

Sistemi Operativi

Damiano Trovato - Lezioni del professore Mario Di Raimondo

Anno 2024/25, secondo semestre

Indice

1 Il sistema operativo	6
1.1 Il ruolo del sistema operativo	6
1.1.1 Modalità di esecuzione (User-Kernel)	6
1.2 Hardware	7
1.2.1 Il processore	7
1.2.2 Il ciclo di esecuzione dei processi	7
1.2.3 I processori (super)scalari	7
1.3 Modalità di esecuzione	8
1.3.1 Più processori	9
1.4 Threads	9
1.4.1 Multithreading	9
1.4.2 L'importanza del multithreading	9
1.5 Memorie	10
1.5.1 Gerarchia delle memorie	10
1.6 Strutture di sistemi operativi	11
2 Processi	12
2.1 Modello dei processi	12
2.1.1 Disambiguiamo processi e programmi	12
2.1.2 CPU virtuale, pseudoparallelismo	12
2.2 Creazione processi	13
2.2.1 Quando viene creato un processo?	13
2.2.2 Daemon	13
2.2.3 Creare processi nei vari sistemi operativi	14
2.2.4 Esempio nei sistemi UNIX	14
2.3 Terminazione di un processo	15
2.3.1 Soft Kill, Hard Kill	15
2.4 Ciclo di vita di un processo	16
2.4.1 Gli stati di un processo	16
2.4.2 Prelazione	17
2.4.3 Busy wait vs Blocking wait	17
2.5 Tabella dei processi	18
2.5.1 PCB - Process Control Block	18
2.5.2 Scheduler	18
2.6 Coda e accodamento	19
2.6.1 La coda dei processi pronti	19
2.6.2 Diagramma di accodamento	19
3 Thread	20
3.1 Introduzione al modello a thread	20
3.2 I thread	20
3.2.1 Piccolo prezzo da pagare: gestione dei thread	21
3.2.2 Scheduling dei thread	21
3.2.3 Cambiamenti di stato dei thread	21
3.2.4 Operazioni tipiche sui thread	21
3.3 Introduzione al multicore programming	22
3.3.1 Programmazione multicore	22
3.3.2 Sistemi single-core vs multi-core	22
3.3.3 Tipologie di thread	22
3.4 Thread a livello utente	23

3.4.1	Come fa la CPU a saltare tra thread?	23
3.4.2	Criticità sui thread a livello utente	23
3.4.3	I punti di forza	23
3.5	Thread a livello Kernel	24
3.5.1	Vantaggi di questo modello	24
3.5.2	Svantaggi	24
3.6	Modello ibrido	24
3.6.1	Come si procede nel modello ibrido?	24
3.7	I thread nei nostri sistemi operativi	24
3.7.1	pthreads	24
4	Interprocess Communication	25
4.1	Introduzione all'IPC	25
4.1.1	Modello pipe	25
4.1.2	Implicazioni del modello pipe	25
4.1.3	Criticità da gestire nell'interprocess communication	26
4.2	Race Condition	26
4.2.1	Sezioni critiche	26
4.2.2	Definire le sezioni critiche nel programma	26
4.2.3	Com'è fatta una buona soluzione per race condition?	27
4.3	Soluzione 1 - Inibire gli interrupt	27
4.3.1	Pro di questa soluzione	28
4.3.2	Contro di questa soluzione	28
4.4	Soluzione (non funzionante) 2 - Variabili di lock	28
4.5	Soluzione 2 - Alternanza Stretta	28
4.6	Soluzione 3 - La soluzione di Peterson	29
4.7	Soluzione 4 - Istruzioni TSL	30
4.7.1	Test and Set Lock	30
4.8	Soluzione 5 - Sleep & Wake Up	30
4.8.1	Problema dell'inversione delle priorità	30
4.8.2	Due nuove chiamate di sistema	30
4.9	Problema del produttore consumatore	31
4.9.1	Descrizione del problema	31
4.9.2	Una prima soluzione	31
4.9.3	Soluzione che potrebbe funzionare?	32
4.10	I semafori	32
4.10.1	Struttura del semaforo	32
4.11	Tipi di semafori	32
4.11.1	Soluzione al problema del produttore-consumatore basata su semafori	33
4.12	Futex - <i>Fast user space mutex</i>	33
4.13	Monitor	34
4.13.1	In cosa consistono	34
4.13.2	Semantiche signal	34
4.14	Problema dei 5 filosofi	35
4.14.1	Soluzione basata sui semafori	35
4.15	Problema dei lettori e scrittori	36
4.15.1	Soluzione basata sui semafori	36
4.16	Le altre soluzioni ai problemi giocattolo	36
4.16.1	Produttore-Consumatore in C	36
5	Scheduler	39
5.1	Introduzione allo scheduling	39
5.1.1	CPU burst	39
5.1.2	Tipologie di processi	39
5.2	Obiettivi degli algoritmi di scheduling	39
5.2.1	Obiettivi comuni.	39
5.2.2	Obiettivi nei sistemi batch	39
5.2.3	Obiettivi nei sistemi interattivi	39
5.2.4	Obiettivi nei sistemi real-time	40
5.3	Scheduling nei sistemi batch	40
5.3.1	Soluzione 1 - First-Come First-Served	40
5.3.2	Soluzione 2 - Shortest Job First	40

5.3.3	Soluzione 3 - Shortest Remaining Time Next	40
5.4	Scheduling nei sistemi interattivi	41
5.4.1	Soluzione 1 - Scheduling Round Robin	41
5.4.2	Soluzione 2 - Scheduling a priorità	41
5.4.3	2.1 - Variante: scheduling a code multiple	42
5.4.4	Soluzione 3 - Shortest Process Next	42
5.4.5	4 - Altre soluzioni note	43
5.5	Scheduling dei thread	43
5.5.1	Thread utente	43
5.5.2	Thread del Kernel	43
5.6	Scheduling su sistemi multiprocessore	44
5.6.1	Alcuni possibili approcci	44
5.6.2	Politiche di scheduling	44
5.7	E i nostri sistemi operativi?	44
5.7.1	Windows	44
5.7.2	MacOS	44
5.7.3	Linux	45
6	La memoria	46
6.1	Memoria centrale e processi	46
6.2	Nessuna astrazione	46
6.2.1	Alcune varianti senza astrazione	46
6.2.2	Multiprogrammazione senza astrazione, perché è impossibile	47
6.3	La prima astrazione - Spazio degli indirizzi	48
6.3.1	Multiprogrammazione con spazio degli indirizzi - Swapping	49
6.3.2	Problemi dello swapping	50
6.4	Soluzione definitiva alla frammentazione - Memoria Virtuale	52
6.4.1	Spazio di indirizzamento	52
6.4.2	Frammentazione (esterna) is no more	52
6.5	Paginazione	54
6.5.1	MMU	54
6.5.2	Tabella delle pagine	54
6.5.3	Dettagli sui record delle tabelle delle pagine	55
6.5.4	Tabella dei frame	55
6.6	Progettazione di una tabella delle pagine - velocità	56
6.6.1	Memoria associativa - TLB	56
6.6.2	Contenuto di un record TLB	56
6.6.3	Conseguenze	56
6.6.4	Osservazioni importanti	57
6.6.5	Flush e address-space id	57
6.6.6	Effective Access Time (EAT)	57
6.6.7	Gestione TLB	57
6.7	Progettazione di una tabella delle pagine - dimensioni	58
6.7.1	Soluzione -Tabella delle pagine multilivello	58
6.7.2	Tabella delle pagine invertita	59
6.7.3	Cache della memoria vs Memoria virtuale	59
6.8	Algoritmi di sostituzione delle pagine	60
6.8.1	Soluzione teorica - OPT	60
6.8.2	Algoritmo NRU - Not Recently Used	60
6.8.3	Algoritmo FIFO - First-In First-Out	61
6.8.4	Algoritmo della Seconda Chance	61
6.8.5	Algoritmo Clock - dell'orologio	61
6.8.6	Algoritmo LRU - Least Recently Used	62
6.8.7	Algoritmo NFU - Not Frequently Used	62
6.8.8	Algoritmo di Aging - Invecchiamento	63
6.9	Confronto nelle prestazioni degli algoritmi di sostituzione delle pagine	64
6.9.1	Valutazione algoritmo OPT	64
6.9.2	Valutazione algoritmo FIFO	65
6.9.3	Valutazione algoritmo LRU	66
6.9.4	Osservazioni su altri algoritmi visti	66
6.9.5	Riepilogo sugli algoritmi	66

6.10	Allocazione dei frame	67
6.10.1	Numero minimo e massimo di frame allocati	67
6.10.2	Scegliere le pagine - Pure Demand Paging	67
6.10.3	Strategie di allocazione dei frame	67
6.11	Strategie di valutazione delle pagine da rimuovere	68
6.11.1	Principio di località e concetto di località	68
6.11.2	Working set	69
6.11.3	Page Fault Frequency	69
6.11.4	Relazione Page Fault Frequency - Working Set	70
6.12	Politica di pulitura	71
6.12.1	Quanto costa gestire un page fault?	71
6.12.2	Paaging demon	71
6.13	Dimensione della pagina	71
6.14	Pagine condivise	72
6.14.1	Scenario 1: più istanze dello stesso programma	72
6.14.2	Pagine condivise come mezzo di IPC	72
6.14.3	Copy-on-Write - Ottimizzazione con memoria condivisa	73
6.14.4	Zero-fill-on-Demand	73
6.14.5	Ottimizzazione con Read-Only Static Zero Page	73
6.14.6	File mappati	73
6.15	Allocazione della memoria per il Kernel	74
6.15.1	Moduli Kernel come comuni processi	74
6.15.2	Slab Allocation	74
6.15.3	Vantaggi della slab allocation	74
7	File System	75
7.1	Introduzione al File System	75
7.1.1	Il File system	75
7.2	Struttura di un File system	77
7.2.1	Master Boot Record	77
7.2.2	GUID Partition Table	77
7.3	Implementazione dei file	78
7.3.1	Strategie principali	78
7.4	Implementazione delle directory	80
7.4.1	Dove vengono inseriti i metadati e attributi?	80
7.5	Condivisione di un File System tra utenti	81
7.5.1	Alcune idee implementative	81
7.6	Gestione dei blocchi liberi	82
7.6.1	Alcune strategie	82
7.7	Controlli di consistenza	82
7.7.1	Flag	82
7.7.2	Controlli di consistenza	82
7.7.3	Journaling	83
7.8	Cache del disco	84
7.8.1	Struttura basata su tabelle hash.	84
7.8.2	Free-behind e Read-ahead	84
7.9	Che file system usano i nostri sistemi operativi?	85
7.9.1	Windows	85
7.9.2	Linux	85
7.9.3	MacOS	85
7.10	Scheduling del disco	85
7.10.1	Ottimizzare il Seek-time	86
7.10.2	Esempio di esercizio	88
7.11	Parallelismo dei dischi (Sistemi RAID)	89
7.11.1	Striping	89
7.12	RAID	90
7.12.1	RAID 0 - <i>striping</i>	90
7.12.2	RAID 1 - <i>mirroring</i>	90
7.12.3	RAID 2 - <i>striping a livello di bit con ECC</i>	91
7.12.4	RAID 3 - <i>striping a livello di bit con bit di parità</i>	91
7.12.5	RAID 4 - <i>striping a livello di bit con XOR sull'ultimo disco)</i>	91

7.12.6 RAID 5 - <i>striping a livello di bit con informazioni di parità distribuite</i>)	92
7.12.7 Recap sui vari RAID	92
7.13 Solid State Disk - SSD	93
7.13.1 Prestazioni	93
7.13.2 Memorie Flash e File System	93
7.13.3 Garbage Collection e TRIM	93

Capitolo 1

Il sistema operativo

1.1 Il ruolo del sistema operativo

Il sistema operativo è un software che ha accesso alle risorse hardware del computer (CPU, RAM, dischi e periferiche), e offre all'utente delle astrazioni semplici per facilitarne la gestione e l'uso. Permette all'utente medio di non dover avere a che fare con i dettagli di basso livello, decisamente più complessi e delicati. **Rende bello e semplice ciò che dietro le quinte è brutto e complesso.**

Tra le astrazioni più note, due casi ben noti sono:

- **Driver dei dispositivi di I/O.**

I driver offrono interfacce semplici per interagire coi dispositivi di input/output. Fornendo delle primitive standard, i driver permettono agli sviluppatori e ai programmi di non dover conoscere troppi dettagli di basso livello relativi ai dispositivi. Sono quindi alla base della compatibilità tra periferiche e calcolatori.

- **File.**

L'astrazione più importante dell'intero sistema operativo: mascherano la complessità dei dati situati all'interno di varie porzioni di memoria, sotto forma **file**. Ciascuno di questi file, ha un'estensione che detta come questo vada interpretato, trattato e gestito.

1.1.1 Modalità di esecuzione (User-Kernel)

Il sistema operativo agisce in due modalità:

- **Modalità utente.**

È la modalità con cui vengono eseguiti tutti i software sopra il sistema operativo. Questa modalità offre privilegi limitati al programma, il quale potrà accedere esclusivamente alla memoria ad esso dedicata, e non permette l'accesso a risorse hardware. A tale scopo, tramite delle chiamate di sistema, un programma può chiedere l'intervento dell'sistema operativo, che eseguirà istruzioni in modalità Kernel. Un crash in modalità utente non mette a rischio il resto del sistema.

- **Modalità Kernel.**

O modalità supervisor. I programmi che girano in questa modalità (quali OS, driver e ulteriori software molto specifici), possono accedere alle risorse hardware senza particolari vincoli. È meno restrittiva, ma *da grandi poteri derivano grandi responsabilità*: un crash in modalità Kernel può far fallire l'intero sistema.

1.2 Hardware

L'hardware è ciò su cui opera il sistema operativo: è quindi fondamentale conoscere la struttura dell'hardware, per studiare i sistemi operativi.

- **CPU**
- **Disk Controller**
- **USB Controller**
- **Graphics Adapter**
- Dei **BUS** che interconnettono le parti precedentemente elencate

1.2.1 Il processore

È l'unità di elaborazione centrale (*Central Processing Unit*). Presenta un insieme di registri che possono contenere dei dati.

I registri speciali

Tra questi, alcuni registri particolari sono il PC (*Program counter*), l'SP (*Stack pointer*) e il PSW (*Program Status Word*).

- **Program Counter.**
Il PC contiene l'indirizzo alla prossima istruzione da elaborare.
- **Stack pointer.**
Lo SP contiene delle informazioni necessarie da mantenere per l'esecuzione di un programma (di istruzioni non ancora elaborate).
- **Program Status Word.**
Il PSW contiene informazioni relative a istruzioni di controllo, flag condizionali, modalità di esecuzione della CPU

Il sistema operativo deve conoscere bene questi registri, in quanto il parallelismo tra processi ne richiede l'utilizzo. Basti pensare che lo stack pointer tiene un riferimento allo stack delle variabili statiche di ciascun processo e che il program counter fa riferimento alla prossima istruzione da eseguire.

Il parallelismo

L'esecuzione parallela di più programmi diventa possibile proprio con l'ausilio di questi registri. Tuttavia, dedicando un quanto di tempo ad ogni processo, c'è un influente (seppur breve) lasso di tempo impiegato per gestire il passaggio dall'esecuzione del processo n , a quello $n + 1$.

1.2.2 Il ciclo di esecuzione dei processi

1. Fetch (nella Fetch Unit)
2. Decode (nella Decode Unit)
3. Execute (nella Execute Unit)

I calcolatori moderni permettono di pseudo-parallelizzare l'esecuzione di fasi differenti del ciclo di esecuzione di più processi, avendo parti di hardware dedicate a ciascuna di queste fasi.

In questo modo, ciò che si ottiene è un piccolo (ma importante) guadagno in termini di velocità di esecuzione. Un guadagno di qualche millisecondo diventa molto più importante di fronte a un grande numero di istruzioni.

1.2.3 I processori (super)scalari

Sono costituiti da molteplici unità di fetch, decode ed execute (di cui molte di loro specializzate). Avere molteplici unità, permette di effettuare contemporaneamente la stessa fase di più processi. È richiesta poi la presenza di un holding buffer usata per stabilire l'ordine in cui eseguire le istruzioni.

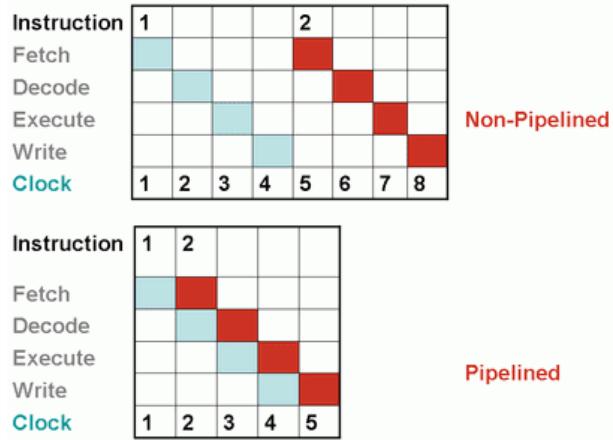


Immagine 1.1: Esempio di Pipelining

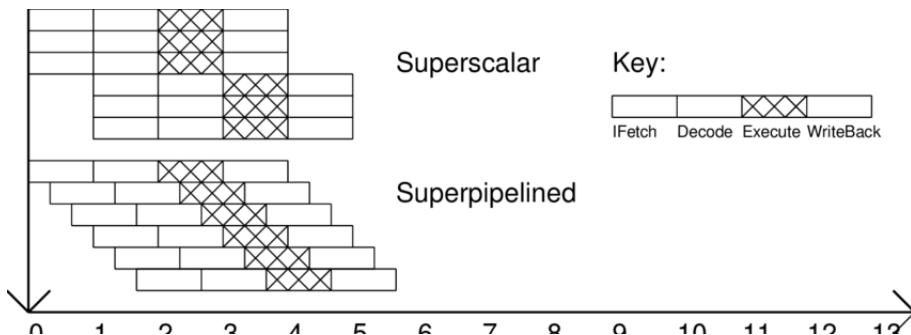


Immagine 1.2: Pipelining e Superscaling a confronto

1.3 Modalità di esecuzione

Abbiamo precedentemente individuato due modalità di esecuzione delle istruzioni, ovvero.

- **Modalità utente.**
- **Modalità Kernel.**

Nella maggior parte dei sistemi operativi, solo una piccola parte delle istruzioni si effettuano in modalità Kernel. La modalità utente garantisce un uso più sicuro del sistema. Inoltre, ogni chiamata a sistema impiega un determinato lasso di tempo.

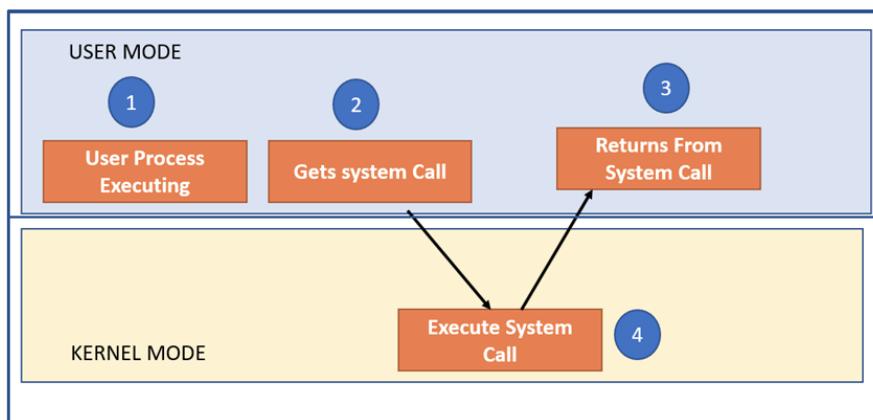


Immagine 1.3: Schema generale di chiamata a sistema

Il passaggio dalla user mode al Kernel mode è effettuato tramite la **trap**. Da user mode a Kernel mode è detta **return**. Premi qui per una pagina di geeksforgeeks su Kernel mode e User mode. Tutti gli interrupt sono eseguiti in Kernel mode.

1.3.1 Più processori

Avere più processori (o più core) ci permette di distribuire i programmi su più core, facendoli lavorare tutti a pieno regime e ottenendo prestazioni migliori. Tuttavia, il sistema operativo dovrà anche tenere conto di quale processo dovrà essere affidato a quale processore (in funzione dei dati di quel processo già presenti nella cache). Le caratteristiche dei multiprocessori sono:

- **Alto throughput.**
Massimizza il numero di processi in esecuzione in un determinato lasso di tempo.
- **Economia in scala.**
- **Affidabilità.**

1.4 Threads

Un thread è una suddivisione in sottoprocessi dello stesso programma che agiscono in maniera "parallela". Ogni thread potrebbe essere definito come un flusso, ma parte dello stesso processo. Il sistema vede ogni thread come un processore.

1.4.1 Multithreading

Col termine **multithreading**, indichiamo la possibilità da parte di un processore di gestire ed eseguire più thread. In ogni istante di tempo è eseguito un solo thread, per questo lo definiremo come uno "pseudo-parallelismo".

1.4.2 L'importanza del multithreading

Il multithreading è alla base dell'esecuzione pseudo-parallela di più processi. I vari thread condividono risorse (dello stesso programma), ma con una gestione ottima (tramite lo scheduler), i processi non competono, ma collaborano, non causando problemi.

1.5 Memorie

All'interno di un calcolatore è presente una gerarchia di memorie. Una buona organizzazione delle memorie permette di ottenere un sistema di memorie apparentemente unitario.

1.5.1 Gerarchia delle memorie

Alla cima della piramide troviamo le memorie più veloci, meno capienti e più costose.

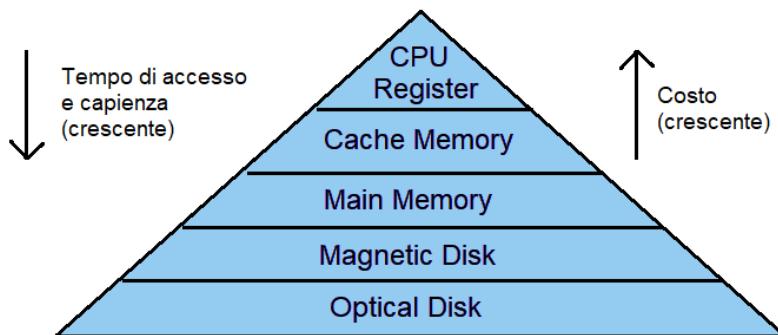


Immagine 1.4: Gerarchia delle memorie

- **Registro.**

Contiene un dato su cui si sta effettuando l'operazione corrente.

- **Cache.**

Esistono due tipi di cache. La cache L1 è interna al Core, L2 può essere interna al core, o condivisa tra più core. Contiene dati che sono stati utilizzati recentemente da processi.

La gestione della L2 unica è più difficile, ma è più probabile che un core trovi il dato richiesto da un processo.

- **Memoria Principale.**

Contengono tutte le porzioni di un programma che servono in un determinato istante di tempo, il tempo di accesso in una casella di memoria è diretto (RAM). È volatile, non permanente. Una parte di RAM tuttavia non è volatile, ed è di sola lettura. È la memoria ROM, non è accessibile dall'utente, e contiene tutte le istruzioni relative all'avvio del dispositivo.

- **Disco elettronico.**

Le memorie flash, sono memorie intermedie alla RAM e all'Hard Disk. Sono memorie permanenti e veloci, ma non sono particolarmente capienti. Sono più facilmente soggetti a malfunzionamenti.

- **Disco magnetico e disco ottico.**

Più economico e capiente. È l'hard disk. È costituito da dischi e testine di lettura e scrittura. L'accesso è sequenziale. Ogni disco è suddiviso in tracce e settori. Il tempo di ricerca di un determinato elemento in una posizione casuale del disco magnetico stimata è di 8-10 millisecondi.

- **Memoria virtuale.**

È un meccanismo che, tramite una Memory Management Unit, permette di mappare indirizzi virtuali a indirizzi fisici. La memoria virtuale è presumibilmente più grande della memoria fisica effettiva. Ci permetterà di eseguire tranquillamente programmi parzialmente nella RAM, parzialmente nell'hard disk.

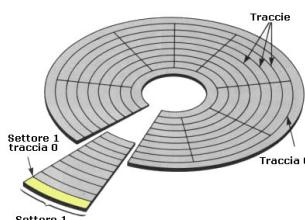


Immagine 1.5: Struttura di un Hard Disk

1.6 Strutture di sistemi operativi

Alcune possibili strutture note per un sistema operativo, sono le seguenti

- **Monolitico.**

Struttura molto comune in cui il sistema operativo, nella sua interezza, è eseguito in modalità Kernel. L'intero sistema è compilato in procedure collegate all'interno di un unico programma. Il fallimento di una singola procedura porta alto rischio di fallimento dell'OS. Il Kernel di Linux è monolitico.

- **A livello (o strati).**

Un'organizzazione gerarchica, in cui ogni livello implementa funzioni impiegando quelle fornite dal livello inferiore. Potrebbe avere problemi di performance a causa delle numerose chiamate implicate dal sistema a livelli.

Tuttavia, semplifica la programmazione, in quanto ogni livello si dovrà interfacciare solo con il livello inferiore e superiore. Principio di singola responsabilità.

- **MicroKernel.**

Si minimizzano le operazioni del Kernel, che nel contesto di questi sistemi chiameremo "MicroKernel.". Ottiene maggiore stabilità e minor rischio di fallimento del sistema, delegando il possibile al livello utente. Il microKernel si occuperà quindi di gestire IPC, scheduling dei processi e interrupt di sistema. Questi sistemi operativi suddividono il livello utente in tre sotto-livelli.

1. Programmi utente.

2. Server.

Si occupa di gestire il file system, gestiscono e distruggono processi. I programmi a livello utente, mandano dei messaggi al livello server, che procederà con una chiamata di sistema POSIX, interfacciandosi ai driver.

3. Driver.

I driver saranno a livello utente: ciò riduce il rischio di fallimenti dell'intero sistema causati da scritture errate in memoria da parte dei driver. Per comunicare con dispositivi I/O, i driver dovranno effettuare delle chiamate di sistema al sistema operativo.

Queste restrizioni danno responsabilità singole ad ogni singolo componente di questa struttura, seguendo il **Principle of Least Authority** (POLA).

- **Struttura a moduli.**

Simile al Kernel monolitico, ma con la possibilità di caricare moduli in maniera dinamica, che implementano aspetti specifici (file system, driver, etc.).

Il Kernel principale è a funzionalità ridotte, e ogni modulo può invocare altri moduli.

Attenzione: i moduli non comunicano tramite messaggi (come nel sistema microKernel), ma tramite interfacce ben note.

- **Macchine virtuali.**

La virtualizzazione viene gestita dagli **hypervisor**. Esistono due tipi di Hypervisor:

1. **Tipo 1.**

Girano direttamente sull'hardware della macchina, ma uno dei sistemi operativi fa da host, monitorando gli altri. Microsoft Hyper-V e VMware sono due esempi.

2. **Tipo 2.**

L'OS virtualizzato è un processo di un OS host. VirtualBox è un esempio.

I primi hanno dei vantaggi in termini di prestazioni e in termini di sicurezza. Nel tipo 2 aumenta il rischio di *virtual machine escape*.

Capitolo 2

Processi

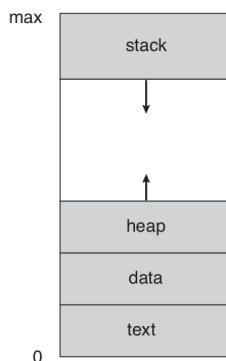
I computer moderni gestiscono un'alta quantità di processi, che questo sia noto all'utente o meno. Avere più processi, che operano in maniera pseudoparallela e/o parallela (nel caso di più processori) permette di offrire un'esperienza utente migliore nei sistemi operativi interattivi, e performance migliori nei sistemi batch e real-time.

Il **modello dei processi** offre un modello concettuale semplice per descrivere e gestire il problema del parallelismo.

2.1 Modello dei processi

Definiamo un **processo** come un'**istanza** di **esecuzione** di un programma. Un processo in memoria, presenta i seguenti elementi:

- Sezione di **testo** contenente il codice eseguibile.
- Sezione **dati**, contenenti **variabili globali e statiche**.
- **Heap** e **Stack**, rispettivamente per la memoria allocata dinamicamente, e per quella relativa alle chiamate a funzione.



Inoltre, ad ogni processo è associato un **Process Control Block**, ovvero un record all'interno di una **Tabella dei Processi** gestita dal Sistema Operativo, di cui parleremo dopo.

2.1.1 Disambiguiamo processi e programmi

Un programma, è del codice presente in memoria. Un processo, è un'istanza in esecuzione di un programma. Due istanze dello stesso programma sono due processi distinti.

2.1.2 CPU virtuale, pseudoparallelismo

Il modello a processi semplifica il ragionamento pensando nel seguente modo: ad ogni processo è garantita una personale **CPU** (chiaramente virtuale) e uno spazio di **memoria** (virtualmente) **personale** e (virtualmente) **infinito**. Ogni **processo** è (virtualmente) **sequenziale**.

Nel pratico, la CPU dedica un **quanto** di tempo al processo n , poi al processo $n + 1$, poi al processo $n + \dots$, secondo delle regole e dei termini che scopriremo durante il corso.

Ad ogni cambio di processo a cui la CPU si dedica, avviene un **cambio di contesto**: le istruzioni da eseguire dalla CPU e le risorse da usare vengono cambiate, in funzione dell'istanza del programma da eseguire, (concettualmente) passando da una CPU virtuale alla prossima.

Otteniamo quindi uno **pseudo-parallelismo**. In un sistema single-core lo pseudo-parallelismo è fondamentale per gestire un sistema complesso di multiprogrammazione. Ricordiamo che, anche quando apparentemente stiamo eseguendo un solo programma, i processi in background sono molteplici, e di fondamentale importanza.

2.2 Creazione processi

In questa sezione diamo un'occhiata alle circostanze che portano alla creazione di un nuovo processo, e al come questo avvenga (in termini di chiamate di sistema) nei vari sistemi operativi.

2.2.1 Quando viene creato un processo?

La **creazione** di un processo può avvenire in quattro principali circostanze.

- All'avvio del sistema, col **boot**.

Solitamente il processo iniziale ha Process Id-1. È la radice dell'albero dei processi. Gli ulteriori processi di avvio saranno inizializzati dal processo radice, o da altri processi, tutti appartenenti allo stesso albero.

- Da parte di un **processo padre**, tramite chiamate di sistema.

Nel caso dei multiprocessori, da dei vantaggi: delegare responsabilità differenti a più processi, permette di distribuire il carico di lavoro su più CPU, soffrendo anche di meno da chiamate bloccanti e circostanze simili.

- Per un'azione dell'**utente**.

All'interno dei sistemi interattivi.

- Inizio di un job in sistemi batch.

Avviene quando lo scheduler del sistema batch, secondo la propria politica di scheduling e le risorse disponibili del sistema, seleziona un job dalla coda di job da eseguire.

2.2.2 Daemon

In sistemi che supportano la multiprogrammazione, esistono processi detti **processi daemon**. Sono processi eseguiti in background che gestiscono attività di vario tipo, garantendo servizi all'utente. Non sono associati a utenti in particolari, ma hanno funzioni specifiche. Prendendo come esempio due processi, mail-client e mail-server:

- **Mail client.**

Offre un'interfaccia rivolta all'utente.

- **Mail server.**

Resta in ascolto su un socket, risvegliandosi all'arrivo di una e-mail, garantendo un servizio al processo client.

La maggior parte dei processi all'avvio sono processi daemon in background.

2.2.3 Creare processi nei vari sistemi operativi

Dipendentemente dal sistema operativo, le metodologie di creazione e di terminazione dei processi, vari-ano.

- **Sistemi UNIX.**

fork del processo padre, con successiva chiamata execve (o una chiamata analoga). La chiamata di sistema fork, da parte di un processo, crea una copia identica di quest'ultimo, con lo stesso spazio degli indirizzi, permessi e file aperti. Questo processo può usare la chiamata chiamata a sistema execve per cambiare la propria immagine di memoria ed eseguire un nuovo programma, diventando un processo diverso da quello da cui è stato forkato.

Il sistema UNIX può risalire al processo padre.

- **Alla Windows.**

È richiesta una singola chiamata di sistema, Win32(CreateProcess) con 10 parametri.

- Nome del programma da eseguire.
- Parametri della riga di comando da passargli.
- Vari attributi di sicurezza.
- Bit di controllo relativo a file aperti ereditati.
- Informazioni di priorità.
- Specifiche di creazione della finestra (del programma, se presente).
- Un puntatore alla struttura in cui viene restituita al chiamante l'informazione riguardo al processo appena creato.

È un'istruzione atomica che fornisce già un nuovo processo con una propria immagine di memoria.

2.2.4 Esempio nei sistemi UNIX

Supponiamo di avere una shell, da cui avviamo il programma sort.

Dietro le quinte, la shell creerà un nuovo processo, copia di se stessa, con gli stessi permessi e spazio degli indirizzi, tramite la syscall fork. Verrà poi chiamata la syscall execve (o simile) per modificare l'immagine di memoria di questo nuovo processo (figlio dello shell), utilizzando il programma sort come "blueprint" di se stesso. Avendo comunque accesso allo stesso spazio di memoria del padre e agli stessi permessi, ha la possibilità di accedere agli stessi file del padre e di reindirizzare il suo output a quest'ultimo (banalmente, lo standard output!).

2.3 Terminazione di un processo

La terminazione di un processo, può avvenire in varie circostanze.

- **Uscita normale (condizione volontaria).**

È la terminazione normale di un programma. I processi terminano per scelta dell'utente o da parte di un altro processo tramite una chiamata di sistema:

- **UNIX.**

`exit`

- **Windows.**

`ExitProcess`

Ritorna un exit status.

- **Uscita su errore (volontario).**

È una terminazione concepita dal programma.

- **Errore critico (involontario).**

È un meccanismo automatico che scatta quando il processo cerca di eseguire azioni anomale o illegali, come violazioni relativi allo spazio di indirizzamento (tentativo di scrittura in zone read-only), divisioni per zero, etc..

In quel caso, l'hardware fa scattare un'eccezione, causando un'interruzione del processo.

In questo caso, non avviene un tentativo di recoveri.

- **Terminato da un altro processo (involontario)** Avviene quando un processo **richiede** la terminazione di un altro processo.

- **UNIX.**

`kill`

- **Windows.**

`TerminateProcess`

Solitamente avviene quando i due processi avvengono nello stesso utente (l'amministratore può tuttavia uccidere i processi di qualsiasi utente).

2.3.1 Soft Kill, Hard Kill

Le richieste di terminazioni possono essere di due tipi:

- **Soft Kill.**

Viene mandato un segnale specifico al programma di cui vogliamo effettuare la chiusura. Il programma può rispondere con una routine può o addirittura rifiutare la chiusura.

- **Hard Kill.**

Consiste nella terminazione del processo a priori.

2.4 Ciclo di vita di un processo

I possibili stati di un processo possono essere astratti tramite un automa a stati finiti, cui stati sono i seguenti:

- 3 stati principali, **ready**, **running**, **blocked**
- I 2 stati iniziali e finali **new**, **terminated**

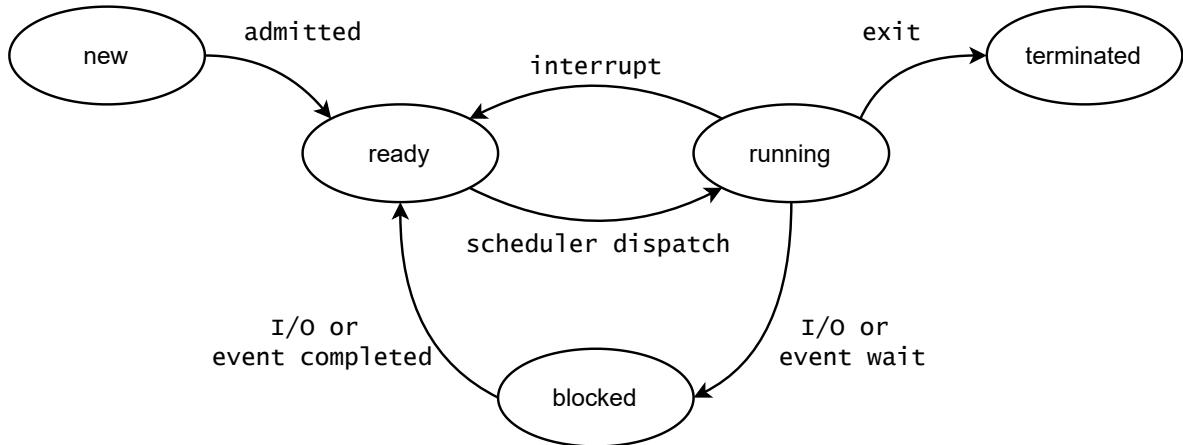


Immagine 2.1: Un diagramma a stati che descrive il ciclo di vita di un processo

2.4.1 Gli stati di un processo

Nello studio degli stati dei processi, supporremo sempre di avere una sola CPU.

New e Terminated

Sono due stati meno importanti.

Lo stato **new** fa riferimento alla fase di predisposizione delle risorse in funzione dell'apertura del processo.
Lo stato **terminated** fa riferimento alla fine dell'esecuzione di un processo.

Ready

Il processo è pronto ad usare la CPU. Il sistema operativo si occupa della gestione dei processi nello stato **ready**, che sono contenuti in una coda chiamata **coda dei processi**, di cui parleremo dopo.

La coda dei processi viene usata dallo scheduler per stabilire che processo far passare allo stato **running**. All'interno della coda si trovano i riferimenti ai processi. Effettuata la scelta, il sistema ha il compito di mettere in esecuzione il processo.

Running

Con i presupposti dell'unica CPU, **solo un processo può essere nello stato di running**. Dallo stato di **running**, in funzione di scenari diversi, si passerà a stati differenti.

- **Attesa di evento o Input/Output (Blocked).**

Il processo **ha bisogno di una chiamata a sistema** (e quindi di una **trap**), in quanto necessita **l'aiuto del sistema operativo**.

Alcune chiamate di sistema sono dette "lente". Sono chiamate che solitamente, hanno tempi di gestione più lenti delle istruzioni della CPU.

Clicca qui per leggere sulle chiamate lente e veloci.

Se la CPU è assegnata ad un processo su cui è stata effettuata una **syscall lenta**, il processo transita nello stato **blocked** (tirandolo fuori dalla coda dei processi). Rienterà nella coda e tornerà allo stato **ready** quando arriverà la risposta alla sua syscall lenta. Nota bene! Questo non vuol dire che il processo verrà ri-eseguito istantaneamente: bensì, tornerà nello stato **running** a discrezione dello scheduler.

- **Interrupt (Ready).**

Fa passare dallo stato di **running** allo stato di **ready**.

È contemplato nei sistemi che presentano il meccanismo di **prelazione**. In questo scenario, un ipotetico processo *A* in stato **ready** può ottenere la CPU, attualmente utilizzata da un processo *B* in stato **running**, aspettando un determinato interrupt mandato da un clock di sistema. Questo farà tornare *B* allo stato **ready**, per non fargli "monopolizzare" la CPU.

2.4.2 Prelazione

È l'operazione tramite cui un processo viene temporaneamente interrotto e portato al di fuori della CPU, senza alcuna cooperazione da parte del processo stesso, al fine di permettere l'esecuzione di un altro processo. Nei sistemi interattivi è fondamentale la prelazione, nessun processo deve poter monopolizzare la CPU. Ad esempio: un processo occupa la CPU per un quanto¹ di tempo. Se il processo non entra nello stato **blocked** entro la fine del quanto di tempo, passa nello stato di **ready**, e la CPU viene dedicata ad un altro processo.

Questo evento è gestito tramite un **interrupt** legato ad un clock di sistema (e viene quindi mandato dall'hardware): ad ogni segnale di clock, viene invocata una routine della CPU, che verifica se il processo ha superato il quanto di tempo dedicato. In tal caso, il processo entra nello stato di **ready**.

2.4.3 Busy wait vs Blocking wait

Notiamo che il meccanismo dello stato di **blocking** evita lo spreco di cicli di CPU. La CPU non controlla ad ogni step se la condizione di attesa è stata soddisfatta, ma è il processo cui risposta viene attesa, che segnala di poter far tornare il processo bloccato nello stato **ready**.

¹unità di misura che useremo per indicare il tempo dedicato ad un determinato processo

2.5 Tabella dei processi

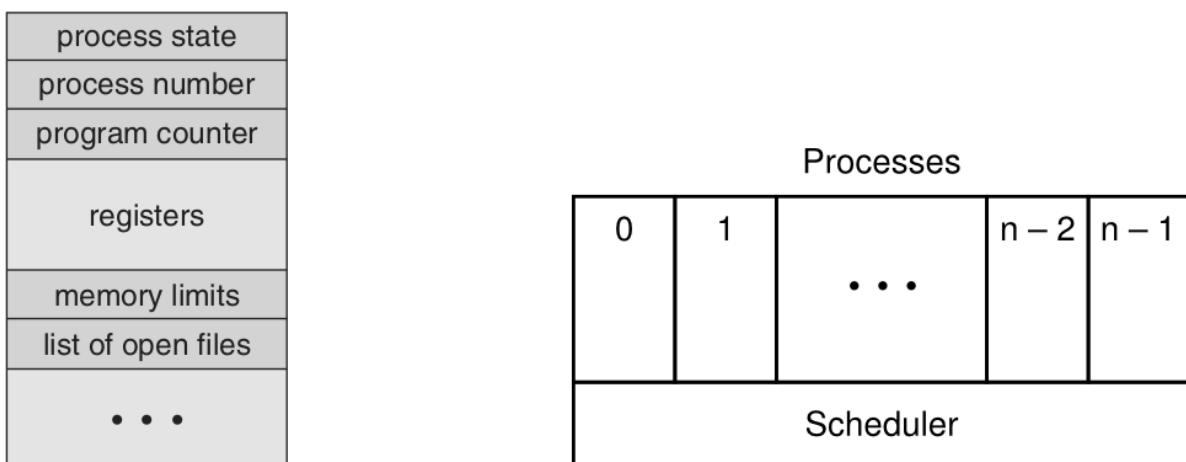
È una struttura dati dinamica, usata dal sistema operativo per gestire i processi, e che contiene al suo interno dei record, chiamati Processo Control Block (PCB). Quando un nuovo processo viene creato, viene allocato un PCB. Quando un processo termina, viene deallocated.

2.5.1 PCB - Process Control Block

Sono i record della tabella dei processi. Ne è presente uno per ogni istanza di un programma, e quindi uno per processo. Contengono al loro interno:

- **Stato** del processo.
- **Process ID (PID)**.
- **Program Counter (PC)**.
- Contenuto dei **registri della CPU**, salvato per ripristinarne il contenuto nel momento dell'esecuzione.
- Limiti di memoria (**spazio degli indirizzi**).
- **File aperti**.
- **Processi imparentati**.

In generale, le PCB contengono tutto ciò che possa servire allo **scheduler** per effettuare i **cambi di contesto**, noti anche come **Context Switch**, che approfondiremo più avanti.



2.5.2 Scheduler

È il componente del Kernel che sceglie quale processo deve essere eseguito, dalla coda dei processi pronti. La scelta del processo da eseguire in un determinato istante non è banale, e stabilire le proprietà variano spesso dal sistema (personal computer, sistemi real-time, batch e HPC avranno priorità differenti).

2.6 Coda e accodamento

I processi nello stato **ready** sono pronti ad usare la CPU, e sono situati in una coda, chiamata **ready queue**. Quando un processo viene selezionato dallo scheduler, a questo è assegnata la CPU, in un processo chiamato **process dispatching**.

2.6.1 La coda dei processi pronti

La tabella dei processi è una struttura dinamica al cui interno sono situati i PCB. La coda dei processi pronti è (chiaramente) un sottoinsieme della tabella dei processi, che coinvolge tutti i processi nello stato **ready**. Sarebbe ridondante proporre una struttura a parte per la coda, considerando che esiste già la tabella dei processi (che in quanto dinamica, è implementata come una lista doppiamente concatenata).

Ne consegue che il modo migliore per gestire la cosa, è sfruttare direttamente la tabella dei processi, aggiungendo dei puntatori dedicati invece alla coda dei processi pronti.

2.6.2 Diagramma di accodamento

Sono tre le circostanze in cui, un processo nello stato **running** può incorrere.

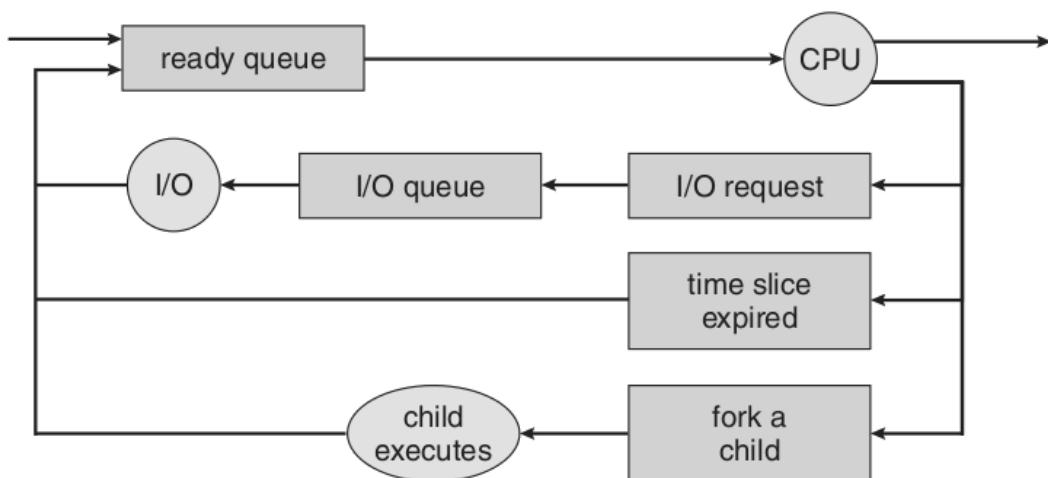


Immagine 2.2: Casi di riaccodamento di un processo in esecuzione

- **Richiesta di I/O:** il processo verrà inserito in una coda I/O di attesa.
- **Una fork di un processo:** genererà un processo figlio, di cui il padre attenderà la terminazione.
- Il processo viene rimosso dallo stato di running a causa di un **interrupt**, tornando così nella **coda dei processi pronti** (quindi, nello stato **ready**).

Il sistema operativo gestisce delle code per ciascuna di queste circostanze.

Prelazione

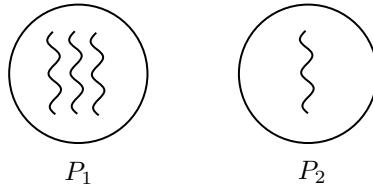
L'ultimo caso, include anche il **timer interrupt**: il clock di sistema, alla conclusione di un timer, invia un segnale di interrupt (detto **clock interrupt**), in modo da non far monopolizzare la CPU ad alcun processo. Questo meccanismo è presente nei sistemi operativi che supportano la **prelazione**. La prelazione è fondamentale in sistemi utente multiprogrammati.

Capitolo 3

Thread

3.1 Introduzione al modello a thread

Detto anche modello a thread multipli, permette di gestire l'esecuzione di un singolo processo su **più flussi di esecuzione indipendenti**. Il disegno sottostante rappresenta il nostro modello:



dove P_1, P_2 sono processi, e le linee all'interno sono i thread. **Processi diversi non hanno risorse comuni**, sono indipendenti. I **thread dello stesso processo hanno invece risorse condivise**. Possono offrire servizi paralleli allo stesso processo, permettendo di ottenere performance migliori e/o meno circostanze bloccanti per l'intero processo. Il motivo per cui preferiamo scomporre i processi in thread e non in processi, è relativo allo spazio degli indirizzi condiviso: il context switch tra thread dello stesso processo è più veloce del context switch di due processi separati

Non possiamo lavorare a singoli thread? What if - Esempi vari

Immaginiamo di avere la possibilità di gestire **un solo thread per processo**:

- Un text editor: il rendering del file si bloccherebbe ad ogni salvataggio. Il controllo lessicale bloccherebbe la scrittura.
- Un web server: le richieste possono essere gestite solo una ad una. Una richiesta molto onerosa blocca tutte le altre richieste, creando una coda non indifferente, e riducendo il throughput.

Avere la possibilità di gestire più flussi di esecuzione, è fondamentale per avere programmi più complessi e responsivi.

3.2 I thread

Un thread è un flusso di esecuzione indipendente di un processo. I thread dello stesso processo hanno lo stesso PID (Process ID). Ciascun thread è caratterizzato da:

- **Program Counter**
- **Registri**
- **Stack**
- **Stato: blocked, ready o running.**

E condividono tutto il resto, ossia spazio di indirizzamento, heap, memoria globale, codice¹ e file aperti.

¹che nel libro è indicato con "testo"

3.2.1 Piccolo prezzo da pagare: gestione dei thread

I thread sono flussi indipendenti, ma non isolati nativamente o protetti vicendevolmente. Ciò consegue che due o più thread che operano sulle stesse risorse, potrebbero causare anomalie di fronte a determinate circostanze, di cui parleremo più avanti. Le chiameremo **race condition**, o **corse critiche**.

3.2.2 Scheduling dei thread

Lo scheduler si occuperà della scelta del thread del programma che andrà ad occupare la CPU in un determinato istante. Nei sistemi moderni, lo scheduler tratta i thread come processi a se stanti.

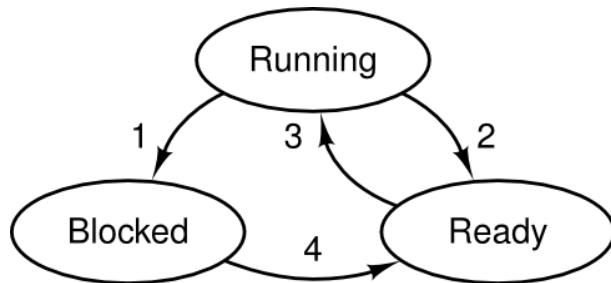
Context switch sui thread

Il concetto di **context switch**² avviene anche nell'ambito dei thread. Quando avviene il context switch, l'MMU viene riprogrammata sul thread entrante.

Il context switch dei thread avviene sempre tra thread dello stesso processo: rispetto al context switch tra processi, quello tra thread è molto più veloce, in quanto i thread condividono lo stesso spazio di memoria, sollevando l'MMU dal dover riprogrammare lo spazio di indirizzamento.

3.2.3 Cambiamenti di stato dei thread

I cambiamenti di stato dei thread sono analoghi a quelli dei processi. Possono essere nello stato di **running**, **ready** o **blocked**.



3.2.4 Operazioni tipiche sui thread

- **thread_create.**
Un thread ne crea un altro.
- **thread_exit.**
Il thread chiamante termina.
- **thread_join.**
Un thread si sincronizza con la fine di un altro thread.
- **thread_yield.**
Il thread chiamante rilascia volontariamente la CPU.

²Ricordiamo cos'è il context switch: consiste nel salvataggio del contesto corrente e nel ripristino del contesto entrante.

3.3 Introduzione al multicore programming

I thread permettono di ottenere performance migliori sui sistemi multithread e multicore. Se in un sistema single-core possiamo ottenere un'esecuzione *interleaved* e pseudo parallela, su un sistema multicore possiamo ottenere un parallelismo vero e proprio: in entrambi i casi si parla di concorrenza, ma solo in uno dei due, il parallelismo è vero e proprio (anche se, le intuizioni relative allo pseudo-parallelismo rimangono sempre fondamentali, in quanto il numero di thread eseguibili in parallelo è limitato).

3.3.1 Programmazione multicore

Progettare programmi che sfruttino le architetture multicore non è banale.

- **Separazione dei task.**

- **Bilanciamento.**

Ottenerne una suddivisione dei task ben bilanciati, significa gestire bene le responsabilità e le durate temporali dei task dei thread, che dovranno essere bilanciate in maniera opportuna.

- **Suddivisione e dipendenze dei dati.**

Bisogna analizzare i dati di cui i thread avranno bisogno, individuare le risorse dati condivise e se esistono dipendenze tra i dati.

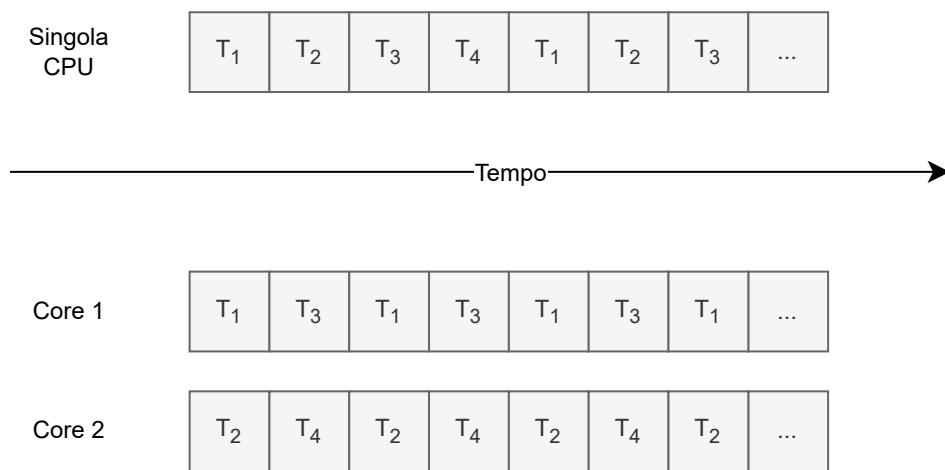
Cosa intendiamo con dipendenza? Il thread *A* ha bisogno di un dato elaborato dal thread *B*.

- **Test e debugging.**

La fase più odiata probabilmente. È molto complesso ottenere un programma multithread corretto: altrettanto difficile è riuscire a stanare, in fase di testing, tutti i casi limite e le circostanze problematiche che possiamo riscontrare nell'esecuzione del codice.

3.3.2 Sistemi single-core vs multi-core

I thread permettono di suddividere l'esecuzione di un programma in più flussi. In sistemi single-core, si ottiene un'esecuzione alternata. In sistemi multicore, si ottiene un'esecuzione parallela (ma anche alternata) vera e propria.



3.3.3 Tipologie di thread

- **Thread a livello utente.**

1-a-molti, artificialmente creati a livello utente su un unico flusso di esecuzione suddiviso da uno scheduler. Non devono essere supportati nativamente dal Kernel.

- **Thread a livello Kernel.**

1-a-1, con un thread a livello utente associato ad ogni thread a livello Kernel. Devono essere supportati nativamente dalla CPU.

- **Thread in modalità ibrida.**

È un modello ibrido, detto molti-a-molti.

3.4 Thread a livello utente

Detto anche **modello 1-a-molti**, è utile per lavorare su un Kernel che non ha supporto nativo ai thread. Il sistema operativo non sa cos'è un thread: dato un programma, offre la possibilità di operare su un solo flusso di esecuzione. Tuttavia, tramite opportune librerie, è possibile introdurre dei thread a livello utente, a pari del nostro codice. Sarà poi l'implementazione dei thread da parte della libreria, a gestire l'esecuzione concorrente.

3.4.1 Come fa la CPU a saltare tra thread?

La chiave per lo switch tra thread è una primitiva che abbiamo introdotto precedentemente: `thread_yield`. Essa effettuerà il salvataggio dei registri sulla tabella dei thread, anche essa nello spazio utente.

3.4.2 Criticità sui thread a livello utente

Il modello 1-a-molti è molto versatile, ma presenta delle criticità.

Chiamate bloccanti Le chiamate lente, o bloccanti, bloccano tutti i thread del processo. Questo, avviene a carico del sistema operativo, che nell'effettivo non conosce i thread a livello utente. **Per il sistema operativo, i thread a livello utente non sono altro che codice di un unico processo.** Ciò è molto grave, soprattutto rispetto a operazioni lente sul disco o operazioni di I/O.

Una soluzione? La chiama a sistema `select`, che verifica se una chiamata, in un determinato istante e contesto, sarà bloccante, ossia a condizione della chiamata lenta soddisfatta.

- Se la chiamata *sarà* bloccante, l'esecuzione procederà regolarmente.
- Se la chiamata *non sarà* bloccante, si usa la primitiva `thread_yield` su quel thread. Ciò avverrà fino a quando quella chiamata non sarà più bloccante.

Esempio semplice: t_1 aspetta un input da tastiera. La richiesta input è una syscall lenta se non è già presente testo nel buffer d'input. `thread_yield` su t_1 per farne rilasciare la CPU. Questo passaggio si ripete fino a quando non sarà presente testo nel buffer. In tal caso, la `select` darà il via libera per il non richiamo della `thread_yield` su t_1 . Implementiamo così un meccanismo a livello utente per far rilasciare la CPU ad un thread. Anche i **page-fault** causano l'interruzione di un processo: a malincuore, anche in questo caso non è possibile fermare esclusivamente il thread.

Possibilità di non rilascio Un thread potrebbe non rilasciare la CPU. Un thread non può forzare un altro thread a rilasciare la CPU. La libreria che implementa i thread dovrà gestire in maniera opportuna questo aspetto.

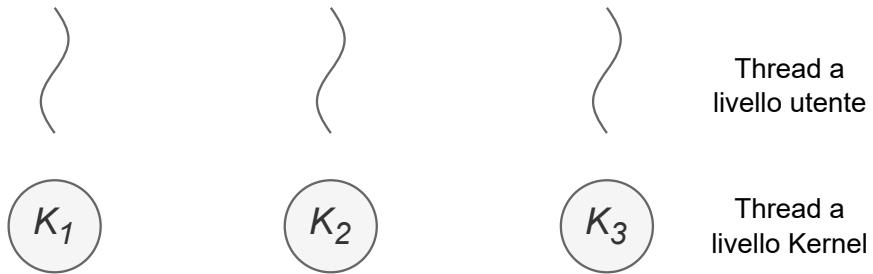
3.4.3 I punti di forza

Scheduling personalizzato I thread a livello utente non possono sfruttare la prelazione, in quanto quest'ultima sfrutta degli interrupt di sistema. Per quanto questa potrebbe risultare come una limitazione, la natura a livello utente di questi thread, permette di implementare un meccanismo di priorità personalizzata dei processi contestuale e meno asettica di quella a sistema operativo: più versatilità.

Dispatching senza trap L'operare interamente a modalità utente di questi thread, evita i tempi di attesa relativi alle chiamate a sistema, notoriamente lunghe. I thread a livello utente sono notoriamente più veloci e meno costosi.

3.5 Thread a livello Kernel

Detto anche "modello 1-a-1". Sfrutta le possibilità del sistema operativo. La thread table diventa a livello Kernel, ed unica per tutti i processi, in quanto è proprio il Kernel a supportare i thread. Nel pratico sono addirittura unificate, in quanto i processi saranno definiti come sottoinsiemi di elementi della tabella dei thread.



3.5.1 Vantaggi di questo modello

In questo modello, una chiamata bloccante può bloccare un solo e unico thread, eliminando la più grande criticità dei Kernel a livello utente!

3.5.2 Svantaggi

Il context switch In questo caso, il context switch è più lento, come conseguenza dell'utilizzo di `trap`.

Il context switch più lento è quello tra thread di due processi differenti. Più veloce sarà quello tra due thread imparentati, in quanto non richiederà la riprogrammazione dell'MMU.

Operazioni costose Creare e distruggere thread diventa un'operazione molto più costosa. Creare thread Kernel frequentemente può rallentare esponenzialmente il sistema.

3.6 Modello ibrido

Detto anche "molti-a-molti". Prende il meglio dal mondo dei thread-utente e dei thread-Kernel.

3.6.1 Come si procede nel modello ibrido?

Si prevede un dato numero di thread Kernel, fisso e non troppo grande. Successivamente, si assegna ad ogni thread Kernel uno o più thread utente. Un esempio? Due thread Kernel, tre thread utente. I due thread utente sullo stesso thread Kernel vanno sul controllo ortografico e sul salvataggio (sono operazioni dietro le quinte), l'ultimo thread utente assegnato ad un singolo thread Kernel si occupa della GUI (più delicata e responsiva all'utente).

3.7 I thread nei nostri sistemi operativi

Quasi tutti gli OS supportano i thread a livello Kernel, ma usano primitive diverse. I thread utente sono supportati tramite apposite librerie.

3.7.1 `pthreads`

La libreria `pthreads`, basata sullo standard POSIX, introduce thread a livello utente e funzioni per renderne la gestione quanto più semplice possibile, con un'interfaccia univoca. Univoca in quanto il codice sarà compilabile e funzionante su molteplici sistemi operativi: ciò che succede dietro le quinte cambia, ma il codice sorgente e i risultati rimangono analoghi.

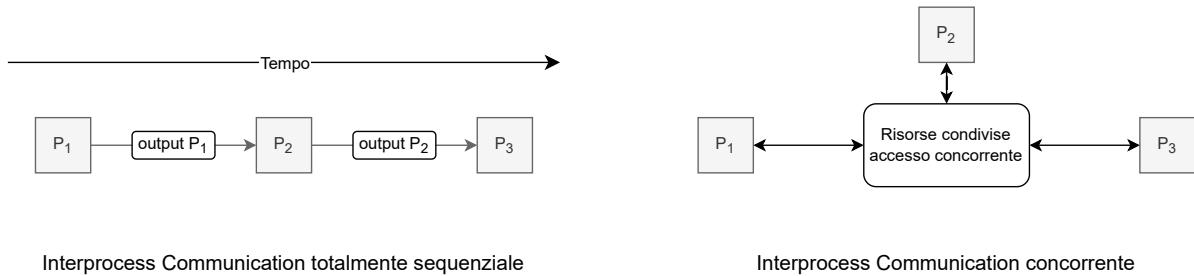
Capitolo 4

Interprocess Communication

Gestire in maniera opportuna la comunicazione tra n processi, è fondamentale per ottenere sistemi reattivi, veloci e performanti.

4.1 Introduzione all'IPC

Il problema della gestione della comunicazione tra processi è un problema cui complessità dipende dalla tipologia di processi e insieme di processi. Un sistema di processi **concorrenti** e **comunicanti** (tramite risorse condivise) è molto più complesso di un sistema in cui un processo P_n è seguito dal processo P_{n+1} , in maniera totalmente sincrona e sequenziale.



4.1.1 Modello pipe

È il modello più semplice che permette di mettere comunicazione più processi. Sfrutta i canali di input/output per far comunicare i processi.

```
> cmd1 | cmd2 | cmd3
```

Con questa sintassi da terminale, vengono messi in comunicazione l'output del cmd1 con l'input del cmd2, e analogamente, l'output del cmd2 nell'input del cmd3. Naturalmente questi tre aspetti sono isolati, ma in questo modo è possibile concatenare più processi.

4.1.2 Implicazioni del modello pipe

cmd2 aspetterà cmd1 in quanto attenderà il suo output, creando una dipendenza di tempismo. Tra ogni due processi, sarà inoltre presente un buffer che permette di conservare gli output di P_n in attesa di P_{n+1} .

4.1.3 Criticità da gestire nell'interprocess communication

- **Lo scambio dei dati.**

Nel modello pipe, P_n non può comunicare con P_{n-1} . I processi avvengono in maniera lineare e unidirezionale.

I dati non sono propriamente condivisi, il che è molto limitante.

Una soluzione: segmento di memoria condiviso.

Un approccio ideale è quello con un segmento di memoria condiviso: gestito in maniera opportuna il coordinamento delle operazioni, è possibile condividere le risorse sfruttando uno spazio di memoria condiviso d'appoggio.

- **Accavallamento delle operazioni su dati comuni.**

L'accesso alle risorse comuni deve essere gestito in maniera opportuna.

- **La sincronizzazione.**

Le operazioni andranno coordinate (o sincronizzate) in maniera opportuna. In un sistema come la pipeline è molto semplice gestire la sincronia. È meno scontato in modelli di comunicazione differenti, come quelli in cui i processi sono concorrenti.

4.2 Race Condition

In italiano, **corse critiche**. Quella delle race condition è una circostanza che può avvenire quando due processi (o thread) concorrenti operano su delle risorse condivise. Immaginiamo di avere due processi concorrenti che operano sugli stessi dati:

$$P_1 : x = x + 10, \quad P_2 : x = x + 10$$

Valore atteso	Valore effettivo
0	0
10	10
20	10
30	20
40	20

4.2.1 Sezioni critiche

Il concetto delle sezioni critiche permette di gestire le race condition.

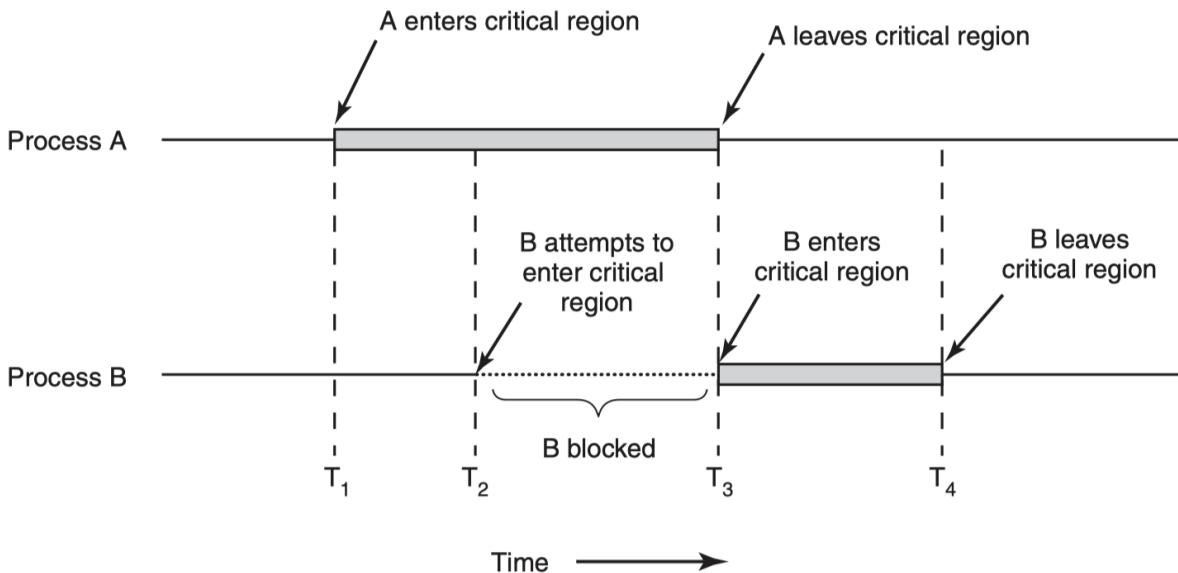
Definiamo come **sezione critica** una sezione di codice in cui si accede a delle risorse condivise. Quando due processi concorrenti P_A, P_B possono accedere alla stessa risorsa, solo uno tra i due processi dovrebbe poter accedere alla sezione critica.

```
A entra nella sezione critica.  
B vuole accedere alla sezione critica.  
B viene bloccato.  
[...]  
A esce dalla sezione critica.  
B entra nella sezione critica.  
[...]  
B esce dalla sezione critica.
```

Le race condition gestite in maniera inopportuna, causano anomalie nei risultati. Queste anomalie porterebbero a risultati inconsistenti a run-time.

4.2.2 Definire le sezioni critiche nel programma

Le sezioni critiche sono individuate dal programmatore: solitamente, è consone definire sezioni critiche differenti per strutture dati indipendenti. Dati n processi, e due strutture dati H e T (indipendenti tra loro) è opportuno definire delle sezioni critiche su H e delle sezioni critiche su T , facendo in modo



4.2.3 Com'è fatta una buona soluzione per race condition?

- Mutua esclusione.
- Nessuna assunzione sulla velocità di esecuzione o sul numero di CPU.
- Nessun processo fuori dalla propria sezione critica può bloccare altri processi.
- Nessun processo dovrebbe restare all'infinito in attesa di entrare nella propria zona critica. Dovrebbe essere garantito un tempo massimo di attesa. Con ciò, non intendiamo introdurre un timer di time-out, ma piuttosto dobbiamo avere delle garanzie relative al tempo che un processo dovrà aspettare prima di entrare nella zona critica. Questo, dovrà essere un tempo finito.

L'idea è quella di individuare le sezioni critiche e circoscriverle con delle generiche funzioni `enter_region()` e `leave_region()`. Soluzioni differenti implicano comportamenti differenti da parte di queste funzioni. Quelle che andremo a vedere, sono alcune tra le intuizioni principali che hanno portato alle soluzioni moderne.

4.3 Soluzione 1 - Inibire gli interrupt

In un sistema mono-CPU, **disabilitare gli interrupt** (quindi il meccanismo di **prelazione**) permette di evitare che l'esecuzione di una sezione critica di un processo venga interrotta.

Le istruzioni che avvengono all'interno della sezione critica a interrupt disabilitati, diventano automaticamente istruzioni atomiche, che non possono essere interrotte da altri processi.

- A entra nella sezione critica
interrupt inibiti
 A prende il valore di x
 A incrementa e assegna x
interrupt ripristinati
- flusso di esecuzione standard, A e B si alternano
in sezioni non critiche di codice grazie alla prelazione
- B entra nella sezione critica
interrupt inibiti
 B prende il valore di x
 B incrementa e assegna x
interrupt ripristinati

4.3.1 Pro di questa soluzione

Questa soluzione funziona E non è cosa da poco! Molte delle intuizioni che vedremo in avanti, non portano alle conclusioni sperate. È inoltre concettualmente molto semplice.

4.3.2 Contro di questa soluzione

Tuttavia, questa soluzione presenta anche un insieme di criticità non indifferenti

Solo thread a livello Kernel Disabilitare gli interrupt è un'operazione che richiede i privilegi della Kernel mode, quindi i thread dello spazio utente non possono utilizzare questa strategia.

Non funziona nei sistemi multicore Se P_1 viene eseguito sulla CPU_1 e P_2 sulla CPU_2 , non ci sarà accesso esclusivo alle zone critiche, in quanto ogni inibizione degli interrupt, è contestuale alla CPU su cui viene effettuata.

Da grandi poteri, derivano grandi responsabilità Siamo sicuri di voler lasciare al programmatore la responsabilità di inibire liberamente gli interrupt di sistema?

4.4 Soluzione (non funzionante) 2 - Variabili di lock

L'idea sarebbe quella di avere una **variabile di lock** cui valore a 0 denota che **nessun processo è nella propria sezione critica**. Se P_1 vuole entrare nella propria sezione critica, verifica se $lock=0$. No? Aspetta. Altrimenti, imposta $lock=1$ e entra.

Race Condition sulla variabile di lock Se un processo P_2 viene schedulato dopo la verifica da parte di P_1 , ma prima dell'incremento di $lock$, sia P_1 che P_2 entreranno nella sezione critica. Verificare una seconda volta il valore di $lock$ dopo l'incremento, non risolve il problema. Si ha una corsa critica se il secondo processo modifica la variabile subito dopo il controllo del primo processo.

Busy Waiting Un'ultima osservazione, riguarda il modo cui, un processo che non può entrare nella propria sezione critica, interroga il valore di $lock$. Questo verrebbe implementato con busy waiting, un approccio naive che spreca molte risorse.

4.5 Soluzione 2 - Alternanza Stretta

I processi si passano il testimone ogni volte che escono dalla propria sezione critica.

```
int N = 2 // numero di processi
int turn

function enter_region(int process):
    while(turn != process):
        do nothing // attende il proprio turno

function leave_region(int process):
    turn = 1 - process // passa il turno al prossimo
```

La variabile `turn` indica di quale processo sia il turno. Quando non è il turno del processo indicato dalla variabile `process`, si entra nel loop `do nothing`. La funzione `leave_region(int process)` commuta il turno.

La soluzione funziona, anche in modalità utente Non richiede privilegi del sistema, ed è quindi utilizzabile in modalità utente, ed è una soluzione totalmente software. Tuttavia...

Busy Waiting sulla variabile `turn`, sprecando cicli di clock.

Impone turni rigidi Viola quindi la terza condizione. Infatti, un processo può essere bloccato da un altro che non è nemmeno interessato a entrare nella propria sezione critica.

Usa una variabile condivisa su cui non si hanno race condition, ma è comunque un aspetto da considerare.

4.6 Soluzione 3 - La soluzione di Peterson

Offre mutua esclusione tra 2 o più usando una variabile e una struttura dati, entrambe comuni ai processi.

- Una variabile `turn` che specifica quale processo può accedere alla sezione critica
- Un vettore `interested[N]`, che contiene, per ogni processo, un valore `true` o `false` che ne indica l'interesse nell'entrare nella sezione critica.

```
int N = 2; // numero di processi
int turn;
bool interested[N]; // tutti a false di default

// in entrata alla sezione critica
void enter_region(int process) {
    int other;
    other = 1 - process;
    interested[process] = true; // il processo in questione si dichiara "interessato"
    turn = process;           // si imposta il turno al proprio numero di processo

    while (interested[other] == true && turn == process) {
        /* do nothing*/
    }
}

// in uscita dalla sezione critica
void leave_region(int process) {
    interested[process] = false;
}
```

La riga `turn = process` con il susseguente `turn == process`, apparentemente inutile e ridondante, permette in realtà di capire quale processo sia arrivato primo **in uno scenario concorrente**.

È una soluzione che funziona, anche se poco scalabile quello che presentiamo è la soluzione su due soli processi. Può essere generalizzato al caso N , ma è molto complesso: ciò che si fa è effettuare una serie di confronti tra coppie di processi, come in una sorta di torneo. Questo stratagemma permette di generalizzare la soluzione di Peterson.

Busy waiting sulla variabile `turn`.

Non funziona sui moderni multiprocessori Può avere problemi sui moderni multi-processori a causa del riordino degli accessi alla memoria centrale. Questa soluzione infrange definitivamente le aspettative verso la soluzione di Peterson. Clicca qui per una spiegazione.

4.7 Soluzione 4 - Istruzioni TSL

Questa soluzione riprende l'idea delle variabili di lock. Fornire un supporto hardware tramite delle istruzioni specializzate per gestire correttamente le variabili di lock, è una soluzione usata nelle architetture più recenti. Ciò avviene tramite un'istruzione ausiliaria: TSL o XCHG (ossia Test and Lock e Exchange), dipendentemente dall'architettura.

4.7.1 Test and Set Lock

È un'istruzione che, in maniera atomica, permette di testare e impostare la variabile di lock.

L'istruzione TSL prende due argomenti. Ricopia il valore della variabile LOCK all'interno di REGISTER, e poi impostare LOCK a 1. Rende atomiche due operazioni di MOV. Questo è il comportamento delle funzioni di `enter_region` e `leave_region`:

```
enter_region:  
    TSL REGISTER, LOCK // registro = lock, lock = 1  
    CMP REGISTER, #0 // se il registro è a 0, entra nella sezione critica  
    JNE enter_region // altrimenti aspetta  
    RET  
  
leave_region:  
    MOVE LOCK, #0 // reimposta lock a 0  
    RET
```

Perché funziona? Questo offre una garanzia rispetto alla prelazione: un'istruzione atomica può arrivare prima, o dopo la prelazione. La prelazione non può scindere un'istruzione atomica. Questa istruzione **blocca per qualche istante il bus di memoria**. Chiaramente, la TSL è *solo* un'istruzione ausiliaria, il resto è fatto da `enter_region` e `leave_region`.

Funziona, ma fa uso di busy waiting È una soluzione che funziona su architetture moderne, funziona in modalità utente, e **in sistemi multicore** ma fa ancora uso di **busy waiting**.

Istruzioni equivalenti in altre CPU

XCHG è disponibile in tutte le CPU Intel X86. Funziona in maniera differente dalla TSL, ma cambiando leggermente le `enter_region` e `leave_region` otteniamo soluzioni equivalenti.

4.8 Soluzione 5 - Sleep & Wake Up

Tutte le soluzioni esposte fino ad ora fanno uso di busy waiting, ed effettuano un così detto **spin lock**. Tuttavia, tutte le istruzioni che fanno uso di busy waiting, soffrono del **problema dell'inversione delle priorità**.

4.8.1 Problema dell'inversione delle priorità

Dati due processi P_H, P_M, P_L , che stanno per High, Medium e Low, riferiti alla priorità dei processi. Questi tre processi lavorano sulla stessa struttura dati, usando la stessa variabile di lock e le istruzioni TSL. P_L può agire solo dopo P_M, P_H , che saranno bloccati. Supponiamo che questo sistema non faccia uso di prelazione. Quando P_L userà la CPU, P_H non potrà più rientrare.

4.8.2 Due nuove chiamate di sistema

Per evitare lo spreco di CPU causato dal busy-waiting, ma anche il problema dell'inversione di priorità, l'OS offre primitive di comunicazione tra processi che permettono di portare un processo nello stato "blocked", invece che di portarlo in una attesa attiva. Un esempio sono proprio le istruzioni `sleep` e `wakeup`:

- `sleep` sospende l'esecuzione del chiamante. Questo potrà tornare ad avere la CPU solo quando sarà risvegliato. Manderà il processo nello stato 'blocked'.
- `wakeup` sveglia un processo, facendolo tornare nello stato 'ready'.

4.9 Problema del produttore consumatore

È un problema giocattolo che descrive una circostanza tipica di sincronizzazione tra processi che operano su strutture dati condivise.

4.9.1 Descrizione del problema

Consideriamo un **produttore**, un **consumatore** e un **buffer condiviso** su cui è possibile inserire o prelevare item. Supponiamo il buffer sia una struttura statica e limitata¹. Il produttore inserisce elementi sul buffer (tranne quando il buffer è pieno) e il consumatore rimuove elementi dal buffer (fino a quando non è vuoto). È un problema formulabile anche con più produttori e/o consumatori. Produttori e consumatori sono i nostri processi o thread.

4.9.2 Una prima soluzione

```
function producer():
    while (true):
        item = produce_item
        if (count = N):
            sleep()
        insert_item (item)
        count = count + 1
        if (count = 1):
            wakeup(consumer)
```

1. Se il buffer è pieno, si addormenta.
2. Quando è sveglio, produce sul buffer, ovvero `count = count + 1`
3. Quando il `count = 1`, il producer sveglia il consumer con `wakeup(consumer)`

```
function consumer():
    while (true):
        if (count = 0):
            sleep()
        item = remove_item()
        count = count - 1
        if (count = N - 1):
            wakeup(consumer)
        consume_item(item)
```

1. Se il buffer è vuoto, si addormenta.
2. Quando è sveglio, consuma dal buffer, ovvero `count = count - 1`
3. Quando il buffer è pieno, il consumer sveglia il producer con `wakeup(consumer)`.

Deadlock

Tenendo bene a mente che i processi operano in maniera **concorrente**, ecco la circostanza che ci porta in deadlock:

1. Il consumatore verifica se `count = 0`, la condizione è soddisfatta.
2. Il consumatore viene interrotto prima di entrare nel corpo dell'`if`, avviene il context switch col produttore.
3. Il produttore inserisce un nuovo elemento del buffer.
4. Il produttore sveglia (ed è qui che inizia l'anomalia) il consumatore (ancora sveglio, quindi non cambierà nulla) in quanto il buffer ha nuovamente almeno un elemento.
5. Quando il controllo della CPU tornerà al consumatore, esso si addormenterà, entrando nel corpo dell'`if`.

¹Una struttura dinamica renderebbe il problema meno generico e di più facile gestione.

6. Il produttore sarà l'unico in funzione, il quale continuerà a inserire elementi del buffer.
7. Riempito il buffer, il produttore si addormenterà. Entrambi gli attori si sono addormentati.

4.9.3 Soluzione che potrebbe funzionare?

Implementare un bit di attesa. Funziona su un produttore e un consumatore, ma non uno di più. Non è una soluzione scalabile. Introduciamo la soluzione definitiva che funzionerà su n produttori ed m consumatori, e che potremmo riutilizzare anche in contesti applicativi reali.

4.10 I semafori

Generalizzando il concetto di sleep e wakeup, otteniamo un **semaforo**. È uno strumento software offerto dal sistema operativo che nasce con lo scopo di gestire operazioni concorrenti.

4.10.1 Struttura del semaforo

- **Variabile S.**

Ha una variabile intera condivisa S . Questo valore rispetta una regola: non può mai diventare negativa. In ogni istante di vita del semaforo, $S \geq 0$.

- **Funzioni wait, signal.**

Due operazioni, conosciute anche come **up** e **down**. Queste due operazioni, incrementano e decrementano la variabile S . Se si prova a effettuare **wait** sul semaforo a 0, l'operazione diventa bloccante. L'incremento tramite **signal** non ha limitazioni specifiche.

Operazioni atomiche del semaforo

Per il corretto funzionamento, bisogna fare in modo che **down** e **up** siano atomiche, per evitare problemi con race condition sul semaforo. Per fare in modo che sia così, possiamo sfruttare:

- **Disabilitazione interrupt.**

Funziona in un sistema mono-core.

- **Spin lock TSL/XCHG.**

Usa una variabile di lock, introducendo un busy wait che dura per quel (minimo, veramente corto) quanto di tempo necessario per le poche operazioni relative alle istruzioni **wait** e **signal**. È un busy wait insignificante e un compromesso che possiamo accettare.

4.11 Tipi di semafori

Applicazioni differenti del semaforo, prendono nomi differenti.

Semaforo numerico È un semaforo cui S assume valori $\in \{0, 1, \dots, N\}$. Si presta ai problemi di conteggio delle risorse, bloccando il thread quando questo esaurisce la risorsa che conta. Nel problema produttore-consumatore, lo usiamo per contare il numero di slot liberi e vuoti. In questo modo, un produttore che cerca di inserire su un buffer pieno, o un consumatore che cerca di estrarre da un buffer vuoto, verranno bloccati.

Semaforo Mutex È un semaforo cui S assume valori da $\in \{0, 1\}$. Viene usato per garantire mutua esclusione nell'accesso a una determinata zona critica. La **enter_region** coinciderà con l'utilizzo di **wait(M)**. La **leave_region** coinciderà con l'utilizzo di **signal(M)**.

4.11.1 Soluzione al problema del produttore-consumatore basata su semafori

```
int N = 100
semaphore mutex = 1
semaphore empty = N
semaphore full = 0
type buffer[N]

function producer():
    while(true):
        item = produce_item()
        down(empty)
        down(mutex)
        insert_item(item)
        up(mutex)
        up(full)

function consumer():
    while(true):
        down(full)
        down(mutex)
        item = remove_item()
        up(empty)
        up(mutex)
        consume_item(item)
```

4.12 Futex - *Fast user space mutex*

Alcuni sistemi operativi supportano un'ottimizzazione relativa ai mutex, chiamata **futex**. I mutex che la implementano, garantiscono migliori performance: `down` e `up` sono delle chiamate di sistema, che hanno dei costi non indifferenti. L'idea dietro i Futex, è quella di gestire tutto in modalità utente, a meno del bloccato dei thread: questo, viene affidato al Kernel. Dividendo in questo modo le responsabilità, si riducono le chiamate a sistema, e quindi l'overhead complessivo. Come sempre, individuare cosa è possibile fare nello spazio utente, e cosa no, è il miglior modo per ottimizzare le risorse.

I thread di `pthreads` in Linux sono implementati come Futex.

Struttura del Futex

Futex sfrutta una coda di thread bloccati a livello Kernel, e una variabile di lock a livello utente. Il Kernel viene richiamato solo per operare sulla coda, o per gestire casi in cui la variabile di lock è contesa tra più thread.

Quando il Kernel non è richiamato, la gestione della variabile di lock è molto più veloce. L'overhead si crea solo quando il Kernel deve bloccare un thread che contende la variabile di lock.

4.13 Monitor

Sono delle astrazioni di alto livello che offrono una gestione semplice della mutua esclusione, e che sollevano il programmatore dalle responsabilità che avrebbe avuto, dovendo usare i mutex e le chiamate down e up.

4.13.1 In cosa consistono

Sono un tipo di dato astratto contenenti dati e metodi. All'interno degli oggetti di tipo monitor, esistono molteplici vincoli sui dati

- Una struttura dati definita all'interno del monitor non è accessibile da fuori.
- I metodi del monitor non possono accedere a strutture esterne.
- Tutti i metodi del monitor sono eseguiti in mutua esclusione.

Queste garanzie sono offerte dal linguaggio di programmazione (che si occupa di implementare questi monitor), facilitando il lavoro del programmatore, e assicurando che, in qualsiasi istante, solo un thread sia nel monitor. Questi sistemi offrono non solo una gestione della mutua esclusione (eventuali blocchi del sistema), ma anche una gestione della sincronizzazione tra i metodi.

Signal e wait

Per garantire la sincronizzazione vengono aggiunte le variabili condizione, etichette su cui il thread stesso dall'interno del monitor può esercitare una `wait` e una `signal`.

- **wait.**
Provoca il bloccaggio, testando una condizione e verificando che essa sia vera: se la condizione è vera, il thread si blocca. Questo meccanismo è stato già visto con il controllo nel produttore e consumatore: se il buffer è pieno, il produttore si deve fermare.
- **signal.**
Intesa come un risveglio, un altro processo che ha accesso alla regione critica può risvegliare un processo usando signal sulla variabile condizione su cui il processo stava attendendo.

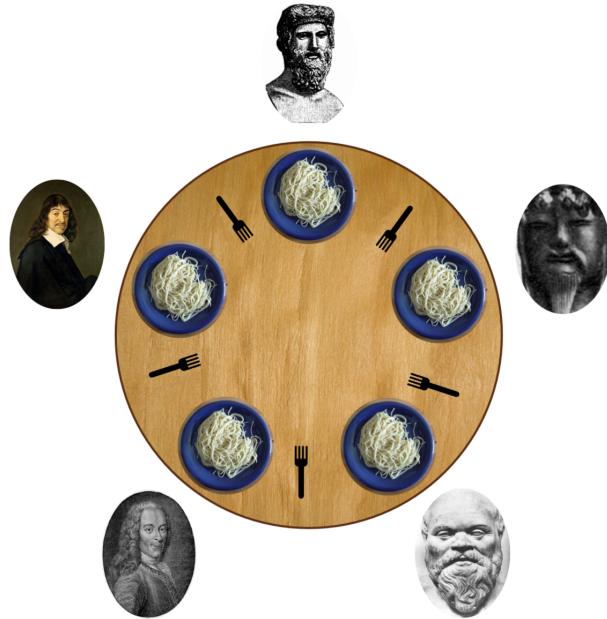
Le variabili condizione non sono semafori, ma etichette senza stato e memoria, che stabiliscono le condizioni di bloccaggio di un processo. Ricordiamo che ogni cosa che avviene nel monitor, avviene in mutua esclusione.

4.13.2 Semantiche signal

	Hoare (Signal-and-Wait)	Mesa (Signal-and-Continue)	Pascal
Chi continua dopo <code>signal()</code> ?	Il segnalato (chi era in <code>wait()</code>)	Il segnalante	Il segnalante
Quando il segnalato riprende?	Subito dopo la <code>signal()</code>	Dopo che il segnalante esce dal monitor	Dopo che tutti i thread attivi escono dal monitor
Chi ha la mutua esclusione subito dopo?	Il segnalato	Il segnalante	Il segnalante
Comportamento del monitor	Il segnalante si sospende	Il segnalato va in coda di attesa	Il segnalato ritarda ancora di più

4.14 Problema dei 5 filosofi

Il problema dei filosofi a cena, altrimenti noto come **problema dei cinque filosofi**, è un esempio giocattolo che illustra un comune problema di controllo della concorrenza tra processi paralleli. I 5 filosofi sono dei processi. Ciascuno dei filosofi può pensare o mangiare, ma mangiare richiede due forchette. Il problema è evidente: bisogna trovare una strategia che non porti né in una situazione di stallo (**deadlock**), né uno di questi cinque filosofi a morire di fame (**starvation**).



4.14.1 Soluzione basata sui semafori

```

int N=5; //numero filosofi
int THINKING = 0
int HUNGRY = 1
int EATING = 2
int state[N]
semaphore mutex = 1
semaphore s[N]={0,...,0}

function philosopher(int i)
    while(true):
        think()
        take_forks(i)
        eat()
        put_forks(i)

function take_forks(int i)
    down(mutex)
    state[i] = HUNGRY
    test(i)
    up(mutex)
    down(s[i])

function test(int i)
    if state[i] = HUNGRY and state[left(i)] != EATING and state[right(i)]!=EATING
        state[i] = EATING
        up(s[i])

function put_forks(int i)
    down(mutex)
    state[i] = THINKING
    test(left(i))
    test(right(i))
    up(mutex)

function left(int i)
    return i-1 mod N

function right(int i)
    return i+1 mod N

```

4.15 Problema dei lettori e scrittori

Fino ad ora abbiamo supposto che gli accessi concorrenti sono sempre pericolosi: essere cauti è ok, ma la mutua esclusione su operazioni che non modificano le strutture dati comuni non è sempre necessaria! Due lettori possono tranquillamente operare assieme, ma uno scrittore può causare problemi. Questo problema modella l'accesso ad un database da parte di più lettori (che possono leggere in maniera concorrente) e scrittori (che possono aggiornare il database, ma in maniera mutualmente esclusiva, e solo quando nessuno sta leggendo).

4.15.1 Soluzione basata sui semafori

```
semaphore mutex = 1
semaphore db = 1
int rc = 0

function reader()
    while (true):
        down(mutex)
        rc = rc + 1
        if (rc = 1) down (db)
        up(mutex)
        read_database()
        down(mutex)
        rc = rc - 1
        if (rc = 0) up(db)
        up(mutex)
        use_data_read()

function writer()
    while (true):
        think_up_data()
        down(db)
        write_database()
        up(db)
```

4.16 Le altre soluzioni ai problemi giocattolo

Gli stessi problemi giocattolo, quali **produttore-consumatore**, **cinque filosofi** e **lettori-scrittori**, possono essere risolti usando approcci alternativi, che ho deciso di non riportare in questi appunti, ma che possono essere trovati all'interno delle slide fornite dal professore, relative all'argomento. Si invita quindi il lettore a vedere lì le soluzioni. Ho deciso comunque di fornire un codice d'esempio, scritto in C, relativo alla risoluzione del problema produttore-consumatoresu una coda FIFO statica, che potete trovare nella repository, o nelle pagine sottostanti.

4.16.1 Produttore-Consumatore in C

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include "lib-misc.h"

#define BUF_SIZE 10

/*
this is our shared buffer. it's a first-in first-out queue.
mutex and semaphores are needed to achieve mutual exclusion
*/
struct common_buffer{
    int buffer[BUF_SIZE];
    int count;
    sem_t full_count;
    sem_t empty_count;
    pthread_mutex_t mutex;
};
```

```

void thread_safe_queue_setup(struct common_buffer *buf) {

    for (int i = 0; i < BUF_SIZE; i++) {
        buf->buffer[i] = 0; /*initialize queue values at 0*/
    }

    buf->count = 0;
    sem_init(&buf->full_count, 0, 0);
    sem_init(&buf->empty_count, 0, BUF_SIZE);
    pthread_mutex_init(&buf->mutex, NULL);
}

void thread_safe_queue_destroy(struct common_buffer *buf) {
    sem_destroy(&buf->full_count);
    sem_destroy(&buf->empty_count);
    pthread_mutex_destroy(&buf->mutex);
}

void thread_safe_print_queue(struct common_buffer *buf) {

    pthread_mutex_lock(&buf->mutex);
    for (int i = 0; i < BUF_SIZE; i++) {
        printf("[ %d ]", buf->buffer[i]);
    }
    printf("\n");
    pthread_mutex_unlock(&buf->mutex);

}

void thread_safe_enqueue(struct common_buffer *buf, int element) {

    sem_wait(&buf->empty_count);
    pthread_mutex_lock(&buf->mutex);

    buf->buffer[buf->count] = element;
    buf->count++;

    pthread_mutex_unlock(&buf->mutex);
    sem_post(&buf->full_count);
}

int thread_safe_dequeue(struct common_buffer *buf) {

    sem_wait(&buf->full_count);
    pthread_mutex_lock(&buf->mutex);

    int out = buf->buffer[0]; /* return value */

    for (int i = 0; i < buf->count - 1; i++) {
        buf->buffer[i] = buf->buffer[i + 1]; /* moves each item forward by one slot */
    }

    buf->buffer[buf->count - 1] = 0; /* the last item in the queue becomes 0 */
    buf->count--;

    pthread_mutex_unlock(&buf->mutex);
    sem_post(&buf->empty_count);

    return out;
}

```

```

void *consumer_func(void * args) {

    struct common_buffer *shared_buffer = (struct common_buffer *)args;

    for (int i = 0; i < 500; i++) {
        thread_safe_dequeue(shared_buffer);
        thread_safe_print_queue(shared_buffer);
    }
    return NULL;
}

void *producer_func(void * args) {

    struct common_buffer *shared_buffer = (struct common_buffer *)args;

    for (int i = 0; i < 500; i++) {
        thread_safe_enqueue(shared_buffer, rand()%10+1);
        thread_safe_print_queue(shared_buffer);
    }
    return NULL;
}

int main() {

    int err;
    pthread_t consumer, producer;
    struct common_buffer shared_buffer;

    thread_safe_queue_setup(&shared_buffer);

    if ((err = pthread_create(&consumer, NULL, consumer_func, (void *)&shared_buffer)) != 0) {
        exit_with_err("pthread_create", err);
    }

    if ((err = pthread_create(&producer, NULL, producer_func, (void *)&shared_buffer)) != 0) {
        exit_with_err("pthread_create", err);
    }

    if ((err = pthread_join(consumer, NULL)) != 0) {
        exit_with_err("pthread_join", err);
    }

    if ((err = pthread_join(producer, NULL)) != 0) {
        exit_with_err("pthread_join", err);
    }

    thread_safe_queue_destroy(&shared_buffer);
}

```

Capitolo 5

Scheduler

5.1 Introduzione allo scheduling

In questo capitolo andremo a trattare il ruolo dello scheduler e i vari algoritmi di scheduling.

5.1.1 CPU burst

Indica fasi in cui la CPU viene utilizzata senza interruzioni di I/O.

5.1.2 Tipologie di processi

Gli algoritmi di scheduling agiscono in funzione dei processi che devono essere gestiti. Tra i processi, riconosciamo due categorie principali:

- Processi CPU-Bounded;
- Processi I/O-Bounded;

5.2 Obiettivi degli algoritmi di scheduling

Cambiano tra i vari ambienti: ambienti batch, interattivi e real-time.

5.2.1 Obiettivi comuni.

- **Equità** nell'assegnazione della CPU. Un algoritmo di scheduling deve essere, by design, equo verso tutti i processi.
- **Bilanciamento** d'uso delle risorse. Si deve evitare, quando possibile, di lasciare dei processi non operativi. Conoscendo

5.2.2 Obiettivi nei sistemi batch

Tre metriche da considerare nei sistemi batch, sono i seguenti.

- Massimizzare la produttività (il throughput).
- Minimizzare il tempo di completamento (**turnaround**).
Il tempo medio deve essere basso, ma la risposta deve rimanere affidabile. Bisogna bilanciare il tempo dedicato ai vari processi, in quanto osservare solo la metrica del **throughput** ci porterebbe a dare priorità ai processi brevi. Ciò causerebbe un aumento del tempo di completamento dei processi più lunghi.
Teniamo bene a mente che il tempo complessivo di ciascun processo non è influenzato dallo scheduler. L'importante è minimizzare il tempo di attesa.
- Minimizzare il tempo d'**attesa**.

5.2.3 Obiettivi nei sistemi interattivi

- Minimizzare il **tempo di risposta**.
Offrire un'esperienza quanto più responsiva.

5.2.4 Obiettivi nei sistemi real-time

- Rispetto delle scadenze (tempestività).
- Prevedibilità.

5.3 Scheduling nei sistemi batch

Ricordiamo che i processi sono situati in una struttura dati.

5.3.1 Soluzione 1 - First-Come First-Served

Basato sullo stesso principio della coda FIFO.

- **Non-preemptive.**

Il processo non può essere interrotto, e deve lasciare autonomamente la CPU.

5.3.2 Soluzione 2 - Shortest Job First

Detto anche scheduling per brevità.

- Non-preemptive.
- È una supposizione forte quella di conoscere il tempo richiesto da un processo per concludere il suo task. Nonostante ciò, nei sistemi batch è possibile fare una stima.
- Ottimale solo se i lavori sono tutti subito disponibili.
Se tutti i processi sono presenti a tempo 0, l'algoritmo è ottimale. Non essendo preemptive, processi più lunghi che entrano in coda prima avranno la priorità su processi più brevi entrati pochi istanti dopo.

5.3.3 Soluzione 3 - Shortest Remaining Time Next

Come il secondo, è definito come un "algoritmo per brevità".

- Versione preemptive dello SJF. Sfrutta la prelazione, ed è ottimo anche quando i processi vengono inseriti in coda.

Il context switch impiega del tempo trascurabile.

5.4 Scheduling nei sistemi interattivi

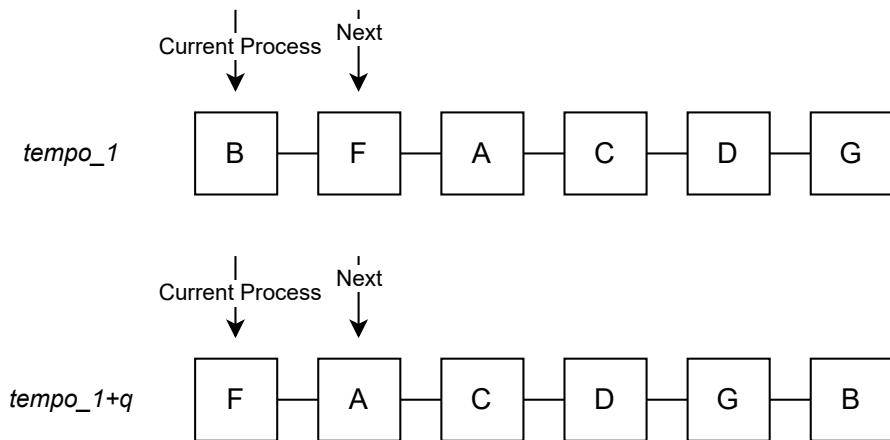
5.4.1 Soluzione 1 - Scheduling Round Robin

Versione con prelazione del FCFS.

Si impone un quanto di tempo (circa 20-50 millisecondi) che può essere concesso, a rotazione, ad ogni processo.

Con n processi e un quanto di q ms, ogni processo avrà diritto a circa $\frac{1}{n}$ della CPU e attenderà al più $(n-1)q$ ms.

Un processo su cui viene applicata la prelazione, non è bloccato: entra allo stato di wait, e torna nella coda dei processi.



Se un processo fa una chiamata bloccante, concederà la CPU al processo successivo in autonomia. Non è concepito un recupero del tempo della CPU perso per rilascio autonomo della CPU: il meccanismo deve rimanere semplice.

Un'altra osservazione importante: se il processo non rilascia la CPU in autonomia, si deve forzare il context switch. In questo caso, sarà un'operazione costosa con un determinato **overhead**.

Stabilire il quanto di tempo

- Con valori di q piuttosto alti, si riduce l'overhead, ma aumenta l'attesa: attese più lunghe riducono la probabilità che il context switch sia forzato.
- Con valori di q più piccoli, aumenta il rischio di far aumentare l'overhead, ma ogni processo attende di meno.

5.4.2 Soluzione 2 - Scheduling a priorità

Regola di base? Si assegna la CPU al processo con più alta priorità. I processi a priorità più bassa potranno usare la CPU solo se non ci sono in coda processi con priorità più alta. 3

Priorità?

- La priorità può essere statica o dinamica.
- Si favoriscono i processi I/O bounded.
I processi I/O bound sono processi che tendono a interrompersi autonomamente (senza overhead) prima della fine del quanto di tempo. I processi I/O bounded hanno brevi burst di CPU, quindi dare loro priorità riduce l'overhead.
- Shortest Job First come sistema a priorità.

Starvation, aging

Il fenomeno di **starvation** indica la possibilità che un processo non venga mai eseguito, magari a causa di processi pronti a priorità maggiore.

Per risolvere il problema, è possibile stabilire una regola, chiamata **aging**: se un processo rimane in una coda con priorità per x tempo, la sua priorità aumenta. Questo ogni x tempo finché non viene eseguita. Quando verrà eseguita, verrà ripristinata la sua priorità originale.

Prelazione o no?

Può essere reso preemptive o non-preemptive.

5.4.3 2.1 - Variante: scheduling a code multiple

Si usano più code, una per ciascuna classe di priorità. La regola è sempre scegliere la coda dalla priorità più alta non vuota (con processi bloccati o terminati). Chiamiamo questo **scheduling verticale**. Tra i vari livelli di priorità, come sceglio il processo da eseguire? Chiamiamo questo **scheduling orizzontale**. È possibile scegliere regole diverse per ogni livello, e quindi scheduler diversi.

Che scheduler orizzontale usare

Immaginiamo di dare una priorità maggiore ai processi I/O bounded e minore a quelli CPU bounded, e 4 livelli. Useremo round-robin su tutti i livelli, e stabiliremo dei quanti di tempo tali che

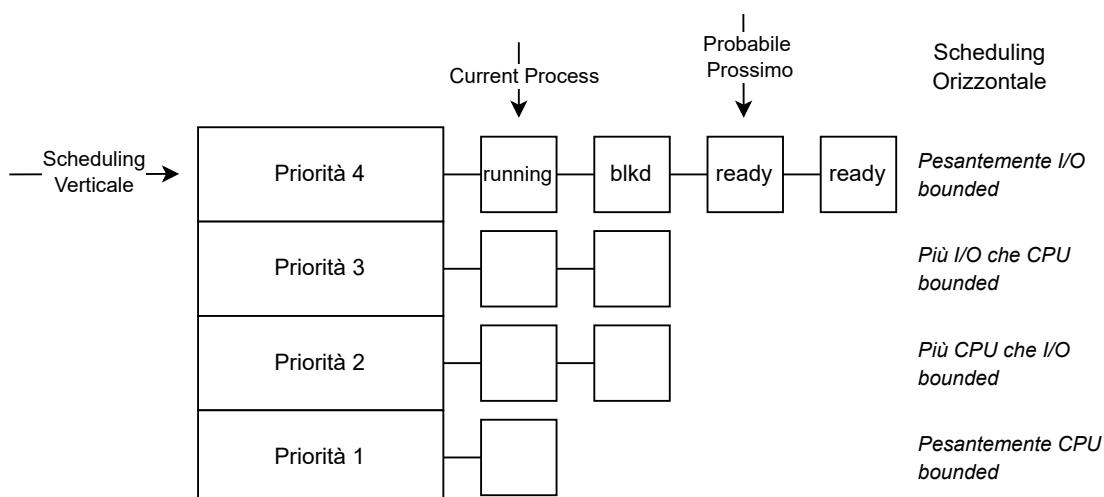
$$q_4 < q_3 < q_2$$

E il livello 1? Metteremo un quanto di tempo $q_1 = +\infty$, ovvero useremo un FCFS. Saranno processi quasi unicamente CPU-bounded, fanno operazioni in background e nessun tipo di interazione I/O. Usare questa strategia crea overhead 0.

E lo scheduling verticale?

L'aging è un'euristica con le sue possibili debolezze. Piuttosto che garantire SEMPRE che il processo a cui è concessa la CPU sia quello a priorità maggiore, possiamo usare un round-robin verticale con delle percentuali di CPU offerte ad ogni livello.

Livello 4, 60% della CPU. Livello 3, 20% della CPU. Livello 2, 15% della CPU. Livello 1, 5% della CPU.



5.4.4 Soluzione 3 - Shortest Process Next

Come si applica una strategia simile alla SJF in un contesto interattivo? Nel caso dei sistemi batch è facile conoscere la durata dei processi. Il nostro obiettivo sarà quello di creare delle stime sulla durata dei CPU burst, usando la media mobile.

$$S_{n+1} = S_n(1 - \alpha) + T_n \alpha$$

con $\alpha = \frac{1}{2}$, S_n n -esima stima, T_n n -esima durata effettiva.

Chiaramente, daremo priorità ai CPU-burst più brevi, che ricordiamo minimizzare l'overhead generale. Inoltre, $0 < \alpha < 1$.

- Un valore di α maggiore favorisce (in termini di peso) le misurazioni piuttosto che le stime, permettendo alla stima di adeguarsi più velocemente.
- Un valore di α minore favorisce (in termini di peso) le stime piuttosto che le misurazioni, permettendo alla stima di risentire meno di possibili picchi isolati nelle misurazioni.

5.4.5 4 - Altre soluzioni note

Scheduling garantito

Viene stabilita una percentuale di utilizzo e viene fatta rispettare¹.

Lo scheduler potrebbe tener conto del tempo residuo di ogni processo che ha autonomamente lasciato la CPU, o implementare meccanismi vari.

Scheduling a lotteria

Imbastire una sorta di lotteria, distribuire a dei processi n biglietti e estrarre randomicamente il processo a cui concedere la CPU. È un criterio chiaro, semplice. Lo scheduler può inoltre dare più biglietti ai processi con priorità maggiore. Questa soluzione è preemptive, in quanto a ogni biglietto viene associato un quanto di tempo. Inoltre, gli n biglietti vengono estratti fino a quando non finiranno, quindi vengono assegnati dei quanti di tempo prestabiliti. Finiti i biglietti, verranno riestratti,

Scheduling fair-share

Realizza un equo uso tra gli utenti del sistema. Non è un vero e proprio scheduler a se stante, ma è una funzione che si potrebbe considerare nel momento in cui si ha un sistema multi-utente. Un utente con più processi finirebbe per ottenere più tempo dalla CPU a causa del maggior numero dei processi. Un meccanismo fair-share permette una maggiore equità tra gli utenti.

5.5 Scheduling dei thread

5.5.1 Thread utente

- Sono ignorati dallo scheduler del Kernel.
- Per lo scheduler del sistema run-time vanno bene tutti gli **algoritmi non-preemptive** visti
- Possibilità di utilizzo di **scheduling personalizzato**

5.5.2 Thread del Kernel

- O si considerano i thread tutti uguali
- O si pesa l'appartenenza a un determinato processo.
Ricordiamo che il context switch tra thread dello stesso processo, è più leggero, a causa della MMU che non dovrà essere riprogrammata.

¹Una promessa simile era quella del round-robin con il suo q fisso: teniamo a mente però, che quando i processi abbandonano autonomamente la CPU, il tempo "residuo" che avrebbe potuto utilizzare, non è tenuto in considerazione.

5.6 Scheduling su sistemi multiprocessore

5.6.1 Alcuni possibili approcci

1 - Multielaborazione asimmetrica

Uno dei processori è master, e si occuperà dello scheduling, gli altri saranno slave. Un vantaggio di questo modello? L'accesso concorrente alle code dei processi non sarà mai un problema: non avviene!

Tuttavia, questo approccio non è ben scalabile, il core master può essere un bottleneck importante sul resto del sistema multi-processore. Praticamente inutilizzato :(.

2 - Multielaborazione simmetrica

Approccio più utilizzato dai sistemi operativi attuali. Ogni processore ha lo stesso ruolo, ma gestire la concorrenza non è facile. Le code potrebbero essere unificate o separate per ogni core.

- **Coda unificata.**

Implica la necessità di avere un lock. Ogni volta che lo scheduler di un core pescherà un processo dalla coda, verrà posto un lock sulla struttura dati fino a fine operazione. Questo porterà scheduler concorrenti a mettersi in coda. Il bilanciamento del carico non sarà richiesto.

- **Code separate.**

Code separate per scheduler di core separati, è la soluzione più usata. Non ci obbliga ad usare lock, anche se effettivamente sono richiesti (scopriremo più avanti il perché). Le code separate favoriscono la predilezione: indica la probabilità che un core si occupi più volte dello stesso processo (in momenti diversi). Questo ha come vantaggio un alto numero di cache hit, sempre ben accetti. Tuttavia, verificare e gestire il bilanciamento del carico nei vari core diventa obbligatorio.

5.6.2 Politiche di scheduling

Bilanciamento del carico

Una coda dei processi potrebbe essere molto più grande di un'altra. L'approccio a code separate è migliore, ma come facciamo a gestire questa circostanza?

- **Migrazione guidata.**

Una CPU qualsiasi verifica il carico sulle code del sistema. Se una coda è più piena delle altre, il carico viene distribuito sulle altre. Detto ciò, questo è il motivo dietro al lock richiesto nell'approccio a code separate.

- **Migrazione spontanea.**

Un core con presumibilmente meno-processi può autonomamente avviare un controllo sulle code degli altri core e decidere di prelevare alcuni dei processi per distribuire il carico. Va contro la predilezione, tuttavia può essere integrato un meccanismo di predilezione forte che forzi un core a occuparsi di un determinato processo o insieme di processi (sempre per favorire i cache hit).

5.7 E i nostri sistemi operativi?

5.7.1 Windows

- Usa uno scheduler basato su **code di priorità** separate.
- Sfrutta **euristiche** per il **migliorare** il servizio dei processi **interattivi** e in particolare di quelli in **foreground**. È importante ricordare che i sistemi Windows sono spesso sistemi desktop per macchine rivolte a utenti, e non per sistemi batch.
- Le euristiche sono anche mirate per **evitare** il problema dell'**inversione della priorità**.

5.7.2 MacOS

Simile a Windows, usa un Mach scheduler basato su code di priorità con euristiche.

5.7.3 Linux

- Linux non gestisce processi e thread, ma una loro generalizzazione chiamata **task**. Thread e processi diventano quindi casi particolari di questi **task**. Anche le `thread_create()` sono in realtà wrapper di funzioni relative ai task, come `clone()`.

- **Completely Fair Scheduler.** (CFS)

Sfrutta un albero rosso-nero che ha come chiavi il virtual run-time dei task. La CPU verrà data al processo col virtual run-time più breve (l'accesso all'elemento minimo dell'albero ha tempo $O(\log n)$). Quando l'elemento con virtual run-time più piccolo non è il processo corrente, viene effettuato un cambio di contesto. Per evitare problemi di overflow del virtual run-time, tutti i valori vengono normalizzati (viene sottratto un valore pari al virtual run-time minimo dell'albero). La priorità non è implementata in maniera tipica, ma tramite dei coefficienti (direttamente proporzionali alla priorità dei task) che moltiplicano il valore di `vruntime`, dettando quindi anche quanto tempo effettivo sarà dedicato ai vari processi. Questo metodo **non causa starving**.

Capitolo 6

La memoria

6.1 Memoria centrale e processi

La memoria RAM, rappresenta una tra le risorse fondamentali dei nostri calcolatori. La gestione della memoria centrale è un compito critico, che stabilisce il grado di multiprogrammazione concesso all'utente, ed è affidato al sistema operativo. In questo capitolo, andremo a focalizzarci sulle varie soluzioni ideate, create e perfezionate al fine di ottenere un sistema multiprogrammato.

6.2 Nessuna astrazione

I primi mainframe e i primi sistemi, come MS-DOS, non contemplavano l'esecuzione di più di un processo. Avere un singolo processo in memoria, significa non avere l'esigenza di proteggere lo spazio di memoria di ciascun processo. **Si può lavorare direttamente con gli indirizzi fisici.**

- I programmi conoscono e **utilizzano direttamente gli indirizzi fisici.**
- Difficilmente possono essere eseguiti due programmi contemporaneamente. **Non si prestano alla multiprogrammazione**, a causa di possibili conflitti di indirizzamento, e del rischio di sovrascrizioni di dati e/o codice.

6.2.1 Alcune varianti senza astrazione

Tra i primi PC, si sono visti sistemi di vario tipo, che si differenziano tra loro per la memoria dedicata al sistema operativo e ai driver di avvio.

- **Sistema operativo in RAM.**
In cui è possibile accedere al codice e ai dati del sistema operativo.
- **Sistema operativo in ROM.**
Le strutture dati del sistema operativo rimangono ancora in RAM, ma il codice è in una read-only-memory.
- **Driver d'avvio in ROM, sistema operativo in RAM.**

Il resto dello spazio sarà dedicato a uno o (improbabile...) più programmi utente.

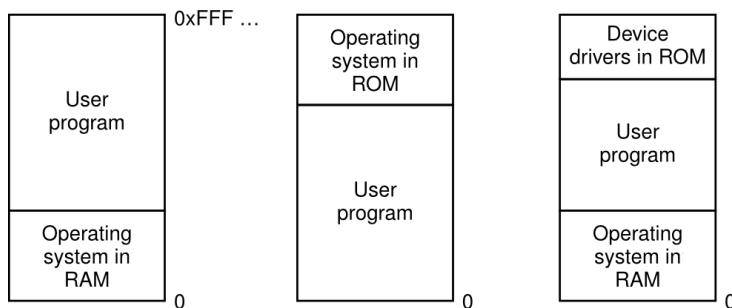


Immagine 6.1: Sistema operativo in RAM, IN ROM e driver di avvio in ROM

6.2.2 Multiprogrammazione senza astrazione, perché è impossibile

Supponiamo, per una sorta di masochismo, di voler effettuare **multiprogrammazione** su una macchina che lavora esclusivamente su **indirizzi fisici**. Gli n processi saranno caricati in **partizioni di memoria differenti**. Osserviamo questo esempio:

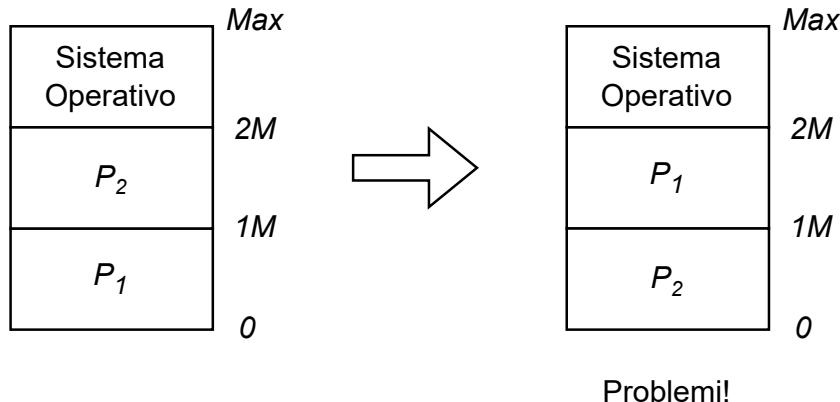


Immagine 6.2: Rilocazione in memoria. A destra, i range di indirizzi.

I programmi P_1 e P_2 conosceranno **indirizzi fisici, e quindi assoluti**. Lavorando su indirizzi assoluti, un cambio di posizione in memoria da parte del codice potrebbe richiedere la totale riscrittura degli indirizzi usati nel codice.

La rilocazione come soluzione

Il **loader** è un componente che può gestire automaticamente questa circostanza, sommando o sottraendo un opportuno offset ad ogni indirizzo del codice che si vuole caricare/compilare. Ciò però non protegge la memoria dedicata ai processi, dai processi stessi. P_1 , P_2 potrebbero manipolare locazioni di memoria critiche per i processi, o sovrascrivere il codice dei programmi.

Una soluzione originale - IBM 360

Una soluzione hardware utilizzata dall'IBM 360 prevede una chiave di protezione per accedere a delle aree di memoria.

Storage protection, made up of the store and fetch" protection features, prevents the unauthorized changing or use of the contents of main storage. Store protection prevents the contents of main storage from being altered by storage addressing errors in programs or input from I/O devices. Fetch protection prevents the unauthorized fetching of data and instructions from main storage.

As many as 15 programs (with associated main storage areas) can be protected at one time.

Protection is achieved by dividing main storage into 2,048-byte blocks and by associating a five-bit storage key with each block. Each storage. key may be thought of as a lock. Each block of storage, then, has its own "lock". Two instructions are provided for assigning and inspecting the key, which contains a four-bit code.

The same code may be used by many blocks, using binary codes 0001-1111.

6.3 La prima astrazione - Spazio degli indirizzi

La prima astrazione che introduciamo, è quella dello spazio degli indirizzi. I processi lavoreranno su indirizzi logici, e conserveranno due registri: il registro base e il registro limite.

È la Memory Management Unit, ovvero la MMU, a fare il controllo degli indirizzi logici e la rilocazione dinamica ad ogni context switch.

Il controllo della protezione?

1. Si controlla se l'indirizzo logico è più piccolo del registro limite.
2. Se il controllo è più grande del registro limite, eccezione:
trap - addressing error.
3. Altrimenti, si somma il registro base a quello logico, ottenendo il registro fisico.

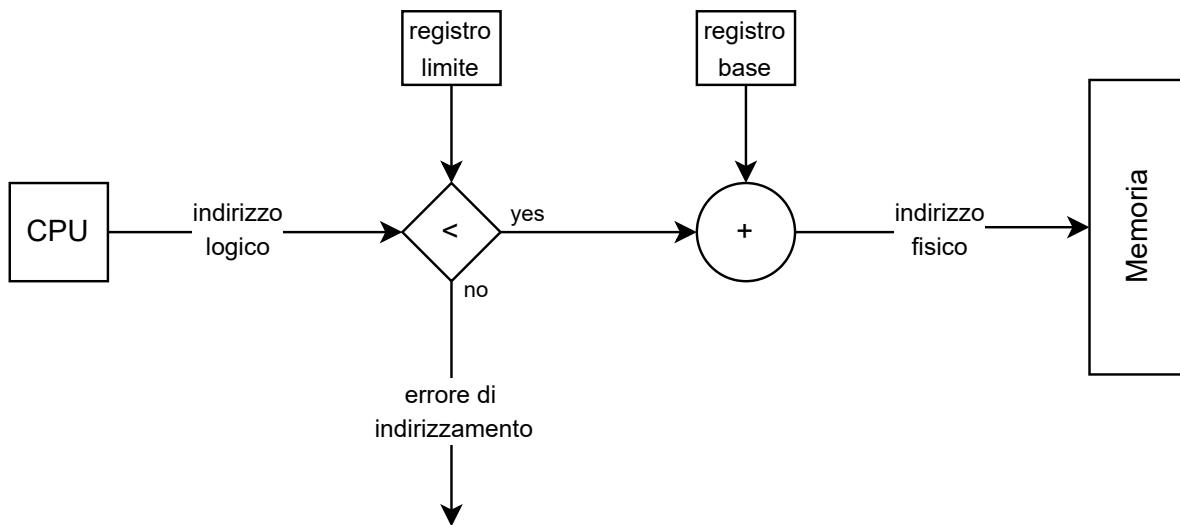


Immagine 6.3: Il processo della MMU

Offre quindi una **protezione della memoria dei processi** (tramite registro limite) e **traduzione degli indirizzi logici in indirizzi fisici** (tramite registro base). È un'ottima astrazione per sistemi multiprogrammati. Tuttavia, la memoria è limitata nello spazio, ed è per questo che andremo ad introdurre una nuova tecnica, lo **swapping**.

6.3.1 Multiprogrammazione con spazio degli indirizzi - Swapping

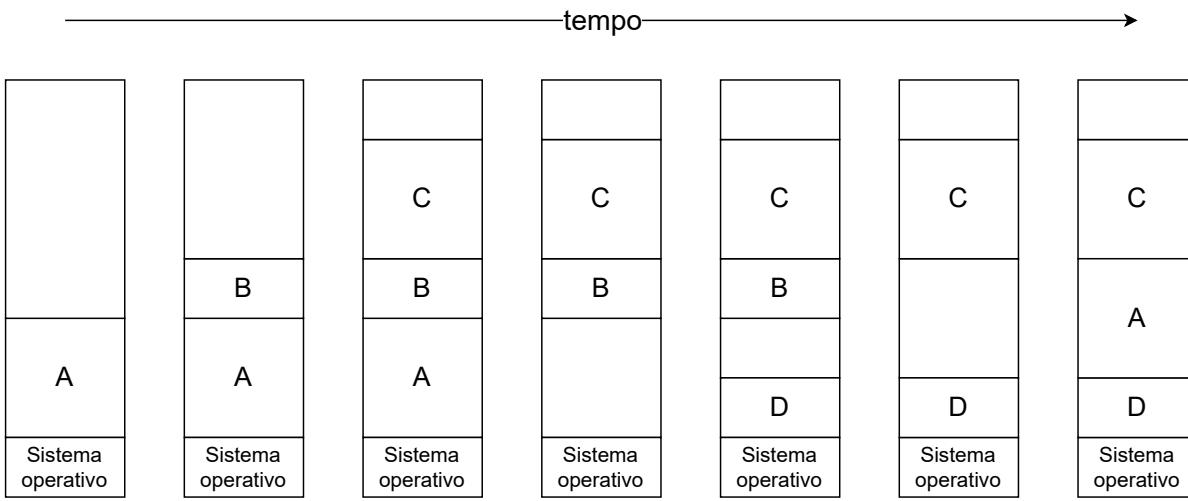
Il livello di multiprogrammazione è seriamente limitato dalla dimensione della memoria centrale: aggiungiamo una nuova tecnica al nostro sistema, lo **swapping**. Permette di ovviare al limite di multiprogrammazione, causato dalla memoria centrale di dimensione limitata.

In cosa consiste lo swapping?

Dati due programmi P_1 , P_2 , con P_1 in esecuzione, desideriamo eseguire P_2 . Lo swapper effettua uno swap-out di P_1 e uno swap-in di P_2 dalla e verso la memoria centrale.

Il processo portato fuori dalla memoria centrale, viene chiaramente penalizzato, in quanto non ha modo di ricevere risorse.

Il compito di scegliere quali processi *swappare* è affidato allo **swapper**, chiamato anche **scheduler di medio termine**. La memoria usata come appoggio per lo swap, è detta memoria secondaria, e tipicamente è il disco.



I criteri con cui i processi vengono swappati verso la memoria centrale, non possono basarsi soltanto sullo spazio che occupa il codice. Le esigenze di memoria di un processo possono crescere durante l'esecuzione: le strategie di allocazione possono contemplare allocazione di dimensione fissa o dinamica.

6.3.2 Problemi dello swapping

Lo swapping è una strategia fondamentale, ma ha delle criticità che bisogna conoscere, per andare a capire le esigenze di introdurre ulteriori meccanismi, sopra un sistema apparentemente perfetto.

Problema 1 - dimensione dell'allocazione

Quanta memoria andiamo ad allocare per un processo che dobbiamo inserire in memoria?

Problema 2 - frammentazione

Un problema di questa strategia è quello relativo alla frammentazione della memoria,

- **Interna.**

Avviene quando un'allocazione per un processo eccede dalle esigenze effettive. *Spazio inutilizzato in una zona allocata.*

- **Esterna.**

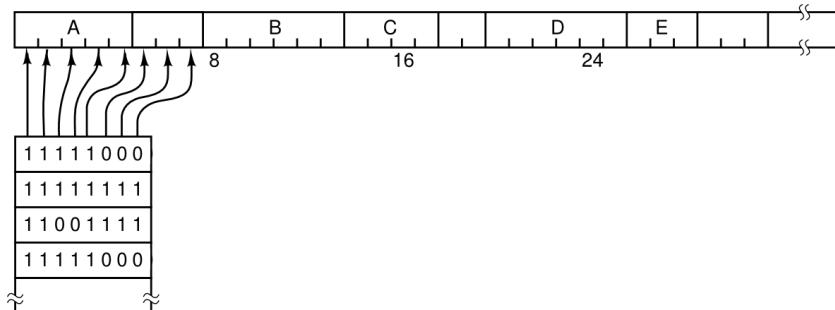
Avviene quando la memoria libera è disponibile ma distribuita su numerosi blocchi non contigui, rendendo difficile l'allocazione. La soluzione è la **memory compaction**. *Piccole zone frammentate non allocate ma quasi inutilizzabili.*

Problema 3 - Come gestire l'allocazione

La memoria viene **suddivisa in blocchi di dimensione arbitraria e fissa** dal Sistema Operativo. Avere troppa granularità di memoria, porta solo complicazioni, anche se permetterebbe di ridurre la frammentazione interna.

- **Strategia a bitmap.**

Avrà tanti bit quanti sono i blocchi allocati. Il k -esimo bit specifica se il k -esimo blocco è occupato (1) o libero (0). Il numero di bit della bitmap è inversamente proporzionale alla dimensione dei blocchi.



La bitmap dovrebbe essere allocata in RAM: è fondamentale valutare qual è la dimensione migliore per i blocchi, anche in funzione di quanto dovrebbe essere grande la bitmap.

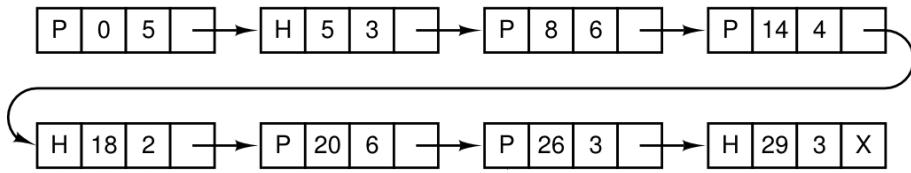
- **Liste dei blocchi liberi e occupati.**

Ogni elemento della lista (doppiamente concatenata) contiene al suo interno un numero di blocchi e un elemento di tipo P (per processo) o H (per hole, buco). I nodi della lista sono ordinati per indirizzo.

Quando devo allocare dello spazio per un processo P_1 , possono avvenire due cose:

- Se ho uno spazio di dimensione esattamente uguale al processo P_1 , sostituisco H con P .
- Altrimenti, sostituisco un nodo con due nodi, uno di tipo P , uno di tipo H .

Quando due nodi adiacenti sono del tipo H o P , questi vengono sostituiti da un unico nodo. Una variante di questa strategia, sfrutta liste separate per blocchi liberi H e blocchi di processi P . Ordinando le liste di blocchi liberi rispetto al loro spazio in memoria (in ordine crescente), è possibile adottare strategie più intuitive per trovare il blocco migliore per soddisfare una richiesta.



Algoritmi di ricerca per allocazione su liste di blocchi

Con P_1 processo da allocare in memoria centrale

- **First fit.**

Il primo blocco di dimensione $\geq P_1$

- **Next fit.**

Strategia analoga al first fit, ma ogni ricerca riprende dal punto in cui si è interrotta la ricerca.

- **Best fit.**

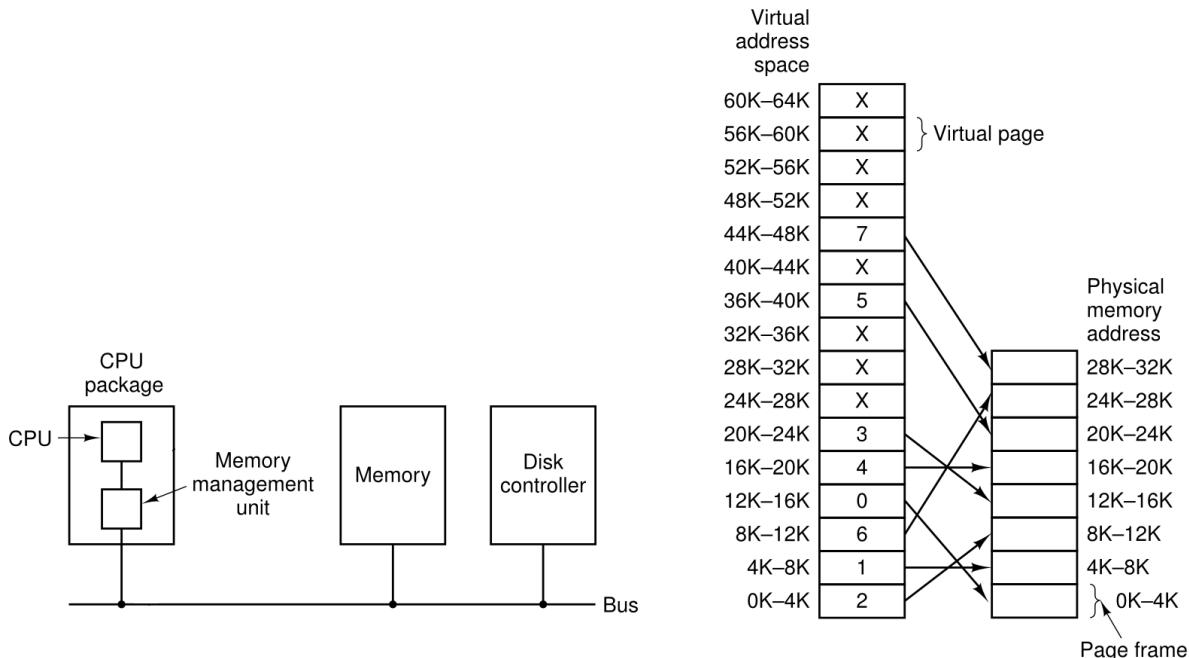
Cerca la migliore scelta, minimizzando lo spazio residuo da un'allocazione, richiedendo tempo $O(n)$. Questa strategia crea frammenti molto piccoli, quindi la strategia best fit non è proprio ottimale.

- **Worst fit.**

Allocà sempre il blocco di dimensione più grande, cercando di massimizzare la dimensione dello spazio residuo.

6.4 Soluzione definitiva alla frammentazione - Memoria Virtuale

Risolviamo i problemi di frammentazione con l'astrazione definitiva: la **memoria virtuale**. L'obiettivo di questa astrazione, è quello di illudere ogni programma di disporre di un'intera memoria centrale privata, continua e protetta, e di una CPU dedicata. Lo spazio di indirizzamento diventerà **spazio di indirizzamento virtuale**. Questo spazio tuttavia non esiste: la dimensione virtuale andrà ben oltre quello della memoria fisica.



La memoria virtuale è gestita dall'MMU, tramite le tabelle delle pagine, di cui parleremo meglio nel paragrafo relativo alla paginazione.

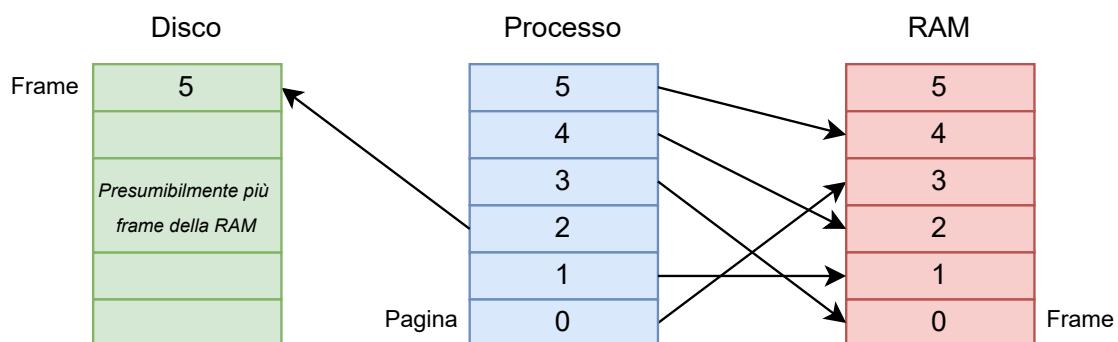
6.4.1 Spazio di indirizzamento

I sistemi a 32 bit dedicano 32 bit per l'indicizzazione della memoria virtuale, permettendo il riferimento a 2^{32} indirizzi di memoria, ossia 4 Gigabyte. I sistemi a 64 bit, offrono un totale molto più vasto di 2^{64} indirizzi.

Questo spazio di indirizzamento viene gestito in porzioni di dimensione fissa (di taglia, solitamente, 4Kb), chiamate **pagine virtuali**. Nonostante però la memoria virtuale sia virtualmente più grande, solo parte del contenuto a cui fa riferimento la memoria virtuale, può essere situato in RAM.

6.4.2 Frammentazione (esterna) is no more

La memoria virtuale, che **memorizza in pagine contigue indirizzi di frame non contigui**, risolve il problema della frammentazione esterna, in quanto la **contiguità dei frame non è richiesta**. Inoltre, **porzioni del codice** possono non essere situate in RAM. Se il sistema operativo permette, è possibile mantenere parti di codice all'interno del disco, ovvero la memoria secondaria.



Page fault

Questa astrazione crea però una circostanza da gestire: che fare se diventa necessario prelevare una risorsa che è nello spazio di indirizzamento virtuale, ma non è in memoria centrale? Chiameremo questa circostanza **page fault**. Il **sistema operativo** procederà alla risoluzione del problema, gestendo questa **eccezione** (di tipo *trap*), e portando in **RAM** tutti i **frame richiesti**. La scelta dei frame da inserire sul disco piuttosto che in RAM, è una vera e propria **scommessa del sistema operativo**. **Valutazioni errate implicano overhead**, ma inserire dati non necessari in un determinato istante di tempo sul disco piuttosto che sulla RAM, ci permetterà di gestire alla perfezione più processi.

E la frammentazione interna?

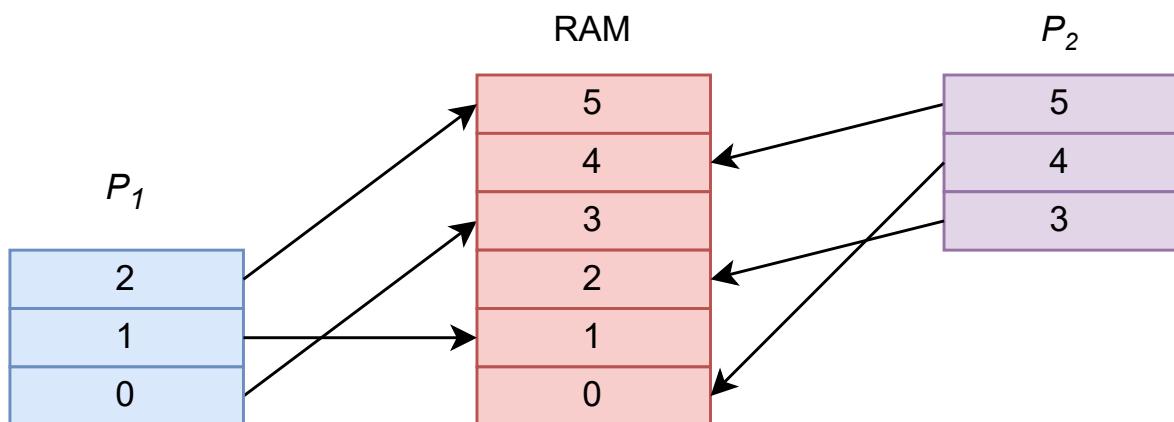
La dimensione prefissata dei frame può causare ancora problemi di frammentazione interna. Sistemi operativi come Solaris gestiscono frame di dimensione variabile, minimizzando il problema. Tuttavia, tra la frammentazione interna e quella esterna, la prima rappresenta il male minore.

Cosa otteniamo in termini di prestazioni?

La gestione dello spazio indirizzi tramite memoria virtuale, aumenta esponenzialmente il numero di processi eseguibili! Mantenendo il *bare minimum* di ogni processo in memoria centrale, aumenteremo il grado di multiprogrammazione, riducendo di veramente poco le prestazioni per ogni processo.

I processi sono protetti tra loro?

Sì! Sia gli indirizzi di memoria virtuale che di memoria fisica di più processi, saranno implicitamente isolati. Un processo conosce solo gli indirizzi virtuali (e fisici) della propria tabella delle pagine.



P_1 conosce solo i propri indirizzi. Lo stesso vale per P_2 , ottenendo così una protezione implicita

6.5 Paginazione

Definiamo la paginazione come una tecnica di suddivisione e di gestione, da parte del sistema operativo, della memoria. Lo spazio virtuale viene suddiviso in pagine, o pagine virtuali. La memoria fisica viene suddivisa in frame.

Le pagine virtuali hanno la stessa dimensione dei frame.

6.5.1 MMU

La paginazione è **gestita dall'MMU**, che associerà agli **indirizzi virtuali** dello spazio degli indirizzi, gli **indirizzi fisici** nella **memoria centrale**.

Non sempre l'MMU riuscirà a tradurre l'indirizzo virtuale in indirizzo fisico, in quanto, spesso e volentieri, i blocchi richiesti saranno nel disco e non in memoria centrale. Questo fenomeno è chiamato page fault, ed è una circostanza totalmente regolare e gestita dal sistema operativo.

6.5.2 Tabella delle pagine

Come fa l'MMU a capire quale informazione è in memoria centrale, e quale su disco? Ad ogni processo sarà dedicata una tabella delle pagine. n processi $\Rightarrow n$ tabelle delle pagine. A ogni pagina di memoria virtuale è associato un record nella tabella delle pagine.

- **Un bit di presenza.**

Indica se la pagina è presente in memoria centrale. Se sì, bene. Altrimenti? Page fault! Chiameremo il sistema operativo con una trap.

- **Indirizzo fisico** del frame corrispondente.

Ne sono presenti anche altri, che affronteremo al dettaglio dopo. La tabella delle pagine di un dato processo, conterrà un numero di righe pari al numero di pagine dello spazio logico degli indirizzi di quel processo.

Traduzione degli indirizzi

Supponiamo di avere una RAM a 32 kilobyte, e di voler accedere all'indirizzo virtuale $v = 8196$. Con dimensione dei frame = 4096, effettuiamo:

$$\frac{8196}{4096} = 2 \text{ col resto di } 4$$

2 sarà il numero di pagina virtuale, 4 sarà l'offset relativo al numero di pagina. Supponiamo che nella nostra tabella delle pagine, la pagina 2 è associata al frame 6. Moltiplichiamo adesso $6 \cdot$ dimensione dei frame = $6 \cdot 4096 = 24576$. Otterremo quindi il primo indirizzo fisico appartenente al frame 6. Sommiamo l'offset ottenuto in precedenza, $24576 + 4 = 24580$.

E che fa il calcolatore?

Il calcolatore effettua questo processo di traduzione in maniera più veloce e intuitiva. La divisione è sempre di potenze di 2, in quanto strettamente legata alla dimensione dei frame. Sarà quindi possibile implementarla con uno shift. Con un frame di $2^{12} = 4096$ bit, i primi 12 bit a destra saranno quelli del resto, ovvero l'offset. I restanti, a sinistra, saranno il quoziente, ovvero l'indirizzo del frame.

```
// indirizzo fornito
10000000000100

10 000000000100

q = 10 = 2 [tradotto poi in 6]
r = 000000000100 = 4

6 * dimensione dei frame =
= 6 * 4096 =
= 110 << 12 = 1100000000000000

OR logico tra il resto e il valore appena ottenuto

1100000000000000 | 100 = 110000000000100 = 24580
```

Esercizio in aula

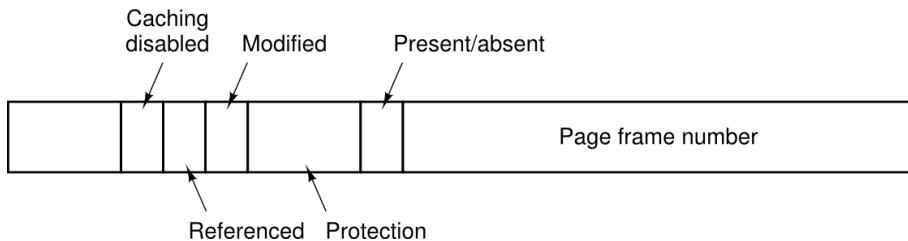
- Virtual memory = $4Mb$
- #voci tabella pagine = 2^{13}
- Numero di frame a 8bit

Quanta RAM ho? Avendo $4Mb = 22\text{byte}$ di virtual memory, con $\frac{2^{22}}{2^{13}}$ avrei la dimensione della singola pagina, ovvero 2^9 .

$$2^9\text{byte} \cdot 2^8 \text{ (numero di frame)} = 128kb$$

6.5.3 Dettagli sui record delle tabelle delle pagine

L'hardware stabilisce la struttura dei record delle tabelle delle pagine. L'MMU "conosce" la struttura dei record di questa tabella.



- **Numero del frame.**
- **Bit presente o assente.**
Per determinare se avverrà un page fault.
- **Protezione - RWX.**
Lettura, scrittura ed eventualmente esecuzione. Sono tre flag che specificano i permessi relativi a quel determinato record. Un record di sola lettura, come il codice di un programma non modificabile, può essere messo in modalità di sola lettura. Due istanze dello stesso programma possono fare riferimento allo stesso codice, evitando di conservare in memoria dati duplicati.
Con il flag di esecuzione X è possibile specificare se del codice è da seguire. È una contromisura alla *code injection*.
- **Dirty bit.**
Indica se la pagina è stata modificata. Le modifiche fatte alla pagina devono essere propagate nel disco, causando overhead quando si rimuove il frame corrispondente.
- **Bit referenziato.**
Impostato quando si fa un riferimento qualunque alla pagina. Viene resettato periodicamente.
- **Bit per disabilitare la cache.**
In alcuni casi si vorrebbe ridurre il rischio di leggere dati non aggiornati, come nel caso dell'I/O mappato in memoria.
- **Bit di validità.**
O di allocazione.

6.5.4 Tabella dei frame

La controparte, relativa ai frame fisici, della tabella delle pagine. Il sistema operativo **tiene traccia dello stato** di occupazione di ogni **frame fisico** attraverso la **tabella dei frame**. Questa tabella conterrà per ogni frame:

- Lo **stato** ((occupato/libero)).
- Se occupato, specificherà **da quale processo**.

Questa tabella viene tipicamente consultata in due circostanze principali

- Ogni volta che viene **creato un nuovo processo**, per creare la relativa tabella delle pagine di quel processo.
- Ogni volta che un processo chiede di allocare **nuove pagine**.

6.6 Progettazione di una tabella delle pagine - velocità

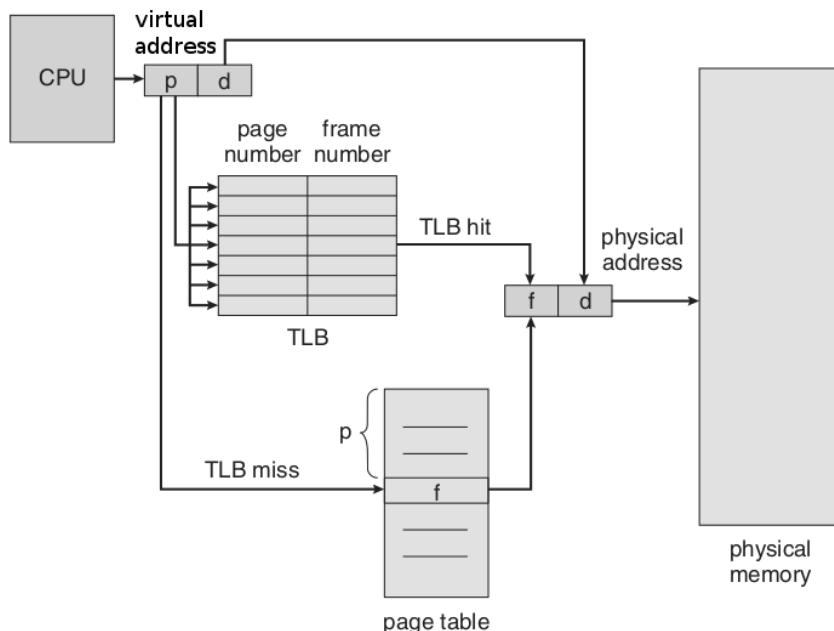
La traduzione degli indirizzi è un'operazione critica. A causa della sua frequenza (per alcune istruzioni potrebbero servire fino a 5/6 traduzioni), bisogna implementare un meccanismo di traduzione quanto più veloce possibile.

Nei sistemi moderni, la tabella delle pagine di ogni processo è memorizzata in RAM, e utilizza un registro chiamato **Page Table Base Register** per puntare all'inizio della tabella. Questa tabella è allocata in maniera contigua ed è quindi indicizzabile.

6.6.1 Memoria associativa - TLB

Un programma, in genere, fa un **gran numero di riferimenti** ad un **piccolo numero di pagine**. L'alto numero di traduzioni richieste da un processo, saranno l'idea dietro la prossima soluzione.

Andremo a implementare un meccanismo di caching, tramite una nuova memoria: stiamo parlando della **Translation Lookaside Buffer**, detta anche **memoria associativa**. È situata tra l'MMU e la tabella delle pagine.



6.6.2 Contenuto di un record TLB

La TLB contiene dai 64 ai 1024 record.

- **Numero di pagina (virtuale).**
- Bit per la **validità della voce** della TLB.
- **Codice di protezione.**
Specifica i permessi di accesso (lettura/scrittura/esecuzione).
- **Dirty bit.**
Quando avviene una modifica ad un record, questo viene marchiato come dirty. Quando questo succede, quel record della TLB dev'essere eliminato, verrà portata la modifica alla tabella delle pagine.
- **Numero di frame (fisico).**

6.6.3 Conseguenze

La ricerca in maniera parallela tramite hardware specializzato (in tempo costante) permette di velocizzare l'associazione indirizzo virtuale - fisico. Consultando la memoria associativa, possiamo ottenere due esiti: **TLB hit** o **TLB miss**. Notiamo (dal disegno) che quando si ha una TLB hit, si ottiene l'**indirizzo fisico del frame**. L'offset è preso direttamente dall'indirizzo virtuale. Ricordiamo che, con dimensione dei frame 2^n , l'offset è stabilito dalle n cifre meno significative.

6.6.4 Osservazioni importanti

- Alcuni attributi dei record delle tabelle delle pagine non sono presenti, come il bit di referenzialmento o di validità di allocazione.
Osserviamo che, se un record è presente nella TLB, dev'essere per forza valido e referenziato.
- La TLB conterrà sempre record relativi a pagine utili per il programma attuale. Inoltre, quando avviene una TLB miss, si andrà ad aggiornare il valore sulla TLB.
- Alcuni voci possono essere vincolate ad essere nella TLB, migliorando le tempistiche.

6.6.5 Flush e address-space id

Le tabelle delle pagine sono una per processo: ne consegue che gli stessi indirizzi virtuali possono essere riutilizzati per rifarsi a indirizzi fisici differenti. Tuttavia, la TLB è unica per la CPU (o il core). Detto ciò, a causa del riutilizzo degli indirizzi virtuali, bisogna avere un modo per disambiguare i record riguardanti ogni processo.

Due strategie:

- **Flush totale della TLB a ogni context-switch.**
Non ideale, favorisce un alto numero di cache miss iniziali.
- **Address-space ID.**
Aggiungendo un identificatore, chiamato ASID, è possibile disambiguare le voci delle TLB con lo stesso indirizzo virtuale, ma appartenenti a processi differenti, favorendo cache hit ad ogni context switch.

La strategia migliore è la seconda. Complica un po' la ricerca, aggiungendo gli ASID alla ricerca del record nella TLB, ma è molto efficiente in ambienti multithread. È un ottimo compromesso.

6.6.6 Effective Access Time (EAT)

Facciamo un esempio

- tempo di accesso alla memoria: $\alpha = 100 \text{ ns}$
- Tempo di accesso alla TLB: $\beta = 20 \text{ ns}$

I tempi effettivi?

- Con un TLB hit abbiamo $100 + 20 = 120 \text{ ns}$.
- Con un TLB miss abbia o $2 \cdot 100 + 20 = 220 \text{ ns}$.

Con un ratio $\text{TLB} = \varepsilon$, tempo di accesso alla TLB = β e tempo di accesso alla memoria α abbiamo

$$\mathbf{EAT} = \varepsilon(\alpha + \beta)(1 - \varepsilon)(2\alpha + \beta)$$

6.6.7 Gestione TLB

Alcune macchine RISC caricano all'interno dell'os i record TLB. Quando si verifica un TLB miss, non sarà la MMU a recuperare le pagine, bensì si genera una TLB fault e il sistema operativo si occuperà del fetch. Quando si gestisce la TLB via software, distinguiamo due tipi di miss:

- **Soft miss.**
Avviene quando la pagina ricercata risiede in memoria. Risolvibile in una ventina d'istruzioni, pochi nanosecondi.
- **Hard miss.**
Avviene quando la pagina ricercata è nel disco. L'accesso al disco richiede più tempo. TLB miss + page fault.

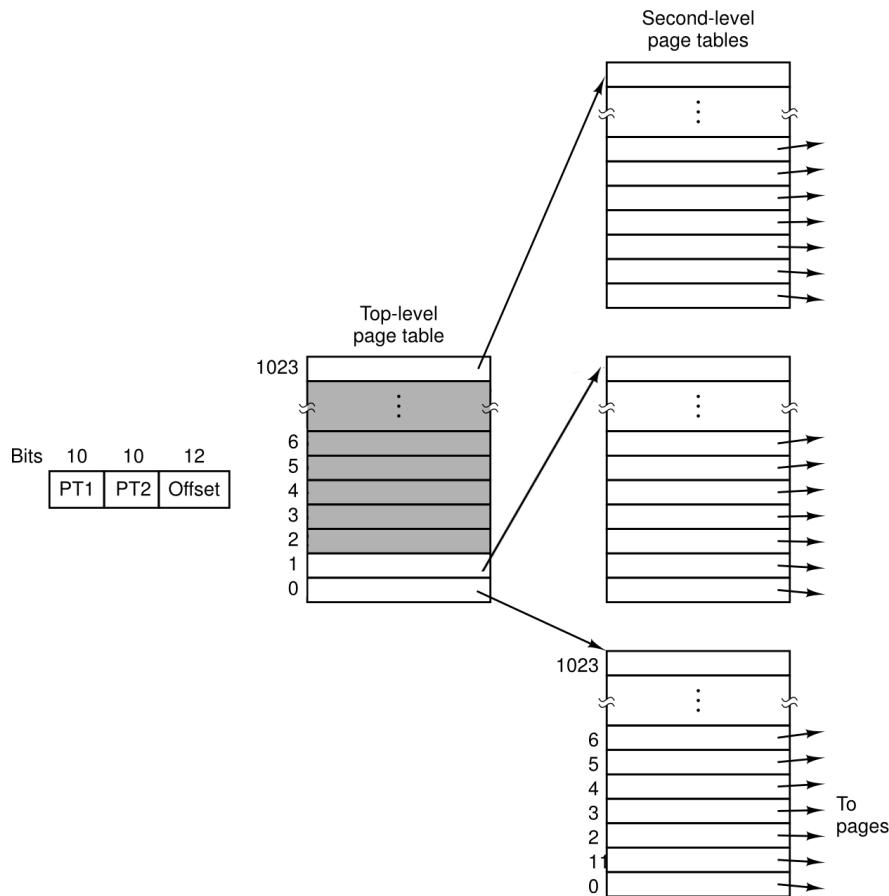
6.7 Progettazione di una tabella delle pagine - dimensioni

In un sistema a 32 bit, una tabella delle pagine pesa circa $4 MB$, e richiede di essere memorizzata in maniera contigua. Non è ottimale già a 32 bit, figuriamoci nei sistemi a 64 bit con

Osserviamo che, con indirizzi a 32 bit, si hanno 12 bit di offset. Servono quindi 2^{20} voci da 32 bit, ovvero 4 byte, confermando i circa $4 MB$ di tabella delle pagine.

6.7.1 Soluzione -Tabella delle pagine multilivello

Non mantenere l'intera tabella in memoria. Sfruttare delle tabelle multilivello, ottenendo una paginazione gerarchica.

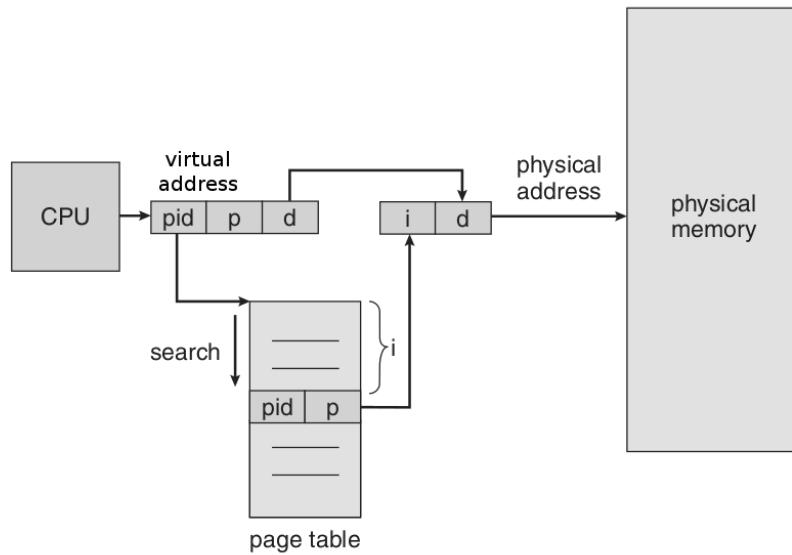


In memoria centrale non verrà memorizzata l'intera tabella delle pagine, bensì delle tabelle contenenti indirizzi ad altre tabelle. Questo fa crescere il numero di accessi necessari alla memoria in funzione dei livelli della tabella.

I bit non dedicati all'offset, possono essere suddivisi in sottosequenze per gestire delle tabelle delle pagine multilivello.

6.7.2 Tabella delle pagine invertita

È un approccio alternativo, al giorno d'oggi quasi estinto. Una voce per ogni frame fisico, in cui ogni voce riporta [id processo, pagina virtuale]. Otterremo una tabella molto più piccola, e unica per tutto il sistema.



Scopriremo poi che questa strategia non è ottimale per molteplici motivi: pagine condivise inutilizzabili, lentezza nella ricerca e altro.

6.7.3 Cache della memoria vs Memoria virtuale

La cache della memoria può essere basata sugli indirizzi fisici, o sugli indirizzi virtuali. Ciò dipenderà rispettivamente, se posizioniamo l'MMU prima o dopo la cache.

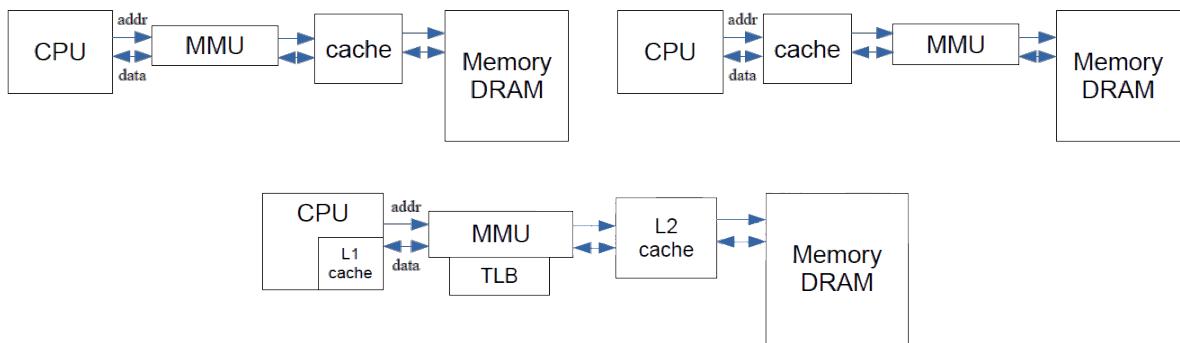
- **Indirizzi fisici.**

Pro: non servirà invalidare la cache sul context-switch. Avendo indirizzi fisici già tradotti nella cache, non si avranno problemi di ambiguità.

Contro: l'MMU diventa un collo di bottiglia per la cache.

- **Indirizzi virtuali.**

Pro: più veloce ed efficace Contro: servono gli ASID per non invalidare la cache.



Soluzione effettiva - approccio ibrido

La cache L1 basata su indirizzi virtuali, si pone prima dell'MMU, basandosi sugli indirizzi virtuali.

La cache L2 e successive dopo l'MMU, basate su indirizzi fisici.

6.8 Algoritmi di sostituzione delle pagine

In caso di **page fault**, nasce l'esigenza di gestire il mettere in memoria la pagina richiesta e non presente in RAM. Se la memoria centrale è piena, bisogna valutare una **pagina vittima** da rimuovere per ospitare la pagina richiesta. Considera i page fault come errori fatali sarebbe sbagliato: **è una regolare circostanza** che avviene all'interno di un sistema.

Il problema della sostituzione delle pagine

Similmente al problema della **gestione della cache**, ci poniamo come obiettivo quello di scegliere il frame ottimale. La scelta ottima, minimizza il numero di page fault in futuro, cercando di **minimizzare l'overhead**, senza creare page fault futuri.

6.8.1 Soluzione teorica - OPT

La scelta cade sulla pagina che verrà referenziata quanto più tardi possibile. È un **algoritmo teorico**, cui realizzazione è difficile. Sarà il nostro **termine di paragone** per le altre soluzioni effettivamente realizzabili.

6.8.2 Algoritmo NRU - Not Recently Used

Raccoglie delle statistiche sulle pagine caricate, sotto forma di due bit.

- Bit di referenziamento (R).
- Bit di modifica (M).

Questi bit sono tipicamente aggiornati in hardware, e azzerati dal sistema operativo. L'azzeramento del bit di referenziamento avviene periodicamente.

Introducirà quindi quattro classi di pagine:

- **Classe 0.**

Non referenziato, non modificato. R: 0, M: 0.

Prima dell'ultimo azzeramento, le pagine di questa classe non sono né state modificate, né referenziate.

- **Classe 1.**

Non referenziato, modificato. R: 0, M: 1.

Dare priorità ad una pagina pulita, piuttosto che a una pagina sporca, nasce dall'overhead che si introduce andando ad eliminare una pagina "sporca" dalla memoria centrale. Prima dobbiamo effettuare la scrittura sul disco.

- **Classe 2.**

Referenziato, non modificato. R: 1, M: 0.

- **Classe 3.**

Referenziato, modificato. R: 1, M: 1.

Il sistema operativo preferirà come **pagina vittima**, la prima pagina non vuota dalla classe più bassa. Lo storico di informazioni che il sistema operativo valuterà per capire che pagina eliminare, inizia dall'ultimo azzeramento.

Priorità a R:0 - M:0

- Il bit di referenziamento da informazioni sull'utilizzo della pagina dall'ultimo azzeramento.
- Il bit di modifica da informazioni relative a quello che sarebbe l'overhead della sostituzione della pagina dalla memoria centrale al disco.

A parità di classe?

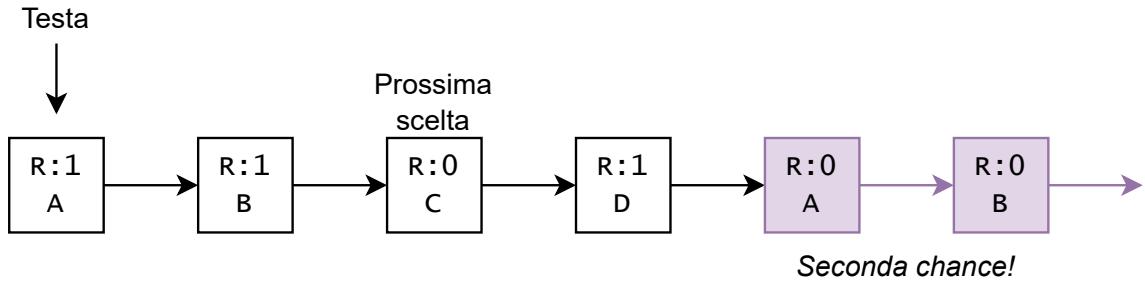
Bisogna introdurre un secondo criterio. Idealmente un principio FIFO.

6.8.3 Algoritmo FIFO - First-In First-Out

Potremmo usare il principio FIFO non come criterio secondario, ma come criterio principale. La scelta sarà **la pagina più vecchia in memoria**. Questo criterio non tiene in considerazione se la pagina più vecchia, che si vuole scartare, è comunque molto referenziata. È un algoritmo valido, ma che potrebbe causare overhead. *Strategia semplice, ma rischia di eliminare pagine vecchie e molto utilizzate.*

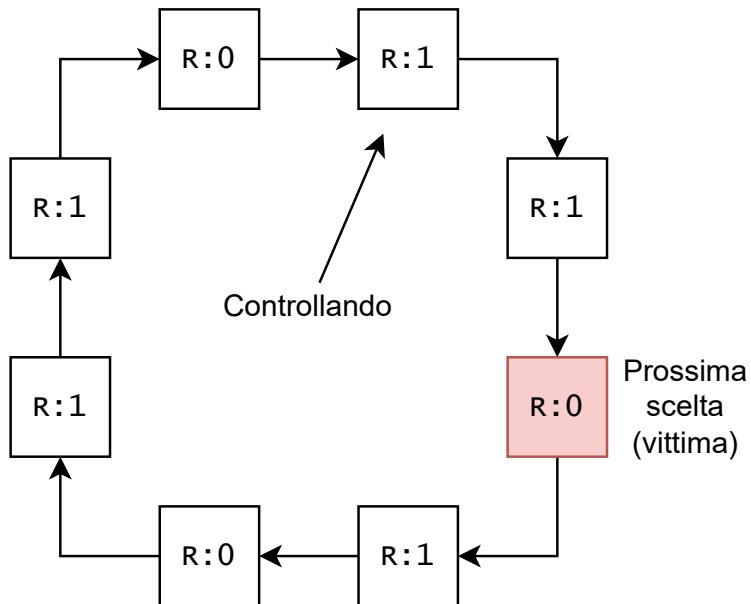
6.8.4 Algoritmo della Seconda Chance

Basato sul principio FIFO, si tiene conto dell'attuale stato del bit di referenziamento. Viene rimossa la pagina più vecchia cui bit di referenziamento è a 0. Se la pagina più vecchia ha R: 1, la pagina viene accodata con il bit R azzerato.



6.8.5 Algoritmo Clock - dell'orologio

È una variante dell'algoritmo seconda chance che sfrutta una coda circolare. Riduce il costo relativo all'enqueue del nodo più vecchio con R: 1.



6.8.6 Algoritmo LRU - Least Recently Used

L'idea di questo algoritmo, è quella di scartare la pagina meno usata di recente (che non coincide sempre con quella usata meno recentemente!).

Informazioni che sono state usate nel recente passato, probabilmente verranno usate nel recente futuro.
L'idea è ottima, ma l'implementazione è dispendiosa, se non in hardware.

Implementazione hardware

Un contatore in ogni pagina delle pagine e nella CPU, permette di capire il numero di volte in cui è stata utilizzata una pagina. Non esiste hardware simile :(, sarebbe stata un'ottima approssimazione dell'LRU.

Implementazione software

- Algoritmo **Not Frequently Used (NFU)**.
- Algoritmo di **Aging**.

6.8.7 Algoritmo NFU - Not Frequently Used

Avremo un contatore in ogni voce della tabella delle pagine e un bit di referenziazione. Questo bit si azzererà periodicamente (lasso di tempo stabilito dal sistema operativo), ma prima di azzerare R, si somma il suo valore con quello del contatore.

La scelta della pagina da eliminare, ricadrà su quella col **contatore più basso**.

Svantaggio

L'algoritmo potrebbe erroneamente privilegiare pagine molto utilizzate in passato ma scarsamente utilizzate di recente. Inoltre, l'intera procedura di azzeramento, somma dei valori, con lassi di tempo troppo brevi, può introdurre overhead.

6.8.8 Algoritmo di Aging - Invecchiamento

Ad ogni scadenza del clock, si shifta a destra (o si divide per due) il contatore associato ad ogni pagina. Ogni volta, viene accostato come cifra più significativa, il valore del bit R. L'invecchiamento di pagine non referenziate diventa molto veloce.

La scelta della pagina vittima, ricadrà chiaramente sulla pagina con valore del contatore più bassa.

	ultimo ciclo di clock	2	3	4	5	6	7	8	Cicli di clock passati
Si		1	0	0	1	0	0	0	0
No		0	0	0	1	0	0	0	0
No		0	0	1	1	1	0	1	0

Referenziato?

Immagine 6.4: Ogni bit è parte di uno storico dei referenziamenti

Piccola pecca

Questo contatore fornisce uno storico di utilizzi della pagina negli ultimi k cicli, con k bit dedicati al contatore. È un LRU con uno storico limitato (ma non abbastanza da non renderlo una valida opzione!).

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

6.9 Confronto nelle prestazioni degli algoritmi di sostituzione delle pagine

Sceglieremo come metrica il **numero di page fault**.

- Valuteremo gli algoritmi con una RAM da 3 frame.
- Useremo una sequenza compatta delle pagine a cui accedere, ad esempio 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1. In queste sequenze, non si dovranno mai avere ripetizioni di numeri, in quanto non influenzano le nostre valutazioni.

6.9.1 Valutazione algoritmo OPT

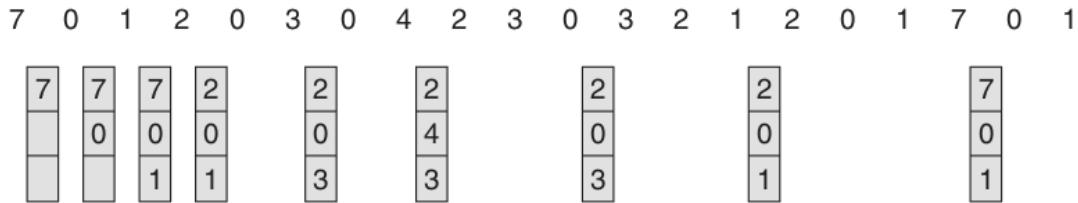


Immagine 6.5: Algoritmo OPT

Con 9 page fault. Ricordiamo che questo algoritmo è solo teorico, in quanto sfrutta, per le sue valutazioni, informazioni impossibili da ottenere, ovvero le risorse richieste nel breve futuro.

6.9.2 Valutazione algoritmo FIFO

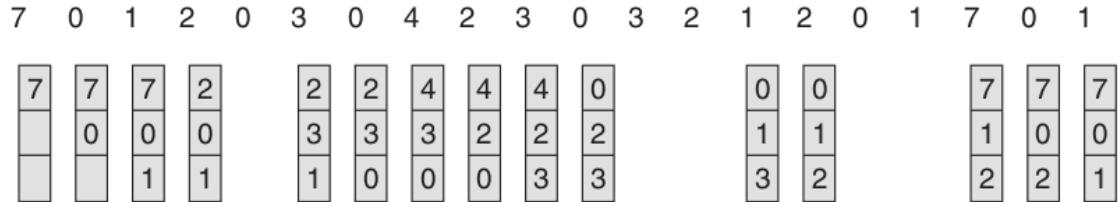
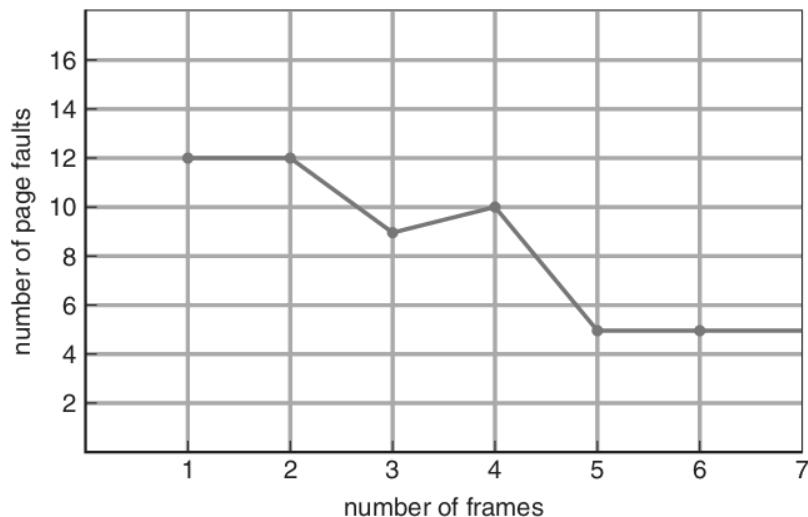


Immagine 6.6: Algoritmo first-in first-out

Con 15 page fault.

Anomalia di Belady

È un'anomalia che associa (cointuitivamente) un aumento dei page-fault con un aumento della memoria RAM. Questa anomalia può verificarsi di fronte a determinate sequenze. A parità di sequenza [1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5], osserviamo questa anomalia tra i 3 e i 4 frame.



6.9.3 Valutazione algoritmo LRU

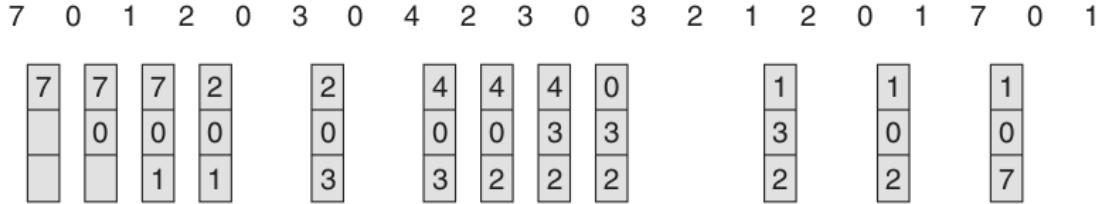


Immagine 6.7: Algoritmo Least-Recently Used

Presenta 12 page fault, e non presenta l'anomalia di Belady, in quanto gode della seguente proprietà.

Proprietà di inclusione

In ogni istante t , con n frame.

$$B_t(n) \subseteq B_t(n+1), \forall t, n$$

Ovvero, l'insieme dei frame che vengono caricati avendo n è contenuto all'interno dell'insieme dei frame che verrebbero caricati avendo $n+1$ frame.

6.9.4 Osservazioni su altri algoritmi visti

- NFU ed Aging godono della proprietà di inclusione (approssimando l'algoritmo LRU).
- Seconda chance e Clock soffrono dell'anomalia (basandosi sul principio FIFO).
- NRU soffre dell'anomalia, (basandosi sul principio FIFO).

6.9.5 Riepilogo sugli algoritmi

- **OPT.**
È il termine di paragone.
- **NRU.**
Approssimazione rozza dell'LRU che soffre dell'anomalia di Belady.
- **Aging.**
Buona approssimazione di LRU con implementazione software efficiente.

6.10 Allocazione dei frame

Con n processi, come si distribuiscono i frame disponibili tra loro? Che pagine carichiamo in memoria? Queste sono tra le domande a cui risponderemo nei prossimi paragrafi.

6.10.1 Numero minimo e massimo di frame allocati

Esistono molteplici strategie, per cui i valori non sono ben fissati, ma esistono dei valori minimi e massimi che possiamo intuire. Dato un processo, abbiamo un minimo e un massimo teorico di frame che possiamo concedergli.

- **Minimo.**

Il minimo strutturale, dipende da fattori come il tipo di set di istruzioni. Non è possibile usare un'istruzione con un operando avendo un solo frame. Il fetch dell'istruzione non causerà page fault, ma il prelevare l'operando sì. L'istruzione andrà in un loop di page fault. Quindi il minimo deve essere due. Ma istruzioni con due operandi potrebbero richiedere 3 frame! E con indirizzamento indiretto, un operando può fare riferimento a delle pagine, facendo richiedere ad ogni operando almeno due frame. Alcune istruzioni, più lunghe di una word, potrebbero cadere tra due frame. Fondamentalmente, il **minimo strutturale è un valore che dipende dall'architettura**, ma è fondamentale da conoscere. Bisogna stare molto sopra questo limite per evitare problemi.

- **Massimo.**

L'intera memoria centrale.

6.10.2 Scegliere le pagine - Pure Demand Paging

Che pagine carichiamo in memoria?

Non avendo informazioni relative alle esigenze del programma, al primo avvio, non si caricherà nulla. Il programma inizierà a richiedere i frame di cui ha bisogno. All'inizio ci sarà molto overhead, ma è il massimo che possiamo ottenere non avendo informazioni preliminari sul processo. Questo approccio si chiama **pure demand paging**, o paginazione su richiesta pura.

6.10.3 Strategie di allocazione dei frame

Due idee principali.

- **Allocazione equa.**

Non privilegiando alcun processo, dati k frame ed n processi, si dovranno dedicare $\frac{k}{n}$ frame a processo.

- **Allocazione proporzionale.**

Al processo di dimensione s_i , con m numero di frame massimo, assegniamo un numero di frame pari a

$$a_i = \frac{s_i}{S} \cdot m$$

con S uguale alla somma della taglia di tutti i processi $\in P$.

$$S = \sum_{i \in P} s_i$$

Otterremo quindi una strategia di allocazione **proporzionale alla dimensione totale del processo**.

6.11 Strategie di valutazione delle pagine da rimuovere

Abbiamo capito che **ogni processo ha un numero massimo di pagine dedicate**. Se x ha occupato tutte le pagine a esso dedicate, un qualsiasi page fault richiederà una rimozione di almeno uno o (probabilmente) più frame. Tuttavia, tra gli algoritmi di rimozione, esistono due strategie ben distinte:

- **Allocazione locale.**

Tra le possibili pagine, considero esclusivamente quelle del processo di cui devo risolvere il page fault,

- **Allocazione globale.**

Considero tutte le pagine, anche quelle di altri processi.

Allocazione globale - troppi pochi frame

Se un processo ha pochi frame allocati, questo lavora male, causando numerosi page fault.

- Se il numero di frame è **sotto il minimo strutturale**, questo viene sospeso, con un successivo swapping su disco.
- Se il numero di frame è **poco superiore il minimo strutturale**, il processo va in **thrashing**. Quando tutti i processi vanno in thrashing, il sistema va in sovraccarico. È indice di un livello troppo alto di multiprogrammazione.

Se la multiprogrammazione non è troppo alta, e sono solo alcuni i processi a soffrire di thrashing, bisogna trovare un criterio che permetta di valutare a quale processo sottrarre delle pagine, senza portarlo in thrashing.

6.11.1 Principio di località e concetto di località

Definiamo come **località** l'insieme di locazioni di memoria, contenente codice e dati, su cui il processo sta lavorando. Se la località non rientra nel numero di frame allocati al processo, è probabile il processo vada in thrashing.

"Durante l'esecuzione di una data istruzione presente in memoria, con molta probabilità le successive istruzioni saranno ubicate nelle vicinanze di quella in corso. Nell'arco di esecuzione di un programma si tende a fare riferimenti continui alle stesse istruzioni".

6.11.2 Working set

È un concetto simile alla località, ma in termini di pagine (e applicato!). Il **working set** di un processo è l'insieme di pagine su cui sta lavorando. Per ogni processo, si mantengono le pagine usate negli ultimi Δ accessi alla memoria. Il valore di Δ è scelto in funzione della **località corrente**.

Il tracciamento del working set (in realtà, l'approssimazione) ci permette di stabilire la richiesta globale di frame $D = \sum_i$ Working Set di S_i , ovvero l'esigenza del mio sistema multiprogrammato. Questa misura può essere confrontata con la memoria disponibile.

Come si calcola il working set?

Tracciando adeguatamente la somma del numero di bit di referenziamento $R = 1$ delle pagine. Questo bit è a 1 in ogni pagina referenziata. Essendo un bit che viene azzerato periodicamente, è un ottimo indice della somma dei Working Set ad un determinato istante di tempo, e in generale, permette di valutare quali pagine sono

6.11.3 Page Fault Frequency

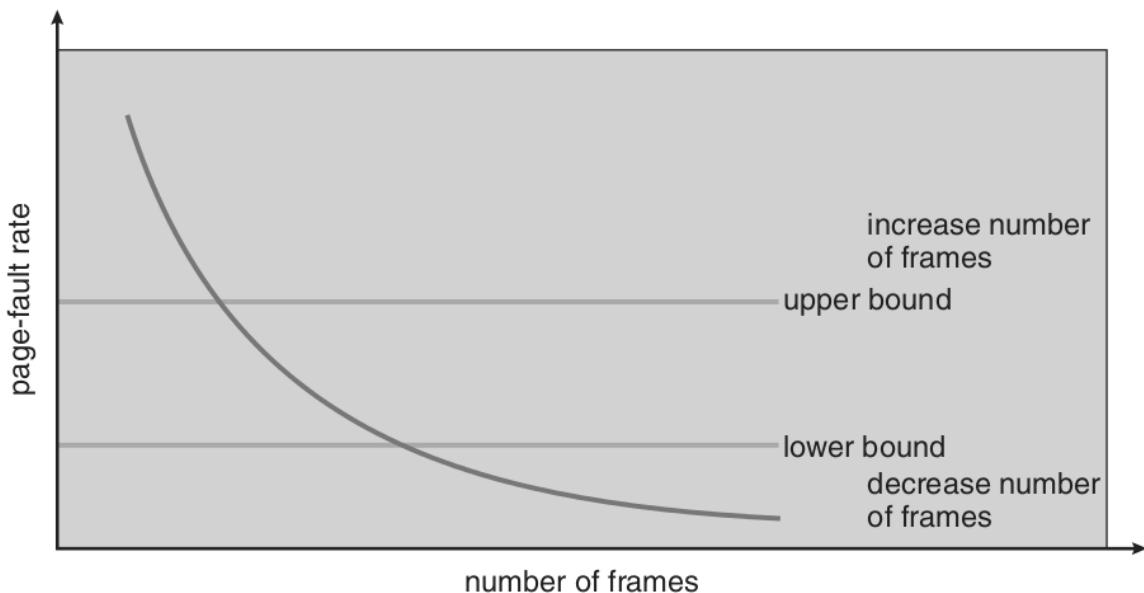
È un modello più diretto per gestire il problema del thrashing. Consiste nel monitoraggio della **frequenza dei page fault** (PFF) dei processi. Stabilendo un upper bound e un lower bound per questa frequenza, è possibile valutare quando dare più frame ad un dato processo. La frequenza fornisce un'indice dello stato del processo.

- **Sopra l'upper bound.**

Il processo sta soffrendo, ha bisogno di più pagine allocate.

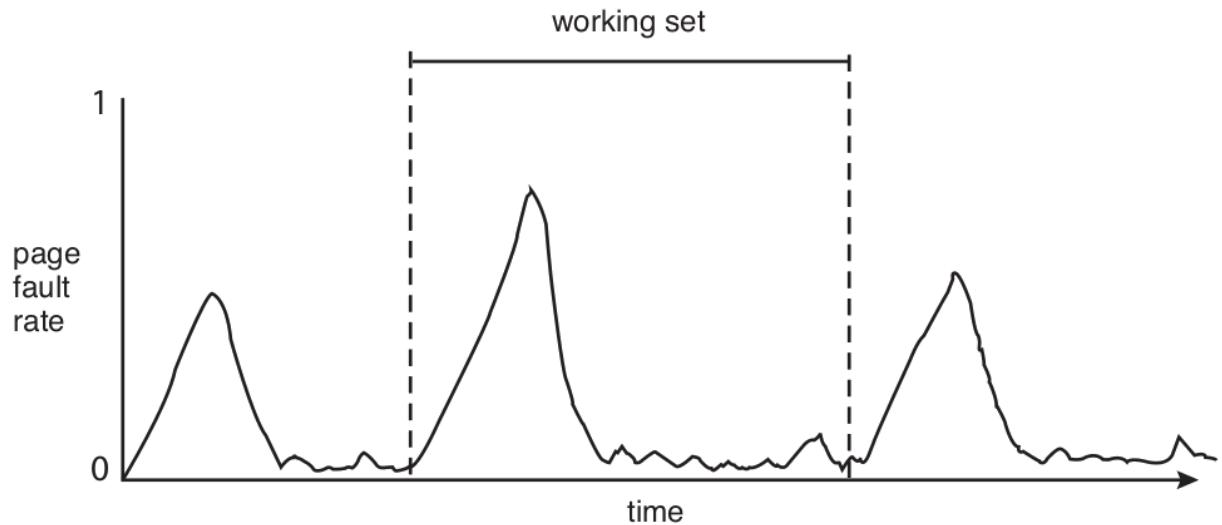
- **Sotto il lower bound.**

Il processo ha pochi page fault, possiamo ridurre il numero di frame allocati.



6.11.4 Relazione Page Fault Frequency - Working Set

Il metodo del Working Set e quello del Page Fault Frequency ottengono più o meno gli stessi risultati.



Osserviamo dei picchi di page fault. Questi, sono inevitabili e totalmente naturali in un sistema multiprogrammato. L'importante è che la frequenza di page fault si normalizzi dopo ogni picco.

Tra i due metodi, quello che viene implementato è quello del Working Set.

6.12 Politica di pulitura

Il meccanismo di gestione dei page fault è efficiente, soprattutto se ci sono frame liberi sempre disponibili.

6.12.1 Quanto costa gestire un page fault?

Sappiamo già che le pagine sporche sono quelle più lente da gestire, in quanto richiedono una scrittura su disco. Generalmente, il meccanismo di gestione dei page fault è veloce quando ci sono **frame liberi** disponibili. Quando non è così, si vorrà rimuovere una pagina. Ne consegue che, nello scenario ideale, sono sempre disponibili dei frame liberi.

6.12.2 Paaging demon

È un **processo di servizio** che controlla l'occupazione globale dei frame del sistema.

- **Seleziona, libera ed, eventualmente, pulisce pagine.**

- **Mantiene un pool di frame liberi.**

Il contenuto dei frame liberi di questo pool non è effettivamente cancellato. Quel frame sarà considerato libero e sovrascritto nel momento del bisogno, ma è possibile un ripescaggio del contenuto dal pool, nel caso in cui ci fosse richiesto. Non è da considerarsi come un vero e proprio page fault.

Perché il paging daemon?

Il paging daemon gestisce anticipatamente la rimozione di pagine che (prevede, stima) non saranno utili in memoria. In questo modo, **un futuro page fault avrà overhead minore**.

6.13 Dimensione della pagina

La scelta della dimensione della pagina di base è importante. Al crescere o decrescere della dimensione delle pagine, otteniamo effetti positivi e negativi.

Pro delle pagine più grandi

- **Tabella delle pagine più piccole.**

Il numero di record della tabella delle pagine è pari al numero di pagine. Il numero di pagine, a parità di memoria, è inversamente proporzionale alla dimensione delle pagine.

- **Migliore efficienza nel trasferimento I/O.**

Utilizzando pagine più grandi, un input/output potrà essere effettuato su pagine più grandi, e quindi un numero minore di pagine a cui accedere. Questi vantaggi si sentono di più all'interno di memorie di tipo elettromeccanico.

- **Minori page fault (e conseguente minor overhead).**

Pagine più grosse ⇒ meno pagine ⇒ meno page fault. Non sempre ne vale la pena, ma è un fatto da tenere in considerazione.

Pro delle pagine più piccole

- **Minore frammentazione interna.**

Pagine più grandi causano più spreco. Pagine

- **Migliore risoluzione nel definire il working set. Meno memoria sprecata!**

Le approssimazioni saranno più precise. Le pagine sono di dimensione fissa: blocchi più piccoli permettono di rispondere alle nostre esigenze in maniera migliore.

I pro di uno, sono i contro dell'altro.

6.14 Pagine condivise

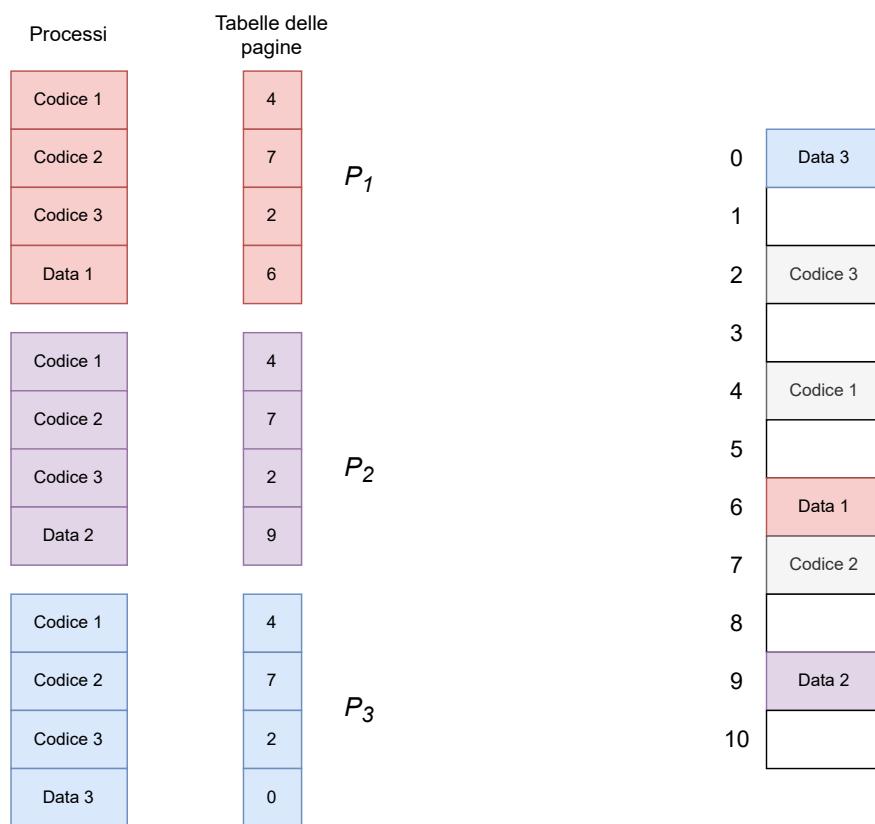
È un'ottimizzazione che il processo operativo può usare per risparmiare memoria usata. Implica una migliore gestione della RAM.

6.14.1 Scenario 1: più istanze dello stesso programma

Supponiamo di avere tre istanze (processi) dello stesso programma. P_1, P_2, P_3 . Alcuni di questi spazi di indirizzamento sono dedicati al codice eseguibile e condiviso, altri sono dedicati a dati e canali di comunicazione tra processi. Ne consegue che:

- **Alcune pagine sono in modalità lettura/scrittura.**
Ogni processo ha bisogno delle proprie pagine dedicate e isolate.
- **Alcune pagine sono in modalità di sola lettura.**
Perché avere più pagine?

Il concetto è semplice, l'implementazione altrettanto! Basterà inserire nella tabella delle pagine di tutti i processi, gli stessi indirizzi fisici per le pagine read-only!



Questa ottimizzazione funziona anche in sistemi con tabelle delle pagine multilivello.

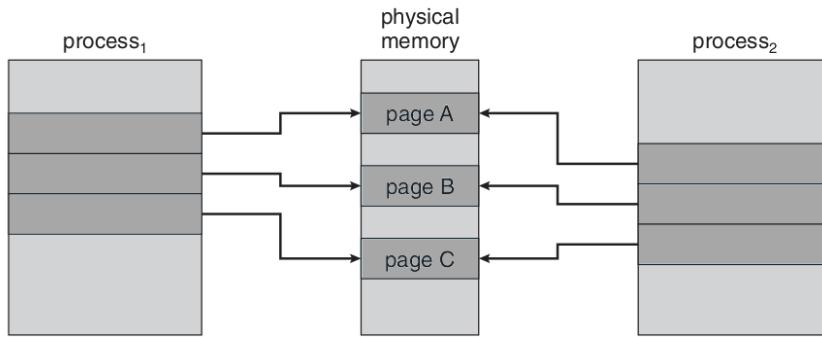
6.14.2 Pagine condivise come mezzo di IPC

Non approfondiremo questo dettaglio, ma avere pagine di memoria condivise è un'opzione di IPC valida. Non parliamo tra thread, ma tra veri e propri processi! Anche in queste circostanze, si ripropongono i meccanismi tipici per la gestione di buffer condivisi (MUTEX e co). I processi sono consapevoli di star condividendo risorse, e agiranno in funzione di questo.

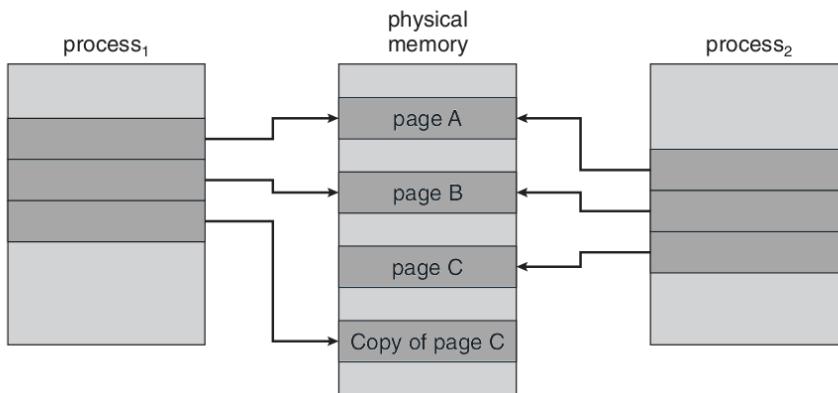
Perché con le tabelle delle pagine invertite non funzionano?

È banale. Le tabelle delle pagine invertite contengono un record per ogni frame fisico. Diventa impossibile associare due indirizzi virtuali allo stesso indirizzo fisico.

6.14.3 Copy-on-Write - Ottimizzazione con memoria condivisa



È un'ottimizzazione usata dal sistema operativo per ottimizzare la memorizzazione di processi con lo stesso contenuto. Effettuata una **fork**, o una qualsiasi copia di un determinato processo, le tabelle delle pagine verranno associate agli state frame. Questo permetterà di quasi dimezzare lo spazio richiesto in memoria da due processi uguali. Se si effettua una qualsiasi scrittura sui dati di uno dei due processi, a questo verrà creata una copia, e verrà rimossa l'associazione alla pagina precedentemente condivisa.



6.14.4 Zero-fill-on-Demand

I frame tenuti all'interno del pool dei frame liberi, sono azzerati solo quando viene fatta l'allocazione, solo on-demand.

6.14.5 Ottimizzazione con Read-Only Static Zero Page

La read-only static zero page una tecnica di ottimizzazione usata nei sistemi operativi, per rappresentare pagine di memoria completamente azzerate senza doverle duplicare fisicamente per ogni processo. Si fa riferimento ad una zero-page, e solo nel momento del bisogno (come nella copy-on-write) viene fornito un frame, precedentemente azzerato secondo il meccanismo della zero-fill-on-demand (se necessario).

6.14.6 File mappati

La mappatura in memoria è un meccanismo che permette di associare un'area della memoria virtuale di un processo al contenuto di un file memorizzato sul disco, permettendo così di accedere a quest'ultimo come se fosse parte della memoria RAM. Questa tecnica consente un accesso rapido ed efficiente ai dati (successivamente a un page fault), migliorando le prestazioni di applicazioni che gestiscono file di grandi dimensioni e facilitando la comunicazione tra processi che condividono memoria.

6.15 Allocazione della memoria per il Kernel

La gestione della memoria interna al **Kernel** avviene in termini diversi, rispetto a quella dei processi utente. Le **entità Kernel**, con i loro **thread**, necessitano spazio in RAM: **strutture dati** di qualsiasi tipo (tabelle dei processi, code, alberi, RB-tree e semafori) sono **dinamicamente allocate** in RAM.

6.15.1 Moduli Kernel come comuni processi

I Kernel moderni, gestiscono i propri moduli come se fossero **processi qualsiasi**, con tutti i vantaggi conseguenti. Questo metodo non è sempre viable, in quanto alcuni moduli Kernel necessitano di operare con indirizzi fisici: quelli relativi al DMA, o che operano con l'hardware, necessitano di frame **fisicamente contigui**. I vari sistemi operativi, gestiscono l'allocazione della memoria in maniera diversa.

6.15.2 Slab Allocation

- **Slab.**

È una sequenza di **frame fisicamente contigui**.

- **Cache.**

Un insieme di slab. Ogni cache ha una taglia: avere cache di varie taglie, permette di ottenere delle vere e proprie **cache specializzate**, una per ogni **struttura dati interna omogenea**.

Le esigenze del Kernel sono ben note, ed è quindi possibile stabilire **un numero di slab da mantenere**. Nella scelta dello slab da allocare, il sistema operativo ne sceglierà sempre di dimensione uguale (o multiplo) dello spazio richiesto, in modo di minimizzare la frammentazione interna.

I frame che vengono dedicati alle esigenze del Kernel, vengono estratti dal pool di indirizzi fisici disponibili nel sistema per la comune paginazione.

6.15.3 Vantaggi della slab allocation

- **Frammentazione esterna nulla.**

Non avviene. Tutto lo spazio non richiesto dal Kernel, è gestito tramite la paginazione. Quando un modulo Kernel richiede più spazio da annettere ad uno slab, individua una pagina vicina, e la alloca.

- **Overhead minimo.**

Operare con indirizzi fisici, minimizza l'overhead.

Capitolo 7

File System

7.1 Introduzione al File System

I **file**, le **directory** e il **file system** sono astrazioni offerte dal sistema operativo. La gestione di **grandi quantità d'informazioni**, in maniera persistente e condivisa tra processi, è l'obiettivo che ci poniamo durante la progettazione di un sistema operativo.

7.1.1 Il File system

Il termine **file system** viene spesso utilizzato per indicare due cose:

- L'interesse dei file in memoria (Cancellare l'intero file system → Cancellare tutti i file in memoria).
- L'astrazione di gestione dei file (Il file system di UNIX è diverso da quello di Windows).

Quando useremo il termine **file system**, ci riferiremo sempre all'astrazione di gestione dei file. Ogni **file system** si differenzia per i propri **dettagli di gestione** e di **implementazione**. Tra questi dettagli di gestione, individuiamo:

- **Nomenclatura.**

Dimensione e formattazione dei nomi (MS-DOS: 7 caratteri + 3 per l'estensione), case sensitivity (UNIX-based OS - case sensitive, Windows - case insensitive).

- **Tipi di file.**

Tra questi, nei sistemi operativi UNIX, abbiamo i **device file**. Rappresentano un dispositivo che appare nel file system, e tramite cui è possibile effettuare operazioni di input/output. Una scheda audio, ad esempio, apparirà come un file nel file system.

- **Tipi di accesso.**

Sequenziale o diretto. Nel primo, il file pointer viene portato avanti ad ogni operazione di lettura e scrittura. Nel secondo, il file pointer è completamente svincolato. Al giorno d'oggi l'accesso diretto è lo standard, e permette di emulare anche un accesso sequenziale.

- **Metadati e attributi.**

Tra cui nome, time-stamp (data creazione - ultima modifica), permessi di modifica, autore del file e maschere di permessi, tramite cui è possibile stabilire accesso protetto ad un file. Ogni utente potrebbe avere il proprio spazio nel file system.

- **Operazioni supportate sui file.** Consentire a due processi di leggere e scrivere sullo stesso file, ci porta alle stesse condizioni delle race condition. Il sistema operativo gestisce l'accesso condiviso ai file tramite dei **lock**. I lock possono essere:

- **Condivisi o Esclusivi.**

Sono due tipologie di lock ben distinte. I lock **condivisi** consentono a più processi di accedere allo stesso file, e vengono utilizzati quando più processi vogliono **leggere** dallo stesso file.

I lock esclusivi garantiscono accesso esclusivo ad un processo su quel file. Questo lock preclude l'accesso a qualsiasi altro processo, e viene usato solitamente per operazioni di **scrittura**.

- **Mandatory o Advisory.**

Con la strategia **mandatory**, i processi sono **obbligati** a rispettare i lock tramite l'intervento diretto del sistema operativo, che blocca il processo che ha intenzione di accedere al file lockato.

Con la strategia **advisory** il sistema operativo informa il processo se è riuscito o meno ad ottenere un lock sul file che ha intenzione di usare. Sconsiglia al processo di operare sul file trovato bloccato, ma non gli vieta di farlo.

- **Strutture dati** per la gestione dei file. Esistono tabelle globali per visualizzare i file in uso.

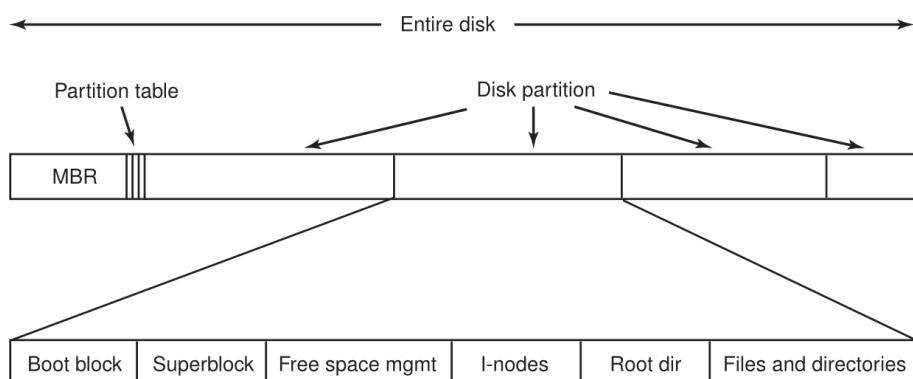
7.2 Struttura di un File system

Il file system risiede sopra una partizione di memoria. Una partizione può ospitare più file system. La gestione delle partizioni non concerne quindi il file system. Il partizionamento può essere gestito in due modi principali.

7.2.1 Master Boot Record

È un metodo storico. Il master boot record contiene quattro voci prestabilite, che tracciano l'inizio e la fine delle partizioni di memoria (esiste quindi un cap limite di massimo 4 partizioni di memoria). Il disco ha quindi in memoria:

- Al più quattro partizioni.
- La tabella delle partizioni.
- Il boot record. È il primo blocco della memoria, i primi 512 byte, contenente codice di avvio della macchina. Carica il sistema operativo presente nella rispettiva partizione.



All'interno di ogni partizione, individuiamo:

- **Boot block.**
Una volta sfruttato per parte del codice di avvio di ciascuno dei sistemi operativi delle partizioni. Il suo utilizzo, che ne spiegava il nome, è ormai deprecato. Rimane ormai inutilizzato, vuoto.
- **Superblock.**
Contiene dei metadati relativi alla partizione: specifica il tipo di file system, la dimensione dei cluster. Il resto della partizione sarà gestita in cluster.
- **Free space management.**
- **I-nodes.**
- **Rood directory.**
La radice della directory del file system.
- File e directories.

7.2.2 GUID Partition Table

È un layout moderno alternativo, preferito al MBR. Una partizione prestabilita, la partizione EFI, ha un compito dedicato per l'avvio del sistema. La partizione EFI è formattata in FAT e all'interno del file system esistono vari file di configurazione e gli eseguibili EFI da lanciare in un ambiente minimale. Questi binari si occuperanno dell'avvio dei sistemi operativi.

7.3 Implementazione dei file

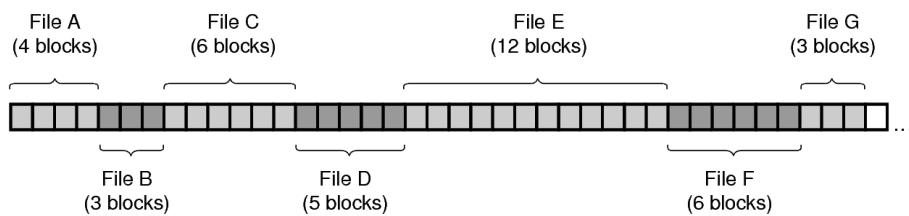
Definire le strategie relative all'implementazione dei file, è fondamentale per capire il funzionamento del file system.

7.3.1 Strategie principali

Ogni oggetto file ha una dimensione specifica in blocchi. Lo spazio richiesto in blocchi è il risultato di un arrotondamento: a ogni file, coinciderà uno spreco (frammentazione interna) causata dalla differenza tra la dimensione di un blocco e la taglia del file.

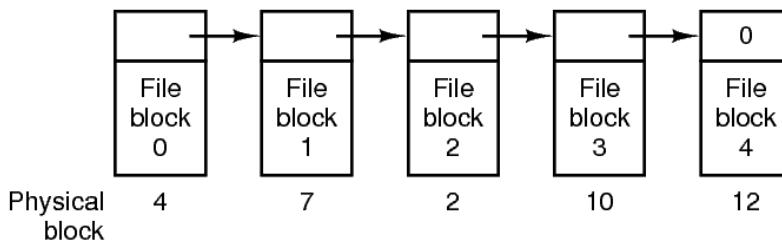
- **Allocazione contigua.**

I blocchi relativi ad un file devono essere contigui. Permette l'accesso diretto ad un blocco in tempo $O(1)$: una sola lettura, sapendo già indirizzo e offset. Sfortunatamente, l'allocazione contigua è ingestibile in situazioni di crescita dinamica dei file. Il vincolo di contiguità è troppo stretto: cosa facciamo se il prossimo blocco è occupato? Spostiamo tutti i blocchi del file da qualche altra parte? Troppo oneroso. Il metodo è utilizzabile solo nei file system di sola lettura.



- **Allocazione con liste collegate.**

Detta anche **allocazione concatenata**. Ogni blocco del file conterrà un indirizzo (di un numero di byte commisurato alla memoria della partizione), che farà riferimento al prossimo blocco del file. Questa strategia presenta un problema: con n blocchi dedicati a un file, la ricerca di un qualsiasi blocco richiederà $O(n)$ accessi in memoria.



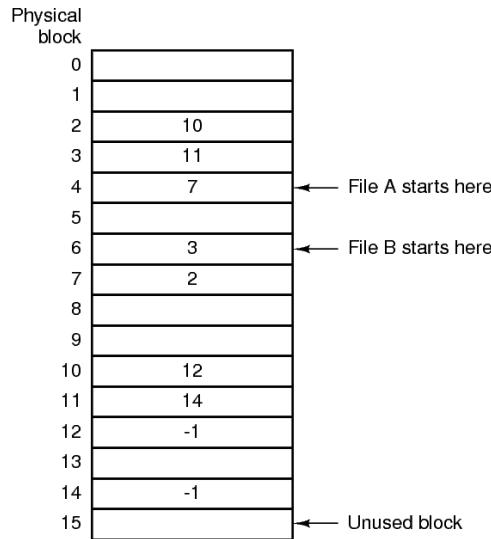
- **FAT - File Allocation Table.**

I blocchi non devono essere contigui. In più, in memoria è presente una tabella dei file allocati: ha un record per ogni blocco. Ecco come funziona:

- La FAT ha un record per ogni blocco della partizione.
- Nel record del blocco i , è contenuto l'indice del prossimo blocco, rispetto al file di i .
- Se è presente l'indice -1 , il blocco sarà l'ultimo del file di appartenenza.
- I **record vuoti** coincidono a **blocchi inutilizzati**¹.

Il FAT è di fondamentale importanza per il file system: il file system non esiste senza questa tabella! Notiamo inoltre che riprende gli stessi presupposti dell'allocazione con liste collegate (non contiguità, riferimenti tra blocchi). Per trovare un blocco qualsiasi di un file con n blocchi, sono richiesti $O(n)$ accessi all'interno della tabella. Supponendo però che l'intera FAT sia caricata in RAM, la ricerca all'interno della tabella richiederebbe $O(n)$ passaggi meno computazionalmente onerosi, rispetto agli $O(n)$ accessi in memoria delle liste collegate.

¹Scopriremo poi che altri file system, come gli i-node, implementano strutture per tracciare i blocchi inutilizzati. Per i file system FAT non è necessario! Le informazioni necessarie sono già presenti nella FAT.

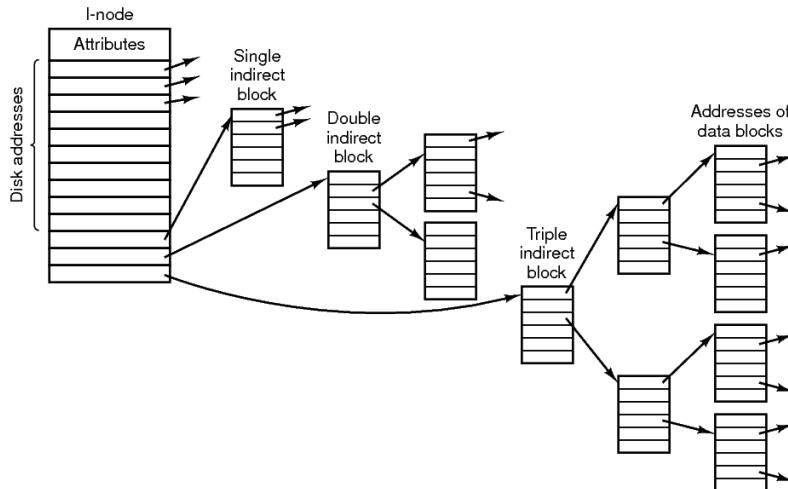


- **Allocazione con i-node**, index-node, nodi indice.

I metadati di un file vengono scritti all'interno della tabella i-node (ad eccezione del nome, che è contenuto nella directory del file). Tra questi metadati abbiamo informazioni relative allo UID e il GUID dell'owner, la dimensione del file e vari timestamps. Tra questi, alcuni sono detti **data block pointers**: questi, sono puntatori a blocchi di memoria.

Questi indirizzi, possono puntare a

- **Un blocco diretto**: è un blocco effettivo del file.
- **Un blocco indiretto singolo**: questo, punta a n blocchi del disco.
- **Un blocco indiretto doppio**: questo, punta a n blocchi indiretti singoli.
- **Un blocco indiretto triplo**: questo, punta a n blocchi indiretti doppi.



Nel file system Unix un i-node contiene quindici puntatori:

- 10 puntatori a blocco diretto. Uno a uno.
- Un puntatore a blocco indiretto singolo. Per arrivare ad un blocco, sono richiesti al più due accessi in memoria. Uno a n .
- Un puntatore a blocco indiretto doppio. Per arrivare ad un blocco, sono richiesti al più tre accessi in memoria. Uno a n^2 .
- Un puntatore a blocco indiretto triplo. Per leggere un blocco, sono richiesti quattro accessi in memoria. Uno a n^3 .

Questa struttura fornisce un upper bound alla ricerca di un blocco di un qualsiasi file, a partire da qualsiasi i-node. Inoltre, a differenza della FAT, non c'è l'esigenza di portare in memoria l'intera tabella: bensì, i blocchi diretti, indiretti singoli, doppi e tripli, verranno portati in memoria RAM solo on-demand. I sistemi moderni prediligono gli i-node, anche se le chiavette usano la FAT32.

7.4 Implementazione delle directory

7.4.1 Dove vengono inseriti i metadati e attributi?

- In un file system FAT, i metadati sono contenuti all'interno della directory del file.
- In un file system con i-node, i metadati sono contenuti all'interno della tabella degli i-node, **tranne il nome!** Il nome sta nella directory.

Tutti gli attributi hanno una lunghezza fissata, **ad eccezione dei nomi**. Nei sistemi operativi moderni, dove il nome può raggiungere centinaia di caratteri e codifiche UNICODE più dispendiose, è fondamentale conservare in maniera opportuna il nome del file.

Come gestire in maniera opportuna i nomi

Le strategie in questione variano relativamente poco tra i file system FAT e con i-node.

- **Strategia 1 - lunghezza variabile.**

Il primo campo del file stabilisce **la lunghezza in byte dell'entry**, (o del nome, negli i-node). L'unico valore di dimensione variabile è il nome: ponendo il nome come ultimo attributo del file, diventa possibile stabilire quando inizia e quando finisce il nome. A fine nome, è presente un **carattere di terminazione**.

- **Strategia 2 - heap, suddivisione tra parte fissata e variabile.**

Le parti di dimensione fissa dei record, vengono unificate in una parte di memoria. In questo modo, si evita totalmente la frammentazione esterna. Tra gli attributi, si ha un puntatore al nome del file. Le parti variabili (i nomi) si trovano in una sezione a parte, chiamata heap. Un'accozzaglia di nomi, separati da carattere di terminazione. Separare parte variabile e fissa, permette di facilitarne la gestione.

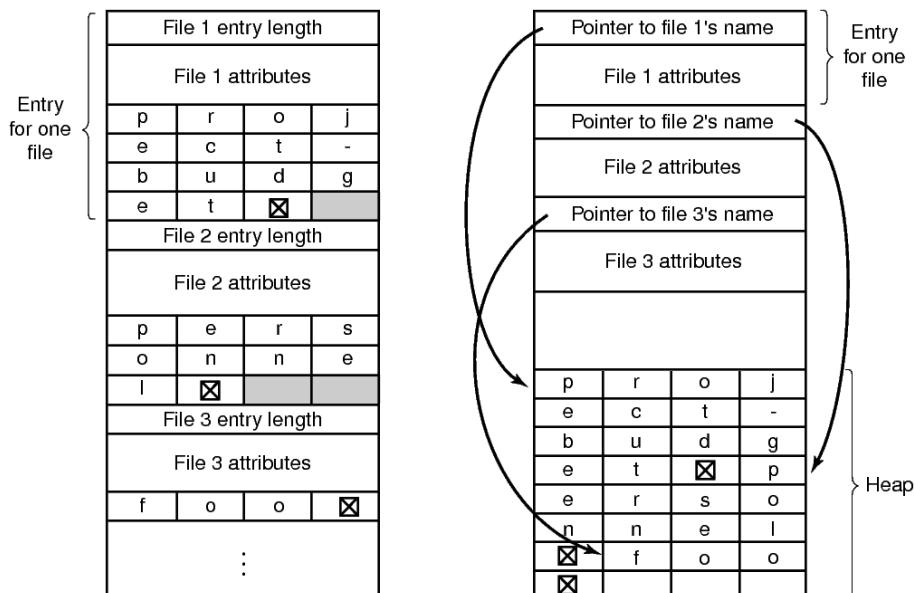


Immagine 7.1: Strategia 1 e 2 a confronto.

7.5 Condivisione di un File System tra utenti

Immaginiamo di voler condividere un file tra due o più utenti.

7.5.1 Alcune idee implementative

- **Duplicando le FAT.**

L'idea potrebbe essere quella di duplicare le liste con i riferimenti ai blocchi ottenendo liste distinte. Come possiamo mantenere coerenti le modifiche al contenuto e ai metadati?

Con un'append, si aggiorna la dimensione in byte: l'aggiornamento dei metadati diventa un problema.

- **Usando i-node, gli hard link.**

All'interno di un i-node, inseriamo un contatore che tiene traccia degli utenti che hanno accesso al file. Possiamo mettere riferimenti diversi allo stesso file. Non ci sarà l'esigenza di aggiornare i metadati, in quanto questi sono situati proprio dentro l'i-node. Il nome può infatti essere diverso tra i vari utenti!

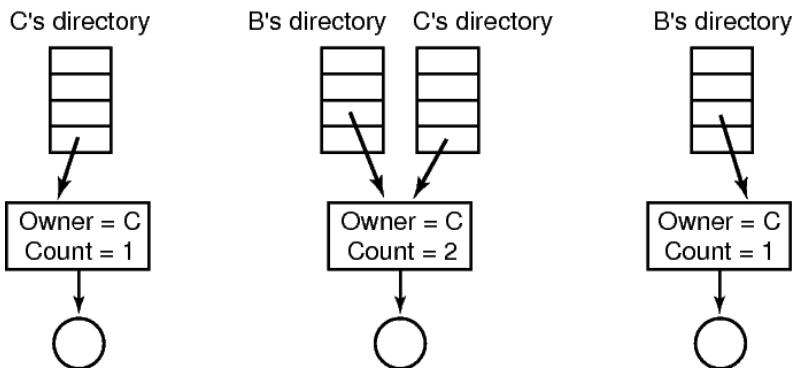


Immagine 7.2: Anomalia con accounting. È impossibile risalire all'owner originale del file.

- **Soft link**, o link simbolici.

Viene creato un i-node per un nuovo oggetto. Questo oggetto conterrà il path al file di cui vogliamo creare il soft-link.

Soft e Hard link a confronto

Nei sistemi UNIX sono supportati entrambi i tipi di link.

- **Creare directory condivise.**

È (teoricamente) possibile con entrambi i tipi di link.

- **Capire qual è il riferimento originale.**

Con la tecnica degli hard link, il riferimento originale è indistinguibile dall'hard link creato successivamente. Ciò potrebbe modificare la struttura ad albero del file system, creando un grafo ciclico, rischiando di creare loop infiniti. Per questo motivo, UNIX vieta gli hard link a directory.

- **Cross file-system links.**

Impossibili tramite gli hard link. UNIX gestisce più file systems agganciandoli all'albero del file system originale. In questo modo, è sempre possibile trovare un cammino dalla root a qualsiasi file di qualsiasi file system. Windows usa un approccio a dischi e file system separati.

Perché non è possibile avere collegamenti tra file systems tramite hard link? L'enumerazione degli i-node è relativa ad ogni file system. È ad esempio impossibile creare un hard link tra la directory `/home` e la directory `/tmp`².

²In Unix, the `/tmp` directory will often be a separate disk partition. In systems with magnetic hard disk drives, performance (overall system IOPS) will increase if disk-head movements from regular disk I/O are separated from the access to the temporary directory.

7.6 Gestione dei blocchi liberi

Anche nel caso dei file system, è fondamentale il tracciamento dei blocchi liberi.

7.6.1 Alcune strategie

- **Usare una bitmap.**

Relativamente piccola, ha una dimensione inversamente proporzionale alla dimensione dei blocchi e direttamente proporzionale al numero di blocchi. È conservata nel disco, ma può essere portata in RAM a fine di velocizzarne l'utilizzo.

- **Usare liste concatenate.**

Un'enorme lista concatenata di **blocchi contenenti indirizzi a blocchi liberi**. Richiede più spazio, ma la struttura è unificata, sfrutta blocchi liberi in memoria. È interessante osservare come la lista dei blocchi liberi si accorcia quando vengono utilizzati blocchi. Un blocco viene sottratto? La lista viene accorciata, liberando spazio sul disco.

Un'ulteriore ottimizzazione potrebbe essere inserire un contatore ad un dato indirizzo, per indicare il numero di blocchi contigui successivi.

La FAT che approccio usa?

La FAT ha già un record per ogni blocco. Dando un valore specifico al numero del blocco non allocato (magari un -2 o uno 0), si possono marcare facilmente i blocchi liberi. Entrambi i meccanismi sarebbero ridondanti.

E i-node?

Usano la strategia bitmap. Questa è inserita in uno spazio dedicato (presente in ogni partizione) chiamato **free space management**. Contiene una bitmap per i blocchi liberi, e una bitmap per gli i-node liberi.

7.7 Controlli di consistenza

Crash di sistema e problemi con l'hardware possono causare inconsistenze nel file system. Lavorare su un file system con delle incoerenze, potrebbe creare problematiche di vario tipo: scrittura consentita su blocchi che sono in realtà allocati per altri file. Parleremo quindi di strategie che permettono di individuare delle incongruenze.

7.7.1 Flag

Quando un file system viene montato, un flag relativo a questo file system viene settato a 1. Quando viene smontato correttamente (e in fase di shutdown) viene settato a 0. Un malfunzionamento lascerà il flag a 1, e quando verrà rimontato, alcune utility avvieranno dei controlli di consistenza.

7.7.2 Controlli di consistenza

Tra i principali controlli di consistenza, abbiamo quelli relativi all'occupazione dei blocchi, e quelli relativi al numero di riferimenti agli i-node.

Sull'occupazione dei blocchi

Si confronta il contenuto della bitmap con l'occupazione effettiva dei blocchi.

Block number	Block number
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 1 0 1 0 1 1 1 1 0 0 1 1 1 1 0 0	1 1 0 1 0 1 1 1 1 0 0 1 1 1 1 0 0
Blocks in use	Blocks in use
0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1	0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 1 1
Free blocks	Free blocks
(a)	(b)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 1 0 1 0 1 1 1 1 0 0 1 1 1 1 0 0	1 1 0 1 0 2 1 1 1 0 0 1 1 1 1 0 0
Blocks in use	Blocks in use
0 0 1 0 2 0 0 0 0 1 1 0 0 0 1 1	0 0 1 0 1 0 0 0 0 0 1 1 0 0 0 1 1
Free blocks	Free blocks
(c)	(d)

1. Dati consistenti.
2. Blocco 2 sia libero che occupato.
- 3.
4. Blocco 5 usato da più file. *Qui, ci scappa il morto!*

Sul numero di riferimenti agli i-node

Dentro ogni i-node è presente un campo contenente il numero di riferimenti ad esso. Il controllo di consistenza scorrerà l'intero albero del file system tenendo conto dei riferimenti di ogni i-node.

- Se il numero di volte in cui un i-node è stato individuato nel file system è zero, il contatore deve essere a zero, e quindi l'i-node non è ancora stato allocato.
- Altrimenti, il numero di riferimenti deve coincidere con quello contato.

7.7.3 Journaling

L'idea è la seguente: un file system qualsiasi può essere corredata con un journal³, un log. Il log tiene conto di tutte le operazioni sui meta-dati (e strutture dati, quali i-node, bitmap e così via), tiene conto delle **macro operazioni** su tutti i file. Il journal sta sul disco, e il sistema operativo verifica che sia aggiornato correttamente. Quando una sequenza di istruzioni è stata conclusa con successo, questa viene totalmente cancellata. Se la sequenza di istruzioni non è stata conclusa, e il flag relativo alle anomalie è stato trovato a 1, tutti i passi nel log vengono rieffettuati.

Il rischio è che alcune operazioni vengano effettuate più volte (una pre e una post crash). Per questo motivo, le operazioni devono essere idempotenti e non creare problemi se reiterati.

L'obiettivo del journaling è creare un log delle operazioni sui meta-dati, per preservare lo stato dei file system nel caso di crash.

- Un esempio di operazione non idempotente: aggiungi questi blocchi alla lista dei blocchi liberi.
- Un esempio di operazione idempotente: aggiungi questi blocchi alla lista dei blocchi liberi, **se non sono già presenti**.

Non tutte le operazioni sui meta-dati sono inserite nel journal. Memorizzare cambiamenti relativi al contenuto sarebbe molto utile e sicuro, ma anche tanto tanto dispendioso. Garantire l'integrità dei metadati (e quindi delle directory e della struttura del file system) è più importante per non propagare i danni all'intera struttura.

I file system moderni

In una situazione di crash, i file system moderni, limitano i controlli di integrità ai blocchi citati all'interno del journal. Questo velocizza di molto il ripristino della coerenza. È inutile fare un controllo a tappeto.

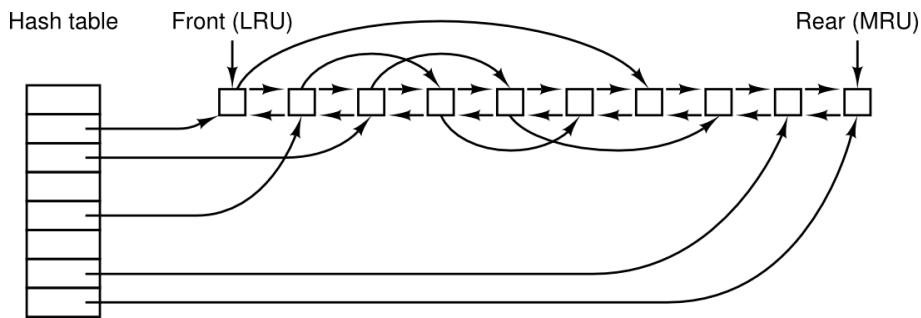
³È una struttura dati che può essere mantenuta nella stessa o in un'altra partizione.

7.8 Cache del disco

Per migliorare le prestazioni dei dischi si usa una **cacne del disco**. Fa da ponte tra una memoria veloce ma di piccole dimensioni, e una memoria lenta ma di grande dimensioni. La cache del disco contiene i dati che sono stati letti di recente e, in alcuni casi, aree di dati adiacenti a cui è probabile che si acceda in seguito. È una **soluzione totalmente software**. Un'idea sarebbe quella di utilizzare la stessa strategia **LRU**, mantenendo un timestamp relativo all'ultimo utilizzo di un blocco.

7.8.1 Struttura basata su tabelle hash.

Con riferimenti a tutti i blocchi presenti nella coda, e in cui individuiamo blocchi **least recently used (LRU)** e **most recently used (MRU)**. La tabella hash consente accessi a tempo costante.



L'implementazione vede in realtà due liste: una per mantenere l'ordine degli elementi rispetto all'ultimo utilizzo, una per accedere agli elementi dell'hash table.

7.8.2 Free-behind e Read-ahead

Immaginiamo di star gestendo un file in lettura o scrittura sequenziale (in generale, un accesso sequenziale, non casuale): ciò che faremo, sarà leggere blocco per blocco il file.

- **Free-behind.**

Effettuata una scrittura sequenziale, si liberano i blocchi precedentemente scritti: raramente immaginiamo di dover leggere gli stessi blocchi.

- **Read-ahead.**

Si ottimizza il processo ad accesso sequenziale, leggendo non solo i blocchi richiesti, ma anche quelli successivi.

Con enormi vantaggi sui dischi elettromeccanici.

Altro da sapere

La cache del disco può essere inibita.

7.9 Che file system usano i nostri sistemi operativi?

7.9.1 Windows

- **exFAT.**
Sui dischi rimovibili
- **NTFS.**
Sui dischi fissi. Supporta **journaling**, **cifratura**, **compressione** (trasparente, prima dell'effettiva scrittura), copia shadow⁴ e dischi multipli (tecniche RAID gestite dal sistema operativo). È un file system moderno e molto complesso.
- **ReFS**, attualmente in sviluppo.
Seguendo la falsa riga di BTRFS.

7.9.2 Linux

- **ext-4.**
Journaling, allocazione efficiente.
- **BTRFS.**
È un file system molto recente. Piuttosto stabile, offre journaling, compressione (trasparente), dischi multipli (con tecniche RAID), checksum dei dati e dei metadati⁵, volumi e clonazione (CoW).

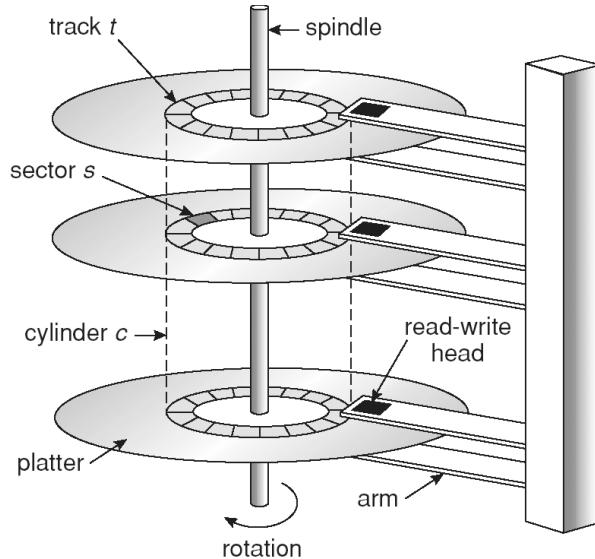
7.9.3 MacOS

HFS+ recentemente soppiantato da APFS.

7.10 Scheduling del disco

Gestione della coda delle richieste pendenti di I/O su disco. Si può mandare mandare al controller del disco solo una richiesta alla volta. Il nostro obiettivo è massimizzare il numero di richieste in unità di tempo (**throughput**), **minimizzando il tempo medio d'accesso**.

Ogni richiesta ha un costo: questo dipende da varie circostanze. Per questo motivo, si possono stabilire delle politiche di selezione per la prossima richiesta da effettuare.



- **FIFO.**
Tecnica banale, non premia particolarmente.

⁴Copy-on-write sui file. Permette di risparmiare spazio in memoria quando si hanno due copie dello stesso file: fino a quando uno dei due verrà modificato, la copia sarà solo un riferimento agli stessi blocchi di memoria. I file comuni verranno separati nel momento di una modifica.

⁵per garantire congruenza tra i dati e metadati prima di lettura e scrittura, per evitare di leggere dati danneggiati. Un checksum può tornare falsi negativi

- **Tempo di posizionamento più corto.**

O *Seek-time*, metrica che studieremo.

- **Latenza di rotazione.**

Il problema di cui parleremo, è chiamato problema dello **scheduling del movimento della testina su disco**.

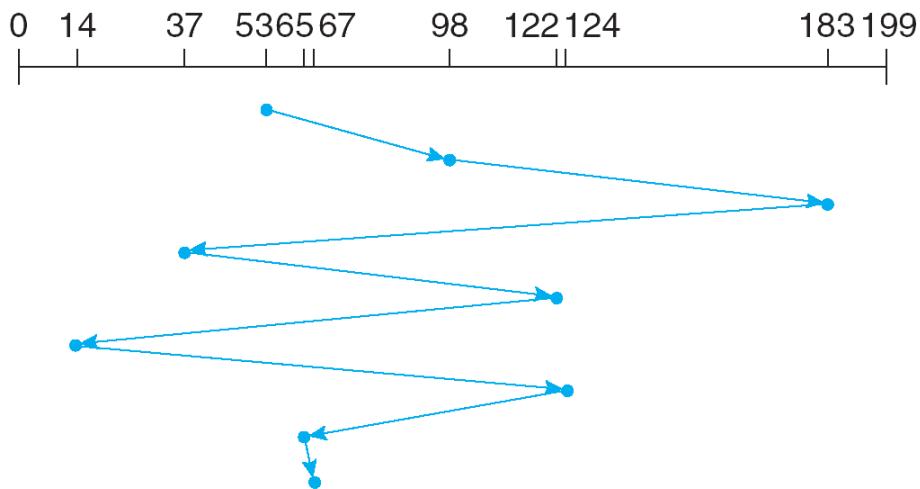
7.10.1 Ottimizzare il Seek-time

La lista delle richieste sarà, partendo dal cilindro 53:

98, 183, 37, 122, 14, 124, 65, 67

1 - Strategia FIFO

First-Come-First-Served.

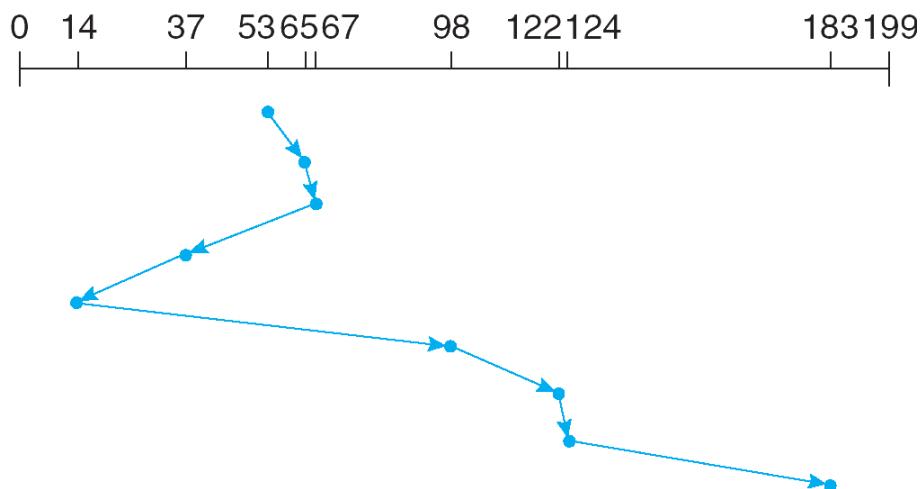


$$(98 - 53) + (183 - 98) + (183 - 37) + (122 - 37) + (122 - 14) + (124 - 14) + (124 - 65) + (67 - 65) = 640$$

Vengono percorse, in totale, 640 tracce. È un algoritmo **semplice** (letteralmente nulla da modificare), **equo, ma non efficiente**.

2 - SSTF Shortest Seek Time First

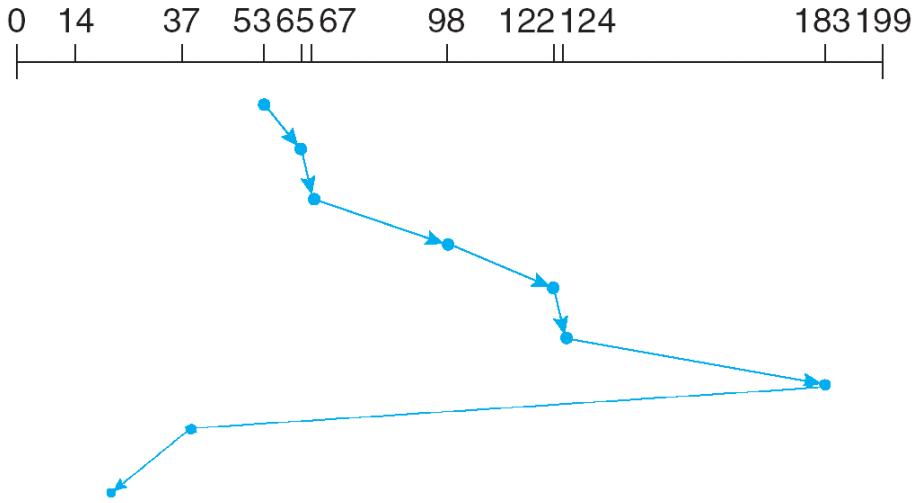
Si minimizza il tempo di ricerca su disco, di ogni richiesta.



236 tracce. Si ottengono ottime prestazioni, ma non è equo (si rischia starvation, soprattutto di fronte ad un alto numero di richieste in arrivo)

3 - Algoritmo dell'ascensore (o scansione, look

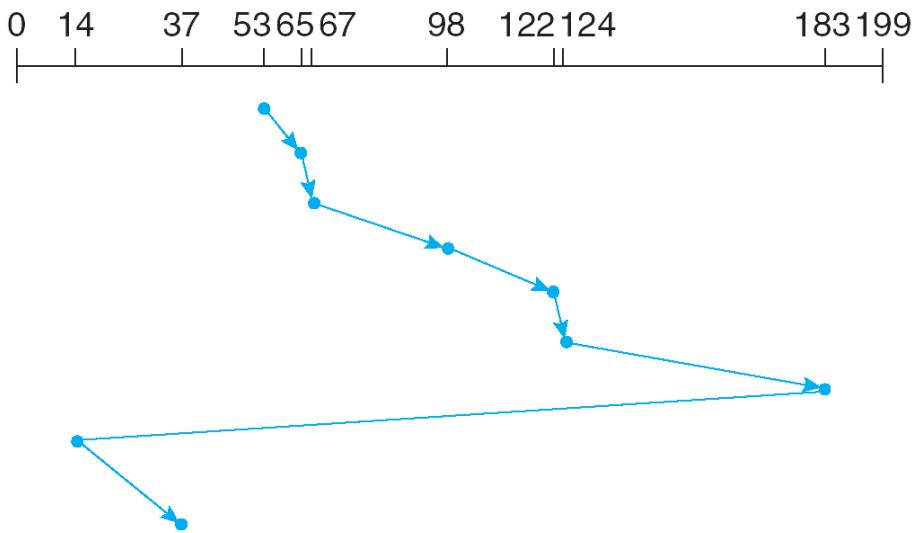
Mantiene un verso fino all'ultima richiesta in tale direzione. Le richieste sono soddisfatte o da sinistra verso destra, o da destra verso sinistra, fino a quando non ci sono richieste da poter soddisfare nel verso stabilito.



299 tracce, più di SSTF. Garantisce un'upper bound ai tempi di attesa per ogni richiesta (in questo caso, 199 tracce), non rischia starvation, e scansiona in maniera uniforme. Il nome deriva dal fatto che, lo stesso algoritmo, viene usato dagli ascensori.

4 - Scheduling per scansione circolare

Considera le posizioni come **collegate in modo circolare**. Arrivato a fine disco (ovvero all'ultima richiesta soddisfacibile dal verso corrente), la testina si sposta alla richiesta più lontana dal verso opposto, ignorando qualsiasi richiesta pendente. È uno spostamento molto veloce.



Con questa sorta di **effetto pac-man**, si abbassa il tempo di attesa medio, in attesa di tante richieste. Il ritorno indietro è molto veloce, minimizzando il tempo di attesa nei casi peggiori.

- In situazioni ad alto carico, il sistema circolare è il migliore.
- A basso-medio carico, si preferisce SSTF o la scansione standard.

7.10.2 Esempio di esercizio

Supponiamo di avere un disco con 200 tracce (numerate da 0 a 199) la cui velocità di seek è di 1 traccia per *ms*. All'istante $t = 0$ il sistema operativo sta servendo una richiesta sulla traccia 100 e in coda ci sono già le seguenti richieste per le tracce (50, 115, 180). Successivamente arrivano altre richieste all'istante $t = 70$ per la traccia 150 e all'istante $t = 130$ per la traccia 90. Si calcoli il tempo di ricerca complessivo (in ms) per servire tutte le richieste secondo la politica LOOK (scansione), Iniziando in ordine ascendente (dalla traccia 0 verso la traccia 199) e trascurando la latenza rotazionale e il tempo di trasferimento. Indicare anche la sequenza di scheduling considerata.

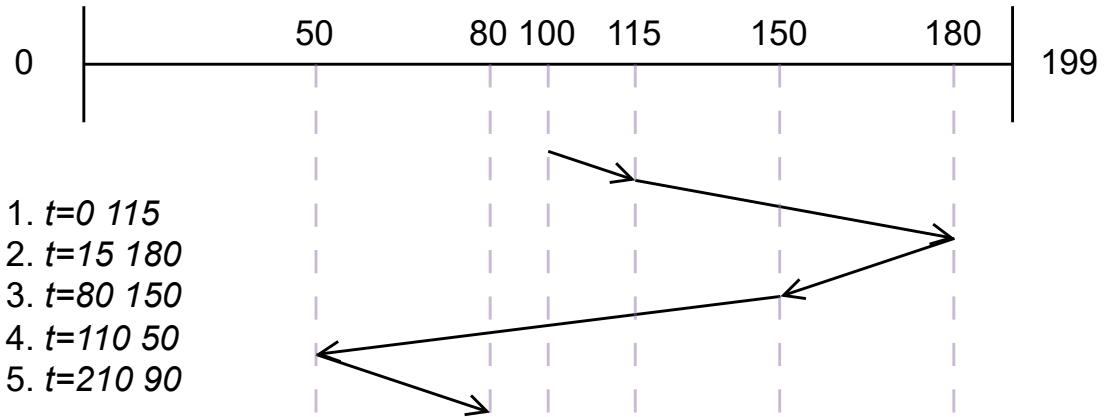


Immagine 7.3: Esercizio svolto in classe

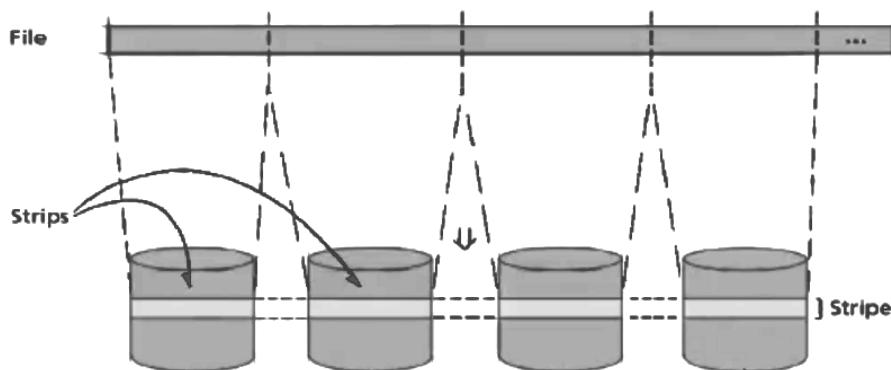
7.11 Parallelismo dei dischi (Sistemi RAID)

I sistemi RAID (*Redundant Array of Independent Disks* - Array Ridondante di dischi indipendenti) offrono una gestione di un'unico disco logico (virtuale) su più dischi fisici. fittizio. Che benefici otteniamo?

Le richieste verso il disco sono pilotate verso il disco virtuale, ma le operazioni di lettura e scrittura verranno effettuate sui dischi effettivi, ovvero tutti i dischi fisici coinvolti. **Permette di sfruttare più dischi indipendenti in maniera parallela**, con fine ultimo quello di **aumentare le prestazioni**, e di migliorare la **resilienza ai guasti**, tramite meccanismi di ricostruzione delle informazioni perse.

7.11.1 Striping

I dati relativi ad un'unità logica (un file, un volume) vengono distribuiti su più dischi: questa procedura è detta **striping**. Gli **stripe** sono multipli del blocco, di medio-grandi dimensioni. Le stripe sono distribuite sui vari dischi in maniera casuale.



I sistemi operativi attuali, preferiscono gestire lo striping tramite software. Si ottengono prestazioni simili tramite hardware (da parte del controller dei dischi).

Rischi relativi all'uso di più dischi - maggiore sensibilità?

Usare più dischi fisici per distribuire un volume logico, significa rendere i dati molto sensibili ai guasti di uno dei dischi coinvolti qualsiasi: i sistemi raid aggiungono ridondanza per ottenere migliore affidabilità, ed è possibile contemplare dischi spare con meccanismi di sostituzione automatica.

Aumento di prestazioni

Le richieste al disco virtuale, sono distribuite parallelamente ai dischi effettivi sottostanti. Questi opereranno in parallelo. Supponendo dischi uguali a pari prestazioni, e con porzioni di file uguali, otterremo un boost delle performance di 4 volte.

Guasti

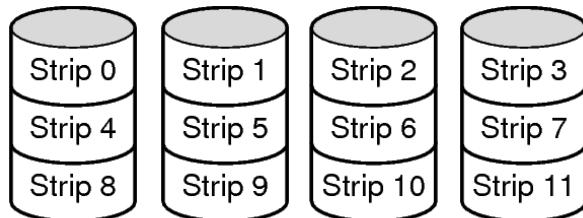
Un guasto su un disco qualsiasi, si propaga sul volume logico RAID: aumentando la ridondanza dei dati, migliora l'affidabilità. Esistono inoltre meccanismi di sostituzione dei dischi tramite dischi di riserva (prima dei guasti non fanno parte del sistema RAID), che vengono ripopolati il prima possibile redistribuendo i dati.

7.12 RAID

Redundant Arrays of Inexpensive Disks, noti anche come Redundant Arrays of Independent Disks. Esistono vari livelli dei sistemi RAID.

7.12.1 RAID 0 - *striping*

Effettua **striping** in modalità **round-robin**. Ottiene prestazioni ottimali tramite un approccio semplice e con letture di grandi volumi.

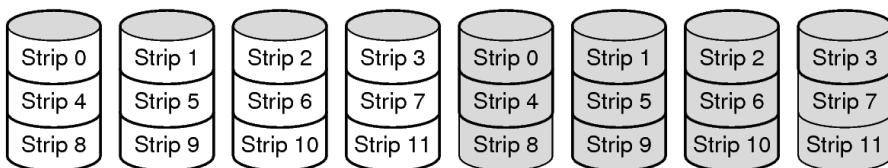


Nella migliore delle ipotesi, con n richieste ed m dischi, ci saranno

$$\frac{n}{m} \text{ rischieste/disco}$$

E se sono sfortunato :/? Tutte le richieste saranno sullo stesso disco. Fortunatamente, nei grandi numeri, la distribuzione delle richieste è più vicina al best case che al worst. Inoltre, **non usa ridondanza**: sono quindi **molto vulnerabili ai guasti**, considerando che ogni disco può guastarsi. Teoricamente, anche se i dischi di oggi sono molto sicuri, la probabilità che si guastino aumenta col numero di dischi del sistema RAID.

7.12.2 RAID 1 - *mirroring*

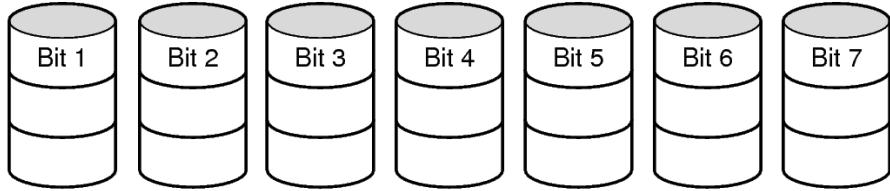


Gli eventuali stripe sono anche chiamati **duplicati**, ma può anche essere usato senza striping. Le prestazioni in lettura raddoppiano (in tutti i casi), ma a causa del doveroso aggiornamento di ambo i dischi, non abbiamo aumenti di performance per la scrittura (rispetto ai RAID 0, se abbiamo striping). Otteniamo tuttavia maggiore tolleranza ai guasti. Caso peggiore in lettura 2×, caso peggiore in scrittura 1×, caso migliore in scrittura 4×. C'è un alto overhead causato dal disco praticamente duplicato. Con n dischi, ne sfrutto $n/2$. *Pago quattro tera per averne due...*

7.12.3 RAID 2 - striping a livello di bit con ECC

Lavora sulle word applicando un codice di correzione degli errori ECC. Può correggere singoli bit di errore tramite i **codici di hamming**.

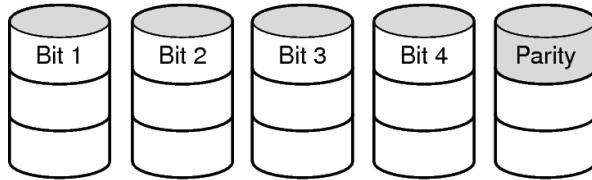
Un esempio? 4 bit con 3 bit di ridondanza. Distribuendo i 7 bit su 7 dischi differenti, e **gestendo opportunatamente la sincronia delle rotazioni dei dischi**, si ottiene una forte **fault tollerance**, resistenza ai guasti.



Guastato un disco, se ne fa un altro. Tramite gli altri 6 dischi, è possibile ricostruire l'informazione nel disco perso, qualsiasi esso sia. Con 7 dischi, ne sfrutta 4. L'ECC è overkill.

7.12.4 RAID 3 - striping a livello di bit con bit di parità

Aggiunge un solo disco con raccolti i **bit di parità**.

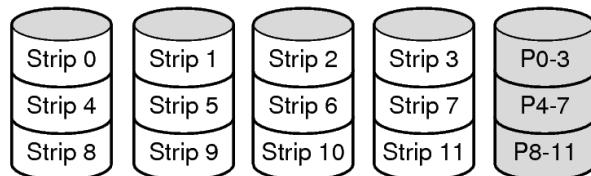


Guastato un disco, fortunatamente, è facile stabilire il valore dei bit persi basandosi sul valore di parità. Offre gli stessi benefici del RAID 2 a costo minore. Serve ancora sincronia tra i dischi. Con 5 dischi, ne sfrutta 4.

I prossimi due RAID sono quelli effettivamente usati.

7.12.5 RAID 4 - striping a livello di bit con XOR sull'ultimo disco)

È **basato sullo striping a blocchi**. L'ultimo disco contiene solo ridondanza: lo **XOR** dei bit degli strip. Il disco dedicato alla parità è molto utilizzato.



$$P_{0-3} = s_0 \oplus s_1 \oplus s_2 \oplus s_3$$

Guasto di s_2 ?

$s_2 = P_{0-3} \oplus s_0 \oplus s_1 \oplus s_3$ Qualsiasi guasto può essere corretto, anche nei dischi di parità.

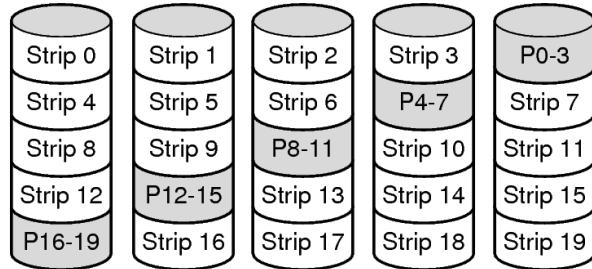
- Prestazioni in lettura: $n \times$.
 - In scrittura, apparentemente, pessimo costo: $n - 2$ letture e 2 scritture per ogni bit.
- Fortunatamente lo **XOR** può essere ricalcolato anche solo conoscendo il vecchio valore del bit riscritto $s_x \rightarrow \overline{s_x}$.

$$\overline{P_{0-3}} = P_{0-3} \oplus s_0 \oplus \overline{s_0}$$

Questo perché due **XOR** si annullano.

7.12.6 RAID 5 - *striping a livello di bit con informazioni di parità distribuite*)

Risolve il problema relativo all'uso maggiore del quinto disco. Ogni scrittura su un disco, in RAID 4, richiede una scrittura fissa sul quinto disco. Distribuendo le informazioni di parità, distribuiamo il carico di lavoro, con analoga resistenza ai guasti e performance. RAID 5 risolve l'unica problematica di RAID 4.



Un'ultimo vantaggio? Potenzialmente il lavoro è distribuibile su tutti i dischi. Nessun disco ha uno scopo fisso, tutti possono soddisfare richieste, tutti hanno delle informazioni di parità.

7.12.7 Recap sui vari RAID

RAID	Tipo di stripe	Ridondanza	In breve
0	Blocco	No	Sensibile a guasti di qualsiasi disco, velocizza in maniera importante le letture di grandi volumi, soprattutto se questi sono ben distribuiti su più dischi.
1	Blocco	Mirroring (non corregge)	Performance di lettura $\times n$ per n dischi, buona fault tollerance ma alto overhead. Il numero di scritture aumenta per ogni mirror, ma la velocità è limitata dal disco più lento.
2	Bit	Hamming (ECC)	Richiede sincronia a causa dei bit, bottleneck del disco fisico di velocità minima, ottima fault tollerance. Numero di dischi stabilito secondo le regole dell'ECC (2^n). Lo striping a livello di bit è molto pesante.
3	Bit	Parità	Un solo disco per la correzione, stessa fault tollerance di RAID 2 a basso costo. Lo striping a livello di bit è molto pesante.
4	Blocco	XOR	Un singolo disco conterrà gli XOR di ogni blocco. Non necessita sincronia, ottima fault tollerance. L'aggiornamento dei blocchi di parità è ottimizzato calcolando lo XOR dei vecchi valori di parità con i nuovi valori dei blocchi aggiornati.
5	Blocco	XOR distribuito	Distribuendo i blocchi di parità su tutti i dischi fisici, si crea meno bottleneck, si distribuisce l'usura su tutti i dischi del volume RAID. Ogni disco ha pari responsabilità.

7.13 Solid State Disk - SSD

Sono dispositivi basati su memorie flash. Usano tecnologie **NAND**, con architetture più simili alle RAM, ma dalla natura non volatile. Offrono **lettura molto veloci**, grazie all'**assenza di parti meccaniche**.

Per questo motivo, gli SSD ignorano i meccanismi di scheduling del disco, questi vengono inibiti, in quanto innecessari: basta un approccio FIFO per ottenere prestazioni ottimali.

7.13.1 Prestazioni

Gli SSD si usurano **ad ogni operazione di cancellazione**. Ogni riscrittura di (parte di) un blocco avviene previa cancellazione dell'intero blocco. Definiamo blocco come un'unità di cancellazione. Un blocco è composto da pagine, definite come **unità di allocazione**.

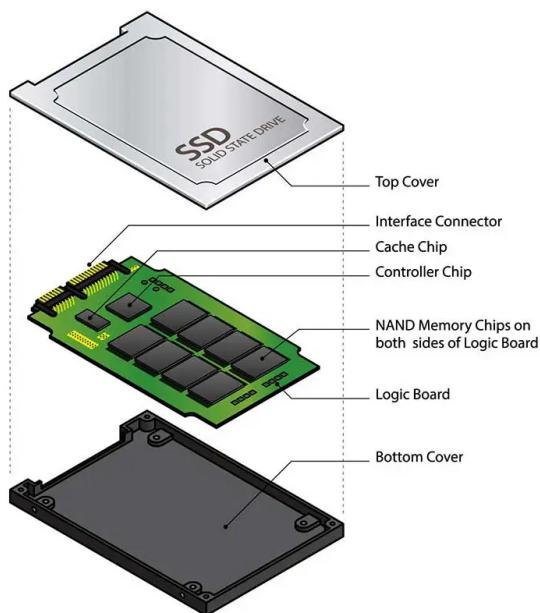
Le letture sono effettuate su pagine, le cancellazioni sono sugli interi blocchi. Per aggiornare un'informazione di una pagina, va effettuata la cancellazione e riscrittura dell'intero blocco.

7.13.2 Memorie Flash e File System

Cancellare ad oltranza singole pagine, usura rapidamente blocchi. Bisogna avere un file system ottimale per minimizzare il deterioramento. Immaginiamo un sistema FAT su un SSD: i blocchi contenenti la FAT verrebbero consumati molto rapidamente.

- **Flash-Friendly File System.**
- **Rimappaggio dei blocchi** tramite controller hardware, detti **Flash Translation Layer**.
A prescindere dal file system, il controller gestirà una sorta di mappa (stile tabella delle pagine) che si pone tra memoria effettiva e file system, mappando blocchi logici a blocchi effettivi. Il controller, in fase di riscrittura di un blocco, invece di cancellarlo, rimapperà lo stesso blocco logico ad un nuovo blocco fisico, in modo da uniformare l'usura dei blocchi.

Una sovrascrittura di un blocco logico, coinciderà con una scrittura ad un altro blocco fisico.



7.13.3 Garbage Collection e TRIM

Le celle piene di **spazzatura** (file modificati e cancellati), vengono cancellate solo quando è necessario. La cancellazione all'ultimo momento possibile, permette di scegliere il miglior blocco da cancellare: un blocco con tante pagine da eliminare e poche da mantenere, sarà migliore.

Il controller stesso, crea spazzatura: i blocchi cancellati nel volume logico, rimangono nel sistema fisico. Il controller non ha visibilità relativa a quali file sono spazzatura o meno: in fase di riscrittura, cancellerà anche i file non più necessari all'OS, usurando velocemente e rendendo lenti dischi virtualmente vuoti. L'operazione **TRIM**, offerta dal Controller al Sistema Operativo, permette di segnalare quali blocchi possono essere ignorati in fase di cancellazione. Il Sistema Operativo potrà informare il controller in fase di cancellazione.