del segmento: determinare in quale segmento si trova il VA (codice, heap, stack, ecc.) usando la segment table valutando in quale range di indirizzi si trova il VA.
2. Estrazione dell'offset:
   - Se il segmento parte da VA=0: non occorre calcolare l'offset.
   - Se il segmento cresce in positivo: $\text{offset} = \text{VA} - \text{virtual\_start}(\text{segmento})$
   - Se il segmento cresce in negativo: $\text{offset} = \text{size\_of}(\text{segmento}) - (\text{virtual\_start}(\text{segmento}) - \text{VA})$
3. Controllo validità dell'offset: $\text{offset} < \text{size\_of}(\text{segmento})$
4. Calcolo dell'Indirizzo fisico $\text{PA} = \text{base} + \text{offset}$

Nota bene: Lo stack cresce in negativo e l'heap in positivo.

Code sharing: Pratica comune nei moderni sistemi operativi che consente di condividere certi segmenti tra gli address spaces, risparmiando memoria.

L'hardware conoscere il segmento a cui si sta riferendo utilizzando:
- **Approccio esplicito**
- **Approccio implicito**

8.1.2 Approccio esplicito

Divide lo spazio degli indirizzi in segmenti basati sui primi bit dell'indirizzo virtuale.

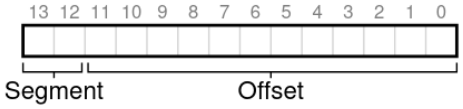


Figura 27: Blocco di memoria a 14 bit.

In un AS grande $1k$ abbiamo bisogno di 10 bit per indirizzare tutti i byte. Il calcolo è $\log_2 1024 = 10$ bit e in generale: $\lceil \log_2 <\text{size\_of\_AS}> \rceil$

Esempio: per selezionare un segmento, supponendo di averne tre a disposizione (codice heap, stack), occorrono 2 bit.

- **00**: l'hardware sa che il virtual address si trova nel segmento **codice**.
- **01**: l'hardware sa che il virtual address si trova nel segmento **heap** ecc.

L'offset facilita il controllo dei limiti perché si controlla che l'offset (in valore assoluto) sia inferiore ai limiti.

Problemi

- Se abbiamo a disposizione solamente 3 segmenti una combinazione di bit non viene utilizzata ($2^2 = 4$ combinazioni di bit). Alcuni mettono codice e heap nello stesso segmento e accedono solamente con due combinazioni.
- Limita l'uso dello spazio degli indirizzi virtuali: ogni segmento ha una dimensione massima (un segmento non può crescere di dimensione).

8.1.3 Approccio implicito

L'hardware determina a quale segmento l'indirizzo fa riferimento in base a come è stato formato.

- Se è stato generato dalla fetch appartiene al segmento **codice**.
- Se l'indirizzo è basato sullo stack o sul base pointer, fa riferimento allo **stack**.
- Altrimenti tutti gli indirizzi rimanenti fanno parte dell'**heap**.

Oltre ad avere base-and-bounds all'hardware occorre sapere se il segmento cresce in positivo o in negativo.

Per risparmiare memoria è utile condividere determinati segmenti (specialmente il codice) tra gli address space. Il **bit di protezione** è stato introdotto per questo. Il registro base utilizza questo bit per ogni segmento e indica se un programma può leggere, scrivere o eseguire un segmento. Lo stesso segmento fisico di memoria poterebbe essere mappato in più address space virtuali. L'hardware deve verificare i bit di protezione quando un processo accede a un segmento.

Segment	Base	Size (max 4K)	Grows Positive?
Code_{00}	32K	2K	1
Heap_{01}	34K	3K	1
Stack_{11}	28K	2K	0

Figura 28: Segment registers with growth support.

8.1.4 Esempio traduzione

```txt
ARG seed 1
ARG address space size 1k
ARG phys mem size 16k

Segment register information:

Seg 0 base  (grows positive): 12513
Seg 0 limit: 290

Seg 1 base (grows negative): 4651
Seg 1 limit: 472

Virtual Address Trace
  VA  0 = 507 --> VIOLATION (SEG0)
  VA  1 = 460 --> VIOLATION (SEG0)
  VA  2 = 667 --> VALID in SEG1: (4294)
  VA  3 = 807 --> VALID in SEG1: (4434)
  VA  4 = 96 --> VALID in SEG0: (12609)
```

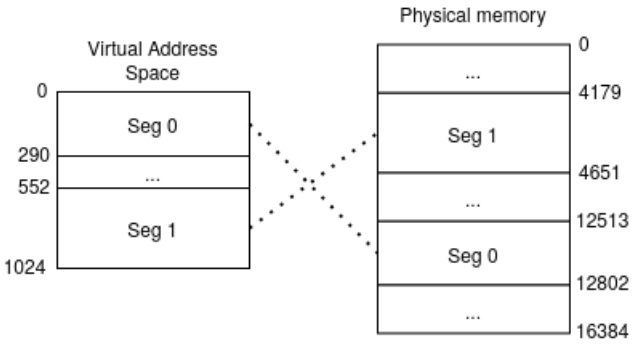


Figura 29: Struttura AS e physical memory dell'esercizio. Schema creato dall'autore.

Traduzione 1

$\text{VA} = 507$

1. Identificazione del segmento:

   $\text{bin}(507) = 01\ 1111\ 1011$
   Il MSB è 0, quindi, il VA appartiene a Seg 0. Siccome $507 >$ 290 l'indirizzo non è valido.

Traduzione 2

$\text{VA} = 460$

1. Identificazione del segmento:

   $\text{bin}(460) = 01\ 1100\ 1100$
   Il MSB è 0, quindi, il VA appartiene a Seg 0. Siccome $507 >$ 290 l'indirizzo non è valido.

Traduzione 3

$\text{VA} = 667$

1. Identificazione del segmento:

   $\text{bin}(667) = 10\ 1001\ 1011$
   Il MSB è 1, quindi, il VA appartiene a Seg 1.

2. Calcolo dell'offset:

   $\text{offset} = 472 - (1024 - 667) = 115$

3. Controllo della validità dell'offset:

   $115 < 472$ quindi l'offset è valido.

4. Calcolo del PA

   $\text{PA} = 4179 + 115 = 4294$

Traduzione 4

$\text{VA} = 807$

1. Identificazione del segmento:

   $\text{bin}(807) = 11\ 0010\ 0111$
   Il MSB è 1, quindi, il VA appartiene a Seg 1.

2. Calcolo dell'offset:

   $\text{offset} = 472 - (1024 - 807) = 255$

3. Controllo della validità dell'offset:

   $255 < 472$ quindi l'offset è valido.

4. Calcolo del PA

   $\text{PA} = 4179 + 255 = 4434$

Traduzione 5

$\text{VA} = 96$

1. Identificazione del segmento:

   $\text{bin}(96) = 00\ 0110\ 0000$
   Il MSB è 0, quindi, il VA appartiene a Seg 0.

2. Calcolo del PA

   $\text{PA} = 12513 + 96 = 12609$

In un AS grande 1k abbiamo bisogno di 10 bit per indirizzare tutti i byte

8.2 Problematiche della segmentazione

Fine-grained-segmentation: l'address space è suddiviso in un numero maggiore di segmenti. Questo consente all'OS di gestire la memoria in modo più granulare e di allocare e deallocare la memoria in modo più efficiente.

Coarse-grained-segmentation: address space suddiviso in un numero minore di segmenti. Questo è più semplice da implementare hardware e software, ma non è così efficiente come la segmentazione fine-grained.

Segment table: struttura dati che mantiene le informazioni sui segmenti di un processo. Ogni segmento è rappresentato da un record nella tabella dei segmenti, che include l'indirizzo base del segmento, la dimensione del segmento e le flag di protezione. Supporta la creazione di un numero molto elevato di segmenti.

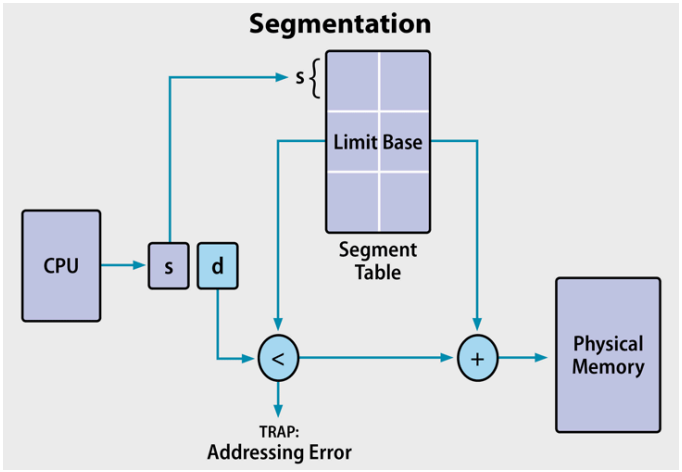


Figura 30: Schema di traduzione. [3]

Problemi della segmentazione:

- **Salvataggio e ripristino dei registri di segmento**: l'OS deve salvare e ripristinare i registri di segmento quando avviene un cambio di contesto. Questo perché ogni processo ha il proprio spazio di indirizzi virtuali e l'OS deve assicurarsi che i registri siano impostati correttamente prima che il processo venga eseguito di nuovo.
- **Creazione e gestione dei segmenti**: l'OS deve creare e gestire i segmenti per ogni processo. Questo include la creazione di nuovi segmenti, la crescita e la contrazione dei segmenti, e la deallocazione dei segmenti non più necessari. La `malloc()` alloca spazio, la system call `sbrk()` (obsoleta ma ancora supportata) aumenta o riduce la dimensione del segmento heap. Ora si utilizza `mmap()`.
- **Gestione della memoria libera**: l'OS deve gestire la memoria libera in modo efficiente. Questo include la riallocazione della memoria libera quando i processi vengono creati, eliminati o quando i segmenti crescono o si contraggono.

Frammentazione esterna: problema che si verifica quando la memoria libera dell'AS è frammentata in piccoli parti che non sono abbastanza grandi per soddisfare una richiesta di allocazione di memoria.

Deframmentazione: operazione complessa e dispendiosa in cui l'OS compatta la memoria fisica riarrangiando i segmenti esistenti. Per eseguirla, il sistema deve fermare tutti i processi

in esecuzione, copiare i loro dati in una regione contigua di memoria, e aggiornare i segment registers affinché puntino alle nuove locazioni. Questo processo consente di ottenere un ampio spazio libero contiguo, ma è bloccante, poiché durante l'operazione nessun programma può essere eseguito.

Ci sono due approcci principali per gestire la frammentazione esterna:

- **Compaction**: consiste nel riorganizzare i segmenti in memoria in modo che i frammenti di memoria libera siano contigui. Questo può essere fatto fermando i processi che sono in esecuzione, copiando i loro dati in una regione di memoria contigua, e quindi aggiornando i registri di segmento dei processi per puntare alle nuove posizioni fisiche. Approccio molto costoso.
- **Gestione della free list**: consiste nel mantenere una lista di tutti i frammenti di memoria libera. Quando viene fatta una richiesta di allocazione di memoria, l'OS cerca il frammento di memoria più grande che è abbastanza grande per soddisfare la richiesta. Meno costosa ma non efficiente.

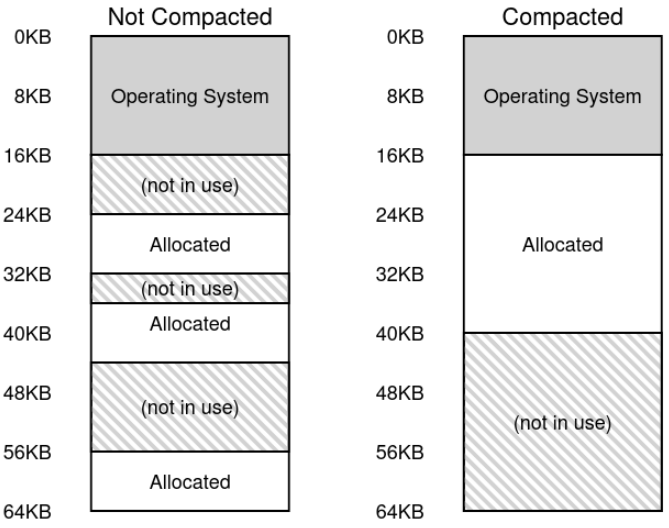
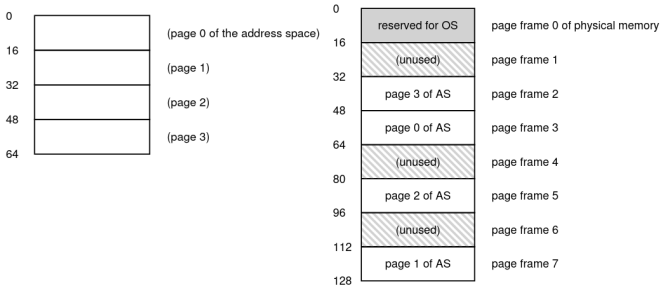


Figura 31: Esempio di come la memoria fisica viene trasformata con la compaction.

9 Paging

Obbiettivi:
- Gestione ottimale dello spazio libero in memoria fisica.
- Gestione ottimale dell'address space di un programma (non vogliamo memorizzare più di quanto non sia necessario).

Per la gestione dello spazio esistono due approcci:
- **Segmentazione**: suddivide la memoria in parti di dimensioni variabili (problema: **frammentazione**).
- **Paging**: suddivide la memoria in parti di dimensioni fisse, chiamate pagine. La memoria fisica è vista come un array di slot con dimensioni fisse, chiamati **frame di pagina**; ogni frame può contenere una singola **pagina** di memoria virtuale.



(a) (b)

Figura 32: (a) Simple 64 byte address space, (b) 64 byte address space in 128 byte of memory.

9.1 Vantaggi del paging

I problemi principali da risolvere quando si implementa il paging sono:
- La virtualizzazione della memoria.
- La scelta della dimensione delle pagine.
- La gestione dei conflitti di pagina.

Vantaggi paging:
- **Flessibilità**: il sistema sarà in grado di supportare l'astrazione di un'address space indipendentemente da come un processo utilizza lo spazio degli indirizzi.
- **Semplicità di gestione dello spazio libero**: quando l'OS vuole inserire l'AS di un processo, trova le pagine libere (prende le prime pagine della free list).

9.2 Page table

Page table: struttura dati presente nella PCB del processo, memorizza le traduzioni degli indirizzi per ogni pagina virtuale dell'AS. La page table è una struttura dati per-process. Se un altro processo dovesse essere mandato in esecuzione, l'OS dovrebbe gestire una page table differente per esso, siccome le sue VP ovviamente sono mappate in frame differenti. Non memorizziamo le page table su un chip dell'MMU per via della loro grandezza. La page table è una struttura dati che viene utilizzata per mappare gli VA (VPN) agli indirizzi fisici (PFN).

Quando un processo genera un indirizzo virtuale, l'OS e l'hardware, lo traducono in indirizzo fisico.

Il processo divide l'indirizzo virtuale in due componenti:
- **VPN** (Virtual Page Number): identifica la pagina virtuale.
- **Offset**: identifica il byte all'interno della pagina.

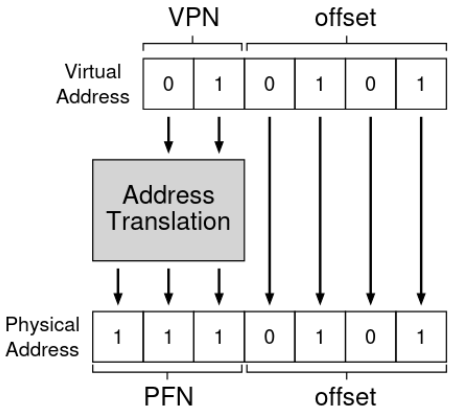


Figura 33: The address translation process.

9.3 Esempio Paging model

Abbiamo:
- Address Space: 32 bit quindi $2^{32} = 4$ GigaByte
- Page Size = 4 Kb

1. Calcolo le dimensioni dell'offset:
   $\text{offset} = \log_2 4096 = 12$ bit

2. Calcolo dimensione della VPN:
   $\text{VPN} = 32 - 12 = 20$ bit

3. Verifico i calcoli:
   $2^{20} \cdot 4096$ KB $= 4$ GB di spazio indirizzabile

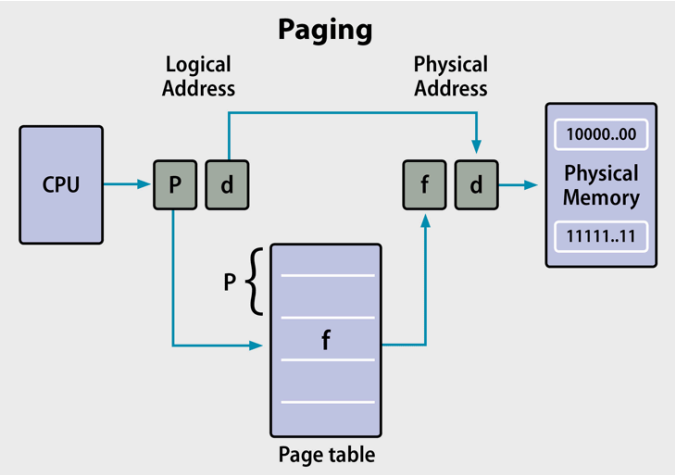


Figura 34: Schema di traduzione. [3]

Le page table possono diventare molto grandi delle segment table o delle coppie base/limiti. In un AS a 32 bit con pagine da 4Kb. Questo VA si divide in un VPN a 20 bit e un offset a 12 bit.

Un VPN a 20 bit implica che l'OS debba gestire 2^{20} traduzioni per ogni processo (circa un milione); assumendo che siano necessari 4 byte (32 bit) per PTE (Page Table Entry) per contenere la traduzione fisica, si ottiene 4MB ($2^{20} \cdot 4$ byte) di memoria necessari per ogni page table. Se i processi in esecuzione sono 100: l'OS ha bisogno di 400MB di memoria solo per tutte quelle traduzioni di indirizzi.

9.4 Page Table Entry (PTE)

Linear Page Table: forma più semplice tra le tipologie di page table. È un array, l'OS indicizza l'array in base al VPN e cerca la PTE in corrispondenza di tale indice per trovare il PFN desiderato.

Bit in ogni PTE:
- **Present bit (P)**: indica se questa pagina è in memoria.
- **Reference bit** o **access bit (A)**: se il bit è impostato, significa che la pagina è stata acceduta di recente e deve essere mantenuta in memoria fisica. Se il bit non è impostato, significa che la pagina non è stata acceduta di recente e può essere spostata fuori dalla memoria fisica se necessario.
- **Dirty bit (D)**: indica se la pagina è stata modificata da quando è stata caricata in memoria. Se la page viene modificata deve essere scritta su disco quando il processo termina.
- **Read/Write (R/W)**: Indica se le letture e/o scritture sono consentite su quella pagina.
- **User/Supervisor bit (U/S)**: indica se i processi in user mode possono accedere alla pagina.
- **Valid bit**: indica se la particolare traduzione è valida (es. spazio tra stack e heap non valido). Supporta l'AS sparso marcando tutte le pagine non utilizzate nell'AS come non valide, e quindi rimuovendo la necessità di allocare frame fisici per tali pagine (risparmiamo memoria).
- **Protection bit**: indica se la pagina può essere letta, scritta o eseguita.
- **PWT, PCD, PAT e G**: determinano come funziona la memorizzazione in cache.

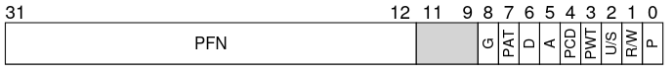


Figura 35: x86 PTE.

9.5 Problemi del paging

Il paging può rallentare il sistema.

Esempio:

```yasm
movl 21, %eax
```

Per fare ciò, l'hardware deve sapere dove si trova la page table per il processo in esecuzione. Supponiamo che un singolo **registro di base della page table** contenga il PA della posizione iniziale della page table.

Una volta che il PA è noto, l'hardware recupera la PTE dalla memoria, estrae il PFN e lo concatena all'offset del VA formando il PA.

```c
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid === False)
RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) === False)
RaiseException(PROTECTION_FAULT)
else
// Access is OK: form physical address and fetch it
offset = VirtualAddress & OFFSET_MASK
PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
Register = AccessMemory(PhysAddr)
```

Codice 2: Implementazione del paging.

9.6 Traduzione da VA a PA con il paging

Dati:
- AS_size $= 16k$
- phys_mem_size $= 64k$
- page_size $= 4k$

Page Table:
[0] `0x8000000c`
[1] `0x00000000`
[2] `0x00000000`
[3] `0x80000006`

Virtual address da tradurre:
- VA_1 = `0x00003229`
- VA_2 = `0x00001369`

Traduzione 1

$VA_1 =$ `0x00003229`

1. Calcolo di quanti bit occorrono per rappresentare il VA:
   $\text{size\_of\_VA} = \log_2 \text{AS\_size} = \log_2 16k \cdot 1024 = 14$ bit
2. Calcolo delle pagine presenti nell'AS:
   $\text{n\_of\_pages} = \frac{\text{AS\_size}}{\text{page\_size}} = \frac{16}{4} = 4$
3. Calcolo dei bit necessari per il VPN:
   $\text{n\_of\_bit\_for\_VPN} = \log_2 \text{n\_of\_pages} = \log_2 4 = 2$ bit
4. Calcolo dei bit che occorrono all'offset:
   $\text{n\_of\_bit\_for\_offset} = \text{size\_of\_VA} - \text{n\_of\_bit\_for\_VPN} = 14 - 2 = 12$ bit
5. Estrazione del VPN:
   1. Conversione in binario del VA:
      $\text{bin(VA)} =$
      $\text{bin(0x00003229)} = 11\ 0010\ 0010\ 1001$
   2. Estrazione del PFN e dell'offset:
      $\underbrace{11}_{\text{VPN}}\ \underbrace{0010\ 0010\ 1001}_{\text{Offset}}$
      - $\text{dec}(11) = 3$ quindi VPN = 3
      - $\text{dec}(0010\ 0010\ 1001) = 553$ quindi offset = 553
6. Check validità della PTE:
   $\text{bin(most\_significant\_nibble\_of\_PFN)} = \text{bin(0x8)} = 1000$
   quindi il MSB = 1, la PTE è valida.
7. Estrazione del PFN:
   $\text{bin(PFN)} = \text{bin(0x80000006)} =$
   $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111$

   Escludendo il byte più significativo che contiene il valid bit, rimane 0110 che tradotto in decimale è 6. Quindi abbiamo che PFN = 6.
8. Calcolo del PA:
   $\text{PA} = 2^{\text{n\_of\_bit\_for\_offset}} \cdot \text{PFN} + \text{offset} = 2^{12} \cdot 6 + 553 = 25129$

Traduzione 2

$VA_2 =$ `0x00001369`

Tutto uguale a prima fino al punto 4.

1. Estrazione del VPN:
   1. Conversione in binario del VA:
      $\text{bin(VA)} =$
      $\text{bin(0x00001369)} = 01\ 0011\ 0110\ 1001$
   2. Estrazione del PFN e dell'offset:
      $\underbrace{01}_{\text{VPN}}\ \underbrace{0011\ 0110\ 1001}_{\text{Offset}}$
      - $\text{dec}(01) = 1$ quindi VPN = 1
      - $\text{dec}(0011\ 0110\ 1001) =$ quindi offset = 873
2. Check validità della PTE:
   $\text{bin(most\_significant\_nibble\_of\_PFN)} = \text{bin(0x0)} = 0000$
   quindi il MSB = 0, la PTE è valida.

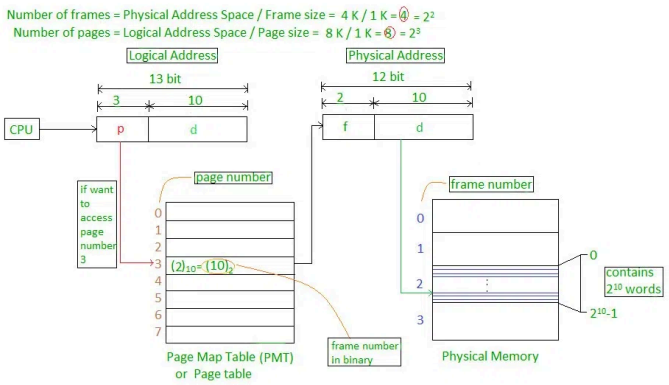


Figura 36: Come funziona la paginazione. [4]

10 Translation Lookaside Buffer

Siccome le informazioni di mappatura risiedono generalmente in memoria fisica, la paginazione richiede un accesso aggiuntivo per ogni indirizzo virtuale generato dal programma. È necessario consultare la page table (in memoria centrale), prima che un'istruzione sia effettivamente prelevata (fetch) o prima che avvenga un accesso esplicito (load-store), il che rende il meccanismo della paginazione poco performante.

L'obiettivo è snellire la tecnica introdotta, cercando di diminuire il numero di accessi a memoria fisica (alla page table).

TLB (Translation Lookaside Buffer): cache hardware che risiede nell'MMU e contiene le traduzioni da virtuali a fisiche più popolari. Sono utilizzate per accelerare la traduzione da VA a PA e per ridurre gli accesi alla page table. Quando il processore deve accedere a un indirizzo virtuale, prima controlla la TLB per vedere se la traduzione è presente. Se la traduzione è presente, il processore può accedere direttamente alla memoria fisica senza dover consultare la tabella delle pagine.

10.1 Vantaggi e svantaggi TLB

Vantaggi delle TLB
- Le TLB possono accelerare la traduzione dell'indirizzo virtuale in indirizzo fisico.
- Le TLB possono ridurre il numero di accessi alla memoria.
- Le TLB possono migliorare le prestazioni dei sistemi che utilizzano la virtualizzazione della memoria.

Svantaggi delle TLB
- Le TLB sono cache e quindi possono contenere traduzioni non aggiornate.
- Le TLB possono essere di dimensioni limitate e quindi non possono memorizzare tutte le traduzioni.
- Le TLB possono essere inefficienti per le traduzioni che vengono utilizzate raramente.

10.2 Funzionamento del TLB

```c
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success === True) // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) === True)
      Offset = VirtualAddress & OFFSET_MASK
      PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
      Register = AccessMemory(PhysAddr)
    else
      RaiseException(PROTECTION_FAULT)
else // TLB Miss
  PTEAddr = PTBR + (VPN * sizeof(PTE))
  PTE = AccessMemory(PTEAddr)
  if (PTE.Valid === False)
    RaiseException(SEGMENTATION_FAULT)
  else if (CanAccess(PTE.ProtectBits) === False)
    RaiseException(PROTECTION_FAULT)
  else
    TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
    RetryInstruction()
```

Codice 3: Implementazione del TLB.

Algoritmo TLB di base:
1. Estrai il numero di pagina virtuale (VPN) dall'indirizzo virtuale.
2. Controlla se il TLB contiene la traduzione per questo VPN.
3. Se lo fa, abbiamo una **TLB hit**, che significa che il TLB contiene la traduzione.
   - Estraiamo il PFN della TLBE pertinente, concatenalo sull'offset dal VA originale e forma il PA, e accedi alla memoria.
4. Se la CPU non trova la traduzione nel TLB (**TLB miss**)
   - L'hardware accede alla page table per trovare la traduzione.
   - Supponendo che la referenza di memoria virtuale generata dal processo sia valida e accessibile, aggiorna il TLB con la traduzione (azioni costose).

5. Aggiornato il TLB, l'hardware riprova l'istruzione e la traduzione si trova nel TLB quindi la referenza di memoria viene elaborata rapidamente.

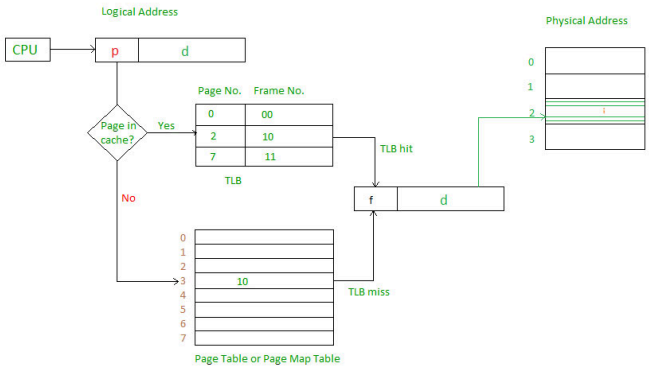


Figura 37: Schema di funzionamento delle TLB. [5]

10.3 Gestione dei TLB miss

Le TLB miss possono causare un rallentamento del programma se si verificano frequentemente (CPU deve accedere alla page table per trovare la traduzione).

La prima volta che il programma accede a un elemento dell'array di 10 elementi, si verifica un TLB miss, perché la traduzione non è ancora memorizzata nel TLB. Tuttavia, le successive accessi all'array causeranno dei TLB hit, perché le traduzioni degli elementi successivi si trovano sulla stessa pagina di memoria della prima traduzione. il TLB ha un hit rate del 70% (valore ragionevole). Se il programma accedesse nuovamente all'array, il TLB hit rate sarebbe probabilmente più alto, perché le traduzioni degli elementi dell'array sarebbero già memorizzate nel TLB.

Spatial locality: tendenza degli elementi di un array a essere contigui in memoria.

Temporal locality: tendenza di un programma a riutilizzare gli stessi indirizzi di memoria più volte. Es. ciclo for.

Il processore può gestire le TLB miss in due modi:
- **Hardware-managed**: il processore è responsabile della gestione delle TLB miss. La CPU accede alla page table e trova la traduzione, aggiorna il TLB con la traduzione e rieseguire l'istruzione che ha causato la TLB miss.
- **Software-managed**: La CPU solleva un'eccezione quando si verifica la TLB miss. Il processore passa il controllo all'OS, che è responsabile della gestione della TLB miss. L'OS accede alla tabella delle pagine per trovare la traduzione, aggiorna il TLB con la traduzione e quindi restituisce il controllo al processore (flessibile ma più lento).

Le architetture moderne hanno il TLB software-managed. In caso di TLB miss, l'hardware solleva l'eccezione, mette in pausa il flusso di istruzioni, aumenta il privilegio e va al trap handler.

Il return-from-trap ha due scopi:
- Ripristinare il contesto del processore come prima dell'interruzione.
- Riprendere l'esecuzione del processo.

Il return-from-trap è diverso per le TLB miss e per le system call:
- Nelle TLB miss, deve riprendere l'esecuzione all'istruzione che ha causato la TLB miss perché dovrà eseguire di nuovo l'istruzione che ha bisogno della traduzione appena aggiornata.
- Nelle system call, deve riprendere l'esecuzione all'istruzione dopo l'interruzione perché la CPU ha già eseguito le istruzioni necessarie per la system call e non ha bisogno di ripeterle.

Se il codice di gestione delle TLB miss provoca una TLB miss, si può creare una catena infinita di TLB miss. Per evitare questo problema, l'OS può:
- Conservare il codice di gestione delle TLB miss in memoria fisica (dove non è soggetto a traduzione di indirizzi).
- Risolvere alcune TLBE per traduzioni sempre valide e utilizzare alcune di queste voci per il codice di gestione delle TLB miss.

Vantaggi dell'approccio software-managed:
- È più flessibile, perché l'OS può utilizzare qualsiasi struttura dati per implementare la tabella delle pagine.
- È più semplice, perché l'hardware non deve fare molto quando si verifica una TLB miss.

10.4 Struttura e implementazione

Valid bit: indica se una traduzione è memorizzata nel TLB. Quando un sistema viene avviato, i valid bit del TLB sono impostati su invalid, perché non ci sono traduzioni ancora memorizzate nel TLB. Una volta che i processi iniziano a funzionare e accedono ai loro spazi di VA, il TLB viene lentamente popolato e i valid bit vengono impostati su valid.

I valid bit sono utili anche durante i context switch. Quando si verifica un context switch, l'OS imposta tutti i valid bit del TLB su invalid. Ciò assicura che il processo che sta per essere eseguito non utilizzi accidentalmente una traduzione da virtuale a fisica da un processo precedente.

TLB fully associative: qualsiasi traduzione può trovarsi in qualsiasi posizione nella TLB.

Campi TLBE :
- **VPN (Virtual Page Number)**: il numero di pagina virtuale
- **PFN (Physical Page Number)**: il numero di pagina fisica
- **Altri bit**: es. bit di validità, i bit di protezione e l'identificatore dello spazio di indirizzi.

10.5 Context-switching e TLB

Il TLB contiene le traduzioni di indirizzi relative a un determinato processo. Quando avviene un context-switch bisogna assicurarsi che un processo usi le traduzioni corrette.

Occorre svuotare il TLB (**flush**) ad ogni cambio di contesto:
- **Approccio software**: ottenuto tramite un'istruzione hardware privilegiata.
- **Approccio hardware**: ottenuto quando si va a modificare il PTBS (Page Table Base Register)

In entrambi i casi per lo svuotamento si impostano i bit di validità a zero. Ogni volta che viene eseguito un processo questo incorre in errori TLB quando tocca i suoi dati e le tabelle dei codici (costo elevato).

Quando si passa da un processo all'altro, le traduzioni nel TLB del processo precedente non sono più valide per il processo successivo. Ci sono due possibili soluzioni a questo problema:
1. Svuotare completamente il TLB su ogni cambio di contesto. Semplice ma costoso perché ogni processo deve ricaricare le sue traduzioni dal tabella delle pagine quando viene eseguito.
2. Aggiungere un campo di identificatore di spazio di indirizzi (**ASID**) al TLB. L'ASID è un numero che identifica univocamente un processo. Con l'ASID, il TLB può memorizzare traduzioni da diversi processi contemporaneamente, purché le traduzioni per un processo specifico abbiano lo stesso ASID.

Quando installiamo una nuova entry nel TLB si va a sostituire con una vecchia entry:
- **LRU** (least-recently-used): sostituisce la nuova entry con quella utilizzata meno recentemente.
- **Random Policy**: sostituisce una vecchia entry a caso con una nuova.

11 Small TLBS

11.1 Problemi delle Page Table

Le page table occupano tanta memoria, la soluzione è quella di creare pagine più grandi.

La soluzione più semplice per ridurre le dimensioni della page table consiste nell'utilizzare pagine di dimensioni maggiori. Considerando un AS a 32 bit e pagine da 16 KB, si otterrebbe una riduzione del fattore quattro delle dimensioni della tabella delle pagine. Otteniamo:
- VPN a 18 bit
- Offset a 14 bit ($16k \cdot 1024 = 2^{14}$)

Questa soluzione soffre di **frammentazione interna** all'interno di ciascuna pagina, poiché le applicazioni potrebbero utilizzare solo parti delle pagine e riempire la memoria con pagine grandi.

Alcune architetture supportano dimensioni multiple delle pagine. Questo approccio è utile per ottimizzare l'accesso a strutture dati grandi e frequentemente utilizzate, riducendo la pressione sulla TLB. L'implementazione di dimensioni multiple delle pagine rende il gestore di memoria virtuale dell'OS complesso.

11.2 Combinazione di segmentation e paging

Utilizzando un approccio ibrido, si combina il paging e la segmentation. Viene creata una page table per ogni segmento logico (es. heap, stack, code).

Composizione della segment table:
- **Registro base**: non contiene più l'indirizzo entry del segmento logico, qui contiene il PA della page table del segmento.
- **Registro bound**: non è più la dimensione della pagina ma il valore della massima pagina valida nel segmento.

Durante l'esecuzione di un processo, il registro base per ciascun segmento contiene l'indirizzo fisico di una tabella delle pagine lineare per quel segmento. Quando avviene un cambio di contesto, questi registri devono essere aggiornati per riflettere la posizione delle tabelle delle pagine del nuovo processo in esecuzione.

L'approccio ibrido può ridurre l'overhead di memoria delle page table in due modi:
- Il numero di voci nella tabella delle pagine può essere ridotto, perché le pagine che appartengono allo stesso segmento possono essere mappate nello stesso frame di memoria.
- Le voci nella tabella delle pagine possono essere più piccole, perché non è necessario memorizzare il numero di segmento per ogni pagina.

In caso di TLB MISS, l'hardware utilizza i bit di segmento SN per determinare quale coppia base and bounds utilizzare. L'hardware prende il PA in base e lo combina con il VPN per formare l'indirizzo della PTE

```txt
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
SN = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
```

In questo modo le pagine non allocate tra stack e heap non occupano più spazio in una PT.

Criticità dell'approccio ibrido: La segmentazione può causare **frammentazione esterna**, poiché i segmenti hanno dimensioni variabili. Questo porta a spazi di memoria liberi ma non contigui, che potrebbero essere insufficienti per soddisfare le richieste di nuovi processi, rendendo tali spazi inutilizzabili e aumentando l'inefficienza complessiva del sistema

11.3 Multi-Level Page Table

Per risolvere il problema delle regioni di memoria non valide nella page table si utilizzano le tabelle delle pagine multilivello. Questo approccio trasforma la page table lineare in una struttura simile a un albero, migliorando l'efficienza e riducendo gli sprechi di memoria.

- La page table viene suddivisa in unità page-sized.
- Se una pagina di una PTE è invalida, non viene allocata.

Page directory: indica quali pagine della page table sono valide. Contiene un'entry PDE (Page Directory Entry) per ogni page della PT. Ogni PDE ha:
- **Bit di validita**: se PDE è valida significa che almeno una delle PTE nelle pagine della PT a cui punta la PDE tramite il PFN, è valida.
- **PFN** (Page Frame Number): indirizzo di memoria dove è situata una page table.

Mentre la page table lineare richiede spazio per regioni non valide, la multi-level page table può «far sparire» parti della tabella delle pagine, liberando quei frame per altri utilizzi e tracciando le pagine allocate con il page directory.

Vantaggi:
- Alloca spazio solamente per la page table in proporzione all'ammontare di AS in uso (supporta quindi address space sparsi).
- Se implementata correttamente, ogni porzione della PT entra ordinatamente in una pagina, rendendo più facile la gestione della memoria; il sistema operativo prende semplicemente la prossima pagina libera quando ha bisogno di allocare o far crescere una PT.

Svantaggi:
- In caso di TLB MISS sono necessari due accessi a memoria (uno per la PDE e l'altro per la PTE) mentre il caso di TLB HIT le prestazioni corrispondono a quelle di una tabella lineare.
- Struttura più complessa rispetto ad una tabella lineare.

Il multi-level page table rappresenta un compromesso tra tempo e spazio.

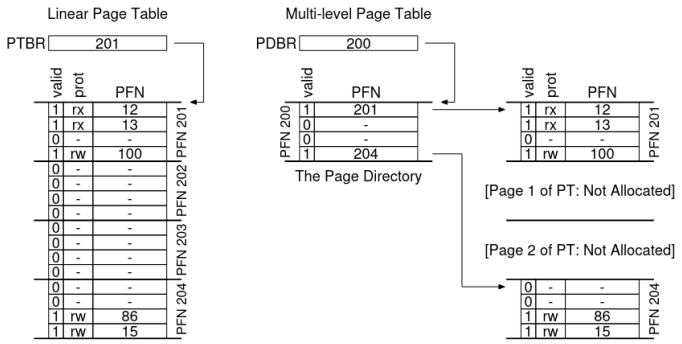


Figura 38: Linear (Left) And Multi-Level (Right) Page Tables.

11.4 Multi-level Page Table a più livelli

Quando lo spazio di indirizzamento aumenta, anche la dimensione della directory delle pagine cresce, rendendo inefficiente l'allocazione di memoria. Per ovviare a questo problema, si aggiungono ulteriori livelli alla tabella delle pagine, creando una struttura gerarchica più profonda. Questo approccio consente di suddividere l'indirizzo virtuale in più segmenti, ciascuno dei quali indirizza un livello specifico della tabella, garantendo che ogni segmento della tabella delle pagine occupi esattamente una pagina di memoria fisica.

Ad esempio, con uno spazio di indirizzamento di 30 bit e pagine di 512 byte, l'indirizzo virtuale può essere suddiviso in:
- 9 bit per l'offset all'interno della pagina.
- 7 bit per l'indice della tabella delle pagine.
- 14 bit per l'indice della directory delle pagine.

Se la directory delle pagine diventa troppo grande per essere contenuta in una singola pagina fisica, si introduce un ulteriore livello di directory, suddividendo la directory originale in più pagine e aggiungendo una directory di livello superiore. Questo processo può essere ripetuto, aggiungendo più livelli, fino a garantire che ogni segmento della tabella delle pagine possa

essere contenuto in una singola pagina fisica, ottimizzando così l'utilizzo della memoria e migliorando l'efficienza del sistema.

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success === True)
  // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) === True)
  Offset
  = VirtualAddress & OFFSET_MASK
  PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
  Register = AccessMemory(PhysAddr)
  else
  RaiseException(PROTECTION_FAULT)
else
  // TLB Miss
  // first, get page directory entry
  PDIndex = (VPN & PD_MASK) >> PD_SHIFT
  PDEAddr = PDBR + (PDIndex * sizeof(PDE))
  PDE = AccessMemory(PDEAddr)
  if (PDE.Valid === False)
    RaiseException(SEGMENTATION_FAULT)
  else
    // PDE is valid: now fetch PTE from page table
    PTIndex = (VPN & PT_MASK) >> PT_SHIFT
    PTEAddr = (PDE.PFN<<SHIFT) + (PTIndex*sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
  if (PTE.Valid === False)
    RaiseException(SEGMENTATION_FAULT)
  else if (CanAccess(PTE.ProtectBits) === False)
    RaiseException(PROTECTION_FAULT)
  else
    TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
    RetryInstruction()
```

Codice 4: Implementazione Multi-Level TLB.

11.4.1 Esercizio Multi-Level Page Table

Dati:
- Address space: 16KB
- Dimensione pagina: 64 bytes
- Numero totale di pagine: 256 (2^8)
- Indirizzo virtuale: 14 bit (8 bit per VPN, 6 bit per offset)
- PTE size = 4 byte

1. Struttura della Page Table Lineare
   - Ogni PTE è di 4 bytes, quindi, Page Table = 256 · 4 bytes = 1KB
   - Con pagine da 64 bytes, quindi, la Page Table è divisa in 16 pagine (ciascuna contiene 16 PTE)

2. Costruzione della Multi-Level Page Table
   - Page Directory con 16 entry (una per ogni pagina della Page Table)
   - Page Table divisa in 16 pagine da 64 bytes
   - Indice VPN suddiviso:
     - PDIndex: 4 bit
     - PTIndex 4 bit
     - Offset: 8 bit

3. Traduzione
   - Estrarre PDIndex (primi 4 bit di VPN) per trovare la PDE:

     ```
     PDEAddr = PageDirBase + (PDIndex * sizeof(PDE))
     ```

   - Se PDE è valida, ottenere il PFN della Page Table e trovare la PTE:

     ```
     PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
     ```

   - Se PTE è valida, estrarre il PFN della pagina fisica e calcolare l'indirizzo fisico:

     ```
     PhysAddr = (PTE.PFN << SHIFT) + offset
     ```

Esempio di Traduzione
- Indirizzo virtuale: `0x3F80`
  In binario: `11 1111 1000 0000`
- PDIndex = `1111` che corrisponde alla Page Table con PFN `101`
- PTIndex = `1110` 14-esima entry della Page Table con PFN `55`
- Offset = `000000` Indirizzo fisico finale: `0x0DC0`

11.4.2 Esercizio all'esame

Richiesta: calcolare il valore dei 3 offset:
- VPNPageDir
- VPNChunkIndex
- Offset della pagina fisica dove risiede il dato

Dati:
- VirtualAddressSize = 48 bit
- PageSize = 16 Kb = 2^{14}
- VA = `0x12A7FEDB`

Inoltre, avendo 48 bit per l'indirizzamento capiamo che siamo su un sistema a 64 bit.

Implicazioni:
- Con PageSize = 2^{14}, abbiamo 14 bit per l'offset.
- Dato che abbiamo 14 bit per l'offset, i bit per la VPN sono 48 - 14 = 34 bit.
- Siccome siamo su architettura x86-64 quindi a 64 bit, le PTE sono grandi 8 byte (64 bit = 8 byte).

Calcoli:

1. Calcolo il numero di PTE per pagina e PTIndex: $\frac{16\text{ Kb}}{8\text{ byte}} = \frac{16 \cdot 1024}{8} = 2048 = 2^{11}$ PTE per pagina. Questo implica che occorrono 11 bit per il PTIndex.

2. Calcolo il numero di bit per il PDIndex: Siccome abbiamo 11 bit per PTIndex, il PDIndex avrà bisogno di 48 - 14 - 11 = 23 bit.

Ricapitolando:
- VPN = 34 bit
  - PDIndex = 23 bit
  - PTIndex = 11 bit
- offset = 14 bit

1. Traduzione:
   `0x12A7FEDB` $_{16}$ =
   `0001 0010 1010 0111 1111 1110 1101 1011`$_2$

   Ottengo che
   - PDIndex = `1001`
   - PTIndex = `010 1001 1111`
   - offset = `11 1110 1101 1011`

   Tradotti in esadecimale:
   - PDIndex = `0x09`
   - PTIndex = `0x029F`
   - offset = `0x3EDB`

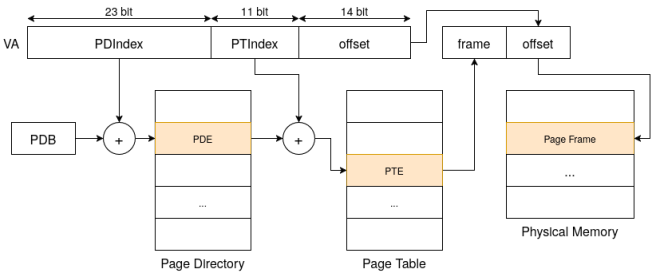


Figura 39: Schema di memoria di un Multi-Level TLB a due livelli su architettura x86-64 con page size = 16 Kb. Schema creato dall'autore.

12 Swapping: Mechanisms

Per supportare AS di grandi dimensioni, l'OS avrà bisogno di posizionare altrove quelle pagine che non sono largamente richieste. Quindi occorre molta memoria che comporta l'utilizzo di un disco rigido grande, dunque, molto lento. Avremmo prestazioni scarse per spostare le pagine dalla memoria centrale a quella di massa e viceversa.

12.1 Swapping

Swap space: spazio sulla memoria di massa per spostare le pagine avanti e dietro.

L'OS legge e scrive nello space swap usando la dimensione di una pagina come unità. L'OS deve ricordare l'indirizzo su disco di una determinata pagina. La dimensione dello space swap determina il numero massimo di pagine in memoria che possono essere usate da un sistema in un determinato momento.

1. **Generazione VA**: Il processo in esecuzione genera dei riferimenti virtuali a memoria (per prelevare istruzioni o accedere dati).
2. **Consultazione della TLB**:L'hardware prima di estrarre la VPN del VA controlla la TLB:
   - **TLB HIT**: produce il PA e lo recupera il contenuto in memoria.
   - **TLB MISS**: l'hardware individua la PT in memoria e cerca la PTE per la pagina utilizzando il VPN come indice.
     - Se la pagina è valida ed è presente nella memoria fisica, l'hardware estrae il PFN dal PTE, lo installa nel TLB e riprova l'istruzione e questa volta genererà un TLB HIT.
     - Se la pagina non è valida significa che non è presente nella memoria fisica. Avviene il **page fault**.

12.2 Page fault

Per gestire il TLB MISS ci sono 2 sistemi:
- **Sistema gestito dall'hardware**: l'hardware cerca nella PT la traduzione desiderata.
- **Sistema gestito dal software**: l'OS cerca nella PT la traduzione.

In entrambi i casi se una pagina non è presente, l'OS deve gestire il page fault (se una pagina non è presente in memoria centrale l'OS dovrà riportare la pagina dalla memoria di massa a quella centrale per risolvere il page fault). L'OS per trovare la pagina utilizzerà i bit della PTE relativi al PFN. L'OS quando si verifica un page fault cerca nella PTE per trovare l'indirizzo e inviare la richiesta della pagina al disco fisico. Al termine, l'OS aggiornerà la PT con il presence bit della relativa pagina impostato a 1, aggiornerà il campo PFN della PTE per registrare la posizione in memoria e infine riproverà l'istruzione fallita.

Mentre c'è I/O il processo è in block state e il processore sarà libero per eseguire altri processi (**overlap**).

La memoria potrebbe essere piena e l'OS potrebbe aver bisogno di una pagina o più, in questo caso occorre sostituire le pagine che non sarebbero utili. Questa politica si chiama **page-replacement policy**.

12.3 Control Flow del page fault

```c
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success === True) // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) === True)
    Offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)
else // TLB Miss
  PTEAddr = PTBR + (VPN * sizeof(PTE))
  PTE = AccessMemory(PTEAddr)
  if (PTE.Valid === False)
    RaiseException(SEGMENTATION_FAULT)
  else
    if (CanAccess(PTE.ProtectBits) === False)
      RaiseException(PROTECTION_FAULT)
    else if (PTE.Present === True)
      // assuming hardware-managed TLB
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
      RetryInstruction()
    else if (PTE.Present === False)
      RaiseException(PAGE_FAULT)
```

Codice 5: Page-Fault Control Flow Algorithm (Software).

Quando si verifica un TLB MISS, ci sono tre casi:
1. La pagina è sia presente che valida (righe 18-20). In questo caso, il TLB miss handler prende semplicemente il PFN dalla PTE e riprova l'istruzione ottenendo un TLB hit.
2. Il page fault handler deve essere eseguito; sebbene la pagina sia valida, non è presente in memoria fisica (righe 22-23).
3. L'accesso potrebbe non essere valido (righe 13-14). In questo caso nessun altro bit della PTE viene preso in considerazione. L'hardware esegue una trap e l'opportuno trap handler dell'OS viene eseguito, terminando il processo.

```c
PFN = FindFreePhysicalPage()
if (PFN === -1) // no free page found
  PFN = EvictPage() // replacement algorithm
DiskRead(PTE.DiskAddr, PFN) // sleep (wait for I/O)
PTE.present = True // update page table:
PTE.PFN = PFN // (present/translation)
RetryInstruction() // retry instruction
```

Codice 6: Page-Fault Control Flow Algorithm (Software).

L'OS per gestire il page fault deve:
1. Trovare il frame fisico per far risiedere la pagina "soon-to-be-faulted-in".
2. Se tale frame non c'è, dobbiamo aspettare per che l'algoritmo di replacement venga eseguito e liberi alcune pagine per riutilizzarle.

Parte hardware del controllo del flusso di pagina Quando il processore tenta di accedere a una pagina di memoria, prima interroga la tabella di traduzione della pagina (TLB) per vedere se la pagina è presente in memoria fisica. Se:
1. **present bit = 1**: la CPU ottiene il numero del frame fisico della pagina dalla TLB e procede ad accedere alla pagina.
2. **present bit = 0**: la CPU genera un errore di pagina e il controllo passa alla parte software del controllo del flusso di pagina.

Parte software del controllo del flusso di pagina Quando l'OS riceve un errore di pagina:
1. Deve trovare un frame fisico libero per la pagina.
   - Se non è disponibile alcun frame fisico libero, l'OS deve eseguire un algoritmo di sostituzione di pagina per rimuovere una pagina dalla memoria fisica.
2. L'OS legge la pagina dalla memoria di swap nel frame fisico.
3. L'OS aggiorna la tabella di traduzione della pagina per indicare che la pagina è ora presente in memoria fisica.

12.4 Replacement delle pagine

I'OS attende fino a quando la memoria non è completamente piena e solo allora sostituisce una pagine per far spazio ad una nuova. Ma gli OS mantengono un po' di memoria libera per:
- Eseguire gli algoritmi di sostituzione di pagina.
- Evitare che il sistema operativo si blocchi a causa di un errore di pagina.
- Consentire ai processi di allocare nuove pagine di memoria.

Per mantenere una quantità minima di memoria libera l'OS utilizza **HW** (high watermark) e **LW** (low watermark) che aiutano a decidere quando iniziare a rimuovere le pagine dalla memoria. Quando il numero di pagine di memoria libere scende al di sotto del valore LW, l'OS avvia un thread in background (**swap daemon** o page daemon) che si occupa di liberare memoria. Il thread in background esegue l'algoritmo di sostituzione di pagina fino a quando il numero di pagine di memoria libere non supera il valore HW. Per supportare il thread in background, il controllo del flusso di pagina deve essere modificato.

L'OS controlla costantemente il numero di pagine libere in memoria. Quando le pagine libere scendono sotto la soglia LW:
1. **Attivazione del Swap Daemon**: lo swap daemon viene attivato per iniziare il processo di liberazione della memoria.
2. **Liberazione della Memoria**: lo swap daemon identifica le pagine meno utilizzate e le trasferisce su disco, liberando spazio in memoria fisica. Questo processo può includere la scrittura di gruppi di pagine sul disco in un'unica operazione, migliorando l'efficienza grazie all'accesso sequenziale al disco.
3. **Conclusione**: quando il numero di pagine libere raggiunge o supera la soglia HW, il swap daemon termina la sua attività e ritorna in stato di inattività, pronto a riattivarsi se necessario.

13 Swapping: Policies

Quando l'OS ha poca memoria a disposizione per le pagine deve sostituire le pagine con quelle utilizzate attivamente. Per fare ciò occorre una replacement policy.

13.1 Access time

La memoria centrale contiene un sottoinsieme di pagine e può essere vista come una cache per le pagine di memoria virtuale. L'obiettivo è ridurre al minimo il numero di **cache misses** (volte in cui dobbiamo recuperare una pagina dal disco perché non è presente in memoria). Possiamo calcolare l'average memory access time (**AMAT**) per un programma:

$$\text{AMAT} = T_M + (P_{\text{Miss}} \cdot T_D)$$

- T_M: è il costo di accesso alla memoria.
- T_D: è il costo di accesso al disco.
- P_{Miss}: è la probabilità di non trovare i dati nella cache (da 0 a 1).

13.2 Classificazione dei miss

Gli architetti di computer classificano i miss in base al tipo, in una delle tre categorie:

- **Compulsory miss**: si verifica perché la cache è vuota all'inizio e questa è la prima istanza di riferimento all'elemento.
- **Capacity miss**: si verifica perché la cache ha esaurito lo spazio e ha dovuto espellere un elemento per portare un nuovo elemento nella cache.
- **Conflict miss**: si verifica nell'hardware a causa dei limiti su dove un elemento può essere posizionato in una cache hardware, a causa di qualcosa noto come set-associative; non si verifica nella cache di pagina dell'OS perché tali cache sono sempre fully-associative, cioè non ci sono restrizioni su dove una pagina può essere posizionata in memoria.

13.3 Politiche di replacement

Optimal Replacement Policy: politica che porta al minor numero di errori di pagina e sostituisce la pagina che verrà acceduta più lontano nel futuro. È difficile da implementare, ma è un buon punto di riferimento per le altre politiche di sostituzione.

FIFO (first-in, first-out) policy: politica di sostituzione di pagina semplice che sostituisce la pagina che è stata caricata per prima in memoria. È facile da implementare, ma non è molto efficiente, poiché può espellere pagine che sono state recentemente accedute.

Random policy: politica che sceglie una pagina da sostituire in modo casuale. È semplice da implementare, ma non è molto efficiente, poiché può espellere pagine che sono state recentemente accedute. Può avere prestazioni migliori o peggiori del FIFO, a seconda della sequenza di accesso alle pagine.

Utilizzo della cronologia:
Per migliorare la previsione del futuro, possiamo utilizzare la storia come guida. Se una pagina è stata acceduta di recente, è probabile che venga acceduta di nuovo a breve.

Una politica di paginazione può utilizzare due tipi di informazioni storiche:
- **Frequenza di accesso**
- **Attualità di accesso**

La famiglia di politiche basate sull'attualità di accesso è basata sul **principio di località**, che afferma che i programmi tendono ad accedere a determinate sequenze di codice e strutture dati in modo molto frequente. In dettaglio, sono:
- **LFU policy** (Least Frequently Used): sostituisce la pagina che è stata utilizzata meno frequentemente.
- **LRU policy** (Least Recently Used): sostituisce la pagina che è stata utilizzata meno di recente.

13.4 LRU approssimato

È possibile fare un approssimazione per trovare le pagine utilizzate meno recentemente. Approssimando LRU otteniamo un algoritmo più flessibile e meno costoso ed è la tecnica che i moderni sistemi adottano. Per fare ciò occorre supporto hardware.

Use bit o reference bit: è contenuto in ogni pagina e ogni volta che una di esse viene riferita, lo use bit è settato dall'hardware a 1.

L'hardware non pulisce mai lo use bit, è l'OS che ha il compito di settarlo a 0.

Il **Clock algorithm** è utilizzato per approssimare l'LRU. Funzionamento dell'Algoritmo Clock:

1. Struttura Circolare: Le pagine in memoria sono organizzate in una lista circolare, simile al quadrante di un orologio, con una «lancetta» che punta a una pagina specifica.

2. Processo di Sostituzione:
   - Quando è necessario sostituire una pagina, il sistema operativo controlla la pagina indicata dalla lancetta.
   - Se il bit di riferimento della pagina è 1, significa che la pagina è stata utilizzata di recente. In questo caso, il sistema operativo:
     - Resetta il bit di riferimento a 0.
     - Sposta la lancetta alla pagina successiva.
   - Questo processo continua finché non si trova una pagina con il bit di riferimento a 0, indicante che non è stata utilizzata di recente e può essere sostituita.

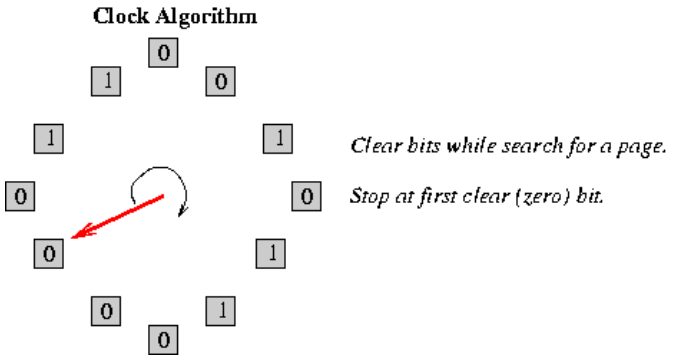


Figura 40: Clock algorithm.

Per migliorare ulteriormente l'efficienza, l'algoritmo può considerare anche il bit di modifica (dirty bit), che indica se una pagina è stata modificata mentre risiedeva in memoria.

- Pagine non modificate (clean): Se una pagina non è stata modificata (dirty bit = 0), può essere rimossa senza necessità di scriverla su disco, riducendo il costo dell'operazione.

- Pagine modificate (dirty): Se una pagina è stata modificata (dirty bit = 1), deve essere scritta su disco prima di essere sostituita, comportando un overhead maggiore.

13.5 Trashing

Quando la richiesta di memoria dei processi in esecuzione supera la memoria fisica disponibile, il sistema entra in una condizione di **trashing**, caratterizzata da un eccessivo utilizzo della paginazione che degrada drasticamente le prestazioni.

Per gestire questa situazione, alcuni sistemi adottano meccanismi di controllo avanzati, come l'**admission control**, che limita il numero di processi in esecuzione per garantire un utilizzo più efficiente delle risorse. L'idea alla base è che eseguire meno processi in modo efficace sia preferibile rispetto a eseguirne troppi con prestazioni pessime.

Linux, invece, utilizza un approccio più diretto con l'**Out-Of-Memory (OOM) Killer**: quando la memoria è sovraccarica, questo demone identifica e termina il processo che sta consumando più memoria, liberando risorse e prevenendo il collasso del sistema.

II Concorrenza

1 Concurrency

Thread: è un sottoinsieme delle istruzioni di un processo, che può essere eseguito in maniera concorrente con altre parti di esso. Viene quindi suddiviso un processo in parti differenti, eseguite in maniera indipendente l'una dall'altra. L'obiettivo dei threads è quello di rendere più veloce l'esecuzione di un processo.

1.1 Programmi multi-thread

Programma multi-thread: ha più punti di esecuzione o PC. Per ogni thread ci sarà uno stack (memoria locale del thread).

I thread del programma condividono lo stesso address space e quindi i dati. Ognuno di essi ha il proprio stato che è privato (registri da ripristinare, quando si passa da un thread ad un altro avviene il context switching). Lo stato di questi è salvato nel TCB (Thread Control Blocks).

In un processo multithread, ogni thread viene eseguito in modo indipendente e ha il suo stack.

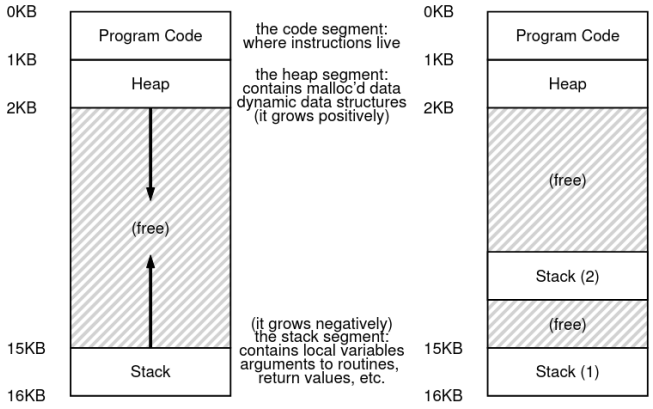


Figura 41: Single-Threaded And Multi-Threaded Address Spaces

1.2 Strutture per i thread

Thread-local storage (TLS): stack relativo a un thread. Consente di memorizzare dati in modo che siano accessibili solo a un thread specifico. Qualsiasi variabile allocata nello stack, parametri, valori di ritorno e altre cose che mettiamo sullo stack andranno nel TLS per separare i singoli stack.

Perché usare i thread:
- **Parallelismo:** i thread possono essere utilizzati per eseguire più attività contemporaneamente, il che può migliorare le prestazioni del programma se il sistema ha più di un processore.
- **Evitare il blocco del programma a causa di I/O lente:** i thread possono essere utilizzati per eseguire altre attività mentre un thread è bloccato in attesa di un'operazione di I/O lenta. (overlap)
- **Facile condivisione dei dati:** I thread condividono lo stesso spazio di memoria, il che rende facile la condivisione dei dati tra di loro.

1.3 Creazione dei thread

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 1) {
            fprintf(stderr, "usage: main\n");
            exit(1);
    }

    pthread_t p1, p2;
    printf("main: begin\n");
        // Attesa completamento dei thread
        // Non è detto che "A" sia stampata prima di "B", dipende dallo scheduler
        // Pthread_create() chiama pthread_create() e si assicura che ritorni zero,
    // in caso contrario viene stampato un messaggio nello stderr.

    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Codice 7: Creazione di thread.

Il programma crea due thread, che eseguono la funzione `mythread()` con argomenti diversi (le stringhe A o B).

- Il thread può iniziare a essere eseguito immediatamente o essere messo in uno stato di «pronto» ma non «in esecuzione» e quindi non ancora eseguito.
- Su un multiprocessore, i thread potrebbero anche essere eseguiti contemporaneamente.
- Il thread principale chiama `Pthread_join()`, che attende il completamento di un particolare thread. Lo fa due volte, assicurando che T1 e T2 vengano eseguiti e completati prima di consentire al thread principale di eseguire di nuovo.

Ecco i possibili ordini di esecuzione del programma:
- Il thread principale viene eseguito prima, seguito da Thread 1 e Thread 2.
- Thread 1 viene eseguito prima, seguito dal thread principale e Thread 2.
- Thread 2 viene eseguito prima, seguito dal thread principale e Thread 1.

Non è possibile sapere in anticipo quale ordine di esecuzione si verificherà, poiché dipende dal programma di schedulazione dell?OS.

Questo esempio mostra che i thread possono rendere difficile prevedere il comportamento di un programma. È importante essere consapevoli di questo quando si scrive codice multithread.

1.4 Dati in condivisione

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#include "common.h"
#include "common_threads.h"

int max;
static volatile int counter = 0; // shared global variable

void *mythread(void *arg) {
    char *letter = arg;
    int i; // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);
    for (i = 0; i < max; i++) {
        counter = counter + 1; // shared: only one
    }
    printf("%s: done\n", letter);
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "
            usage: main-first <loopcount>\n
        ");
        exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: begin [counter = %d] [%x]\n", counter,
    (unsigned int) &counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [counter: %d]\n [should: %d]\n",
    counter, max*2);
    return 0;
}
```

Codice 8: Esempio di codice con dato condiviso.

Il programma crea due thread che tentano di aggiornare una variabile condivisa, counter. La variabile counter è inizialmente impostata a 0.

I thread eseguono un loop in cui incrementano counter di 1. Il numero di volte che il loop viene eseguito è 10 milioni (1e7). Il risultato desiderato è che counter sia impostata a 20 milioni (20000000) alla fine del programma. Tuttavia, il programma non sempre produce il risultato desiderato. A volte, il valore finale di counter è diverso da 20 milioni. Ad esempio, il valore finale può essere 19345221 o 19221041. Il motivo di questo comportamento è che i thread accedono alla variabile counter contemporaneamente. Quando due o più thread accedono alla stessa variabile condivisa, si verifica una condizione nota come **race condition**. In una race condition, il risultato dell'esecuzione del programma dipende dall'ordine in cui i thread accedono alla variabile condivisa.

In questo caso, il risultato dell'esecuzione del programma dipende dall'ordine in cui i thread incrementano counter nel loop. Se il thread A incrementa counter prima del thread B, il valore finale di counter sarà maggiore di 20 milioni. Se il thread B incrementa counter prima del thread A, il valore finale di counter sarà minore di 20 milioni. Per evitare le race condition, è necessario utilizzare meccanismi di sincronizzazione. I meccanismi di sincronizzazione consentono di coordinare l'accesso dei thread alle variabili condivise.

Se un thread dopo aver incrementato il contatore viene interrotto e l'OS fa context switching, parte il secondo thread ed esegue le tre istruzioni salvando l'incremento in memoria. T2

viene interrotto, T1 riprende il controllo ed esegue l'ultima istruzione rimanente ma nel suo stato il contatore vale uno in meno e salva quest'ultimo. In sostanza il codice viene eseguito due volte ma il contatore definitivamente è incrementato solamente di un'unità.

Race condition: condizione di errore che si verifica quando due o più thread accedono alla stessa variabile condivisa contemporaneamente e il risultato dell'esecuzione del programma dipende dall'ordine in cui i thread accedono alla variabile.

Critical section: pezzo di codice che accede ad una variabile condivisa tra i thread.

Esclusione reciproca (mutual exclusion): garantisce che gli altri thread non possano eseguire una critical section se un thread è già in esecuzione in quell'area.

Atomicità: un'operazione o un insieme di operazioni di un programma eseguite interamente senza essere interrotte prima della fine del loro corso. Questo concetto si applica a una parte di un programma di cui il processo o il thread che lo gestisce non cederà il monopolio su determinati dati a un altro processo durante tutto il corso di questa parte.

Transazione: raggruppamento di molte azioni in un'unica azione atomica.

Programma indeterminato: consiste in una o più race condition; l'output del programma varia da esecuzione a esecuzione, a seconda di quali thread sono stati eseguiti quando. Il risultato non è quindi deterministico.

2 Thread API

2.1 `pthread_create()`

In POSIX, la funzione `pthread_create()` viene utilizzata per creare un nuovo thread.

La funzione `pthread_create()` ha quattro argomenti:
- `thread`: puntatore a una struttura che verrà utilizzata per interagire con il nuovo thread.
- `attr`: puntatore a una struttura che può essere utilizzata per specificare attributi del thread, come la dimensione dello stack o la priorità di scheduling.
- `start_routine`: puntatore alla funzione che il nuovo thread eseguirà quando verrà creato.
- `arg`: argomento che verrà passato alla funzione start_routine.

La funzione `pthread_create()` restituisce un intero che indica se la creazione del thread è stata completata con successo.

2.2 `pthread_join()`

La funzione `pthread_join()` viene utilizzata per attendere il completamento di un thread.

La funzione `pthread_join()` prende due argomenti:
- `thread`: un puntatore alla struttura che identifica il thread da attendere.
- `value_ptr`: un puntatore a una variabile in cui verrà memorizzato il valore di ritorno del thread.

La funzione `pthread_join()` blocca il thread chiamante fino a quando il thread specificato non è terminato.

Funzionamento:
- Se il thread specificato non è ancora terminato, attenderà fino al suo termine.
- Se il thread specificato è già terminato, restituirà immediatamente.
- Se si passa NULL come valore di `value_ptr`, non memorizzerà il valore di ritorno del thread.

2.3 Locks o mutex

I lock sono un meccanismo di mutua esclusione che consente a un solo thread alla volta di accedere a una sezione critica.

La libreria POSIX fornisce due routine di base per l'utilizzo dei lock:

```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

La routine `pthread_mutex_lock()` acquisisce il lock e la routine `pthread_mutex_unlock()` lo rilascia.

```c
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1;
pthread_mutex_unlock(&lock);
```

- **Problema 1:** Il lock non è stato inizializzato correttamente. I lock devono essere inizializzati prima di essere utilizzati.
- **Problema 2:** Il codice non controlla gli errori quando chiama le routine di lock. Le routine di lock possono fallire, quindi è importante controllare gli errori per assicurarsi che il lock sia stato acquisito correttamente.

Per inizializzare il lock ci sono 2 modi:
- Modo 1:

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- Modo 2:

```c
int rc = pthread_mutex_init(&lock, NULL);
assert(rc === 0); // always check success!
```

Per risolvere il problema 2, è necessario controllare gli errori quando si chiamano le routine di lock. Questo può essere fatto

utilizzando l'istruzione `assert()` o controllando il valore restituito dalla routine.

Esistono due altre routine di lock:
- `pthread_mutex_trylock()`: tenta di acquisire il lock e restituisce un errore se il lock è già acquisito da un altro thread.
- `pthread_mutex_timedlock()`: tenta di acquisire il lock entro un timeout specificato e restituisce un errore se il lock non è stato acquisito entro il timeout.

2.4 Condition variables

Condition variables: meccanismo di sincronizzazione tra thread che consentono a un thread di attendere che si verifichi una condizione specifica. Utilizzate quando un thread deve aspettare che un altro thread modifichi un dato valore o stato. Implementate utilizzando un lock necessario per garantire che la condizione non venga modificata da un altro thread mentre il primo thread è in attesa.

```c
pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
pthread_cond_signal(pthread_cond_t *cond);
```

- `pthread_cond_wait()`, mette il thread chiamante in attesa. Il thread rimane in attesa fino a quando un altro thread chiama la routine `pthread_cond_signal()` per indicare che la condizione è stata modificata.
- `pthread_cond_signal()`, sveglia uno o più thread che sono in attesa sulla condizione specificata.

È importante utilizzare le condition variables in modo appropriato per evitare problemi di sincronizzazione. Ad esempio, è importante assicurarsi che il lock sia acquisito prima di chiamare la routine `pthread_cond_wait()` e rilasciato dopo aver chiamato la routine `pthread_cond_signal()`.

3 Locks

```c
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Lock o mutex: primitiva di sincronizzazione, un meccanismo che impone dei limiti all'accesso a una risorsa quando ci sono molti thread di esecuzione. Utilizzato per garantire la mutua esclusione. Nella pratica è una variabile che rappresenta lo stato del lock (locked/unlocked).

3.1 Funzionamento dei lock

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
Pthread_mutex_lock(&lock); // wrapper; exits on failure
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

I lock vanno inizializzati perché possono assumere solamente due valori:
- Disponibile
- Acquisito

Funzionamento:
1. Viene chiamata la routine `lock()` per acquisire il lock.
2. Se disponibile, il thread chiamante riceve il lock e può entrare in sezione critica. Se acquisito, il thread chiamante rimarrà "bloccato" nella routine `lock()` fino a quando il thread in sezione critica non termina e invoca la `unlock()`.
3. Una volta acquisito il lock un thread può operare in sezione critica, una volta terminato lo rilascia tramite `unlock()`.

`lock()`: routine chiamata dal thread che tenta di acquisire il lock in modo tale da permettere al thread (owner of the lock) di accedere alla sezione critica. Viene acquisito il lock se nessun altro thread lo detiene. Non è possibile acquisire il lock quando non è libero.

`unlock()`: routine chiamata dal proprietario del lock che libera il lock.

Obbiettivi dei lock:
- **Mutua esclusione**: impedimento ai thread di entrare nelle sezioni critiche.
- **Equità**: i thread che si contendono i lock hanno la stessa possibilità di acquisirlo una volta che il lock è libero.
- **Performance**: costo aggiuntivo in termini di tempo.

Primitive di sincronizzazione
- Disabilitare gli interrupt
- Load/Stores
- TestAndSet
- Load-linked and store conditional
- Fetch-And-AddFetch

3.1.1 Disabilitare gli interrupt

```c
void lock() {
  DisableInterrupts();
}

void unlock() {
  EnableInterrupts();
}
```

Codice 9: Lock che disabilitano l'interrupt.

Disattivando gli interrupt prima di entrare in una sezione critica, ci assicuriamo che il codice all'interno della sezione non verrà interrotto e quindi verrà eseguito come se fosse atomico. Si ottiene l'esclusione reciproca (mutual execution).

Vantaggi: semplicità della politica.

Svantaggi:
- Richiede un'operazione privilegiata (ON/OFF interrupts): ci potrebbe essere un abuso tramite la routine `lock()`.

- Non funziona su sistemi a multi processore: se molti thread sono in esecuzione su più CPU, non importa se gli interrupt sono disabilitati, i thread potrebbero comunque accedere alla sezione critica tramite altre CPU.
- Gli interrupt potrebbero essere disabilitati per tanto tempo.
- L'approccio può essere inefficiente: il codice che attiva e disabilita gli interrupt è tendenzialmente più lento.

3.1.2 Loads/Stores

```c
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
  // 0 -> lock is available, 1 -> held
  mutex->flag = 0;
}

void lock(lock_t *mutex) {
  while (mutex->flag === 1) // TEST the flag
  ; // spin-wait (do nothing)
  mutex->flag = 1; // now SET it!
}

void unlock(lock_t *mutex) {
  mutex->flag = 0;
}
```

Codice 10: Lock con load/store.

Per costruire un lock efficace, dobbiamo sfruttare le istruzioni hardware della CPU. Un primo approccio consiste nell'usare una variabile flag per indicare se un thread possiede il lock.

Funzionamento del lock semplice:
1. Un thread chiama `lock()`, verifica che `flag === 0` e imposta `flag = 1`, ottenendo il lock.
2. Dopo aver terminato, chiama `unlock()` e ripristina `flag = 0`.
3. Se un altro thread chiama `lock()` mentre il primo è nella sezione critica, rimane bloccato in un ciclo di attesa (`spin-waiting`) finché il flag non viene liberato.

Problemi:
1. **Errore di correttezza (mancanza di mutua esclusione)**: Se due thread eseguono `lock()` quasi simultaneamente, entrambi possono vedere `flag = 0` e impostarlo a `1`, accedendo contemporaneamente alla sezione critica. Questo viola la mutua esclusione.
2. **Problema di prestazioni**: L'uso dello spin-waiting fa sì che un thread consumi inutilmente tempo CPU controllando continuamente il flag. Questo è particolarmente inefficiente su sistemi monoprocessore, dove il thread bloccante non può avanzare fino al prossimo context switch.

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Figura 42: Trace: No Mutual Exclusion.

3.1.3 TestAndSet

Sistemi multiprocessore moderni, anche quelli con un singolo processore, supportano primitiva hardware per il locking, come il Test-and-Set.

```c
typedef struct __lock_t {
  int flag;
} lock_t;

void init(lock_t *lock) {
  // 0: lock is available, 1: lock is held
  lock->flag = 0;
}

void lock(lock_t *lock) {
  while (TestAndSet(&lock->flag, 1) === 1)
    // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
  lock->flag = 0;
}
```

```c
int TestAndSet(int *old_ptr, int new) {
  int old = *old_ptr; // fetch old value at old_ptr
  *old_ptr = new; // store 'new' into old_ptr
  return old; // return the old value
}
```

Codice 11: Lock con `TestAndSet()`.

Utilizziamo un flag per indicare se un thread possiede un lock.

L'operazione `TestAndSet()` esegue due operazioni atomiche in un'unica istruzione:
- Legge il valore attuale di una variabile condivisa (solitamente un flag di lock).
- Aggiorna la variabile con un nuovo valore.
- Restituisce il valore precedente della variabile.

Funzionamento:
1. Il thread chiamerà `lock()`;
2. Verifica se il flag vale "1" (se ha un lock).
   - Se il flag vale "0" il thread può holdare il lock. Al termine della sezione critica, il thread chiama `unlock()`.
   - Se il flag vale "1" il thread già holda un lock.

Se un thread B chiama `lock()` mentre thread A è nella sezione critica, thread B aspetterà (**spin-wait**) in un ciclo while finché thread A non avrà fatto `unlock()`. Dopo l'unlock di thread A, thread B esce dal ciclo while e imposta il suo flag a "1" ed entra nella sezione critica.

Spinlock: meccanismo che permette a un thread di attendere attivamente finché una risorsa condivisa non diventa disponibile.

Spinlock è più adatto con uno scheduler preemptive, in quanto un thread in attesa può essere interrotto periodicamente per dare opportunità ad altri thread di eseguire. Senza la prelazione, gli spinlock potrebbero essere inefficaci su un singolo processore, poiché un thread in attesa non rinuncerebbe mai al controllo del processore.

Vantaggi:
- Facili da implementare con supporto hardware.
- Garantisce esclusione mutua con operazioni atomiche.
- Utile su sistemi multiprocessore.

Svantaggi:
- Consumo di CPU: Se il lock è occupato, il thread continua a cicli di attesa (inefficiente su singola CPU senza preemption).
- Busy-Waiting: Blocca risorse inutilmente se il lock rimane occupato a lungo.

3.1.3.0.1 Valutazione di uno spinlock

- **Correttezza**: lo spin lock effettua l'esclusione reciproca.
- **Equità**: non forniscono garanzia di equità. Non garantisce che un thread in attesa entri mai nella sezione critica, potenzialmente portando a situazioni di stallo e fame dei thread.
- **Performance**: su sistemi a singola CPU i costi sono molto alti. Per sistemi multi processore i costi sono sostenibili finché il numero dei thread non supera quello delle CPU.

3.1.4 Compare-and-swap

Primitiva hardware alternativa al Test-And-Compare. Viene confrontato il valore del puntatore (`*ptr`) con quello aspettato (expected), se questi risultano uguali il valore del puntatore viene aggiornato con uno nuovo.

Compare-and-Swap è una primitiva più potente rispetto a test-and-set poiché consente di confrontare il valore e aggiornarlo solo se corrisponde a un valore atteso. Entrambi i metodi usano un ciclo di spin, ma CAS offre maggiore flessibilità per altre applicazioni come la sincronizzazione lock-free.

```c
void lock(lock_t *lock) {
  while (CompareAndSwap(&lock->flag, 0, 1) === 1)
  ; // spin
}
```

```c
int CompareAndSwap(int *ptr, int expected, int new) {
  int original = *ptr;
  if (original === expected)
    *ptr = new;
  return original;
}
```

Codice 12: Lock con `CompareAndSwap()`.

3.1.5 Load-linked and store conditional

load-linked« e store-conditional lavorano insieme per costruire sezioni critiche.

La Load-Linked legge un valore dalla memoria e lo immagazzina in un registro, mentre Store-Conditional aggiorna il valore solo se nessuna modifica è avvenuta nel frattempo all'indirizzo di memoria.

- **load-linked**: preleva un valore dalla memoria e lo inserisce in un registro.
- **store-conditional**: controlla che il load abbia aggiornato il puntatore, se lo ha fatto aggiorna il puntatore e restituisce "1" altrimenti ritorna "0".

Funzionamento:
1. Load-Linked: Carica un valore dalla memoria in un registro.
2. Store-Conditional:
   - Aggiorna il valore solo se non è stato modificato da un'altra operazione tra il Load-Linked e il Store-Conditional.
   - Se l'aggiornamento ha successo, restituisce 1; altrimenti, restituisce 0.

```c
void lock(lock_t *lock) {
  while (1) {
    while (LoadLinked(&lock->flag) === 1)
      // spin until it's zero
    if (StoreConditional(&lock->flag, 1) === 1)
      return; // if set-to-1 was success: done
              // otherwise: try again
    }
}

void unlock(lock_t *lock) {
  lock->flag = 0;
}
```

```c
int LoadLinked(int *ptr) {
  return *ptr;
}

int StoreConditional(int *ptr, int value) {
  if (no update to *ptr since LL to this addr) {
    *ptr = value;
    return 1; // success!
  } else {
      return 0; // failed to update
  }
}
```

Codice 13: Lock con `LoadLinked()` e `StoreConditional()`

Nel codice di `lock()`, un thread attende che il flag sia impostato a 0, indicando che il blocco non è detenuto. Poi, tenta di acquisire il blocco usando la `StoreConditional()`. Se questa operazione ha successo, il thread ha cambiato atomicamente il valore del flag a 1 e può procedere nella sezione critica.

`StoreConditional()` può fallire se un altro thread ha modificato il valore del flag tra la `LoadLinked()` e la `StoreConditional()`. In caso di fallimento, il thread deve ritentare l'acquisizione del blocco.

```c
void lock(lock_t *lock) {
  while (LoadLinked(&lock->flag) || !StoreConditional(&lock->flag, 1))
    ; // spin
}
```

3.1.6 FetchAndAdd

La primitiva fetch-and-add è un'operazione atomica che incrementa un valore in memoria e restituisce il valore precedente. Questa primitiva è utilizzata per creare un ticket block. Per creare un lock si utilizzano in modo combinato un `ticket` e una variabile `turn`. Quando un thread ha `myturn === turn`, può entrare nella sezione critica.

```c
int FetchAndAdd(int *ptr) {
  int old = *ptr;
  *ptr = old + 1;
  return old;
}
```

Codice 14: Lock con `FetchAndAdd()` (ticket lock)

```c
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

Quando un thread desidera acquisire il lock, esegue un'operazione `FetchAndAdd()` atomica sul valore del `ticket`; questo valore diventa `turn` di quel thread `myturn`. La variabile `lock->turn` viene utilizzata per determinare quale thread ha il proprio turno. Quando `(myturn === turn)` per un determinato thread, è il turno di quel thread di entrare nella sezione critica. Il rilascio del lock viene eseguito incrementando la variabile `turn`, consentendo al thread successivo in attesa (se presente) di entrare nella sezione critica.

Questa implementazione assicura il progresso per tutti i thread. Una volta assegnato un valore di `ticket` a un thread, sarà pianificato in qualche momento futuro.

Funzionamento: Quando un thread desidera acquisire un lock, esegue atomicamente Fetch-And-Add sul valore del ticket (turn). lock → turn determina quale thread ha il turno. Quando il turno del thread è uguale al turno. Quando il turn del thread e il turn globale coincidono, il thread entra nella sezione critica. L'`unlock()` viene fatto all'incremento del turno. Garantisce l'equità perché al momento che il thread riceve il valore del ticket riceverà un ordine di scheduling.

Vantaggi del Ticket Lock
- Il ticket lock offre un'importante progresso garantito rispetto ad altri lock come il test-and-set. Una volta che un thread riceve il suo biglietto, entrerà nella sezione critica dopo che tutti i thread davanti a lui avranno completato il loro turno.

- Garantisce fairness: ogni thread otterrà il proprio turno in ordine, senza rischiare di essere bloccato indefinitamente.

A differenza di test-and-set, che non garantisce alcun progresso (un thread può rimanere in attesa indefinitamente), il ticket lock garantisce che ogni thread acquisisca il lock in ordine, migliorando la fairness. Il ticket lock è particolarmente utile in ambienti multi-threading ad alta concorrenza, dove si vuole evitare la starvation.

3.2 Problema dei lock con il context switch

```c
void init() {
  flag = 0;
}

void lock() {
  while (TestAndSet(&flag, 1) === 1)
    yield(); // give up the CPU
}

void unlock() {
  flag = 0;
}
```

Quando un thread è interrotto durante una sezione critica (ad esempio, a causa di un context switch), gli altri thread che stanno aspettando il lock potrebbero entrare in un ciclo di spin infinito, in attesa che il thread interrotto riprenda l'esecuzione.

Soluzione iniziale proposta: invece di far girare indefinitamente, il thread cede la CPU ad un altro thread. Questo è ciò che viene descritto con il concetto di yield.

Si utilizza un'operazione Test-and-Set per verificare se il lock è disponibile e acquisirlo se non lo è. Se il lock è già acquisito, il thread non spinna (non consuma cicli della CPU inutilmente), ma esegue `yield()`, che cede la CPU ad un altro thread.

Se ci sono solo due thread e un thread ottiene il lock, il secondo thread chiama `lock()`, rileva che il lock è occupato, e cede la CPU. Questo è un miglioramento rispetto al comportamento di spin, in quanto un thread non consuma cicli di CPU inutili, ma il contesto di switching è più efficiente.

Problema di costo: Se ci sono molti thread che contendono per il lock (es. 100), e uno di questi thread acquisisce il lock ma viene preempted prima di rilasciarlo, gli altri 99 thread entreranno nel ciclo di lock() e chiameranno yield(). Ogni thread in attesa cede la CPU, causando un gran numero di context switch. Un gran numero di switch di contesto può diventare costoso, con sprechi di tempo notevoli.

Problema di starvation: un thread potrebbe finire intrappolato in un ciclo di yield senza mai ottenere l'opportunità di entrare nella sezione critica, mentre altri thread entrano e escono dal blocco. Questo dimostra che l'approccio yield non risolve completamente il problema della fairness e della starvation.

3.3 Lock con le code

Utilizziamo le **code** per tracciare i thread in attesa del lock e l'OS per controllare i successivi thread detentori del lock.

L'approccio di yield può comportare un uso inefficiente della CPU con costosi context switch e non risolve completamente il problema della starvation, in cui alcuni thread potrebbero essere bloccati indefinitamente in attesa del lock.

La soluzione migliore è quella di mettere i thread in attesa (sleep) invece di farli «girare» in attesa del lock. Questo è ciò che viene fatto con l'uso di una coda di attesa (queue) che tiene traccia di quali thread sono in attesa di acquisire il lock. In questo modo, l'OS ha più controllo su quale thread acquisisce il lock successivamente, evitando starvation e migliorando l'efficienza.

- `park()`: per mettere il thread in attesa.
- `unpark()`: per svegliare il thread quando il lock è disponibile. La gestione del lock avviene con una coda che contiene i thread in attesa.

`park()` e `unpark()` sono due routine utilizzate per creare un lock che mette nello stato di stop il chiamante tenta di acquisire un blocco trattenuto e lo riattiva quando il lock è libero.

Non evita del tutto l'attesa di rotazione: potrebbe essere interrotto durante l'acquisizione o il rilascio del lock e fare in modo che altri thread ruotino in attesa che questo venga eseguito nuovamente (tempo trascorso a spinnare limitato).

Quando un thread non può acquisire il lock, il processo chiamante va in coda (chiamando gettid per ottenere l'ID del thread corrente), imposta guard a zero e rilascia la CPU.

`setpark()`: quando un thread chiama questa routine può indicare se sta per fare park(). Se capita di essere interrotto e un altro thread chiama unpark() prima che il park venga effettivamente chiamato, il park successivo ritorna immediatamente invece di dormire.

```c
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) === 1)
        ; //acquire guard lock by spinning

    if (m->flag === 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) === 1)
        ; //acquire guard lock by spinning

    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
    // (for next thread!)
    m->guard = 0;
}
```

Vantaggi di questo Approccio:

- **Evitare lo Spin**: Questo approccio evita l'effetto negativo dello spinning continuo e del context switch costoso. Invece di occupare cicli di CPU inutilmente, i thread dormono e sono risvegliati solo quando necessario.
- **Controllo del Scheduler**: Grazie all'uso delle primitive park e unpark, l'operazione di lock non dipende esclusivamente dalla scelta del sistema operativo su quale thread eseguire. I thread vengono messi in attesa e risvegliati nell'ordine in cui sono arrivati.
- **Evitare la Starvation**: Poiché i thread sono gestiti tramite una coda, l'ordine di acquisizione del lock è determinato in modo equo. Ogni thread attende il suo turno per entrare nella sezione critica, eliminando così il problema di starvation.

Problematiche e soluzioni:
- **Race Condition (Guard e Park)**: Potrebbe esserci un race condition tra il momento in cui un thread verifica che il lock è occupato (dando l'impressione di dover dormire) e il momento in cui un altro thread rilascia il lock. Se il thread che sta per dormire viene interrotto proprio in quel momento, potrebbe rischiare di dormire all'infinito. Per risolvere questo, viene introdotto un meccanismo chiamato setpark() che segnala al sistema operativo che il thread sta per essere messo a dormire. Se il lock viene rilasciato prima che il thread effettivamente si addormenti, questo viene immediatamente svegliato.
- **Guard Lock e Coda**: L'uso del guard lock è necessario per evitare che altri thread possano manipolare flag e la coda mentre un thread sta tentando di acquisirli. Se un thread viene interrotto mentre acquisisce o rilascia il lock, altri thread dovranno aspettare che il guard lock venga rilasciato.
-

3.4 Futex

Futex (fast user space mutex): primitiva di sincronizzazione in Linux. È associato a un'area di memoria fisica specifica e ha una coda in-kernel dedicata. Ogni futex è associata a una locazione specifica nella coda del kernel.

`futex_wait(indirizzo, valore_atteso)`: mette il thread chiamante in attesa, assumendo che il valore all'indirizzo sia uguale a quello atteso. Se non è uguale, la chiamata restituisce immediatamente.

`futex_wake(indirizzo)`: sveglia un thread in attesa nella coda.

Un esempio pratico è dato dal codice in lowlevellock.h nella libreria nptl di glibc. Questo codice utilizza un singolo intero per tracciare lo stato del lock e il numero di attese. L'ottimizzazione è implementata per il caso in cui non ci sia competizione per il lock, riducendo al minimo il lavoro in situazioni di singolo thread.

3.5 Two-Phase Locks

I Two-Phase Locks sono un meccanismo di sincronizzazione ibrido che combina attesa attiva (spinning) e sospensione (sleeping) per migliorare le prestazioni nell'acquisizione di un lock.

Fasi del Two-Phase Lock:
- **1° fase (spin)**: Il thread tenta di acquisire il lock rimanendo in attesa attiva (spinning). Se il lock è rilasciato rapidamente, il thread lo acquisisce senza dover effettuare un costoso cambio di contesto.
- **2°fase (sleep)**: se il thread non riesce ad acquisire il lock entro un certo tempo, entra in stato di sospensione (sleep), riducendo il consumo di CPU. Il thread viene riattivato solo quando il lock diventa disponibile.

4 Lock-based concurrent data structure

4.1 Introduzione

Le strutture dati concorrenti utilizzano lock per garantire sicurezza nei thread e prevenire race conditions. Tuttavia, il modo in cui vengono applicati i lock influisce sia sulla correttezza che sulle prestazioni della struttura. L'obiettivo è trovare un equilibrio tra concorrenza e performance.

4.1.1 Problemi della concorrenza

- **Race Condition**: quando due o più thread accedono a una risorsa condivisa in modo non sincronizzato.
- **Deadlock**: situazione in cui due o più thread rimangono bloccati aspettando l'uno il lock dell'altro.
- **Starvation**: un thread non riesce mai ad ottenere il lock perché altri thread monopolizzano la risorsa.
- **Overhead dei lock**: un numero eccessivo di lock può rallentare l'esecuzione anziché migliorarla.

4.2 Contatori concorrenti

Un contatore è una struttura semplice, ma quando è condiviso tra più thread può causare problemi di concorrenza.

4.2.1 Contatore Senza Lock (non sicuro)

```c
typedef struct __counter_t {
    int value;
} counter_t;

void increment(counter_t c) {
    c->value++; // NON ATOMICO
}
```

Problema: Più thread possono modificare `value` contemporaneamente, generando risultati inaffidabili.

4.2.2 Contatore con Mutex (non scalabile)

```c
typedef struct __counter_t {
    int value;
    pthread_mutex_t lock;
} counter_t;

void increment(counter_t c) {
    pthread_mutex_lock(&c->lock);
    c->value++;
    pthread_mutex_unlock(&c->lock);
}
```

Vantaggio: Sicuro.

Svantaggio: Non scalabile su più CPU, poiché i thread devono aspettare il rilascio del lock.

4.2.3 Contatore Approssimato (scalabile)

- Divide il contatore in contatori locali per CPU.
- Ogni CPU aggiorna il proprio contatore locale, riducendo la contesa sul lock globale.
- Periodicamente, i valori locali vengono sommati in un contatore globale.

```c
typedef struct __counter_t {
    int global;
    pthread_mutex_t glock;
    int local[NUMCPUS];
    pthread_mutex_t llock[NUMCPUS];
    int threshold;
} counter_t;
```

Vantaggio: Scalabilità maggiore rispetto all'approccio con singolo lock.

Svantaggio: Il valore letto dal contatore potrebbe non essere sempre preciso.

4.3 Liste Concorrenti

Le liste concatenate devono gestire correttamente l'inserimento e la ricerca in modo sicuro.

4.3.1 Lista Concorrente con singolo Lock

```c
typedef struct __list_t {
    node_t head;
    pthread_mutex_t lock;
} list_t;

int List_Insert(list_t L, int key) {
    pthread_mutex_lock(&L->lock);
    node_t new = malloc(sizeof(node_t));
    new->key = key;
    new->next = L->head;
    L->head = new;
    pthread_mutex_unlock(&L->lock);
    return 0;
}
```

Vantaggio: Implementazione semplice.

Svantaggio: Scarsa scalabilità, poiché un solo thread può accedere alla lista alla volta.

4.3.2 Hand-over-Hand Locking

- Ogni nodo ha un lock separato.
- Durante l'attraversamento della lista, si acquisisce il lock del nodo successivo prima di rilasciare quello attuale.

Vantaggio: Maggiore concorrenza. Svantaggio: Maggiore overhead per il numero elevato di lock/unlock.

4.4 Code Concorrenti

Le code sono strutture fondamentali nei sistemi concorrenti, utilizzate per buffer, scheduler, ecc.

4.4.1 Coda Concorrente con Doppio Lock

Anche chiamato (Michael-Scott Queue).
- Due lock separati: uno per la testa e uno per la coda.
- Permette di parallelizzare enqueue e dequeue.

```c
typedef struct __queue_t {
    node_t head;
    node_t tail;
    pthread_mutex_t head_lock, tail_lock;
} queue_t;
```

Vantaggio: Maggiore concorrenza rispetto alla coda con singolo lock.

Svantaggio: Più complessa da implementare.

4.5 Hash Table Concorrente

Una tabella hash concorrente consente accesso rapido agli elementi mantenendo la sicurezza nei thread.

4.5.1 Implementazione con Lock per Bucket

- Ogni bucket ha una sua lista concatenata con un lock separato.
- Questo permette accessi concorrenti a diversi bucket, migliorando le prestazioni.

```c
typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;
```

Vantaggio: Ottima scalabilità rispetto alla lista concatenata con singolo lock.

4.6 Two-Phase Locks

Strategia ibrida che combina:
1. Spin-wait breve se il lock sarà presto disponibile.
2. Passaggio in stato di sleep se il lock non viene acquisito rapidamente.

Utilizzato in Linux (futex) per evitare il consumo eccessivo di CPU nei lock lunghi.

4.7 Compare-and-Swap (CAS) vs. Test-and-Set

4.7.1 Test-and-Set

```c
int TestAndSet(int ptr, int new) {
    int old = ptr;
    ptr = new;
    return old;
}
```

Problema: Può causare attesa attiva prolungata (spin lock inefficiente).

4.7.2 Compare-and-Swap (CAS)

- Più potente di Test-and-Set.
- Cambia il valore solo se è ancora quello atteso, evitando race conditions.

```c
bool CompareAndSwap(int ptr, int expected, int new_value) {
    if (ptr == expected) {
        ptr = new_value;
        return true;
    }
    return false;
}
```

Vantaggio: Usato per lock-free synchronization.

5 Condition variables

Usare una variabile condivisa per un thread che controlla una condizione prima di continuare la sua esecuzione (es. wait(): padre che attende il figlio) è inefficiente perché la CPU per controllare la variabile condivisa perderebbe tempo. vorremmo è mettere il genitore in sleep fino a quando la condizione che stiamo aspettando (cioè che il figlio termini la sua esecuzione) diventi vera.

5.1 Condition variable

Condition variable: meccanismo di sincronizzazione che consente a un thread di attendere che si verifichi una condizione specifica. È una coda esplicita in cui i thread possono mettersi in attesa quando una condizione non è soddisfatta; un altro thread, quando cambia la condizione, può quindi svegliare uno (o più) di quei thread in attesa e quindi consentire loro di continuare (segnalando sulla condizione).

```c
pthread_cond_t c;
```

Codice 15: Come creare una condition variable.

Una condition variable ha due operazioni associate:

- `pthread_cond_wait()`: mette il thread corrente in attesa di una condizione specificata. La condizione è rappresentata da un oggetto di tipo `thread_cond_t`. Il thread corrente viene risvegliato quando la condizione è soddisfatta o quando viene interrotto da un segnale.
- `pthread_cond_signal()`: sveglia uno o più thread che sono in attesa della condizione specificata.

```c
int pthread_cond_wait(
    pthread_cond_t cond, pthread_mutex_t mutex
);
int pthread_cond_signal(pthread_cond_t cond);
```

5.2 Problema del buffer limitato

Anche chiamato «The Producer/Consumer Problem». Problema di sincronizzazione comune in cui uno o più thread produttori generano dati e li inseriscono in un buffer, mentre uno o più thread consumatori prelevano i dati dal buffer e li consumano. Il buffer è una risorsa condivisa, quindi è necessario sincronizzare l'accesso ad essa per evitare race condition.

```c
int buffer;
int count = 0; // initially, empty

void put(int value) {
    assert(count === 0);
    count = 1;
    buffer = value(
)

int get() {
    assert(count === 1);
    count = 0;
    return buffer(
)
```

Codice 16: The Put And Get Routines (v1).

Occorrono routine che sappiano quando è possibile accedere al buffer per inserire/estrarre dati. Così:

```c
void producer(void arg) {
    int i;
    int loops = (int) arg;
    for (i = 0; i < loops; i++) {
        put(i);
    }
}

void consumer(void arg) {
    while (1) {
        int tmp = get();
        printf("%d\n", tmp);
    }
}
```

Codice 17: Producer/Consumer Threads (v1).

5.2.1 Soluzione v1 (broken, if statement)

```c
int loops; // must initialize somewhere...
cond_t cond;
mutex_t mutex;

void producer(void arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex); // p1
        if (count === 1) // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i); // p4
        Pthread_cond_signal(&cond); // p5
        Pthread_mutex_unlock(&mutex); // p6
    }
}

void consumer(void arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex); // c1
        if (count === 0) // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get(); // c4
        Pthread_cond_signal(&cond); // c5
        Pthread_mutex_unlock(&mutex); // c6
        printf("%d\n", tmp);
    }
}
```

Codice 18: Producer/Consumer: Single CV And If Statement.

Problema: quando ci sono più consumatori è possibile che uno consumi il buffer prima dell'altro, quando il consumatore tenterà di consumare un buffer vuoto causerà un errore.

T_{c1} viene eseguito:
1. Acquisisce il lock (c1)
2. Controlla se ci sono buffer pronti per il consumo (c2) ma non ce ne sono.
3. Attende (c3) (rilasciando il blocco).

T_p viene eseguito:
1. Acquisisce il lock (p1)
2. Controlla se tutti i buffer sono pieni (p2) ma i buffer sono vuoti
3. Riempie il buffer (p4).
4. Segnala che un buffer è stato riempito (p5).

T_{c1} ora è pronto per l'esecuzione fino a quando non si rende conto che il buffer è pieno. Rilascia il lock e si mette a dormire (p6, p1-p3).

Problema: T_{c2} si intrufola e consuma il buffer (c1, c2, c4, c5, c6, saltando l'attesa in c3 perché il buffer è pieno).

T_{c1} va in esecuzione:
1. Poco prima di tornare dall'attesa, riacquisisce il blocco e poi torna.
2. Chiama `get()` (c4) e non ci sono buffer da consumare!

3. Si attiva un'asserzione e il codice non ha funzionato come desiderato.

Avremmo dovuto impedire a T_{c1} di provare a consumare perché T_{c2} si è intrufolato consumando il buffer che era stato prodotto.

Il problema si verifica perché il consumatore T_{c1} non è stato in grado di verificare che il buffer fosse effettivamente pronto prima di chiamare `get()`.

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Run	0	
	Sleep		Ready	p2	Run	0	
	Sleep		Ready	p4	Run	1	Buffer now full
	Ready		Ready	p5	Run	1	T_{c1} awoken
	Ready		Ready	p6	Run	1	
	Ready		Ready	p1	Run	1	
	Ready		Ready	p2	Run	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Run		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Run		Sleep	1	
	Ready	c4	Run		Sleep	0	... and grabs data
	Ready	c5	Run		Ready	0	T_p awoken
	Ready	c6	Run		Ready	0	
c4	Run		Ready		Ready	0	Oh oh! No data

Figura 43: Trace dei thread con l'if.

Semantica Mesa: un segnale a un thread è solo un suggerimento che lo stato del mondo è cambiato. Il thread svegliato non ha alcuna garanzia che lo stato del mondo sarà ancora come lo desiderava.

Semantica di Hoare: un segnale a un thread garantisce che il thread verrà eseguito immediatamente dopo essere stato svegliato. Questo significa che il thread svegliato avrà la possibilità di verificare che lo stato del mondo sia ancora come lo desiderava.

5.2.2 Soluzione v2 (broken, while statement)

```c
int loops;
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex); // p1
        while (count === 1) // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i); // p4
        Pthread_cond_signal(&cond); // p5
        Pthread_mutex_unlock(&mutex); // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex); // c1
        while (count === 0) // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get(); // c4
        Pthread_cond_signal(&cond); // c5
        Pthread_mutex_unlock(&mutex); // c6
        printf("%d\n", tmp);
    }
}
```

Codice 19: Producer/Consumer: Single CV And While.

Cambiando l'`if` con il `while`:

Il thread T_{c1}:
1. Si sveglia e (con il lock acquisito).
2. Controlla lo stato della cv (c2).
3. Se il buffer è vuoto, il consumatore torna semplicemente a dormire (c3).

L'`if` e cambiato in `while` anche nel produttore (p2).

Problema 2: tutti e tre i thread rimangono inattivi.

Il problema si verifica quando due consumatori vengono eseguiti per primi (T_{c1} e T_{c2}) e si addormentano entrambi (c3).

Il thread T_p:
1. Inserisce un valore nel buffer e sveglia uno dei consumatori (diciamo T_{c1}).
2. Torna indietro (rilasciando e riacquistando il blocco lungo il percorso) e tenta di inserire più dati nel buffer e vede il buffer pieno, il produttore invece attende sulla condizione (così dormendo).

il thread T_{c1}:
1. Due thread sono in attesa su una condizione (T_{c2} e T_p).
2. Si sveglia quindi tornando da wait() (c3)
3. Ricontrolla la condizione (c2) e il buffer pieno
4. Consuma il valore (c4).
5. Invia un segnale sulla condizione (c5), svegliando solo un thread che è in attesa.

Quale thread dovrebbe svegliare?

Poiché il consumatore ha svuotato il buffer, dovrebbe chiaramente svegliare il produttore. Ma se sveglia il consumatore T_{c2} (che è sicuramente possibile, a seconda di come viene gestita la coda di attesa), abbiamo un problema:

- T_{c2} si sveglierà e troverà il buffer vuoto (c2) e tornerà a dormire (c3).
- T_p, che ha un valore da inserire nel buffer, rimane addormentato.
- T_{c1}, torna anche a dormire.

Tutti e tre i thread rimangono inattivi.

È evidente che è necessario inviare un segnale, ma deve essere più diretto. Un consumatore non dovrebbe svegliare altri consumatori, solo produttori, e viceversa.

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Run		Ready		Ready	0	
c2	Run		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Run		Ready	0	
	Sleep	c2	Run		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Run	0	
	Sleep		Sleep	p2	Run	0	
	Sleep		Sleep	p4	Run	1	Buffer now full
	Ready		Sleep	p5	Run	1	T_{c1} awoken
	Ready		Sleep	p6	Run	1	
	Ready		Sleep	p1	Run	1	
	Ready		Sleep	p2	Run	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Run		Sleep		Sleep	1	Recheck condition
c4	Run		Sleep		Sleep	0	T_{c1} grabs data
c5	Run		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Run		Ready		Sleep	0	
c1	Run		Ready		Sleep	0	
c2	Run		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Run		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

Figura 44: Trace dei thread senza il while.

5.2.3 Soluzione a singolo buffer

```c
cond_t empty, fill;
mutex_t mutex;

void producer(void arg) {
   int i;
   for (i = 0; i < loops; i++) {
      Pthread_mutex_lock(&mutex);
      while (count === 1)
         Pthread_cond_wait(&empty, &mutex);
      put(i);
      Pthread_cond_signal(&fill);
      Pthread_mutex_unlock(&mutex);
      }
   }

void consumer(void arg) {
   int i;
   for (i = 0; i < loops; i++) {
      Pthread_mutex_lock(&mutex);
      while (count === 0)
         Pthread_cond_wait(&fill, &mutex);
      int tmp = get();
      Pthread_cond_signal(&empty);
      Pthread_mutex_unlock(&mutex);
      printf("%d\n", tmp);
      }
   }
```

Codice 20: Producer/Consumer: Two CVs And While.

Il problema del produttore/consumatore con buffer singolo può essere risolto utilizzando due variabili di condizione.

I produttori aspettano sulla condizione empty e segnalano fill. I consumatori aspettano su fill e segnalano empty. In questo modo un consumatore non può mai svegliare accidentalmente un altro consumatore, e un produttore non può mai svegliare accidentalmente un altro produttore. La soluzione è corretta ma no in generale.

5.2.4 Soluzione corretta

Modifica che porta maggiore concorrenza e efficienza.

Aggiungiamo più slot del buffer, in modo che più valori possano essere prodotti prima di dormire, e allo stesso modo più valori possono essere consumati prima di dormire.

Con un solo produttore e consumatore, questo approccio è più efficiente in quanto riduce gli switch di contesto; con più produttori o consumatori (o entrambi), consente anche la produzione o il consumo concorrente, aumentando così la concorrenza.

```c
int buffer[MAX];
int fill_ptr = 0;
int use_ptr = 0;
int count = 0;
void put(int value) {
buffer[fill_ptr] = value;
fill_ptr = (fill_ptr + 1) % MAX;
count++;
}
int get() {
int tmp = buffer[use_ptr];
use_ptr = (use_ptr + 1) % MAX;
count--;
return tmp;
}
```

Codice 21: The Correct Put And Get Routines.

```c
cond_t empty, fill;
mutex_t mutex;

void * producer(void * arg) {
   int i;
   for (i = 0; i < loops; i++) {
      Pthread_mutex_lock( & mutex); // p1
      while (count === MAX) // p2
         Pthread_cond_wait( & empty, & mutex); // p3
      put(i); // p4
      Pthread_cond_signal( & fill); // p5
      Pthread_mutex_unlock( & mutex); // p6
   }
}

void * consumer(void * arg) {
   int i;
   for (i = 0; i < loops; i++) {
      Pthread_mutex_lock( & mutex); // c1
      while (count === 0) // c2
         Pthread_cond_wait( & fill, & mutex); // c3
      int tmp = get(); // c4
      Pthread_cond_signal( & empty); // c5
      Pthread_mutex_unlock( & mutex); // c6
      printf("%d\n", tmp);
   }
}
```

Codice 22: The Correct Put And Get Routines.

5.3 Covering Conditions

Un produttore dorme solo se tutti i buffer sono attualmente pieni (p2). Un consumatore dorme solo se tutti i buffer sono attualmente vuoti (c2).

```c
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t c;
mutex_t m;

void allocate(int size) {
      Pthread_mutex_lock( & m);
      while (bytesLeft < size)
         Pthread_cond_wait( & c, & m);
      void  ptr = ...; // get mem from heap
      bytesLeft -= size;
      Pthread_mutex_unlock( & m);
      return (tr;
)

void free(void  ptr, int size) {
   Pthread_mutex_lock( & m);
   bytesLeft += size;
   Pthread_cond_signal( & c); // whom to signal??
   Pthread_mutex_unlock( &(m);
)
```

Codice 23: Covering Conditions example.

Considera zero byte liberi.

1. T_a chiama `allocate(100)`,
2. T_b che richiede meno memoria chiamando `allocate(10)`.
3. Entrambi T_a e T_b quindi aspettano sulla condizione e si addormentano; non ci sono abbastanza byte liberi per soddisfare nessuno di questi requisiti.
4. T_c, chiama free(50)
5. T_c chiama signal per svegliare un thread in attesa e potrebbe non svegliare il thread corretto
6. T_b, che sta aspettando solo 10 byte da liberare
7. T_a dovrebbe rimanere in attesa, poiché non è ancora disponibile abbastanza memoria.

Il codice non funziona, il thread che sveglia altri thread non sa quale thread svegliare.

Soluzione suggerita da Lampson e Redell (condizione di copertura):
Sostituire la chiamata `pthread_cond_signal()` con `pthread_cond_broadcast()`, che sveglia tutti i thread in attesa in modo da svegliare tutti thread che dovrebbero essere svegli. Può impattare sulle prestazioni ma i thread si svegliati inutilmente ricontrolleranno la condizione e poi torneranno a dormire.

6 Semaphores

Semaforo: meccanismo per gestire la concorrenza che può essere usato come una condition variable o come lock. È un oggetto con un valore intero che può essere manipolato con due routine: `sem_wait()` e `sem_post()`. Siccome valore iniziale del semaforo determina il suo comportamento, per usare un semaforo, occorre prima inizializzarlo a un valore.

```c
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // Semafori inizializzato a 1.
```

- **`sem_init()`**: inizializza un semaforo.

  ```c
  int sem_init(sem_t *sem, int pshared, unsigned int value);
  ```

  - `sem`: Puntatore alla variabile del semaforo.
  - `pshared`: Se `0`, il semaforo è usato solo tra thread dello stesso processo. Se diverso da `0`, può essere condiviso tra processi diversi (ma deve risiedere in memoria condivisa).
  - `value`: Valore iniziale del semaforo.

  Ritorna `0` se ha successo, altrimenti `-1` con `errno` settato.

- **`sem_wait()`**: decrementa il semaforo.

  ```c
  int sem_wait(sem_t *sem);
  ```

  - Se il valore del semaforo è maggiore di `0`, lo decrementa e continua.
  - Se è `0`, il thread si blocca finché il semaforo non diventa positivo.

  Ritorna `0` se ha successo, `-1` in caso di errore.

- **`sem_post()`**: incrementa il semaforo.

  ```c
  int sem_post(sem_t *sem);
  ```

  - Aumenta il valore del semaforo di `1`.
  - Se altri thread sono in attesa (`sem_wait()`), ne sblocca uno.

  Ritorna `0` se ha successo, `-1` in caso di errore.

6.1 Semafori binari

È un semaforo che può assumere solo due valori:
- 0 (libero)
- 1 (occupato)

Possono essere utilizzati come lock, è necessario circondare la sezione critica di interesse con una coppia `sem_wait()`/`sem_post()`.

Utilizzo:
- Il thread 0 chiama `sem_wait()` per decrementare il valore del semaforo a 0 (thread 0 ha acquisito il lock).
- Il thread 0 quindi esegue l'operazione critica.
- Quando thread 0 ha terminato l'operazione, chiama `sem_post()` per incrementare il valore del semaforo a 1 (lock rilasciato).

Value of Semaphore	Thread 0	Thread 1
1		
1	call `sem_wait()`	
0	`sem_wait()` returns	
0	`(crit sect)`	
0	call `sem_post()`	
1	`sem_post()` returns	

Figura 45: Singolo thread che utilizza il semaforo.

Un caso più interessante si verifica quando il Thread 0 ha il lock (ovvero ha chiamato `sem_ wait()` ma non ha ancora chiamato `sem_post()`), e un altro thread (Thread 1) tenta di accedere alla sezione critica chiamando `sem_wait()`.

- Thread 1 decrementerà il valore del semaforo a −1 e quindi attenderà (metterà se stesso a dormire e rilascerà il processore).

- Quando Thread 0 viene eseguito di nuovo, alla fine chiamerà `sem_post()`, incrementando il valore del semaforo a zero e quindi risvegliando il thread in attesa (Thread 1), che potrà quindi acquisire il blocco per sé.
- Quando Thread 1 termina, incrementerà nuovamente il valore del semaforo, ripristinandolo a 1.

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call `sem_wait()`	Run		Ready
0	`sem_wait()` returns	Run		Ready
0	`(crit sect begin)`	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call `sem_wait()`	Run
-1		Ready	`decr sem`	Run
-1		Ready	`(sem<0)→sleep`	Sleep
-1		Run	*Switch→T0*	Sleep
-1	`(crit sect end)`	Run		Sleep
-1	call `sem_post()`	Run		Sleep
0	`incr sem`	Run		Sleep
0	`wake(T1)`	Run		Ready
0	`sem_post()` returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	`sem_wait()` returns	Run
0		Ready	`(crit sect)`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`sem_post()` returns	Run

Figura 46: Due thread che usano lo stesso semaforo.

6.2 Semafori per ordinamento

I semafori possono essere utilizzati per ordinare gli eventi in un programma concorrente.

Un thread può desiderare di attendere che una lista diventi non vuota, in modo da poter eliminare un elemento.

In questo modello di utilizzo, si trova spesso un thread che attende che qualcosa accada, e un altro thread che fa accadere quella cosa e poi segnala che è successo, risvegliando così al thread in attesa.

Stiamo utilizzando il semaforo come una primitiva d'ordinamento.

Immagina che un thread crei un altro thread e poi voglia aspettare che completi la sua esecuzione. Vogliamo ottenere:
1. parent: begin
2. child
3. parent: end

6.2.1 Primo caso

Come dovrebbe essere con i semafori:
1. Il thread genitore esegue, decrementa il semaforo (a −1), quindi attende (dormendo).

- Quando il thread figlio finalmente esegue, chiamerà `sem_post()`, incrementerà il valore del semaforo da 0 a 1.
- Quando il genitore ottiene quindi la possibilità di eseguire, chiamerà `sem_wait()` e troverà il valore del semaforo a 1; il genitore decrementerà quindi il valore (a 0) e tornerà da `sem_wait()` senza aspettare, ottenendo anche l'effetto desiderato.

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists, can run)*	Ready
0	call `sem_wait()`	Run		Ready
-1	`decr sem`	Run		Ready
-1	`(sem<0)→sleep`	Sleep		Ready
-1	*Switch→Child*	Sleep	`child runs`	Run
-1		Sleep	call `sem_post()`	Run
0		Sleep	`inc sem`	Run
0		Ready	`wake(Parent)`	Run
0		Ready	`sem_post()` returns	Run
0		Ready	*Interrupt→Parent*	Ready
0	`sem_wait()` returns	Run		Ready

Figura 47: Trace dei thread nel primo caso.

6.2.2 Secondo caso

Si verifica quando il figlio esegue fino al completamento prima che il genitore possa chiamare `sem_wait()`. In questo caso, il figlio chiamerà prima `sem_post()`, aumentando così il valore del semaforo da 0 a 1. Quando il genitore ottiene quindi la possibilità di eseguire, chiamerà `sem_wait()` e troverà il valore del semaforo a 1; il genitore decrementerà quindi il valore (a 0) e tornerà da `sem_wait()` senza aspettare, ottenendo anche l'effetto desiderato.

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists; can run)*	Ready
0	*Interrupt→Child*	Ready	`child runs`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`inc sem`	Run
1		Ready	`wake(nobody)`	Run
1		Ready	`sem_post()` returns	Run
1	`parent runs`	Run	*Interrupt→Parent*	Ready
1	call `sem_wait()`	Run		Ready
0	`decrement sem`	Run		Ready
0	`(sem≥0)→awake`	Run		Ready
0	`sem_wait()` returns	Run		Ready

Figura 48: Trace dei thread nel secondo caso.

6.3 Problema del buffer limitato

Il problema del produttore e consumatore è problema di sincronizzazione negli OS e nella programmazione concorrente. Si verifica quando uno o più produttori generano dati e li inseriscono in un buffer condiviso, mentre uno o più consumatori prelevano questi dati dal buffer per elaborarli.

6.3.1 Primo tentativo

Utilizziamo due semafori, `empty` e `full`, che i thread useranno per indicare quando una voce del buffer è stata svuotata o riempita, rispettivamente.

```c
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
  buffer[fill] = value; // Line F1
  fill = (fill + 1) % MAX; // Line F2
}

int get() {
    int tmp = buffer[use]; // Line G1
    use = (use + 1) % MAX; // Line G2
    return tmp;
}
```

Codice 24: Codice di `put()` e `get()`

```c
sem_t empty;
sem_t full;

void producer(void arg) {
  int i;
  for (i = 0; i < loops; i++) {
      sem_wait(&empty); // Line P1
      put(i); // Line P2
      sem_post(&full); // Line P3
  }
}

void consumer(void arg) {
  int tmp = 0;
  while (tmp != -1) {
    sem_wait(&full); // Line C1
    tmp = get(); // Line C2
    sem_post(&empty); // Line C3
    printf("%d\n", tmp);
  }
}

int main(int argc, char argv[]) {
  // ...
  sem_init(&empty, 0, MAX); // MAX are empty
  sem_init(&full, 0, 0); // 0 are full
  // ...
}
```

Codice 25: Codice del produttore e del consumatore.

Il produttore attende che un buffer diventi vuoto per potervi inserire dati, e il consumatore attende in modo simile che un buffer diventi pieno prima di utilizzarlo.

Supponiamo `MAX = 1` (cioè ci sia solo un buffer nell'array) e vediamo se funziona.

Supponiamo di nuovo che ci siano due thread, un produttore e un consumatore.

Si assume che il consumatore venga eseguito per primo:
1. Esegue C1, chiamando `sem_wait(&full)`. `full` è stato inizializzato al valore 0, la chiamata decrementerà `full` (a −1).
2. Va in sleep e attenderà che un altro thread chiami `sem_post()` su `full`.

Il produttore va in esecuzione:
1. Fa P1 chiamando `sem_wait(&empty)`. A differenza del consumatore, il produttore continuerà attraverso questa linea, perché `empty` è stato inizializzato al valore `MAX` (in questo caso, 1).
2. `empty` verrà decrementato a 0 e il produttore inserirà un valore di dati nella prima voce di buffer (riga P2).
3. Fa P3 e chiamerà `sem_post(&full)`, cambiando il valore del semaforo full da −1 a 0 e risvegliando il consumatore.

Ora possono accadere due cose:
- Se il produttore continua a funzionare, spinnerà di nuovo e colpirà la linea P1 di nuovo. Questa volta si bloccherà, poiché il valore del semaforo `empty` è 0.
- Se invece il produttore veniva interrotto e il consumatore iniziava a funzionare, sarebbe tornato da `sem_wait(&full)` (riga C1), avrebbe trovato che il buffer era pieno e lo avrebbe consumato. In entrambi i casi, otteniamo il comportamento

desiderato.

Lo stesso esempio con più thread:
- Supponiamo ora `MAX` > 1 (diciamo MAX = 10).
- Supponiamo che ci siano più produttori e più consumatori, quindi, abbiamo una race condition.

Problema: due produttori (P_a e P_b) chiamano `put()` nello stesso momento. Si assume che il produttore P_a venga eseguito per primo e stia riempiendo la prima voce del buffer (`fill = 0` alla riga F1). Prima che P_a possa avere la possibilità di incrementare il contatore di riempimento a 1, viene interrotto. P_b inizia a funzionare e alla riga F1 inserisce anche i suoi dati nell'elemento 0 di buffer, il che significa che i dati vecchi vengono sovrascritti!

6.3.2 Aggiungere la mutua esclusione

Per evitare race condition è necessario utilizzare la mutua esclusione.

Problema: è possibile che due processi accedano contemporaneamente al buffer e all'indice del buffer portando a un deadlock.

Deadlock (stallo): una situazione in cui due o più processi sono bloccati e non possono continuare a eseguire.

Esempio di deadlock:
1. Il consumatore viene eseguito prima e acquisisce mutex (c0).
2. Chiama quindi la `sem_wait(&full)` (c1). Tuttavia il consumatore possiede ancora il lock.
3. Viene eseguito un produttore che, se fosse possibile, riempirebbe il buffer di dati e sveglierebbe il consumatore. Sfortunatamente, la prima cosa che fa è chiamare la sem_wait(&mutex) (p0). Il lock, tuttavia, è già in possesso del consumatore e il produttore è bloccato ad aspettare.

Quindi:
- Il consumatore è bloccato su `sem_wait(&full)`, aspettando un elemento che il produttore deve inserire.
- Il produttore è bloccato su `sem_wait(&mutex)`, aspettando che il consumatore rilasci il lock.

```c
void  producer(void  arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait( & mutex); // Line P0 (NEW LINE)
    sem_wait( & empty); // Line P1
    put(i); // Line P2
    sem_post( & full); // Line P3
    sem_post( & mutex); // Line P4 (NEW LINE)
  }
}

void  consumer(void  arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait( & mutex); // Line C0 (NEW LINE)
    sem_wait( & full); // Line C1
    int tmp = get(); // Line C2
    sem_post( & empty); // Line C3
    sem_post( & mutex); // Line C4 (NEW LINE)
    printf("%d\n", tmp);
  }
}
```

Codice 26: Nuovo codice del produttore e del consumatore.

6.3.3 Soluzione definitiva

Occorre ridurre la portata del lock spostando l'acquisizione e il rilascio appena intorno alla sezione critica.

```c
void  producer(void  arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait( & empty); // Line P1
    sem_wait( & mutex); // Line P1.5 (MUTEX HERE)
    put(i); // Line P2
    sem_post( & mutex); // Line P2.5 (AND HERE)
    sem_post( & full); // Line P3
  }
}

void  consumer(void  arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait( & full); // Line C1
    sem_wait( & mutex); // Line C1.5 (MUTEX HERE)
    int tmp = get(); // Line C2
    sem_post( & mutex); // Line C2.5 (AND HERE)
    sem_post( & empty); // Line C3
    printf("%d\n", tmp);
  }
}
```

Codice 27: Nuovo codice del produttore e del consumatore (soluzione al problema).

6.4 Reader-Writer Locks

Altro problema: non esiste una primitiva di locking abbastanza flessibile per soddisfare tutti i requisiti delle operazioni su strutture dati concorrenti. Le operazioni di lettura sono molto più veloci delle operazioni di scrittura e non richiedono di modificare lo stato della struttura dati. Le operazioni di scrittura sono più lente e richiedono di modificare lo stato della struttura dati.

Soluzione: il **lock reader-writer** che consente a più lettori di accedere contemporaneamente a una risorsa, ma solo a un singolo scrittore. Questo è utile per strutture dati in cui le letture sono più comuni delle scritture, come le liste.

Implementazione: se un thread vuole aggiornare la struttura dati in questione, deve chiamare la nuova coppia di operazioni di sincronizzazione:

- `rwlock_acquire_writelock()`: per acquisire un lock di scrittura.
- `rwlock_release_writelock()`: per rilasciare il lock di scrittura.

Internamente utilizzano il semaforo `writelock` per garantire che solo un singolo scrittore possa acquisire il lock.

```c
typedef struct _rwlock_t {
  sem_t lock; // binary semaphore (basic lock)
  sem_t writelock; // allow ONE writer/MANY readers
  int readers; // #readers in critical section
}
rwlock_t;
void rwlock_init(rwlock_t * rw) {
  rw -> readers = 0;
  sem_init( & rw -> lock, 0, 1);
  sem_init( & rw -> writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t * rw) {
  sem_wait( & rw -> lock);
  rw -> readers++;
  if (rw -> readers === 1) // first reader gets writelock
    sem_wait( & rw -> writelock);
  sem_post( & rw -> lock);
}

void rwlock_release_readlock(rwlock_t * rw) {
  sem_wait( & rw -> lock);
  rw -> readers--;
  if (rw -> readers === 0) // last reader lets it go
    sem_post( & rw -> writelock);
  sem_post( & rw -> lock);
}

void rwlock_acquire_writelock(rwlock_t * rw) {
  sem_wait( & rw -> writelock);
}

void rwlock_release_writelock(rwlock_t * rw) {
  sem_post( & rw -> writelock);
}
```

Codice 28: Implementazione di un semplice lock reader-writer.

Quando acquisisce un lock di lettura, il lettore acquisisce prima il lock e poi incrementa la variabile readers per tenere traccia di quanti lettori sono attualmente all'interno della struttura dati. Il passo importante quindi intrapreso all'interno di `rwlock_acquire_readlock()` si verifica quando il primo lettore acquisisce il lock; in tal caso, il lettore acquisisce anche il lock di scrittura chiamando `sem_wait()` sul semaforo `writelock` e quindi rilasciando il lock chiamando `sem_post()`.

Quando il lettore ha acquisito un lock di lettura, altri lettori saranno autorizzati ad acquisire il lock di lettura; tuttavia, qualsiasi thread che desidera acquisire il lock di scrittura dovrà aspettare che tutti i lettori siano terminati; l'ultimo a uscire dalla sezione critica chiama `sem_post()` su `writelock` e quindi abilita un writer in attesa ad acquisire il lock.

Approccio funzionante ma presenta alcuni lettori potrebbero soffrire di starvation.

6.5 Problema dei 5 filosofi

Si supponga che ci siano cinque «filosofi» seduti intorno a un tavolo. Tra ogni coppia di filosofi c'è una sola forchetta (e quindi, cinque in totale). I filosofi hanno periodi in cui pensano, e non hanno bisogno di forchette e periodi in cui mangiano. Per mangiare, un filosofo ha bisogno di due forchette, sia quella alla sua sinistra che quella alla sua destra.

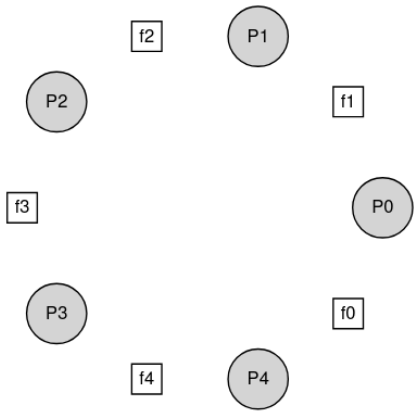


Figura 49: Problema dei 5 filosofi.

La sfida è scrivere le routine `get_forks()` e `put_forks()` in modo che non ci sia deadlock, nessun filosofo muoia di fame e non riesca mai a mangiare, e la concorrenza sia alta (cioè, il maggior numero possibile di filosofi può mangiare allo stesso tempo).

```c
while (1) {
  think();
  get_forks(p);
  eat();
  put_forks(p);
}
```

Usiamo due funzioni ausiliarie:

- `int left(int p) { return p; }`
- `int right(int p) { return (p + 1) % 5; }`

Quando il filosofo p desidera fare riferimento alla forchetta alla sua sinistra, chiama semplicemente `left(p)`.

La forchetta alla destra di un filosofo p è indicata chiamando `right(p)`; l'operatore modulo in questo caso gestisce il caso in cui l'ultimo filosofo (`p=4`) tenta di afferrare la forchetta alla sua destra, che è la forchetta 0. Sono inoltre necessari alcuni semafori per risolvere questo problema. Supponiamo di avere cinque, uno per ogni forchetta: `sem_t forks[5]`.

6.5.1 Soluzione broken

Inizializziamo ogni semaforo a 1 in `sem_t forks[5]`. Supponiamo che ogni filosofo conosca il proprio numero p.

```c
void get_forks(int p) {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

Codice 29: Funzioni `get_forks()` e `put_forks()`.

Per ottenere le forchette acquisiamo il lock prima sulla forchetta di sinistra e poi su quella di destra. Finito di mangiare rilasceremo il lock.

Problema: deadlock! Se ogni filosofo afferra la forchetta alla sua sinistra prima che qualsiasi filosofo possa afferrare la forchetta alla sua destra, ognuno rimarrà bloccato con una forchetta e aspettandone un'altra, per sempre.

6.5.2 A Solution: Breaking The Dependency

Una possibile soluzione è cambiare il modo in cui le forchette vengono acquisite dai filosofi.

Il filosofo 4 può prendere forchette in un ordine diverso dagli altri. In questo modo, non si verifica una situazione in cui ogni filosofo afferra una forchetta e rimane bloccato in attesa di un altro.

```c
void get_forks(int p) {
    if (p === 4) {
      sem_wait(&forks[right(p)]);
      sem_wait(&forks[left(p)]);
    } else {
      sem_wait(&forks[left(p)]);
      sem_wait(&forks[right(p)]);
    }
}
```

Codice 30: Interruzione della dipendenza in `get_forks()`.

6.6 Accodamento dei thread

Throttling dei thread: metodo per limitare il numero di thread che eseguono contemporaneamente un pezzo di codice.

Un modo semplice per implementare il throttling dei thread è usare un semaforo. Quando un thread desidera accedere alla regione critica, deve acquisire il semaforo. Se il semaforo è già acquisito da un altro thread, il thread deve attendere fino a quando il semaforo viene rilasciato.

Come il throttling dei thread può essere utilizzato:

Immaginiamo di creare centinaia di thread per lavorare in parallelo. In una parte del codice, ogni thread acquisisce una grande quantità di memoria per eseguire una parte del calcolo.

Se tutti i thread entrano nella regione ad alta intensità di memoria allo stesso tempo, la somma di tutte le richieste di allocazione di memoria supererà la quantità di memoria fisica sulla macchina. La macchina inizierà a swappare e il calcolo rallenterà.

Un semaforo può risolvere questo problema. Inizializzando il valore del semaforo al numero massimo di thread che si desidera far entrare nella regione ad alta intensità di memoria contemporaneamente, solo un numero limitato di thread potrà accedere alla regione alla volta.

6.7 Come implementare i semafori

```c
typedef struct __Zem_t {
  int value;
  pthread_cond_t cond;
  pthread_mutex_t lock;
}
Zem_t;
// only one thread can call this
void Zem_init(Zem_t * s, int value) {
  s -> value = value;
  Cond_init( & s -> cond);
  Mutex_init( & s -> lock);
}

void Zem_wait(Zem_t * s) {
  Mutex_lock( & s -> lock);
  while (s -> value <= 0)
    Cond_wait( & s -> cond, & s -> lock);
  s -> value--;
  Mutex_unlock( & s -> lock);
}

void Zem_post(Zem_t * s) {
  Mutex_lock( & s -> lock);
  s -> value++;
  Cond_signal( & s -> cond);
  Mutex_unlock( & s -> lock);
}
```

Codice 31: Implementazione dei semafori con i lock e le CV.

III Persistenza

1 I/O devices

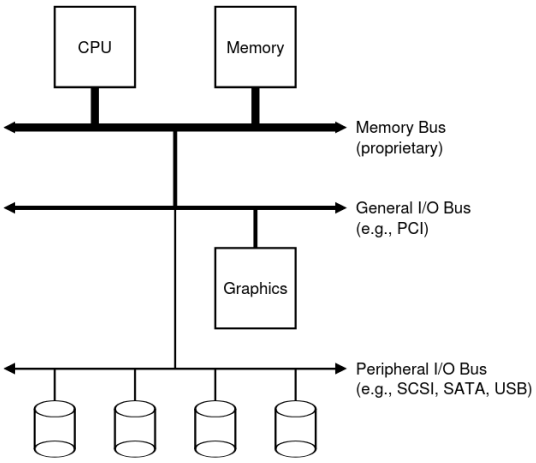


Figura 50: Prototipo dell'architettura di un sistema.

Vantaggi struttura gerarchica:
- I componenti ad alta performance sono più vicini alla CPU, il che migliora le loro prestazioni (bus corto).
- I componenti a bassa performance sono più lontani dalla CPU, il che riduce il loro impatto sulle prestazioni della CPU.
- È possibile posizionare un gran numero di dispositivi su un bus periferico.
- È possibile utilizzare bus diversi per diversi tipi di dispositivi, a seconda delle loro esigenze di prestazioni.

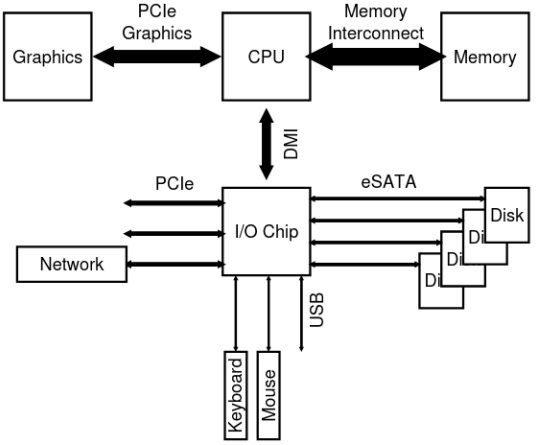


Figura 51: Architettura moderna di un sistema.

Architettura di un sistema moderno:
- Utilizzano chipset specializzati e interconnessioni p2p.
- La CPU si collega direttamente alla memoria RAM e alla GPU per maggiori prestazioni.
- Tramite DMI (Direct Media Interface) si collega al chip I/O.
- Il chip I/O collega i dischi.
- Sotto il chip I/O ci sono le connessioni USB (Universal Serial Bus).
- Infine ci sono le connessioni PCIe (Peripheral Component Interconnect Express) per dispositivi più performanti.

1.1 Interfacce e registri

Un dispositivo canonico ha due componenti importanti:
- **Interfaccia hardware**: consente al software di sistema di controllare il funzionamento del device.
- **Struttura interna**: è specifica dell'implementazione e si occupa di implementare l'astrazione che il dispositivo presenta al sistema.

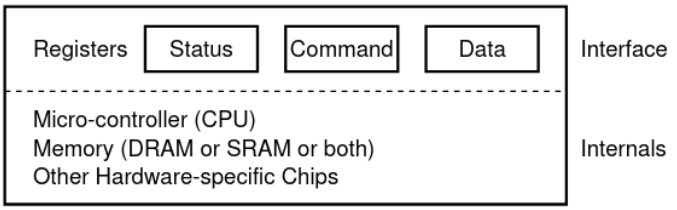


Figura 52: Interfaccia e struttura interna.

L'interfaccia di un dispositivo ha tre registri:
- **Registro di stato**: letto per vedere lo stato attuale del device.
- **Registro dei comandi**: indica al device di eseguire un determinato compito.
- **Registro dei dati**: utilizzato per passare/ottenere dati dal device.

1.2 Canonical protocol

Il protocollo canonico ha quattro fasi:
1. **Polling**: l'OS attende che il dispositivo sia pronto leggendo ripetutamente il suo registro di stato.
2. l'OS invia i dati al dispositivo attraverso il registro dati. Quando la CPU principale è coinvolta nel trasferimento dei dati, il metodo di trasferimento è chiamato I/O programmato (PIO).
3. L'OS scrive un comando nel registro comandi in modo da informare il device che i dati sono presenti e che deve iniziare a lavorare sul comando.
4. La CPU attende che il dispositivo finisca ripetendo il polling in un loop, aspettando di vedere se è terminato (potrebbe quindi ottenere un codice di errore per indicare il successo o il fallimento).

1.3 Ottimizzazioni dell'I/O

In questo approccio ci sono inefficienze (es. polling che spreca tempo della CPU). È possibile ridurre il sovraccarico della CPU utilizzando gli interrupt. Invece di interrogare ripetutamente il dispositivo, l'OS può:
1. Inviare una richiesta.
2. Mettere il processo chiamante in attesa.
3. Passare al contesto di un'altra attività.
4. Quando il dispositivo terminerà l'operazione, invierà un'interruzione hardware che causerà il salto della CPU nell'OS in una routine di servizio di interrupt (ISR) o a un gestore di interrupt.

Il gestore di interrupt:
1. Completerà la richiesta.
2. Risveglierà il processo in attesa dell'I/O che potrà procedere.

Le interruzioni consentono quindi di sovrapporre (overlay) computazione e I/O, migliorando l'efficienza del sistema. Le interruzioni possono essere inefficienti se il dispositivo è veloce, in tal caso, meglio utilizzare il polling. Se la velocità del dispositivo non è nota è possibile utilizzare un metodo ibrido: fare polling per un po' e se il device non ha ancora finito, utilizzare gli interrupt.

Gli interrupt non sono sempre la miglior soluzione quando si hanno molte richieste. In questi casi, è meglio utilizzare il polling per avere un controllo migliore sul sistema e consentire ai processi di livello utente di eseguire le richieste.

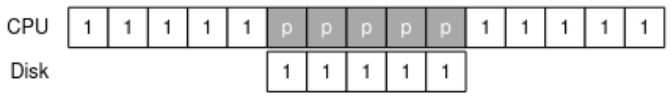


Figura 53: Performance senza l'utilizzo di interrupt.

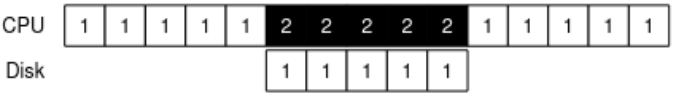


Figura 54: Performance con l'utilizzo di interrupt.

Coalescenza (coalescing): altra ottimizzazione basata sulle interruzioni, che consente di ridurre il sovraccarico della

gestione delle interruzioni. Riesce ad aggregare più interrupt perché non genera subito l'interruzione ma aspetta.

Quando si utilizza PIO per trasferire grande mole di dati, la CPU è sovraccarica e perde tempo.

1.4 DMA (Direct Memory Access)

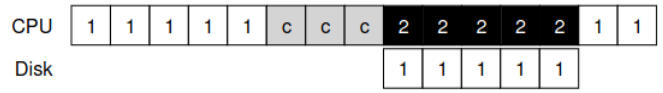


Figura 55: Performance senza l'utilizzo di DMA.

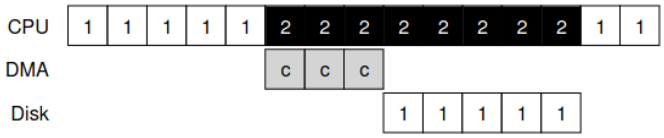


Figura 56: Performance con l'utilizzo di DMA.

DMA (Direct Memory Access): device che consente di accedere direttamente alla memoria centrale del computer, bypassando il processore.

Si vuole trasferire dati a un device. Funzionamento:
1. L'OS programma il DMA specificando dove sono i dati, quanti ne sono e a chi inviarli.
2. L'OS torna a fare altro.
3. Il DMA genera un interrupt per avvisare l'OS che il trasferimento è stato completato.

Due metodi principali di comunicazione con i dispositivi:
- **Istruzioni I/O esplicite**: specificano un modo per l'OS di inviare dati a registri del dispositivo. Occorre che tali istruzioni siano privilegiate.
- **I/O mappato in memoria**: l'hardware rende i registri dei dispositivi disponibili come se fossero locazioni di memoria. Per accedere a un registro particolare, l'OS emette un caricamento (per leggere) o una memorizzazione (per scrivere) l'indirizzo. L'hardware instrada il caricamento/archiviazione al dispositivo invece alla memoria principale.

Le istruzioni I/O esplicite sono più semplici da implementare, ma richiedono istruzioni privilegiate. L'I/O mappato in memoria è più efficiente, ma richiede hardware più complesso.

I device hanno interfacce specifiche e l'OS deve essere in grado di interagire con i dispositivi indipendentemente dalla loro interfaccia specifica (per fare ciò, si utilizza l'astrazione tramite i driver).

Driver: un pezzo di software nell'OS che conosce come funziona un dispositivo.

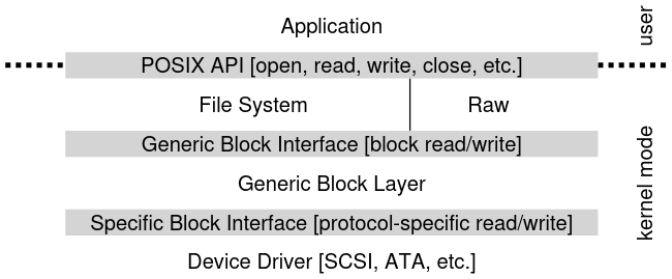


Figura 57: File system stack.

Rappresentazione approssimativa e approssimativa dell'organizzazione del software Linux.

Il FS (File System) non deve conoscere i dettagli specifici del dispositivo hardware a cui è collegato. Il FS emette richieste di lettura/scrittura di blocchi al livello di blocco generico. Il livello di blocco instrada le richieste al driver del dispositivo appropriato. Il driver del dispositivo gestisce i dettagli dell'emissione della richiesta specifica.

Raw interface: abilita applicazioni speciali a leggere e scrivere blocchi direttamente senza utilizzare l'astrazione del file.

L'interfaccia generica può avere lati negativi, poiché può impedire alle applicazioni di sfruttare le capacità speciali di alcuni dispositivi.

2 Hard Disk Driver (HDD)

Il disco rigido è composto da un gran numero di settori (blocchi da 512 byte), ognuno dei quali può essere letto o scritto. Sono possibili operazioni multi-settore. Molti FS leggeranno o scriveranno 4 KB alla volta (o più). L'unica garanzia che i produttori di drive offrono è che una singola scrittura da 512 byte è atomica. Accedere a due blocchi vicini l'uno all'altro all'interno dello spazio di indirizzi del drive sarà più veloce che accedere a due blocchi che sono distanti. Accedere ai blocchi in un blocco contiguo (cioè una lettura o una scrittura sequenziale) sia la modalità di accesso più veloce e di solito molto più veloce di qualsiasi altro modello di accesso casuale.

2.1 Struttura e funzionamento

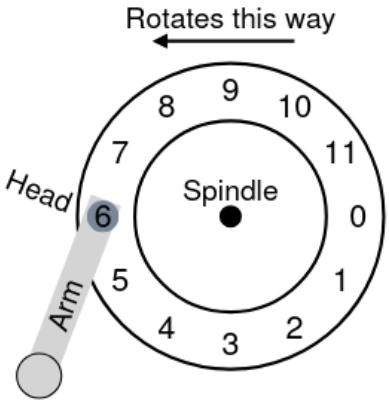


Figura 58: Singola traccia con testa.

Un disco rigido è composto da un piatto (**platter**) rotante che contiene n tracce (**track**) da tutti e due i lati (**surface**). Ogni traccia è composta da un numero di settori (**sector**), che sono le unità di base di memorizzazione dei dati. I settori sono numerati da 0 a $n-1$. Il disco è dotato di una testina (**disk head**) che può leggere e scrivere i dati sui settori. La testina è montata su un braccio (**disk arm**) che può essere spostato per accedere a diverse tracce. Il disco è in continuo movimento, quindi la testina deve essere posizionata sulla traccia corretta al momento giusto per leggere o scrivere i dati.

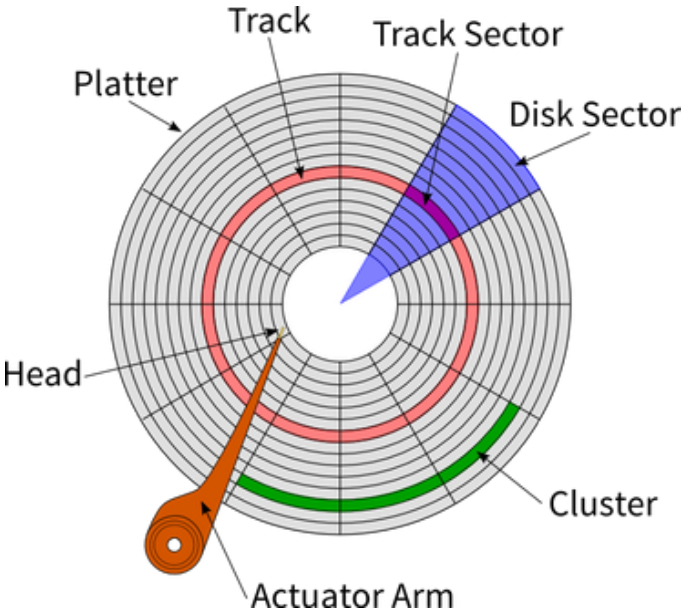


Figura 59: Nomenclatura delle componenti di un HDD.

La velocità di rotazione è costante ed è misurata in RPM (Rotazioni Per Minuto). Il piatto ruota in senso antiorario.

2.2 Tempi di accesso

Ritardo di rotazione: tempo che il disco impiega per far ruotare il settore desiderato sotto la testina di lettura/scrittura. Nel caso peggiore, il ritardo di rotazione è pari alla metà del tempo di rotazione del disco.

La ricerca di un blocco ha più fasi:
- **Acceleration**: il braccio si muove.
- **Coasting**: il braccio si muove a velocità piena per inerzia.
- **Deceleration**: il braccio rallenta.
- **Settling**: il braccio si stabilizza mentre la testina viene posizionata con attenzione.

Seek time: tempo necessario per spostare la testina da una traccia all'altra.

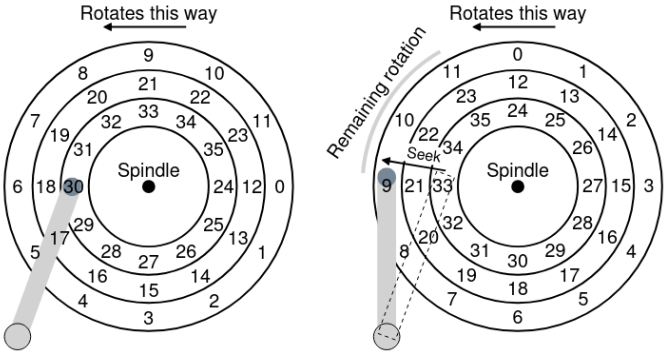


Figura 60: Tre track e una testa.

I settori di un disco rigido sono spesso sfalsati perché quando il braccio si sposta da una traccia all'altra, ha bisogno di tempo per riposizionarsi (anche su tracce adiacenti). Se i settori non fossero sfalsati, il braccio si sposterebbe sulla traccia successiva ma il blocco successivo desiderato sarebbe già passato.

Le tracce esterne tendono ad avere più settori delle tracce interne. Queste tracce sono spesso chiamate **dischi multi-zoned**, dove il disco è organizzato in più zone e ogni zona è un insieme consecutivo di tracce su una superficie. Ogni zona ha lo stesso numero di settori per traccia e le zone esterne hanno più settori delle zone interne.

2.3 Politiche di scrittura

Cache o track buffer: piccola quantità di memoria che il disco può utilizzare per memorizzare i dati letti o scritti sul disco. Ad esempio, quando si legge un settore dal disco, il disco potrebbe decidere di leggere tutti i settori su quella traccia e memorizzarli nella sua memoria; questo consente al disco di rispondere rapidamente a qualsiasi richiesta successiva alla stessa traccia.

In scrittura il disco ha una scelta:
- **Caching write back**: riconoscere che la scrittura è stata completata quando ha messo i dati nella sua cache.
- **Write through**: riconoscere che la scrittura è stata completata dopo che la scrittura è stata effettivamente scritta sul disco.

2.4 Analisi delle performance

Tempo per una rotazione completa:

$$\text{Time (ms)} = \frac{1 \text{ minuto}}{\text{RPM}}$$

Esempio con 10000 RPM:

$$\text{Time (ms)} = \frac{60000 \text{ ms}}{10000 \text{ rotazioni}} = 6 \text{ ms/rotazione}$$

Tempo di trasferimento di un blocco: Se dobbiamo trasferire 512 KB con una velocità di 100 MB/s, il tempo di trasferimento è:

$$T_{\text{transfer}} = \frac{512 \text{ Kb}}{100 \text{ MB/s}}$$

Convertendo le unità:

$$T_{\text{transfer}} = \frac{512 \times 10^3 B}{100 \times 10^6 \text{ B/s}} = \frac{512}{100000} s = 5.12 \text{ ms}$$

Tempo di I/O totale:

$$T_{\text{I/O}} = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}}$$

Velocità di I/O:

$$R_{\text{I/O}} = \frac{\text{Size}_{\text{transfer}}}{T_{\text{I/O}}}$$

Workload sequenziale: legge molti settori contigui sul disco.

Workload casuale: emette piccole letture (es. 4KB) in posizioni randomiche del disco.

Date un insieme di richieste di I/O, lo scheduler del disco le esamina e decide quale programmare successivamente. La durata effettiva di un I/O è sconosciuta ma è possibile stimarla ipotizzando un ritardo della ricerca e il ritardo di rotazione di una richiesta. In questo modo lo scheduler del disco, se è SJF (Shortest Job First), potrà scegliere quale richiesta gestire per prima.

2.5 Disk scheduling

SSTF (Shortest Seek Time First) o **SSF (Shortest Seek First)**: ordina la coda delle richieste I/O in base alla traccia. Completerà prima le richieste vicine alla traccia corrente. Problemi:
1. **Geometria non disponibile**: l'OS lavora con i blocchi senza conoscere la geometria dell'unità. Siccome l'OS vedi il disco come un insieme di blocchi, il problema viene risolto con NBF (Nearest Block First), un algoritmo che pianifica la richiesta che ha l'indirizzo più vicino a quello corrente.
2. **Starvation**: se ci fosse un flusso importante di richieste tutte a blocchi vicini, i blocchi lontani soffrirebbero di starvation.

Sweep: passaggio della testina da una traccia interna su una esterna o viceversa.

Algoritmo dell'ascensore o SCAN: muove la testina in modo sequenziale avanti e indietro. Inizialmente, si muove in una direzione soddisfacendo le richieste lungo il percorso. Quando raggiunge l'estremità, inverte la direzione e continua a soddisfare le richieste nella direzione opposta. Il problema è che le richieste alle estremità, essendo in fondo alla coda, soffriranno di fame. Ha due varianti:

- **F-SCAN**: durante una scansione (sweep), F-SCAN «freeza» la coda in modo che le richieste arrivate durante la scansione vengano messe in una coda temporanea per essere elaborate successivamente. Questo evita la possibilità di fame delle richieste più lontane, ritardando il servizio delle richieste arrivate tardi ma più vicine.
- **C-SCAN o Circular SCAN**: dopo aver raggiunto l'estremità in una direzione, torna rapidamente all'inizio. Questo significa che le richieste agli estremi della coda non subiscono ritardi significativi dovuti all'inversione della direzione.

Questi algoritmi non aderiscono al principio SJF. Soluzione: SPTF (Shortest Positioning Time First).

SPTF (Shortest Positioning Time First): seleziona la richiesta successiva da servire in base al tempo di posizionamento più breve, cercando di ridurre al minimo il tempo di ricerca della testina. SPTF è vantaggioso quando ricerca e rotazione sono approssimativamente equivalenti, ma la sua implementazione è complessa.

La scelta tra richieste più vicine o più lontane dipende dal rapporto tra il tempo di ricerca e il ritardo di rotazione. Se il tempo di ricerca è molto più alto del ritardo di rotazione, algoritmi come SSTF sono adatti, ma se la ricerca è notevolmente più veloce della rotazione, può essere preferibile servire richieste più lontane.

2.6 Formulario per i Dischi Rigidi

Tempo di I/O totale

$$T_{\text{I/O}} = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}}$$

Dove:
- T_{seek} è il tempo di seek (posizionamento della testina sulla traccia corretta).
- T_{rotation} è il tempo di rotazione (attesa che il settore passi sotto la testina).
- T_{transfer} è il tempo di trasferimento (lettura/scrittura dei dati).

Throughput

$$\text{Throughput} = \frac{\text{Size}_{\text{Transfer}}}{T_{\text{I/O}}}$$

Misura la quantità di dati trasferiti per unità di tempo.

Tempo Medio di Seek

$$T_{\{\text{seek}\}} = \frac{1}{3} T_{\text{seek}}^{\max}$$

Dove $T_{\text{seek}}^{\max}$ è il tempo massimo per muoversi tra le tracce più distanti.

Tempo di Rotazione Medio

$$T_{\text{rotation}} = \frac{1}{2} T_{\text{rotation}}^{\max}$$

Dove il tempo di rotazione massimo è dato da:

$$T_{\text{rotation}}^{\max} = \frac{60000}{\text{RPM}} \text{ (in ms)}$$

Tempo di Trasferimento

$$T_{\text{transfer}} = \frac{\text{Size}_{\text{Transfer}}}{\text{Transfer Rate}}$$

Dove:
- $\text{Size}_{\text{Transfer}}$ è la dimensione del trasferimento dei dati.
- Transfer Rate è la velocità di trasferimento del disco.

Velocità di Trasferimento Massima

$$\text{Max Transfer Rate} = \frac{\text{Settori per traccia} \times \text{Dimensione settore} \times \text{RPM}}{60}$$

Dove:
- «Settori per traccia» è il numero di settori per traccia.
- «Dimensione settore» è tipicamente 512B o 4KB.
- «RPM» indica le rotazioni per minuto.

Distanza Media di Seek

$$\frac{1}{N^2} \sum_{x=0}^{N} \sum_{y=0}^{N} |x - y| = \frac{N}{3}$$

Dove N è il numero totale di tracce.

Rateo di I/O

$$R_{\text{I/O}} = \frac{\text{Size}_{\text{Transfer}}}{T_{\text{I/O}}}$$

Indica la velocità con cui le operazioni di I/O vengono eseguite.

Utilizzo del Disco

$$U = \frac{T_{\text{I/O}}}{T_{\text{cycle}}}$$

Dove T_{cycle} è il tempo totale tra due richieste consecutive.

3 Redundant Array of Independent Disks (RAID)

RAID (Redundant Array of Independent Disks): tecnica per utilizzare più dischi insieme per avere più velocità, più capienza e più affidabilità. Viene visto come un gruppo di blocchi che si possono leggere o scrivere. È un sistema complesso con più dischi, memoria volatile e persistente e anche un processore per gestire il sistema.

Vantaggi:
- **Prestazioni**: l'utilizzo di dischi in parallelo accelera i tempi di I/O.
- **Capacità**: è possibile storare grandi quantità di dati.
- **Affidabilità**: consente di avere forme di ridondanza.

Quando il file system invia una richiesta di I/O al RAID, il RAID deve calcolare a quale disco accedere.

Il RAID fornisce tutto ciò in maniera trasparente all'OS che lo vede come un unico grande disco. Quindi, non occorre fare alcuna modifica all'OS per renderlo compatibile al RAID. Quando il file system invia una richiesta di I/O al RAID, il RAID deve calcolare a quale disco accedere. Il RAID è spesso hardware separato formato da un microcontroller che esegue il software per il funzionamento e una memoria volatile DRAM per bufferizzare i blocchi che vengono letti/scritti.

Sono progettati per rilevare e ripristinare gli errori del disco.

Modello fail-stop: il disco o è funzionante o è guasto. Se funziona, tutti i blocchi possono essere scritti/letti. Se è guasto, i dati sono persi in modo permanente.

Il controller RAID (software o hardware) riconosce se il RAID è guasto.

Obbiettivi del RAID:
1. Distribuire l'informazione su più dischi in modo da parallelizzare una parte delle operazioni di accesso ai dati e guadagnare in prestazioni.
2. Duplicare su più dischi l'informazione memorizzata, in modo tale che, in caso di guasto di un disco, sia comunque possibile mantenere funzionante il sistema recuperando in qualche modo l'informazione perduta.

Come valutare un RAID:
- **Capacità**: dati N dischi con B blocchi, la capacità disponibile:
  - se non si ha ridondanza è $N \cdot B$.
  - se si hanno 2 copie per ogni blocco è $\frac{N \cdot B}{2}$.
- **Affidabilità**: quanti errori del disco può tollerare
- **Prestazioni**: dipende dal workload dato al RAID.

3.1 RAID 0

RAID 0 (striping): non è considerato un vero RAID. Consiste nel distribuire i blocchi disponibili in modo RR tra i dischi disponibili per aumentare il parallelismo in lettura/scrittura.

Alcune definizioni per valutare le performance del RAID:
- $\boldsymbol{S}$: MB/s under a sequential workload.
- $\boldsymbol{R}$: MB/s under a random workload.

Proprietà:
- **Capacità**: $N \cdot B$ blocchi di capacità utili.
- **Affidabilità**: qualsiasi guasto comporta la perdita di tutti i dati.
- **Performance**: eccellente perché vengono utilizzati tutti i dischi in parallelo.
  - Latenza di una singola richiesta: quasi identica a quella per singolo disco dato che il RAID deve inviare la richiesta a un suo disco.
  - Throughput sequenziale in stato stazionario: intera larghezza di banda ($N \cdot S$).
  - Throughput random in stato stazionario: $N \cdot R$.

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figura 61: Raid 0 o striping.

Chunk size: numero di blocchi in un chuck. Incide sulle performance del RAID.

In Figura 61 abbiamo che:
- Chunk size = 1 block.
- Block size = 4 Kb.
- Stripe size = 16 Kb.

Abbiamo un problema di mapping dato che il RAID deve sapere in quale disco e l'offset a cui accedere. Soluzione:
- $\text{Disk} = \text{Address\_of\_block} \ \% \ \text{Number\_of\_disk}$
- $\text{Offset} = \frac{\text{Address\_of\_block}}{\text{Number\_of\_disk}}$

Dimensione dei chunk:
- **Chunk troppo grande**: riduce il parallelismo intra-file, contando maggiormente su richieste multiple contemporanee per ottenere un'elevata capacità di trasferimento.
- **Chunk troppo piccolo**: più piccolo implica che molti file saranno distribuiti su molti dischi, aumentando così il parallelismo a discapito dei tempi di posizionamento.

Valutazione delle performance generali:
- **Latenza di una singola richiesta**: permette di capire quanto parallelismo puo esistere durante una singola richiesta I/O.
- **Throughput in stato stazionario di un RAID**: capacità totale di gestire molte richieste contemporanee. Consideriamo due tipi di carichi di lavoro:
  - **Workload Sequenziale**: le richieste coinvolgono blocchi contigui di dati.
  - **Workload Random**: richieste piccole e rivolte a una posizione casuale sul disco.

Valutazioni del RAID 0:
- **Capacità**: dati N dischi, ognuno composto da B blocchi, il RAID di livello zero fornisce $N \cdot B$ blocchi di capacità utile.
- **Affidabilità**: il RAID di livello 0 è terribile: ogni disk failure porterà a una perdita di dati.
- **Performance**: eccellente. tutti i dischi sono utilizzati, spesso in parallelo, per servire le richieste di I/O.
- **Single-block-latency**: dovrebbe essere identica a quella di un singolo disco; dopotutto, RAID-0 semplicemente reindirizza la richiesta a uno dei dischi.
- **Steady-state throughput**: ci aspettiamo di avere tutta la larghezza di banda del sistema. Il throughput sarà uguale a N (il numero di dischi) moltiplicato per S (la banda sequenziale di un singolo disco). Per un elevato numero di I/O random (workload introdotto la lezione scorsa), possiamo ancora una volta usare tutti i dischi, ottenendo quindi $N \cdot R$ MB/s.

Nota bene: $S = \frac{\text{Size}_{\text{transfer}}}{\text{Time}_{\text{I/O}}}$ rappresenta l'I/O rate di un workload sequenziale, $R = \frac{\text{Size}_{\text{transfer}}}{\text{Time}_{\text{I/O}}}$ quello di un workload random.

3.2 RAID 1

RAID 1 (Mirroring): consiste nel creare più di una copia di ogni blocco su un disco separato. Ad ogni richiesta di lettura, è possibile leggere una copia del blocco richiesto da qualsiasi disco. In caso di una richiesta di scrittura, il RAID deve aggiornare tutte le copie su tutti i dischi per preservare l'affidabilità.

Proprietà (mirroring = 2):
- **Capacità**: $\frac{N \cdot B}{2}$ blocchi di capacità utili (la metà della capacità disponibile).
- **Affidabilità**: tollera il guasto di uno dei due dischi.
- **Performance**: normali.
  - **Latenza di una singola richiesta di lettura**: identica a quella per singolo disco.

- **Latenza di una singola richiesta di scritture**: occorrono due scritture per il completamento della richiesta (tempo identico circa).
- **Throughput sequenziale in stato stazionario**: ogni scrittura logica deve corrispondere a due scritture fisiche ($\frac{N}{2} \cdot S$).
- **Throughput random in stato stazionario**: miglior soluzione, distribuzione delle letture su tutti i dischi $N \cdot R$.

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Figura 62: Raid 1 o mirroring.

Combinazioni di RAID 1 e di RAID 0:

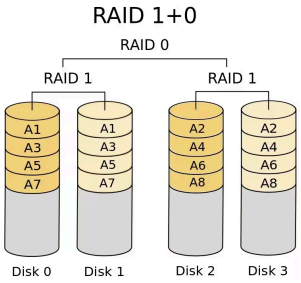


Figura 63: Raid 1 + 0.[6]

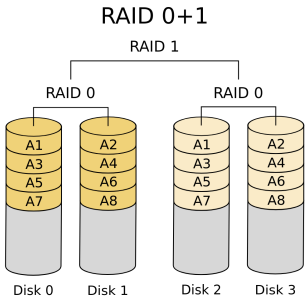


Figura 64: Raid 0 + 1 mirroring.[7]

3.3 RAID 4

RAID 4: i dati vengono distribuiti su più dischi e un disco è dedicato per la parità.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figura 65: RAID 4 con parità.

Calcolo della parità: si utilizza la funzione XOR. L'XOR di un insieme di bit restituisce:
- 0 se il numero di bit pari a 1 è pari.
- 1 se il numero di bit pari a 1 è dispari.

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

Figura 66: Calcolo della parità.

Block0	Block1	Block2	Block3	Parity
00	10	11	10	11
10	01	00	01	10

Figura 67: Calcolo della parità con blocchi.

Il RAID livello 4 risparmia dischi rispetto al RAID di livello 1 e garantisce lo stesso il mantenimento dei dati in caso di guasto di un disco, ma al costo di una maggiore inefficienza. Infatti, ogni qualvolta una strip di un disco viene modificata, occorre leggere anche ricalcolarne la parità. Inoltre, il disco che ospita le strip di parità è pesantemente coinvolto in ogni operazione di scrittura sul RAID e può facilmente diventare un collo di bottiglia.

Proprietà:
- **Capacità**: $(N-1)B$, poiché un disco viene usato per la parità.
- **Affidabilità**: RAID 4 tollera il fallimento su un solo disco, non di più.
- **Performance**: Le performance di questi sistemi RAID non sono il massimo, anzi. Sono abbastanza sicuri ma, avendo un unico disco con le parità, siamo costantemente in attesa e di conseguenza tutte le operazioni a quel disco sono sequenziali.
- **Single-block-latency**: la latenza di una singola write richiede 2 read e 2 write (due per il dato e due per la parità). Le read possono essere fatte in parallelo, così come le write, la latenza totale è quindi doppia rispetto al singolo disco (con qualche differenza visto che dobbiamo aspettare che entrambe le reads vengano portate a termine).
- **Steady-state throughput**: consideriamo i nostri due workload, sequenziale e random. Nel caso di:
  - Scrittura sequenziale in tutta una striscia: la banda equivale a (N - 1) S MB/s (immaginandoci di ricevere una write per i blocchi 0, 1, 2 e 3, nonostante utilizziamo anche il blocco di parità in parallelo non ne traiamo un reale beneficio, per questo la banda è N-1 e non N).
  - Una random read: anche in questo caso le performance saranno pari a (N - 1) R MB/s perché non abbiamo bisogno di leggere il disco di parità.
  - Random writes invece: immaginiamo di voler scrivere il blocco 1 ad esempio. Potremmo semplicemente sovrascriverne il contenuto ma sorge un problema: il valore di parità va aggiornato perché potrebbe non essere più corretto.

Ci sono due metodi per andare a risolvere questo problema:
- **Additive parity**: per sapere il valore di parità si legge in parallelo il valore dei blocchi nella striscia e si esegue una XOR con il valore del nuovo blocco. Il problema di questa tecnica è che maggiore sono i dischi e più costosa diventa.
- **Subtractive parity**.

3.3.1 Small Write Problem

Il small write problem è una limitazione delle prestazioni nei sistemi RAID con parità (come RAID 4, 5 e 6) causata dalla gestione delle scritture su piccoli blocchi di dati.

Quando viene modificato un singolo bit (es. C2 $\rightarrow$ $C2_{new}$), il disco di parità deve essere aggiornato per mantenere la consistenza. Questo aggiornamento segue il metodo subtractive, che prevede:
1. Lettura del dato originale ($C2_{old}$) e della parità corrente (P_{old}).
2. Calcolo della nuova parità confrontando il vecchio e il nuovo valore del dato:
3. $P_{new} = (C_{old} \oplus C_{new}) \oplus P_{old}$
4. Scrittura del nuovo dato e della nuova parità.

Soluzioni al Small Write Problem:
- **RAID con parità distribuita (es. RAID 5 e RAID 6)**: distribuiscono il carico della parità su più dischi, riducendo il problema.
- **Cache di scrittura (NVRAM o SSD)**: accumula piccole scritture e le applica in batch per ridurre l'overhead sul disco di parità.
- **RAID 10 (Striping + Mirroring)**: evita il problema eliminando la necessità di calcolare la parità.

3.4 RAID 5

RAID 5: simile al RAID 4 ma il blocco di parità ruota sui dischi per risolvere il problema della scrittura ridotta.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Figura 68: RAID 5 con rotated parity.

Proprietà:

- **Capacità**: la capacità è uguale a quella del RAID-4, $(N-1)B$.
- **Affidabilità**: il RAID-5 e il RAID-4 offrono lo stesso livello di affidabilità.
- **Performance**: il RAID-5 rispetto al livello 4 ha performance notevolmente migliorate grazie alla disposizione dei blocchi di parità all'interno dei dischi. Con questa disposizione siamo in grado di eliminare l'effetto "collo di bottiglia" delle small write.
- **Single-write-latency**: è invariata rispetto al RAID-4, una write richiede 2 read e 2 write, la latenza è doppia rispetto a quella di un singolo disco.
- **Steady-state throughput**: nel workload random, il RAID-5 funziona leggermente meglio rispetto al RAID-4 perché ora possiamo utilizzare tutti i dischi. Infine, le performance di una write random è notevolmente migliorata rispetto al RAID-4 visto che ora siamo in grado di parallelizzare le richieste e non dobbiamo più accedere sequenzialmente al disco di parità.

Immaginiamo una write al blocco 1 e una write al blocco 10; verrà tradotta in una richiesta al disco 1 e una al disco 4 (per il blocco 1 e la sua parità) e una richiesta per il disco 0 e 2 (per il blocco 10 e la sua parità). In questo modo possono procedere in parallelo. Possiamo assumere che, dato un elevato numero di richieste random, saremo in grado di tenere occupati praticamente tutti i dischi. Se è questo il caso, allora la nostra larghezza di banda (bandwidth) per piccole writes sarà di (N/4)R MB/s. Il fattore di perdita 4 è dovuto al fatto che ogni write in un sistema RAID-5 genera 4 operazioni di I/O, che è semplicemente il costo dovuto all'impiego della parità (read data, read parity, write data, write parity)

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \cdot B$	$(N \cdot B)/2$	$(N-1) \cdot B$	$(N-1) \cdot B$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S^1$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S^1$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4}R$
Latency				
Read	T	T	T	T
Write	T	T	$2T$	$2T$

Figura 69: Capacità, affidabilità e performance dei RAID.

4 File e Directory

Dispositivo di archiviazione permanente: disco rigido o solido che memorizza informazioni in modo permanente.

File: array lineare di byte scrivibili e leggibili. Ogni file ha un nome a basso livello o **inode number** (numeri di qualche tipo).

File system (FS): archivia i dati in modo persistente su disco.

Directory: file che contiene un elenco di coppie (nome leggibile, inode number), anche le directory hanno un inode number e quindi ogni entry dell'elenco si riferisce a un file o a una directory.

Albero delle directory: creato quando si inseriscono directory in altre directory. La gerarchia inizia dalla root directory (`/`).

Indicare **l'estensione** del file nel nome arbitrario dopo il punto "." è una convenzione.

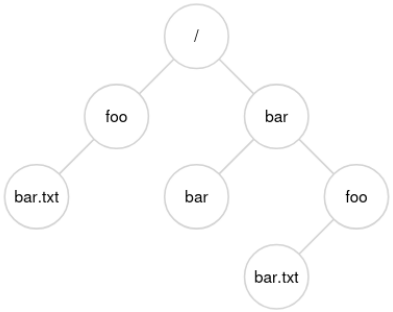


Figura 70: Albero delle directory.

4.1 `unlink` e `open()`

`unlink()`: system call per rimuovere i file.

```c
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

- `foo`: path/nome del file.
- `O_CREAT`: crea il file se non esiste.
- `O_WRONLY`: apre il file in sola scrittura.
- `O_TRUNC`: rade a zero il contenuto del file.
- `S_IRUSR`: l'utente ha il permesso di leggere il file.
- `S_IWUSR`: l'utente ha il permesso di scrivere sul file.

`open()` restituisce un **file descriptor** (numero intero privato per processo) utilizzato per accedere al file. I descrittori sono gestiti dall'OS in base al processo. Uno **struct proc** mantiene un array che tiene traccia dei file (c'è un limite massimo NOFILE) aperti in base al processo. Ogni entry dell'array è un puntatore a **struct file** che traccia le informazioni sul file letto/scritto.

Definizione semplificata di struct file:

```c
struct file {
  int ref;
  char readable;
  char writable;
  struct inode *ip;
  uint off;
};
```

Ogni processo mantiene un array di descrittori di file, ognuno dei quali si riferisce a una voce nella **tabella di file aperti** (struct file) a livello di sistema. Ogni voce tiene traccia di quale file corrisponde al descrittore, dell'offset corrente, se il file è leggibile o scrivibile ecc.

4.2 Lettura e scrittura

`strace`: traccia ogni sys call effettuata da un programma in esecuzione e la stampa a schermo.

Ogni processo in esecuzione ha tre file aperti:
- STDIN (0): letto dal processo per ricevere input.
- STDOUT (1): scritto dal processo per stampare a schermo.
- STDERR (2): scritto dal processo per comunicare errori.

`read()`: legge ripetutamente byte da un file e restituisce il numero di byte letti.

```c
ssize_t read(int fd, void *buf, size_t count);
```

- `fd`: file descriptor del file che si vuole leggere.
- `*buf`: puntatore a un buffer in cui posizionare il risultato di read.
- `count`: dimensione del buffer.

`read()`: legge ripetutamente byte da un file e restituisce il numero di byte scritti.

```c
ssize_t write(int fd, void *buf, size_t count);
```

- `fd`: file descriptor del file che si vuole scrivere.
- `*buf`: puntatore a un buffer in cui posizionare il risultato di read.
- `count`: dimensione del buffer.

Funzionamento di `cat`:
1. `cat` chiama `printf()` e non `write()` sul fd di stdout standard. `printf()` gestisce tutti i dettagli di formattazione e scrive sull'stdout.
2. `cat` cerca di leggere altri dati dal file, ma siccome non ci sono più byte nel file, la chiamata `read()` restituisce 0 e il programma sa che ha letto tutto il file.
3. `cat` chiama `close()` per terminare l'utilizzo del file.

Scrittura di un file:
1. Il file viene aperto.
2. Viene chiamata la funzione `write()` (ripetutamente per file più grandi).
3. Viene chiamata la funzione `close()`.

Accesso sequenziale: lettura o scrittura dall'inizio alla fine del file.

4.3 lettura/scrittura non sequenziale

`lseek()`: sposta il puntatore di lettura/scrittura di un file. Restituisce il nuovo offset del puntatore di lettura/scrittura.

```c
off_t lseek(int fd, off_t offset, int whence);
```

- `fd`: file descriptor del file.
- `offset`: nuovo offset del puntatore di lettura/scrittura.
- `whence`: specifica il punto di riferimento per l'offset.
  - `SEEK_SET`: offset è relativo all'inizio del file.
  - `SEEK_CUR`: offset è relativo alla posizione corrente del puntatore di lettura/scrittura.
  - `SEEK_END`: offset è relativo alla fine del file.

Metodi di aggiornamento per offset corrente:
1. Quando avviene una lettura o scrittura di N byte, N va aggiunto all'offset corrente.
2. Usare `lseek()`.

Il mapping tra fd e entry nella tabella dei file aperti è in genere un'associazione uno a uno. Quindi due processi che scrivono sullo stesso file avranno entry diverse. Ogni processo ha il proprio accesso al file e non può modificare il contenuto che è stato modificato da un altro processo.

Quando un processo padre crea un processo figlio con `fork()`, un'entry nella tabella dei file aperti è condivisa. Il processo figlio può accedere ai file aperti dal processo padre e modificarne il contenuto.

`dup()`: crea un nuovo descrittore che fa riferimento a un file già aperto. `dup2()`: sostituisce il descrittore con uno nuovo.

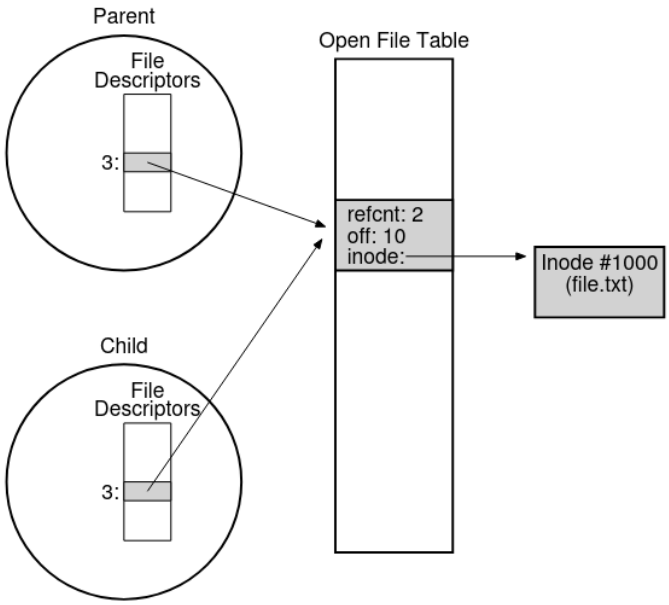


Figura 71: Processi che condividono una voce della tabella dei file aperti.

Quando l'OS chiama `write()`, il FS bufferizza i dati per poi scriverli effettivamente su disco in un secondo momento. Il problema e che questi dati, in caso di arresto anomalo, potrebbero andare persi. Per forzare la scrittura si utilizza `fsync(int fd)` che scrive immediatamente sul disco.

4.4 `remane()`

Per eseguire il comando (Rinominare un file da "foo" a "bar"):

```bash
mv foo bar
```

mv utilizza la syscall `rename(char *old, char *new)`:
- `char *old`: nome originale.
- `char *new`: nuovo nome.

Spesso, `rename()` è implementata in modo atomico.

```c
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

Modifica atomica di un file foo.txt:
1. Viene creato un file temporaneo
2. Viene aggiunta una riga al file temporaneo.
3. Viene forzata la scrittura sul file temporaneo.
4. Chiusura file temporaneo.
5. Sostituzione del file temporaneo con quello effettivo (ridenominandolo).

Il FS mantiene informazioni sui file utilizzando i metadati (visibili tramite `stat()` e `fstat()`). Le chiamate riempiono una struttura stat del tipo:

```c
struct stat {
    dev_t st_dev; // ID of device containing file
    ino_t st_ino; // inode number
    mode_t st_mode; // protection
    nlink_t st_nlink; // number of hard links
    uid_t st_uid; // user ID of owner
    gid_t st_gid; // group ID of owner
    dev_t st_rdev; // device ID (if special file)
    off_t st_size; // total size, in bytes
    blksize_t st_blksize; // block size for filesystem I/O
    blkcnt_t st_blocks; // number of blocks allocated
    time_t st_atime; // time of last access
    time_t st_mtime; // time of last modification
    time_t st_ctime; // time of last status change
```

```
15 };
```

Ogni FS conserva queste informazioni in un inode.

Per rimuovere un file, l'OS chiama `unlink()`.

Il formato di una directory è considerato metadati del FS, quindi, non è possibile aggiornare una directory direttamente.

4.5 Creazione di directory

`mkdir()`: syscall che crea una cartella.

Una nuova directory è considerata vuota ma ha due voci:
- Entry che fa riferimento a se stessa ..
- Entry che fa riferimento al parent ...

Per aprire una directory non bisogna trattarla come un file, esistono le chiamate:
- `opendir()`: apre una directory specificata dal percorso e restituisce un puntatore a una struttura di tipo `DIR`.
- `readdir()`: utilizzata per leggere la prossima voce della directory aperta con `opendir()`. Restituisce un puntatore alla struttura `struct dirent` che contiene le informazioni sull'elemento della directory.
- `closedir()`: chiude la directory precedentemente aperta con `opendir()`. Rilascia le risorse associate alla gestione della directory.

```c
int main(int argc, char *argv[]) {
  DIR *dp = opendir(".");
  assert(dp != NULL);
  struct dirent *d;
  while ((d = readdir(dp)) != NULL) {
  printf("%lu %s\n", (unsigned long) d->d_ino,
  d->d_name);
  }
  closedir(dp);
  return 0;
}
```

Il ciclo legge una voce della dir alla volta e ne stampa il nome e il numero di inode di ogni file nella dir.

```c
struct dirent {
  char d_name[256]; // filename
  ino_t d_ino; // inode number
  off_t d_off; // offset to the next dirent
  unsigned short d_reclen; // length of this record
  unsigned char d_type; // type of file
};
```

Informazioni disponibili all'interno di ogni entry.

- `rmdir()`: utilizzata per rimuovere una dir vuota.
- `link()`: crea un'entry nell'albero del FS.

```c
int link(const char *oldpath, const char *newpath);
```

- oldpath: il percorso del file esistente al quale si desidera creare un collegamento.
- newpath: il percorso del nuovo nome (link) che si desidera associare al file esistente.

4.6 Hard Link

Quando un nome di un file viene linkato a un altro, si crea un modo per fare riferimento allo stesso file. Quindi `link()` crea un altro nome per fare riferimento allo stesso numero inode del file originale.

`unlink()` rimuove il collegamento tra nome leggibile e inode e decrementa il reference count. Se il reference count è 0, il file viene effettivamente cancellato.

Hard Link: collegamento fisico tra due nomi di file differenti che puntano allo stesso inode. Gli hard link condividono lo stesso spazio di archiviazione e le stesse informazioni di inode.

Limiti degli hard link:
- Non è possibile creare un hard link a una directory per evitare la creazione di cicli nella struttura delle directory.
- Non è possibile creare hard link a file su altre partizioni del disco perché i numeri di inode sono univoci solo all'interno di uno specifico FS.

4.7 Symbolic Link

Soft Link (Link Simbolico): è un file separato che contiene un percorso che punta al file di destinazione. Il soft link è un riferimento indiretto al file di destinazione. Contiene il percorso del file di destinazione. Se il file di destinazione viene rimosso, si crea una «dangling reference», e il link simbolico punterà a un percorso inesistente.

Siccome i file sono condivisi tra i processi e gli utenti, all'interno del FS è presente un modo per abilitare vari tipi di permessi.

Permission bits: determinano chi può accedere a un file e come. Sono rappresentati da nove caratteri, suddivisi in tre gruppi:

- Owner.
- Gruppo di utenti.
- Altri utenti.

Le autorizzazioni includono la lettura, la scrittura e l'esecuzione. L'owner può modificare i permessi con `chmod`.

Execute Bit: esecuzione determina se un file può essere eseguito. Nelle directory, abilita la navigazione e la creazione di file.

Access Control Lists (ACL): alcuni file system, come AFS, le utilizzano per controlli più sofisticati. Consentono di specificare dettagliatamente chi può accedere a una risorsa, superando le limitazioni dei permission bits.

4.8 Montaggio di un FS

Per creare un FS si utilizza `mkfs` (make fs). Si fornisce a `mkfs` un dispositivo (es. /dev/sda1) e un tipo di FS (es. ext3). Poi `mkfs` scrive un FS vuoto sulla partizione del disco specificata, iniziando con una directory radice.

Per rendere accessibile il FS si utilizza `mount`, che effettua la chiamata di sistema `mount()` che monta effettivamente il FS. `mount` prende un percorso di directory esistente come punto di montaggio e incolla il nuovo FS nell'albero delle directory in quel punto.

```bash
mount -t ext3 /dev/sda1 /home/users
```

4.9 TOCTTOU

Problema del Time Of Check To Time Of Use (TOCTTOU) : si verifica quando c'è un intervallo tra la verifica di validità di una condizione e l'operazione associata a tale verifica, aprendo la possibilità di manipolazioni da parte di un attaccante.

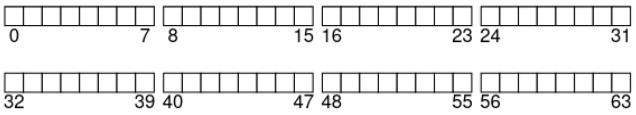
5 File System (FS)

Per la costruzione di un file system VSFS (Very Simple File System) bisogna comprendere due aspetti principali:
1. **Strutture dati del file system**: che tipo di strutture utilizza il file system su disco per mantenere e organizzare dati e metadati.
2. **Metodo di accesso**: come vengono mappate le chiamate fatte da un processo (es. `open()`, `write()`).

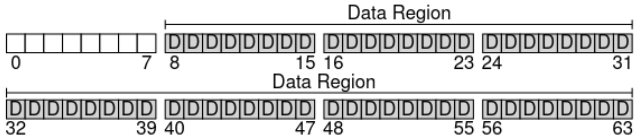
5.1 Struttura di VSFS

Organizzazione del File System sul disco:

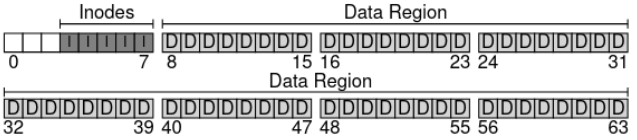
1. Il disco viene diviso in **blocchi** (es. 4KB).



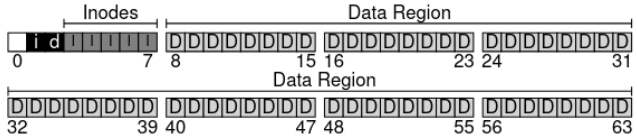
2. La maggior parte dello spazio del FS sarà destinato ai dati dell'utente (data region).



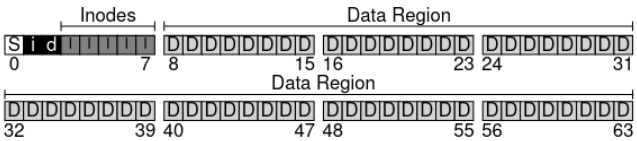
3. Il FS traccia le informazioni di ciascun file (**metadati**) e le conserva in una struttura chiamata **inode**. Va riservato lo spazio per la inode table. Supponendo 256 byte per inode, un blocco da 4 KB può contenerne 16. Quindi il massimo numero di file che si può avere è 16 inode · 5 blocchi = 80 file.



4. Occorre una struttura per tracciare se gli inode o i blocchi dati sono allocati o liberi. possiamo utilizzare:
   - **Free List**: punta al primo blocco libero e così via.
   - **Bitmap**: si utilizza una data bitmap e un'inode bitmap che indicano se l'oggetto/blocco corrispondente è libero (0) oppure in uso (1).



5. Lo spazio rimanente sarà occupato dal **superblocco**: contiene informazioni sul file system (n. inode, n. blocchi, ecc.), include anche il magic number per identificare il tipo di FS. Il super blocco verrà letto dall'OS per montare il FS.



5.2 Inode

inode (index node): struttura che contiene i metadati di un file. Ogni inode è implicitamente identificato da un numero detto «i-number», che funge da nome di basso livello.

inode number: identificatore univoco assegnato a ogni file o directory.

Dato un i-number, calcolare la posizione dell'inode corrispondente su disco. Supponendo di avere:

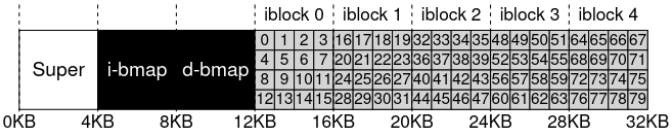


Figura 72: Inode table.

L'obbiettivo è leggere l'i-number 32.

Assunzioni:
- SizeOf(inode) = 512 byte.
- InodeStartAddr = 12 Kb

1. Calcolare l'offset della regione dell'inode:

$$\text{InodeOffset} = 32 \cdot \text{SizeOf(inode)}\} = 8192 \text{ byte}$$

2. Trova l'indirizzo assoluto del blocco dell'inode cercato:

$$\text{Indirizzo Assoluto} = \\ \text{InodeOffset} + \text{InodeStartAddress} = 20 \text{ Kb}$$

3. Trasforma l'indirizzo assoluto in un numero di settore:

$$\text{SectorNumber} = \frac{\text{AbsoluteAddr}}{\text{SectorSize}} = \frac{20 \cdot 1024}{512} = 40$$

Formula generale:

$$\text{Block} = \frac{\text{iNumber} \cdot \text{SizeOf(iNode)}}{\text{BlockSize}}$$

$$\text{Sector} = \frac{21}{\text{SectorSize}}$$

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Figura 73: Ext2 Inode Simplified.

5.3 Indice a più livelli

È importante come un inode fa riferimento ai blocchi dei dati. Approcci:
- **Puntatori diretti**: all'interno dell'inode dove ciascun puntatore si riferisce a un blocco di disco appartenente al file. L'approccio è limitato perché in caso di file di grosse dimensioni servirebbero troppi puntatori che superano la dimensione del blocco.
- **Puntatore indiretto**: invece di puntare a un blocco che contiene dati, punta a un blocco che contiene puntatori a dati utente. Un inode contiene un numero fisso di puntatori diretti o un singolo puntatore indiretto. Se il file cresce, viene allocato un blocco indiretto (nel data region) e lo slot dell'inode per un puntatore indiretto è impostato per puntare ad esso. Così aumenta la dimensione massima del file.
- **Doppio puntatore indiretto**: puntatore a un blocco che contiene altri puntatori a blocchi indiretti, ciascuno dei quali contiene puntatori a blocchi dei dati. Così aumenta la dimensione massima del file.

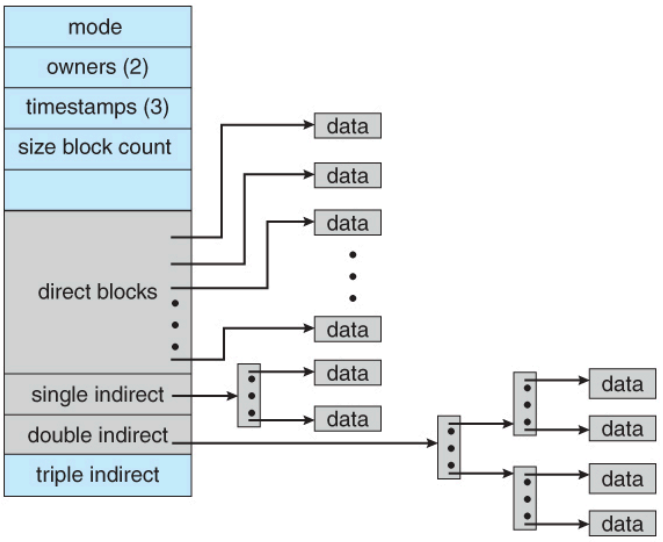


Figura 74: Esempi dei puntatori.

- **Direct indexing**: il file può essere grande massimo 12 blocchi, ciascuno di dimensione 4 K.
- **Single indirect indexing**: il puntatore non punta a un blocco che contiene dati ma a un blocco che contiene altri puntatori. Ciascuno di questi puntatori punta a un blocco che contiene i dati utente. Avendo puntatori da 4-bytes (perché ogni indirizzo è grande 4-bytes), si ha un blocco di puntatori suddiviso in fette da 4 byte (1024 indirizzi da 4 bytes ciascuno).

Un file può essere grande $(12 + 1024) * 4\ \text{Kb} = 4144\ \text{Kb}$.

3. **Double indirect indexing**: il puntatore punta a un blocco di puntatori, ciascuno dei quali punta a un altro blocco di puntatori, ciascuno dei quali, infine, punta a un blocco che contiene i dati utente. In questo caso un file può essere grande massimo $(12 + 1024 + 10242) \cdot 4\ \text{Kb} = 4\ \text{Gb}$.

- Triple indirect indexing: in questo caso riesco a indicizzare un numero enorme di blocchi,

riuscendo a supportare file di dimensioni notevoli. Nello specifico si ha $(12 + 1024 + 10242 + 10243) * 4\ \text{Kb} = 4\ \text{Tb}$.

1. **Lista concatenata**: l'inode contiene un solo puntatore che indica il primo blocco del file, e per gestire file più grandi, vengono aggiunti ulteriori puntatori ai blocchi successivi. Ma questo può avere prestazioni inferiori. Per migliorare ciò, alcuni sistemi mantengono una tabella in memoria dei puntatori successivi per consentire accessi casuali. Un esempio di tale approccio è il sistema di file FAT utilizzato nei sistemi Windows precedenti a NTFS.

5.4 Gestione dello spazio libero

Un file system deve tracciare quali inode e blocchi dati sono liberi o occupati per allocare spazio ai nuovi file e directory.

- In vsfs, questo compito è svolto da due bitmap, una per gli inode e una per i blocchi dati.
- Quando si crea un file, il file system cerca un inode libero, lo assegna e lo segna come usato aggiornando la bitmap su disco.
- Lo stesso processo avviene per l'allocazione dei blocchi dati.

Alcuni file system, come ext2 ed ext3, adottano strategie per ottimizzare le prestazioni, ad esempio cercando blocchi contigui (es. gruppi di 8 blocchi liberi) per migliorare l'accesso sequenziale ai file. Questo metodo, detto pre-allocazione, aiuta a mantenere le porzioni dei file più compatte sul disco, aumentando l'efficienza.

In VSFS (Very Simple File System) una directory contiene coppie voce-inode number per ogni file o directory.

I FS vedono le directory come file speciali. La directory ha un inode nella tabella degli inode. I blocchi di dati della directory sono puntati dall'inode e risiedono nella regione dei blocchi di dati del sistema di file.

5.5 Directory Organization

Nei file system come VSFS, una directory è semplicemente una lista di coppie (nome file, numero inode) memorizzate nei blocchi dati della directory.

Struttura di una directory:

- Numero inode del file o directory.
- Lunghezza del record (dimensione del nome più eventuale spazio extra).
- Lunghezza del nome.
- Nome del file o directory.
- Ogni directory include sempre due voci speciali:
  - . (dot): riferimento alla directory stessa.
  - .. (dot-dot): riferimento alla directory padre.

```
inum | reclen | strlen | name
   5      12        2     .
   2      12        3     ..
  12      12        4     foo
  13      12        4     bar
  24      36       28     foobar_is_a_pretty_longname
```

Quando un file viene eliminato (`unlink()`), si crea uno spazio vuoto nella directory. Questo spazio può essere riutilizzato per nuove entry di dimensioni maggiori.

Le directory sono gestite come file speciali, con un proprio inode. Gli inode contengono puntatori ai blocchi dati della directory, memorizzati nella regione dei dati del file system.

File system avanzati come XFS usano strutture più efficienti come B-tree, migliorando le prestazioni delle operazioni di creazione ed eliminazione di file rispetto alle liste lineari.

5.6 Lettura su disco

Per leggere un file, il file system deve individuare il suo inode partendo dal suo percorso completo. Supponiamo di aprire `/foo/bar`, un file da 12KB (3 blocchi).

```c
1     open("/foo/bar", O_RDONLY);
```

1. Apertura del File (`open()`)
   1. Il file system inizia dalla root (`/`), leggendo il suo inode. L'inode della root è un valore fisso (tipicamente 2 nei sistemi UNIX).
   2. Legge i blocchi dati della root per trovare l'inode di `foo`.
   3. Ripete il processo per `foo`, leggendo il suo inode e i suoi blocchi dati per individuare `bar`.
   4. Una volta trovato, legge l'inode di `bar`, verifica i permessi e assegna un file descriptor al processo.

2. Lettura del File (`read()`)
   - Il primo read() legge il primo blocco del file, consultando l'inode per trovare la sua posizione su disco.
   - Aggiorna il tempo di ultimo accesso nell'inode.
   - Il puntatore del file viene aggiornato per consentire le letture successive.

3. Chiusura del File (`close()`)

- Il file descriptor viene deallocato, ma non sono necessarie operazioni su disco.

Considerazioni sulle prestazioni:

- Il numero di operazioni I/O dipende dalla lunghezza del percorso: più directory significa più inode e blocchi dati da leggere.
- Directory grandi peggiorano le prestazioni perché richiedono la lettura di più blocchi per trovare un file.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			**read**	**read**	**read**	**read**	**read**			
read()					**read** **write**			**read**		
read()					**read** **write**				**read**	
read()					**read** **write**					**read**

Figura 75: Read timeline.

5.7 Scrittura su file

Scrivere un file segue un processo simile alla lettura, con alcune differenze importanti: se il file non esiste, deve essere creato, e ogni scrittura può allocare nuovi blocchi su disco.

1. Apertura del File (`open()`)

- Se il file esiste, viene aperto come nella lettura.
- Se il file è nuovo, il file system:
  1. Trova e alloca un inode libero, aggiornando la bitmap degli inode.
  2. Inizializza il nuovo inode su disco.
  3. Aggiorna i dati della directory per collegare il nome del file al suo inode.
  4. Aggiorna l'inode della directory.

2. Scrittura del File (`write()`). Ogni scrittura può comportare:
   1. Lettura della bitmap dei blocchi per trovare un blocco libero.
   2. Scrittura della bitmap aggiornata.
   3. Lettura dell'inode per aggiornare i puntatori ai blocchi.
   4. Scrittura dell'inode aggiornato.
   5. Scrittura del blocco dati effettivo.

3. Chiusura del File (`close()`)

- Il file descriptor viene deallocato, senza operazioni su disco.

Efficienza della Scrittura:
- Creare un file comporta molte operazioni su disco, rendendolo un'operazione costosa.
- Allocare nuovi blocchi peggiora le prestazioni perché richiede aggiornamenti multipli alle bitmap e agli inode.
- Sistemi avanzati ottimizzano la scrittura con strategie come buffering e journaling per ridurre il numero di scritture effettive su disco.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		**read** **write**	**read**	**read**	**read** **write**	**read**	**read**	**write**		
write()	**read** **write**				**read** **write**			**write**		
write()	**read** **write**				**read** **write**				**write**	
write()	**read** **write**				**read** **write**					**write**

Figura 76: File creation timeline.

5.8 Caching e buffering

Le operazioni di I/O su file sono costose in termini di tempo, poiché la lettura e la scrittura su disco richiedono numerosi accessi ai dati. Senza **caching**, ogni apertura di file comporterebbe molte operazioni di lettura, soprattutto con percorsi lunghi.

I file system hanno introdotto una cache di dimensione fissa (**fixed-size cache**) per memorizzare blocchi frequentemente utilizzati. Strategie come LRU (Least Recently Used) vengono impiegate per determinare quali blocchi mantenere in cache. Inizialmente, la cache veniva allocata staticamente al boot per circa il 10% della memoria totale. Questo metodo presentava però un problema, poiché l'allocazione fissa poteva risultare inefficiente se la memoria non veniva utilizzata appieno.

I sistemi moderni adottano un approccio più flessibile con la page cache unificata, che gestisce dinamicamente la memoria tra cache del file system e memoria virtuale (**partizionamento dinamico**). Questo sistema permette di assegnare la memoria in base alle necessità attuali, evitando sprechi. Quando un file viene aperto per la prima volta, si genera traffico I/O, ma le aperture successive possono sfruttare la cache e ridurre le operazioni di lettura.

La cache ha un impatto significativo anche sulle operazioni di scrittura. Mentre le letture possono essere evitate se i dati sono già in cache, le scritture devono comunque essere registrate su disco per garantire la persistenza. Il buffering delle scritture consente di raggruppare aggiornamenti, riducendo il numero di operazioni di I/O. Inoltre, permette di ottimizzare la pianificazione delle scritture e in alcuni casi di evitarle del tutto, come quando un file viene creato e poi eliminato prima di essere effettivamente scritto su disco.

I file system moderni ritardano le scritture tra cinque e trenta secondi per ottimizzare le prestazioni. Questo introduce un compromesso tra performance e sicurezza dei dati, poiché in caso di crash del sistema prima della scrittura definitiva su disco, i dati presenti in memoria vengono persi. Alcune applicazioni, come i database, non possono accettare questo rischio e adottano strategie alternative, come la chiamata a `fsync()` per forzare la scrittura su disco, l'uso di I/O diretto per bypassare la cache o l'accesso diretto al disco senza passare dal file system.

La gestione del caching e del buffering nei file system bilancia le prestazioni con l'affidabilità, offrendo opzioni per adattarsi alle esigenze specifiche delle applicazioni.

6 Fast FIle System (FFS)

Il primo file system di UNIX, era molto semplice e organizzava i dati su disco in tre sezioni principali:
- Super block, inode region e data blocks. Tuttavia, nonostante la sua semplicità e facilità d'uso, soffriva di gravi problemi di prestazioni.

6.1 Problema: Scarse Prestazioni

- Il file system trattava il disco come una memoria ad accesso casuale, ignorando i costi di posizionamento fisico.
- I blocchi di dati di un file erano spesso lontani dal suo inode, causando costosi spostamenti della testina del disco.
- La gestione della lista dei blocchi liberi portava a frammentazione, con file distribuiti in diverse aree del disco invece che in blocchi contigui.
- Il blocco di dimensioni ridotte (512 byte) aumentava i costi di trasferimento dei dati.

6.2 Soluzione: disk aware

Il Fast File System (FFS) è stato progettato per essere **disk aware** (consapevole della struttura fisica sottostante del disco su cui opera e ottimizza le sue operazioni di conseguenza), migliorando così le prestazioni rispetto al precedente file system UNIX. Pur mantenendo la stessa interfaccia e API (`open()`, `read()`, `write()`, `close()`), FFS ha introdotto nuove strutture e politiche di allocazione più efficienti. Questo approccio ha aperto la strada alla ricerca sui file system moderni, che continuano a ottimizzare gli interni per migliorare prestazioni, affidabilità e compatibilità con le applicazioni esistenti.

6.3 Struttura con i cylinder group

FFS migliora l'organizzazione dei dati su disco suddividendolo in **cylinder groups**, ciascuno composto da N cilindri consecutivi.

Cilindro: insieme di tracce alla stessa distanza dal centro del disco su più superfici, e raggruppare più cilindri riduce la frammentazione e migliora le prestazioni minimizzando gli spostamenti della testina.

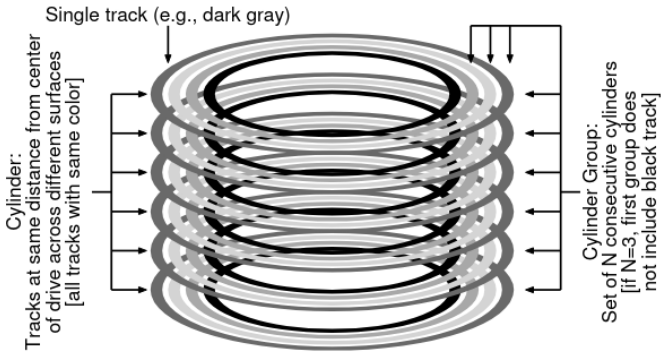


Figura 77: Struttura del cilindro.

Nei file system moderni, come ext2, ext3 ed ext4, questa struttura è implementata come **block groups**, poiché i dischi moderni astraggono la geometria fisica esponendo solo uno spazio logico di blocchi. Tuttavia, il principio rimane lo stesso: collocare file correlati nello stesso gruppo per ridurre i tempi di seek.

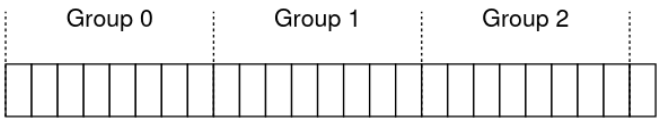


Figura 78: Struttura di block groups.

Ogni cylinder group contiene:
- **Superblock (S)**: una copia del superblocco per ridondanza, utile in caso di corruzione.
- **Bitmap degli inode (ib)** e dei **dati (db)**: utilizzate per tracciare quali inode e blocchi dati sono allocati o liberi. L'uso delle bitmap facilita l'allocazione di blocchi contigui, riducendo la frammentazione.
- **Regione degli inode** e **blocchi dati**: simile alla struttura del vecchio file system UNIX, ma organizzata per ottimizzare l'accesso ai file.

Grazie a questa organizzazione, FFS migliora significativamente le prestazioni rispetto al vecchio sistema, ottimizzando sia l'allocazione che l'accesso ai dati.

6.4 Creazione di un file con FFS

Quando un nuovo file viene creato (es. `/foo/bar.txt` di 4KB), diverse strutture dati devono essere aggiornate su disco:

1. Allocazione dell'inode:
   - Viene assegnato un nuovo inode, che deve essere scritto su disco.
   - La bitmap degli inode viene aggiornata per segnare l'inode come allocato.

2. Allocazione del blocco dati:
   - Il file ha contenuto, quindi è necessario un blocco dati.
   - La bitmap dei dati viene aggiornata per indicare l'allocazione.

3. Aggiornamento della directory:
   - La directory `foo` deve essere aggiornata per includere bar.txt.
   - Questo aggiornamento può avvenire in un blocco dati esistente o richiederne uno nuovo.
   - L'inode della directory viene modificato per aggiornare la dimensione e i metadati (es. last-modified-time).

6.5 Politiche di allocazione in FFS

FFS segue il principio «mantieni vicine le cose correlate e distanti quelle non correlate» per migliorare le prestazioni, riducendo i seek del disco. Per farlo, utilizza alcune euristiche di posizionamento per file e directory.

Allocazione delle Directory:
- FFS cerca un cylinder group con poche directory allocate e molti inode liberi per posizionare una nuova directory.
- Questo bilancia la distribuzione delle directory e garantisce spazio per i file futuri.

Allocazione dei File:
1. I blocchi dati di un file vengono posizionati nello stesso gruppo del suo inode, evitando seek lunghi.
2. I file nella stessa directory vengono collocati nello stesso cylinder group per preservare la località dei nomi.

Per esempio, se l'utente crea i file /a/c, /a/d, /a/e e /b/f, FFS organizza i file così:
- Gruppo 1 → Contiene /a, /a/c, /a/d, /a/e (vicini tra loro).
- Gruppo 2 → Contiene /b e /b/f (vicini tra loro).

```txt
group inodes     data
0     /--------- /---------
1     acde------ accddee---
2     bf-------- bff-------
3     ---------- ----------
4     ---------- ----------
5     ---------- ----------
6     ---------- ----------
7     ---------- ----------
```

Un'allocazione che distribuisce uniformemente gli inode tra i gruppi, senza considerare la località, porta a file della stessa directory sparsi su più gruppi. Questo peggiora le prestazioni, poiché i file correlati sono lontani e richiedono più seek per essere letti insieme.

```txt
group inodes     data
0     /--------- /---------
1     a--------- a---------
2     b--------- b---------
3     c--------- cc--------
4     d--------- dd--------
```

```
7 5    e---------  ee--------
8 6    f---------  ff--------
9 7    ----------  ----------
```

Le euristiche di FFS non derivano da studi complessi ma da buon senso: i file nella stessa directory sono spesso usati insieme (es. compilazione di codice), quindi tenerli vicini riduce i tempi di accesso e migliora le prestazioni del file system.

6.6 Misurare la Località dei File

Per valutare se le euristiche di FFS abbiano senso, è stato analizzato l'accesso ai file utilizzando i tracciamenti SEER [K94], misurando la distanza tra accessi consecutivi nella gerarchia delle directory.

Metodo di misurazione:
- Se un file f viene aperto e poi riaperto subito dopo, la distanza è 0.
- Se si accede a due file nella stessa directory (es. dir/f → dir/g), la distanza è 1.
- Se si accede a file in sottodirectory diverse, la distanza è misurata in base al primo antenato comune.

L'analisi ha mostrato che:
- 7% degli accessi erano allo stesso file aperto in precedenza (distanza 0).
- 40% degli accessi erano a file nello stesso file o nella stessa directory (distanza 0 o 1).
- 25% degli accessi erano a file con distanza 2, tipico di strutture organizzate gerarchicamente (es. proj/src/foo.c seguito da proj/obj/foo.o).

Limiti di FFS: FFS ottimizza l'accesso ai file con distanza 0 o 1, ma non cattura bene la località a distanza 2, causando più seek. Questo accade quando gli utenti organizzano file in più livelli (es. sorgenti e oggetti separati in directory diverse).

Confronto con un accesso casuale:
- Un accesso casuale ai file mostra meno località, come previsto.
- Tuttavia, anche nel caso casuale c'è una certa località, poiché tutti i file condividono un antenato comune (es. la root del filesystem).

L'analisi dei tracciamenti conferma l'efficacia delle strategie di località di FFS, anche se potrebbe essere migliorata per strutture più profonde.

6.7 L'eccezione per i file grandi

La politica standard di FFS assegna file e metadati nello stesso gruppo per ridurre i seek. Tuttavia, per file molto grandi, questa strategia presenta un problema:
- Il file potrebbe riempire interamente un gruppo, impedendo ad altri file "correlati" di essere collocati lì.
- Questo riduce la località di accesso ai file futuri.

6.7.1 Soluzione FFS: Divisione in Chunk

Per evitare questo problema, FFS spezza i file grandi in blocchi e li distribuisce su gruppi diversi:
1. I primi blocchi (es. 12 blocchi, che corrispondono ai puntatori diretti nell'inode) vengono memorizzati nel primo gruppo.
2. Successivamente, i blocchi indiretti vengono spostati in gruppi diversi (scelti in base alla loro disponibilità).

6.7.2 Esempio senza l'eccezione per file grandi

Un file di 30 blocchi in un FFS con 10 inode e 40 blocchi dati per gruppo finirebbe per occupare quasi tutto un gruppo:

```
group inodes     data
  0   /a-------- /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a---------
  1   ---------- ---------- ---------- ---------- ----------
  2   ---------- ---------- ---------- ---------- ----------
```

Se ora vengono creati altri file nella root /, non c'è più spazio nel gruppo 0 per i loro dati, riducendo la località.

6.7.3 Esempio con l'eccezione per file grandi

Con la politica di suddivisione in chunk di 5 blocchi, il file viene distribuito su più gruppi:

```
group inodes     data
  0   /a-------- /aaaaa----- ---------- ---------- ----------
  1   ---------- aaaaa----- ---------- ---------- ----------
  2   ---------- aaaaa----- ---------- ---------- ----------
  3   ---------- aaaaa----- ---------- ---------- ----------
  4   ---------- aaaaa----- ---------- ---------- ----------
  5   ---------- aaaaa----- ---------- ---------- ----------
  6   ---------- ---------- ---------- ---------- ----------
```

Ora altri file nella root / possono essere salvati nel gruppo 0 senza problemi.

Distribuire un file grande su più gruppi aumenta i tempi di seek, specialmente negli accessi sequenziali. Tuttavia, questa scelta:
- Evita che un file riempia un gruppo intero.
- Permette di mantenere la località tra file correlati.

Per bilanciare seek vs. trasferimento, si usa il concetto di ammortizzazione:
- Se il chunk è troppo piccolo, si passa troppo tempo in seek.
- Se il chunk è grande, si massimizza il tempo di trasferimento riducendo il seek overhead.

Per esempio, con:
- Tempo medio di posizionamento: 10 ms
- Velocità di trasferimento: 40 MB/s

Se vogliamo spendere metà del tempo in seek e metà in trasferimento, dobbiamo trasferire almeno: $40 \frac{\text{Mb}}{s} \cdot 10 \text{ ms} = 409.6 \text{ Kb}$
Maggiore è la percentuale di utilizzo della banda desiderata, più grande deve essere il chunk:
- 50% della banda → 409.6 KB
- 90% della banda → 3.6 MB
- 99% della banda → 39.6 MB

6.7.4 Strategia di FFS

FFS non ha scelto chunk in base a questo calcolo, ma ha seguito una strategia più semplice:
- I primi 12 blocchi diretti rimangono nel gruppo dell'inode.
- Ogni blocco indiretto e i suoi puntatori vengono assegnati a gruppi diversi.
- Con blocchi di 4KB, ciò significa che ogni 4MB del file sono salvati in gruppi separati.

Evoluzione nel tempo:
- La velocità di trasferimento dei dischi migliora rapidamente.
- Il tempo di seek e la rotazione migliorano molto più lentamente.
- Ciò significa che nel tempo il costo del seek diventa relativamente più alto, rendendo necessaria l'ammortizzazione con chunk più grandi.

L'eccezione per i file grandi in FFS riduce il problema dell'allocazione di spazio nei gruppi, migliorando la località per altri file. Tuttavia, ha il costo di maggiori seek durante l'accesso sequenziale. La chiave per bilanciare queste due esigenze è scegliere la giusta dimensione del chunk, per minimizzare i tempi di seek e massimizzare l'efficienza del trasferimento dati.

6.8 Altri miglioramenti

FFS ha introdotto diverse innovazioni significative per migliorare l'efficienza e l'usabilità dei file system.

1. **Sub-blocchi per file piccoli**: FFS ha risolto il problema dell'inefficienza dei blocchi da 4KB per file di dimensioni ridotte (spesso intorno ai 2KB) utilizzando sub-blocchi da 512 byte. Questo ha ridotto la frammentazione interna. Quando un file piccolo cresceva, FFS consolidava i sub-blocchi in blocchi da 4KB.

2. **Layout ottimizzato per le prestazioni**: FFS ha risolto il problema del posizionamento consecutivo dei settori sul disco, «parametrizzando» il layout. Saltando blocchi tra loro, evitava i ritardi dovuti alla rotazione del disco, migliorando così le prestazioni di lettura sequenziale.

3. **Miglioramenti dell'usabilità**: FFS ha reso il sistema più facile da usare introducendo nomi di file lunghi, collegamenti simbolici (per riferimenti più flessibili a file e directory) e un'operazione di rinomina atomica per i file. Sebbene non siano innovazioni tecniche rivoluzionarie, queste migliorie hanno contribuito a una maggiore adozione e fruibilità di FFS.

Queste scelte progettuali hanno mostrato un equilibrio tra innovazione tecnica e miglioramenti pratici, rendendo FFS più efficiente e accessibile per gli utenti.

7 Journaling

Crash-consistency problem (problema di consistenza in caso di crash): si riferisce alla sfida di garantire che un sistema rimanga in uno stato coerente e valido dopo un guasto improvviso del sistema, come una perdita di energia o un crash del software. Supponiamo di voler scrivere in un file. Per farlo, dobbiamo sostanzialmente eseguire tre writes:
1. Aggiornare l'inode del file.
2. Aggiungere un nuovo blocco nella regione dati.
3. Aggiornare la data bitmap.

Se una di queste writes non viene completata per via di un crash, il file system viene lasciato in uno stato «inconsistente».

7.1 The Crash Consistency Problem

Immaginiamo che una sola write precedenti abbia successo. I possibili scenari sono i seguenti:

- **Solo il data-block scritto su disco**: il blocco esiste fisicamente, ma non è riferito da alcun inode né segnato come occupato nella bitmap. Non c'è problema di crash-consistency, ma l'utente potrebbe perdere dati.

- **Solo l'inode scritto su disco**: l'inode punta a blocchi non scritti, che contengono dati spazzatura. Questo causa inconsistenza del file system poiché l'inode indica dati validi mentre la bitmap li segna come liberi.

- **Solo la bitmap scritta su disco**: la bitmap segnala un blocco come occupato, ma non esiste alcun inode che lo punti. Questo causa inconsistenza e può portare a un space leak, dove il blocco non sarà mai utilizzato correttamente.

Supponiamo ora che due writes su tre abbiano avuto successo. Scenari:

Nel caso in cui due scritture su tre abbiano avuto successo, si verificano i seguenti scenari:

- **Inode e bitmap scritti su disco**: il blocco è marcato come «allocato» nella bitmap e l'inode contiene i puntatori al blocco, che però contiene «garbage data» perché i dati corretti non sono scritti. Non si crea un'inconsistenza del file system, ma l'utente perde dati, possibilmente sensibili.

- **Inode e blocco scritti su disco**: i data-blocks contengono i dati desiderati e l'inode punta correttamente a questi blocchi. Tuttavia, il blocco potrebbe essere sovrascritto, poiché è marcato come libero nella bitmap e quindi non sarà accessibile.

- **Bitmap e blocco scritti su disco**: il blocco è fisicamente presente e marcato come occupato nella bitmap, ma nessun inode lo punta. Di conseguenza, il blocco non sarà accessibile e non è chiaro a quale file appartenga.

7.1.1 Soluzione 1: FS check

Un approccio iniziale per risolvere il problema di crash consistency consisteva nel permettere che il sistema continui a funzionare durante l'errore, per poi correggere le incongruenze all'avvio. Questa operazione veniva generalmente affidata a uno strumento chiamato fsck.

fsck ha lo scopo di individuare e riparare le «inconsistenze» del file system, anche se non può risolvere tutte le problematiche (ad esempio, se un inode punta a «garbage data», non è possibile recuperare i dati corretti). Il tool viene eseguito prima che il file system venga montato e reso disponibile, assumendo che non ci siano operazioni in corso sul file system durante la sua esecuzione. Al termine, fsck restituisce un file system consistente e accessibile all'utente. Le fasi di esecuzione di fsck sono le seguenti:

1. **Superblock**: Verifica la correttezza del superblock, che contiene metadati cruciali sul file system. Durante questa fase viene eseguito un sanity check, che verifica, ad esempio, che la dimensione del file system sia maggiore del numero di blocchi allocati. Se il superblock è corrotto, viene sostituito con una copia funzionante.

2. **Free blocks**: Scansiona gli inodes, i blocchi indiretti, e così via, per generare una versione corretta della data bitmap basata sulle informazioni contenute negli inodes. Viene eseguita una scansione analoga per gli inodes e la loro bitmap.

3. **Inode state**: Ogni inode viene verificato per eventuali danni o anomalie. Se un inode è compromesso e non può essere facilmente riparato, viene «pulito» (clear) e la relativa bitmap aggiornata.

4. **Inode links**: fsck controlla il link count di ogni inode allocato, cioè il numero di directory che contengono un riferimento a quel file. Scansionando l'intero albero delle directory a partire dalla root, fsck confronta il proprio link count con quello presente nel file system e aggiorna quest'ultimo se necessario.

5. **Duplicates**: fsck verifica l'eventuale presenza di puntatori duplicati, come nel caso in cui due inode diversi puntano allo stesso blocco. Se un inode è corrotto, può essere «pulito» oppure, se necessario, il blocco viene duplicato, garantendo che ogni inode abbia la propria copia del blocco.

6. **Bad blocks**: Viene eseguito un controllo sui puntatori che potrebbero indirizzare a blocchi corrotti. Un puntatore è considerato corrotto se fa riferimento a un blocco al di fuori di un range valido (ad esempio, se punta a un blocco più grande della partizione). In tal caso, fsck rimuove semplicemente il puntatore, senza ulteriori operazioni.

7. **Directory checks**: fsck esegue una verifica sull'integrità delle directory. Si assicura che le voci «.» e «..» siano le prime entries e che ogni inode riferito a una voce di directory sia allocato correttamente. Inoltre, verifica che le directory non contengano cicli, garantendo che siano collegate correttamente tra loro.

Il principale problema di questo approccio è la sua lentezza. Più grande è la partizione, più tempo è necessario per eseguire fsck. Con l'introduzione dei dischi RAID, il processo è diventato ancora più complesso e rallentato. Sebbene fsck funzioni correttamente, è una soluzione che comporta un significativo spreco di tempo.

7.1.2 Soluzione 2: Journaling

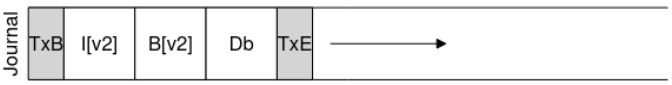
Una soluzione comune per affrontare la problematica di crash consistency è il write-ahead logging, noto nel contesto dei file system come **journaling o Write-Ahead Logging**. Consiste nell'anticipare l'aggiornamento delle strutture dati su disco, registrando prima un «log» (un registro) che descrive l'operazione imminente. Questo log viene scritto in una zona del disco di cui si conosce la posizione e viene chiamato appunto journal (da cui il termine write-ahead logging). In questo modo, se si verifica un crash durante l'aggiornamento delle strutture dati, il sistema può consultare il log per determinare esattamente cosa correggere e come farlo.

Con il journaling, dopo un crash, invece di eseguire una scansione completa del disco tramite fsck, è possibile andare direttamente alla sorgente del problema, riducendo significativamente il tempo necessario per il recupero. Sebbene l'introduzione di un log aggiunga un piccolo carico durante la fase di aggiornamento, essa riduce notevolmente i tempi di recupero dopo un crash.

Per comprendere come funziona, prendiamo ad esempio il file system Linux ext3. Il disco è suddiviso in groups-blocks, ciascuno dei quali contiene una bitmap per gli inodes, una bitmap per i dati e gli stessi inodes e blocchi di dati. Una struttura chiave in questo contesto è il blocco «journal», che occupa uno spazio ridotto all'interno della partizione (o su un altro dispositivo).

7.1.3 Esempio pratico:

Supponiamo di voler eseguire l'aggiornamento di un inode (I[v2]), una bitmap (B[v2]) e un data block (Db).



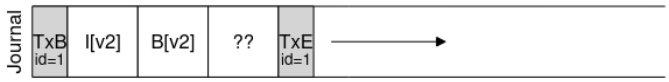
Prima di scrivere questi dati su disco, il sistema registra le informazioni necessarie nel log (journal), in particolare:
- **TxB**: «Transaction Begin», che contiene informazioni sull'aggiornamento imminente, come gli indirizzi finali di I[v2], B[v2] e Db, oltre a un identificatore della transazione (TID).
- **TxE**: «Transaction End», che segna la fine della transazione e include anch'esso l'TID.
- Tra **TxB** e **TxE**, i blocchi di dati contengono esattamente il contenuto degli inodes e della bitmap (questo è noto come physical logging). Esiste anche l'alternativa del logical logging, in cui vengono registrate rappresentazioni logiche dell'aggiornamento.

Una volta che il blocco journal è stato scritto su disco, si può procedere con l'aggiornamento delle strutture dati originali (questo processo è noto come checkpointing). Il file system scrive I[v2], B[v2] e Db nelle locazioni appropriate su disco.

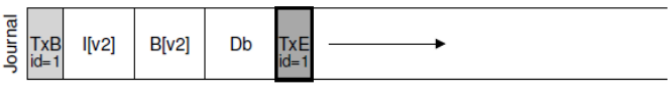
La sequenza delle operazioni è quindi la seguente:
1. **Journal Write**: Il sistema scrive nel log le informazioni necessarie per la transazione.
2. **Checkpoint**: Il sistema applica gli aggiornamenti alle strutture dati sul disco.

7.1.4 Gestione dei crash durante la scrittura

Cosa succede se un crash avviene durante la scrittura del blocco journal?



Il sistema sta cercando di scrivere i blocchi della transazione su disco (TxB, I[v2], B[v2], Db, TxE). Una soluzione semplice potrebbe essere quella di eseguire una scrittura per volta, attendendo che ogni scrittura sia completata prima di passare alla successiva. Sebbene questo approccio funzioni, è inefficiente. D'altro canto, scrivere tutti i blocchi in una sola volta non è sicuro, poiché il disco potrebbe ottimizzare l'I/O e scrivere i blocchi nell'ordine sparso (ad esempio, prima TxB, poi I[v2], B[v2], TxE, e infine Db). Se il crash avviene dopo che TxE è stato scritto ma prima di Db, il file system troverà una transazione «completa» (con un inizio e una fine), ma in realtà il sistema non può sapere che la transazione è incompleta. Al riavvio, il sistema potrebbe erroneamente copiare il contenuto del «garbage-block» nel blocco Db, con il rischio di perdita di dati.



Per evitare questo problema, il file system adotta una scrittura in due fasi. Prima vengono scritti nel journal tutti i blocchi tranne TxE (questi blocchi vengono scritti in una sola operazione). A questo punto, il file system può scrivere anche TxE nel journal, garantendo così l'atomicità dell'operazione. Il disco garantisce che le scritture da 512 byte (dimensione di un settore) vengano eseguite in modo atomico, quindi TxE deve essere di 512 byte per garantire il corretto funzionamento.

7.1.5 Sequenza di Aggiornamento del FS

Il protocollo di aggiornamento del file system, quindi, sarà:
1. **Journal Write**: Scrittura del contenuto della transazione nel journal, inclusi TxB, metadati e dati. Il sistema attende che queste scritture siano completate.
2. **Journal Commit**: Scrittura del «commit-block» (TxE) nel journal e attesa del completamento di questa scrittura.

3. **Checkpoint**: Applicazione degli aggiornamenti nelle locazioni appropriate su disco.

Con questa strategia, il file system è in grado di gestire i crash in modo più sicuro e ridurre il tempo necessario per il recupero, pur mantenendo l'affidabilità e la consistenza delle operazioni.

7.2 Recovery

Abbiamo precedentemente visto che il blocco journal (la «nota») può essere utilizzato per il recupero (recovery) dopo un crash. Un crash, o una perdita di corrente, può verificarsi in qualsiasi momento. Se il crash accade prima che la transazione sia completata (ovvero prima del «journal commit»), il processo di recupero è relativamente semplice: l'operazione in sospeso viene semplicemente ignorata. Se, invece, il crash si verifica dopo che la transazione ha scritto nel commit block ma prima che il checkpoint sia completato, il file system è in grado di riprendere l'aggiornamento.

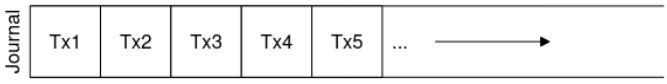
Durante l'avvio del sistema dopo un crash, il processo di recupero del file system esamina il log per identificare le transazioni che erano state «committed», ossia quelle che avevano tentato di scrivere su disco. Le transazioni vengono quindi eseguite nuovamente, nell'ordine in cui sono state registrate, per aggiornare le strutture dati appropriate. Questo processo, noto come **redo logging**, garantisce che il file system riporti le sue strutture su disco a uno stato consistente.

Nel caso in cui il crash avvenga durante il checkpoint, il contenuto del journal può essere letto durante il recovery e l'aggiornamento verrà ripetuto. Anche se il crash avviene all'ultima fase del processo, il sistema eseguirà tutte le scritture necessarie, considerando che i crash del file system sono eventi rari.

7.2.1 Performance del recovery

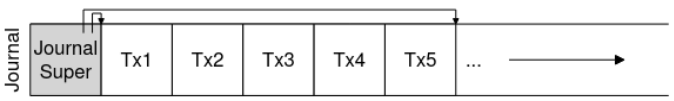
Il protocollo di journaling potrebbe generare un traffico I/O aggiuntivo. Ad esempio, supponiamo di creare due file, «file1» e «file2», nella stessa directory. Per creare ogni file è necessario aggiornare diverse strutture su disco: la inode bitmap (per allocare un nuovo inode), l'inode stesso, il blocco dati della directory contenente la nuova entry, e l'inode della directory (che ora ha un nuovo access time).

Nel caso del journaling, tutte queste informazioni vengono scritte nello stesso blocco journal per entrambe le operazioni di «file creation». Poiché i file si trovano nella stessa directory e gli inodes sono probabilmente nello stesso blocco su disco, potrebbe succedere che lo stesso blocco venga scritto più volte (ad esempio, scriviamo prima l'inode di «file1», poi quello di «file2», sovrascrivendo quello di «file1», e così via). Una soluzione a questo problema consiste nell'utilizzare un buffer globale nel quale vengono raccolte le transazioni. Tuttavia, se il buffer continua ad accumulare transazioni, potrebbe riempirsi rapidamente.



I problemi derivanti dall'accumulo di transazioni sono i seguenti:
- Più grande è il log, più tempo sarà necessario per eseguire il recovery.
- Quando il log si riempie o è quasi pieno, non sarà possibile registrare nuove transazioni, rendendo di fatto il file system inutilizzabile.

Per affrontare questi problemi, i file system con journaling trattano il log come una struttura dati circolare, riutilizzandola continuamente. Questo è il motivo per cui il journaling è noto come **circular log**. Quando una transazione è completata, il file system libera lo spazio relativo ad essa. Una soluzione semplice consiste nel marcare le transazioni vecchie e «non-checkpointed» all'interno di un superblock del journal.



Nel superblock del journal ci sono informazioni sufficienti per determinare quale blocco non ha ancora raggiunto la fase di checkpoint. Questo approccio riduce notevolmente il tempo di recupero, poiché non è necessario ripetere tutte le transazioni, ma solo quelle incompletate.

7.2.2 Schema del protocollo di recovery

Il nostro protocollo di journaling si evolve nel seguente modo:
1. **Journal Write**: Scrittura di TxB e dei tre blocchi intermedi.
2. **Journal Commit**: Scrittura del blocco TxE.
3. **Checkpoint**: Applicazione degli aggiornamenti alle strutture dati nel file system.
4. **Free**: Se la transazione è completata, viene marcata come liberata nel superblock.

7.2.3 Ottimizzazione del traffico I/O

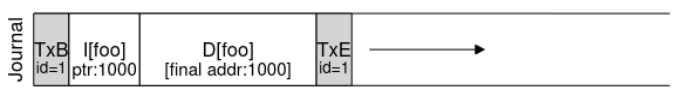
Le operazioni di scrittura su disco sono costose, e un aspetto problematico del journaling è che scriviamo due volte le stesse informazioni: una volta nel journal e una volta nella locazione finale su disco. Questo doppio processo aumenta il traffico di dati su disco, anche se il recovery è molto rapido. Una possibile soluzione per ridurre il traffico e ottimizzare il procedimento è il **data journaling** (come in ext3 di Linux), che registra tutti i dati utente nel journal, oltre ai metadati del file system.

Un'altra soluzione è il **ordered journaling** (o metadata journaling), in cui vengono registrati solo i metadati nel journal e non i dati utente (eliminando il blocco Db dal journal). Tuttavia, questo approccio introduce delle difficoltà: l'ordine in cui vengono scritte le informazioni su disco diventa cruciale. Se il blocco dei dati viene scritto dopo che la transazione è stata completata, i puntatori dell'inode potrebbero puntare a dati corrotti (ad esempio, le bitmap e gli inodes sono aggiornati, ma Db no). Inoltre, il recovery diventa più complesso, poiché Db non è più nel journal.

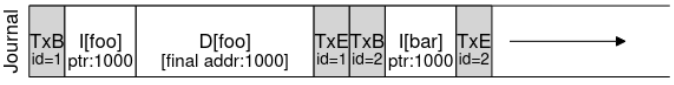
Un protocollo per risolvere queste problematiche potrebbe essere il seguente:
1. **Data Write**: I dati vengono scritti nella locazione finale.
2. **Journal Metadata Write**: Vengono scritti l'inizio del blocco e i metadati nel journal.
3. **Journal Commit**: Viene scritta la fine della transazione (TxE) nel journal.
4. **Checkpoint Metadata**: I metadati vengono scritti nella locazione finale nel file system.
5. **Free**: La transazione viene marcata come liberata nel superblock.

7.2.4 Block Reuse

Immaginiamo di usare una forma di metadata journaling. Supponiamo di avere una directory chiamata «foo» e che l'utente aggiunga una voce a questa directory (creando un nuovo file). Il contenuto della directory viene scritto nel log, poiché le directory sono considerate metadati. Immaginiamo ora che il blocco dati di «foo» abbia l'indirizzo «1000». In questo caso, il log conterrà informazioni come:



Supponiamo che l'utente elimini tutto dalla directory e successivamente la directory stessa, liberando il blocco «1000». Poi, l'utente crea un nuovo file chiamato «foolbar», che finisce per utilizzare lo stesso blocco «1000», che in precedenza apparteneva alla directory «foo». L'inode di «foolbar» viene scritto su disco, così come i suoi dati, ma solo l'inode viene scritto nel journal.



Se si verifica un crash e queste informazioni sono ancora nel log, si crea un grosso problema, poiché il blocco «1000» (contenente i dati di «foolbar») viene sovrascritto con i dati della directory «foo». Per evitare questo problema, il file system può utilizzare un tipo speciale di record nel journal chiamato revoke record. Quando una directory viene eliminata, come nell'esempio, viene scritto un revoke record nel journal. Durante il recovery, il sistema esaminerà prima il revoke record e non riscriverà i dati che sono stati «revocati».

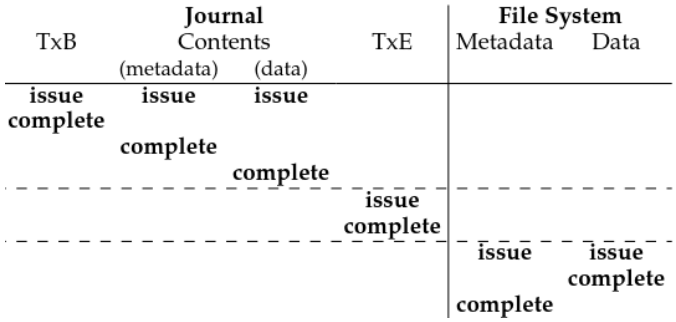


Figura 79: Data journaling timeline.

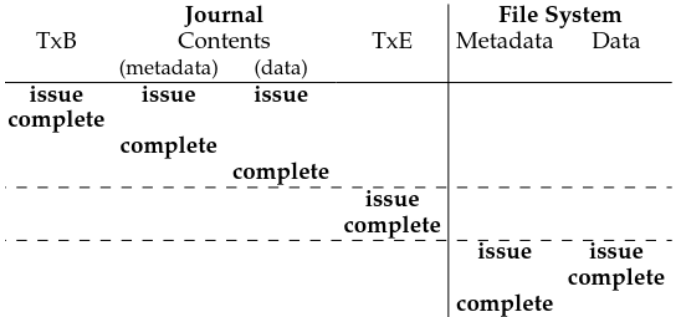


Figura 80: Metadata journaling timeline.

7.2.5 Approcci Alternativi

Esistono anche altri approcci al di fuori del journaling. Uno di questi è **Soft Updates**, che si basa sull'ordinamento accurato delle scritture al file system per garantire che le strutture su disco non vengano mai lasciate in uno stato inconsistente (ad esempio, scrivendo un blocco dati prima che l'inode punti ad esso). Un altro approccio è **Copy-on-Write** (COW), che implica la scrittura di nuove copie di un blocco di dati invece di modificarlo direttamente. Questi sono solo alcuni degli approcci adottati dai file system per garantire la consistenza e l'affidabilità.

Bibliografia

[1] R. H. Arpaci-Dusseau e A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.10 ed. Arpaci-Dusseau Books, 2023.

[2] «Process Control Block (PCB)». Consultato: 20 gennaio 2025. [Online]. Disponibile su: https://notes.yxy.ninja/OS/Process/Process-Control-Block-%28PCB%29

[3] V. Lavecchia, «Differenza tra paginazione e segmentazione in informatica | Informatica e Ingegneria Online». Consultato: 16 gennaio 2025. [Online]. Disponibile su: https://vitolavecchia.altervista.org/differenza-tra-paginazione-e-segmentazione-in-informatica/

[4] «Paging in operating system». [Online]. Disponibile su: https://www.geeksforgeeks.org/paging-in-operating-system/

[5] «Translation Lookaside Buffer (TLB) in Paging». [Online]. Disponibile su: https://www.geeksforgeeks.org/paging-in-operating-system/

[6] «Confronto: RAID 10 vs. RAID 50». Consultato: 19 gennaio 2025. [Online]. Disponibile su: https://recoverit.wondershare.it/windows-tips/raid-10-and-raid-50.html

[7] «File:RAID 01.svg - Wikipedia». Consultato: 19 gennaio 2025. [Online]. Disponibile su: https://en.m.wikipedia.org/wiki/File:RAID_01.svg