

# Sistemi operativi

---

Appunti personali



Università  
di Catania

Università degli Studi di Catania  
Dipartimento di Matematica e Informatica  
Corso di Laurea triennale in Informatica (L-31)

**Autore**  
Emanuele Galiano

**Versione**  
5 febbraio 2026



# Licenza

Questo documento, intitolato *Sistemi operativi – Appunti personali*, è distribuito sotto licenza **Creative Commons Attribution–ShareAlike 4.0 International (CC BY-SA 4.0)**.

La presente licenza è stata scelta con l'obiettivo di favorire la diffusione, la condivisione e il riutilizzo del materiale didattico, garantendo al contempo il riconoscimento degli autori originali e la preservazione della natura open delle opere derivate.

**Autori:** Emanuele Galiano

**Affiliazione:** Università degli Studi di Catania – Dipartimento di Matematica e Informatica

**Anno accademico:** Anno Accademico 2024/2025

## Diritti concessi

In conformità con i termini della licenza Creative Commons BY-SA 4.0, è consentito a chiunque di:

- copiare e ridistribuire il materiale in qualsiasi mezzo o formato;
- adattare, modificare e trasformare il materiale;
- utilizzare il materiale anche per scopi commerciali.

Tali diritti sono concessi a titolo gratuito e non possono essere revocati, purché siano rispettate le condizioni indicate nella sezione seguente.

## Condizioni

L'utilizzo del materiale è subordinato al rispetto delle seguenti condizioni:

- **Attribuzione (BY):** deve essere fornita un'adeguata attribuzione dell'opera, citando **titolo**, **autori** e, ove possibile, la **fonte**. L'attribuzione deve essere effettuata in modo ragionevole e non tale da suggerire che gli autori originali approvino l'uso o le modifiche apportate.
- **Condividi allo stesso modo (SA):** nel caso in cui il materiale venga modificato, trasformato o utilizzato per creare opere derivate, tali opere devono essere distribuite sotto la *stessa licenza* Creative Commons Attribution–ShareAlike 4.0 International.

Non è consentito applicare termini legali o misure tecnologiche che limitino giuridicamente altri utenti dall'esercitare i diritti concessi dalla licenza.

## **Assenza di garanzia**

Il materiale è fornito “*così com’è*”, senza garanzie di alcun tipo, esplicite o implicite. In particolare, gli autori non garantiscono l’accuratezza, la completezza o l’assenza di errori nel contenuto del documento e declinano ogni responsabilità per eventuali danni derivanti dall’uso del materiale.

## **Testo completo della licenza**

Il testo legale completo della licenza Creative Commons Attribution–ShareAlike 4.0 International è disponibile al seguente indirizzo:

<https://creativecommons.org/licenses/by-sa/4.0/>

Copyright © 2026  
Emanuele Galiano

# Indice

<b>Licenza</b>	<b>iii</b>
Diritti concessi . . . . .	iii
Condizioni . . . . .	iii
Assenza di garanzia . . . . .	iv
Testo completo della licenza . . . . .	iv
<b>1 Descrizione dei SO, Hardware e Virtualizzazione</b>	<b>1</b>
1.1 Introduzione . . . . .	1
1.1.1 Definizione di sistema operativo . . . . .	1
1.1.2 Il sistema operativo come macchina estesa e gestore di risorse . . . . .	3
1.2 Analisi delle componenti di un calcolatore . . . . .	4
1.2.1 Processore . . . . .	4
1.2.2 Scheda video (GPU) . . . . .	8
1.2.3 Memoria . . . . .	8
1.2.4 Chiamate di sistema . . . . .	10
1.2.5 Dispositivi di I/O . . . . .	10
1.2.6 Bus . . . . .	12
1.3 Panoramica e struttura dei sistemi operativi . . . . .	13
1.3.1 Tipologie di sistema operativi . . . . .	13
1.3.2 Struttura di sistema operativi . . . . .	15
1.4 Virtualizzazione . . . . .	17
1.4.1 Macchine virtuali . . . . .	17
1.4.2 Paravirtualizzazione . . . . .	18
1.4.3 Hypervisor . . . . .	19
<b>2 Processi, Thread, IPC e Scheduling</b>	<b>21</b>
2.1 Processi . . . . .	21
2.1.1 Spazio di indirizzamento . . . . .	21
2.1.2 Tabella dei processi . . . . .	21
2.1.3 Pseudo-parallelismo . . . . .	22
2.1.4 Creazione di un processo . . . . .	22
2.1.5 Terminazione di un processo . . . . .	22
2.1.6 Stati di un processo . . . . .	23
2.1.7 Interrupt e prelazione . . . . .	23
2.1.8 Accodamento dei processi . . . . .	24
2.1.9 Comunicazione tra processi . . . . .	25

---

2.2	Thread . . . . .	26
2.2.1	Thread e risorse condivise . . . . .	26
2.2.2	Operazioni sui thread . . . . .	27
2.2.3	Programmazione multithread e multicore . . . . .	27
2.2.4	Tipologie di thread . . . . .	28
2.2.5	Implementazioni di thread nei sistemi operativi moderni . . . . .	30
2.3	Sincronizzazione nel parallelismo . . . . .	30
2.3.1	Race conditions . . . . .	30
2.3.2	Mutua esclusione e sezioni critiche . . . . .	31
2.3.3	Requisiti per una soluzione corretta . . . . .	31
2.4	Come realizzare la mutua esclusione . . . . .	31
2.4.1	Disabilitare gli interrupt . . . . .	31
2.4.2	Variabili di lock . . . . .	31
2.4.3	Alternanza stretta . . . . .	32
2.4.4	Soluzione di Peterson . . . . .	32
2.4.5	Istruzioni TSL (Test-and-Set) e XCHG . . . . .	33
2.4.6	Sleep e <i>wakeup</i> . . . . .	34
2.4.7	Semafori . . . . .	34
2.4.8	Tipologie dei semafori . . . . .	35
2.4.9	Mutex . . . . .	35
2.4.10	Monitor . . . . .	36
2.5	Problemi classici di sincronizzazione . . . . .	38
2.5.1	Produttore - Consumatore . . . . .	38
2.5.2	Problema dei 5 filosofi . . . . .	39
2.5.3	Problema dei lettori e scrittori . . . . .	42
2.6	IPC: InterProcess Communication . . . . .	44
2.6.1	Primitive di base . . . . .	45
2.6.2	Buffering e sincronizzazione . . . . .	45
2.6.3	Barriere . . . . .	45
2.6.4	Indirizzamento: diretto vs indiretto . . . . .	45
2.6.5	Esempi . . . . .	46
2.7	Scheduling . . . . .	47
2.7.1	Introduzione e obiettivi dello scheduling . . . . .	47
2.7.2	Scheduling per sistemi batch . . . . .	48
2.7.3	Scheduling nei sistemi interattivi . . . . .	49
2.8	Scheduling nei thread . . . . .	52
2.8.1	Thread utente . . . . .	52
2.8.2	Thread kernel . . . . .	52
2.9	Scheduling su sistemi multiprocessore . . . . .	52
2.10	Scheduler nei moderni sistemi operativi . . . . .	52
<b>3</b>	<b>Gestione della memoria</b>	<b>55</b>
3.1	Gerarchia della memoria . . . . .	55
3.2	Gestione dei processi in memoria centrale . . . . .	55
3.2.1	Rilocazione . . . . .	56
3.3	Spazio degli indirizzi . . . . .	57

3.4	Swapping . . . . .	58
3.4.1	Rilocazione e ripresa. . . . .	58
3.4.2	I/O pendenti. . . . .	58
3.4.3	Allocazione della memoria e contiguità. . . . .	59
3.5	Gestione dell'allocazione . . . . .	59
3.5.1	Unità minima allocabile: compromessi . . . . .	59
3.5.2	Bitmap . . . . .	60
3.5.3	Liste di partizioni . . . . .	60
3.5.4	Politiche di posizionamento (scelta del buco) . . . . .	60
3.5.5	Ottimizzazioni pratiche . . . . .	61
3.6	Memoria virtuale . . . . .	61
3.6.1	Paginazione . . . . .	62
3.6.2	Uso di una tabella di pagine . . . . .	63
3.6.3	Dettagli di una voce della tabella delle pagine (PTE) . . . . .	64
3.6.4	Tabella dei frame . . . . .	65
3.6.5	Progettazione di una tabella delle pagine . . . . .	65
3.6.6	TLB (memoria associativa) . . . . .	66
3.6.7	EAT: Effective Access Time . . . . .	67
3.6.8	Cache della memoria vs. memoria virtuale . . . . .	70
3.7	Algoritmi di sostituzione delle pagine . . . . .	72
3.7.1	Not Recently Used (NRU) . . . . .	72
3.7.2	Algoritmo FIFO e “seconda chance” . . . . .	72
3.7.3	Clock (seconda chance con lista circolare) . . . . .	73
3.7.4	Least Recently Used (LRU) . . . . .	73
3.7.5	Not Frequently Used (NFU) . . . . .	75
3.7.6	Aging . . . . .	75
3.7.7	Analisi delle prestazioni . . . . .	76
3.8	Allocazione dei frame . . . . .	79
3.8.1	Minimo strutturale di frame . . . . .	79
3.8.2	Strategie di caricamento iniziale . . . . .	80
3.8.3	Politiche di ripartizione dei frame tra processi . . . . .	80
3.8.4	Ambito del rimpiazzo: locale vs globale . . . . .	80
3.8.5	Trashing e controllo del grado di multiprogrammazione . . . . .	80
3.8.6	Osservazioni conclusive . . . . .	81
3.8.7	Modello di località . . . . .	81
3.8.8	Working set . . . . .	81
3.8.9	Page-Fault Frequency (PFF) . . . . .	82
3.9	Pagine condivise . . . . .	83
3.9.1	Condivisione del codice rientrante . . . . .	83
3.9.2	Memoria condivisa esplicita (IPC) . . . . .	83
3.9.3	Funzionamento con tabelle multilivello . . . . .	84
3.9.4	Limiti con <i>inverted page table</i> (aliasing) . . . . .	84
3.9.5	Copy-on-Write (COW) . . . . .	85
3.9.6	Zero-fill-on-demand . . . . .	86
3.9.7	Librerie condivise con linking . . . . .	87
3.9.8	Librerie condivise con file mappati . . . . .	87

---

3.10	Allocazione della memoria per il kernel . . . . .	88
3.10.1	Slab allocator . . . . .	88
<b>4</b>	<b>File System e Dischi</b>	<b>89</b>
4.1	File . . . . .	89
4.1.1	Denominazione di file . . . . .	89
4.1.2	Estensioni . . . . .	89
4.1.3	Struttura . . . . .	89
4.1.4	Tipologie . . . . .	90
4.1.5	Accesso ai file . . . . .	92
4.1.6	Attributi dei file . . . . .	93
4.1.7	Operazioni sui file . . . . .	93
4.2	Directory . . . . .	93
4.2.1	Sistemi di directory . . . . .	94
4.2.2	Nomi e percorsi . . . . .	94
4.2.3	Operazioni sulle directory . . . . .	95
4.3	File System . . . . .	95
4.3.1	Definizione . . . . .	95
4.3.2	Scopo e utilizzi . . . . .	95
4.3.3	Layout del file system . . . . .	95
4.3.4	Implementazione di file . . . . .	97
4.3.5	Implementazioni di directory . . . . .	99
4.3.6	File condivisi . . . . .	101
4.3.7	Gestione dei blocchi (spazio su disco) . . . . .	101
4.3.8	Controlli di consistenza . . . . .	103
4.3.9	Journaling . . . . .	103
4.3.10	Uso della cache . . . . .	104
4.3.11	Sistemi operativi moderni . . . . .	105
4.4	Dischi . . . . .	106
4.4.1	Architettura di un disco . . . . .	106
4.4.2	Frammentazione . . . . .	106
4.4.3	Scheduling del disco . . . . .	106
4.4.4	RAID . . . . .	108
4.5	SSD . . . . .	111
4.5.1	Organizzazione (pagine e blocchi). . . . .	112
4.5.2	Implicazioni per il file system. . . . .	112
4.5.3	Flash Translation Layer (FTL). . . . .	112
4.5.4	Garbage collection e comando TRIM. . . . .	112
4.5.5	Accessi sequenziali e casuali negli SSD . . . . .	112
<b>5</b>	<b>Esercizi</b>	<b>115</b>
5.1	File system . . . . .	115
5.1.1	FAT . . . . .	115
5.1.2	File system UNIX con i-node . . . . .	116
5.2	Dischi . . . . .	117
5.2.1	Scheduling su disco con politica LOOK . . . . .	117

5.3	Thread . . . . .	117
5.3.1	Attesa con semafori . . . . .	117
5.4	Paginazione . . . . .	118
5.4.1	Numero di pagine virtuali . . . . .	118
5.4.2	Numero di pagine virtuali con tabella multilivello . . . . .	119
5.4.3	Paginazione a due livello con EAT . . . . .	119
5.4.4	Altro su i-node . . . . .	120
5.4.5	I-node con indiretto singolo/doppio e calcolo dei frame fisici . . . . .	121
5.5	Frame fisici . . . . .	122
5.5.1	Allocazione uguale e proporzionale . . . . .	122



# Capitolo 1

## Descrizione dei SO, Hardware e Virtualizzazione

### 1.1 Introduzione

#### 1.1.1 Definizione di sistema operativo

Un moderno calcolatore è costituito da uno o più processori, una quantità variabile di memoria centrale, dischi fissi o unità flash e diverse possibili periferiche esterne.

Non tutti gli informatici sono in grado di comprendere il funzionamento di tutte le possibili periferiche a basso livello che esistono, come tutti i componenti del calcolatore; per questo si utilizzano i **sistemi operativi**, il cui compito è di fornire ai programmi utente un modello del computer migliore, più semplice e più chiaro e di gestire le risorse del calcolatore sopracitate.

Nella figura 1 viene rappresentata una panoramica generale dei componenti principali dell'architettura di un sistema operativo.

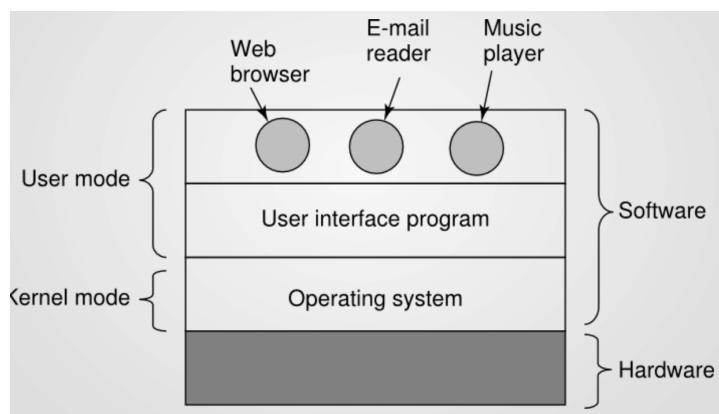


Figura 1.1: Architettura a strati del sistema operativo

Il sistema operativo è il componente software basilare, che sta esattamente sopra all'hardware, e viene eseguito in **modalità kernel** (anche chiamata modalità supervisor) per quanto riguarda la parte delle sue funzionalità. Questa modalità ha il totale accesso non limitato a tutto l'hardware e può eseguire qualsiasi istruzione eseguibile dalla macchina.

Il resto del software viene eseguito in **modalità utente**, nella quale è disponibile solo un sottoinsieme di istruzioni macchina, in particolare quelle che non pregiudicano il controllo della macchina, non

determinano i limiti della sicurezza e non gestiscono **I/O** (input/output). Il sistema operativo fornisce quindi due operazioni:

1. da una parte fornisce **un insieme semplice e pulito di risorse astratte**, anziché il chaos delle risorse hardware.
2. dall'altra **gestisce le risorse hardware** in questione.

### 1.1.2 Il sistema operativo come macchina estesa e gestore di risorse

Il sistema operativo agisce come una macchina estesa al calcolatore, ovvero: **nasconde la complessità dell'hardware e fornisce astrazioni più semplice e comode** da usare per programmati e utenti. Come esempio, basti pensare alla scrittura su memoria secondaria: anziché essere grezza (ossia scrivendo byte per byte) si organizza la memoria in file organizzati in directory scritte da processi.

Per astrazione si intende quindi nascondere i dettagli complessi e mostrare i dettagli essenziali e rilevanti per l'utente (come si vede nella figura 2). L'astrazione è quindi la chiave per gestire la complessità.

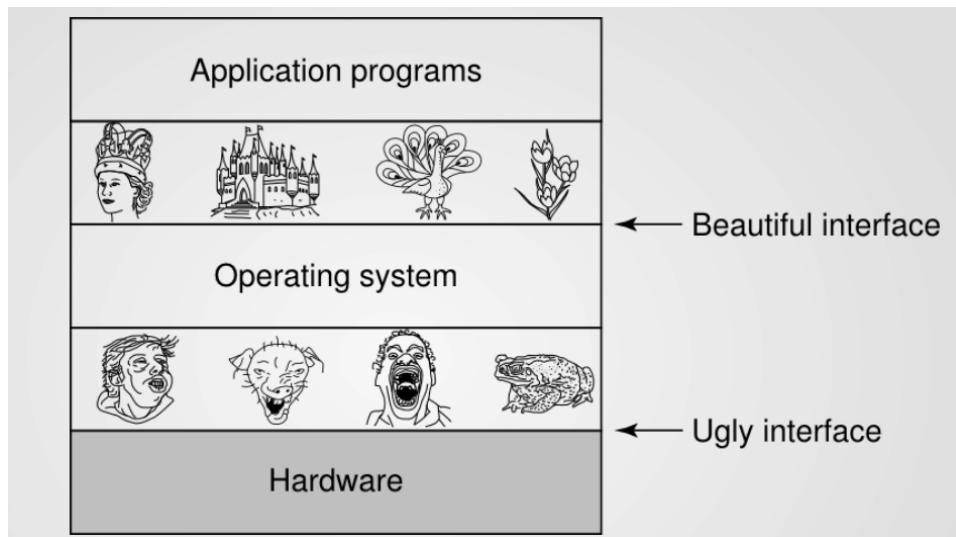


Figura 1.2: Interfaccia prodotta dal sistema operativo

I sistemi operativi permettono di gestire e suddividere, ove possibile, le risorse (ovvero di implementare le astrazioni) e quindi di eseguire più programmi contemporaneamente.

*Per multiprogrammazione si intende un calcolatore chiamato a eseguire più programmi contemporaneamente.*

Quando un calcolatore possiede più processi diventa ancora più difficile gestire le risorse, poiché non condividono solo l'hardware ma anche la memoria. In generale c'è il rischio che interferiscano l'uno con l'altro. La condivisione delle risorse avviene in due modalità diverse:

- **Multiplexing nel tempo:** quando programmi diversi condividono la stessa risorsa temporalmente, il sistema operativo alloca la CPU prima a un programma, poi dopo un intervallo di tempo sufficiente ad un altro programma che ne prende possesso e così via. Questa tecnica di timesharing dà all'utente un'illusione di parallelismo (pseudoparallelismo visto che la CPU è una sola).

Determinare per quanto tempo un programma può utilizzare la risorsa spetta al sistema operativo. Per salvare lo stato di un processo in esecuzione, il sistema utilizza una tecnica chiamata **Context Switch**.

- **Multiplexing nello spazio:** ogni cliente ottiene una parte della risorsa. Ad esempio, nella memoria centrale possono esserci più programmi memorizzati contemporaneamente, anche perché non avrebbe senso allocare tutta la memoria solo ad un programma, specialmente se ne occupa solo una frazione.

## 1.2 Analisi delle componenti di un calcolatore

### 1.2.1 Processore

#### 1.2.1.1 Cicli

Il cuore di ogni CPU è un ciclo che si ripete all'infinito di 3 fasi essenziali:

1. **Fetch**: preleva dalla memoria l'istruzione all'indirizzo indicato dal registro **PC** (Program Counter) e incrementa il PC.
2. **Decode**: l'unità di controllo interepta l'opcode, individua eventuali operandi e predisponde le unità funzionali internet.
3. **Execute**: l'ALU, o altri blocchi, eseguono l'operazione e, se necessario, vengono letti/scritti registri o memoria primaria.

#### 1.2.1.2 Registri

Questo ciclo viene governato da 3 registri con funzioni precise:

- **PC** (Program Counter): contiene l'indirizzo della prossima istruzione da eseguire.
- **SP** (Stack Pointer): Punta al top dello stack corrente; è utilizzato per chiamate di procedure, salvataggio di contesti, variabili locali ecc.
- **PSW** (Program Status Word): contiene i bit con il codice di condizione, impostati da istruzione di confronto, la priorità della CPU, la modalità (utente o kernel) e altri diversi bit di controllo.

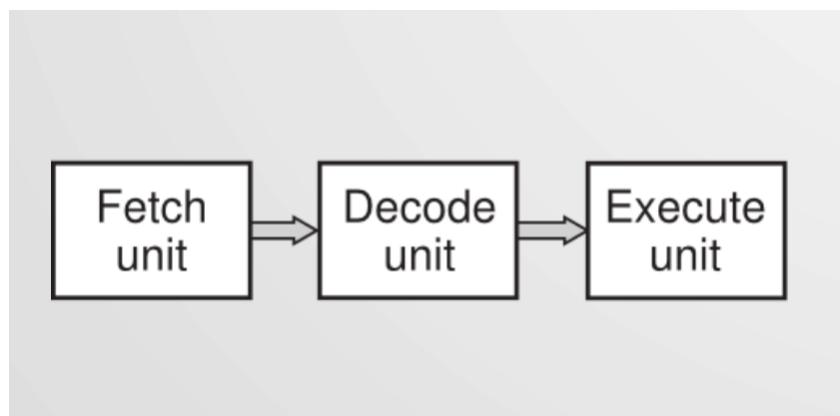


Figura 1.3: Fasi del processore

Durante un **Context Switch** il sistema operativo salva e ripristina *tutti* questi registri per consegnare alla CPU l'immagine esatta del processo successivo.

#### 1.2.1.3 Pipeline e processori multi-scalari

I moderni processori non aspettano, però, che un'istruzione termini per iniziare la successiva: suddividono il ciclo fetch-decode-execute in stadi. In questo modo:

- ogni clock può trovarsi simultaneamente a **diversi stadi di esecuzione** di istruzioni distinte.
- si ottiene un throughput prossimo a una istruzione per ciclo (salvo stalli).

Questa viene chiamata **pipeline** e sono possibili grazie alla separazione delle componenti: per esempio si può avere una CPU separata da prelievo, decodifica ed esecuzione delle istruzioni, così che mentre sta eseguendo l'istruzione  $n$  può decodificare l'istruzione  $n + 1$  e prelevare l'istruzione  $n + 2$ .

Quando la pipeline da sola non basta, i core moderni adottano architetture **superscalari**: più unità funzionali permettono di emettere diverse istruzioni per ciclo. Per il sistema operativo questo significa che:

- Maggiore complessità nel calcolo delle prestazioni reali.
- Necessità di disabilitare o svuotare certe cache/pipeline in evento critici (es. context switch).

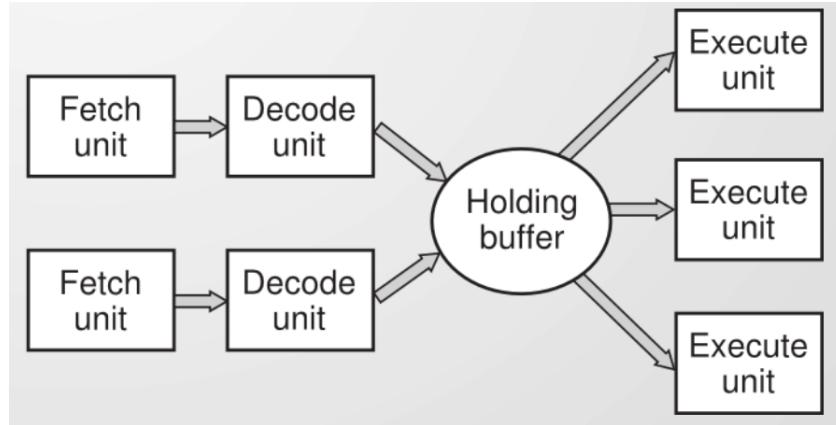


Figura 1.4: Fasi di un processore superscalare

#### 1.2.1.4 Multi-threading e multicore

Il multithreading e il multiprocessore/multicore sono sistemi di esecuzione in parallelo delle istruzioni, in particolare:

- Il multithreading è l'esecuzione di più **thread** (flussi di controllo) all'interno dello stesso processo o dello stesso core. L'hyper-threading hardware è un particolare tipo di multi-threading dove vengono fatti avanzare più thread in modo alternato.
- Il multicore è un sistema con più unità di **elaborazioni fisiche** (core) all'interno dello stesso processore. Tutti i core lavorano così in parallelo
- Il multiprocessore è un sistema dove, nella scheda madre, sono presenti più processori.

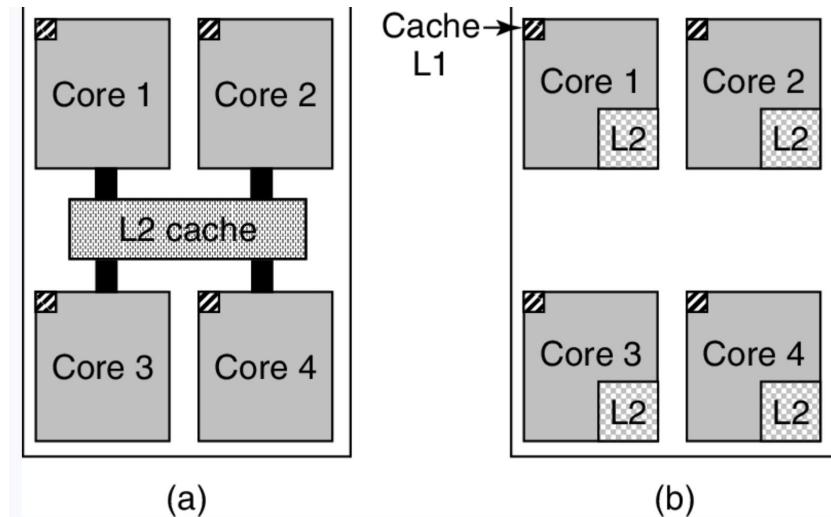


Figura 1.5: (a) Un chip a quattro core con memoria cache di tipo L2 condivisa. (b) Un chip a quattro core con memoria cache di tipo L2 separate.

Aspetto	Multithreading	Multicore
Tipo di parallelismo	Logico: thread alternati sullo stesso core	Fisico: core distinti realmente in parallelo
Granularità	Stesso spazio indirizzi; cambiano solo PC, registri, stack	Ogni core ha registri/cache propri; RAM condivisa
Vantaggi	Maschera latenze; usa meglio la pipeline; cambio rapido	Alto throughput; economia di scala; affidabilità
Svantaggi	Nessun boost su job CPU-bound; contesa cache/ALU	Maggiore complessità; serve codice parallelizzabile
Ruolo del SO	Scheduler thread; gestione lock e contesti	Load-balancing fra core; affinità cache e sync
Quando scegliere	App interattive e server I/O-bound leggeri	Calcolo scientifico, ML, database

Tabella 1.1: Confronto tra multithreading e multiprocessore

### 1.2.2 Scheda video (GPU)

Un altro tipo di parallelismo è legato alla **GPU** (Graphic Processing Unit). Sono composte da migliaia di core e si occupano dell'esecuzione parallela di piccoli calcoli.

Ciascuna unità di elaborazione ha una piccola memoria indipendente (limitata ma veloce), ma può accedere anche a una memoria condivisa. Il sistema operativo non fa quasi nulla di suo per gestire questa esecuzione in parallelo.

### 1.2.3 Memoria

Un altro componente principale di ogni calcolatore è la memoria. Idealmente, la memoria dovrebbe essere estremamente veloce (più veloce dell'esecuzione di un'istruzione in modo da non bloccare la CPU), molto capiente e a basso costo ma nessuna tecnologia attuale soddisfa questi tre requisiti in contemporanea; vi è una molteplicità di dispositivi adibiti alla memorizzazione con caratteristiche distinte organizzate in modo gerarchico.

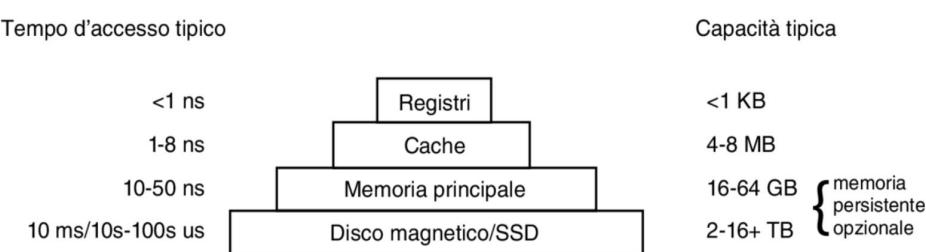


Figura 1.6: Gerarchia delle memoria in ordine crescente

#### 1.2.3.1 Registri

Il primo strato è quello dei registri interni alla CPU. L'accesso non comporta alcun ritardo, dato che la memoria è estremamente veloce (trovandosi anche nello stesso chip, non subendo quindi ritardi dovuti alla distanza); tuttavia è anche quella con una capacità più piccola (1 word).

#### 1.2.3.2 Cache

Il secondo strato è quello della cache, principalmente controllata dall'hardware. Essa ha il compito di mitigare i tempi lenti di accesso alla RAM. Ci sono una serie di locazioni dedicate a copie di informazioni: quando un programma ha bisogno di leggere una parola dalla memoria, l'hardware della cache controlla se già l'informazione richiesta è contenuta.

La natura dell'elaborazione dei nostri calcolatori si basa sul **principio di località**: il codice tende ad eseguire ciclicamente una manciata di informazioni in posizioni vicine, quando una nuova linea di cache viene portata al suo interno bisogna decidere cosa buttare fuori (se la cache è piena) e in molti casi si scarta l'informazione **meno usata di recente** (LRU).

Esistono diversi livelli di memoria cache (aumentando la capienza diminuisce la velocità di accesso). Tipicamente ogni core ha una cache e quando ci sono più core essi possono condividere un'unica cache L2 o averne una ciascuno (rispettivamente approccio di Intel e AMD). Nel primo caso si possono verificare problemi legati alla concorrenza del parallelismo, nel secondo caso è importante che le progettazioni delle cache siano coerenti tra loro: supponiamo che il core A e il core B condividano una variabile x; quando A fa riferimento per la prima volta ad x, la linea che contiene viene inserita

nella cache di A (lo stesso vale per B); A modifica x, ma se la modifica rimane locale alla cache di A, B continuerà a vedere il valore originale. Di conseguenza le modifiche andrebbero propagate alle altre cache.

### 1.2.3.3 Disco

Cache e registri sono volatili, motivo per cui si usufruisce di memoria di massa come i dischi (gestiti dal filesystem). I tempi di accesso sono più lunghi ma la capienza è maggiore. Non si può fare riferimento diretto a un'informazione su disco, quindi prima essa va spostata nella RAM. Il disco meccanico è composto da un braccio e da dei piatti metallici disposti uno sopra l'altro che girano a migliaia di giri al minuto. Ogni piatto viene diviso in regioni anulari dette tracce e tutte le tracce di una certa posizione del braccio compongono un cilindro. Ogni testina del braccio legge una certa traccia e a loro volta le tracce sono divise in settori numerati, dove ognuno di loro corrisponde a delle coordinate. Il sistema operativo farà in modo che i processi vedano questi settori, disposti normalmente in maniera "dimensionale" in modo linearizzato (astrazione).

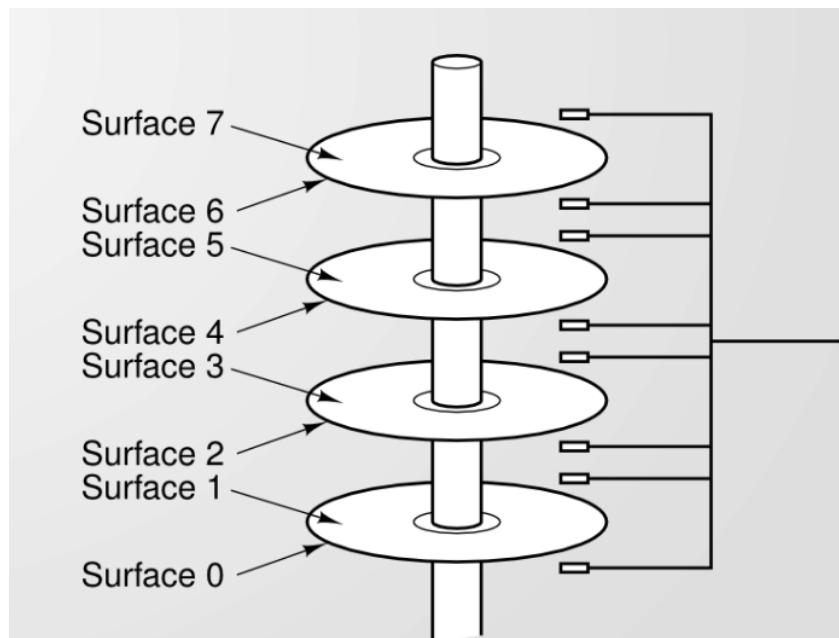


Figura 1.7: Superfici di un disco

Il tempo di prelievo è determinato da:

1. **Seek time:** tempo che impiega la testina per raggiungere la traccia.
2. **Tempo di rotazione:** tempo di cui ha bisogno il disco per posizionare l'informazione sotto la testina.
3. **Tempo di trasferimento:** tempo che impiega la testina a leggere dal disco.

E' possibile prelevare più informazioni in una volta, così il tempo di trasferimento e di rotazione siano unici per ogni informazione.

### 1.2.4 Chiamate di sistema

Sebbene le chiamate di sistema non siano un componente hardware, fanno parte ormai dello standard dei sistemi operativi moderni. Esse svolgono un compito molto importante, ovvero effettuare **le richieste** per ottenere servizi dal sistema operativo e quindi entrando in modalità kernel. Per fare ciò si utilizza l'istruzione **TRAP** che permette di passare dalla modalità utente alla modalità kernel. Una volta che il kernel ha completato il suo lavoro, il controllo ritorna al programma utente e si riparte dall'istruzione successiva alla syscall.

Dopo la chiamata di sistema, tutti i registri (sia generici che speciali) vengono salvati temporaneamente nello stack del kernel (quello stack utilizzato quando il SO è in modalità kernel), in modo che l'istruzione kernel abbia a disposizione tutti i registri liberi e che il contenuto originale di essi non venga intaccato dall'istruzione kernel.

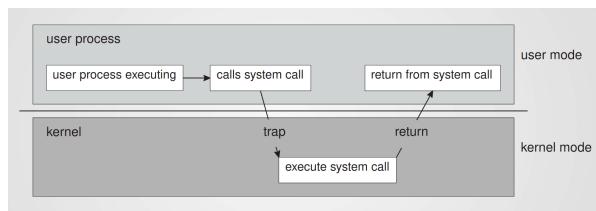


Figura 1.8: Schema di una chiamata di sistema

### 1.2.5 Dispositivi di I/O

Un dispositivo di I/O (input/output) possiede due componenti fondamentali

1. **Il dispositivo:** con una interfaccia elementare ma complicata da gestire, esponendo segnali e registri.
2. **Il controller:** un chip (o un insieme) che fisicamente controlla il dispositivo, accettando comandi dal sistema operativo (per esempio di leggere dati dal dispositivo e trasportarli).

E' importante che i dispositivi siano standard, in modo tale che un controller possa gestire qualunque dispositivo di quel tipo (esempio controller SATA con dischi).

Il sistema operativo, però, non può conoscere tutte le procedure del controller, è necessario un'interfaccia software che faccia da interprete tra controller e sistema operativo chiamato **driver** che solitamente viene fornito insieme al dispositivo. Il driver viene eseguito in modalità kernel, un processo utente normalmente non può infatti interagire con driver e controller (poiché le istruzioni di I/O sono privilegiate). E' possibile **mappare** le porte di I/O su alcuni indirizzi di memoria però, ovvero associarle ad alcune porte un indirizzo di memoria. Si tratta di una delega controllata che non necessita della modalità kernel, poiché non bisogna usare le istruzioni di I/O.

#### 1.2.5.1 Operazioni su I/O

Le operazioni sulla memoria secondaria sono molto lente rispetto alle possibilità della CPU, per questo si chiamano **bloccanti**. Quando un processo effettua una chiamata di sistema, esso viene messo in standby del sistema operativo e fa eseguire alla CPU altri processi nel frattempo.

Nel caso di operazioni di I/O ci sono 3 metodi diversi per eseguirle:

- **Busy Waiting:** un processo utente esegue una sys-call, il kernel poi traduce in chiamata di procedura al driver. Il driver attiva l'I/O e si sofferma in un ciclo interrogando di continuo il

controller per sapere se tutto sia stato eseguito (all'interno del controller o del dispositivo ci sono alcuni registri con dei bit che indicano se il dispositivo è occupato). Se l'operazione va a buon fine i dati vengono memorizzati nel buffer del controller e dopo in RAM. Il problema del busy waiting è che una re-iterazione di controlli e spreca risorse della CPU.

- **Interrupt:** quando il controller possiede l'informazione inoltra un'interrupt, che interromperà la CPU. La procedura sblocca il sistema operativo e gli comunica che l'operazione è andata a buon fine. La CPU preleva l'informazione dal buffer e la mette in RAM.

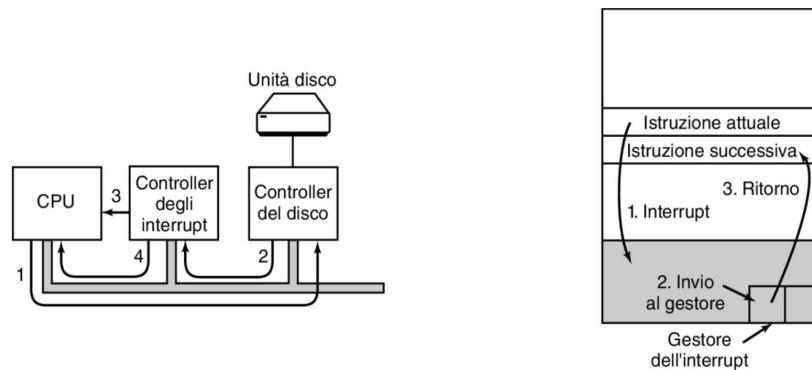


Figura 1.9: Operazioni su I/O con interrupt

- **DMA (Direct Memory Access):** normalmente il sistema operativo richiede il contenuto del buffer del controller tramite le porte di I/O. Questo lavoro viene bypassato tramite DMA. Il DMA permette di creare un canale diretto tra controller e RAM, senza usare costantemente la CPU e senza coinvolgere il sistema operativo.

La CPU imposta il chip DMA, comunicandogli quanti byte trasferire, il dispositivo e l'indirizzo di memoria coinvolti e la direzione di trasferimento. Quando il chip DMA ha finito, causa un interrupt, per comunicare che il trasferimento è avvenuto.

### 1.2.6 Bus

Durante l'avvento dei calcolatori era sufficiente un unico bus per far comunicare la CPU con le diverse componenti, tuttavia con l'aumento delle performance di quest'ultime è stato necessario inserire bus addizionali che legano componenti specifiche alla CPU. Attualmente un sistema x86 ha la struttura mostrata nella figura 9: bus di cache, bus memoria, PCIe, PCI, USB, SATA e DMI. Ognuno con una velocità di trasferimento e funzioni diverse e, per poter gestirli tutti, il sistema operativo deve esserne a conoscenza.

Il principale è il bus **PCIe** (Peripheral Component Interconnect express), un altro è il DMI (Direct Media Interface) che connette la CPU al controller delle porte I/O.

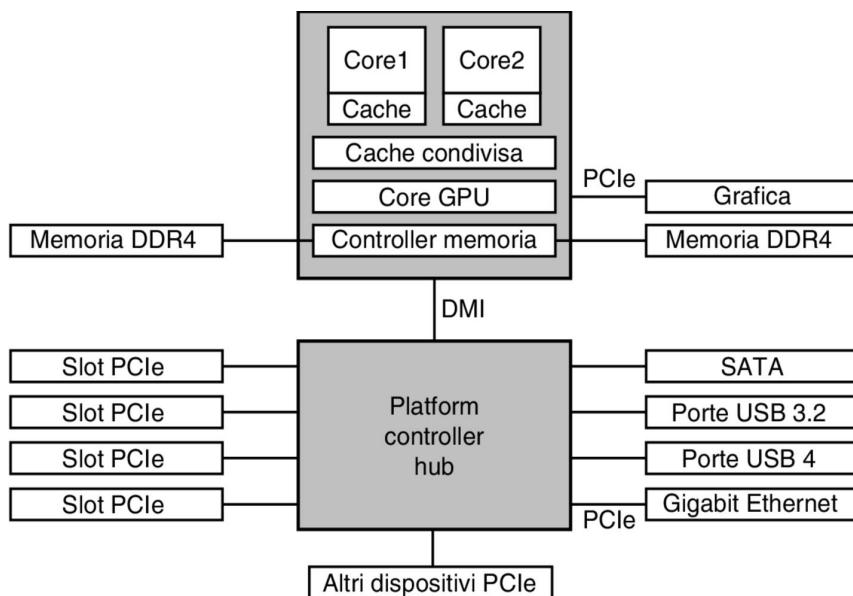


Figura 1.10: Bus di un sistema x86

I bus possono essere:

- **Parallel**: trasmettono ogni di bit di una word su una linea diversa, un esempio è PCI.
- **Serial**: trasmettono i bit di una word uno dietro l'altro, un esempio è USB.

Il PCIe è un caso particolare: permette di trasmettere più word diverse in parallelo usando linee diverse, che sono seriali.

## 1.3 Panoramica e struttura dei sistemi operativi

Esistono differenti tipologie di sistema operativo, in base al compito che essi devono svolgere.

### 1.3.1 Tipologie di sistema operativi

#### 1.3.1.1 Mainframe

Sono caratterizzati da compiti tipicamente assegnati a sistemi chiamati (in gergo) **batch**, ovvero che non richiedono l'interazione dell'utente. Sono composti da numero CPU e molte memorie secondarie (e in generale dispositivo di I/O); questi servizi necessitano di sistemi operativi specializzati, orientati all'esecuzione di numero lavori alla volta, molti dei quali necessitano di quantità enormi di lavori I/O in parallelo.

#### 1.3.1.2 Server

Si occupano di gestire uno o più servizi di varia natura gestiti attraverso richieste in una rete. Le richieste confluiscono attraverso una scheda di rete e devono essere gestite il più velocemente possibile.

#### 1.3.1.3 PC (Personal Computer)

I computer moderni supportando la multiprogrammazione e il loro scopo è fornire un valido supporto un singolo utente. Un PC fa un pessimo lavoro quando le applicazioni non reagiscono subito. Per raggiungere l'interattività e aumentare la reattività non serve un PC più veloce, perché molte delle caratteristiche che rallentano il programma dipendono dal sistema operativo. Esistono quindi dei processi nel sistema operativo usati proprio per l'interattività.

Dato che i PC saranno utilizzati da utenti comuni molto inesperti è necessario facilitarne l'uso (per esempio introducendo le GUI).

#### 1.3.1.4 Dispositivi mobile

Hanno esigenze simili a quelle dei PC con la differenza nella gestione dei permessi e delle risorse. Le GUI sono pensate per un tipo diverso di input e i processi sono completamente isolati tra loro. In base alla struttura del sistema operativo ci sono alcuni accorgimenti sull'operato del kernel.

#### 1.3.1.5 Sistemi embedded

Sono sistemi ancora riconducibili ad un calcolatore ma difficili da identificare come tali (per esempio, un router). All'interno esistono flussi di esecuzione e hanno capacità limitate rispetto ai PC. I loro sistemi operativi sono chiusi e pilotano pochi processi, già noti agli sviluppatori.

#### 1.3.1.6 Real-time

Generalmente sono sistemi operativi pensati per lavori di tipo industriale, legati al tipo di servizio che devono offrire. Devono infatti gestire una tempestiva risposta a dei casi specifici (come la saldatura di una catena di montaggio, non troppo tardi ma né troppo presto). Per garantire questa tempestività ci sono pochi processi ciascuno dei quali può usare la CPU per un determinato tempo ma al massimo delle sue capacità. Se un'azione deve necessariamente avvenire in un determinato momento si parla di sistemi real-time stretti.

Linux è un caso particolare: il modo in cui è stato progettato e implementato permette modularità, scalarità e flessibilità, che lo rendono adatto a gestire sistemi operativi mainframe, server, pc e dispositivi mobile.

### 1.3.2 Struttura di sistema operativi

#### 1.3.2.1 Monolitico

La struttura di tipo monolitico è poco articolata e coincide con i primi sistemi operativi. Tutto il codice è racchiuso all'interno di un unico blocco, quindi il kernel è contenuto tutto in un solo programma binario eseguibile. L'hardware non era molto complesso, anche se vi è un minimo di stratificazione e ogni componente è libero di usare tutti gli altri senza un'organizzazione specifica.

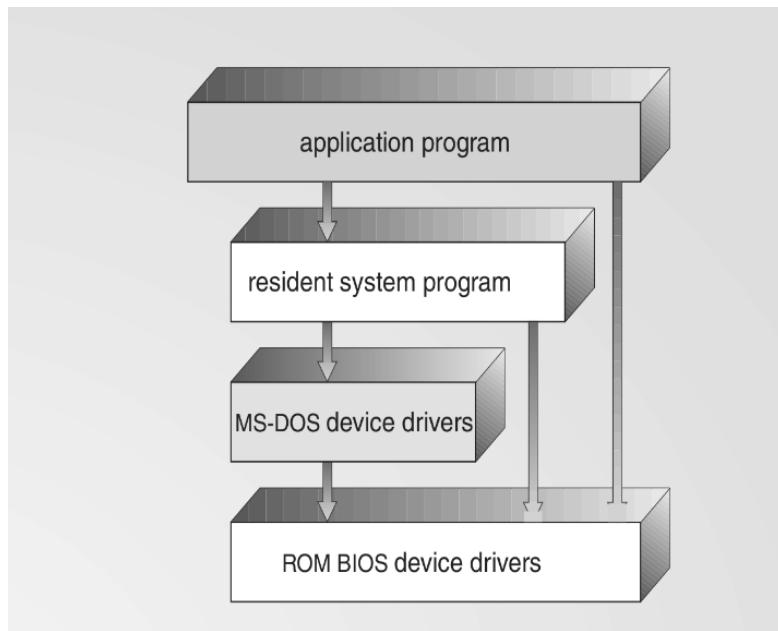


Figura 1.11: Struttura di un sistema operativo monolitico

#### 1.3.2.2 A livelli

Questa struttura utilizza una **gerarchia di livelli**, lo stesso concetto alla base della programmazione ad oggetti. La componente hardware può essere vista come uno strato su cui si piazza il software e ogni livello implementa una particolare funzione del sistema operativo. Si utilizza un approccio bottom-up: uno strato implementa un servizio usando ciò che offrono gli strati sottostanti e sviluppare la memoria al livello 1 è indipendente dallo sviluppare la CPU al livello 2 (si suppone la correttezza dello strato inferiore). Per ogni livello esistono chiamate di invocazione e se a un'app serve un servizio di livello più basso deve passare per tutti i livello intermedi.

Poiché ogni strato ha una virtual CPU, può avvenire l'esecuzione in parallelo, perché lo strato della CPU dovrebbe ovviamente trovarsi sopra quello della virtual memory altrimenti non potrebbe appoggiarsi ad essa.

Tra i problemi principali di questa struttura vi sono la decisione dell'ordine dei livelli e le chiamate nidificate, che creano **overhead**. Per risolvere, si può utilizzare:

- **Anelli concentrici:** I livelli sono fisicamente separati tra loro, non solo a livello di vincoli dati al programmatore ma anche grazie al punto di vista dell'hardware. Grazie a questa separazione forzata dell'hardware, ciascuno strato presenta un livello di protezione diverso (come si vede in figura 1.12).

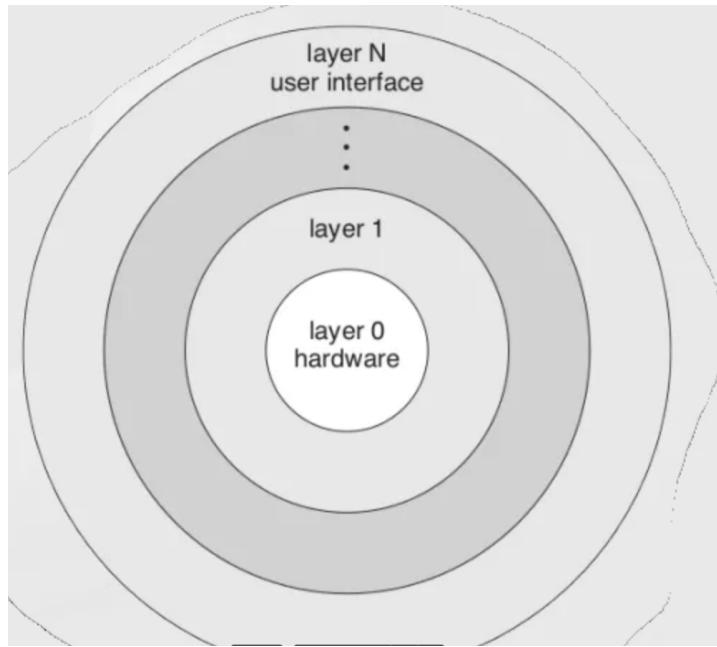


Figura 1.12: Un sistema ad anelli concentrici

- **Microkernel:** l'idea alla base è quella di suddividere il sistema operativo in numerosi moduli di piccole dimensioni, facendo sì che solo le funzionalità strettamente necessarie vengano eseguite in modalità kernel mentre le altre verranno eseguite in modalità utente. Tutti i frammenti di codice sono piccoli e ben studiati, e ognuno di loro svolge un compito semplice, in questo modo è difficile che si presentino bug, e anche se dovesse succedere, dato che ciascun processo è isolato dagli altri, esso non va a danneggiare le altre componenti.

Uno dei problemi è, che quando il microkernel riceve una chiamata di sistema, la TRAP può far partire uno o più messaggi ad altri servizi, e questo può introdurre un overhead oltre il desiderabile. Al di fuori del kernel, il sistema è strutturato in tre strati di processi che girano tutti in modalità utente:

1. **Driver dei dispositivi:** visto che girano in modalità utente, non hanno accesso fisico allo spazio della porta di I/O e non possono inviare comandi di I/O direttamente. Invece, per programmare un dispositivo di I/O, il driver costruisce una struttura che indica quali valori scrivere e in che porte di I/O e fa una chiamata al kernel richiedendogli di effettuare la scrittura. In questo modo il kernel controlla se il driver è autorizzato a usare quello che sta scrivendo o leggendo usando i dispositivi di I/O e di conseguenza un driver audio difettoso non può accidentalmente scrivere sul disco.
2. **Server:** questo livello compone la maggior parte del lavoro del sistema operativo. Uno o più server gestiscono il file system, creano il gestore del processo, cancellano e gestiscono processi e così via. I programmi utente ottengono servizi del sistema operativo inviando brevi messaggi ai server. Il **reincarnation server** è uno di questi, che gestisce il corretto funzionamento dei servizi sostituendolo, nel caso sia guasto, immediatamente.

Il microkernel può concedere a un servizio in modalità utente di interagire con il controller, tramite una mappatura delle porte. Il microkernel manda alcune porte a quello specifico processo e dopodiché esso viene avviato.

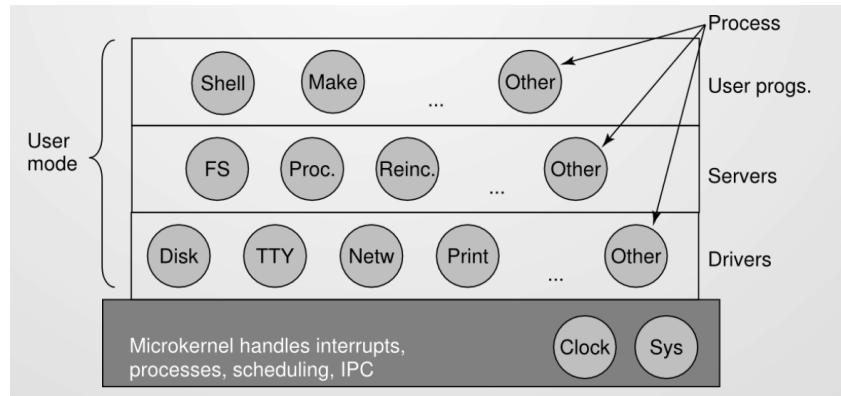


Figura 1.13: Un sistema con microkernel

- **Struttura a moduli**: l'idea di questa struttura è quella di avere un kernel in cui le varie funzionalità vengono incapsulate in moduli. Un modulo è una piccola componente che si dà un compito (come lo scheduling, il file system etc). Una volta che il modulo si trova in memoria diventa parte del sistema operativo e viene eseguito in modalità kernel. Questo aumenta l'efficienza del meccanismo di chiamata e dal punto di vista dell'organizzazione, ma così un bug potrebbe avere un impatto maggiore.

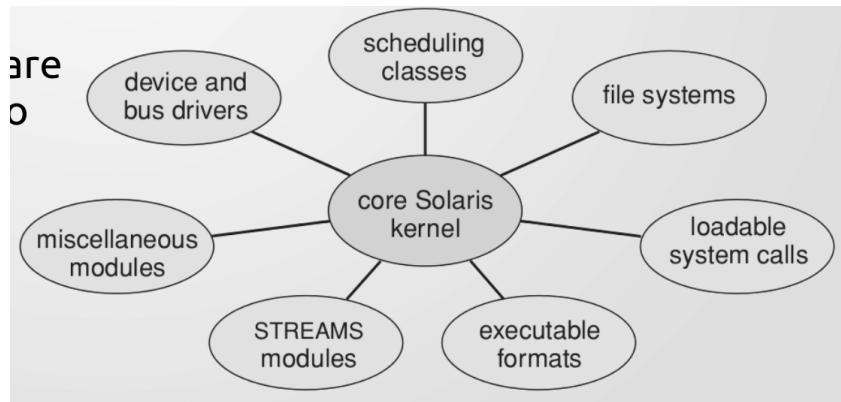


Figura 1.14: Un sistema a moduli

## 1.4 Virtualizzazione

### 1.4.1 Macchine virtuali

Partendo dall'idea che il sistema operativo sia un **fornitore di astrazioni** che, se estremizzata, porta alla virtualizzazione. In una macchina virtuale è possibile far girare un altro sistema operativo diverso da quello ospitante. La virtualizzazione offre ai developer ambienti coerenti tra sistemi operativi diversi e, soprattutto, maggior sicurezza grazie all'isolamento: se un servizio vulnerabile (es. un web server) viene compromesso su una singola macchina che ospita tutto con lo stesso utente, l'attaccante può muoversi lateralmente ed effettuare escalation fino a root; separando i servizi in VM distinte, il danno resta circoscritto. Nella virtualizzazione, il codice del guest esegue direttamente sulla CPU reale (con overhead ridotto) mentre vengono virtualizzati soprattutto i dispositivi di I/O; le complessità emergono quando si interagisce con periferiche virtuali. L'emulazione, invece, simula via software una CPU diversa e l'hardware sottostante, con costi prestazionali sensibilmente superiori.

### 1.4.2 Paravirtualizzazione

Nella virtualizzazione pura, l'astrazione della macchina è equivalente a una macchina reale. Viene emulato il software che non sa di essere emulato. Nella paravirtualizzazione, invece, il sistema ospitato è consapevole di essere virtualizzato e può comunicare con l'OS principale.

### 1.4.3 Hypervisor

L'Hypervisor è il software che si occupa della gestione della macchina virtuale. L'Hypervisor permette di eseguire la TRAP in modalità kernel virtuale, cioè simula la modalità kernel sul sistema operativo virtuale. Questo perché chiaramente il processo associato alla macchina virtuale sul sistema operativo reale viene eseguito in modalità utente.

#### 1. Hypervisor di tipo 1

Gli hypervisor di tipo 1 girano direttamente sull'hardware e svolgono il ruolo di un sistema operativo minimale per la gestione delle VM. In assenza di un OS ospitante, l'overhead (soprattutto in latenza) è inferiore e il controllo delle risorse è più diretto (Figura 1.15).

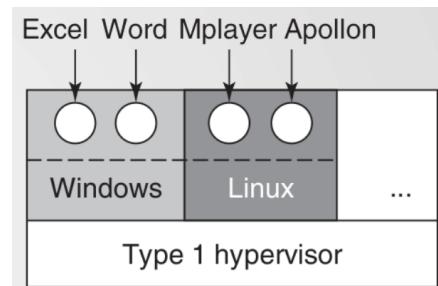


Figura 1.15: Schema di un hypervisor di tipo 1

#### 2. Hypervisor di tipo 2

Gli hypervisor di tipo 2 sono normali processi in *user space* che girano sopra un sistema operativo ospitante. Istanziando VM, delegano molte operazioni (I/O, gestione dispositivi, system call) ai servizi e ai driver dell'host. Le prestazioni dipendono fortemente dal supporto di virtualizzazione hardware della CPU e della piattaforma: in sua assenza l'overhead cresce; le CPU moderne lo offrono nativamente, migliorando la resa complessiva (Figura 1.16).

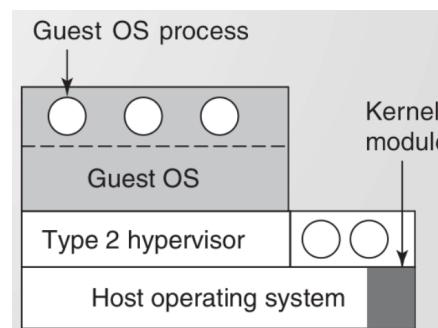


Figura 1.16: Schema di un hypervisor di tipo 2



## Capitolo 2

# Processi, Thread, IPC e Scheduling

### 2.1 Processi

#### 2.1.1 Spazio di indirizzamento

Un **processo** è un'istanza di esecuzione di un programma. In un programma posso avere più processi distinti e ogni processo è definito dallo **spazio di indirizzamento** (porzione di RAM riservata solo a lui). Questo spazio è diviso in:

- **Codice:** istruzioni macchina in sola lettura. Spesso condivisibile con altri processi che eseguono lo stesso binario.
- **Dati:** variabili globali/statiche, visibili a tutto il processo (e a tutti i suoi thread).
- **Heap:** area per allocazioni dinamiche (`malloc/new` e `free/delete`). Cresce/riduce a runtime; può soffrire di frammentazione e memory leak.
- **Stack:** frame di chiamata (return address, registri salvati, parametri, locali). Si espande/contrae a ogni call/return.

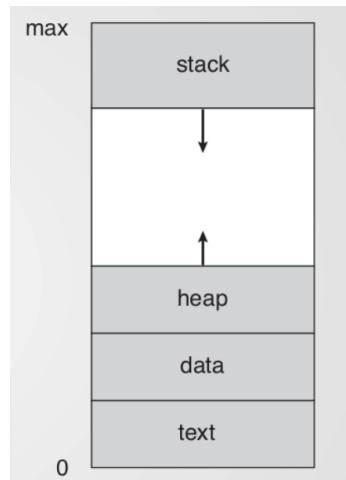


Figura 2.1: Spazio di indirizzamento di un processo

Non tutto quello che riguarda un processo sta in memoria, infatti lo stato di un processo si trova nei registri della CPU. Nel caso di CPU virtuali, i registri hanno informazioni sullo stato dei vari processi (PC e SP) e queste informazioni tornano utili per i context switch. Esistono altri componenti possibili di un processo, come i file (oppure gli allarmi): i file aperti dal processo saranno associati ad esso. In alcuni sistemi operativi però, esistono parentele tra i processi le quali comportano delle implicazioni. Anche queste informazioni fanno parte dello stato del processo.

#### 2.1.2 Tabella dei processi

Tutte le informazioni citate si trovano nel **PCB** (Process Control Block), la generica entry legata alla **tabella dei processi**. La tabella ha una entry per ogni processo attivo e gli indici della tabella specificano il PCB (di conseguenza anche il processo). L'indice viene detto **PID** (Process ID) e quando un processo termina il PCB si svuota e può essere riutilizzato. Ciò significa che il PID è univoco solo

nel momento in cui distingue tra tutti i processi attivi. Nello slot si possono avere anche riferimenti ad altre risorse del processo.

### 2.1.3 Pseudo-parallelismo

I processi non vengono eseguiti in maniera sequenziale, questo sarebbe molto sconveniente. Il sistema operativo infatti gestisce uno pseudo-parallelismo e si attiva tramite le CPU virtuali, dando l'illusione di parallelismo ma nella realtà quello che avviene è un interlacciamento dei processi (mentre si effettuano context-switch, Figura 2.2).

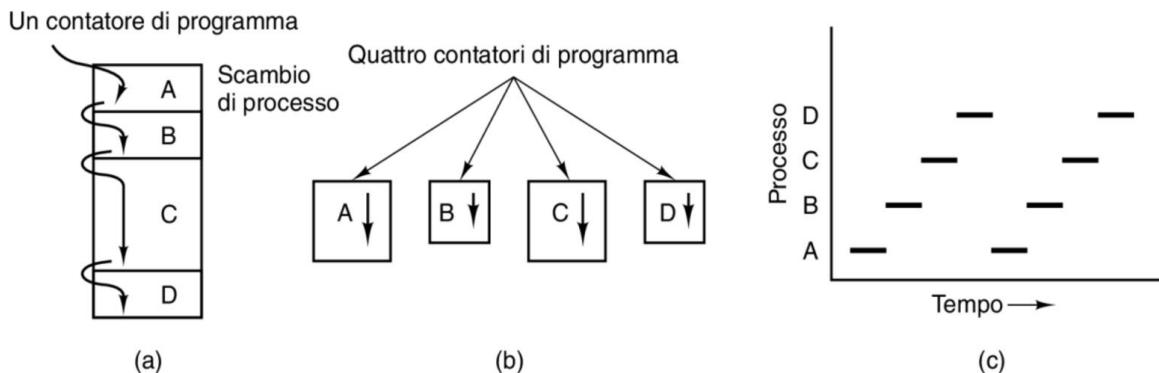


Figura 2.2: (a) Multiprogrammazione di quattro programmi. (b) Modello concettuale di quattro processi sequenziali indipendenti. (c) Solo un programma è attivo per volta

### 2.1.4 Creazione di un processo

Il primo processo si **autocrea** durante l'inizializzazione del sistema operativo e viene chiamato **INIT**. In genere, i processi vengono creati da altri processi. La creazione avviene tramite una chiamata di sistema, bisogna quindi passare dal sistema operativo e in base ad esso la creazione viene effettuata in maniera diversa:

- **Fork (UNIX)**: il comando `fork` ha il compito di creare un figlio F. F è un clone di suo padre P (il processo originale). Ci sono poche differenze, prevalentemente il PID e tutte le componenti sono sè e se dovessi modificare qualcosa in P non si ripercuoterà in F. Il padre può decidere cosa farà il figlio dopo essere nato. Insieme al comando `fork` esiste un'altra syscall `exec` che non crea un nuovo processo ma sostituisce l'immagine del processo chiamante con il nuovo programma indicato. Il PID rimane lo stesso ma vengono rimpiazzati codice, dati, bss, heap e stack.
- **CreateProcess (Windows)**: il comando `CreateProcess` è più semplice, ma crea ambienti più complessi di lavoro poiché crea un processo da 0 senza clonarlo da altri, specificando quale codice dovrà eseguire e gli altri dati necessari. È un procedimento più intuitivo ma non si può fare lo stesso procedimento di `exec`.

Entrambi i sistemi operativi, una volta creato il processo, assegnano uno spazio di indirizzi privato.

### 2.1.5 Terminazione di un processo

Un processo può terminare in vari modi. In pratica `exit()` invoca la syscall di uscita, quindi possiamo considerarla una chiamata di sistema. Dopo la terminazione il processo resta **zombie** finché il padre

non chiama `wait()`/`waitpid()`.

- **Uscita ordinaria (volontaria):** il programma chiude con `exit(status)` o rientrando da `main`. Il sistema rilascia le risorse. Su Windows si usa `ExitProcess`.
- **Uscita su errore (volontaria):** il programma rileva un problema, mostra un messaggio e termina con codice diverso da 0.
- **Errore critico (involontario):** istruzione non valida, accesso a memoria non valida o divisione per zero. Il sistema invia un segnale e il processo può chiudere con un *core dump*.
- **Terminato da un altro processo (involontario):** un altro processo invia `kill(pid, segnale)`. `SIGTERM` è un soft kill e permette chiusura pulita. `SIGKILL` è un hard kill e interrompe subito.

### 2.1.6 Stati di un processo

Ogni processo può assumere diversi stati:

- **New:** il kernel sta creando il PCB e inizializzando le strutture, prima che il processo entri nello scheduler
- **Ready:** è pronto a essere eseguito non appena una CPU si libera; i PCB dei pronti sono accodati in una o più code dei pronti da cui lo scheduler seleziona il prossimo
- **Running:** sta eseguendo sulla CPU; con una sola CPU c'è un solo processo in esecuzione, con  $N$  core fino a  $N$  processi contemporaneamente; il dispatcher gestisce il context switch salvando e ripristinando i registri dal PCB
- **Blocked:** è in attesa di un evento, tipicamente il completamento di un'operazione di I/O o l'esito di una chiamata bloccante (`wait()` su un figlio, primitive di sincronizzazione); non usa la CPU anche se è libera e tornerà in **ready** quando l'evento si verifica
- **Terminated:** ha finito, resta come *zombie* finché il padre non ne preleva lo stato con `wait()`/`waitpid()`, poi il kernel libera le risorse

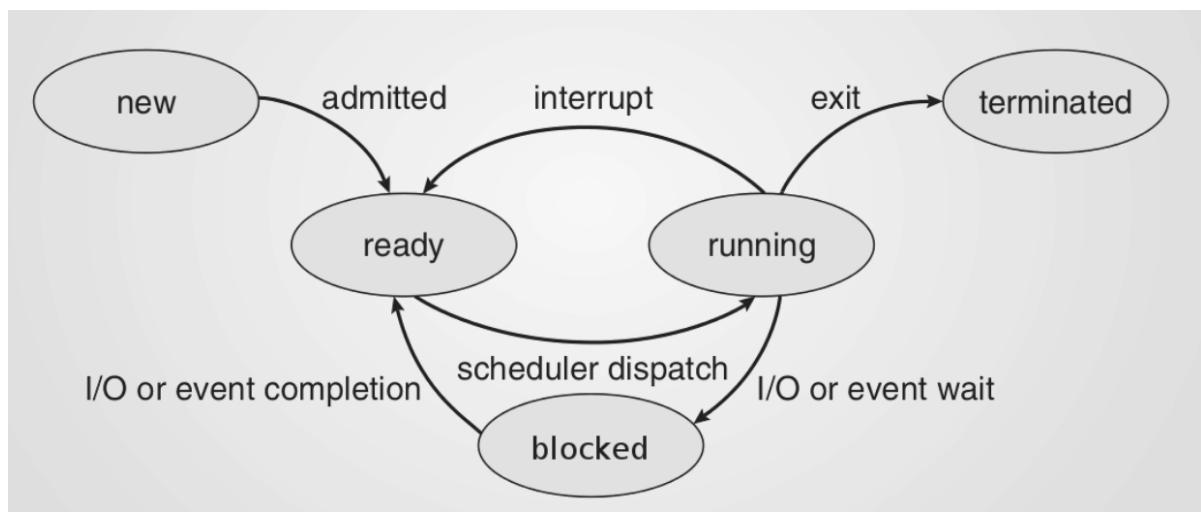


Figura 2.3: Possibili stati di un processo

### 2.1.7 Interrupt e prelazione

Un *interrupt* è un evento asincrono (da timer o periferiche) che interrompe temporaneamente il codice in esecuzione e trasferisce il controllo al kernel. Non “si individua il processo che l’ha causato”: l’hardware

e il driver identificano la *sorgente* dell'evento leggendo il vettore e i registri di stato del dispositivo. Se l'interrupt segnala il completamento di un'I/O, il processo che era **blocked** per quell'operazione passa a **ready**. Se l'interrupt proviene dal timer, abilita la *prelazione (preemption)*: il processo **running** può essere sospeso e rimesso in **ready** per far posto a un altro, ma non viene "terminato".

In un sistema *preemptive* ogni processo usa la CPU per un *time slice* limitato. Alla scadenza del quanto scatta un interrupt del timer. Il kernel salva lo stato del processo corrente, lo reinserisce tra i pronti e lascia allo scheduler la scelta del prossimo da eseguire. Questo meccanismo permette di rispettare priorità e reattività senza uccidere i processi.

#### 2.1.7.1 Sequenza tipica di gestione

Quando arriva un interrupt, la CPU esegue una sequenza standard. L'hardware salva sullo stack privilegiato il **PC** e il **PSW** e commuta in *kernel mode*. Carica dall'*interrupt vector* l'indirizzo della routine di servizio. Il kernel completa il salvataggio del contesto generale del processo nel suo **PCB** e, se necessario, attiva uno stack del kernel. La *interrupt service routine* gestisce l'evento: riconosce il dispositivo, ne azzerà la condizione di interrupt, aggiorna le strutture del kernel e rende **ready** i processi che attendevano quell'evento. Al termine, il kernel invoca lo scheduler per decidere quale processo far proseguire. Se viene scelto un processo diverso, il dispatcher ripristina dal relativo PCB registri e mappa di memoria e completa il *context switch*. Infine la CPU esegue il *return from interrupt* e riprende l'esecuzione del processo selezionato.

#### 2.1.8 Accodamento dei processi

Non esiste soltanto la coda dei processi pronti, ma esiste anche un'altra coda: quella dei **processi che sono in attesa di un'operazione di I/O**. Un processo viene messo in questa coda in seguito a una richiesta di I/O da parte sua e dopo il completamento dell'operazione di I/O, viene reinserito nella coda dei processi pronti. Esiste una coda di questo genere per ogni controller di dispositivo presente.

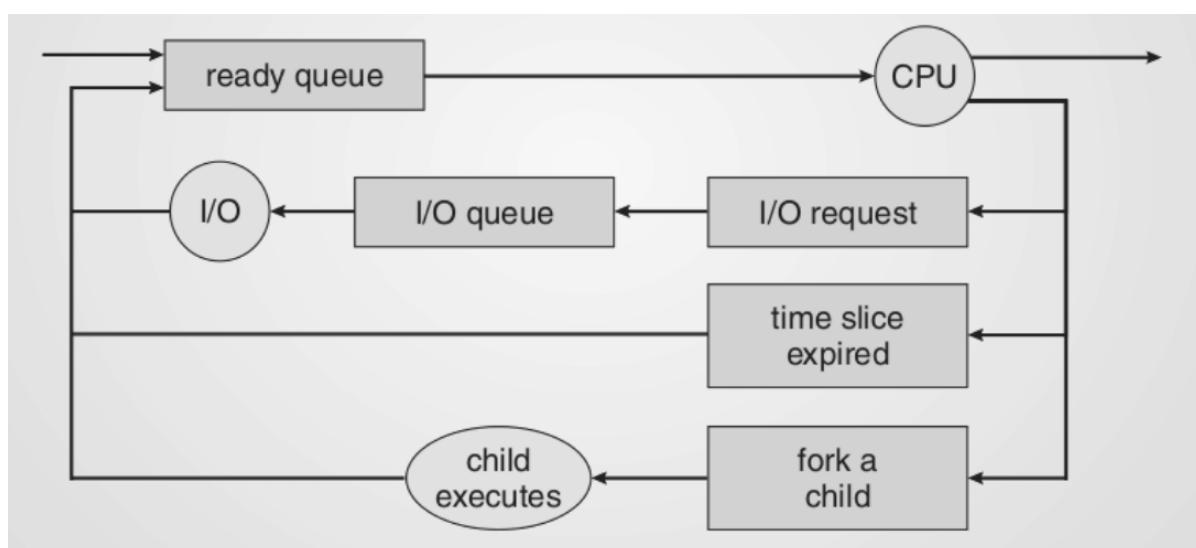


Figura 2.4: Diagramma di accodamento. I/O: fase di attesa del controller I/O, CPU: fase di esecuzione. Child Executes: attesa da esecuzione del figlio

### 2.1.9 Comunicazione tra processi

Nelle *pipeline* i processi comunicano collegando l'uscita di uno all'ingresso del successivo. Il coordinamento è implicito: il consumatore attende quando non ci sono dati e il produttore attende quando il *buffer* intermedio (a capacità finita) è pieno. Il buffer fa da cuscinetto e riduce i vincoli di sincronia, ma se i ritmi sono sbilanciati si accumula *backpressure* e l'intera catena rallenta. Il modello resta lineare e monodirezionale, quindi poco adatto a schemi di comunicazione complessi.

La *memoria condivisa* offre uno spazio comune mappato in più processi: massimizza le prestazioni evitando copie intermedie e consente topologie generali di scambio. Richiede però un protocollo d'accesso e meccanismi di sincronizzazione (mutua esclusione e attese su eventi) per evitare corse critiche e incoerenze sui dati condivisi.

## 2.2 Thread

Un processo può essere considerato come una collezione di risorse (memoria, registri, file aperti, ecc) e si può distinguere anche un flusso di esecuzione: il codice eseguito in un determinato momento dal processo. Questo flusso viene chiamato **thread**. Dentro un processo possono esserci più flussi di esecuzione, parliamo infatti di programmazione **multithread** (non multiprogrammazione) permettendo così a un processo di eseguire più compiti senza bloccarsi. I thread di un processo sono infatti indipendenti tra di loro, hanno una CPU tutta per loro (virtuale o non) ma condividono le risorse su cui operano, ovvero quelle del processo. Un thread principale si occupa di assegnare i compiti agli altri thread. Il fatto che i thread collaborino è un grande vantaggio, ma allo stesso tempo questo coordinamento porta a un rallentamento.

### 2.2.1 Thread e risorse condivise

Un **thread** è caratterizzato da *PC*, registri, *stack* e stato di esecuzione. Nei sistemi operativi moderni lo *scheduler* opera direttamente su entità schedulabili di tipo thread (spesso dette LWP). Questo non significa che esista “solo” una tabella dei thread: il kernel mantiene sia strutture *per-thread* (TCB) sia strutture *per-processo* (ad es. spazio degli indirizzi, tabella dei file, credenziali).

All’interno dello stesso processo i thread **condividono** codice, dati, spazio di indirizzi e descrittori di file aperti, quindi se un thread apre un file gli altri possono usarne lo stesso descrittore. Ciò comporta vantaggi ma anche possibili corse critiche su offset e buffer condivisi, da gestire con opportune primitive di sincronizzazione.

Il **cambio di contesto** tra thread dello stesso processo è più leggero di quello tra processi diversi, perché non richiede di cambiare spazio di indirizzi né di riprogrammare la MMU. Restano però il salvataggio/ripristino dei registri e lo *switch* dello stack del thread. Gli scheduler tendono a considerare affinità di CPU e località di cache, per cui talvolta conviene far proseguire un thread “fratello” dello stesso processo, ma non è una regola assoluta: la scelta dipende dalla politica di scheduling e dal carico del sistema.

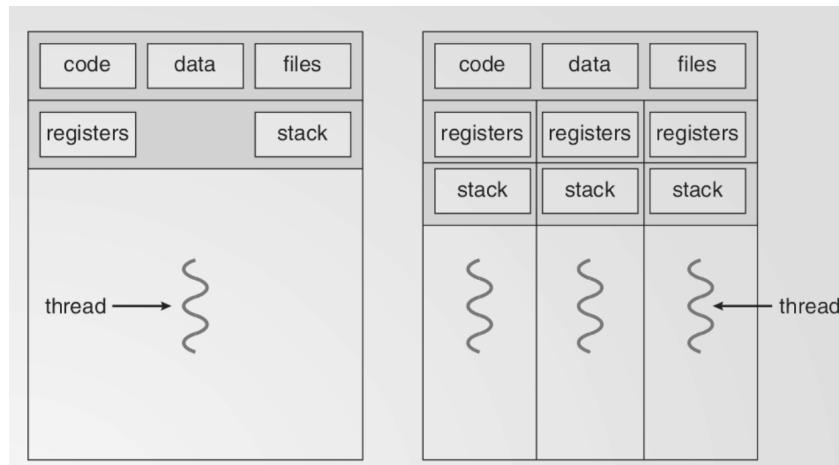


Figura 2.5: Un rappresentazione grafica delle risorse di un thread

### 2.2.2 Operazioni sui thread

Esistono diverse operazioni che un thread può adoperare su altri thread (non coinvolgono il kernel):

- **Creazione:** un thread ne crea un altro
- **Uscita:** il thread chiamante termina (un processo però, termina quando tutti i suoi thread terminano).
- **Join:** un thread si sincronizza con la fine di un altro thread (un'operazione simile alla chiamata wait dei processi).
- **Yield:** il thread chiamante rilascia volontariamente la CPU. Questo favorisce la collaborazione tra thread.

I thread, similmente ai processi, possono assumere degli stati (figura 2.6).

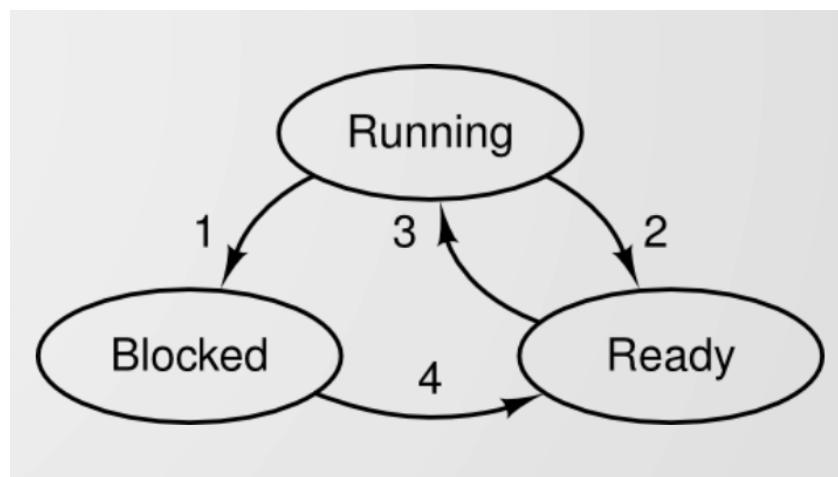


Figura 2.6: Possibili stati di un thread

### 2.2.3 Programmazione multithread e multicore

Avere più thread può migliorare **scalabilità** e **prestazioni**. Su una *CPU single-core* c'è *pseudoparallelismo*: i thread avanzano a turno grazie al time-slicing del sistema operativo (se presente SMT/Hyper-Threading, più CPU virtuali condividono lo stesso core ma non è vero parallelo). Su sistemi *multicore* i thread possono invece eseguire realmente in parallelo distribuendosi sui vari core.

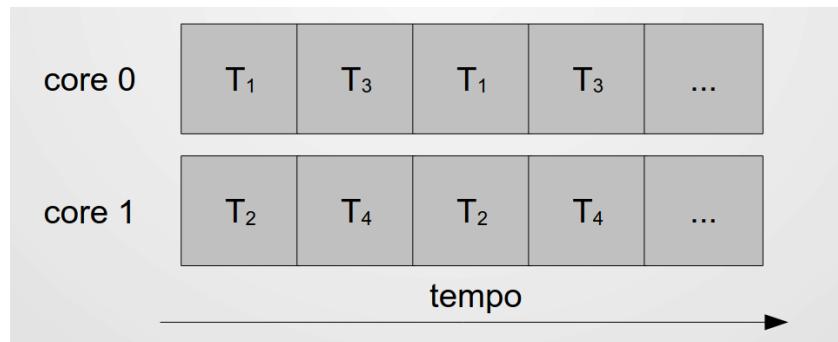


Figura 2.7: Distribuzione del carico di lavoro su 2 core

### 2.2.3.1 Aspetti progettuali del parallelismo

- **Separazione dei task:** individuare compiti coesi e indipendenti da assegnare ai thread, evitando spezzettamenti arbitrari.
- **Bilanciamento:** compiti né troppo grossi né troppo minimi; considerare durata, carico computazionale e chiamate bloccanti (I/O, sincronizzazione).
- **Dati e dipendenze:** decidere quali strutture dati ogni thread usa o condivide; coordinare gli accessi con mutex/condition variable/semafori. Gestire i casi produttore-consumatore (coda vuota/piena) con attese e risvegli corretti.
- **Ordine di esecuzione:** i flussi sono concorrenti, quindi l'ordine non è garantito; imporre il necessario *happens-before* tramite le primitive di sincronizzazione.
- **Debug & test:** più complessi per via di race condition, deadlock/livelock, starvation e *false sharing*.

### 2.2.4 Tipologie di thread

Esistono diverse tipologie di thread, in base all'implementazione che desideriamo.

#### 2.2.4.1 Thread a livello utente (modello 1 a molti)

Quando il sistema operativo non fornisce thread a livello kernel, l'intero *pacchetto* di thread vive nello spazio utente. Una libreria, detta *runtime system*, offre le primitive per creare, sincronizzare e schedulare i thread, svolgendo in user space il lavoro che altrimenti farebbe il kernel. In questo modello il kernel vede un solo entità schedulabile per processo: i thread dello stesso processo non possono quindi eseguire simultaneamente su più core. Lo *scheduler* della libreria decide quale thread far correre e fornisce primitive come `yield()` per cedere volontariamente la CPU.

Il *context switch* tra thread è interamente in user space: non richiede trap al kernel e quindi ha overhead ridotto. La libreria salva lo stato del thread nel suo TCB (registri utente e puntatore di stack); il PSW e gli altri bit privilegiati restano di competenza del kernel.

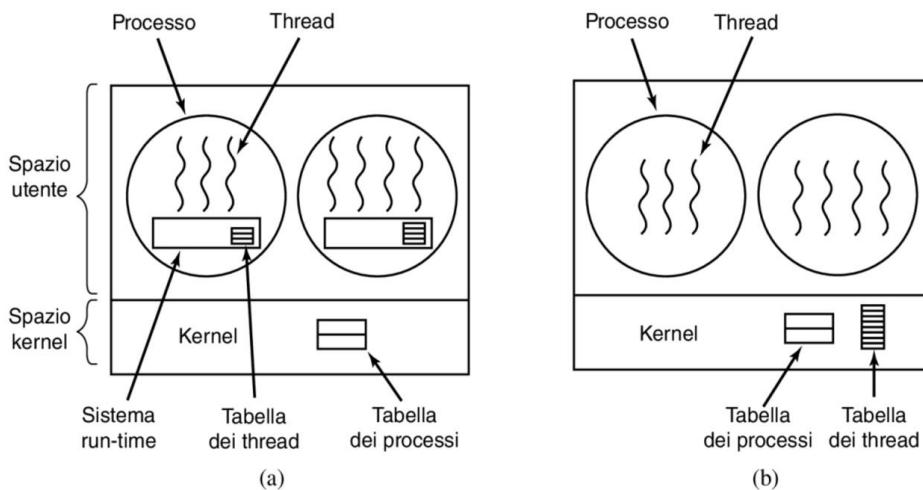


Figura 2.8: (a) un pacchetto di thread a livello utente, (b) un pacchetto di thread gestito dal kernel

**Chiamate bloccanti e page fault** C'è però un limite importante: se un thread effettua una *chiamata bloccante* del kernel (per esempio una `read()` che deve attendere I/O) l'intero processo risulta bloccato,

perché per il kernel esiste un solo processo schedulabile. Per attenuare il problema si usano primitive come `select()`, `poll()` o `epoll`: il thread verifica se un descrittore è pronto e, se non lo è, esegue `yield()` lasciando correre un altro thread dello stesso processo. In alternativa si invoca `select()` in modo bloccante così che il thread “dorma” fino a quando l’operazione diventa pronta, evitando giri a vuoto.

Un altro caso non aggirabile in user space è il *page fault*: quando una pagina non è in memoria, l’hardware genera un’eccezione e il kernel sospende il processo finché la pagina viene caricata dal disco. Poiché il kernel vede un solo schedulabile, l’intero processo si ferma, indipendentemente da quale thread abbia causato il fault.

**Scheduling interno personalizzabile** Lo scheduling dei thread all’interno del processo è curato dal *runtime system* e può essere adattato alle esigenze dell’applicazione (priorità interne, politiche cooperative, code di pronti nel TCB). Questo approccio rende la commutazione molto rapida ma non offre parallelismo reale tra i thread del medesimo processo in assenza di supporto a thread kernel.

#### 2.2.4.2 Thread a livello kernel (modello 1 a 1)

Con i thread implementati dal *kernel*, ogni thread è un’entità schedulabile a sé stante con un TCB nel kernel; lo scheduler del sistema opera *direttamente* sui thread. Il Context Switch avviene entrando in modalità kernel (tramite trap/interrupt), quindi è più costoso che nei thread a livello utente, ma abilita due proprietà cruciali: **parallelo reale** su più core e **chiamate bloccanti isolate** (blocca solo il thread interessato, non tutto il processo). La creazione e distruzione passano per syscall e comportano overhead; per ridurlo, molte applicazioni usano un *thread pool*: un insieme di thread *worker* creati una volta, che restano in attesa su una coda/condizione e vengono risvegliati quando c’è lavoro; a fine lavoro tornano in attesa senza essere distrutti.

**Pro:** parallelismo su multicore, blocchi per I/O non fermano l’intero processo, gestione uniforme di priorità e segnali. **Contro:** creazione/distruzione e context switch più costosi rispetto ai thread in user space; limite pratico al numero di thread del kernel.

#### 2.2.4.3 Modello ibrido (molti a molti)

Nel modello *M:N* una libreria di *user-level threads* (ULT) effettua lo scheduling in user space e li mappa su un insieme più piccolo di entità del kernel (LWP/thread kernel). In questo modo si combinano i vantaggi: **switch rapidi e politiche personalizzate** in user space, evitando però che una chiamata bloccante o un page fault fermino tutto il processo (si blocca solo l’LWP coinvolto, gli altri ULT continuano sugli altri LWP). L’applicazione o la libreria possono anche *vincolare* alcuni ULT a un LWP dedicato (es. GUI reattiva), mentre altri ULT condividono un pool di LWP per I/O o calcolo.

**Pro:** parallelismo su multicore, nessun blocco globale per I/O/page fault, flessibilità nello scheduling. **Contro:** maggiore complessità (due livelli di scheduler, gestione del mapping), debugging più impegnativo.

Se dovessimo stilare un confronto tra la velocità del context switch in base alla tipologia di implementazione avremo:

- **Thread fratelli** con thread implementati a livello utente, poiché non c’è trap.
- **Thread fratelli** con thread implementati a livello kernel, poiché c’è necessità di effettuare la trap.

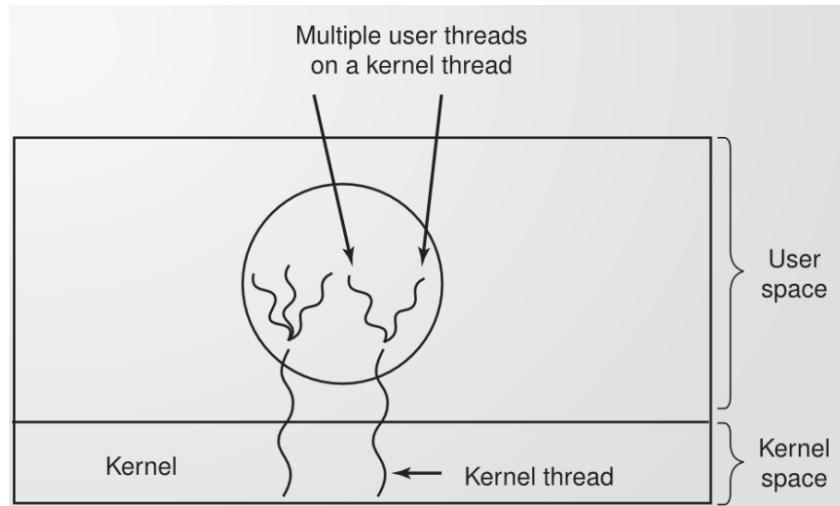


Figura 2.9: Un pacchetto di thread ibrido

- **Processo-processo:** in questo caso i thread non sono parenti, bisogna cambiare area di memoria e quindi riprogramma la MMU (Memory Management Unit).

### 2.2.5 Implementazioni di thread nei sistemi operativi moderni

Quasi tutti i sistemi operativi moderni implementano i thread a livello del kernel. Ad esempio Windows vede soltanto i thread, linux vede il tutto a livello di task (ovvero un ibrido tra processo e thread). Esiste una chiamata detta clone che è più modulabile di una fork e che può funzionare anche come una thread-create, anche in tipologie ibride.

La libreria standard di accesso ai kernel è la libreria **pthreads** conforme agli standard **POSIX** (Portable Operating System Interface) e i sistemi operativi che aderiscono sono UNIX, MacOS e Solaris (Windows non aderisce ma, dalle ultime versioni, contiene WSL che offre un sostituto). Altre librerie consentono di gestire i thread a livello utente e sono diverse per ogni sistema operativo e scendendo ulteriormente, si può vedere che i thread sono implementati a livello di singolo linguaggio.

## 2.3 Sincronizzazione nel parallelismo

All'interno di un programma che sfrutta il parallelismo con processi o thread, si deve pensare ad un modo di gestire le risorse condivise altrimenti si potrebbe incappare in una **race condition**. Esistono diversi metodi e diverse soluzioni al problema della sincronizzazione in base alla tipologia di problema dato.

### 2.3.1 Race conditions

Una *race condition* si verifica quando l'esito di un'elaborazione che coinvolge uno *stato condiviso* dipende da un uso concorrente non corretto. Se due attività aggiornano la stessa variabile (ad esempio il saldo di un conto) con un incremento **non atomico** (per atomico si intende un'operazione che non può essere interrotta dalla CPU), l'esecuzione sovrapposta delle fasi *leggi–calcola–scrivi* può produrre la *perdita di aggiornamenti* (valore finale errato). Il fenomeno si manifesta sia tra processi, sia tra thread, e può riguardare anche strutture del kernel (tabelle dei processi, code dei pronti, metadati di file) se non protette da adeguati meccanismi.

### 2.3.2 Mutua esclusione e sezioni critiche

La **mutua esclusione** garantisce che, quando un'attività entra nella porzione di codice che accede o modifica dati condivisi, chiamata **sezione critica**, le altre attività che vogliono accedervi vengano temporaneamente sospese o rimangano in attesa. Terminato il lavoro nella sezione critica, il controllo viene rilasciato e un'altra attività può proseguire. La delimitazione della sezione critica dovrebbe essere *minima* (solo le operazioni che devono essere atomiche) per ridurre contesa e ritardi.

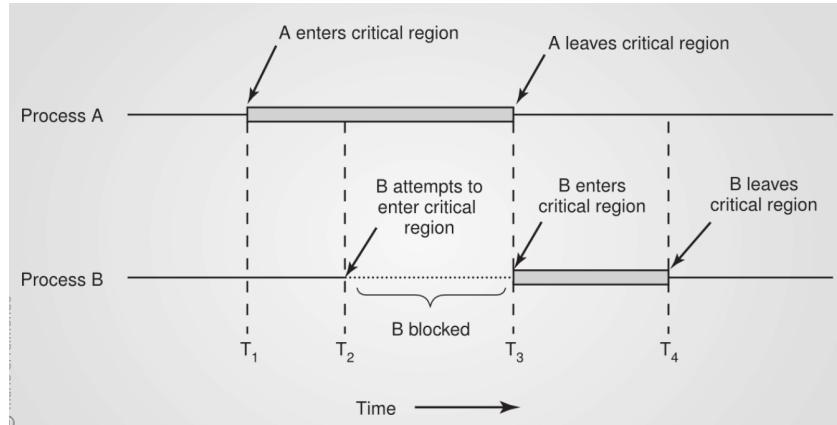


Figura 2.10: Corretto parallelismo tra due processi A e B

### 2.3.3 Requisiti per una soluzione corretta

Per considerare corretta una soluzione al problema delle sezioni critiche, si richiedono in genere:

- **Mutua esclusione** nell'accesso alle rispettive sezioni critiche.
- **Nessuna assunzione** sulla velocità di esecuzione o sul numero di CPU.
- **Nessun processo** fuori dalla propria sezione critica può bloccare un altro processo.
- **Nessun processo** dovrebbe restare all'infinito in attesa di entrare nella propria sezione critica.

## 2.4 Come realizzare la mutua esclusione

Si definiscono le operazioni `enter_region()` e `leave_region()` per delimitare la sezione critica. Queste operazioni si occuperanno di eseguire le operazioni necessarie a realizzare la mutua esclusione.

### 2.4.1 Disabilitare gli interrupt

Un approccio consiste nel **disabilitare gli interrupt** quando si entra nella sezione critica, in modo da prevenire prelazione e interruzioni. È praticabile solo in modalità kernel e scala male su sistemi multicore: disabilitare gli interrupt su un core non impedisce interferenze dagli altri.

### 2.4.2 Variabili di lock

Un altro approccio usa una **variabile di lock** condivisa: se vale 0 si entra e la si pone a 1; se vale 1 si attende (*spin lock*). Tuttavia senza operazioni atomiche il lock stesso diventa una nuova risorsa condivisa soggetta a race, quindi non risolve il problema in generale e crea una situazione di **busy waiting**: una infinita attesa che spreca risorse (e viola il requisito numero 4 di una buona soluzione al problema delle sezioni critiche).

### 2.4.3 Alternanza stretta

Anche in questo caso è presente una variabile condivisa, turn, però stavolta non è più uno switch 0/1 ma assume come valore l'ID del processo abilitato ad entrare nella sezione critica. Quando un processo chiama enter\_region le passa il suo ID e se il valore di turn è diverso dall'ID del processo, vuol dire che non è ancora il suo turno.

```

int N=2
int turn

function enter_region(int process)
    while (turn != process) do
        nothing

function leave_region(int process)
    turn = 1 - process

```

Figura 2.11: Pseudocodice dell'alternanza stretta

Questa soluzione risolve il problema della race condition per turn, perché il suo valore è univoco e può essere infatti **generalizzata** a  $N$  processi. Viola, però, la **condizione 3**: consideriamo due processi  $P_0$  e  $P_1$  e ipotizziamo che  $P_0$  entri nella sua regione critica, la completa ed esce. Adesso è il turno di  $P_1$ , che però sta facendo altro e se  $P_0$  dovesse avere nuovamente bisogno di entrare nella sezione critica dovrà aspettare che  $P_1$  arrivi alla sua sezione critica e la completi e quindi si blocca, l'algoritmo forza quindi una **staffetta**.

### 2.4.4 Soluzione di Peterson

La soluzione di Peterson usa due elementi condivisi: un array booleano interested[2] che indica l'intenzione di entrare e una variabile turn che risolve il conflitto. L'idea operativa è: *annuncia l'intenzione, imposta il turno al proprio ID e poi attendi finché l'altro è interessato e il turno coincide col tuo*. Quando si esce dalla sezione critica si pone il proprio interested a false. Così si garantiscono mutua esclusione, progresso e attesa limitata in presenza di un modello di memoria sequenziale. Se entrambi richiedono contemporaneamente l'ingresso, l'ultimo che ha scritto turn rimane in attesa; l'altro entra.

```

int N=2
int turn
int interested[N]

function enter_region(int process)
    other = 1 - process
    interested[process] = true
    turn = process
    while (interested[other] = true and turn = process) do
        nothing

function leave_region(int process)
    interested[process] = false

```

Figura 2.12: Pseudocodice della soluzione di peterson

#### 2.4.4.1 Esempio di concorrenza

Se P0 è nella sezione critica e P1 chiede di entrare, P1 annuncia l'intenzione e imposta turn. La condizione d'attesa (*l'altro è interessato e il turno è il mio*) restaL'idea combina un vera, quindi P1 attende. Quando P0 esce e ritira la propria intenzione, P1 può procedere.

#### 2.4.4.2 Esempio multicore e riordini

Consideriamo due thread T0 e T1 su variabili condivise x e y inizialmente a 0:

```
T0: x = 1;
R1 = y;
R2 = x;

T1: y = 1;
```

A seconda dell'interacciamento sono “naturali” tre esiti: (R1=0, R2=1), (R1=1, R2=0), (R1=1, R2=1). Su architetture a memoria *rilassata* può comparire anche (R1=0, R2=0) a causa di riordini e buffer di scrittura. Senza barriere di memoria/primitive atomiche lo stesso problema può invalidare Peterson su multiprocessore.

#### 2.4.4.3 Contesa simultanea (2 processi)

Se entrambi i processi vogliono entrare nello stesso istante, ciascuno pone interested[i]=true e scrive in turn il proprio ID. Poiché turn è condivisa, l'*ultimo* che la scrive ne lascia il valore a se stesso; di conseguenza la condizione

```
interested[other] && turn == i
```

è vera per lui e resta in attesa, mentre l'altro procede nella sezione critica. In pratica “vince” chi ha scritto turn per primo (l'ultimo arrivato cede il passo).

#### 2.4.4.4 Nota su N processi: schema a tornei

Per più di due processi si può comporre Peterson in un **torneo** a livelli (binary tree di lock a due vie). Ogni processo parte da una foglia e, per entrare nella sezione critica, affronta una sequenza di “duelli” Peterson salendo verso la radice: al *livello 1* compete con il vicino, il vincitore passa al *livello 2*, e così via fino alla radice; solo chi vince *tutti* i livelli entra nella sezione critica. All'uscita rilascia i lock lungo il percorso, permettendo agli sconfitti di proseguire. Questo schema (detto anche *tournament/filter lock*) conserva mutua esclusione, progresso e attesa limitata, con contesa locale per livello e un numero di passi proporzionale a  $\log_2 N$ . In alternativa, si può usare l'algoritmo a *panetteria* di Lamport, che generalizza a  $N$  senza la struttura ad albero.

#### 2.4.5 Istruzioni TSL (Test-and-Set) e XCHG

Per gestire una variabile di lock in modo sicuro si possono usare istruzioni hardware atomiche. La **TSL** (Test-and-Set Lock) esegue in *un solo passo indivisibile* due azioni: legge il valore corrente della lock e, subito dopo, imposta la lock ad un valore “occupato” (tipicamente 1). In pseudonota:

```
TSL R, [LOCK] ⇒ R ← LOCK; LOCK ← 1
```

È equivalente concettualmente a due MOV, ma racchiuse in un’*unica* operazione atomica; quindi nessuna prelazione o interleaving può inserire istruzioni tra le due azioni.

L'atomicità vale anche su sistemi multiprocessore: l'hardware serializza l'accesso alla linea di cache/memoria della lock (tramite protocolli di coerenza e meccanismi di "lock") così che nessun altro core possa osservare stati intermedi. Non è necessario "vedere" due istruzioni separate: per tutti gli altri core l'operazione appare come un'unica lettura + scrittura indivisibile.

Un'istruzione alternativa è **XCHG** (scambio atomico) che permuta i contenuti tra un registro e una parola di memoria. Pur con semantica diversa, TSL e XCHG sono equivalenti come primitive di base per costruire lock.

Queste istruzioni, però, forniscono solo *mutua esclusione* a basso livello: l'attesa resta tipicamente *attiva* (spin-waiting), con possibile consumo di CPU e rischio di scarsa equità. Per eliminare lo spreco di cicli si usano primitive bloccanti (ad es. sleep/wakeup) o lock più evoluti.

```
enter_region:
TSL REGISTER,LOCK
CMP REGISTER,#0
JNE enter_region
RET
```

```
leave_region:
MOVE LOCK,#0
RET
```

Figura 2.13: Operazioni enter region e leave region con TSL

#### 2.4.6 Sleep e *wakeup*

L'uso di lock con attesa attiva (*spin lock*) può portare a spreco di CPU e, in certe combinazioni di scheduling, a veri e propri stalli: immaginiamo tre processi con priorità alta (**PH**), media (**PM**) e bassa (**PL**) che condividono una struttura dati protetta da una lock. Se **PL** entra per primo nella sezione critica e **PH** si risveglia iniziando a controllare la lock con *busy waiting*, **PH** occupa la CPU impedendo a **PL** di proseguire ed uscire: la lock non verrà mai rilasciata e l'intero sistema resta in stallo. Questo mostra il limite dell'attesa attiva, a maggior ragione in presenza di politiche a priorità.

La soluzione è trasformare l'attesa *attiva* in attesa *passiva*: invece di ciclare, il processo che non può entrare nella sezione critica si **blocca** volontariamente. Il sistema operativo fornisce a tal fine primitive come **sleep** e **wakeup**. Con **sleep** il processo viene rimosso dalla coda dei processi pronti e non consuma tempo di CPU finché l'evento d'interesse non si verifica; con **wakeup** un altro componente (o il dispositivo/driver) lo risveglia quando la condizione è soddisfatta (per esempio quando la lock viene rilasciata o quando nuovi dati arrivano in un buffer).

Questo meccanismo evita sprechi di CPU e previene gli stalli dovuti al *busy waiting*; inoltre consente allo scheduler di rispettare le priorità senza bloccare l'avanzamento del processo che deve rilasciare la risorsa. In pratica, **sleep/wakeup** forniscono il mattone di base per realizzare forme più ricche di sincronizzazione bloccante, eliminando la necessità di cicli attivi.

#### 2.4.7 Semafori

Un semaforo è una generalizzazione del concetto di sleep e wakeup: è uno strumento software offerto dal sistema operativo che usa forme di bloccaggio passivo e quindi non spreca CPU. La variabile intera condivisa *S* indica lo stato del semaforo e fa da "contatore di sveglie" (questo valore non sarà mai negativo). Per incrementare o decrementare il contatore si usano le operazioni **down** (o **wait**) e **up** (o **signal**) e si può pensare alla prima come una sleep e alla seconda come a una wakeup. Entrambe tengono conto del vincolo di non negatività e introducono una pausa passiva. Si possono verificare race condition sul valore del semaforo, dato che la variabile è condivisa e allora bisogna garantire che

la up e la down siano atomiche, quindi se qualcuno sta facendo una up su un semaforo i tentativi di altri processi di fare un'altra up o una down devono essere interrotti. Per applicare la mutua esclusione, dato che siamo nel kernel, si potrebbero disabilitare gli interrupt ma come già visto ha problemi nei sistemi multicore, allora si adatta la soluzione con le istruzioni TSL/XCHG rendendole atomiche. Ad ogni semaforo si associa una variabile di lock, con delle `enter_region` e `leave_region`. Questa soluzione fa **busy waiting** ma può essere tollerata per due motivi:

1. Wait e signal hanno codice piccolo, lo spin lock non durerà molto.
2. Non si presenta il problema dell'inversione di priorità perché in questo contesto non esiste il concetto di priorità.

#### 2.4.7.1 Nota per i sistemi operativi moderni

Nei sistemi operativi moderni in realtà viene implementata una soluzione senza spin lock, poiché è a livello kernel è possibile sfruttare le strutture dati kernel. Cioè ad ogni semaforo sarà associata una coda dei processi bloccati da esso e quando ci sono sveglie disponibili viene estratto da questa coda e risvegliato.

#### 2.4.8 Tipologie dei semafori

Esistono due modi per utilizzare i semafori:

**Semaforo mutex** : associamo un semaforo  $M$  a una struttura dati, come se fosse una variabile di lock, inizializzandolo a 1. Dobbiamo demarcare le sezioni critiche con una `enter_region`, in questo caso la wait del semaforo  $M$  e una `leave_region` che corrisponde alla signal di  $M$ . Quando un processo entra nella regione critica  $M$  transita a 0. Se un altro processo tenta di entrare nella sezione critica viene bloccato. Appena il processo  $P_1$  termina si sblocca la wait di  $P_2$ . Se implementiamo la mutua esclusione con semafori non c'è busy waiting, quindi il blocco è passivo, a differenza dell'utilizzo della variabile lock.

**Conteggio risorse (sincronizzazione)** : questa parte viene spiegata nella **sezione 2.5.1 soluzione con i semafori**.

#### 2.4.9 Mutex

Ipotizzando che i thread utente abbiano bisogno di garantire mutua esclusione, i semafori non funzionano: usando una wait si blocca tutto il processo, dato che il sistema operativo non vede i thread ma l'intero processo. Potremmo pensare di usare una variabile di lock (con TSL) ma anche questo crea problemi: se il thread 1 monopolizza la CPU non può passarla ai suoi fratelli e nessuno cambierebbe il valore di lock. Si può utilizzare una tipologia di mutua esclusione chiamata **Mutex**: viene inizializzato a 0 (che significa che nessuno è entrato in una sezione critica), dopo la TSL memorizza il valore corrente di mutex e poi lo pone a 1 (ovvero che la parte di dati condivisa può essere utilizzata).

#### 2.4.9.1 Futex

I **mutex** puramente in spazio utente funzionano con un *test-and-set* e, sotto contesa, restano in *busy waiting* (spreco di CPU); quelli puramente kernel, invece, richiedono una system call anche quando il

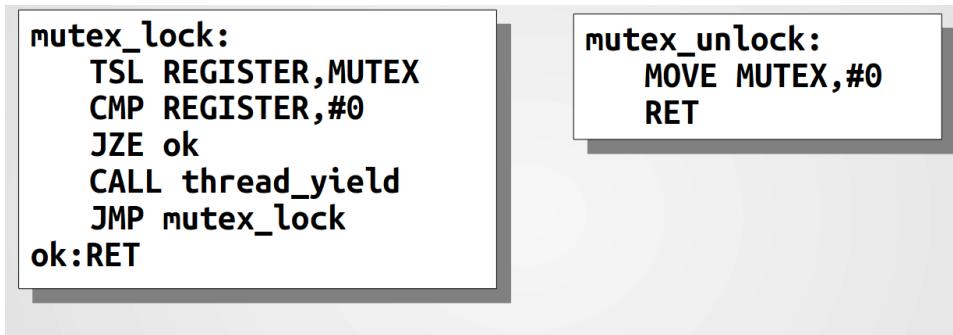


Figura 2.14: Caption

lock è libero (overhead inutile). **Linux** introduce i **futex** (*fast user-space mutex*) per avere il *fast path* in user space e passare nel kernel solo quando serve.

**Struttura e principio di funzionamento** Un futex ha due componenti coordinate:

1. **User space:** una parola (*word*) condivisa che rappresenta lo stato del lock. Il thread tenta l'acquisizione con un'operazione atomica (es. CAS/TSL). Se riesce, entra subito nella sezione critica (*nessuna syscall*).
2. **Kernel:** se l'acquisizione fallisce (lock occupato), il thread invoca la syscall di attesa (`futex wait`) che lo *parcheggia* passivamente; al rilascio, chi sblocca aggiorna la word e chiama `futex wake` per risvegliare *un* attendente. Così si evita lo *spin* prolungato e il “thundering herd”.

**Vantaggi principali** I vantaggi principali di avere un futex al posto di un mutex sono il mantenere niente chiamate di sistema se non sono necessarie e avere una attesa passiva al posto di uno spin lock.

**Nota sui thread a livello utente (N:1)** I futex rendono efficienti i lock tra **thread del kernel** (mappatura 1:1). Con **thread a livello utente** puri (N:1), una syscall bloccante addormenta l'intero processo: in quel modello servono primitive del *runtime* (lock/semafori in user space) e I/O non bloccante. I futex non “risolvono” questo limite del modello N:1; risolvono l'*overhead* e la gestione della contesa per i lock usati da thread schedulati dal kernel.

#### 2.4.10 Monitor

Il monitor è un costrutto ad alto livello, che svolge gli stessi compiti di un semaforo e che nasconde molti dettagli implementativi (è utilizzabile solo dai thread). In sostanza è un tipo di dato astratto implementato da alcuni dei linguaggi più moderni ed è definito a livello utente. Il monitor viene implementato dal compilatore sfruttando strumenti kernel già esistenti (semafori e mutex) ed è quindi caratterizzato da strutture dati e metodi, garantisce mutua esclusione nel codice dei metodi che contiene. I metodi in questione servono ad accedere alle strutture dati citate precedentemente e manipolarle, in modo da evitare race condition.

Per risolvere il problema della sincronizzazione dei thread, vengono definiti all'interno di quest'ultimi delle **variabili di condizione** (etichette) su cui veranno chiamate operazioni wait e signal. Se un thread deve bloccarsi, chiama la wait e aggiorna la variabile di condizione, bloccandosi. Ad ogni variabile di condizione è associata una coda, in cui si trovano i thread bloccati da un determinato evento: quando un altro thread chiama la signal, un thread viene prelevato da questa coda e risvegliato (simile alla coda di un semaforo). A differenza dei semafori, le variabili di condizioni non hanno uno stato ben preciso

perché non serve contare le sveglie dato che un monitor può essere usato da un thread per volta. Se viene inviato un segnale a variabile di condizione con nessuno in attesa, il segnale è perso per sempre. In altre parole, la wait **dove** arrivare prima della signal.

#### 2.4.10.1 Semantica della `signal` nei monitor

I monitor così presentati però hanno un problema, come esempio prendiamo questo scenario: il thread **T1** è bloccato in `wait()` dentro il metodo **A** del monitor; il thread **T3** entra nel metodo **B**, modifica lo stato e invoca `signal(cond)`. L'effetto della `signal` dipende dalla semantica adottata:

**Monitor di Hoare (signal-and-wait).** **T3** chiama `signal` e *cede immediatamente* il monitor a **T1**.

**T3** resta sospeso *dentro* il monitor; **T1** riprende *subito* l'esecuzione di **A** dopo la `wait`. Quando **T1** esce (o torna in `wait`), la proprietà del monitor ritorna a **T3**. La mutua esclusione è sempre rispettata.

**Monitor di Mesa (Java/POSIX, signal-and-continue).** **T3** chiama `signal` ma *continua* ad eseguire **B** mantenendo il lock fino all'uscita dal monitor. **T1** diventa “pronto”, ma potrà ripartire solo *dopo* che qualcuno rilascia il lock e *riacquisendolo*. Poiché lo stato può cambiare ancora, **T1** deve ricontrillare la condizione in un `while` (semantica di Mesa).

**Signal-and-return (Concurrent Pascal).** `signal` è l'ultima istruzione del metodo **B**: **T3** *lascia subito* il monitor. Il prossimo a entrare è **T1**, che riprende **A** dopo la `wait`; quando **T1** termina, **T3** è già uscito.

**Osservazione** In nessuna delle tre semantiche due thread eseguono *contemporaneamente* dentro il monitor: cambia *quando* il risvegliato riparte e *quali garanzie* ha sulla condizione (Hoare: valida al risveglio; Mesa: da verificare con `while`).

Aspetto	Semaforo	Mutex	Monitor
Scopo	Contatore intero con <i>down/up</i> ; sincronizza e conta risorse (binario o di conteggio).	Lock esclusivo per sezioni critiche; posseduto da un thread.	Costrutto ad alto livello: lock implicito + variabili di condizione (metodi <i>wait/signal</i> ).
Possesso / Stato	Nessun “owner”; <i>up/down</i> anche da attori diversi.	Ha un owner; in genere <i>unlock</i> dallo stesso thread.	Lock gestito dal runtime/language; <i>wait</i> rilascia il lock e sospende.
Attesa	In OS moderni blocco passivo (coda); possibili versioni con spin solo per brevi tratti.	Fast path user-space, blocco passivo su coda (es. futex).	Attesa su condition; con semantica Mesa si usa while per ricontrillare la condizione.
Uso tipico	Conteggio risorse (producer-consumer, pool), barriere.	Proteggere strutture dati condivise, sezioni critiche brevi.	Incapsulare stato + politiche di sync in metodi sicuri/leggibili.
Punti di forza	Versatile, inter-processo.	Semplice, ownership chiaro.	Più alto livello, meno errori di protocollo.
Limiti	Facile uso scorretto (perdita/doppio <i>up</i> ); no ownership; rischio starvation se non gestito.	Solo 0/1 (non conta risorse); deadlock se ordine errato.	Dipende dal linguaggio/RT; non nativo in C (si emula con mutex+condvar); semantica <i>signal</i> da conoscerre.

Tabella 2.1: Confronto sintetico tra semafori, mutex e monitor.

## 2.5 Problemi classici di sincronizzazione

### 2.5.1 Produttore - Consumatore

Il produttore e il consumatore sono due processi che condividono un buffer, di dimensione fissa. Il produttore inserisce i dati nel buffer e il consumatore le preleva. Tuttavia, sia il produttore sia il consumatore potrebbero bloccarsi: se il produttore vuole inserire un nuovo elemento nel buffer ma lo trova pieno, entra in stato di sleep (e verrà risvegliato quando il consumatore rimuoverà uno o più elementi), lo stesso succede al consumatore quando vuole rimuovere un elemento dal buffer ma lo trova vuoto (verrà risvegliato quando il produttore avrà inserito uno o più elementi). Il codice che si vede in figura 2.14 potrebbe causare race condition.

```

1 function producer()
2   while (true) do
3     item = produce_item()
4     if (count = N) sleep()
5     insert_item(item)
6     count = count + 1
7     if (count = 1)
8       wakeup(consumer)

```

```

1 function consumer()
2   while (true) do
3     if (count = 0) sleep()
4     item = remove_item()
5     count = count - 1
6     if (count = N - 1)
7       wakeup(producer)
8     consume_item(item)

```

Figura 2.15: Produttore-consumatore (pseudocodice).

#### 2.5.1.1 Soluzioni con semafori

I semafori, in questo caso, contano le risorse presenti del problema. Quindi esiste un semaforo, dati  $N$  slot di una coda liberi, che conta quanti sono vuoti e uno che conta quanti sono pieni. Per sincronizzare

il rilascio e l'aggiunta di elementi inoltre, è presente un semaforo che garantisce la mutua esclusione (figura 2.15). I semafori in questo contesto, aspettano che il buffer sia vuoto o pieno (rispettivamente produttore e consumatore) e dopo producono/consumano per poi inviare il segnale al giusto semaforo.

```
int N = 100
semaphore mutex = 1
semaphore empty = N
semaphore full = 0
```

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

(a) Producer

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

(b) Consumer

Figura 2.16: Produttore-consumatore con semafori: inizializzazione e pseudocodice.

### 2.5.1.2 Soluzione con monitor

Un *monitor* incapsula il buffer, una variabile intera count (numero di elementi presenti) e due *condition variables* full ed empty. Il produttore invoca il metodo del monitor *insert*: se il buffer è pieno (*count* = *N*) attende su full; quando inserisce un elemento incrementa count e, se il buffer passava da vuoto a non vuoto (*count* = 1), segnala empty per risvegliare un consumatore. Il consumatore chiama *remove*: se il buffer è vuoto (*count* = 0) attende su empty; quando rimuove un elemento decrementa count e, se il buffer passava da pieno a non pieno (*count* = *N* − 1), segnala full per risvegliare un produttore. All'esterno, i thread chiamano semplicemente i metodi del monitor: il produttore produce un item e lo inserisce; il consumatore preleva e poi consuma. Con semantica “Mesa” (come in POSIX/Java) le attese si implementano con while attorno alle wait; qui mostriamo lo pseudocodice essenziale.

### 2.5.2 Problema dei 5 filosofi

Il problema modella l'accesso esclusivo (temporaneo) a risorse limitate da parte di più processi concorrenti. Cinque filosofi siedono attorno a un tavolo circolare; tra ogni coppia di filosofi c'è una forchetta (in totale 5). Ogni filosofo alterna due attività: *think()* e *eat()*. Per mangiare ha bisogno di *entrambe* le forchette adiacenti (sinistra e destra); due filosofi adiacenti non possono dunque mangiare nello stesso istante.

#### 2.5.2.1 Obiettivo di sincronizzazione

Progettare un protocollo che garantisca sicurezza, assenza di deadlock e di livelock (il sistema non si blocca e progredisce), assenza di starvation (ovvero ogni filosofo prima o poi mangia) e concorrenza massima.

```

monitor pc_monitor
    condition full, empty
    integer count = 0

    function insert(item)
        if count = N then wait(full)
        insert_item(item)
        count = count + 1
        if count = 1 then signal(empty)

    function remove()
        if count = 0 then wait(empty)
        item = remove_item()
        count = count - 1
        if count = N - 1 then signal(full)

```

(a) Monitor: stato e metodi.

```

function producer()
    while (true) do
        item = produce_item()
        pc_monitor.insert(item)

```

(b) Produttore

```

function consumer()
    while (true) do
        item = pc_monitor.remove()
        consume_item(item)

```

(c) Consumatore

Figura 2.17: Produttore–consumatore realizzato con un monitor.

### 2.5.2.2 Quattro approcci e relativi problemi

1. **Variabili di lock per forchetta.** Ogni forchetta ha un lock; `take_fork` blocca se la forchetta è occupata, `put_fork` la rilascia. **Problema: deadlock.** Se tutti prendono simultaneamente la forchetta di destra, ognuno resta in attesa della sinistra: si verifica attesa circolare.

```

int N = 5
function philosopher(int i)
    think()
    take_fork(i)           // forchetta sinistra
    take_fork((i+1) mod N) // forchetta destra
    eat()
    put_fork(i)
    put_fork((i+1) mod N)

```

2. **Controlla–rilascia–riprova (ritento dopo tempo fisso).** Un filosofo che riesce a prendere solo una forchetta la *rilascia* subito e, dopo un intervallo *fisso*, riprova a prenderle entrambe. **Problema: livelock.** Se tutti seguono lo stesso ritmo, tutti prendono, rilasciano e riprovano all'unisono senza avanzare.
3. **Controlla–rilascia–riprova con attesa casuale.** Come sopra, ma l'attesa prima di riprovare è *random*. Questo rompe la sincronizzazione perfetta e nella pratica elimina il livelock. **Limiti:** efficienza non ottimale, progresso solo probabilistico, possibili squilibri/fairness non garantita.
4. **Semaforo mutex globale.** Si protegge con un unico mutex l'operazione “prendo entrambe le forchette”: un solo filosofo alla volta entra nella sezione critica e può prelevare le due adiacenti; poi

mangia e rilascia. **Pro:** semplice, niente deadlock. **Contro:** serializza l'accesso (spesso mangia un solo filosofo per volta), quindi riduce la concorrenza possibile.

### 2.5.2.3 Soluzione con semafori

Ogni filosofo può trovarsi in tre stati: THINKING (pensa), HUNGRY (vuole mangiare ma non ha le risorse), EATING (sta mangiando). Manteniamo:

- un vettore condiviso `state[N]` con lo stato di ciascun filosofo;
- un **mutex** (semaforo binario) per proteggere l'accesso atomico a `state`;
- un array di semafori `s[N]` (tutti inizializzati a 0): `s[i]` è la “campanella” con cui il filosofo  $i$  si auto-sospende e viene risvegliato quando può davvero mangiare.

L'idea è: `take_forks(i)` prova ad acquisire *entrambe* le forchette. Se non è possibile, il filosofo  $i$  si addormenta su `s[i]` e si risveglierà solo quando qualcuno (tipicamente un vicino che ha finito) verificherà che può passare a EATING. La verifica è incapsulata in `test(i)`. Il rilascio avviene con `put_forks(i)`.

```

const int N = 5
const int THINKING = 0, HUNGRY = 1, EATING = 2

int state[N]           // stato di ciascun filosofo
semaphore mutex = 1    // protegge 'state'
semaphore s[N] = {0, ..., 0} // semafori privati

function take_forks(int i)
    down(mutex)
    state[i] = HUNGRY
    test(i)
    up(mutex)
    down(s[i])      // attende finché non
    ↳ può mangiare

function put_forks(int i)
    down(mutex)
    state[i] = THINKING
    test(left(i))   // prova a sbloccare il
    ↳ vicino sinistro
    test(right(i))  // ... e quello destro
    up(mutex)

function left(int i) = (i-1+N) mod N
function right(int i) = (i+1) mod N

function test(int i)
    if state[i] == HUNGRY
        and state[left(i)] != EATING
        and state[right(i)] != EATING then
            state[i] = EATING
            up(s[i])      // sveglia i, che ora
            ↳ può mangiare

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

(a) Acquisizione e rilascio.

(b) `test` e ciclo del filosofo.

Figura 2.18: Pseudocodice della soluzione con semafori.

## Funzionamento

- **Mutua esclusione:** tutte le letture/scritture di `state` avvengono sotto `mutex`.
- **Niente deadlock:** nessuno rimane bloccato dentro la sezione critica; se un filosofo non può mangiare, rilascia `mutex` e si sospende su `s[i]`.
- **Concorrenza massima:** `test()` consente a più non-adiacenti di mangiare in parallelo.

- Perché **state[i]==HUNGRY**? Serve quando un vicino termina: put\_forks chiama test(left/right) e deve sbloccare solo chi sta effettivamente aspettando (HUNGRY), non chi sta pensando.

#### 2.5.2.4 Soluzioni con monitor

Anche qui usiamo un vettore state[N] con gli stati THINKING/HUNGRY/EATING. Al posto di un semaforo privato per filosofo, ogni filosofo ha una **variabile di condizione** self[i] con cui può auto-sospendersi se non può entrare in EATING. I metodi del monitor (take\_forks, put\_forks, test) sono eseguiti in mutua esclusione, quindi non serve un mutex esterno. La logica è la stessa della soluzione con semafori: test(i) promuove i a EATING quando i vicini non mangiano; se take\_forks non riesce, il filosofo fa wait(self[i]) e verrà risvegliato con signal(self[i]) da chi libera le forchette.

```

int N=5;  int THINKING=0;  int HUNGRY=1;  int EATING=2

monitor dp_monitor
    int state[N]
    condition self[N]

    function take_forks(int i)
        state[i] = HUNGRY
        test(i)
        if state[i] != EATING then
            wait(self[i])

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

    function test(int i)
        if state[left(i)] != EATING
            and state[right(i)] != EATING
            and state[i] == HUNGRY then
                state[i] = EATING
                signal(self[i])
end monitor

```

```

function left (int i) = (i-1+N)
    ← mod N
function right(int i) = (i+1)
    ← mod N

function philosopher(int i)
    while (true) do
        think()
        dp_monitor.take_forks(i)
        eat()
        dp_monitor.put_forks(i)
    
```

Figura 2.19: Pseudocodice della soluzione con monitor

#### 2.5.3 Problema dei lettori e scrittori

Modella l'accesso concorrente a un “database”: più *lettori* possono coesistere senza conflitti, mentre uno *scrittore* richiede esclusione totale. L'obiettivo è permettere letture parallele, impedire conflitti in scrittura e scegliere una politica di priorità (lettori, scrittori o bilanciata) evitando starvation (il thread non ottiene le risorse della CPU, in questo caso continuano ad entrare lettori ma lo scrittore non può).

#### Soluzione con semafori

Si usa un contatore condiviso rc (numero di lettori attivi) protetto da mutex e un semaforo db che chiude/apre l'accesso al database. Il primo lettore che entra fa down(db), l'ultimo che esce fa up(db). Gli scrittori usano direttamente down(db) / up(db). Vantaggio: letture massivamente concorrenti. Limite: lo scrittore può attendere a lungo (starvation) se continuano ad arrivare lettori.

```

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

function reader()
    while true do
        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 do
        think_up_data()
        down(db)
        write_database()
        up(db)

```

### Soluzione 1 con monitor (ancora lettori preferiti)

Il database resta *fuori* dal monitor; dentro ci sono solo le funzioni di arbitraggio. rc conta i lettori attivi, busy\_on\_write segnala scrittura in corso; read, write sono variabili di condizione. I lettori si bloccano solo se c'è *una scrittura in corso*; se uno scrittore è in attesa ma non sta scrivendo, nuovi lettori possono entrare: di nuovo, bias verso i lettori.

```

monitor rw_monitor
    int rc = 0; boolean busy_on_write = false
    condition read, write

    function start_read()
        if (busy_on_write) wait(read)
        rc = rc + 1
        signal(read)

    function end_read()
        rc = rc - 1
        if (rc == 0) signal(write)

    function start_write()
        if (rc > 0 OR busy_on_write) wait(write)
        busy_on_write = true

    function end_write()
        busy_on_write = false
        if (in_queue(read)) signal(read)
        else                  signal(write)

```

### Soluzione 2 con monitor (no starvation per gli scrittori)

Si aggiunge in `start_read()` il controllo sulla coda degli scrittori: se c'è qualcuno in attesa, nuovi lettori si fermano. Così, quando i lettori attuali finiscono, passa almeno uno scrittore; poi un lettore risveglia "a cascata" gli altri lettori. È la politica più equilibrata tra throughput e fairness.

```
function start_read()
    if (busy_on_write OR in_queue(write)) wait(read)
    rc = rc + 1
    signal(read)
```

### Soluzione 3 con monitor (scrittori preferiti)

Come la 2, ma alla fine della scrittura si privilegiano altri scrittori se presenti. Massimizza il throughput in scenari write-heavy, ma può ritardare a lungo i lettori.

```
function end_write()
    busy_on_write = false
    if (in_queue(write)) signal(write)
    else                  signal(read)
```

#### 2.5.3.1 Sintesi differenze (monitor)

Caratteristica	Semafori	Monitor S1	Monitor S2	Monitor S3
Ingresso lettore	se db libero se o già lettori on_write = (il primo fa false down(db))	busy_- se = on_write = false e in_- queue(write) = false	busy_- come S2	
Uscita scrittore: chi svegliare?	up(db) risve- glia uno; se è lettore poi entrano tutti a cascata	read se c'è, al- trimenti write altrimenti read	read se c'è, al- trimenti write altrimenti read	<b>write</b> se c'è, altrimenti read
Bias / rischio starvation	<b>Lettori</b> (possibi- le attesa indefi- nita degli scri- ttori)	<b>Lettori</b>	Bilanciata (evi- ta writer starva- riti (ritardo let- tori))	<b>Scrittori</b> prefe- rita (ritardo let- tori)

*Concorrenza ammessa:* più lettori in parallelo; uno scrittore per volta.

Tabella 2.2: Sintesi differenze: Semafori vs Monitor (tre varianti).

## 2.6 IPC: InterProcess Communication

Lo *scambio di messaggi* è un modello d'IPC in cui i processi cooperano inviando e ricevendo pacchetti di dati tramite primitive di alto livello fornite dal sistema operativo. Lo stesso modello si applica sia a processi sulla stessa macchina sia a nodi diversi in rete. Questo modello di comunicazione è **asincrono**.

### 2.6.1 Primitive di base

Le chiamate minime sono:

- `send(destinazione, messaggio)`
- `receive(sorgente, messaggio)`

Per impostazione comune `receive` è **bloccante** se non c'è un messaggio disponibile; `send` può diventare bloccante quando il buffer sottostante è pieno. Molte API offrono varianti non-bloccanti o con timeout.

### 2.6.2 Buffering e sincronizzazione

Il kernel gestisce una coda di messaggi (buffer) che disaccoppia mittente e destinatario:

- **Capienza 0** (rendezvous): `send` e `receive` si incontrano; nessun accodamento.
- **Capienza finita  $N$** : fino a  $N$  messaggi accodati; oltre  $N$  la `send` blocca/fallisce.
- **Capienza “illimitata”**: modello teorico; nella pratica si usano quote e limiti.

L'atomicità dell'invio è garantita per messaggi piccoli (dipende dall'OS/API).

### 2.6.3 Barriere

Un altro metodo di sincronizzazione è quello delle **barriere**: alcune applicazioni vengono suddivise in fasi e hanno la regola che nessun processo può eseguire nella fase successiva senza che gli altri lo abbiano fatto. Difatti, quando un processo raggiunge una barriera è bloccato fin quando tutti gli altri processi non la raggiungono.

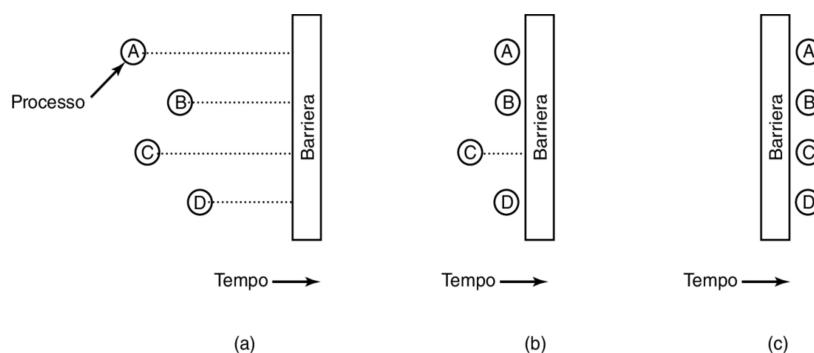


Figura 2.20: Uso di una barriera. (a) I processi si avvicinano a una barriera. (b) Tutti i processi escluso uno sono bloccati alla barriera. (c) Quando l'ultimo processo raggiunge la barriera, tutti vengono lasciati passare.

### 2.6.4 Indirizzamento: diretto vs indiretto

**Diretto.** Il mittente indica esplicitamente il destinatario (`send(P2, m)`): adatto a comunicazioni 1-1.

**Indiretto (mailbox/porte).** I processi usano una **mailbox** nominata: `send(M, m)` e `receive(M, m)`. Questo generalizza a  $N$  produttori e  $M$  consumatori, con politiche di consegna (FIFO dei messaggi; risveglio del primo ricevente in attesa). Le mailbox hanno permessi, capienza e ciclo di vita propri.

### 2.6.5 Esempi

#### 2.6.5.1 Esempio base (produttore/consumatore)

##### Producer

```
function producer()
  while true do
    item = produce_item()
    send(consumer, item)
```

##### Consumer

```
function consumer()
  while true do
    receive(producer, item)
    consume_item(item)
```

#### 2.6.5.2 Esempio con mailbox ( $N$ produttori, $M$ consumatori)

##### Producer

```
mailbox M // coda con capienza N

function producer(i)
  while true do
    x = produce_item()
    send(M, x) // blocca se M è piena
```

##### Consumer

```
function consumer(j)
  while true do
    receive(M, x) // blocca se M è vuota
    consume_item(x)
```

### Varianti di semantica comuni

- **Send:** bloccante / non-bloccante / con timeout.
- **Receive:** bloccante / non-bloccante / selettiva (da una certa sorgente) / *peek*.
- **Ordine e priorità:** FIFO per mailbox, o priorità sui messaggi.
- **Gestione errori:** mailbox inesistente/piena, destinatario non raggiungibile, ecc.

### Pro e contro (sintesi)

**Vantaggi:** nessuna memoria condivisa, isolamento e sicurezza maggiori, naturale estensione a sistemi distribuiti, disaccoppiamento fra mittenti e destinatari.

**Svantaggi:** overhead di *syscall* e copie dati (mitigabile con tecniche zero-copy o pagine condivise per messaggi grandi), possibili attese dovute a buffer pieni/vuoti (*backpressure*).

## 2.7 Scheduling

### 2.7.1 Introduzione e obiettivi dello scheduling

#### 2.7.1.1 Definizione, scheduler e dispatcher

Lo *scheduling* decide quale processo pronto usa la CPU quando si libera. La scelta è presa dallo *scheduler* secondo un algoritmo; il *dispatcher* ripristina lo stato del processo selezionato tramite *context switch*, introducendo una **latenza di dispatch** (overhead). I processi alternano fasi di uso della CPU (*CPU burst*) a periodi di attesa per I/O.

#### 2.7.1.2 CPU burst e tipologie di processi

Distinguiamo:

- **CPU-bound**: burst lunghi, I/O sporadico;
- **I/O-bound**: burst brevi, I/O frequente.

Un processo può mutare natura nel tempo (es. un compilatore). Con CPU più veloci, a parità di compito, i burst si accorciano e il carico medio diventa più I/O-bound; conviene quindi privilegiare chi rilascia presto la CPU per massimizzare la sovrapposizione CPU/I/O.

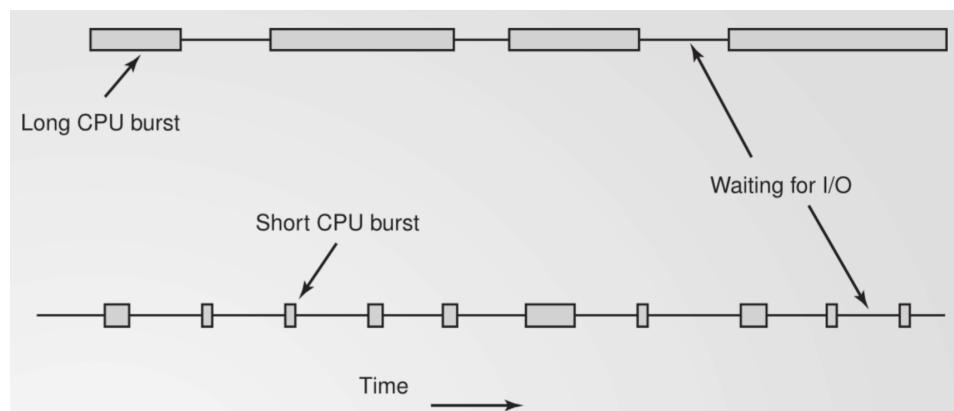


Figura 2.21: Esempio di scheduling con CPU burst differenti.

#### 2.7.1.3 Quando si attiva lo scheduler

Lo scheduler interviene alla terminazione o creazione di processi, su chiamate bloccanti (e relativi interrupt di completamento) e, nei sistemi *preemptive*, anche su interrupt periodici del timer. La decisione deve essere rapida per ridurre l'overhead complessivo.

#### 2.7.1.4 Obiettivi per ambiente

Obiettivi *comuni*: equità nell'uso della CPU e bilanciamento delle risorse (anche tra core su sistemi multicore).

**Batch**: massimizzare il *throughput* e minimizzare *turnaround* e *tempo di attesa*, bilanciando l'equità per non penalizzare sistematicamente i job lunghi.

**Interattivo**: minimizzare il *tempo di risposta*; la *prelazione* è essenziale per evitare il monopolio della CPU, spesso favorendo i processi I/O-bound.

**Real-time:** rispetto delle scadenze e prevedibilità; la prelazione si usa quando necessario in base alle caratteristiche dei task, in genere brevi e reattivi.

### 2.7.2 Scheduling per sistemi batch

Nei sistemi batch si studiano principalmente tre politiche: **FCFS**, **SJF** e **SRTN**. Le metriche di interesse sono il *tempo di attesa* (quanto un job resta pronto senza eseguire) e il *tempo di completamento/turnaround*.

#### 2.7.2.1 FCFS (First-Come First-Served)

Coda FIFO, non-preemptive: quando la CPU si libera, si estrae il primo job in ordine di arrivo e lo si esegue fino al termine. In caso di pareggio si usa l'ordine di inserimento.

**Pro/Contro:** Semplice ed “equo” ma soffre dell’*effetto convoglio*: un job lungo all’inizio può far attendere a lungo quelli brevi.

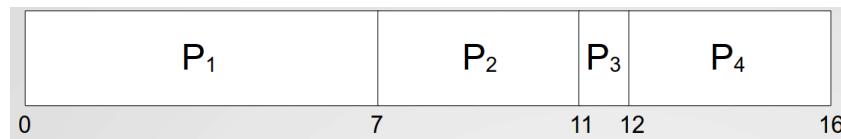


Figura 2.22: Scheduling di tipo FCFS

#### 2.7.2.2 SJF (Shortest Job First)

Non-preemptive: tra i job già disponibili si sceglie quello con *durata totale* più breve e lo si esegue fino al termine. Richiede di conoscere (o stimare) il burst di CPU.

**Pro/Contro:** Minimizza l’attesa media solo se le durate sono note e i job sono tutti già in coda; può causare *starvation* dei job lunghi.

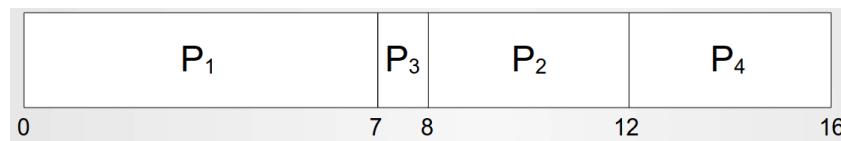


Figura 2.23: Scheduling di tipo SJF

#### 2.7.2.3 SRTN (Shortest Remaining Time Next)

Versione *preemptive* di SJF: ad ogni arrivo (o evento di timer) si confronta la *durata residua* del job in esecuzione con quelle dei job pronti; se esiste un job con residuo minore, si *preleva* la CPU e si esegue quello.

**Pro/Contro:** Ottime medie di attesa/completamento, ma più overhead (più context switch) e possibile *starvation* per job molto lunghi.

#### 2.7.2.4 Esempio unico (stesso workload, tre politiche)

**Workload:**  $P_1(0, 7)$ ,  $P_2(2, 4)$ ,  $P_3(4, 1)$ ,  $P_4(5, 4)$  (formato: arrivo, durata).

**FCFS.** Ordine fissato dagli arrivi senza prelazione:  $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$ . Attesa media  $\approx 4.75$ ; completamento medio  $\approx 8.75$ . **SJF.** A  $t = 0$  è disponibile solo  $P_1$  (gli altri non sono ancora arrivati),

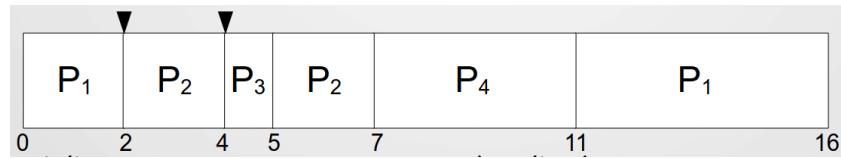


Figura 2.24: Scheduling di tipo SRTN

quindi  $P_1$  esegue fino a  $t = 7$ . Poi, tra i job pronti, si sceglie sempre il più corto:  $P_3 \rightarrow P_2 \rightarrow P_4$ . Attesa media  $\approx 4$ ; completamento medio  $\approx 8$ . **SRTN**. Si parte con  $P_1$ ; a  $t = 2$  arriva  $P_2$  (residuo  $P_1 = 5 > 4$ ) e preleva la CPU. A  $t = 4$  arriva  $P_3$  (durata 1) e preleva  $P_2$ . Finito  $P_3$  ( $t = 5$ ), tra i pronti il residuo minore è  $P_2$  (2), poi  $P_4$  (4) e infine si chiude con  $P_1$  (5): sequenza  $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_2 \rightarrow P_4 \rightarrow P_1$ . Attesa media  $\approx 3$ ; completamento medio  $\approx 7$ .

**Cosa cambia tra le tre:** FCFS rispetta l'ordine d'arrivo ma può far “trainare” i brevi dietro a un lungo; SJF migliora le medie perché, quando può scegliere, dà priorità ai burst brevi (ma non può anticipare job non ancora arrivati); SRTN spinge oltre il vantaggio prelevando la CPU ai job lunghi quando arrivano job più corti, riducendo ulteriormente attese e tempi di completamento medi a costo di più preemption.

### 2.7.3 Scheduling nei sistemi interattivi

Nei sistemi interattivi l'obiettivo primario è la *reattività*: interessa ridurre il *tempo di risposta* percepito dall'utente più che ottimizzare medie di attesa/turnaround tipiche dei batch. Poiché non possiamo conoscere a priori la durata dei job e i processi I/O bound tendono a rilasciare rapidamente la CPU mentre i CPU bound possono monopolizzarla, è indispensabile la *prelazione* per evitare l'effetto convoglio e garantire servizio a tutti.

#### 2.7.3.1 Round Robin (RR)

RR è la versione preemptive del FCFS: quando la CPU si libera si estrae il primo processo dalla coda dei pronti e lo si esegue per un *quanto temporale prefissato* (*timeslice q*). Alla scadenza del quanto, se il processo non si è bloccato volontariamente (es. per I/O) viene prelazionato e reinserito in fondo alla coda; in caso di blocco rientrerà più tardi, sempre in fondo, con un nuovo quanto. Valori tipici di  $q$  sono 20–50 ms: se  $q$  è troppo piccolo crescono i context switch (overhead), se è troppo grande la prelazione diventa inefficace. Con  $n$  processi pronti, ciascuno ottiene circa  $1/n$  della CPU e attende al più  $(n - 1)q$  ms prima di essere di nuovo servito. RR riduce il rischio di starvation e, di fatto, fa emergere la natura dei processi: i CPU-bound sono prelazionati spesso, gli I/O-bound cedono spontaneamente la CPU prima della scadenza.

#### 2.7.3.2 Scheduling con priorità

Nella pratica i processi non sono tutti uguali: si assegna dunque una *priorità* (valore numerico) e si sceglie sempre il processo con priorità più alta tra quelli pronti. Le priorità possono essere *statiche* (impostate una volta) oppure *dinamiche* (variano in base al comportamento recente, es. favorendo chi usa poca CPU o chi è più interattivo). La versione preemptive effettua prelazione immediata quando un processo di priorità più alta arriva o viene promosso. Queste politiche possono causare *starvation* dei processi a bassa priorità; per evitarla si applica *aging* (incremento graduale della priorità durante

l'attesa). Notare che SJF può essere visto come un particolare scheduling a priorità in cui la “priorità” è proporzionale alla brevità prevista del burst.

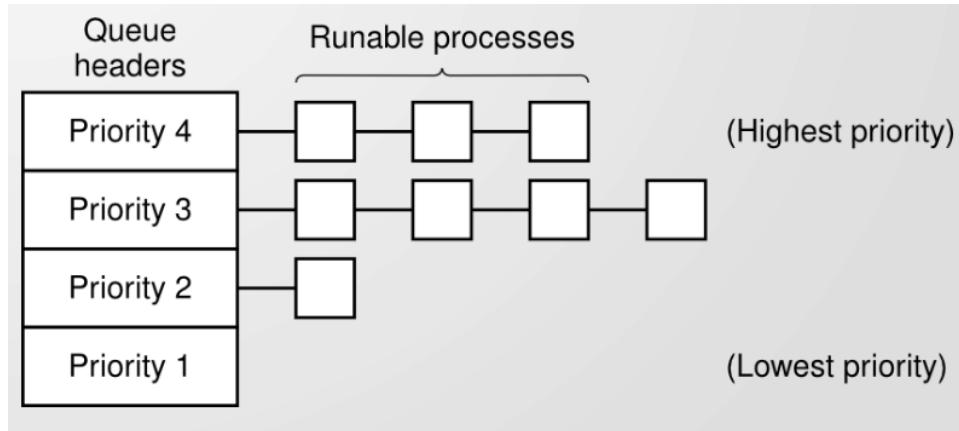


Figura 2.25: Scheduler con coda di priorità

#### 2.7.3.3 Code multiple (con retroazione)

Si organizzano più code, ciascuna con una classe di priorità e, spesso, con un quanto diverso (più alto  $\Rightarrow$  quanto più corto). Lo scheduler “verticale” decide da quale coda estrarre (tipicamente dalla più alta non vuota), mentre quello “orizzontale” sceglie il prossimo processo all’interno della coda (spesso RR). Con *retroazione (feedback)* si spostano i processi tra code in base all’uso effettivo della CPU: chi consuma tutto il quanto scende (più CPU bound), chi cede spesso la CPU prima del quanto sale (più I/O bound). In questo modo si privilegiano i carichi interattivi senza affamare gli altri, specie se si combinano quanta crescenti verso il basso e meccanismi di aging: un processo che non viene eseguito da tanto, viene aumentato di priorità grazie a questi meccanismi evitando dunque la **starvation**.

#### 2.7.3.4 Shortest Process Next (SPN)

Per i carichi interattivi è utile stimare la durata del *prossimo* burst di CPU e privilegiare chi lo ha più breve. Si usa una previsione esponenzialmente pesata:

$$S_{n+1} = (1 - a) S_n + a T_n,$$

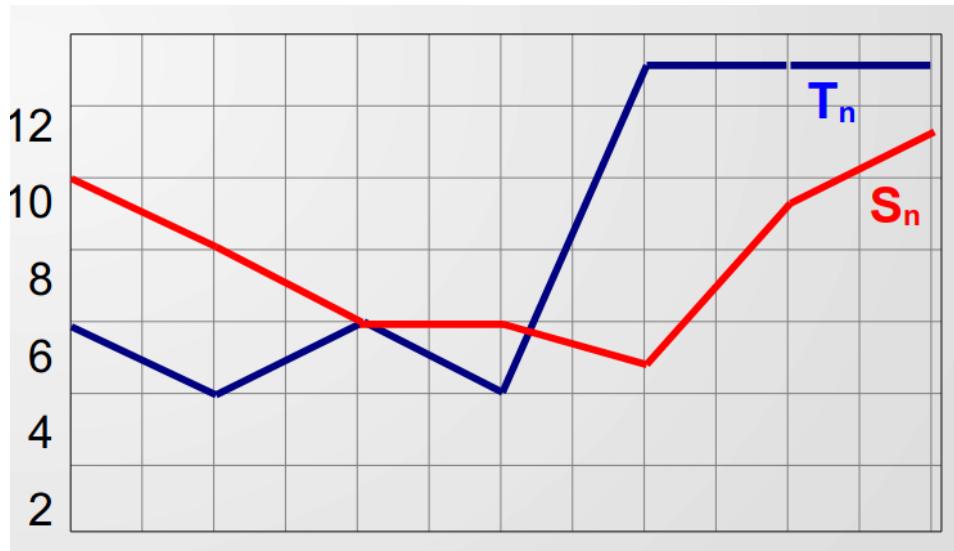
dove  $T_n$  è l’ultimo burst osservato,  $S_n$  la stima precedente e  $a \in [0, 1]$  regola quanto “memoria” o “reattività” vogliamo:  $a$  grande segue più da vicino l’ultima misura,  $a$  piccolo stabilizza. SPN è l’analogo interattivo dello SJF, ma usa stime invece di durate note.

#### 2.7.3.5 Scheduling garantito

Qui si stabilisce a priori una quota “promessa” di CPU (percentuale) per ciascun processo o classe e si cerca di farla rispettare nel tempo. Lo scheduler misura l’uso effettivo e favorisce chi è “rimasto indietro” rispetto alla propria quota, fornendo così una nozione pratica di equità temporale.

#### 2.7.3.6 Scheduling a lotteria

Si assegnano *ticket* ai processi e, ad ogni decisione, si estrae a caso un ticket: in media l’uso di CPU è proporzionale al numero di ticket assegnati (facile da ponderare o trasferire tra processi cooperanti). È

Figura 2.26: Esempio di scheduler SPN con  $a = 1/2$ 

un modo semplice per realizzare quote elastiche e, grazie alla casualità, ridurre la sincronizzazione rigida tra processi.

#### 2.7.3.7 Fair-share scheduling

L'equità può essere definita a livello di *utente* o *gruppo*, non solo di processo: si ripartisce la CPU tra entità logiche superiori e poi si divide la loro quota tra i processi appartenenti. In ambiente multiutente evita che chi ha molti processi superi, in media, chi ne ha pochi.

## 2.8 Scheduling nei thread

### 2.8.1 Thread utente

Qui il kernel ignora l'esistenza dei thread a livello utente, per lo scheduler del sistema run-time vanno bene infatti tutti gli algoritmi visti sopra ma senza l'utilizzo della prelazione. Lo scheduler infatti conosce solo i processi e ne sceglie uno da attenzionare, dopo la libreria di livello utente con il suo scheduler sceglie il thread.

### 2.8.2 Thread kernel

In questo caso, si può schedulare in base alla pesantezza del context-switch (alternativa: considerare tutti i thread uguali). In pratica si tende a scegliere, nei limiti dell'equità (prima o poi si dovrà schedulare qualche thread di qualche altro processo), thread imparentati tra loro. Se scelgo un thread fratello di quello uscente c'è un vantaggio perché nel fare il context-swtich non si deve riprogrammare la MMU.

## 2.9 Scheduling su sistemi multiprocessore

Con più CPU (o core) non basta scegliere *chi* esegue: occorre anche stabilire *dove* farlo eseguire, distribuendo il carico in modo che tutti i core lavorino senza colli di bottiglia. Due sono gli approcci classici. Nella *multielaborazione asimmetrica* (ASMP) un core *master* gestisce le funzioni di sistema (scheduler e routine di kernel), mentre gli altri core eseguono principalmente codice utente: è semplice, ma scala male perché il master diventa presto il collo di bottiglia. La *multielaborazione simmetrica* (SMP), oggi dominante, assegna a ogni core gli stessi compiti (scheduler incluso) e consente di scalare aumentando i core.

In SMP la distribuzione del lavoro dipende dall'organizzazione delle code dei pronti. Con una *coda condivisa* tutti i core pescano dallo stesso contenitore: è facile riutilizzare gli algoritmi visti (RR, priorità, SJF/SRTF, ecc.), ma serve sincronizzazione (lock) e, all'aumentare dei core, la coda può diventare un punto di contesa. Alternativamente si usano *code percore*: ogni core schedula dalla propria coda, riducendo la contesa; resta però il problema del *bilanciamento del carico*, perché le code si svuotano e si riempiono dinamicamente in funzione di arrivi, blocchi e terminazioni.

Per bilanciare si adottano meccanismi di *migrazione* dei processi tra code. La *migrazione guidata* (*push*) è periodica: una routine confronta le code e sposta processi da quelle più piene a quelle più vuote; richiede lock ma in modo sporadico. La *migrazione spontanea* (*pull*) è reattiva: quando una coda si svuota, il relativo core "ruba" processi da una coda più ricca. In pratica i sistemi moderni usano un approccio ibrido (*push/pull*), così da mantenere i core occupati con overhead contenuto.

Un ulteriore aspetto è la *predilezione* per il processore (*processor affinity*), utile per sfruttare la località delle cache. La *predilezione debole* consente al processo di migrare liberamente (nessun vincolo rigido); la *predilezione forte* vincola un processo a un core specifico, violabile solo in casi eccezionali. Bilanciare *affinità* e *bilanciamento* è cruciale: troppa migrazione raffredda le cache, troppa rigidità crea squilibri. Qualunque sia l'organizzazione, all'interno di ciascuna coda si possono riapplicare gli algoritmi di scheduling già studiati, senza doverne inventare di nuovi.

## 2.10 Scheduler nei moderni sistemi operativi

**Windows:** adotta code di priorità multiple con priorità di base configurabili e *dinamiche* per migliorare l'interattività: in particolare applica *boost* temporanei ai processi che rientrano da I/O (o ai task

in foreground), così da ridurre la latenza percepita dall'utente. Per mitigare fenomeni di *inversione di priorità*, il kernel impiega euristiche di *autoboost* che rialzano momentaneamente la priorità dei task che detengono risorse attese da task più prioritari. In configurazione multiprocessore Windows usa runqueue percore e meccanismi periodici di bilanciamento con rispetto dell'affinità quando possibile.

**Linux:** unifica processi e thread nel concetto di *task* e usa il *Completely Fair Scheduler* (CFS), uno scheduler “garantito” che approssima una ripartizione equa del tempo di CPU. Ogni task ha un *virtual run time* (vruntime) che cresce in funzione del tempo di CPU effettivamente consumato, pesato in base alla priorità (peso derivato dal *nice*). I task sono organizzati in un albero Red–Black: si esegue sempre il task con vruntime più piccolo; quando il suo vruntime “superà” i concorrenti, scatta una nuova selezione (prelazione). I task I/O bound, cedendo spesso la CPU, accumulano vruntime più lentamente e ricevono naturalmente più prontezza; la fame (*starvation*) è evitata perché chi non gira non accumula vruntime e prima o poi torna minimo. In multiprocessore Linux usa runqueue perCPU con bilanciamento push/pull e rispetta l'affinità compatibilmente con l'equità.

**MacOS:** (kernel XNU, di derivazione Mach + BSD) impiega uno scheduler a *code di priorità* con politiche di *timesharing* e classi realtime, arricchito da euristiche per favorire l'interattività dell'interfaccia grafica. Anche qui sono presenti affinità per il processore, code percore e politiche di bilanciamento per mantenere uniformemente occupati i core.



# Capitolo 3

## Gestione della memoria

### 3.1 Gerarchia della memoria

La memoria di un calcolatore è organizzata gerarchicamente e ordinata per *tempo di accesso* (dalla più veloce alla più lenta) e, inversamente, per *capacità*. In cima ci sono *registri* e *cache* gestiti direttamente dalla CPU; seguono la *memoria centrale* (RAM), quindi i dispositivi di *memoria secondaria* e d'archivio. Le prime sono *volatili*, le seconde sono *non volatili* e molto più capienti, ma non sono direttamente indirizzabili dalla CPU: vi si accede tramite il sottosistema di I/O e i driver del sistema operativo. Nella gerarchia, la RAM rappresenta il livello più basso *direttamente utilizzabile* dalla CPU per l'esecuzione dei programmi. L'astrazione della memoria che i processi vedono viene introdotta progressivamente dal sistema operativo con livelli via via più complessi.

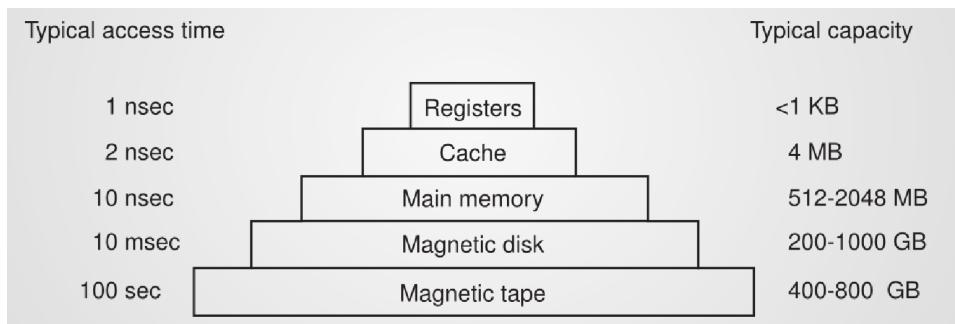


Figura 3.1: Gerarchia di memoria, dalla più lenta alla più veloce

### 3.2 Gestione dei processi in memoria centrale

Storicamente, i primi sistemi (mainframe anni '60 e i primi PC) non introducevano astrazioni sulla memoria: i programmi usavano *indirizzi fisici* e il modello era pensato per l'esecuzione di un singolo programma per volta. In questo contesto eseguire più programmi contemporaneamente è difficile perché occorre dividere a priori lo spazio fisico e ogni programma deve essere preparato per girare in una regione prefissata. Tale impostazione si ritrova ancora in sistemi embedded molto semplici, dove l'overhead di ulteriori astrazioni non è giustificato.

Quando si tenta la *multiprogrammazione senza astrazione*, cioè mantenere più processi in memoria contemporaneamente, emergono subito due problemi principali:

1. **Rilocazione** degli indirizzi:

- *a compile-time* (indirizzi assoluti “cablati” nel codice);
  - *rilocazione statica in fase di caricamento* (il loader corregge tutti i riferimenti), con il rovescio della medaglia di un *rallentamento del caricatore*.
2. **Protezione della memoria:** occorre evitare che un processo acceda alle aree di un altro o del kernel. Una soluzione storica è lo schema *lock & key*, con blocchi di memoria dotati di “chiavi” e un *PSW* (Program Status Word) che contiene la chiave associata al processo in esecuzione.

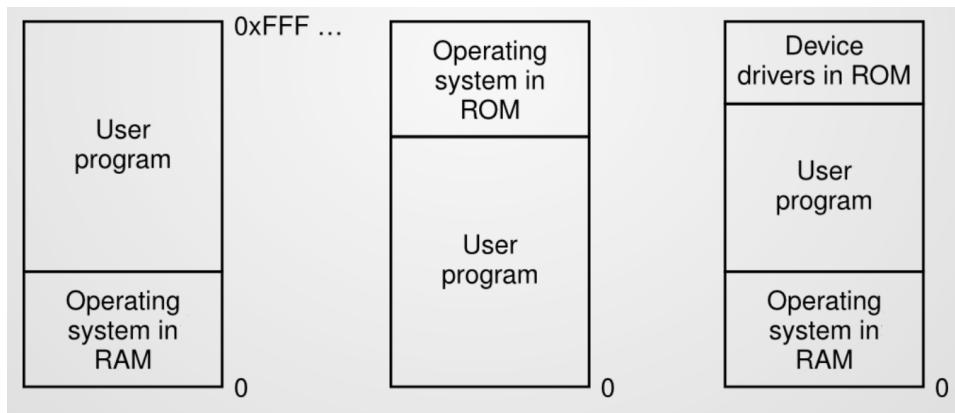


Figura 3.2: Tre possibili organizzazioni della memoria *senza astrazione*: (a) sistema operativo in RAM e programma utente nello spazio restante; (b) sistema operativo in ROM con programma utente in RAM; (c) sistema operativo in RAM con driver di dispositivo in ROM.

Senza un’astrazione adeguata, la suddivisione dello spazio deve essere decisa *a priori*: o si riserva una parte fissa alla memoria del sistema operativo e il resto ai programmi, oppure si carica l’OS in ROM/flash (o lo si porta in RAM tramite un piccolo firmware di bootstrap), ma in ogni caso rimangono i limiti di flessibilità e di portabilità del codice. La conseguenza pratica è che o si eseguono i processi in modo esclusivo, oppure si deve “adattare” (ricompilare/ricollocare) il codice per la diversa partizione di memoria. Queste difficoltà motivano l’introduzione delle astrazioni moderne (spazio degli indirizzi per processo, rilocazione *dinamica* con registri base/limite o MMU e, più avanti, memoria virtuale), che però appartengono agli stadi successivi della trattazione.

### 3.2.1 Rilocazione

Quando si mantengono più programmi contemporaneamente in RAM senza alcuna astrazione, ogni programma è stato scritto usando uno *spazio di indirizzamento logico* che tipicamente parte da 0 e arriva ad un proprio limite ( $[0, LP]$ ). Se due programmi vengono semplicemente collocati in memoria fisica in regioni diverse, i riferimenti presenti nel loro codice (salti, chiamate, puntatori a dati) continuano però a usare indirizzi logici come se ciascuno occupasse l’intervallo  $[0, LP]$ . Ne segue che un salto logico a 1000, per esempio, nel programma  $P_2$  potrebbe puntare dentro la regione fisica del programma  $P_1$  se non si interviene: è dunque necessaria la **rilocazione**, cioè l’adattamento degli indirizzi usati dal codice alla regione fisica effettivamente assegnata al processo. Senza supporti hardware o astrazioni dedicate, la rilocazione si può realizzare solo in due modi:

**Rilocazione a *compile-time*.** Gli indirizzi assoluti vengono fissati durante la compilazione/assemblaggio in base ad un layout predefinito della memoria. Se si cambia la destinazione (ad es. si decide di caricare

il programma in un'altra regione), è necessario ricompilare o riassemblare il codice. La tecnica è semplice ma poco flessibile.

**Rilocazione statica a loading-time.** Il programma viene prodotto con indirizzi *relocabili* rispetto a 0 (indirizzi logici). Al momento del caricamento in RAM, il *loader* determina la base fisica dell'area assegnata al processo e *corregge* ogni riferimento marcato come relocabile sommando la base: se a  $P_2$  è stato assegnato l'intervallo fisico [20000, 30000], un riferimento logico 3000 diventa 23000. In pratica, il formato eseguibile contiene una *tabella di rilocazione* che indica quali campi debbano essere modificati. Il costo è un caricamento più lento (il loader deve scansionare e patchare i riferimenti), ma il programma, una volta rilocato, può essere eseguito senza ulteriori interventi.

**Protezione della memoria.** In assenza di un meccanismo di protezione, un processo potrebbe leggere o scrivere aree appartenenti ad altri processi o al sistema operativo. Storicamente, una soluzione elementare è lo schema *lock & key*: la RAM è divisa in blocchi (storicamente da 2 KB) etichettati con una *chiave* di protezione; la chiave del processo in esecuzione è contenuta nella *Program Status Word* (PSW). Per ogni accesso in memoria la chiave della PSW viene confrontata con quella del blocco: se coincidono l'accesso è consentito, altrimenti negato. Ad ogni *context switch* la chiave nella PSW viene aggiornata con quella del nuovo processo. Questo meccanismo è semplice ma poco scalabile (numero limitato di chiavi) e non offre la flessibilità dei meccanismi moderni.

### 3.3 Spazio degli indirizzi

Lo *spazio degli indirizzi* è la prima astrazione che consente a ciascun processo di vedere una memoria propria, indipendente dalla posizione fisica in RAM. L'idea base è associare al processo due registri hardware: il **registro base** (RB), che contiene l'indirizzo fisico iniziale della sua area, e il **registro limite** (RL), che contiene la *lunghezza* dell'area stessa. Questi valori sono parte dello stato del processo (PCB) e il sistema operativo li salva e li ripristina ad ogni context switch. La verifica e la traduzione degli indirizzi vengono effettuate dalla CPU (o, nei sistemi più evoluti, dalla *Memory Management Unit* - MMU).

Con la **rilocazione dinamica** ogni accesso in memoria prodotto dal programma usa un indirizzo *logico*  $L$  (nel suo spazio, tipicamente a partire da 0). In hardware si eseguono due operazioni in tempo di esecuzione (*run-time*):

1. **Controllo di protezione:** se  $L \geq RL$  l'accesso è fuori dallo spazio del processo; viene generata una *trap* e il controllo passa al kernel, che gestisce l'errore (tipicamente terminando il processo o segnalando una violazione d'accesso).
2. **Traduzione (rilocazione):** se il controllo va a buon fine, l'indirizzo *fisico* è calcolato come  $F = RB + L$ .

Questo meccanismo fornisce simultaneamente *rilocazione* (il programma può essere caricato ovunque) e *protezione* (un processo non può accedere fuori dal proprio intervallo). I registri base/limite devono essere aggiornati dal SO quando cambia il processo in esecuzione, così che ogni processo veda il proprio intervallo. Tale schema è stato storicamente usato in sistemi come CDC 6600 e Intel 8088, ed è alla base dell'idea moderna di gestione dello spazio degli indirizzi affidata alla MMU.

È utile confrontarla con la *rilocazione statica*: lì gli indirizzi del programma vengono *corretti una sola volta* dal loader in fase di caricamento (patch degli offset) e poi restano fissi; qui invece la somma  $RB + L$  e il controllo con  $RL$  avvengono *ad ogni accesso*, ma in hardware, quindi con costo trascurabile

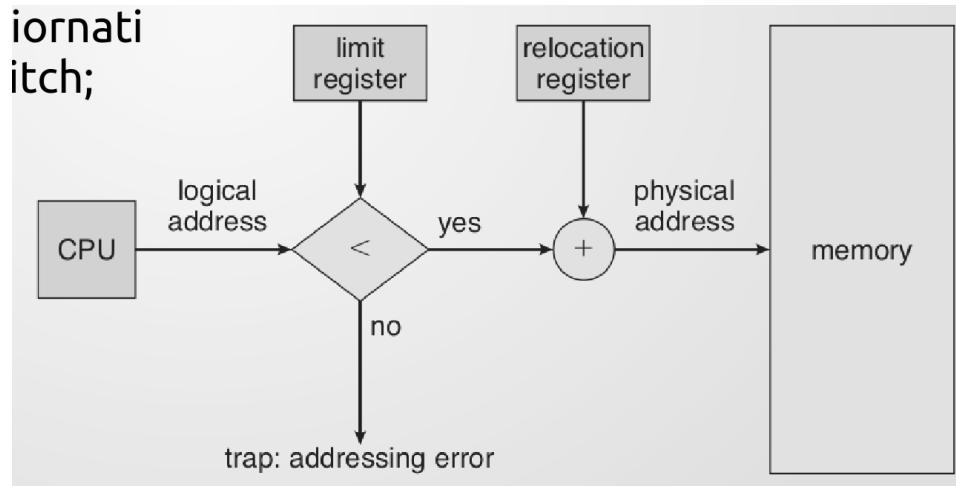


Figura 3.3: Diagramma a blocchi della rilocazione dinamica

per il software applicativo. Inoltre, con RB/RL il SO può spostare un processo in un'altra regione fisica (ad esempio dopo uno *swapping*) semplicemente aggiornando RB quando il processo non è in esecuzione, senza dover riscrivere il codice.

## 3.4 Swapping

In un sistema multiprogrammato la somma delle richieste di RAM dei processi attivi può superare la memoria fisica disponibile. La tecnica più semplice per regolare il *grado di multiprogrammazione* è lo **swapping**: il sistema operativo seleziona uno o più processi residenti e ne *sposta* l'immagine (codice, dati, stack) su un'area di disco dedicata (*backing store/swap area*), liberando RAM per altri processi. Il processo “parcheggiato” non è terminato: il suo PCB resta in memoria e lo stato cambia in (*ready/blocked*)-*suspended*. Quando torna spazio, il processo viene *swappato in* e riprende esattamente dal punto di sospensione. La decisione di chi spostare è compito dello **medium-term scheduler** (o *swapper*), distinto dallo scheduler di breve termine: tipicamente privilegia processi che liberano molta memoria o che sono in background.

### 3.4.1 Rilocazione e ripresa.

Con *rilocazione statica* (correzione degli indirizzi al caricamento), se il processo rientra in una regione fisica diversa da quella originale occorre rieseguire la fase di patch (o ricaricare il codice) prima di ripartire. Con *rilocazione dinamica* (registri base/limite o MMU) è sufficiente aggiornare il registro di base: il flusso di esecuzione non cambia e non serve riscrivere il codice.

### 3.4.2 I/O pendenti.

È pericoloso swappare un processo che ha I/O in corso (es. DMA) verso buffer nel suo spazio utente: i dati potrebbero sovrascrivere la memoria nel frattempo riassegnata ad altri. Le soluzioni classiche sono: (i) *non swappare* processi con I/O pendente; (ii) instradare l'I/O tramite buffer del kernel e copiare i dati al completamento; (iii) “*pinnare*” (bloccare) le aree coinvolte in I/O per impedirne lo spostamento.

### 3.4.3 Allocazione della memoria e contiguità.

Nel modello a **partizioni contigue** lo spazio di un processo deve risiedere in un intervallo fisico contiguo, perciò lo swapping interagisce con la politica di allocazione:

- **Partizioni a dimensione fissa:** la memoria è suddivisa in blocchi di taglia prefissata. È semplice (lo swap-in trova velocemente un “posto”), ma comporta *frammentazione interna*: se il processo non riempie il blocco, la parte restante è sprecata.
- **Partizioni a dimensione variabile:** si crea dinamicamente un blocco della misura più aderente alla richiesta del processo. Riduce lo spreco interno, ma nel tempo genera *frammentazione esterna*: lo spazio libero si spezzetta in “buchi” non contigui, talvolta incapaci di ospitare nuovi processi nonostante memoria totale sufficiente.

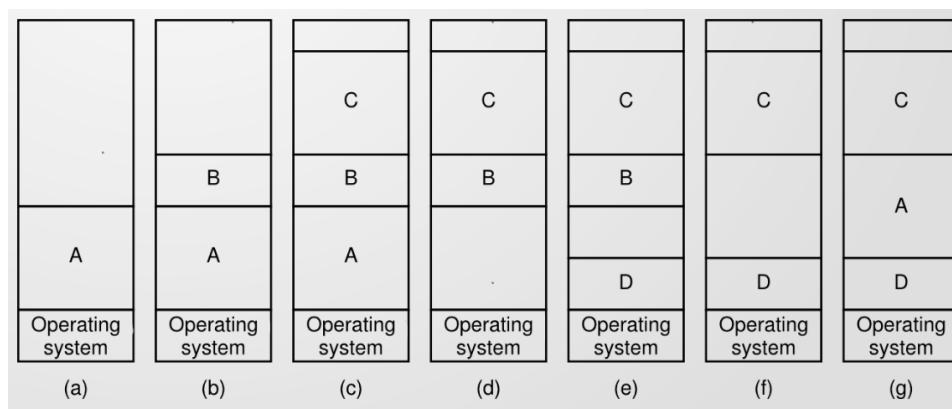


Figura 3.4: Cambiamenti nell'allocazione della memoria quando i processi arrivano in memoria e la lasciano.

Per mitigare la frammentazione esterna si può eseguire la **compattazione** (spostare in RAM le partizioni per riunire i buchi), ma è costosa e introduce pause; in alternativa si ricorre più spesso allo swapping per liberare intervalli ampi. Bisogna inoltre considerare che la *dimensione effettiva* di un processo può variare a run-time (crescita di heap/stack): si può lasciare “margine” interno (introducendo frammentazione interna) oppure riallocare/spostare la partizione quando occorre.

In sintesi, lo *swapping* regola il numero di processi residenti spostando intere immagini su disco; funziona bene con indirizzamento contiguo ma soffre di problemi di frammentazione e di gestione dell’I/O pendente. Le tecniche più moderne (segmentazione/paginazione e memoria virtuale) superano il vincolo di contiguità fisica, riducendo la necessità di spostare interi processi.

## 3.5 Gestione dell'allocazione

Per sapere quali regioni di RAM sono occupate e quali libere il sistema operativo mantiene strutture dati dedicate che registrano l’evoluzione delle allocazioni. Tutte le tecniche assumono la definizione di una **unità minima allocabile** (o *blocco*): la memoria fisica viene concettualmente suddivisa in blocchi tutti della stessa taglia, e le strutture tengono traccia dello stato di tali blocchi.

### 3.5.1 Unità minima allocabile: compromessi

La scelta della taglia influisce su spazio e prestazioni:

- se è *troppo piccola* aumentano il numero di blocchi e la dimensione delle strutture di gestione (overhead in RAM);
- se è *troppo grande* cresce l'*arrotondamento* delle richieste e quindi la **frammentazione interna**.

### 3.5.2 Bitmap

L'idea è associare ad ogni blocco un bit in una tabella:

- **struttura**: un vettore di bit (1 = occupato, 0 = libero) di lunghezza pari al numero di blocchi della RAM;
- **vantaggi**: rappresentazione compatta, test e scansioni lineari semplici;
- **limiti**: la dimensione è *fissa* a prescindere dal numero di processi in memoria; troppo finezza nella taglia dei blocchi fa esplodere il numero di bit (overhead).

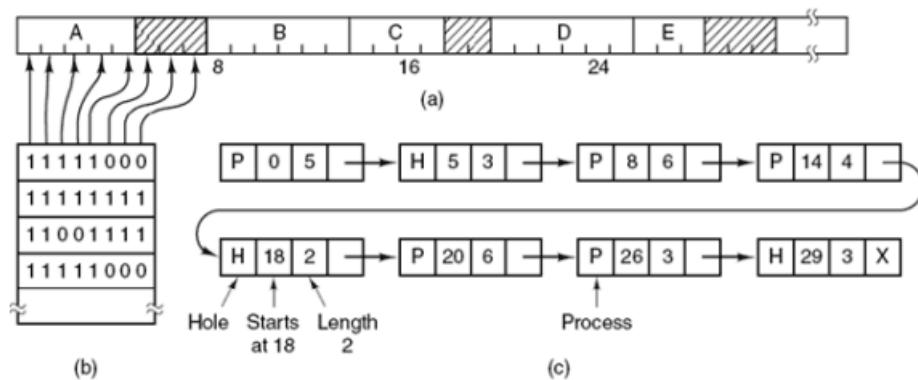


Figura 3.5: Strutture per tracciare l'allocazione in RAM: a sinistra *bitmap* (1=occupato, 0=libero) con mappatura blocco→bit; a destra *liste* ordinate per indirizzo (nodi P=partizione, H=buco) con coalescenza, usate da first/next/best/worst fit.

### 3.5.3 Liste di partizioni

Per evitare una struttura di dimensione fissa si usa una **lista doppiamente concatenata** (abitualmente ordinata per indirizzo fisico) i cui nodi rappresentano o una *partizione occupata* o un *buco* (intervallo libero). Ogni nodo contiene: identificativo/processo, indirizzo di inizio, lunghezza in blocchi.

- **coesescenza**: quando una partizione si libera, se i nodi liberi adiacenti sono contigui si fondono in un unico nodo più grande, riducendo il numero di elementi e i tempi di ricerca;
- **memoria**: la lista risiede in RAM (tipicamente in area libera). Paradossalmente, quando lo spazio libero si riduce, la lista può crescere leggermente, ma l'overhead resta di ordini di grandezza inferiori alla memoria gestita;
- **perché doppia**: facilita la coesescenza (si raggiunge rapidamente il predecessore) e le operazioni di rimozione/inserimento durante le ricerche.

### 3.5.4 Politiche di posizionamento (scelta del buco)

Per soddisfare una nuova richiesta si ricerca nella struttura il “miglior” buco secondo una politica.

**First fit** Si scansiona dall'inizio e ci si ferma al *primo* buco di taglia sufficiente. Veloce, ma tende ad addensare le allocazioni nelle zone basse e a produrre piccoli residui liberi.

**Next fit** Come first fit, ma la scansione successiva riparte da dove si era interrotta l'ultima volta. Distribuisce leggermente meglio i residui, mantiene complessità bassa.

**Best fit** Si sceglie il buco *più piccolo* tra quelli sufficienti (idealmente uno della taglia esatta). Riduce lo spreco immediato ma, nel tempo, genera molti *micro-buchi* inutilizzabili (peggiora la frammentazione esterna). Richiede in genere la scansione completa.

**Worst fit** Si sceglie il buco *più grande* così che il residuo resti abbastanza ampio da essere riutilizzabile. Anche qui spesso serve la scansione completa.

### 3.5.5 Ottimizzazioni pratiche

- **Due liste separate:** mantenere una lista delle *partizioni occupate* e una dei *buchi* riduce le scansioni per first/next fit (si guardano solo i liberi).
- **Ordinamento per taglia dei buchi:** se la free-list è ordinata per dimensione, *best fit* può prendere la testa e *worst fit* la coda (se mantenute strutture adeguate), riducendo i costi di ricerca; aumenta però il costo di inserimenti/coalescenze.

In tutte le varianti, le politiche first/next fit sono più snelle (meno confronti), mentre best/worst richiedono in genere più lavoro per trovare il candidato ideale; la coalescenza è essenziale per mitigare la **frammentazione esterna** che inevitabilmente si accumula con allocazioni e deallocazioni ripetute.

## 3.6 Memoria virtuale

La *memoria virtuale* è un'astrazione con cui ogni processo vede uno *spazio di indirizzi* proprio, continuo e protetto, indipendente dalla posizione dei dati nella RAM. Come per la *virtual CPU* ottenuta via time-sharing, qui la separazione è ottenuta cooperando tra sistema operativo e hardware (MMU, *Memory Management Unit*), che traducono in tempo reale gli indirizzi generati dal programma in indirizzi fisici.

Uno spazio di indirizzi virtuale ha ampiezza definita dalla larghezza degli indirizzi della CPU (ad es. in un sistema a 32 bit si hanno fino a  $2^{32}$  indirizzi; nei sistemi a 64 bit lo spazio è molto più ampio, pur se spesso non interamente utilizzato). Questo spazio viene suddiviso in porzioni di *uguale dimensione* chiamate **pagine** ( $\approx$  qualche KiB o più), identificate da un *numero di pagina* e da un *offset* interno. Anche la memoria fisica è suddivisa in blocchi della stessa dimensione, detti **frame**. Qualunque pagina virtuale può essere collocata in qualunque frame: non c'è più vincolo di contiguità fisica e la **frammentazione esterna** scompare.

La corrispondenza pagina virtuale → frame fisico è memorizzata in una **tavella delle pagine** (per processo), con bit di protezione (lettura/scrittura/esecuzione), presenza/assenza, riferimento, modifica, ecc. Per accelerare le traduzioni ripetute l'hardware usa una cache di traduzioni (TLB). Quando il processo accede a una pagina *non presente* in RAM si genera un **page fault** (un'eccezione gestita, non un errore fatale): il kernel seleziona un frame libero (o sceglie una pagina vittima da sostituire), carica dal disco la pagina richiesta nell'area di *swap/file* di backing, aggiorna la tabella delle pagine e riavvia l'istruzione che aveva causato il fault. A differenza dello *swapping* tradizionale (che spostava l'intero

processo), qui si spostano solo le *paginae* necessarie: il processo resta eseguibile finché le pagine che tocca sono residenti.

La validità di questa tecnica poggia sulla **località** dell'accesso alla memoria (spaziale e temporale): in ogni fase un processo usa un sottoinsieme relativamente piccolo delle proprie pagine (*working set*), quindi mantenerne in RAM solo una frazione è sufficiente nella pratica. L'eliminazione della contiguità fisica e il *caricamento su richiesta* (*demand paging*) aumentano il grado di multiprogrammazione: a parità di RAM, più processi possono avanzare, tenendo occupate CPU e sottosistemi di I/O. Va però gestito il rischio di *thrashing* qualora le pagine richieste cambino troppo rapidamente e il sistema passi più tempo a sostituirle che a eseguirle.

La protezione è intrinseca al modello: spazi di indirizzi diversi sono isolati e le autorizzazioni sono per-pagina. Due processi possono usare lo stesso indirizzo virtuale (p. es. 0x4000\_0000) ma riferirsi a frame fisici differenti; la MMU garantisce che un processo non possa leggere o scrivere pagine dell'altro. Resta possibile una minima **frammentazione interna** (fino a una pagina per intervallo allocato) perché la pagina è l'unità di gestione; la scelta della dimensione di pagina bilancia dimensione delle tabelle, hit del TLB e spreco interno.

In sintesi, la memoria virtuale disaccoppia lo spazio di indirizzi visto dai programmi dalla RAM fisica, offrendo rilocazione trasparente, protezione forte e caricamento su richiesta delle sole porzioni realmente utilizzate.

### 3.6.1 Paginazione

La **paginazione** associa le pagine dello spazio di indirizzi *virtuale* di un processo ai *frame* della memoria *fisica* tramite una **tabella delle pagine** (*page table*) mantenuta per ogni processo. Il numero di voci della tabella dipende dalla dimensione dello spazio virtuale del processo e dalla dimensione di pagina (uguale per tutti i processi).

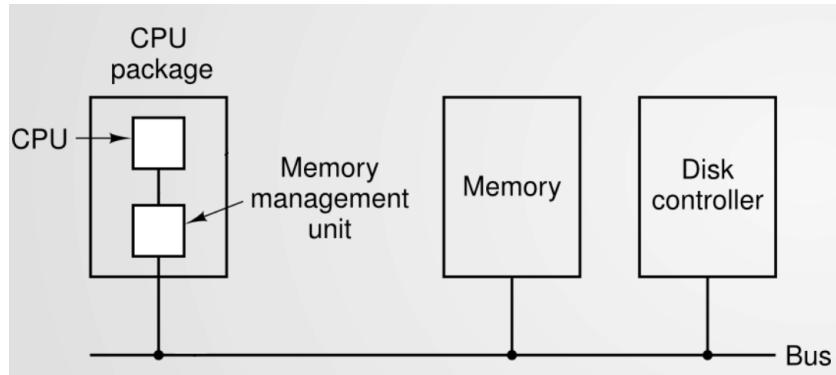


Figura 3.6: Processo di paginazione

#### 3.6.1.1 Traduzione indirizzi (MMU)

Quando la CPU genera un indirizzo virtuale  $V$ , la MMU lo divide in:

$$\text{numero di pagina} = \left\lfloor \frac{V}{\text{pagesize}} \right\rfloor, \quad \text{offset} = V \bmod \text{pagesize}.$$

Con  $V=8196$  e  $\text{pagesize}=4096$ : pagina = 2, offset = 4. La MMU consulta la PTE (Page Table della pagina):

1. se **non presente** (0)  $\Rightarrow$  *page fault*: il kernel carica la pagina dal backing store in un frame libero (o sostituisce una pagina vittima), aggiorna la PTE e ripete l'istruzione;
2. se **presente** (1) e con **permessi** adeguati (lettura/scrittura/esecuzione)  $\Rightarrow$  ottiene il *numero di frame*  $F$  e calcola l'indirizzo fisico  $P = F \cdot \text{pagesize} + \text{offset}$ . Nell'esempio, se la PTE della pagina 2 indica  $F=6$ , allora  $P = 6 \cdot 4096 + 4 = 24580$ .

Operativamente, molte architetture memorizzano nelle PTE anche bit di *accesso* ( $A$ ) e *modifica* ( $D$ ), che la MMU può impostare automaticamente per supportare il rimpiazzo e l'ottimizzazione.

### 3.6.1.2 Funzionamento

Il **sistema operativo** crea/aggiorna le tabelle (mappature, permessi, gestione dei fault); la **MMU** esegue la traduzione in hardware ad ogni accesso e interagisce con una cache di traduzioni (TLB) per evitare che le consultazioni della tabella in RAM diventino un collo di bottiglia. Un accesso con pagina non presente genera un *page fault*; un accesso con permessi violati genera una *protezione* (fault di protezione).

### 3.6.1.3 Esempio di mappatura.

Supponiamo uno spazio virtuale di 64 KiB e pagine da 4 KiB. Lo spazio è diviso in 16 pagine, numerate da 0 a 15; la pagina 0 copre gli indirizzi virtuali [0, 4095], la pagina 1 [4096, 8191], e così via. Anche la RAM è suddivisa in frame da 4 KiB, numerati con un *numero di frame*. Ogni voce di tabella (PTE) indica se la pagina è presente in RAM (bit di *presenza*) e, se sì, in quale frame è collocata; altrimenti la voce porta *presenza*=0 (e, a seconda dell'architettura, metadati per il recupero da disco).

### 3.6.2 Uso di una tabella di pagine

Nel modello a memoria virtuale la MMU vede gli indirizzi come *parole di bit*. Se le dimensioni sono potenze di due, la traduzione si riduce a semplici *shift* e *mask*: l'indirizzo virtuale  $V$  si scomponete in  $\langle$ numero di pagina, offset $\rangle$  dove l'offset ha  $n = \log_2(\text{pagesize})$  bit e il numero di pagina occupa i rimanenti  $m - n$  bit (con  $2^m$  indirizzi virtuali totali). L'indirizzo fisico ha  $t$  bit; i suoi  $t - n$  bit più significativi identificano il *frame*, gli ultimi  $n$  sono copiati come offset. In pratica:

$$\begin{aligned} \text{pagina} &= \left\lfloor \frac{V}{2^n} \right\rfloor \quad (\text{shift a destra di } n), \\ \text{offset} &= V \bmod 2^n \quad (\text{mask sugli } n \text{ bit meno significativi}), \\ \text{PA} &= (\text{frame} \ll n) \mid \text{offset}. \end{aligned}$$

La tabella delle pagine del processo (una voce per pagina virtuale) fornisce, per ogni pagina, il numero di frame e i bit di controllo: *presenza* (pagina residente), permessi, *referenziamento* e *dirty*. Se la voce ha presenza = 0 la MMU genera un *page fault*; il kernel carica la pagina in un frame e aggiorna la voce, quindi l'istruzione viene rieseguita.

**Generalizzazione dei bit.** Con spazio virtuale  $2^m$  e pagina  $2^n$ :

- numero di pagine =  $2^{m-n} \Rightarrow$  campo “numero di pagina” lungo  $m - n$  bit;
- offset lungo  $n$  bit;
- con spazio fisico  $2^t \Rightarrow$  numero di frame =  $2^{t-n}$  e campo “frame” lungo  $t - n$  bit.

### Esempio

**Parametri.** Spazio di indirizzi *virtuale* del processo:  $2^{32}$  (indirizzi a 32 bit,  $m = 32$ ). Dimensione di pagina:  $4 \text{ KiB} = 2^{12}$  ( $n = 12$ ). Memoria fisica:  $128 \text{ MiB} = 2^{27}$  ( $t = 27$ ).

**Grandezze derivate.** Numero di pagine per processo =  $2^{m-n} = 2^{20}$ . Numero di frame fisici =  $2^{t-n} = 2^{15} = 32768$ . Traduzione in hardware:

$$\text{pagina } p = V \gg n, \quad \text{offset } o = V \& (2^n - 1), \quad \text{PA} = (f \ll n) | o.$$

**Estratto della tabella delle pagine (PTE).** Valori esemplificativi per tre pagine:

Pagina $p$	Pres.	Frame $f$
0x01234	1	0x003A
0x01235	0	—
0x0ABC0	1	0x1F20

**A) Traduzione senza fault.** Sia  $V = 0x01234FED$ . Allora  $p = 0x01234$ ,  $o = 0x0FED$ . Dalla PTE:  $f = 0x003A$  (presenza= 1). Indirizzo fisico:

$$\text{PA} = (0x003A \ll 12) | 0x0FED = 0x003A000 | 0x0FED = \boxed{0x003AFED}.$$

**B) Page fault e ritentativo.** Sia  $V' = 0x01235004$ .  $p' = 0x01235$ ,  $o' = 0x0004$ . La PTE ha presenza = 0  $\Rightarrow$  *page fault*. Il kernel sceglie un frame libero, ad es.  $f' = 0x0050$ , carica la pagina, aggiorna la PTE (presenza = 1,  $f' = 0x0050$ ) e l'istruzione viene ripetuta. Ora:

$$\text{PA}' = (0x0050 \ll 12) | 0x0004 = 0x0050000 | 0x0004 = \boxed{0x0050004}.$$

Questo esempio riassume: (i) scomposizione  $\langle p, o \rangle$ , (ii) consultazione della PTE con bit di presenza, (iii) calcolo dell'indirizzo fisico e (iv) gestione del *page fault* con aggiornamento della PTE.

### 3.6.3 Dettagli di una voce della tabella delle pagine (PTE)

Quando si verifica un *page fault* la MMU non trova in RAM la pagina richiesta: l'eccezione passa il controllo al kernel, che recupera la pagina dal *backing store* (area di swap o file), la colloca in un frame libero (o ne sostituisce uno) e aggiorna la voce corrispondente nella **page table**. La tabella delle pagine è quindi usata sia dall'hardware (MMU) sia dal software (kernel): il SO decide le mappature e i permessi, mentre la MMU esegue le traduzioni e aggiorna alcuni flag.

Ogni **Page Table Entry (PTE)** contiene tipicamente i campi seguenti:

- **Numero di frame (page frame number):** identifica il frame fisico in cui risiede la pagina quando è presente in RAM.
- **Bit presente/assente (present):** vale 1 se la pagina è residente; vale 0 se è su disco. Se è 0, un accesso genera un *page fault*.
- **Protezione (permessi R/W/E):** due bit per lettura/scrittura e, spesso, un terzo per esecuzione. Le pagine di *codice* sono in genere di sola lettura e con esecuzione abilitata; disabilitare l'esecuzione su heap/stack (bit NX) aiuta a prevenire esecuzioni indesiderate (*code injection*).

- **Bit modificato (dirty)**: impostato dalla MMU quando la pagina è stata scritta. Se la pagina viene sostituita e il bit è 1, il kernel deve prima risincronizzarla su disco; se è 0 (*clean*) può essere scartata senza I/O.
- **Bit referenziato (accessed/referenced)**: posto a 1 dalla MMU al primo accesso; il SO può azzerarlo periodicamente per stimare l'uso recente e supportare gli algoritmi di rimpiazzo.
- **Bit di disabilitazione cache (caching disabled)**: per pagine mappate su dispositivi o aree di I/O memoria-mappato si può chiedere alla CPU di non usare la cache, evitando incongruenze con l'hardware.
- **Bit di validità/allocazione**: marca le regioni dell'indirizzo virtuale che il processo *può* usare (es. tra heap e stack). Un accesso a pagine non valide causa un'eccezione di protezione (es. *segmentation fault*).

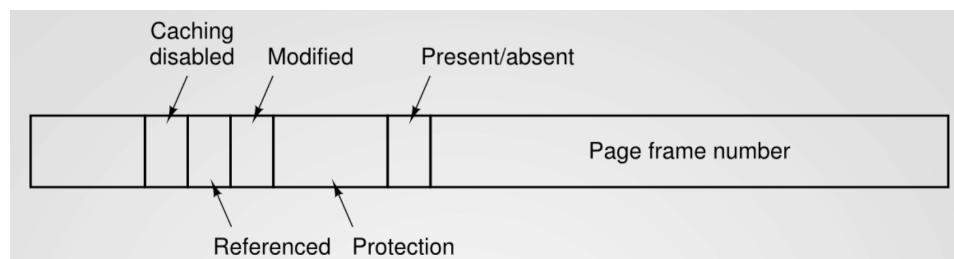


Figura 3.7: Schema di una PTE: campi di controllo (presenza, protezione, referenced, dirty, caching) e numero di frame fisico.

#### 3.6.4 Tabella dei frame

La tabella dei frame ci dà una mappatura dei frame e ovviamente questa tabella è unica, con un record per ciascun frame. La sua dimensione è proporzionale alla quantità di RAM disponibile, ogni record contiene informazioni sullo stato del frame che può essere libero o occupato e, se occupato, viene indicato l'ID del processo che lo occupa. Non viene indicato quale pagina è contenuta dentro il frame e si può occupare qualunque frame libero, indipendentemente dal processo che lo richiede, risolvendone così la frammentazione esterna. La tabella dei frame viene consultata quando una pagina deve essere inserita in memoria, per esempio nel caso di un page fault.

#### 3.6.5 Progettazione di una tabella delle pagine

Nel progettare un sistema a paginazione emergono due esigenze fondamentali: (i) la traduzione da indirizzo virtuale a fisico deve essere molto rapida; (ii) lo spazio occupato dalle tabelle deve restare gestibile anche quando lo spazio virtuale per processo è grande. Il primo aspetto dipende soprattutto da **dove** risiede la tabella delle pagine.

**Tabelle nei registri della MMU.** Nei sistemi storici, con poche pagine per processo, la MMU esponeva un set di registri sufficiente a contenere *tutta* la tabella. In questo modo la consultazione era istantanea (accesso a registri, non a RAM) e quindi la traduzione risultava velocissima. Il difetto era la scarsa *scalabilità*: ad ogni *context switch* il kernel doveva riprogrammare tutti i registri della MMU con la tabella del nuovo processo; all'aumentare del numero di pagine (e dei processi) questa soluzione diventa impraticabile.

**Tabelle in RAM con registro base (PTBR).** L’approccio moderno prevede che ogni processo abbia la propria tabella *in RAM*, tipicamente in un’area del kernel, allocata *contiguamente*. La MMU espone un *Page-Table Base Register* (PTBR) che punta alla base fisica della tabella del processo corrente. Il cambio di processo richiede quindi di aggiornare *solo* il PTBR: la soluzione è scalabile e semplice da gestire. Il costo è nei tempi di accesso: per risolvere un indirizzo servono *due* riferimenti in memoria (uno per leggere la PTE dalla RAM, uno per accedere alla parola/byte richiesto), riducendo di fatto il throughput degli accessi. In pratica questo svantaggio viene poi mitigato dall’uso di cache dedicate alla traduzione (TLB), trattate separatamente.

In sintesi, memorizzare la tabella nei registri è velocissimo ma non scala; mantenerla in RAM con PTBR è scalabile e gestibile nei context switch, al prezzo di un ulteriore accesso alla memoria per ogni traduzione.

### 3.6.6 TLB (memoria associativa)

La **Translation Lookaside Buffer** è una piccola cache posta tra CPU/MMU e tabella delle pagine, specializzata nel memorizzare *alcune* voci di traduzione (pagina virtuale → frame fisico). Poiché contiene pochi ingressi ma consultabili in parallelo (*associatività*), riduce il numero di accessi alla RAM necessari per tradurre gli indirizzi.

**Contenuto di una voce TLB.** Ogni voce include tipicamente: numero di *pagina virtuale*, numero di *frame fisico*, bit di *validità*, bit di *protezione* (R/W e, talvolta, X), e un bit *dirty* (M) che indica se la pagina è stata scritta. Il bit di referenziamento non è necessario: se la voce è in TLB è per definizione stata usata di recente.

**Ricerca e traduzione.** Dato un indirizzo virtuale  $V$ , la MMU estrae il numero di pagina  $p$  e l’offset  $d$ . *In parallelo* confronta  $p$  con tutte le chiavi presenti in TLB:

- **TLB hit:** se trova una voce valida e i permessi non sono violati, preleva il frame  $f$  dalla TLB e produce l’indirizzo fisico  $(f, d)$ .
- **TLB miss:** se la voce non è in TLB, consulta la *page table* in RAM. L’indirizzo della PTE si ottiene da

$$\text{PTE}(p) = \text{PTBR} + p \times \text{sizeof(PTE)}.$$

Recuperato  $f$ , la MMU (o il kernel, a seconda dell’architettura) *inserisce* la traduzione in TLB *sfrattando* una voce esistente; il prossimo accesso avrà con alta probabilità un *hit*.

Se i permessi non consentono l’operazione (es. scrittura su pagina di sola lettura), viene generata un’eccezione di *protezione*.

**Rimpiazzo e località.** La politica di rimpiazzo può essere implementata *in hardware* (spesso pseudo-LRU) o *in software* su alcune architetture; concettualmente si usa una variante di LRU per trattenere le voci usate di recente. La TLB è efficace per lo stesso motivo della cache: la *località* d’accesso fa sì che poche traduzioni vengano riutilizzate di frequente.

**Context switch, flush e ASID.** Ogni processo ha il proprio spazio di indirizzi: al cambio di processo il kernel cambia il *PTBR*. Le traduzioni in TLB del processo precedente, se non identificate, diventerebbero ambigue; per questo si esegue un *flush* della TLB, con un calo temporaneo di prestazioni (più miss).

Per evitarlo molte TLB supportano un identificatore di spazio di indirizzi, **ASID/PCID**, che etichetta le voci: in tal modo voci appartenenti a processi diversi possono coesistere senza conflitti.

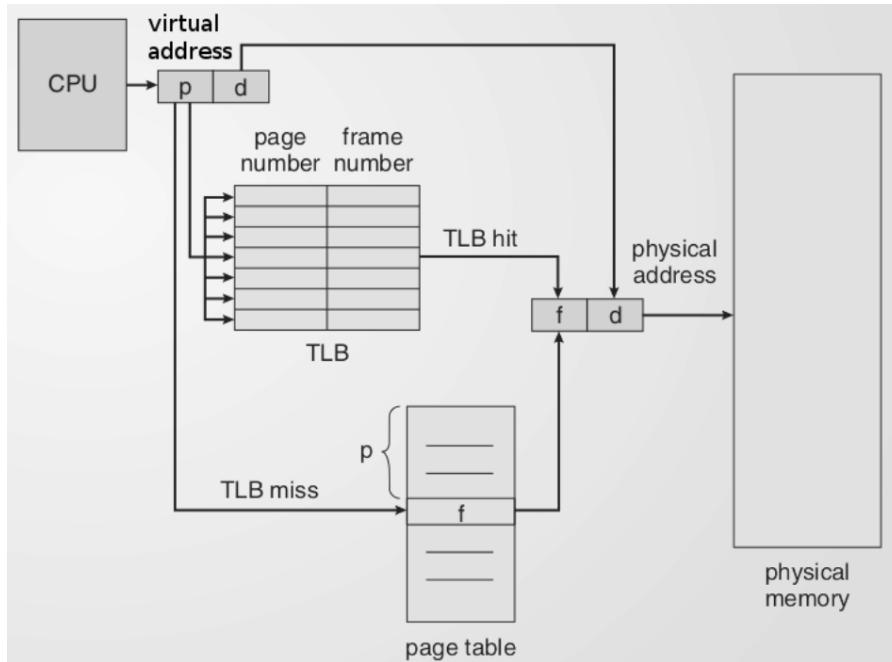


Figura 3.8: Funzionamento della TLB: in caso di *hit* si ottiene direttamente il frame *f*; in caso di *miss* si consulta la page table (indirizzata via PTBR) e si aggiorna la TLB.

### 3.6.7 EAT: Effective Access Time

#### 3.6.7.1 Definizione

L'**Effective Access Time (EAT)** è il tempo *medio* per soddisfare un accesso alla memoria quando l'indirizzo virtuale deve essere tradotto in indirizzo fisico. L'EAT incorpora sia il *lookup* nella TLB sia gli eventuali accessi alla *page table* in RAM effettuati dalla MMU durante il *page walk*.

#### 3.6.7.2 Calcolo a 1 livello

Indichiamo con  $\alpha$  il tempo di accesso alla RAM, con  $\beta$  il tempo di accesso alla TLB e con  $\varepsilon$  la probabilità di *TLB hit*.

- **TLB hit:** si paga il lookup TLB più *un* accesso alla RAM ( $\beta + \alpha$ ).
- **TLB miss:** si paga il lookup TLB, *un* accesso alla page table (PTE) e *un* accesso al dato ( $\beta + 2\alpha$ ).

Per il valore atteso (considerando  $\epsilon$  la probabilità che avvenga una hit nella TLB):

$$\text{EAT}_{k=1} = \varepsilon(\beta + \alpha) + (1 - \varepsilon)(\beta + 2\alpha)$$

Senza TLB, ogni accesso richiede sempre due accessi in RAM:  $\text{EAT}_{\text{noTLB}} = 2\alpha$ .

#### 3.6.7.3 Tabella delle pagine multilivello

Con spazi virtuali grandi (es. 64 bit) una tabella piatta sarebbe enorme: con pagine da 4 KiB avremmo  $2^{52}$  PTE per processo. La soluzione è una **page table gerarchica (multilivello)**: l'indirizzo virtuale è

scomposto in indici di livello ( $PT_1, PT_2, \dots$ ) e *offset*. La voce del livello  $i$  punta alla tabella del livello  $i+1$ ; l'ultimo livello contiene la PTE con numero di frame e permessi. Vantaggi: si alloca memoria per le sole sottotabelle effettivamente usate (buona *sparsity* e scalabilità). Svantaggio: su *TLB miss* il *page walk* attraversa tutti i livelli prima di poter accedere al dato, aumentando il numero di accessi a RAM (mitigato da TLB, page-walk cache e, ove possibile, huge pages).

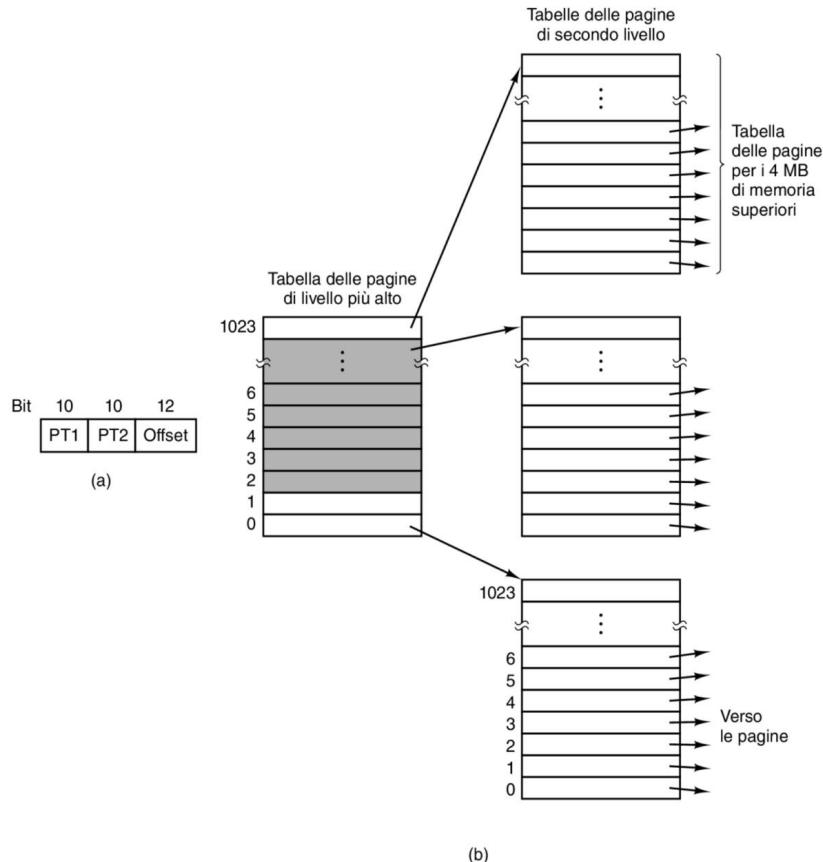


Figura 3.9: (a) Un indirizzo a 32 bit con due campi per le tabelle delle pagine. (b) Tabelle delle pagine a due livelli.

#### 3.6.7.4 Calcolo a $k$ livelli

Se la page table ha  $k$  livelli, un *miss* richiede  $k$  accessi per il *page walk* + 1 accesso al dato. Il costo elementare diventa:

$$t_{\text{hit}} = \beta + \alpha, \quad t_{\text{miss}} = \beta + (k + 1)\alpha.$$

Quindi (sempre considerando  $\epsilon$  come la probabilità che avvenga una hit nella TLB):

$$\boxed{\text{EAT} = \epsilon(\beta + \alpha) + (1 - \epsilon)(\beta + (k + 1)\alpha)}$$

Senza TLB:  $\text{EAT}_{\text{noTLB}} = (k + 1)\alpha$ .

#### 3.6.7.5 Alcune accortezze

- L'EAT resta vicino al tempo della RAM se  $\beta$  è piccolo e l'hit ratio  $\epsilon$  è alto. La penalità cresce *linearmente* con il numero di livelli  $k$  in caso di miss.

- Per un limite di overhead relativo  $\rho$  rispetto ad  $\alpha$ , la soglia sull'hit ratio è

$$\varepsilon \geq 1 - \frac{\rho\alpha - \beta}{k\alpha},$$

da cui si vede che se  $\beta > \rho\alpha$  l'obiettivo è impossibile anche con  $\varepsilon \approx 1$ .

- La progettazione pratica punta a: TLB capiente/efficiente (ASID/PCID), uso di pagine grandi per regioni estese, e page-table gerarchiche per ridurre la memoria impegnata dalle PTE, mantenendo però elevata la probabilità di TLB hit.

### 3.6.7.6 Tabella delle pagine invertita

Per ridurre drasticamente lo spazio occupato dalle strutture di traduzione si può usare una **tabella delle pagine invertita** (IPT). Invece di mantenere *una page table per processo* con una voce per ogni pagina virtuale, si mantiene *una sola tabella di sistema* con una voce per *ogni frame fisico*. Ogni voce descrive *chi* sta occupando quel frame.

**Idea di base.** La  $i$ -esima voce dell'IPT corrisponde al frame fisico  $i$  e contiene almeno:

- l'identità del proprietario (*PID*);
- il numero di pagina virtuale (*VPN*) di quel processo che è mappata nel frame;
- i bit di stato utili al rimpiazzo (es. *referenced/dirty*).

Il *bit di presenza* e il *numero di frame* sono impliciti: se la voce esiste, quel frame è occupato; l'indice di tabella è il frame. Questa impostazione rimpiazza tutte le page table per-processo con una sola struttura di dimensione  $O(\#frame)$ .

**Ricerca: hashing.** La traduzione diretta  $\langle PID, VPN \rangle \rightarrow$  frame non è più un accesso indicizzato come nelle page table classiche, bensì una ricerca. Per renderla efficiente si usa una **tabella hash** indicizzata da  $\text{hash}(PID, VPN)$ ; ciascun *bucket* punta a una lista di voci IPT che collidono. Su *TLB miss* la MMU (o il kernel) calcola l'hash, scorre la lista e, se trova la coppia (PID, VPN), ottiene il frame. Con una buona funzione di hash, il tempo atteso è  $O(1)$  (peggiora a  $O(L)$  con liste più lunghe).

### Vantaggi e limiti.

- **Pro:** memoria occupata proporzionale ai frame fisici (scalabile su spazi virtuali enormi); una sola struttura sostituisce le molte page table dei processi; resta compatibile con l'uso della **TLB** per evitare ricerche frequenti.
- **Contro:** la ricerca non è indicizzata per pagina, quindi senza TLB è più lenta delle page table classiche; alcune informazioni tipicamente *per-processo* (es. maschere di protezione R/W/X, informazioni per la localizzazione su disco ai fini del *page fault*) non sono naturalmente rappresentate nella sola IPT e richiedono strutture ausiliarie (metadati per regione o piccole tabelle d'appoggio per processo). Inoltre le pagine *condivise* tra processi vanno rappresentate con voci multiple nella struttura hash (una per ciascuna coppia (PID, VPN) che aliasa lo stesso frame), con gestione attenta delle collisioni.

### Flusso tipico (su TLB miss).

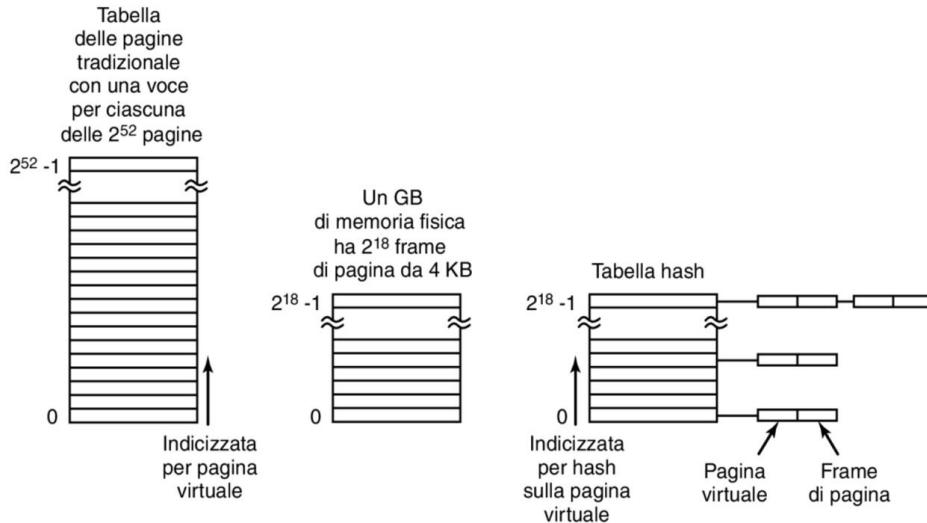


Figura 3.10: Confronto fra una tabella delle pagine tradizionale e una tabella invertita delle pagine.

1. Si calcola  $key = (\text{PID}, \text{VPN})$  e l'hash corrispondente.
2. Si percorre il bucket cercando  $key$ .
3. Se trovata, si ottiene il *frame* e si aggiorna la TLB; altrimenti si genera *page fault* e il kernel decide il caricamento da disco e l'aggiornamento della struttura.

In sintesi, la tabella invertita riduce il consumo di memoria delle strutture di traduzione a  $O(\#\text{frame})$ , ma sposta il problema sull'efficienza della ricerca (risolta con hashing) e sulla necessità di mantenere, accanto all'IPT, i metadati per-processo che non si deducono direttamente dal solo mapping  $(\text{PID}, \text{VPN}) \rightarrow \text{frame}$ .

### 3.6.8 Cache della memoria vs. memoria virtuale

Dove collocare la cache rispetto alla MMU? In linea di principio sono possibili entrambe le soluzioni; la scelta incide su latenza, coerenza e gestione dei context switch.

**Cache dopo la MMU (cache fisica).** La CPU emette indirizzi virtuali; la MMU li traduce in indirizzi fisici e solo *dopo* si interrogano i livelli di cache. La cache è quindi *indicizzata/taggata fisicamente*. Vantaggi: nessuna ambiguità tra processi (gli indirizzi fisici sono globali), perciò non è necessario svuotare la cache al context switch; coerenza più semplice con DMA e dispositivi. Svantaggio principale: la cache deve *attendere* la traduzione, aggiungendo la latenza del TLB/page walk al cammino critico dell'accesso.

**Cache prima della MMU (cache virtuale).** Qui la cache lavora con indirizzi virtuali: un *hit* evita sia il page walk sia l'accesso alla DRAM, riducendo la latenza. Per contro emergono ambiguità: *omonimi* (stessa VA di processi diversi per dati diversi) e *sinonimi* (VA diverse che mappano lo stesso frame fisico, tipiche con regioni condivise o alias). La soluzione pratica include:

- taggare le linee con un *ASID/PCID* per distinguere i processi, così da evitare il flush della cache ad ogni context switch;

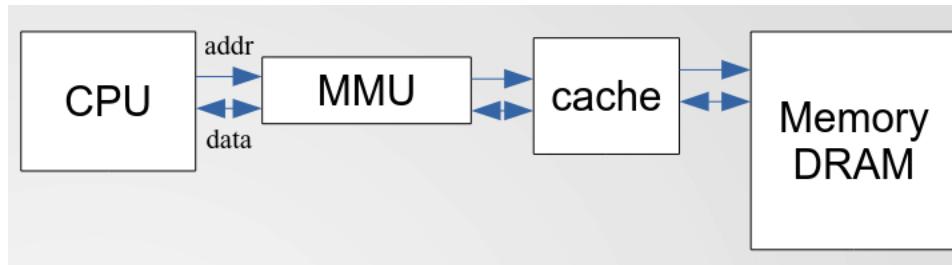


Figura 3.11: Organizzazione con cache fisica

- vincoli di indicizzazione/tag (o politiche di invalidazione selettiva) per controllare i sinonimi e mantenere la coerenza con l'indirizzo fisico.

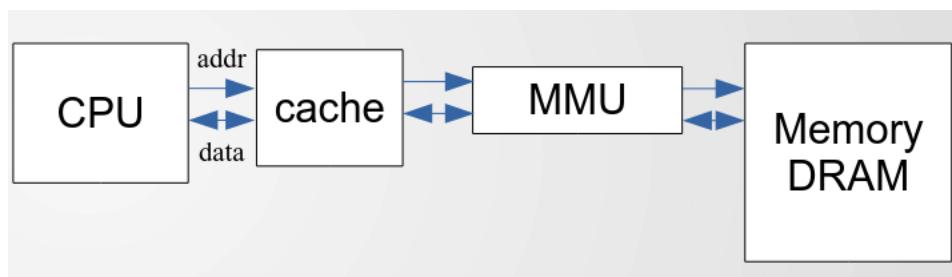


Figura 3.12: Organizzazione con cache virtuale

**Approccio ibrido (pratico).** I sistemi moderni adottano una combinazione:

- **L1** piccola e molto veloce, *virtualmente indicizzata e fisicamente taggata* (VIPT): l'indicizzazione può avvenire in parallelo al lookup TLB; quando la traduzione arriva, il tag fisico conferma l'hit. Con una dimensione della cache/linea adeguata si evitano i conflitti di sinonimia.
- **L2/L3** più capienti ma più lente, *fisicamente indicizzate/taggati* (post-MMU), così da semplificare coerenza e condivisione tra core.

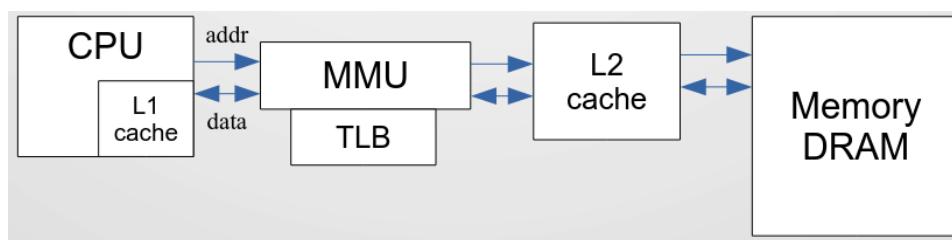


Figura 3.13: Organizzazione reale con cache di livelli 1 e 2 (L1, L2).

**Sintesi.** Mettere la cache *prima* della MMU riduce la latenza ma richiede ASID e contromisure ai sinonimi; metterla *dopo* semplifica coerenza e context switch al prezzo della latenza di traduzione. La soluzione VIPT in L1 con livelli successivi fisici combina i vantaggi di entrambi gli approcci.

### 3.7 Algoritmi di sostituzione delle pagine

Quando si verifica un *page fault* è l'hardware (MMU) a rilevarlo leggendo il bit di presenza. Viene generata un'eccezione e il controllo passa al kernel, che dovrà caricare la pagina dal backing store (swap o file) in un frame libero. Se tutti i frame sono occupati, occorre scegliere una *vittima* con un algoritmo di sostituzione.

Un riferimento teorico è l'algoritmo **OPT** (ottimo): rimuove la pagina che verrà utilizzata nel futuro più lontano. È irrealizzabile in pratica (richiederebbe conoscenza del futuro), ma fornisce un *lower bound* su cui confrontare gli algoritmi reali.

#### 3.7.1 Not Recently Used (NRU)

NRU usa informazioni a breve termine fornite dai bit **R** (referenziamento) e **M** (modifica, *dirty*). Il sistema operativo azzera periodicamente tutti i bit *R* (la MMU li riporterà a 1 al primo accesso), così da distinguere le pagine *recentemente usate* da quelle no. L'idea è evitare di rimuovere ciò che è stato usato di recente e, a parità di uso recente, preferire il rimpiazzo di pagine *non modificate* (così si evita l'I/O di scrittura).

Ogni pagina è classificata in una delle seguenti quattro classi:

- **Classe 0:**  $R = 0, M = 0$  (non referenziata, non modificata);
- **Classe 1:**  $R = 0, M = 1$  (non referenziata, modificata);
- **Classe 2:**  $R = 1, M = 0$  (referenziata, non modificata);
- **Classe 3:**  $R = 1, M = 1$  (referenziata, modificata).

La scelta della vittima avviene cercando la classe *non vuota* con indice più basso ( $0 \rightarrow 3$ ). All'interno della classe selezionata si può scegliere una pagina, ad esempio, con una semplice politica **FIFO** (la più “vecchia” in RAM).

NRU è semplice ed economico, ma grossolano: distingue se una pagina è stata usata di recente, non quanto sia stata usata. Di conseguenza può rimuovere una pagina referenziata molto spesso ma più “vecchia” rispetto ad un'altra referenziata di rado, causando più fault rispetto ad approcci che stimano la frequenza/recenza in modo più fine.

#### 3.7.2 Algoritmo FIFO e “seconda chance”

Una politica elementare di rimpiazzo è **FIFO**: le pagine sono mantenute in una coda, e in caso di page fault la *vittima* è semplicemente la testa della coda, ossia la pagina residente da più tempo. Questo può però espellere pagine ancora molto utili (vecchie ma frequentemente referenziate), causando ulteriori fault.

La variante **seconda chance** (*Second-Chance*) conserva l'ordinamento FIFO ma sfrutta il bit di *referenziamento R* per evitare di rimuovere pagine usate di recente. La procedura è:

1. Seleziona la pagina in testa alla coda (candidata vittima).
2. Se  $R = 0$ : la pagina non è stata usata dall'ultimo azzeramento, quindi viene scelta come vittima ed espulsa.
3. Se  $R = 1$ : la pagina ha avuto un accesso di recente; si concede una “seconda chance” azzerando *R*, si sposta la pagina in fondo alla coda e si prosegue con la nuova testa.

In tal modo le pagine referenziate di recente “scivolano” verso la coda e non vengono rimosse subito; quelle poco usate, invece, rimangono in testa e vengono selezionate come vittime. Nel caso limite in cui tutte le pagine abbiano  $R = 1$ , l’algoritmo percorre l’intera coda azzerando i bit e termina espellendo la prima pagina incontrata (la più vecchia tra quelle che hanno ricevuto la seconda chance).

Dal punto di vista implementativo, la politica è spesso realizzata come **orologio** (*clock*): le pagine sono disposte in una lista circolare e un puntatore (*hand*) scorre le voci; quando trova  $R = 0$  sceglie la vittima, altrimenti azzerà  $R$  e avanza. La seconda chance è una semplice ed efficace approssimazione di LRU, con costo simile a FIFO e qualità sensibilmente migliore.

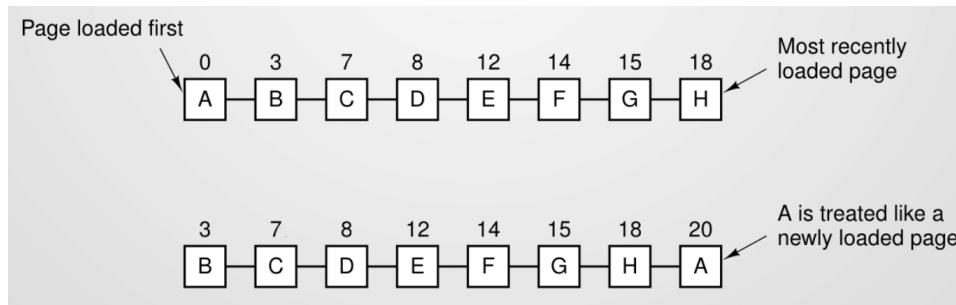


Figura 3.14: Second-Chance su coda FIFO. **Sopra:** stato iniziale della coda, ordinata per tempo di caricamento e  $A$  è la più vecchia. **Sotto:** applicazione della seconda chance alla testa  $A$  con  $R = 1$ : si azzerà  $R$  e  $A$  viene spostata in coda, assumendo un nuovo timestamp (20), ottenendo l’ordine  $B - C - D - E - F - G - H - A$ . La vittima sarà la prima testa incontrata con  $R = 0$ .

### 3.7.3 Clock (seconda chance con lista circolare)

L’algoritmo **Clock** realizza la politica della *seconda chance* in modo più efficiente. Tutti i frame residenti sono disposti in una *lista circolare*; un puntatore, detto *lancetta* (*hand*), individua la “testa” corrente. Ogni voce contiene almeno il bit di referenziamento  $R$  (ed eventualmente il bit di modifica  $M$ ).

**Procedura.** All’arrivo di un page fault si ripete:

1. si esamina la voce puntata dalla lancetta;
2. se  $R = 0$  la pagina non è stata usata di recente: viene scelta come *vittima* ed espulsa;
3. se  $R = 1$  si concede una *seconda chance*: si pone  $R \leftarrow 0$  e si avanza la lancetta al frame successivo, continuando la ricerca.

Nel caso limite in cui tutte le pagine abbiano  $R = 1$ , la lancetta compie un giro completo azzerando i bit; la prima pagina riesaminata con  $R = 0$  diventa la vittima.

**Vantaggi.** Clock mantiene il comportamento della seconda chance ma riduce il costo delle operazioni sulla coda: lo spostamento della “testa” in coda diventa un semplice *avanzamento* della lancetta, senza estrazioni/inserimenti. Il costo è ammortizzato, prossimo a FIFO nei casi semplici, con qualità di sostituzione nettamente migliore grazie all’uso del bit  $R$ .

### 3.7.4 Least Recently Used (LRU)

L’algoritmo **LRU** sceglie come vittima la pagina *meno recentemente usata*, cioè quella il cui ultimo riferimento è il più lontano nel tempo. A differenza di FIFO, LRU non guarda da quanto tempo la

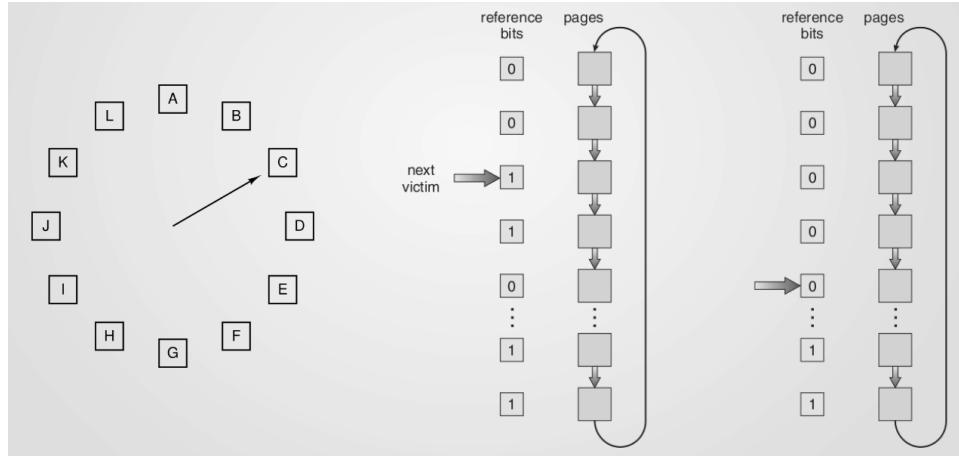


Figura 3.15: Esempio dell'algoritmo clock

pagina risiede in RAM (criterio poco informativo), ma usa la *recenza* dei riferimenti: pagine toccate di recente è probabile che vengano ritoccate presto; pagine non usate da molto probabilmente resteranno inutilizzate ancora per un po'.

**Meccanismo ideale.** Si assume un contatore globale (tipicamente a 64 bit) che si incrementa ad ogni riferimento di memoria. Quando una pagina  $p$  viene referenziata, la MMU scrive il valore corrente del contatore nel suo *timestamp* (campo nella voce di tabella/nel record del frame). Al momento del rimpiazzo, si scansionano i frame residenti e si sceglie la pagina con il timestamp *minimo* (ultimo uso più vecchio).

**Note implementative.** Aggiornare il timestamp *ad ogni accesso* richiede supporto hardware dedicato (MMU/TLB) per evitare un costo di store in kernel ad ogni riferimento. Per questo motivo LRU *puro* è costoso; i sistemi operativi pratici lo *approssimano* con politiche che ne catturano la recenza in modo più economico (p. es. Second-Chance/Clock, Aging).

### 3.7.5 Not Frequently Used (NFU)

Per evitare il costo di un LRU “puro”, si usa l’approssimazione **NFU** (*Not Frequently Used*), che privilegia le pagine più spesso utilizzate nel tempo, senza richiedere aggiornamenti ad ogni accesso.

**Idea.** Ad ogni frame residente è associato un contatore intero  $C$ . A intervalli regolari (timer del kernel), per *ogni* pagina si legge il bit di referenziamento  $R$  e si esegue:

$$C \leftarrow C + R, \quad R \leftarrow 0.$$

In questo modo  $C$  approssima quante volte la pagina è risultata “attiva” nei vari intervalli. Quando serve una vittima, si sceglie la pagina con il *contatore minimo* (in caso di parità, si può usare FIFO o qualunque criterio di tie-break).

**Proprietà e limiti.** NFU non conta i singoli riferimenti, ma il numero di *periodi* in cui si è osservato almeno un accesso; è semplice da implementare e non richiede supporto hardware oltre al bit  $R$ . Tuttavia pagine molto usate tempo indietro mantengono  $C$  elevato e tendono ad essere ingiustamente preservate. Per mitigare questo problema si introducono versioni a *decadimento* (p. es. *Aging*, che effettua shift e inserisce  $R$  come bit più significativo), rendendo la politica più sensibile alla recenza.

### 3.7.6 Aging

L’algoritmo di **Aging** approssima LRU mantenendo, per ogni frame residente, un contatore a  $k$  bit  $C$  che sintetizza *quanto di recente* la pagina è stata usata. A intervalli regolari (tick di timer) il kernel aggiorna tutti i contatori leggendo il bit di referenziamento  $R$  della pagina e applicando:

$$C \leftarrow (C \gg 1) \mid (R \ll (k - 1)), \quad R \leftarrow 0.$$

Cioè: si *shifta a destra* il contatore (l’evidenza più vecchia “scivola” verso i bit meno significativi e viene perduta oltre  $k$  tick) e si inserisce il valore corrente di  $R$  come *bit più significativo* (MSB). Una pagina referenziata nell’ultimo intervallo riceve un ‘1’ in testa; se non è stata usata, entra uno ‘0’. La *vittima* è la pagina con il *contatore più piccolo* (a parità si può usare FIFO come tie-break).

**Idea.** I bit più a sinistra pesano di più: un riferimento *recente* mantiene alto il valore di  $C$ ; riferimenti più lontani nel tempo “invecchiano” via via per effetto degli shift. Così Aging privilegia pagine usate spesso e di recente, penalizzando quelle fredde, avvicinandosi al comportamento LRU ma con costo molto più basso.

#### Proprietà pratiche.

- Non conta *quante* volte la pagina è stata usata dentro l’intervallo, ma solo *se* è stata usata (tramite  $R$ ).
- La *profondità temporale* è limitata da  $k$ : con un contatore a  $k$  bit si ricorda al massimo lo storico degli ultimi  $k$  tick.
- La scelta del periodo di aggiornamento determina la “granularità” temporale: troppo corto  $\Rightarrow$  overhead; troppo lungo  $\Rightarrow$  scarsa sensibilità.

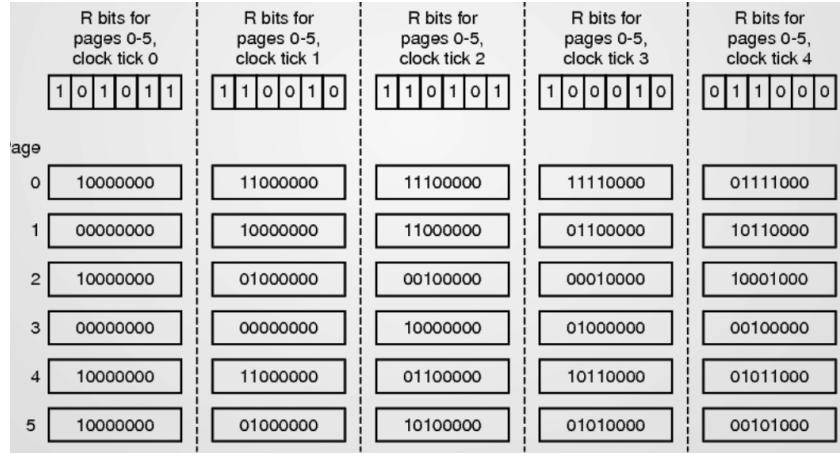


Figura 3.16: Evoluzione dell’algoritmo di *Aging*. In alto, per ciascun tick, i bit di referenziamento  $R$  delle pagine 0–5; per ogni pagina, il contatore a 8 bit è aggiornato come  $\text{counter} \leftarrow (\text{counter} \gg 1) | (R \ll 7)$ . I riferimenti recenti inseriscono 1 nel MSB e “valgono di più”; gli 1 più vecchi slittano a destra e si perdono oltre la profondità del contatore. La vittima è la pagina col contatore minimo (nell’ultimo tick, la pagina 3).

*Aging* è quindi una buona approssimazione software di LRU: semplice, scalabile e facile da implementare usando i bit  $R$  forniti dall’hardware.

### 3.7.7 Analisi delle prestazioni

Per confrontare gli algoritmi di rimpiazzo si misura il **numero di page fault** generati da una stessa sequenza di riferimenti, al variare del numero di frame fisici assegnati al processo. In generale, più frame  $\Rightarrow$  più pagine tenute in RAM  $\Rightarrow$  meno fault; tuttavia questa monotonia può *non* valere per tutte le politiche (anomalia di Belady, più sotto).

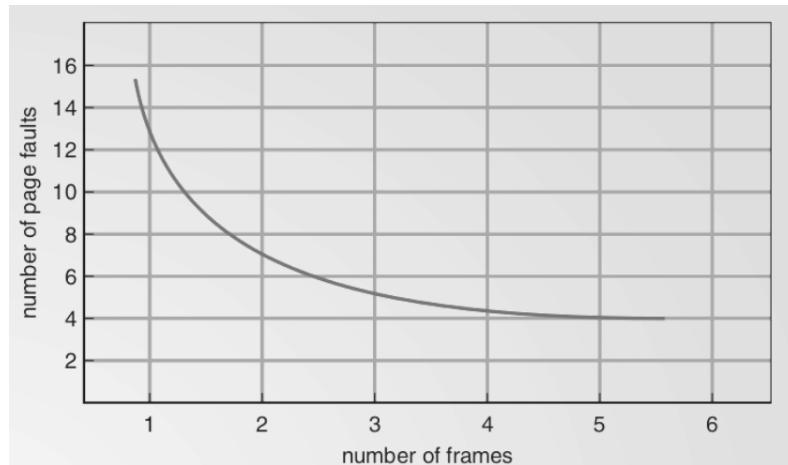


Figura 3.17: Andamento tipico: i page fault decrescono all’aumentare dei frame.

**Proprietà di inclusione (stack property).** Per uno stesso flusso di riferimenti e in ogni istante  $t$ , un algoritmo *stack* garantisce che l’insieme delle pagine residenti con  $n$  frame sia contenuto in quello con  $n+1$  frame:

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

Esempio intuitivo con una politica “più-recenti”: con 3 frame possono trovarsi in RAM le tre pagine più recenti, poniamo  $\{A, B, C\}$ ; con 4 frame, nello stesso istante le pagine residenti sono  $\{A, B, C, D\}$ . Il contenimento è evidente, quindi aumentare i frame *non può* aumentare i page fault.

**Sequenza di prova.** Consideriamo la sequenza (senza ripetizioni consecutive) e 3 frame disponibili.

$$\langle 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 \rangle$$

### 3.7.7.1 Ottimo (OPT)

OPT rimuove la pagina il cui uso è nel futuro più lontano. È irrealizzabile in pratica, ma fornisce il limite inferiore. Con la sequenza data si ottengono **9** fault (minimo possibile).

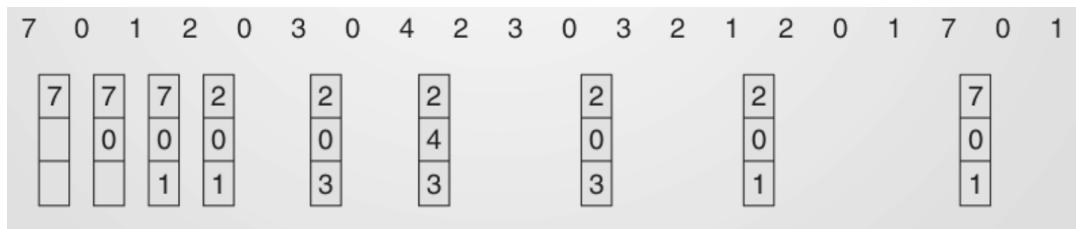


Figura 3.18: Simulazione con OPT: minimo teorico di 9 page fault.

### FIFO

FIFO espelle la pagina residente da più tempo. Sulla stessa sequenza produce **15** fault.

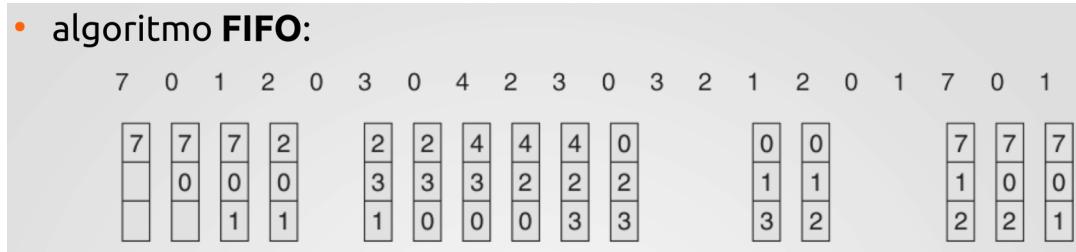


Figura 3.19: Simulazione con FIFO sulla sequenza: in totale 15 fault.

**Anomalia di Belady.** FIFO può peggiorare quando aumentano i frame: esistono sequenze (es.  $\langle 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 \rangle$ ) in cui passare da 3 a 4 frame *aumenta* i fault. È la **anomalia di Belady**: FIFO non gode della *proprietà di inclusione*.

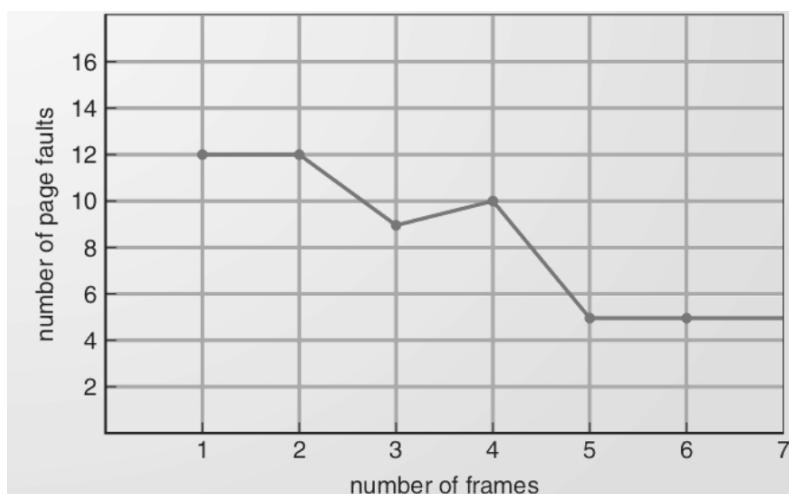


Figura 3.20: Esempio di anomalia di Belady per FIFO: più frame  $\Rightarrow$  più fault.

## LRU

LRU (Least Recently Used) espelle la pagina meno recente. Nella nostra sequenza produce **12** fault, intermedio tra OPT e FIFO, ed è *stabile*: soddisfa l'inclusione

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

dove  $B_t(n)$  è l'insieme delle  $n$  pagine in RAM all'istante  $t$ . Quindi aumentando i frame i fault non possono aumentare.

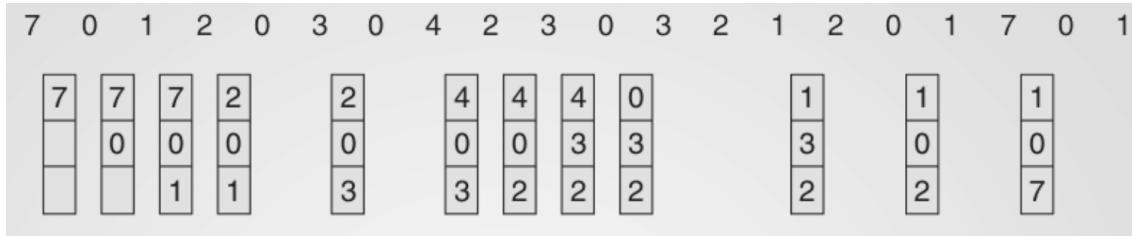


Figura 3.21: Simulazione LRU: 12 page fault; LRU è inclusiva (niente anomalia di Belady).

## Confronto e proprietà di inclusione.

- **OPT** è il riferimento teorico (non implementabile).
- **LRU** è una buona approssimazione pratica: è *inclusiva*, quindi all'aumentare dei frame i fault non crescono.
- **NFU** e **Aging** approssimano LRU (contatori/registri di età) e tendono a preservare l'inclusione.
- **Second-Chance** e **Clock** usano FIFO come base, mitigata dal bit  $R$ : in molti casi migliorano, ma non garantiscono inclusione e possono manifestare l'anomalia di Belady.
- **NRU** seleziona per classi  $(R, M)$  e spesso usa FIFO *intra-classe*: anch'esso può incorrere nell'anomalia di Belady.

**Nota implementativa (OS reali).** I kernel moderni adottano varianti *second-chance arricchite (Clock enhanced)*: si considerano i bit  $R$  e  $M$  (preferendo  $R=0$ , e tra questi  $M=0$  per evitare write-back), talvolta combinati con aging, prefetch e politiche distinte per pagine *file-backed* vs *anonymous*.

In sintesi: il numero di fault è la metrica chiave; OPT fornisce il limite inferiore, LRU è stabile e ben comportata; politiche FIFO-like possono violare la monotonia e indurre l'anomalia di Belady.

## 3.8 Allocazione dei frame

In un sistema paginato occorre decidere *quanti frame fisici* assegnare a ciascun processo e *quali pagine* tenere in memoria. La scelta incide direttamente sul numero di page fault e quindi sulle prestazioni globali del sistema.

### 3.8.1 Minimo strutturale di frame

Esiste una soglia di frame *minimi* per consentire l'esecuzione senza rimanere bloccati su una singola istruzione. Il valore dipende dall'**architettura** e dal **formato delle istruzioni**. Esempi tipici:

- Con un’istruzione semplice ADD reg, [mem] servono almeno le pagine di *codice* e *dato*  $\Rightarrow$  almeno 2 frame.
- Con due operandi in memoria ADD [m1], [m2]  $\Rightarrow$  almeno 3 frame (codice + due pagine dati).
- Con *indirizzamento indiretto* ogni operando può richiedere *due* pagine (pagina del puntatore + pagina del dato). Con due operandi indiretti: 5 frame (1 codice +  $2 \times 2$  dati).

Sotto il *minimo strutturale* si entra in un ciclo di fault (thrash su una singola istruzione). Restare appena sopra il minimo consente l’esecuzione ma con prestazioni pessime per via di molti page fault.

### 3.8.2 Strategie di caricamento iniziale

Una scelta semplice è la **paginazione pura su richiesta** (*pure demand paging*): all’avvio non si carica nulla; le pagine arrivano in RAM solo al primo uso (page fault). L’avvio è più lento (burst di fault), ma si evita di caricare pagine inutili. Strategie più evolute possono effettuare *precaricamenti* basati su statistiche (pagine di codice tipicamente usate all’avvio di applicazioni note).

### 3.8.3 Politiche di ripartizione dei frame tra processi

Sia  $N$  il numero totale di frame,  $m$  il numero di processi,  $s_i$  la taglia (in pagine) del processo  $i$  e  $S = \sum_i s_i$ .

**Allocazione equa:** ogni processo ottiene circa  $N/m$  frame, eventualmente rispettando un minimo strutturale  $f_i^{\min}$ . È semplice ma ignora la taglia: i processi piccoli possono ricevere frame in eccesso, quelli grandi troppo pochi.

**Allocazione proporzionale:** ogni processo riceve una quota proporzionale alla propria taglia, ad es.

$$A_i = \max(f_i^{\min}, \lfloor \frac{s_i}{S} N \rfloor).$$

In questo modo i processi più grandi hanno più frame e, in generale, meno fault.

Le scelte di ripartizione interagiscono con lo *scheduling*: un processo con più frame tende ad avanzare più velocemente (meno fault), e può meritare priorità maggiori in sistemi che bilanciano throughput e reattività.

### 3.8.4 Ambito del rimpiazzo: locale vs globale

Le politiche di sostituzione possono operare su insiemi diversi di pagine:

**Allocazione locale:** quando un processo genera un page fault, la vittima viene scelta *solo* tra le sue pagine residenti. La quota di frame per processo rimane stabile (buon isolamento, prevedibilità).

**Allocazione globale:** la vittima può appartenere a *qualsiasi* processo. Questo consente al sistema di *riallocare* dinamicamente i frame dove servono (adattivo), ma può degradare le prestazioni di processi altrimenti in equilibrio e rende più difficile il controllo della variabilità.

### 3.8.5 Trashing e controllo del grado di multiprogrammazione

Se un processo dispone di pochissimi frame (appena sopra il minimo strutturale), i suoi page fault restano elevati: il sistema fatica a mantenere in RAM le pagine “calde”. Con **allocazione globale** il

kernel può *prestare* frame a quel processo per stabilizzarlo; se questo non basta e *multi* processi sono in sofferenza, si va in **trashing di sistema** (fault rate elevato, CPU usata per gestire fault). La soluzione pratica è *ridurre il grado di multiprogrammazione* (sospendere/uccidere alcuni processi o applicare working-set/WSL trimming) finché il sistema rientra in una regione stabile.

### 3.8.6 Osservazioni conclusive

- Esiste un *minimo strutturale* di frame per processo, dettato dall'ISA.
- Ripartire i frame in modo *proporzionale* alla taglia evita squilibri tipici dell'allocazione equa e riduce i fault complessivi.
- La scelta *locale* garantisce isolamento; la *globale* offre adattività ma può introdurre rumore tra processi.
- Il sistema deve monitorare il fault rate e, se necessario, ridurre il livello di multiprogrammazione per evitare il trashing.

### 3.8.7 Modello di località

Il modello di località è un concetto legato all'esecuzione dei processi da parte della CPU: una **località** è un insieme di locazioni di memoria utili al processo in un dato momento. La località, ovviamente, cambia nel tempo, dato che le esigenze cambiano, così come il codice eseguito. Anche la dimensione della località non è prefissata.

Se la quantità di frame assegnati a un processo può contenere la località, allora il processo ha in RAM tutto ciò che gli serve, altrimenti si hanno dei page fault.

### 3.8.8 Working set

Il **working set** di un processo all'istante  $t$  è l'insieme delle pagine referenziate nelle ultime  $\Delta$  istruzioni (o negli ultimi  $\Delta$  tick di timer); è quindi una misura dinamica della *località*: la sua cardinalità varia nel tempo e rappresenta quante pagine sono realmente necessarie in quel periodo per lavorare senza eccesso di fault; in pratica il kernel lo approssima con un interrupt periodico che azzera/shifta il bit di referenziamento  $R$  aggiornando un contatore di *aging*: una pagina che ha  $R=1$  almeno una volta nella finestra  $\Delta$  viene considerata nel working set del processo, e questa informazione consente di assegnare al processo un numero di frame proporzionale al bisogno, prevenendo il *thrashing*.

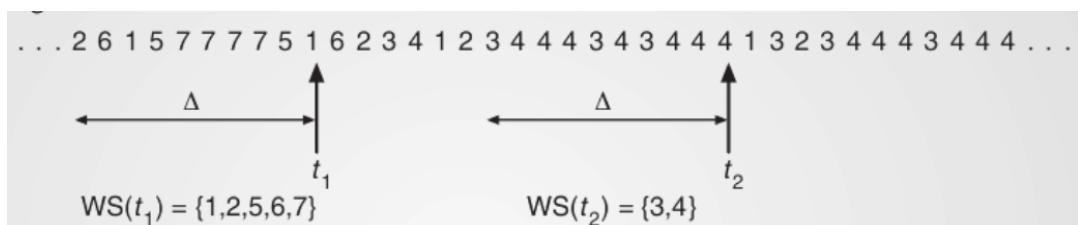


Figura 3.22: Esempio con finestra  $\Delta$ : a  $t_1$  il working set è  $\{1, 2, 5, 6, 7\}$ , a  $t_2$  è  $\{3, 4\}$ ; il set segue i cambi di località.

### 3.8.9 Page-Fault Frequency (PFF)

La **Page-Fault Frequency** misura la frequenza dei page fault di un processo e la confronta con due soglie: un *upper bound* (oltre il quale il processo è in sofferenza e conviene aumentare i frame) e un *lower bound* (sotto il quale è sovra-provisionato e si possono ritirare frame); la PFF è complementare al working set: i picchi compaiono ai cambi di località, poi dopo aver assegnato le nuove pagine la frequenza torna bassa.

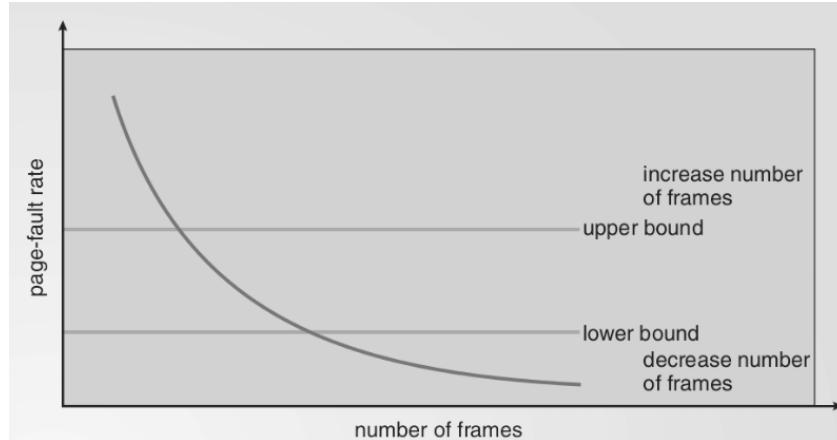


Figura 3.23: Controllo PFF con soglie: sopra l'upper bound si aumentano i frame, sotto il lower bound si riducono.

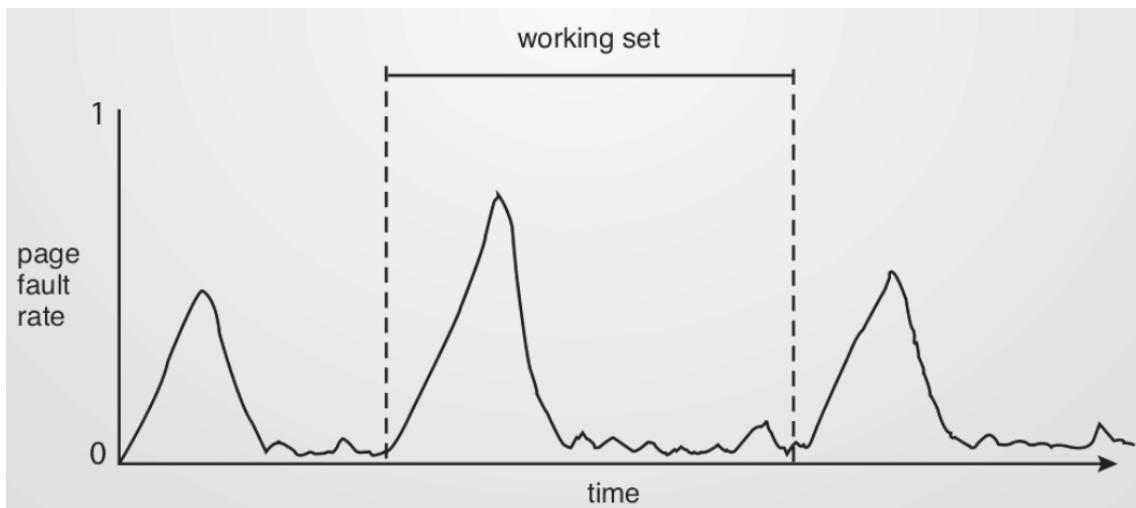


Figura 3.24: PFF nel tempo: picchi quando si entra in un nuovo working set; dopo l'adattamento dei frame la frequenza si normalizza.

#### 3.8.9.1 Politica di pulitura

Anche con buone politiche di rimpiazzo i *page fault* non scompaiono, perciò conviene ottimizzare la loro gestione: quando arriva un fault il kernel cerca anzitutto un frame libero; se non ce ne sono, invoca l'algoritmo di sostituzione e, qualora la vittima risulti *modificata* (dirty), deve effettuare anche il *write-back* su disco prima di riutilizzare il frame, motivo per cui si preferiscono pagine pulite. Per evitare di fare tutto questo lavoro nel percorso critico del fault, i sistemi reali usano un **paging daemon** che mantiene un *pool* di frame liberi: quando la riserva scende sotto una soglia, in background seleziona

pagine poco usate (politica globale) e le libera; se sono dirty, le scrive su disco in modo asincrono e poi le mette nel pool. Così, al prossimo fault, il kernel ha già un frame pronto e la latenza si riduce sensibilmente. Se la scelta di “sfratto” si rivelasse sfortunata (la pagina serve subito dopo), l’effetto è limitato: finché il frame non è stato riutilizzato da altri, il riaccesso si risolve come *minor/soft fault* senza I/O (la pagina è ancora in RAM nella lista dei liberi); solo quando il frame viene riassegnato ad altro processo si torna a un *major/hard fault* con trasferimento da disco.

### 3.8.9.2 Dimensione della pagina

La dimensione di pagina (e quindi di frame) è scelta dall’OS tra i valori supportati dall’hardware e tipicamente è una potenza di due, per rendere semplici ed efficienti gli shift nella traduzione indirizzi; in generale conviene che sia un multiplo della dimensione del blocco su disco per evitare letture/scritture parziali. **Pagine più grandi** portano a tabelle più piccole (meno voci da gestire), trasferimenti I/O più efficienti – specie su HDD, dove il costo di posizionamento domina e leggere/scrivere più dati contigui è vantaggioso –, e in media meno fault (più dati utili portati in RAM a ogni caricamento); inoltre aumentano la *TLB reach* (la quantità totale di RAM mappabile dalla TLB a parità di numero di voci). Di contro, aumentano la *frammentazione interna* (spazio non usato dentro la pagina) e rendono più grossolana la rappresentazione del *working set* (si è costretti a tenere in RAM un’intera pagina anche se serve una piccola porzione). **Pagine più piccole** hanno l’effetto opposto: riducono la frammentazione interna e permettono di approssimare meglio il *working set* (meno memoria sprecata), ma richiedono tabelle più grandi e possono aumentare TLB miss e fault; la scelta pratica è un compromesso tra questi fattori e tra vincoli hardware e carichi attesi.

## 3.9 Pagine condivise

Quando più istanze dello stesso programma sono in esecuzione, molte delle loro pagine *virtuali* contengono byte identici (tipicamente il codice eseguibile e le librerie). Replicarle in più frame fisici è uno spreco: l’OS può far sì che diverse page table puntino allo stesso frame fisico, ottenendo risparmio di RAM senza rinunciare all’isolamento per processo. L’ottimizzazione dev’essere *trasparente*: ciascun processo continua a vedere il proprio spazio di indirizzi, ma la traduzione MMU fa convergere più pagine virtuali sul medesimo frame.

### 3.9.1 Condivisione del codice rientrante

Il caso più semplice è il **codice rientrante** (pure code): non modifica se stesso né mantiene stato interno scrivendo nel segmento codice. L’OS può marcare tali pagine come **sola lettura** nelle PTE dei processi e farle puntare allo stesso frame. In figura, tre processi  $P_1, P_2, P_3$  condividono le pagine di codice, mentre i dati (diversi per processo) risiedono in frame distinti. Se una pagina condivisa venisse accidentalmente scritta, la protezione *R/W/X* genererebbe una trap: in molti sistemi si applica la tecnica *copy-on-write* (COW), che al primo tentativo di scrittura crea una copia privata della pagina e aggiorna la PTE del solo processo che scrive; gli altri continuano a condividere l’originale in sola lettura.

### 3.9.2 Memoria condivisa esplicita (IPC)

Oltre al codice, due (o più) processi possono chiedere al kernel di condividere intenzionalmente un’area di memoria (segmenti `shm`, `mmap` su file, …). Il kernel mappa l’area in ciascuno spazio di indirizzi e

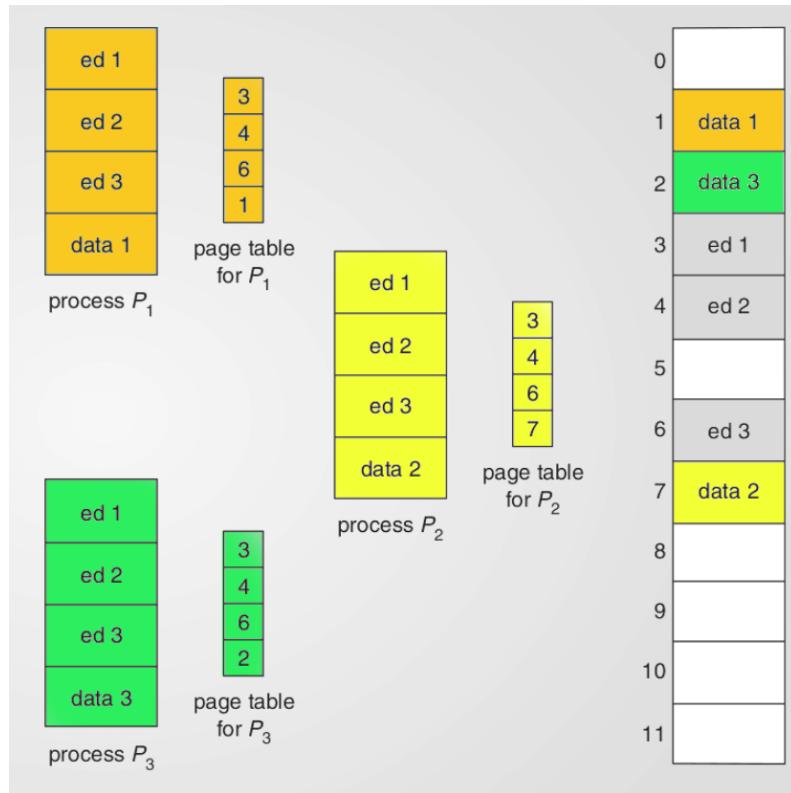


Figura 3.25: Condivisione delle pagine di codice tra processi distinti: PTE diverse mappano sullo stesso frame fisico; i dati restano privati.

fa puntare le rispettive PTE agli stessi frame. Gli indirizzi virtuali possono differire tra i processi, ma la traduzione porta agli stessi frame: le scritture sono viste da tutti i partecipanti secondo i permessi impostati (tipicamente lettura/scrittura).

### 3.9.3 Funzionamento con tabelle multilivello

La condivisione è compatibile con le page table multilivello: basta che le PTE foglia dei processi coinvolti contengano lo stesso *numero di frame* e la stessa maschera di protezione. Le directory intermedie (livelli superiori) possono anche essere distinte: ciò che conta è l'identità dell'entry foglia che mappa pagina→frame.

### 3.9.4 Limiti con *inverted page table* (aliasing)

Con una **tabella delle pagine invertita** (una entry per *frame* e non per pagina virtuale) la chiave di lookup è la coppia ( $PID$ , *page\_virt*). Il problema delle pagine condivise è l'*aliasing*: più chiavi diverse (una per processo) devono mappare lo stesso frame. Se l'implementazione ammette una sola chiave per entry, la condivisione costringerebbe a “ri-etichettare” la voce quando cambia il processo che accede, con costi elevati e pessima scalabilità (specialmente su multicore). Implementazioni pratiche di IPT evitano il problema usando strutture *hashed* con *liste di collisione* (più voci ( $PID, VP$ ) → *frame* che possono riferire lo stesso frame) oppure tabelle ausiliarie per gli alias. In assenza di tali accorgimenti, la condivisione su IPT è inefficiente.

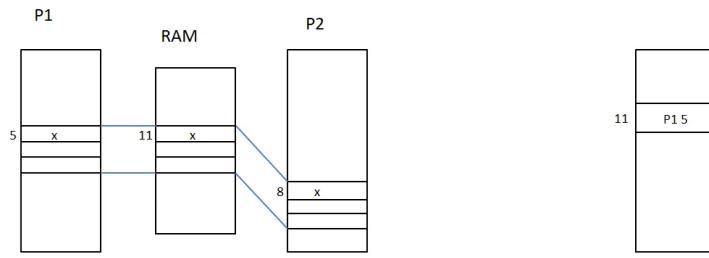


Figura 3.26: Esempio didattico con IPT: il frame 11 contiene la pagina 5 di  $P_1$ . Un accesso dalla pagina 8 di  $P_2$  deve risolversi allo stesso frame (aliasing). Senza supporto per alias multipli, l'OS sarebbe costretto a ri-etichettare l'entry, con costi elevati.

## Riepilogo

- La condivisione riduce la RAM usata (una sola copia di codice/librerie/file mappati) e migliora il *working set* complessivo del sistema.
- È sicura se le PTE sono protette: codice condiviso in sola lettura; per i dati si usa IPC o COW.
- Funziona naturalmente con page table tradizionali (anche multilivello); con IPT richiede supporto esplicito ad alias multipli, tipicamente tramite tabelle hash o strutture ausiliarie.

### 3.9.5 Copy-on-Write (COW)

Quando due (o più) processi condividono le stesse pagine fisiche, vogliamo risparmiare RAM finché nessuno le modifica, ma garantire copie private quando avviene una scrittura. La tecnica **copy-on-write** realizza proprio questo comportamento.

Le pagine condivise vengono mappate nelle page table dei processi in *sola lettura* (bit di scrittura a 0). Finché si eseguono letture, tutti vedono la stessa copia fisica. Al *primo tentativo di scrittura* su una pagina condivisa, l'hardware rileva la violazione di protezione (fault di scrittura su pagina read-only) e invoca l'handler del kernel, che:

1. alloca un nuovo frame libero;
2. *copia* il contenuto della vecchia pagina nel nuovo frame;
3. aggiorna la PTE del *solo* processo che ha scritto per farla puntare al nuovo frame, abilitando il bit di scrittura;
4. lascia le altre PTE puntare alla vecchia pagina (tipicamente ancora read-only).

Da quel momento le due pagine sono indipendenti. In questo modo si mantengono i benefici della condivisione finché possibile, pagando il costo della copia soltanto quando serve (*lazy copy*).

**Fork() e COW.** In UNIX/POSIX, dopo `fork()` il figlio condivide *tutte* le pagine del padre: il kernel duplica la *struttura* delle page table (non i contenuti) e marca le PTE come read-only con il flag COW. Così `fork()` non richiede copie di memoria immediate. Se il padre o il figlio scrivono, si innesca la copia *solo* delle pagine toccate (le altre restano condivise); se il figlio chiama `execve()`, le sue vecchie mappature vengono rimpiazzate e la condivisione si dissolve senza alcuna copia.

**Dettagli operativi.** Il fault generato è un *page/protection fault* su scrittura: l'handler riconosce il caso COW tramite i bit della PTE (es. flag COW e permessi), alloca/copia/ritocca i permessi e poi *ri-esegue* l'istruzione che aveva faultato. Se più figli condividono la stessa pagina e uno di essi scrive, il kernel crea una *singola* nuova copia per quel processo; gli altri continuano a condividere l'originale finché non scrivono a loro volta.

**Vantaggi e costi.** *Vantaggi:* riduzione dell'RSS, meno allocazioni e traffico memoria/disco, avvio rapidissimo dopo `fork()`. *Costo:* un overhead al *primo write* (allocazione, copia, invalidazioni TLB) e qualche invalidazione/flush TLB per sincronizzare le PTE tra CPU.

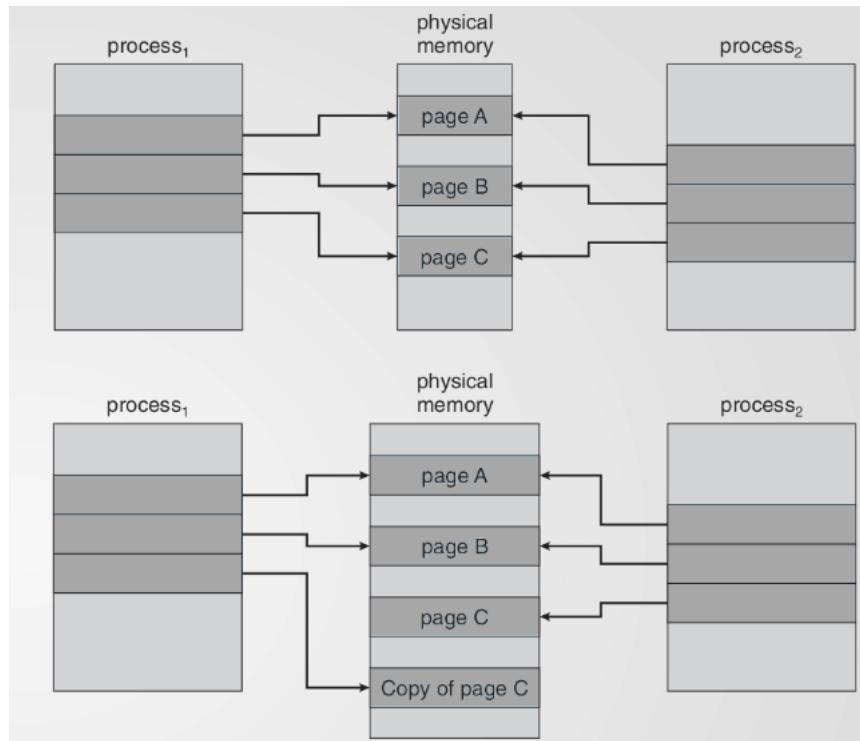


Figura 3.27: Copy-on-Write: inizialmente una pagina è condivisa in sola lettura; al primo tentativo di scrittura il kernel alloca e popola una copia privata e aggiorna la PTE del processo che ha scritto.

### 3.9.6 Zero-fill-on-demand

Quando un processo richiede nuove pagine anonime (es. crescita dell'heap via `sbrk()` o `mmap()` anonimo), il kernel evita di preparare subito tanti frame fisici azzerandoli uno per uno. Invece usa una *pagina zero condivisa (read-only static zero page)*: una singola pagina fisica riempita di zeri e marcata in sola lettura, che viene mappata contemporaneamente in tutte le nuove voci della page table. Così il processo “vede” immediatamente memoria valida e inizializzata a zero, ma nessun frame aggiuntivo è stato ancora assegnato. Al primo accesso in scrittura su una di queste pagine si genera un fault di protezione (write su read-only) che l'handler riconosce come caso *copy-on-write*: il kernel preleva un frame libero, lo *azzerà* (garantendo che non contenga residui di altri processi), copia logicamente il contenuto “tutto zero”, aggiorna la PTE del processo per puntare al nuovo frame e imposta i permessi in scrittura; quindi ripete l'istruzione. In lettura non avviene alcun fault: tutte le letture restituiscono zero dalla pagina condivisa. Questo schema differisce dall'azzeramento anticipato dei frame perché sposta il costo di pulizia al *primo uso (on demand)* e solo per le pagine effettivamente toccate, riducendo latenza.

e traffico di memoria al momento della richiesta. In sintesi, lo zero-fill-on-demand combina pagina zero condivisa + COW per fornire memoria anonima inizializzata a zero in modo "pigro", efficiente e sicuro (niente leakage di vecchi dati).

### 3.9.7 Librerie condivise con linking

Molti programmi invocano funzioni fornite da librerie esterne. Esistono due modi principali per includerle.

**Linking statico:** il codice oggetto della libreria viene copiato dentro l'eseguibile al momento del linking. Vantaggi: nessuna dipendenza esterna a runtime e avvio immediato; svantaggi: eseguibili più grandi, niente condivisione in RAM tra processi, aggiornamenti della libreria non propagati automaticamente.

**Linking dinamico:** l'eseguibile contiene solo i riferimenti ai simboli; al execve l'*dynamic linker/loader* (es. ld.so su ELF/Linux) mappa le librerie condivise (.so/DLL) nello spazio di indirizzi del processo e risolve i simboli, subito (*eager*) o al primo uso. Il codice delle librerie (text) è mappato in sola lettura ed è *condiviso* tra tutti i processi: più processi vedono lo stesso insieme di frame fisici. Le sezioni dati sono in genere mappate *private, copy-on-write*: all'inizio condividono i frame di origine; alla prima scrittura il kernel duplica la pagina per il solo processo che ha scritto, preservando l'isolamento.

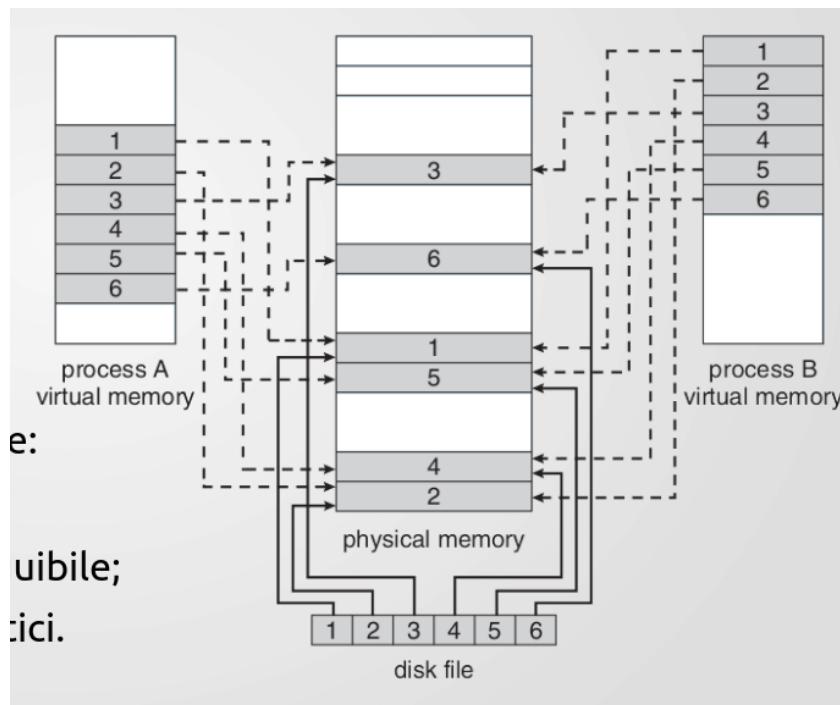


Figura 3.28: Due processi mappano la stessa libreria: il codice (numeri) è condiviso in frame fisici comuni; le scritture su dati avvengono tramite COW o, se richiesto, sono propagate al file.

### 3.9.8 Librerie condivise con file mappati

Un'alternativa (in realtà il meccanismo di base usato anche dal dynamic loader) è la mappatura di file con `mmap`. Il kernel inserisce nel VA del processo un intervallo che "punta" al file su disco; inizialmente

le PTE sono non presenti: al primo accesso si genera un *page fault* e la pagina richiesta viene letta dal file (*demand paging*) e messa in RAM. Solo le pagine effettivamente toccate vengono caricate; le pagine del codice e dei dati read-only sono automaticamente condivisibili fra processi che mappano lo stesso file, riducendo occupazione di memoria. Se una pagina mappata è modificata (mappatura MAP\_SHARED), diventa *dirty* e il kernel la riscrive sul file quando serve liberare il frame; con MAP\_PRIVATE la scrittura avviene su una copia privata (COW) e non ritorna sul file. Lo stesso approccio è usato all'inizializzazione di un processo: l'eseguibile ELF viene *mappato*, non copiato; le sezioni `text`/`rodata` sono condivise e protette in sola lettura; la sezione `.data` è privata con COW; la `.bss` (dati non inizializzati) è fornita tramite *zero-fill-on-demand* usando la pagina zero condivisa; lo *heap* cresce con `sbrk/mmap` ed è gestito dall'allocatore utente (lista/arena) sopra la memoria virtuale, mitigando la frammentazione esterna reale.

## 3.10 Allocazione della memoria per il kernel

Il kernel è composto da più sottosistemi e driver che, durante l'esecuzione, devono creare e distruggere continuamente strutture dati (processi, inode, dentry, socket, buffer I/O, ecc.). A differenza dei processi utente, il kernel talvolta ha bisogno di *memoria fisica contigua* e indirizzi fisici reali (per esempio per il DMA, che accede alla DRAM bypassando la MMU); in altri casi può usare normalmente la memoria virtuale. Per questo le allocazioni del kernel seguono vie diverse: per richieste “grandi” si riservano pagine (o insiemi di pagine), per oggetti “piccoli” si usa un allocatore specializzato che minimizza overhead e frammentazione.

### 3.10.1 Slab allocator

La memoria fisica è suddivisa in frame (pagine). Uno **slab** è un gruppo di pagine *fisicamente contigue*; una **cache** è una collezione di slab che servono oggetti di *una specifica taglia* (o di un tipo omogeneo). Il kernel mantiene più cache: specializzare per taglia permette di soddisfare le richieste in tempo costante e di ridurre la frammentazione. Ogni slab è suddiviso in *slot* tutti della stessa dimensione; qualunque slot libero può ospitare un nuovo oggetto. Gli slab hanno tipicamente tre stati: *vuoto*, *parziale*,  *pieno*; gli oggetti rilasciati ritornano alla cache e possono essere riutilizzati senza riallocare o reinizializzare memoria (*object reuse*). Questo riuso, unito a eventuali *constructor/destructor* di cache, riduce drasticamente il costo di allocazioni frequenti in percorsi critici. La frammentazione *esterna* è mitigata perché gli slab possono trovarsi ovunque in RAM (non serve un grande intervallo contiguo globale; serve contiguità solo *intra-slab*); quella *interna* è contenuta perché ogni cache è pensata per una taglia precisa (l'eventuale spazio di allineamento dentro una pagina resta limitato). In pratica, quando una cache non ha slot liberi, il kernel chiede nuove pagine al livello sottostante (tipicamente un *buddy allocator*) per creare un altro slab; quando molti slot tornano liberi, uno slab può essere restituito al sistema. Questo schema consente allocazioni/deallocazioni a bassa latenza, deterministiche e con buona locality (gli oggetti della stessa classe cadono vicino in memoria), qualità necessarie nel contesto kernel e nelle routine di interrupt.

# Capitolo 4

## File System e Dischi

### 4.1 File

I file sono un meccanismo di astrazione: forniscono un metodo per salvare informazioni sul disco e leggerle in seguito. Ciò deve avvenire in modo da nascondere all'utente i dettagli implementativi.

#### 4.1.1 Denominazione di file

Le regole per la denominazione possono variare a seconda dal sistema, ma in generale tutti i sistemi operativi moderni considerano sempre validi i nomi composti da stringhe di lettere, frequentemente numeri e caratteri speciali. Alcuni file system fanno distinzione tra maiuscole e minuscole (UNIX) e altri no (MS-DOS).

#### 4.1.2 Estensioni

Nella maggior parte dei sistemi operativi moderni il nome di un file è diviso in due parti separate da un punto, dove la prima è il nome e la seconda è detta **estensione di file** e indica generalmente una particolarità del file. Per esempio, un'estensione può essere .zip (per indicare una tipologia di compressione) o .pdf (per indicare un documento di tipo pdf). Queste estensioni però non sempre sono forzate ma sono solo convenzioni (alla fine un file è comunque una sequenza di byte), come nel caso di UNIX. Alla fine infatti, un file chiamato file.txt potrebbe essere un file di testo, ma l'estensione vale più da promemoria per il proprietario per capire come aprire quel file. Queste convenzioni sono utili quando un programma supporta diverse tipologie di file, come il compilare gcc che supporta file .c, file assembly, file di libreria e così via. In ogni caso, per capire di che tipologia è un file si utilizza un preambolo di byte chiamato **magic number** che identifica univocamente il tipo (generalmente e solo su UNIX).

#### 4.1.3 Struttura

Un **file** è una collezione nominata di informazioni persistenti conservate su memoria di massa. Al file sono associati *attributi* (nome, tipo, dimensione, proprietario e permessi, timestamp, ecc.) e un *contenuto* organizzato secondo un modello logico. Nei sistemi moderni coesistono tre schemi principali.

**(A) Sequenza di byte.** Il file è visto come uno *stream* di byte senza struttura interna interpretata dal kernel. L'OS fornisce operazioni di lettura/scrittura a offset arbitrari; l'interpretazione (record, righe,

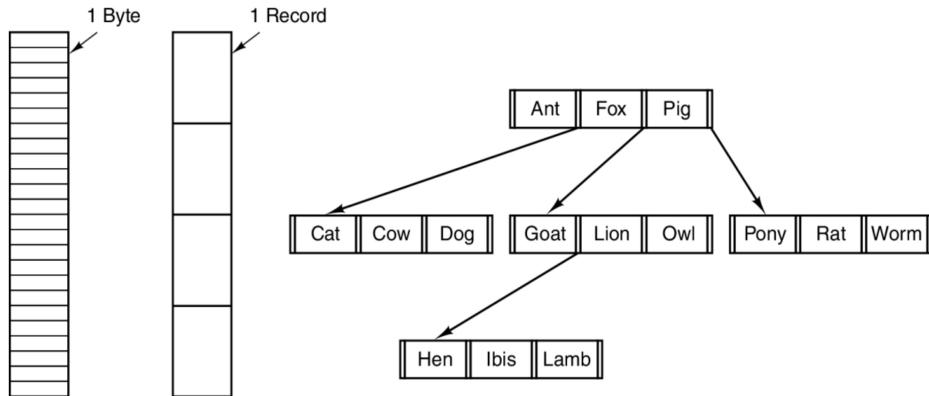


Figura 4.1: Tre strutture di file: (a) Sequenza di byte. (b) Sequenza di record. (c) Ad albero.

oggetti) è tutta in capo all'applicazione. È il modello adottato da Unix e Windows: semplice, flessibile, adatto a qualsiasi formato, con costi bassi nel kernel.

**(B) Sequenza di record.** Il file è una sequenza di *record* (di lunghezza fissa o variabile). L'interfaccia offre primitive orientate al record (`read_record`, `write_record`, `seek_record i`). Spesso ogni record ha un *numero* o una *chiave*; il sistema può mantenere tabelle o indici per accesso diretto al record *i* o per ricerca per chiave. È tipico dei sistemi transazionali/mainframe e di alcuni formati log/telemetria: facilita I/O strutturato e validazioni, ma è meno generale dello stream di byte.

**(C) File indicizzato ad albero.** Il contenuto è organizzato tramite una struttura ad *albero* (di norma B/B<sup>+</sup>-tree o ISAM) che indica record/cluster per *chiave*. L'accesso avviene tramite ricerche e range query efficienti; inserimenti/cancellazioni aggiornano nodi e foglie. Questo schema è usato da DBMS e da alcuni formati/feature di FS per file “grandi” o key-value (es. log strutturati, archivi indicizzati): offre lookup e scansioni ordinate con complessità  $O(\log n)$  al prezzo di metadati e aggiornamenti più complessi.

In sintesi, la *sequenza di byte* massimizza la generalità e delega il layout all'applicazione; la *sequenza di record* standardizza l'unità logica di I/O; l'*indice ad albero* integra nel file un meccanismo di accesso efficiente per chiave/ordine.

#### 4.1.4 Tipologie

**File regolari.** Un *file* è l'unità logica di memorizzazione offerta dal file system: un contenitore *astratto* di dati identificato da un nome e da un insieme di metadati (dimensione, proprietario, permessi, istanti di creazione/modifica/accesso, ecc.). Nella maggior parte dei sistemi moderni (Unix-like inclusi) un file regolare è visto come una *sequenza di byte* senza struttura imposta dal kernel; la struttura interna (record, campi, header) è responsabilità del programma che lo legge/scrive. L'accesso può essere sequenziale o posizionale (random access) tramite l'offset del file. Nei sistemi Unix i metadati sono mantenuti nell'*inode*, mentre il nome è conservato nella directory che contiene l'*entry*.

**Directory.** Una *directory* è un file speciale che realizza una tabella di associazione tra nomi e oggetti del file system (tipicamente *inode*). In questo modo le directory implementano una gerarchia ad albero radicata che permette percorsi assoluti e relativi, e rende possibile la condivisione di oggetti tramite collegamenti (*hard link* verso lo stesso *inode* e *symbolic link* come riferimenti a un percorso). Anche

le directory possiedono permessi (almeno lettura per elencare i nomi, esecuzione per attraversarle, scrittura per crearvi o rimuovervi voci).

**File speciali a caratteri.** I *character special files* rappresentano dispositivi che scambiano flussi di byte in modo orientato al carattere (tipicamente terminali, porte seriali, tastiere, alcuni dispositivi sensori). L'I/O è in genere non bufferizzato dal dispositivo e non è garantita la possibilità di *seek*. In Unix questi nodi compaiono come entry in /dev e sono identificati da una coppia *major/minor* che seleziona il driver e l'istanza di dispositivo. Dal punto di vista delle chiamate di sistema, leggere e scrivere un file a caratteri equivale a leggere e scrivere un file regolare, realizzando l'astrazione “*everything is a file*”.

**File speciali a blocchi.** I *block special files* modellano dispositivi a blocchi (dischi, SSD, eMMC, partizioni) che espongono un indirizzamento a blocchi di dimensione fissa e supportano il posizionamento casuale. Il kernel di solito interpone cache e schedulazione dell'I/O (*page cache, buffer cache*) per aggregare richieste e ottimizzare la latenza. Anche questi nodi vivono in /dev e sono identificati da numeri *major/minor*. Tipicamente un file system viene “montato” sopra un dispositivo a blocchi (o una sua partizione) per fornire file e directory.

#### 4.1.4.1 File binari

Con *file binario* si intende un file il cui contenuto non è testo codificato in caratteri leggibili, ma byte interpretati secondo un formato specifico (ad es. immagini, archivi, database, eseguibili). Gli eseguibili e le librerie condivise usano formati strutturati (p. es. ELF su Unix/Linux) con intestazioni, tabelle di simboli e sezioni; il kernel e il *loader* interpretano tali strutture al *load time*, mentre per i dati applicativi è il programma a imporre e rispettare il layout. In ogni caso, a livello di file system, anche i binari restano sequenze di byte con metadati uniformi alle altre tipologie.

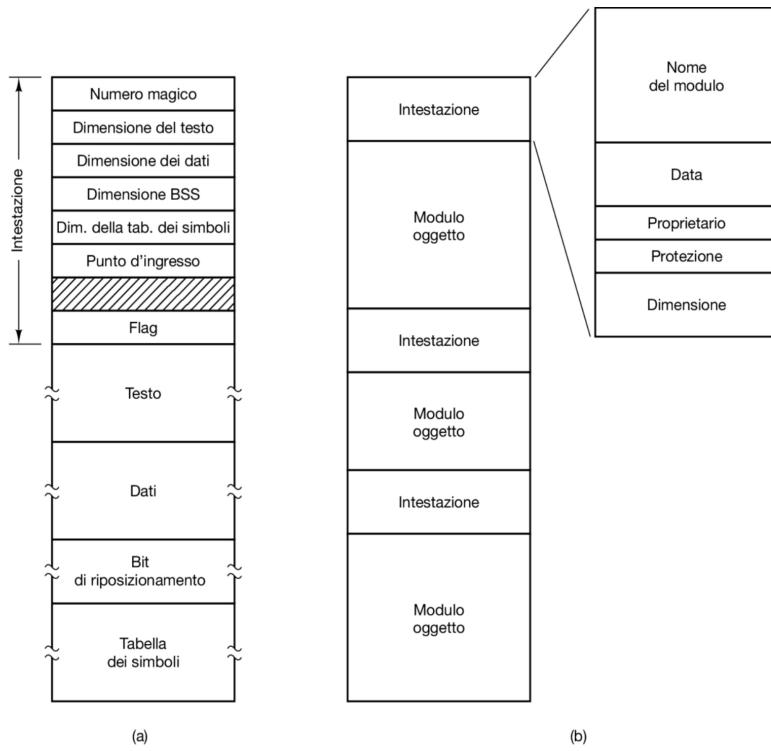


Figura 4.2: (a) Un file eseguibile. (b) Un archivio

#### 4.1.5 Accesso ai file

Un *metodo di accesso* stabilisce come un processo percorre i byte di un file e quali primitive il sistema operativo mette a disposizione per farlo. I sistemi operativi offrono più modelli, spesso coesistenti dietro la stessa interfaccia (es. POSIX), così che ogni applicazione possa scegliere quello più adatto al proprio pattern d'uso.

**Accesso sequenziale** È il modello più semplice: letture e scritture procedono in ordine a partire da una *posizione corrente* (file offset) mantenuta dal kernel per ogni file aperto. Dopo ogni read/write l'offset avanza automaticamente; lo spostamento esplicito è possibile con lseek/fseek. Varianti come pread/pwrite eseguono I/O “posizionale” senza modificare l'offset, utile in presenza di concorrenza.

**Accesso diretto (o casuale)** Il programma specifica di volta in volta *dove* andare nel file (byte offset o numero di blocco/record) e poi effettua l'operazione. È naturale per file organizzati in blocchi omogenei (es. database a pagine, immagini disco, indici), perché evita scansioni lineari e consente salti  $O(1)$  verso l'area d'interesse. In POSIX si realizza combinando lseek+I/O o usando pread/pwrite.

**Accesso indicizzato** Per ricerche veloci su *chiavi* logiche (es. “record con chiave  $K$ ”) il file è affiancato da una struttura d'indicizzazione (tipicamente un  $B^+$ -tree) che mappa chiavi  $\rightarrow$  posizioni fisiche. L'accesso si scompone in: (i) consultazione dell'indice; (ii) accesso diretto alla pagina/record. Si ottengono ricerche logaritmiche e ordinamento naturale dei record, al costo di spazio extra e aggiornamenti dell'indice.

#### Osservazioni

- Le API moderne espongono sia modalità sequenziale (offset implicito) sia diretta (offset esplicito); la scelta può essere fatta per singola operazione.
- Con accessi concorrenti, l'I/O posizionale evita corse critiche sull'offset condiviso; per accodare in sicurezza si usa l'apertura con O\_APPEND, che rende atomiche le scritture in coda.

#### 4.1.6 Attributi dei file

Ogni file ha un nome e i propri dati. Tutti i sistemi operativi associano ulteriori informazioni a ciascun file, per esempio la data e l'ora in cui è stato modificato l'ultima volta e la dimensione chiamati **attributi** (o metadati). Questo elenco cambia in base al sistema. Alcuni di questi possono essere la protezione, ovvero chi può accedere al file e in che modo, la password, il creatore, i flag di sola lettura/nascosto/sistema/archivio/binario ...

#### 4.1.7 Operazioni sui file

Un file system espone ai processi un insieme di primitive per creare, usare e gestire file e directory. Le operazioni fondamentali sono: *creazione/apertura/chiusura, lettura/scrittura, posizionamento, rinomina/rimozione e gestione degli attributi*. In apertura si ottiene un *file descriptor* (o uno *stream* in libreria C); la chiusura rilascia le strutture kernel e scarica eventuali buffer. In ambiente POSIX le chiamate tipiche sono open/creat/close con flag per modalità d'accesso, creazione esclusiva, troncamento, accodamento (O\_RDONLY/O\_WRONLY/O\_RDWR, O\_CREAT, O\_EXCL, O\_TRUNC, O\_APPEND). Il trasferimento dati avviene con read/write (I/O non bufferizzato) oppure via libreria stdio (fread/fwrite, fgets/fputs) che riducono il numero di chiamate di sistema grazie al buffering . Il *file offset* (posizione corrente) può essere letto/spostato con lseek rispetto a inizio, posizione corrente o fine file; sono disponibili varianti “posizionali” atomiche per I/O concorrente (pread/pwrite) che evitano corse critiche tra lseek e read/write . Sulle dimensioni si può operare con truncate/ftruncate (troncamento/estensione che può creare “*hole*”). Gli *attributi* (permessi, proprietari, timestamp, tipo, i-node, dimensione) si ispezionano con stat/lstat/fstat; i permessi si modificano con chmod e i proprietari con chown; la maschera di creazione (umask) limita i permessi iniziali . Per la *nomenclatura* sono previste rename (rinomina atomica) e operazioni su collegamenti: link/unlink (hard link) e symlink/readlink (link simbolici), strumenti chiave anche per la condivisione a livello di file system :contentReference[oaicite:5]index=5. Le directory si creano e rimuovono con mkdir/rmdir, si attraversano con opendir/readdir/closedir; chdir/getcwd gestiscono la *current working directory* :contentReference[oaicite:6]index=6. Infine, molte piattaforme offrono la *mappatura di file* (mmap/munmap/msync) che integra i file nello spazio di indirizzi del processo: le pagine della mappa sono caricate a domanda e le scritture possono essere condivise o private (copy-on-write), fornendo un'alternativa ad alte prestazioni a read/write. In sintesi, le “operazioni sui file” coprono ciclo di vita (crea/apri/chiudi), I/O, gestione della posizione, metadati e nome, oltre a primitive per directory e memoria mappata; le slide del corso collocano queste primitive tra i “dettagli d'implementazione” essenziali dell'astrazione file system.

## 4.2 Directory

Le directory sono un'altra astrazione che il sistema operativo fa per tenere traccia dei file, ma in realtà sono anch'esse dei file.

### 4.2.1 Sistemi di directory

Un *sistema di directory* definisce come i nomi dei file sono organizzati e risolti all'interno del file system. Lo scopo è fornire un *namespace* per nominare, raggruppare e trovare gli oggetti (file, directory, link, dispositivi) in modo efficiente e comprensibile per utenti e programmi. Di seguito contrapporremo due modelli classici: *a livello singolo* e *gerarchico*.

**Directory a livello singolo** Nel modello a livello singolo esiste un'unica directory che contiene tutte le voci del sistema. Tutti i file devono avere nomi univoci a livello globale poiché condividono lo stesso contenitore. Il vantaggio è la massima semplicità: non esiste risoluzione di percorsi complessi e la ricerca può essere lineare o supportata da un indice unico. Gli svantaggi emergono subito in ambienti multiutente o con molti file: conflitti di nomi frequenti, assenza di raggruppamento logico per progetto/utente/tipo, difficoltà a gestire permessi differenziati e a mantenere ordine semantico. Questo modello è oggi usato solo in sistemi molto piccoli o come astrazione didattica.

**Directory a livello gerarchico** Il modello gerarchico organizza le directory in un *albero* con una radice (es. / in Unix). Ogni directory mappa nomi locali ad oggetti (file o sotto-directory), consentendo raggruppamenti naturali per utente, progetto o funzione. La risoluzione dei nomi avviene *percorrendo* le componenti di un *pathname*: percorsi *assoluti* partono dalla radice; percorsi *relativi* sono interpretati rispetto alla *current working directory* del processo. Questo modello scala bene perché suddivide il namespace in sottoalberi indipendenti; permette politiche di permesso granulari (lettura/listing, attraversamento, creazione/rimozione voci), e abilita la composizione di file system diversi nello stesso albero tramite *mount* su punti di attacco. Per la condivisione si usano *hard link* (più nomi per lo stesso oggetto, tipicamente nel medesimo file system) e *symbolic link* (riferimenti a percorsi arbitrari), mantenendo l'albero aciclico a livello di directory effettive. La ricerca dei file può essere accelerata con cache di dentry/inode e strutture indicizzate interne; le operazioni su directory (creazione, ridenominazione, attraversamento) sono atomiche secondo le garanzie del file system sottostante. In sintesi, la gerarchia offre ordinamento semantico, isolamento e controllo dei permessi, nonché integrazione trasparente di volumi e servizi eterogenei, risultando il modello di fatto nei sistemi moderni.

### 4.2.2 Nomi e percorsi

Un *nome di percorso* identifica un oggetto del file system specificando la sequenza di directory da attraversare fino alla voce desiderata. I percorsi **assoluti** iniziano dalla radice (es. /home/alice/doc.txt) e sono indipendenti dalla directory corrente; i percorsi **relativi** sono risolti rispetto alla *current working directory* del processo (es. ../doc.txt). Le componenti speciali . e .. denotano rispettivamente la directory corrente e quella padre. La risoluzione di un percorso procede componente per componente, verificando i permessi necessari su ciascuna directory attraversata (almeno *execute* per attraversare, *read* per elencare le voci). I sistemi definiscono limiti su lunghezza dei nomi e del percorso e regole di *nomenclatura* (case-sensitive vs case-insensitive, caratteri ammessi, codifica). Per robustezza e sicurezza è comune la *normalizzazione/canonicalizzazione* (collazzo di //, rimozione di . e risoluzione di .. quando possibile) e l'attenzione a link simbolici durante il traversal. I namespace possono essere composti tramite *mount*: l'aggancio di un file system a un punto dell'albero rende trasparente la provenienza fisica dei dati, mantenendo un'unica vista gerarchica.

### 4.2.3 Operazioni sulle directory

Le directory realizzano mappe *nome* → *oggetto* e offrono primitive per gestire il namespace. Le operazioni tipiche includono: creazione e rimozione di directory (`mkdir/rmdir`); aggiunta e rimozione di voci (`link/unlink` per hard link; `symlink/readlink` per link simbolici); rinomina atomica (`rename`) che aggiorna una voce senza stati intermedi visibili; apertura e scansione del contenuto (`opendir/readdir/closedir`). La risoluzione dei nomi usa la directory come tabella: cercare una voce richiede almeno i permessi di *execute* sull'intera catena di directory e di *read* per elencarne le entry. Per evitare cicli nell'albero, gli hard link alle directory sono in genere proibiti (fanno eccezione `.` e `..`); i link simbolici, essendo riferimenti a percorsi, non alterano la struttura fisica dell'albero ma influenzano la risoluzione. In presenza di mount e file system eterogenei, rename tra volumi diversi può essere implementata come copia+unlink. Le implementazioni moderne impiegano cache di dentry/inode per velocizzare lookup ripetuti, mentre il journaling/CoW garantisce consistenza dei metadati (creazioni, ridenominazioni, rimozioni) anche in caso di crash.

## 4.3 File System

### 4.3.1 Definizione

Un **file system** è l'insieme di strutture dati, regole e primitive con cui il sistema operativo organizza, nomina, memorizza, condivide e protegge informazioni su memorie di massa, offrendo ai processi l'astrazione di *file* e *directory* indipendente dall'hardware.

### 4.3.2 Scopo e utilizzi

Gli obiettivi principali del file system sono la **persistenza** dei dati oltre la vita dei processi, la **condivisione controllata** tra più processi/utenti, l'**affidabilità** e il recupero dopo eventuali guasti, le **prestazioni** tramite caching e politiche di allocazione, la **protezione** tramite permessi e la **portabilità e trasparenza** dei dati rispetto ai dispositivi fisici.

### 4.3.3 Layout del file system

Un file system su disco non è semplicemente un insieme di file: occupa aree ben definite del dispositivo, con strutture dedicate all'avvio, ai metadati e ai contenuti. In generale si distinguono due piani: il *partizionamento* del disco e il *layout* del file system all'interno di ciascuna partizione.

**Partizionamento e bootstrap.** Sui dischi tradizionali è presente un settore iniziale di avvio (MBR), che solitamente è il settore 0, con tabella delle partizioni e un piccolo codice di bootstrap. Oggi è più utilizzato invece **UEFI**: Unified Extensible Firmware Interface che supera i limiti di MBR in velocità e capacità. Intel ha proposto di sostituirlo ed è attualmente il modo più diffuso di avviare i PC. UEFI, diversamente da MBR, cerca la posizione della **tavella delle partizioni** nel secondo blocco del dispositivo riservando il primo blocco come marcatore speciale per il software che si aspetta di trovarsi un MBR (che in realtà non c'è). La **GPT** (GUID Partition Table) contiene informazioni sulla posizione delle varie partizioni sul disco, GUID infatti sta per *Globally Unique IDentifiers*.

**Strutture on-disk del file system.** Dentro la partizione, il layout tipico comprende:

- **Superblocco:** descrive il volume (dimensione, dimensione dei blocchi, conteggio di blocchi e inode, feature, stato) e punta alle aree principali. È critico per il mount, perciò spesso è replicato.

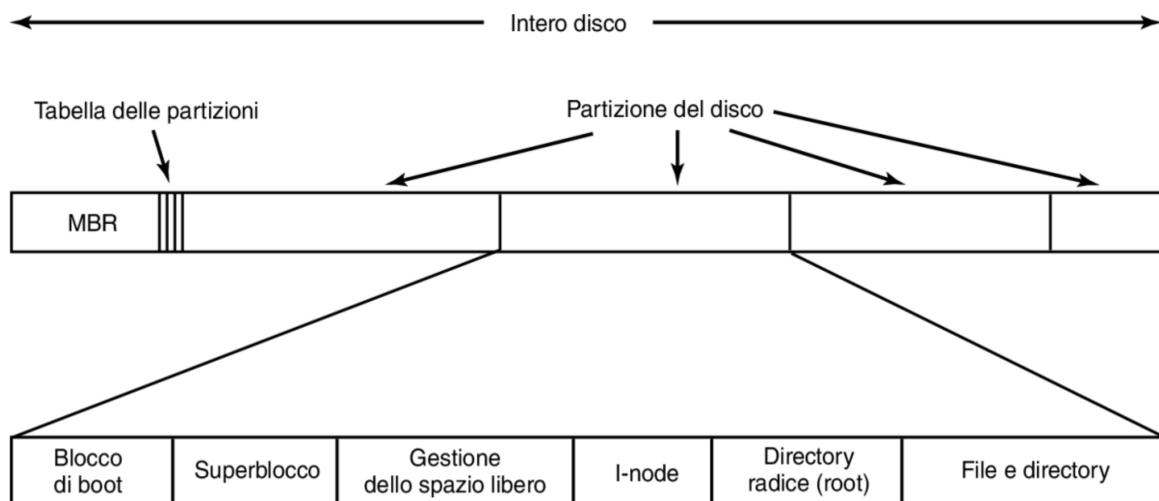


Figura 4.3: Possibile layout file system.

- **Gestione dello spazio libero:** bitmap e/o strutture indicizzate che marcano i blocchi liberi/occupati e supportano politiche di allocazione (vicinanza, estensione, riduzione della frammentazione).
- **Inode (nodi-indice):** record che descrivono file e directory (tipo, permessi, proprietari, timestamp, dimensione) e contengono i riferimenti ai blocchi dati (diretti/indiretti o *extents*). Gli inode possono essere organizzati in gruppi per migliorare la località.
- **Directory:** file speciali che mappano *nome* → *inode*; la risoluzione dei percorsi attraversa queste mappe componenti per componente.
- **Area dati:** blocchi contenenti il contenuto effettivo dei file e le pagine di metadati delle directory.

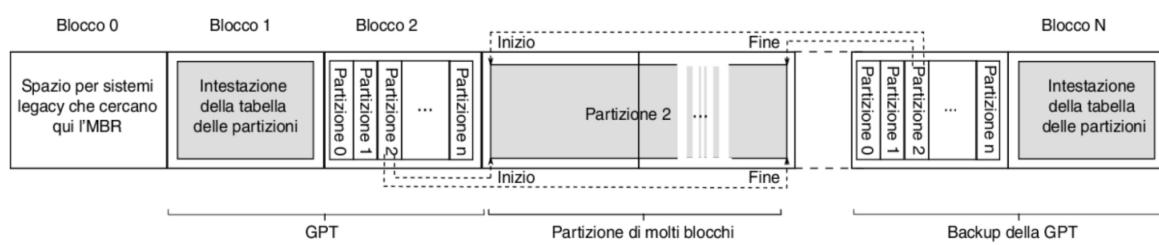


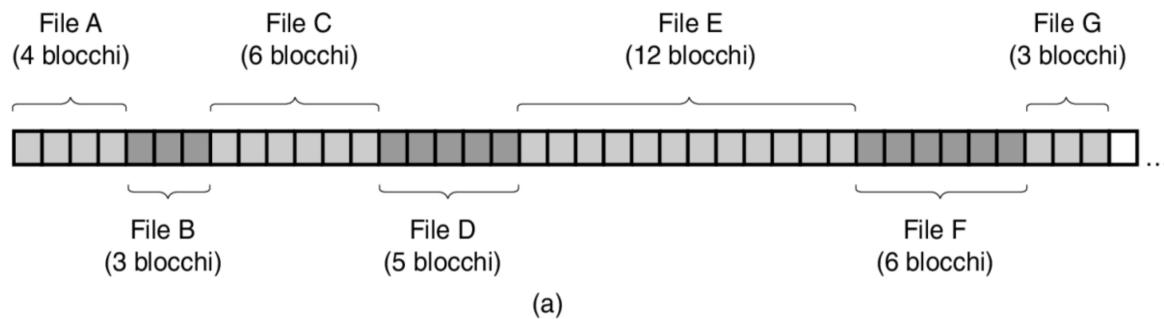
Figura 4.4: Layout di UEFI con tabella delle partizioni.

**Robustezza e recupero.** Per garantire consistenza dopo crash molti file system usano un *journal* per i metadati: le operazioni strutturali (creazione, rinomina, aggiornamenti a bitmap e inode) sono registrate in un log e applicate in modo da poter essere ripetute in sicurezza al riavvio. Altri sistemi impiegano tecniche *copy-on-write* per metadati (e talvolta dati), committando in modo atomico un nuovo root dell'albero dei puntatori.

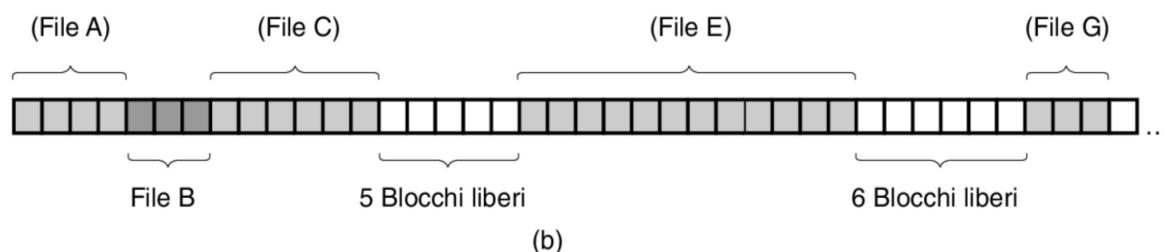
#### 4.3.4 Implementazione di file

Dal punto di vista del file system, “implementare un file” significa stabilire come i suoi byte logici sono mappati in *blocchi* fisici su disco e quali metadati servono per ritrovare tali blocchi. In genere si separano: (i) un *descrittore* per file (inode/record FCB) con attributi e puntatori; (ii) le *voci di directory* che mappano *nome* → *identificatore del file* (es. numero di inode); (iii) strutture per lo *spazio libero* (bitmap/liste).

**Allocazione contigua.** Il file occupa blocchi adiacenti; il descrittore contiene *indirizzo iniziale* e *lunghezza*. Pro: throughput ottimo per accesso sequenziale, calcolo diretto dell’offset (un solo seek); metadati minimi. Contro: *frammentazione esterna* e difficoltà a far *crescere* il file senza ricopiare (come si vede in figura 4.5); richiede stima preventiva della dimensione. Una mitigazione moderna è usare *extents* (coppie <*inizio*, *lunghezza*>): pochi intervalli contigui al posto di una contiguità perfetta.



(a)



(b)

Figura 4.5: (a) Allocazione contigua dello spazio del disco per sette file. (b) Lo stato del disco dopo la rimozione dei file D e F.

**Allocazione concatenata (a lista collegata).** Ogni blocco del file contiene un puntatore al successivo; il descrittore conserva il *primo* (e spesso l’ultimo) blocco. Pro: niente frammentazione esterna, crescite facili. Contro: accesso casuale pessimo (bisogna seguire la catena), overhead di puntatori in ogni blocco, scarsa affidabilità in caso di corruzione di un link.

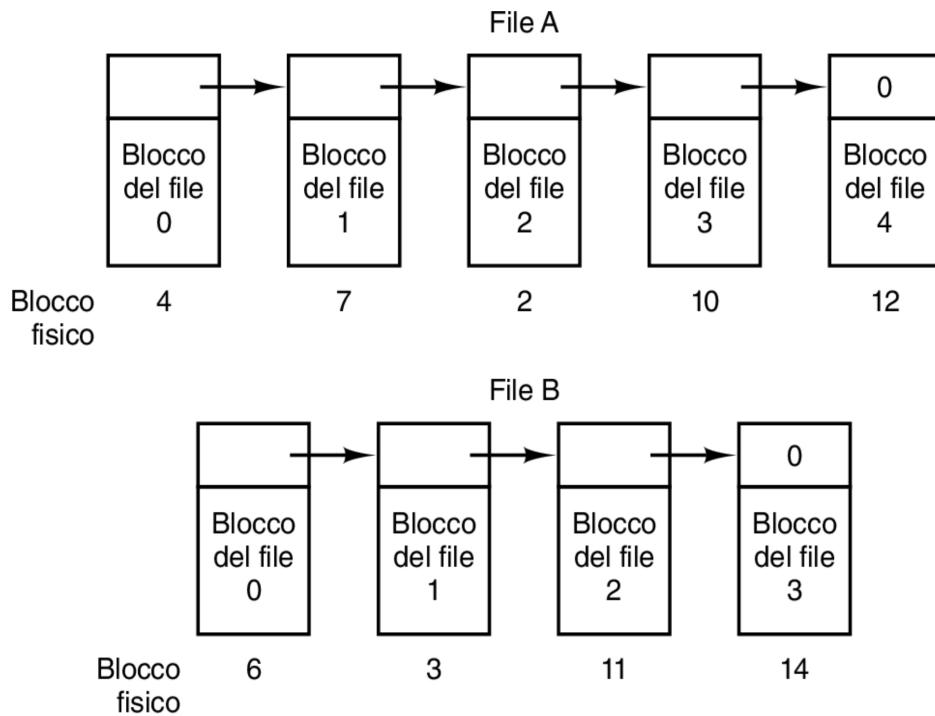


Figura 4.6: Memorizzazione di un file come lista concatenata di blocchi del disco

**Allocazione indicizzata (inode).** Ogni file ha un descrittore (inode) con un *vettore di puntatori* ai blocchi. Per scalare: un mix di *puntatori diretti* (per file piccoli) e *indiretti singoli/doppi/tripli* che puntano a blocchi-indice contenenti ulteriori riferimenti. Pro: buon accesso casuale e sequenziale, niente puntatori nei blocchi dati, adatto a file di ogni taglia. Contro: più metadati e un numero maggiore di letture per raggiungere blocchi molto lontani (in presenza di indirezioni profonde). Molti FS moderni sostituiscono/affiancano i puntatori con *extents* indicizzati.

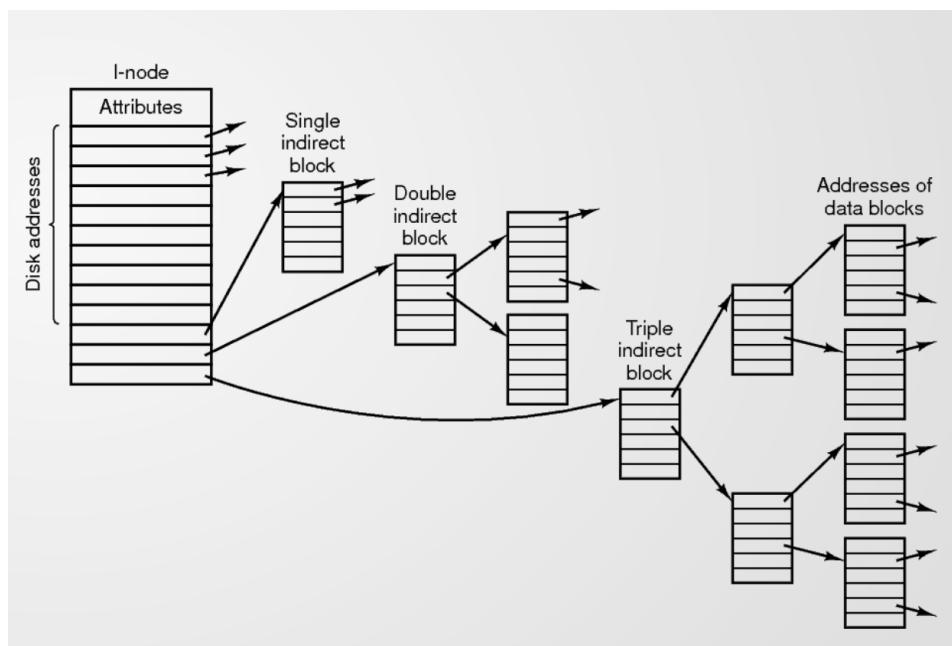


Figura 4.7: Un esempio di i-node.

### FAT (File Allocation Table).

Variante della concatenata: la catena dei blocchi non è nei blocchi dati, ma in una *tabella centrale* (FAT) indicizzata per numero di blocco. **Pro:** l'accesso casuale migliora (la catena si segue in RAM), i blocchi dati restano “puliti”. **Contro:** la tabella può diventare grande ed essere un collo di bottleneck; serve coerenza tra cache della FAT e disco.

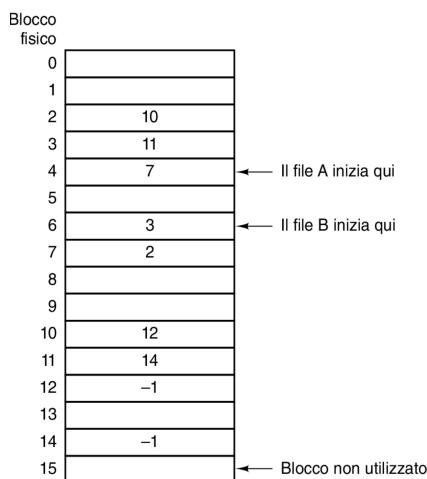


Figura 4.8: FAT: catena dei blocchi nella tabella centrale.

**Ottimizzazioni pratiche.** Per *file piccoli* si può memorizzare una parte dei dati *inline* (nell'inode/record) evitando l'allocazione di un blocco intero; per *file grandi* si usano alberi (B/B<sup>+</sup>-tree) di extents per ridurre i metadati e migliorare le ricerche. La *località* è favorita raggruppando inode e dati “vicini” (gruppi/cylinder groups) e con politiche di posizionamento. La gestione dello *spazio libero* usa spesso *bitmap* (un bit per blocco) per trovare rapidamente porzioni contigue. Per la *consistenza* dopo crash si impiegano *journaling* (log delle modifiche a metadati) o approcci *copy-on-write* che aggiornano atomi di struttura. Infine, *page cache*, *read-ahead* e *write-back* nascondono la latenza del disco e riducono il numero di I/O fisici.

#### 4.3.5 Implementazioni di directory

Una *directory* è un contenitore che mappa *nome* → *oggetto* (file o sotto-directory). L'implementazione influenza soprattutto **prestazioni di lookup/crea/elimina**, gestione di **nomi lunghi**, **consistenza** e **scalabilità** quando il numero di entry cresce.

**Voci di directory e metadati.** Due modelli classici: (i) directory come elenco di coppie (*nome, identificatore*) dove l'identificatore punta al descrittore del file (p. es. inode in Unix); (ii) directory che contiene direttamente il *File Control Block* (FCB) con parte dei metadati. Il primo separa nettamente nomi e metadati (più flessibile e condivisibile via hard link), il secondo riduce un accesso in più ma complica rinomina/spostamento e aumenta il costo di aggiornamento.

**Liste lineari: semplici ma  $O(n)$ .** L'implementazione più semplice è una *lista* di entry scandita linearmente per risolvere un nome: ottima per directory piccole, degrada all'aumentare delle voci. Per supportare *nomi a lunghezza variabile* si usano entry con un campo “lunghezza” e slack interno; la cancellazione lascia *buchi* che si recuperano con *coalescenza* o compattazioni periodiche.

**Indicizzazione: hash e alberi bilanciati.** Per directory grandi si affianca un indice:

- **Hash per directory:** una tabella (o più bucket su disco) mappa l'hash del nome all'offset dell'entry; lookup medio  $\approx O(1)$ . Serve gestire collisioni (liste nei bucket) e versionare l'indice per coerenza con il log/journal.

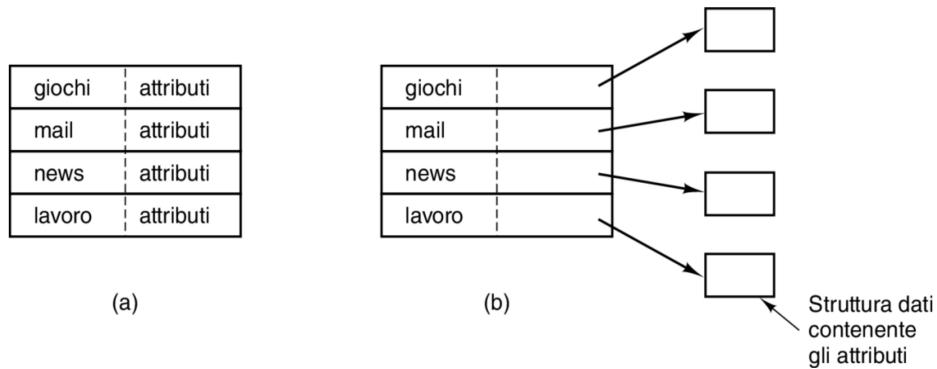


Figura 4.9: (a) Una semplice directory contenente voci a dimensione fissa con gli indirizzi del dsico e gli attributi nella voce della directory. (b) Una directory in cui ogni voce fa soltanto riferimento a un i-node.

- **B/B<sup>+</sup>-tree:** indicizzazione ordinata per nome (o per hash del nome) con complessità  $O(\log n)$ , buona località e supporto efficiente a directory da decine/centinaia di migliaia di entry. Usata in FS moderni (es. “htree” in famiglia ext, B<sup>+</sup>-tree in NTFS/APFS/btrfs).

Gli indici riducono i seek di lookup e accelerano inserimenti/rimozioni, a costo di metadati aggiuntivi e manutenzione in aggiornamento/crash-recovery.

**Gestione dei nomi.** Una voce di directory realizza la mappa *nome* → *oggetto* e contiene almeno il *nome* e un riferimento all’oggetto (es. numero di inode/ID), spesso accompagnati da *lunghezza del nome* e/o da un *hash del nome* per accelerare i lookup. Per supportare **nomi lunghi** i file system usano *entry a lunghezza variabile* che incorporano la stringa oppure un *name-heap* separato: l’entry resta di taglia fissa e punta a un’area compatta dove i nomi sono “impacchettati”; rinomina e cancellazione si gestiscono aggiornando l’offset e riciclando spazio nel name-heap (con eventuale compattazione periodica). La **codifica** dei nomi (UTF-8, UTF-16, ecc.) e la **case sensitive** sono scelte di progetto del FS: sistemi *case-sensitive* confrontano i byte/ code-point così come sono; sistemi *case-insensitive* (spesso *case-preserving*) effettuano *normalizzazione* e *case-folding* coerenti anche per l’hashing, così che lo stesso “nome canonico” conduca alla medesima entry. I file system fissano limiti massimi per *lunghezza del nome* e *del percorso*, e definiscono regole sui caratteri ammessi e sul separatore di directory. **Permessi e traversal:** la risoluzione di un percorso richiede il permesso di *attraversamento* (execute) su ogni directory lungo il cammino; la lettura dell’elenco voci richiede *read*; la creazione/rimozione di voci richiede tipicamente *write+execute* sulla directory. **Link:** gli *hard link* aggiungono nomi alla stessa entità (stesso inode) e sono di norma vietati verso le directory (per preservare l’acyclicità, eccetto . e ..); i *link simbolici* sono entry speciali che memorizzano un percorso (assoluto o relativo) e possono attraversare volumi diversi. **Operazioni sui nomi:** rename all’interno dello stesso volume è progettata come atomica; un link rimuove la voce e decremente il contatore di link; tra volumi distinti la rinomina può degradare a copia+unlink. **Prestazioni:** implementazioni moderne affiancano liste/hash/B<sup>+</sup>-tree con cache di dentry/inode e *negative entries* per evitare I/O ripetuti nei lookup falliti.

**Operazioni e coerenza.** *Lookup, creazione, rimozione* e *rinomina* lavorano sull’insieme delle entry. La rinomina all’interno dello stesso volume è progettata come *atomica*. FS journaling/CoW registrano in modo sicuro gli aggiornamenti alle strutture (entry, indici, bitmap), così che dopo un crash l’albero

rimanga consistente. A livello di kernel, *name/dentry cache* e *inode cache* evitano I/O ripetuti nella risoluzione di percorsi frequenti.

#### 4.3.6 File condivisi

Quando due (o più) utenti vogliono *condividere* lo stesso file, l'obiettivo è avere **un'unica copia** dei dati con **più nomi/percorsi** che la referenziano: ogni modifica è immediatamente visibile a tutti, senza duplicazioni e senza ambiguità su chi “possegga” i blocchi. Le soluzioni dipendono dal tipo di file system.

**Scenario.** Due utenti, /home/alice e /home/bob, desiderano accedere allo stesso file come /home/alice/relazione.txt e /home/bob/relazione.txt. Idealmente entrambe le voci di directory puntano alla *stessa entità* di memorizzazione; cancellare *solo* un nome non deve distruggere i dati se esistono altri riferimenti validi.

**Soluzioni su FAT (tabella di allocazione).** Nei file system tipo FAT le directory contengono un *entry* con metadati minimi e il *primo cluster* del file; la catena dei cluster è tenuta nella tabella centrale (FAT). FAT **non** prevede un descrittore condiviso con *contatore di riferimenti*: creare due entry che puntano allo stesso cluster iniziale genera “*cross-linked files*” e corruzione alla cancellazione (la prima rimozione libera la catena ancora usata dall'altra entry). In pratica, FAT **non supporta hard link** sicuri; la “condivisione” si ottiene solo a livello applicativo (es. collegamenti/shortcut che memorizzano un percorso), senza garanzie del file system.

**Con i-node (stile Unix).** Le directory mappano *nome* → *numero di i-node*. L'i-node è il *descrittore* del file (attributi, puntatori ai blocchi) e mantiene un **link count**. Più nomi possono puntare allo stesso i-node (**hard link**). Cancellare un nome decremente il contatore; i dati sono liberati solo quando il link count scende a 0 e nessun processo ha il file ancora aperto. Proprietà: (i) *condivisione reale* dei dati; (ii) semantica robusta alla rinomina/spostamento all'interno dello stesso file system; (iii) hard link di directory vietati (salvo . e ..) per preservare l'acyclicità; (iv) hard link *non* attraversano i confini di file system.

**Approccio con soft link (link simbolici).** Un *link simbolico* è un'entry speciale che **contiene un pathname** verso il bersaglio. Pro: può attraversare file system diversi, non richiede condividere l'i-node, è leggero da creare/eliminare. Contro: può diventare **dangling** (se il bersaglio è rimosso o rinominato), può introdurre **cicli** nei percorsi (mitigati da limiti sull'inseguimento), e aggiunge un passo di risoluzione a ogni accesso. Le autorizzazioni si applicano al file di destinazione; il link in sé ha metadati propri.

#### 4.3.7 Gestione dei blocchi (spazio su disco)

La *gestione dello spazio su disco* riguarda come il file system tiene traccia dei **blocchi liberi** e come li **assegna/recupera** in modo efficiente e consistente. Gli obiettivi sono: trovare rapidamente spazio libero, favorire la località (blocchi vicini tra loro), ridurre frammentazione e garantire il ripristino dopo crash.

**Bitmap (mappa di bit).** Si associa *un bit per blocco*: 0 = libero, 1 = occupato (o viceversa). Un disco con  $n$  blocchi però richiede una bitmap con  $n$  bit, quindi nel caso di un disco da 1 TB serve una mappa da 1 miliardo di bit.

**Liste concatenate e contatori.** Alternativa storica alla bitmap: mantenere una *lista dei blocchi liberi*; per ridurre la lunghezza della lista si usano **contatori di estensioni** (*extents* di blocchi consecutivi) memorizzando coppie ⟨inizio, lunghezza⟩. Pro: ottima quando l'allocazione e la liberazione avvengono a grandi intervalli contigui; meno spazio per metadati rispetto a una lista “uno a uno”. Contro: ricerca di uno spazio adatto può richiedere scansioni lineari; la lista può frammentarsi col tempo; più complessa la manutenzione in crash-recovery rispetto a una bitmap, inoltre occupa 32 bit per ogni blocco.

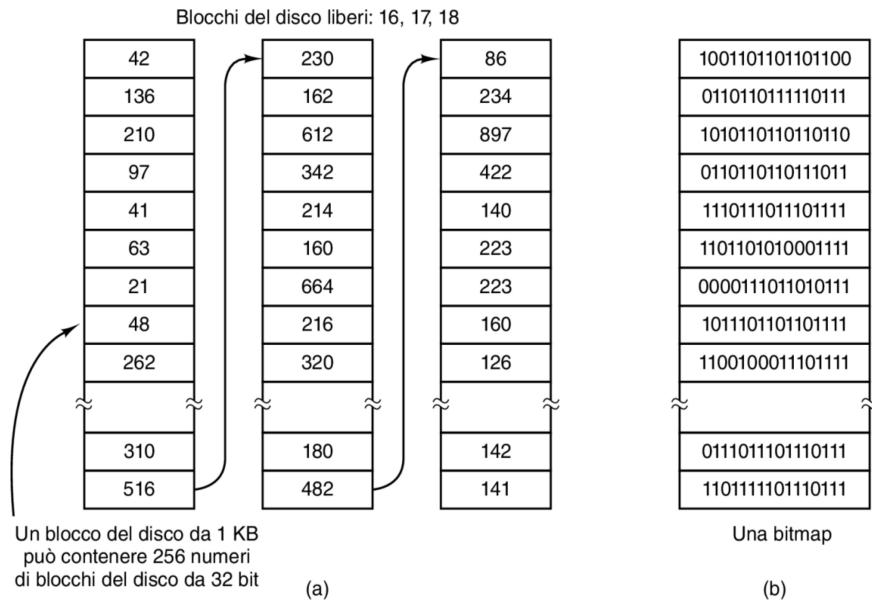


Figura 4.10: (a) Memorizzazione della free list in una lista concatenata. (b) Una bitmap.

**Quote del disco.** Per impedire agli utenti di occupare troppo spazio, i sistemi operativo multiutente forniscono spesso un meccanismo per imporre le quote del disco. L'idea è che l'amministratore di sistema assegna a ciascun utente un numero massimo di file e blocchi e che il sistema operativo si accerti che queste quote non vengano superate. Quindi una seconda tabella contiene i record delle quote di ogni utente che un file attualmente aperto, anche se il file è stato aperto da qualcun altro.

#### Dove tenere le strutture (strategie di allocazione dei metadati).

- **Tutto in memoria:** caricare l'intera bitmap/indice in RAM accelera le decisioni di allocazione; al commit si scrivono solo i blocchi modificati. Pro: velocità massima; Contro: consumo di RAM e necessità di politiche di scrittura sicure.
- **Un blocco alla volta:** leggere/scrivere on demand la porzione necessaria (es. lo “span” di bitmap del gruppo locale). Pro: meno RAM; Contro: più I/O piccoli e latenza nelle ricerche.
- **Paginata con VM (Virtual Memory):** trattare bitmap/indici come pagine cache del kernel (page cache/buffer cache), lasciando al sistema di memoria virtuale prefetch, write-back e sostituzione. Pro: bilanciamento automatico tra RAM e I/O; Contro: maggiore complessità di interazione FS-VM.

**Politiche di allocazione.** Indipendentemente dalla rappresentazione, il FS tende a scegliere blocchi *vicini* all’inode/ai blocchi già allocati (*località*), usando first/next/best fit su insiemi ristretti (gruppi/cluster allocation).

#### 4.3.8 Controlli di consistenza

A seguito di un **crash** del sistema i file-system potrebbero diventare inconsistenti. Per affrontare questo problema molti computer hanno un’utility che verifica la coerenza (fsck, sfc) e esistono due tipologie di controllo che si possono eseguire: quello dei blocchi e dei file.

**Controllo dei blocchi.** In questo caso il programma costruisce due tabelle, ciascuna contenente un contatore per ogni blocco inizialmente impostato a 0. I contatori della prima tabella tengono traccia di quante volte ogni blocco è presente in un file e i contatori nella seconda registrano quanto spesso ogni blocco sia presente nella lista (o bitmap) dei blocchi liberi.

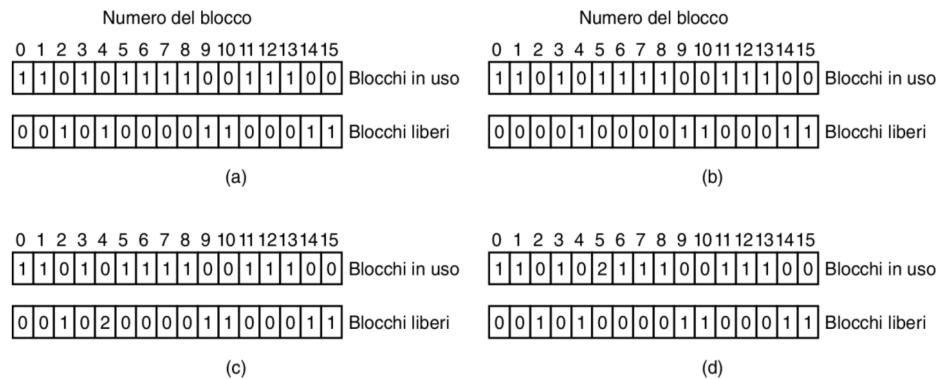


Figura 4.11: Stati del file system. (a) Coerente. (b) Blocco mancante. (c) Blocco duplicato nella lista dei blocchi liberi. (d) Blocco di dati duplicato.

**Controllo dei riferimenti agli i-node.** Si costruisce una tabella con un contatore per ogni i-node allocato, inizialmente a 0. Percorrendo *tutte* le directory, per ogni entry si incrementa il contatore dell’i-node referenziato. Al termine si confronta il conteggio osservato con il *link count* memorizzato nell’i-node: se differiscono, si propone la correzione (aggiornare il link count al valore reale). Gli i-node con conteggio 0 ma marcati *allocati* sono *orfani*: si offre la scelta tra liberarli (restituendo i blocchi allo spazio libero) o collegarli in una directory di recupero (es. *lost+found*). Si verifica inoltre che ogni entry punti a un i-node valido/allocato, che le directory contengano le voci . e .. corrette, e che la struttura resti aciclica (nessun hard link a directory, eccetto . e ..).

**Controllo dei file.** Per ogni i-node di tipo file si attraversano i puntatori diretti/indiretti verificando che i numeri di blocco siano nel range del volume, non riservati e coerenti con il controllo dei blocchi (niente duplicati). Si controlla la consistenza tra *dimensione del file* e numero di blocchi effettivamente allocati (eventuale *truncate* o correzione dei puntatori invalidi); i riferimenti fuori range o corrotti vengono azzerati e i blocchi eventualmente rilasciati.

#### 4.3.9 Journaling

Aggiornare un file system richiede spesso modificare *più blocchi* (bitmap dello spazio libero, inode, directory, ecc.). Se il sistema si arresta tra le scritture, lo stato on-disk può diventare *incoerente* (blocchi

“persi”, riferimenti spezzati). Il **journaling** risolve il problema registrando in anticipo (*write-ahead*) le operazioni critiche in un *log* sequenziale (il *journal*), così che, al riavvio, basti *riprodurre* (replay) le operazioni completate e *scartare* quelle parziali, evitando lunghe scansioni (fsck) dell’intero volume.

**Funzionamento.** Il file system journaled scrive inazitutto una voce di log che elenca le tre azioni da completare. La voce di log viene poi scritta su disco e solo dopo la scrittura cominciano le varie operazioni. Dopo che le operazioni si sono concluse con successo, la voce di log viene cancellata e se il sistema ha un crash, durante il ripristino il file system può verificare il log per vedere se ci sono operazioni in attesa. Per far sì che il journaling funzioni, però, le operazioni devono essere **idempotenti**, ossia possono essere ripetute tutte le volte che servono senza fare danni (es. "Aggiornare la bitmap per contrassegnare l’inode  $k$  o il blocco  $n$  come liberi").

**Transazioni.** Le modifiche strutturali sono raggruppate in **transazioni atomiche** (concetto simile alle basi di dati): un gruppo di azioni può essere inserito tra le operazioni `begin transaction` e `end transaction` ed il file system sa che deve completare tutte le operazioni fra parentesi senza interrompersi.

#### 4.3.10 Uso della cache

La tecnica più comunemente utilizzata per ridurre i tempi di accesso alla memoria secondaria è la **block cache** o **buffer cache**. In questo contesto, una cache è una raccolta di blocchi appartenenti logicamente al disco, ma che sono tenuti in memoria per ragioni di prestazioni.

**Gestione della cache.** Per la gestione della cache si possono usare diversi algoritmi, il più comune è quello che controlla tutte le richieste di lettura per verificare se il blocco necessario sia nella cache e, se lo è, la richiesta di lettura è soddisfatta senza accedere al disco (molto più veloce). Se non lo è, per prima cosa viene letto dal disco e portato nella cache e quindi copiato ovunque ve ne sia bisogno, così che le successive richieste possano essere soddisfatte dalla cache.

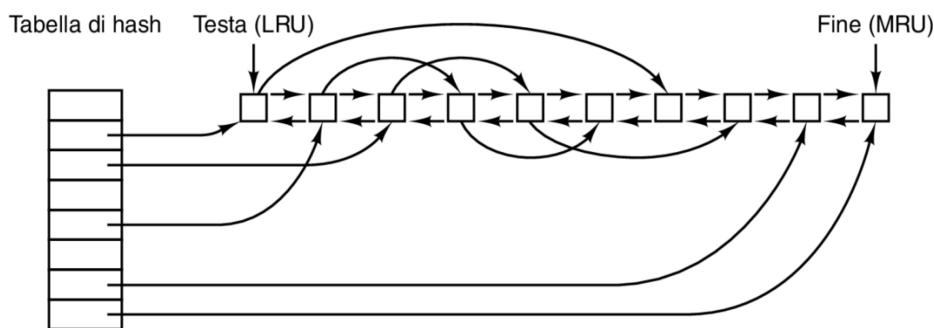


Figura 4.12: Struttura dati della cache

**Algoritmi utilizzati.** Il metodo abituale, per controllare che un blocco sia in cache, è generare un valore hash a partire dal dispositivo e dall’indirizzo su disco e cercare il risultato in una tabella hash (di tipo concatenato). Su questa memoria si potrebbero utilizzare LRU (e anche quelli visti nella paginazione), ma c’è un problema: se un blocco critico, come un blocco di i-node, viene letto nella cache e modificato ma non riscritto su disco, un crash del file system lo lascerebbe in uno stato incoerente. Allora ci si pongono due domande:

1. E' probabile che presto il blocco servirà ancora?
2. Il blocco è fondamentale per la coerenza del sistema?

Nella pratica si adotta una *LRU modificata*, così da non scegliere come vittime i blocchi "critici" per la coerenza (es. metadati come i-node/directory(bitmap) e privilegiare invece blocchi dati ordinari quando è possibile.

**Sincronizzazione con il disco.** In questo problema, Windows e UNIX hanno due approcci diversi:

- **UNIX** offre una chiamata di sistema sync che sincronizza la cache con il disco e viene chiamata circa ogni 30 secondi.
- **Windows** invece ha un tipo di cache chiamata **write-through** che sincronizza immediatamente ogni blocco scritto con ogni blocco nel disco, questo è utile perché in caso di crash non si perde niente però richiede molto più I/O.

#### 4.3.11 Sistemi operativi moderni

Nei sistemi reali i file system combinano affidabilità, prestazioni e funzioni avanzate (journaling o copy-on-write, indicizzazione efficiente delle directory, grandi volumi/file, strumenti di controllo e ripristino). Di seguito una sintesi dei più diffusi.

**exFAT (Microsoft).** Pensato per memorie flash/SD e interscambio tra piattaforme: supporta file e volumi molto grandi, struttura semplice (bitmap dello spazio libero, directory con nomi lunghi), assenza di journaling per ridurre scritture e overhead. Puntato a portabilità e basso costo computazionale, non alle garanzie di un FS journaled.

**NTFS (Windows).** File system journaled con *Master File Table* (MFT) che contiene metadati e, per file piccoli, dati *residenti*. Supporta ACL dettagliate, compressione, link simbolici/hard link, *reparse points*, *alternate data streams*. Il journaling dei metadati velocizza il ripristino post-crash; le directory sono indicizzate ( $B^+$ -tree).

**ext4 (Linux).** Evoluzione di ext2/3 con journaling (tipicamente modalità *ordered*), *extents* per ridurre la frammentazione e migliorare file grandi, allocazione differita, directory indicizzate (*htree*), timestamp estesi e volumi/file di grandi dimensioni. Buon compromesso tra compatibilità, stabilità e prestazioni.

**Btrfs (Linux).** File system *copy-on-write* per dati e metadati: snapshot e *clone* efficienti, checksum end-to-end, *subvolume*, *send/receive*, bilanciamento e profili RAID integrati a livello FS. Progettato per integrità e gestione avanzata, con costo di scrittura tipico dei CoW e tuning richiesto per carichi specifici.

**macOS (HFS+, APFS).** Storicamente HFS+ (journaled, catalogo a B-tree, case-insensitive per default). Oggi APFS è lo standard: design *copy-on-write* con *snapshots*, *clones* di file/dir, *space sharing* tra volumi, cifratura a più chiavi, ottimizzazioni per SSD (TRIM, allocazioni rapide). Obiettivo: integrità, funzionalità moderne e buona efficienza su storage flash.

## 4.4 Dischi

### 4.4.1 Architettura di un disco

I dischi rigidi magnetici sono caratterizzati dal fatto che le operazioni di lettura e di scrittura sono ugualmente veloci, rendendoli l'ideale come memoria secondaria. Sono organizzati in cilindri, ciascuno contenente tante tracce quante sono le testine impilate verticalmente, le tracce sono divise in settori e il numero dei settori lungo la circonferenza raggiunge generalmente varie centinaia, il numero di testine invece varia da 1 a 16. I dischi più vecchi avevano poca elettronica e fornivano un semplice flusso di bit seriale e il controller eseguiva la maggior parte del lavoro. Oggi, in particolare sui **SATA**, l'unità disco contiene un microcontrollore che svolge considerevole parte del lavoro e permette al controller vero e proprio di inviare un insieme di comandi ad alto livello.

### 4.4.2 Frammentazione

La **frammentazione** (esterna) nei dischi si verifica quando i file non riescono a essere memorizzati in *settori contigui*. In condizioni ideali, un file viene scritto in blocchi adiacenti: in questo modo la testina dell'HDD non deve spostarsi su altre tracce, ma basta una semplice rotazione dei piatti per leggere (o scrivere) l'intero file (ove possibile) in maniera sequenziale. Se invece lo spazio libero è frammentato, (un po' come succede con i processi in RAM) il file viene suddiviso in più parti distribuite in zone diverse del disco. In questo caso, l'unità HDD deve effettuare più movimenti meccanici per accedere ai vari frammenti, con un notevole rallentamento delle prestazioni.

Difatto questo fenomeno è particolarmente penalizzante negli HDD, perché la differenza di throughput tra operazioni sequenziali e casuali è molto elevata. Negli SSD, invece, non essendoci parti meccaniche, l'accesso casuale è solo leggermente più lento di quello sequenziale. L'operazione che permette di ricompattare i file, riordinandoli in settori contigui, si chiama **deframmentazione**.

### 4.4.3 Scheduling del disco

Quando sono presenti più richieste concorrenti, il sistema operativo può ordinarle per migliorare il throughput e ridurre i tempi d'attesa, mantenendo al contempo un livello accettabile di equità. Il sistema operativo può adoperare diverse politiche e queste influiscono ottimizzando per **tempo di posizionamento** (seek-time) e **latenza di rotazione**. Di seguito le politiche classiche, illustrate sullo stesso scenario di riferimento: coda (per numero di cilindro)  $\langle 98, 183, 37, 122, 14, 124, 65, 67 \rangle$ , posizione iniziale della testina 53.

**FCFS (First-Come, First-Served).** Serve le richieste nell'ordine d'arrivo: è semplice ed “equo”, ma non ottimizza i movimenti della testina e può risultare inefficiente. Sullo scenario d'esempio produce molta strada complessiva percorsa (ordine invariato: 98, 183, 37, 122, 14, 124, 65, 67).

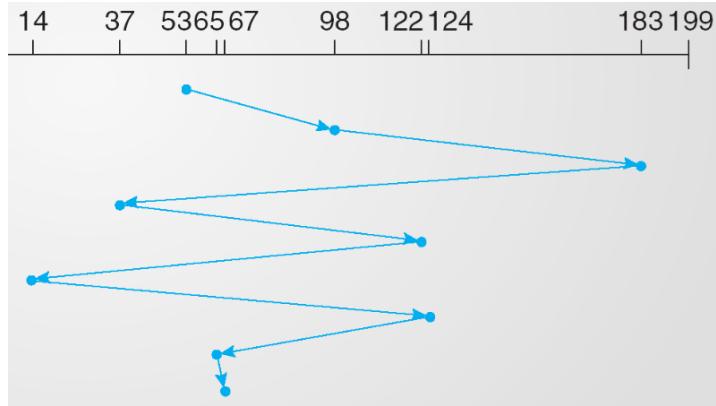


Figura 4.13: Esempio con algoritmo FCFS.

**SSTF (Shortest Seek Time First).** Seleziona la richiesta più vicina alla posizione corrente della testina, minimizzando il prossimo spostamento. Sullo stesso esempio l'ordine diventa 65, 67, 37, 14, 98, 122, 124, 183, con una riduzione marcata della distanza totale; ottime prestazioni medie, ma rischio di *starvation* per richieste lontane.

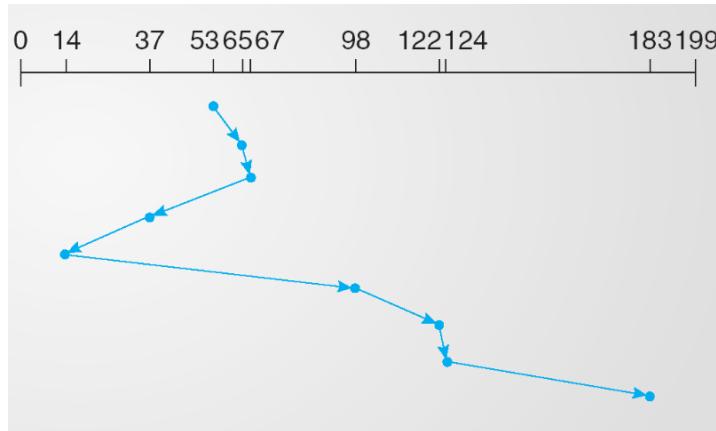


Figura 4.14: Esempio con algoritmo SSTF.

**SCAN (algoritmo dell'ascensore).** La testina “scansiona” in un verso servendo tutte le richieste incontrate, poi inverte direzione alla fine (come un ascensore). Garantisce un limite superiore al tempo d’attesa ed una distribuzione più uniforme rispetto a FCFS; nell’esempio (partendo “in su”) l’ordine è 65, 67, 98, 122, 124, 183, 37, 14.

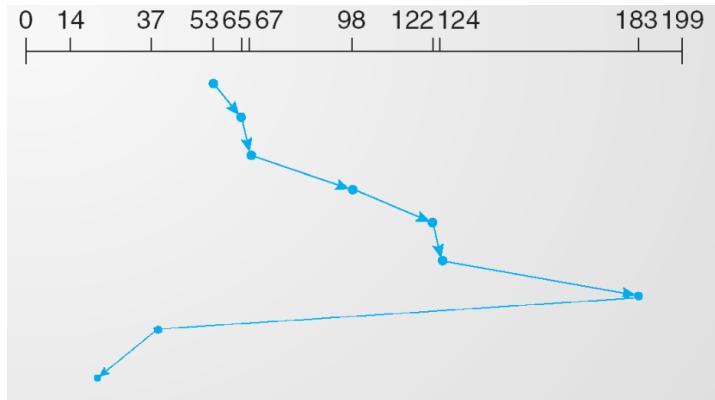


Figura 4.15: Esempio con algoritmo dell'ascensore (SCAN).

**C-SCAN (SCAN circolare).** Considera i cilindri come disposti in modo circolare: avanza in un solo verso servendo le richieste; raggiunta l'estremità, ritorna all'inizio *senza* servire richieste, e riparte nello stesso verso. Offre tempi d'attesa medi più regolari ad alto carico; sullo stesso esempio l'ordine è 65, 67, 98, 122, 124, 183, 14, 37. In pratica, si preferisce C -SCAN con carichi elevati, mentre a carichi bassi SCAN o anche SSTF possono risultare competitivi.



Figura 4.16: Esempio con algoritmo C-SCAN

#### 4.4.4 RAID

I **Redundant Array of Independent Disks (RAID)** nascono per aumentare le prestazioni dell'I/O su disco sfruttando il *parallelismo*: i dati di una stessa unità logica (file o volume) vengono *distribuiti* su più dischi (*striping*) in modo trasparente all'utente. Poiché più dischi implicano una maggiore probabilità di guasto del volume logico, ai benefici prestazionali si affianca *ridondanza* per migliorare l'affidabilità; sono possibili realizzazioni via *hardware* (trasparenti al S.O.) o via *software* (con carico sulla CPU) e la sostituzione automatica con dischi *spare*.

##### 4.4.4.1 RAID 0

Il RAID 0 consiste nel vedere un singolo disco virtuale (formato in realtà da  $n$  dischi) che viene suddiviso in *strip* di  $k$  settori ciascuna, con la strip 0 costituita dai settori da 0 a  $k - 1$ , la strip 1 formata da  $k$  a  $2k - 1$  e così via (quindi per  $k = 1$  ogni strip è un settore, per  $k = 2$  una strip è costituita da 2 settori, ecc ...). L'organizzazione di questo RAID scrive strip consecutive sulle unità in modalità round

robin, come si vede nella figura 4.17. Il RAID di tipo 0 aumenta del doppio l'efficienza, ma elimina completamente la ridondanza (bassa tolleranza ai guasti).

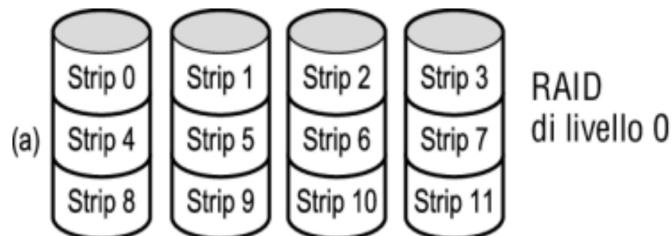


Figura 4.17: RAID di tipo 0, 4 dischi con dati divisi equamente in ciascuno.

#### 4.4.4.2 RAID 1

Il RAID 1 consiste nella duplicazione di tutti i dischi, come si vede in figura 4.18 (quindi con 2 dischi, 1 sarebbe "l'originale" e uno la "copia"). In scrittura ogni strip è **scritta due volte**, ma in lettura può essere usata qualunque copia, distribuendo il carico su più unità. Di conseguenza, le prestazioni della scrittura peggiorano ma la **fault tollerance** (tolleranza ai guasti) aumenta.

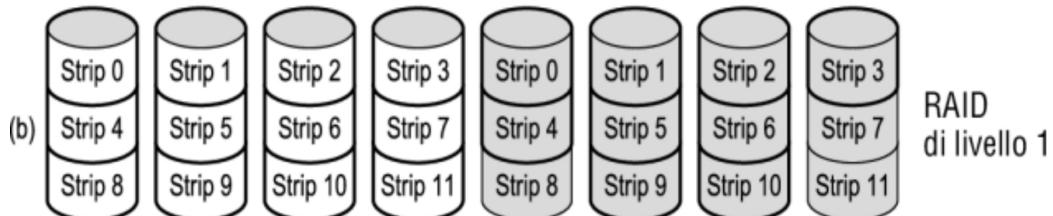


Figura 4.18: RAID di tipo 1, 8 dischi con 4 dischi originali e 4 copie.

#### 4.4.4.3 RAID 2

Il RAID 2, diversamente dai RAID 0 e 1 che operano in strip di settori, opera sulla **base delle parole** ed eventualmente anche sulla base dei byte. Si può immaginare che ciascun byte del singolo disco virtuale sia suddiviso in una coppia di due parti da 4 bit e quindi sia aggiunto a ciascuno un codice di Hamming a formare una parola di 7 bit in cui i bit 1, 2, e 4 siano i bit di parità. Si immagini inoltre che le sette unità della figura 4.19 siano sincronizzate in termini di posizione del braccio e di rotazione. Sarebbe quindi possibile scrivere la parola di 7 bit codificata con il codice di Hamming sulle sette unità, un bit ciascuna. In generale, il RAID di tipo 2 divide i vari bit nei dischi, aggiungendo la ridondanza del codice di Hamming per garantire sicurezza dei dati, questo crea un altro **throughput di scrittura e lettura** ma anche un grande **overhead di capacità** e un'ottima **fault tollerance**.

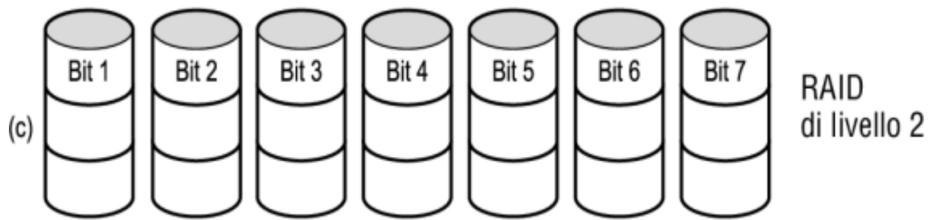


Figura 4.19: RAID di tipo 2, sette dischi con un 1 bit per ciascuno da una parola di 7 bit.

#### 4.4.4.4 RAID 3

Il RAID di livello 3 è una versione semplificata del RAID di livello 2, viene creato un singolo bit di parità per ogni parola di dati che viene scritto in un'unità di parità. Come nel RAID 2, le unità devono essere perfettamente sincronizzate perché le singole parole dati sono distribuite su molteplici unità. Questo sistema non riesce a correggere esattamente errori casuali nei dischi, ma riesce tuttavia a correggere errori **dovuti a crash**, poiché la posizione del bit errato è nota.

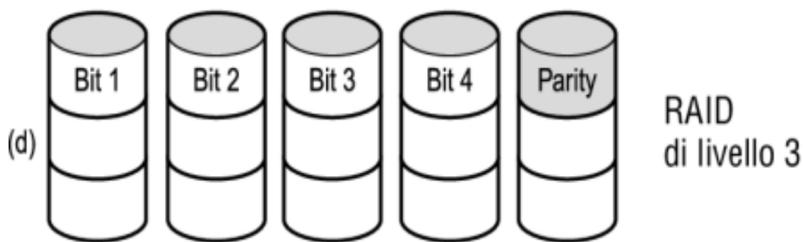


Figura 4.20: RAID di tipo 3, 5 dischi con 4 dischi usati per memorizzazione e 1 disco usato per parità.

#### 4.4.4.5 RAID 4

Diversamente dai RAID 2 e 3, il RAID 4 è suddiviso in strip come il RAID 0 e i dati vengono distribuiti nei vari dischi *dati* e un **disco separato** conserva, per ogni strip, il blocco di parità calcolato come **XOR** dei blocchi dati nella riga (come si vede in figura 4.21). Questo schema protegge dalla perdita di un'unità, ma ha prestazioni scadenti per piccoli aggiornamenti, al contrario invece ha ottime prestazioni per letture/scrittura sequenziali di intere strip. Il più grande difetto è che il disco di parità ha un altissimo carico di lavoro e questo potrebbe creare un **collo di bottiglia**.

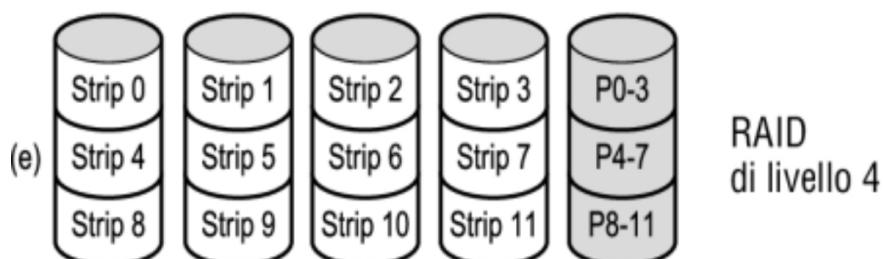


Figura 4.21: RAID di tipo 4, 4 dischi dati e 1 disco di parità calcolato come XOR dei precedenti.

#### 4.4.4.6 RAID 5

Il RAID 5 risolve il problema del collo di bottiglia causato dal RAID 4 distribuendo i bit di parità uniformemente su tutte le unità, in stile round-robin (come si vede in figura 4.22). La ricostruzione di un'unità durante un eventuale crash tuttavia è un processo altamente complesso e dispendioso.

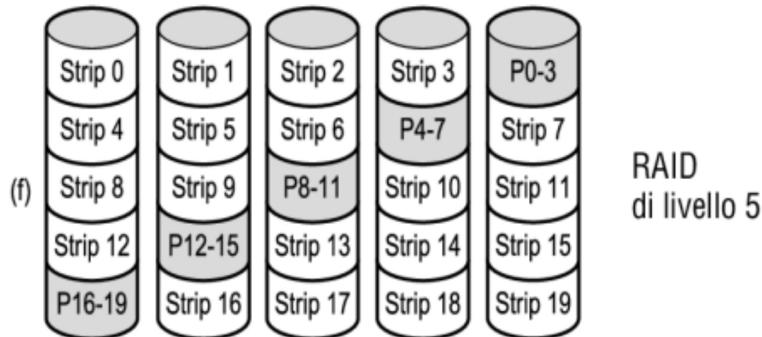


Figura 4.22: RAID di tipo 6, 5 dischi di dati, divisi con 1 strip di parità.

#### 4.4.4.7 RAID 6

Il RAID 6 è molto simile al RAID 5, aggiunge però un ulteriore blocco di parità nei dischi. Quindi si fa uno striping dei dati su tutti i dischi con due blocchi di parità invece di uno solo. Come risultato, le operazioni di scrittura sono un po' costose a causa dei calcoli di pariàt, ma le letture non hanno svantaggi prestazionali. Inoltre offre una maggior affidabilità (se RAID 5 dovesse incontrare un blocco corrotto mentre ricostruisce l'unità il blocco da ricostruire sarebbe perso).

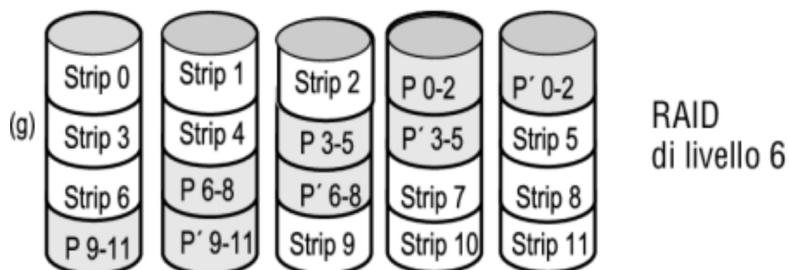


Figura 4.23: RAID di tipo 6, 5 dischi di dati, divisi con 2 strip di parità.

## 4.5 SSD

I dispositivi a stato solido basati su memoria *flash* (tipicamente tecnologia *NAND*) hanno caratteristiche fisiche diverse dai dischi magnetici. Le **lettura** sono in genere molto più rapide delle **scritture**; inoltre, prima di poter riscrivere del contenuto è necessario **cancellare** l'area interessata. La cancellazione non avviene a livello di singola pagina, ma a livello di **blocco di cancellazione** (erase block), composto da più **pagine** che sono l'unità minima di lettura/scrittura. Ogni blocco sopporta un **numero limitato di cicli** di programmazione/cancellazione: questo impone di distribuire nel tempo le scritture (usura).

#### 4.5.1 Organizzazione (pagine e blocchi).

L'indirizzamento logico visto dal sistema operativo è mappato su pagine fisiche raggruppate in blocchi. Si leggono/scrivono *pagine*; si cancella un *blocco* intero. L'aggiornamento “in place” non è possibile: un dato modificato viene scritto in una *nuova* pagina libera e la vecchia viene marcata come *invalidata* fino alla successiva cancellazione del blocco.

#### 4.5.2 Implicazioni per il file system.

I file system classici sono stati progettati assumendo dispositivi riscrivibili in place e penalizzano questi vincoli. Per i supporti flash sono nati FS *ad hoc* (ad es. *Flash-Friendly File System*, F2FS e vari FS di tipo *log-based*) che limitano gli aggiornamenti casuali, privilegiano scritture sequenziali e tengono conto dell'usura.

#### 4.5.3 Flash Translation Layer (FTL).

Molti dispositivi espongono al sistema operativo una normale interfaccia a blocchi e demandano al *controller* interno la rimappatura *logico*→*fisico*. Il **FTL** gestisce: scritture fuori posto, coalescenza dei validi, cancellazioni di blocchi, e politiche di *wear leveling* per distribuire l'usura.

#### 4.5.4 Garbage collection e comando TRIM.

Poiché gli aggiornamenti producono pagine invalidate, il dispositivo esegue **garbage collection**: copia le pagine ancora valide fuori da un blocco, cancella il blocco e lo riporta nell'elenco dei liberi. Se il sistema operativo non collabora, le prestazioni degradano nel tempo (più copie e cancellazioni). Il comando **TRIM** permette all'OS di segnalare al dispositivo i blocchi logici non più in uso: il controller potrà così riciclarli senza doverli prima leggere/copiare, riducendo *write amplification* e mantenendo costanti le prestazioni. L'uso efficace di GC e TRIM richiede un adeguamento del file system e della sua politica di allocazione per supporti flash.

#### 4.5.5 Accessi sequenziali e casuali negli SSD

**Lettura.** Negli SSD la differenza di prestazioni tra accessi sequenziali e casuali è relativamente ridotta. Tuttavia, gli accessi casuali richiedono di individuare più blocchi rispetto a quelli sequenziali, introducendo quindi un overhead aggiuntivo. I tempi di lettura delle singole pagine rimangono invece pressoché identici.

**Scrittura.** La scrittura è più complessa a causa del wear leveling. In condizioni normali, la maggior parte delle pagine contiene già dati (utilizzati o meno). Ciò significa che, per scrivere una nuova pagina, spesso è necessario cancellare l'intero blocco e riscriverlo. Durante questa operazione vengono riscritte sia la nuova pagina sia le pagine ancora valide, mentre quelle segnate per la cancellazione vengono ignorate. Altre informazioni utili sono:

1. **Scritture sequenziali:** il blocco viene cancellato e riscritto con le nuove pagine in modo relativamente efficiente.
2. **Scritture casuali:** se ogni nuova pagina si trova in un blocco diverso, l'operazione di cancellazione/riscrittura deve essere ripetuta più volte, con un costo molto maggiore in termini di prestazioni.

Difatti a causa di questa particolarità i tempi di scrittura degli SSD tendono ad essere molto più lenti di quelli di lettura.



# Capitolo 5

## Esercizi

Questa è una raccolta di esercizi di *esempio* degli argomenti visti in queste dispense.

### 5.1 File system

#### 5.1.1 FAT

**Consegna.** Supponiamo di avere un file-system che utilizza una *File Allocation Table* (FAT) come segue e che nella cartella radice il file `pippo.txt` punti al blocco iniziale 7.

indice	FAT[indice]	nome	primo blocco
1	4		
2	3		
3	15		
4	5		
5	10	pippo.txt	7
6	12		
7	1		...
8	2		
9	3		
10	-1		
	...		

Indicare:

1. in quale **numero di blocco** del disco ricade l'*offset* 10100 del file `pippo.txt`;
2. la **dimensione minima presunta** del file (in byte), sapendo che la dimensione di blocco è 4 KiB = 4096 B.

**Soluzione.**

1. **Catena dei blocchi del file.** Dalla FAT e dal primo blocco 7 si ottiene

$$7 \rightarrow 1 \rightarrow 4 \rightarrow 5 \rightarrow 10 \rightarrow \text{fine} (-1).$$

2. **Blocco contenente l'offset** 10100. Con dimensione di blocco  $B = 4096$  B,

$$k = \left\lfloor \frac{10100}{4096} \right\rfloor = 2.$$

L'offset cade dunque nel  $k$ -esimo blocco del file a base zero, cioè nel **terzo blocco** della catena ⇒ **numero di blocco** = 4.

3. **Dimensione minima presunta del file.** La catena ha 5 blocchi. La dimensione minima compatibile è quella in cui i primi 4 blocchi sono pieni e l'ultimo contiene almeno 1 byte:

$$\min | \text{pippo.txt} | = 4 \cdot 4096 + 1 = \boxed{16385 \text{ byte}}.$$

(Per completezza, la dimensione massima con 5 blocchi è  $5 \cdot 4096 = 20480$  byte.)

### 5.1.2 File system UNIX con i-node

**Consegna.** Consideriamo un file-system UNIX basato su i-node: l'i-node di un file contiene 13 puntatori ai blocchi su disco (10 diretti, 1 indiretto singolo, 1 indiretto doppio, 1 indiretto triplo). La dimensione di blocco è 4 KiB e gli indirizzi di blocco occupano 4 byte.

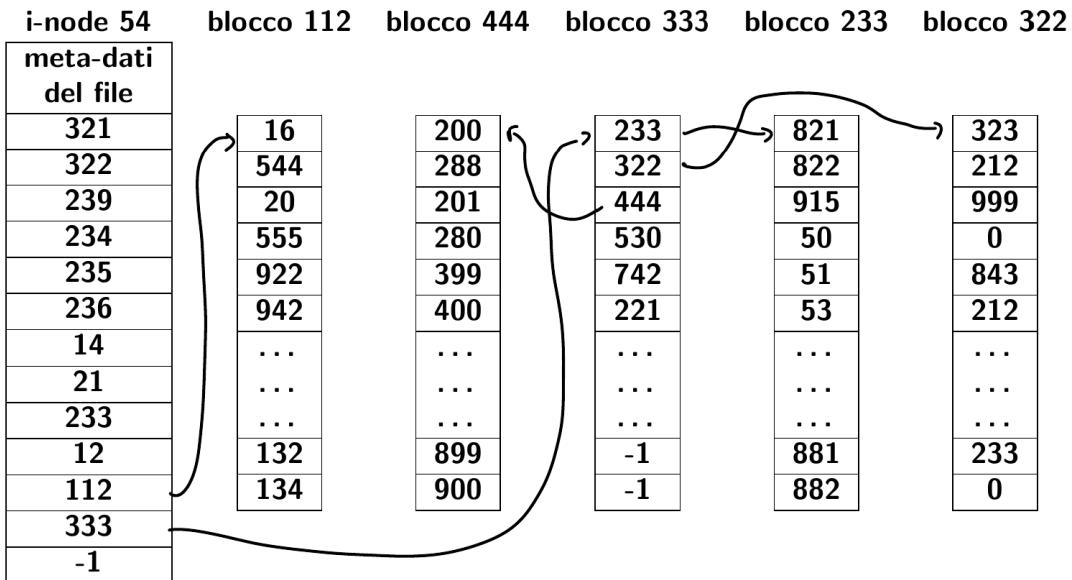


Figura 5.1: Schema riassuntivo dell'i-node e dei blocchi (diretti, indiretto singolo e doppio).

L'i-node del file contiene i seguenti puntatori:

- **Diretti (10):** 321, 322, 239, 234, 235, 236, 14, 21, 233, 12.
- **Indiretto singolo:** 112.
- **Indiretto doppio:** 333.
- **Indiretto triplo:** -1 (assente).

Contenuto dei blocchi di indirizzi mostrati in figura:

- **Blocco 112** (indiretto singolo) → 16, 544, 20, 555, 922, 942, ..., 132, 134.
- **Blocco 333** (indiretto doppio) → 233, 322, 444, 530, 742, 221, ..., -1, -1.

- **Blocco 233** (secondo livello) → 821, 822, 915, 50, 51, 53, ..., 881, 882.
- **Blocco 444** (secondo livello) → 200, 288, 201, 280, 399, 400, ..., 899, 900.
- **Blocco 322** (secondo livello) → 323, 212, 999, 0, ..., 233, 0.

**Richiesta.** Determinare in quale numero di blocco ricadono i byte di alcuni offset (es. x, y, z) del file referenziato dall'i-node. Gli offset sono espressi in byte e partono da 0.

**Dati utili.** Un blocco contiene 4096 byte; un indirizzo occupa 4 byte, quindi un blocco di indirizzi contiene 1024 voci. I primi 10 blocchi del file si raggiungono tramite puntatori diretti; i successivi tramite l'indiretto singolo, poi il doppio indiretto e infine il triplo (se presente).

## 5.2 Dischi

### 5.2.1 Scheduling su disco con politica LOOK

**Consegna.** Disco con 200 tracce (0–199), velocità di seek: 1 traccia/ms. Al tempo  $t = 0$  la testina è su traccia 100 (richiesta appena servita). Richieste già in coda:  $\{50, 115, 180\}$ . Arrivi successivi:  $t = 70 \Rightarrow 150$ ,  $t = 130 \Rightarrow 90$ . Usare **LOOK** con direzione iniziale *ascendente* (verso le tracce più alte). Trascurare latenza rotazionale e tempo di trasferimento.

**Sequenza LOOK.**

$$100 \rightarrow 115 \rightarrow 180 \rightarrow 150 \rightarrow 90 \rightarrow 50.$$

**Calcolo dei tempi (1 traccia = 1 ms).**

Passo	Spostamento	Incremento [ms]	Cumulato [ms]
$100 \rightarrow 115$	$ 115 - 100  = 15$	15	15
$115 \rightarrow 180$	$ 180 - 115  = 65$	65	80
$180 \rightarrow 150$	$ 150 - 180  = 30$	30	110
$150 \rightarrow 90$	$ 90 - 150  = 60$	60	170
$90 \rightarrow 50$	$ 50 - 90  = 40$	40	210

**Risultato.** Tempo di ricerca complessivo:

$$15 + 65 + 30 + 60 + 40 = \boxed{210 \text{ ms}}.$$

**Nota.** Con **SCAN** (ascendente) la testina raggiungerebbe il bordo superiore (traccia 199) prima di invertire, comportando un tempo maggiore.

## 5.3 Thread

### 5.3.1 Attesa con semafori

**Consegna.** Determinare l'output del processo P3 assumendo che il valore iniziale di x è 1 e che i 3 semafori abbiano i seguenti valori iniziali: S=1, R=0, T=0.

**Codice.**

P1 :	P2 :	P3 :
wait(S)	wait(R)	wait(T)
$x = x - 2$	$x = x + 2$	if ( $x < 0$ ) signal(R)
signal(T)	signal(T)	wait(T)
wait(S)	wait(R)	print(x)
$x = x - 1$		
signal(T)		

**Stato iniziale.**  $x = 1, S = 1, R = 0, T = 0$ .

**Ordine di esecuzione.**

- (1) **P1** esegue `wait(S)`  $\Rightarrow S = 0$ .
- (2) **P1** esegue  $x \leftarrow x - 2 \Rightarrow x = -1$ .
- (3) **P1** esegue `signal(T)`  $\Rightarrow T = 1$ .
- (4) **P3** esegue `wait(T)`  $\Rightarrow T = 0$ .
- (5) **P3** trova  $x < 0 \Rightarrow \text{signal}(R) \Rightarrow R = 1$ .
- (6) **P2** sblocca `wait(R)`  $\Rightarrow R = 0$ .
- (7) **P2** esegue  $x \leftarrow x + 2 \Rightarrow x = 1$ .
- (8) **P2** esegue `signal(T)`  $\Rightarrow T = 1$ .
- (9) **P3** esegue il secondo `wait(T)`  $\Rightarrow T = 0$ .
- (10) **P3** esegue `print(x)`  $\Rightarrow$  stampa 1.
- (11) (Poi **P1** tenta il secondo `wait(S)` e resta bloccato; **P2** tenta il secondo `wait(R)` e resta bloccato.  
Non influisce sull'output.)

**Output di P3.**  $x = \boxed{1}$ .

## 5.4 Paginazione

### 5.4.1 Numero di pagine virtuali

**Consegna.** Sistema con indirizzi virtuali a 48 bit, indirizzi fisici a 32 bit e dimensione di pagina  $P = 4\text{ KB}$ . Determinare il **numero di pagine virtuali** dello spazio di indirizzamento (e, facoltativamente, il numero di frame fisici).

**Soluzione.** Poiché  $P = 4\text{ KB} = 2^{12}$ , i bit di offset sono 12.

$$\text{pagine virtuali} = 2^{48-12} = 2^{36} = \boxed{68,719,476,736}.$$

(Opzionale) Con indirizzo fisico a 32 bit:

$$\text{frame fisici} = 2^{32-12} = 2^{20} = \boxed{1,048,576}.$$

### 5.4.2 Numero di pagine virtuali con tabella multilivello

**Consegna.** Sistema con indirizzi virtuali a 32 bit e dimensione di pagina  $P = 4 \text{ KB}$ . Determinare, per una tabella delle pagine a **2 livelli** ( $k = 2$ ), il numero di pagine necessarie per rappresentare l'intero spazio di indirizzi. Assumere dimensione di una entry della page table pari a 4 byte.

**Soluzione.** Con  $P = 4 \text{ KB} = 2^{12}$  si hanno 12 bit di offset e  $32 - 12 = 20$  bit di numero di pagina virtuale. Una pagina di tabella contiene

$$\frac{4096}{4} = 1024 = 2^{10} \text{ entry.}$$

Con due livelli, si usa tipicamente una suddivisione 10 bit per il livello 1 e 10 bit per il livello 2:

- Tabella di *livello 1*:  $2^{10} = 1024$  entry  $\Rightarrow$  sta in 1 pagina.
- Tabelle di *livello 2*: una per ciascuna entry L1, quindi  $2^{10} = 1024$  tabelle  $\Rightarrow$  1024 pagine.

Quindi, per coprire l'intero spazio virtuale (tutte le tabelle allocate):

$$\text{pagine totali di page table} = 1 + 1024 = \boxed{1025}.$$

(Numero totale di pagine virtuali:  $2^{20} = \boxed{1,048,576}$ .)

### 5.4.3 Paginazione a due livello con EAT

**Consegna.** Un sistema usa indirizzi virtuali a 32 bit e pagine da 4 KiB. La tabella delle pagine è a **due livelli** e ogni voce di tabella (PTE) occupa 4 byte. È presente una TLB con tempo di consultazione  $T_{\text{TLB}} = 10 \text{ ns}$ . Il tempo di accesso alla memoria principale è  $T_M = 100 \text{ ns}$  (trascura cache, rotazione, trasferimento). Il **tasso di hit** della TLB è  $\alpha = 0,97$ .

1. Determina la scomposizione dell'indirizzo virtuale nei campi [indice 1° livello | indice 2° livello | offset], e il numero di voci per ciascuna tabella.
2. Calcola l'**EAT** (senza page fault).
3. Se la probabilità di **page fault** per accesso è  $p = 10^{-5}$  e il tempo di servizio del page fault è  $F = 5 \text{ ms}$  (incluso swap-in e gestione kernel), ricalcola l'EAT.

**Soluzione.** **1) Scomposizione indirizzo.** Con pagine da 4 KiB =  $2^{12}$  byte, l'*offset* occupa 12 bit. Restano  $32 - 12 = 20$  bit di numero di pagina virtuale, che in una tabella a due livelli si ripartiscono tipicamente in 10 bit + 10 bit:

$$\underbrace{\quad}_{\text{indice L1}} \quad \underbrace{\quad}_{\text{indice L2}} \quad \underbrace{\quad}_{\text{offset}}.$$

Un PTE è di 4 byte, quindi in un *blocco-tabella* da 4 KiB ci stanno  $\frac{4096}{4} = 1024 = 2^{10}$  voci: perfetto per i 10 bit di indice. Dunque: L1 = 10 bit, L2 = 10 bit, offset = 12 bit, con 1024 voci per ciascuna tabella di livello.

**2) EAT senza page fault.** - *TLB hit*: costo  $T_{\text{TLB}} + T_M = 10 + 100 = 110 \text{ ns}$ . - *TLB miss*: occorrono 2 accessi a memoria per le due tabelle (L1 e L2), poi 1 accesso ai dati: totale  $3 T_M$ . Costo =  $T_{\text{TLB}} + 3 T_M = 10 + 3 \cdot 100 = 310 \text{ ns}$ .

Quindi

$$EAT = \alpha \cdot (110) + (1 - \alpha) \cdot (310) = 0,97 \cdot 110 + 0,03 \cdot 310 = 106,7 + 9,3 = \boxed{116,0 \text{ ns}}.$$

**3) EAT con page fault** ( $p = 10^{-5}$ ,  $F = 5 \text{ ms}$ ). Portiamo tutto in nanosecondi:  $F = 5 \text{ ms} = 5,000,000 \text{ ns}$ . Assumiamo che  $F$  includa tutto il servizio del fault (swap-in + aggiornamenti), al termine del quale l'accesso riprende con il costo "normale". Allora:

$$EAT_{\text{pf}} = (1-p) \cdot 116 \text{ ns} + p \cdot (F + 116 \text{ ns}) = 0,99999 \cdot 116 + 10^{-5} \cdot 5,000,116 \approx 115,99884 + 50,00116 = \boxed{166,0 \text{ ns}}.$$

**Osservazione.** Anche con un  $p$  molto piccolo, il contributo del page fault (millisecondi) fa crescere sensibilmente l'EAT (decine di nanosecondi in più).

#### 5.4.4 Altro su i-node

**Dati.** File system con blocchi da  $4 \text{ KiB} = 4096 \text{ byte}$ . Ogni indirizzo di blocco occupa 4 byte. Un i-node contiene i seguenti puntatori:

- **Diretti (3):** 7, 9, 11
- **Indiretto singolo:** 20
- **Indiretto doppio/triplo:** assenti (-1)

Il contenuto del blocco 20 (indiretto singolo) è:

$\boxed{30, 31, 45, 46, \dots}$

#### Richieste.

1. In quale **numero di blocco** cade l'offset 5000 B del file? (Gli offset partono da 0.)
2. In quale **numero di blocco** cade l'offset 13000 B del file?
3. Supponi che il file utilizzi esattamente i blocchi dati 7, 9, 11, 30, 31 e che l'ultimo byte utile si trovi al **byte 1000 del blocco 31** (contati da 0). Qual è la **dimensione del file**?

#### Soluzione.

1.  $k = \lfloor \frac{5000}{4096} \rfloor = 1 \Rightarrow$  è il *secondo* blocco del file (indice 1). Tra i diretti:  $\{7, 9, 11\} \Rightarrow \boxed{9}$ . Offset interno al blocco:  $5000 - 1 \cdot 4096 = \boxed{904}$  B.
2.  $k = \lfloor \frac{13000}{4096} \rfloor = 3 \Rightarrow$  è il *quarto* blocco del file (indice 3). I primi 3 blocchi sono diretti; dal quarto in poi si usa l'indiretto singolo. Indice nell'indiretto:  $3 - 3 = 0 \Rightarrow$  prima voce del blocco 20  $\rightarrow \boxed{30}$ . Offset interno:  $13000 - 3 \cdot 4096 = 13000 - 12288 = \boxed{712}$  B.
3. Blocchi pieni:  $7, 9, 11, 30 \Rightarrow 4 \cdot 4096 = 16384$  B. Nel blocco 31 si usano i byte  $0 \dots 1000 \Rightarrow 1001$  B. Dimensione del file:  $16384 + 1001 = \boxed{17385 \text{ byte}}$ .

Schema generale: con blocco da  $B$  byte, l'offset  $x$  cade nel blocco d'indice  $k = \lfloor x/B \rfloor$ . Se  $k < n_{\text{diretti}}$  si usa il puntatore diretto  $k$ -esimo, altrimenti si passa all'indiretto singolo con indice  $k - n_{\text{diretti}}$  (e analogamente per doppio/triplo indiretto).

### 5.4.5 I-node con indiretto singolo/doppio e calcolo dei frame fisici

#### Dati.

- Dimensione di blocco:  $B = 4 \text{ KiB} = 4096 \text{ B}$ . Ogni indirizzo di blocco occupa 4 B  $\Rightarrow$  un blocco di indirizzi contiene  $N = \frac{4096}{4} = 1024$  voci.
- i-node di un file (10 diretti + 1 indiretto singolo + 1 indiretto doppio; triplo assente):

Diretti (10) : [321, 322, 239, 234, 235, 236, 14, 21, 233, 12]

Ind. singolo : **112**

Ind. doppio : **333**

Ind. triplo : -1

- Contenuto dei blocchi d'indirizzi noti:

**Blocco 112** : [16, 544, 20, 555, 922, 942, 120, **[132]**, 134, 140, 141, 142, 143, 144, 145, 146, ...]

**Blocco 333** : [**[233]**, **[322]**, **[444]**, 530, 742, 221, ...] (indirizzi di secondo livello)

**Blocco 233** : [821, 822, 915, **[50]**, 51, 53, ...] (indirizzi di dati)

**Blocco 444** : [200, 288, 201, 280, **[399]**, 400, ...] (indirizzi di dati)

- Architettura di memoria: indirizzi virtuali a 32 bit, pagina da 4 KiB; indirizzo fisico a 28 bit.

#### Richieste.

- Per ciascun offset, indicare **il numero di blocco dati** che lo contiene (gli offset partono da 0) e **l'offset interno** nel blocco:

- a)  $x_1 = 9000$
- b)  $x_2 = 17 \cdot 4096 + 257$
- c)  $x_3 = 1037 \cdot 4096 + 900$
- d)  $x_4 = 3086 \cdot 4096 + 123$

*Suggerimento:* il blocco logico del file è  $k = \lfloor x/B \rfloor$ .

- Quante **lettture di blocco** servono per ottenere il byte, se si assume che l'i-node sia già in RAM e che ad ogni accesso (indipendente) i blocchi d'indirizzi necessari vadano letti da disco? (Conta i soli blocchi su disco: 1 per il dato, +1 per ogni livello indiretto usato.)
- Calcolare la **dimensione massima del file indirizzabile** con questa struttura (10 diretti + singolo + doppio; il triplo è assente).
- Calcolare il numero di **frame fisici** disponibili in RAM.

#### Soluzione.

- Determiniamo l'indice di blocco logico  $k = \lfloor x/4096 \rfloor$  e poi il blocco fisico:

- a)**  $x_1 = 9000$ :  $k = \lfloor 9000/4096 \rfloor = 2$ . I primi 10 blocchi sono diretti  $\Rightarrow$  usa il diretto n. 2: **[blocco 239]**. Offset interno:  $9000 - 2 \cdot 4096 = \boxed{808}$  B.

- **b)**  $x_2 = 17 \cdot 4096 + 257$ :  $k = 17$ . Dal blocco  $k = 10$  in poi si usa l'indiretto singolo. Indice dentro il blocco 112:  $17 - 10 = 7 \Rightarrow$  voce 7 vale 132. Offset interno: 257 B.
- **c)**  $x_3 = 1037 \cdot 4096 + 900$ :  $k = 1037$ . La regione del singolo copre i blocchi  $10..(10 + 1024 - 1) = 10..1033$ . Dunque siamo nel *doppio* indiretto. Spostamento nel doppio:  $k - (10 + 1024) = 1037 - 1034 = \boxed{3} \Rightarrow$  indice di primo livello  $i_1 = \lfloor 3/1024 \rfloor = 0 \rightarrow$  dal blocco 333 si prende la voce 0: 233. Indice di secondo livello  $i_2 = 3 \bmod 1024 = 3 \rightarrow$  dal blocco 233 la voce 3 è 50. Risposta: blocco 50. Offset interno: 900 B.
- **d)**  $x_4 = 3086 \cdot 4096 + 123$ :  $k = 3086$ . Spostamento nel doppio:  $3086 - 1034 = \boxed{2052}$ .  $i_1 = \lfloor 2052/1024 \rfloor = 2 \rightarrow$  dal blocco 333 la voce 2 è 444.  $i_2 = 2052 \bmod 1024 = \boxed{4} \rightarrow$  dal blocco 444 la voce 4 è 399. Risposta: blocco 399. Offset interno: 123 B.

**2. Letture di blocco necessarie (inode già in RAM, accessi indipendenti):**

Caso	Livello usato	# letture
a) diretto	nessun indiretto	1 (solo blocco dati)
b) singolo	+1 blocco indiretto singolo	2
c) doppio	+1 primo livello & +1 secondo livello	3
d) doppio	idem	3

**3. Massima dimensione indirizzabile (senza triplo).** Numero massimo di blocchi dati indicizzabili:

$$B_{\max} = 10 + 1024 + 1024^2 = 10 + 1024 + 1\,048\,576 = 1\,049\,610 \text{ blocchi.}$$

In byte:

$$|F|_{\max} = B_{\max} \cdot 4096 = (10 + 1024 + 1024^2) \cdot 4096 = \boxed{4\,299\,202\,560 \text{ byte}} \approx 4.00 \text{ GiB.}$$

**4. Frame fisici in RAM.** Con pagina/blocco da  $4 \text{ KiB} = 2^{12} \text{ B}$  e indirizzo fisico a 28 bit: bit di frame =  $28 - 12 = 16 \Rightarrow$

$$\boxed{F = 2^{16} = 65\,536 \text{ frame}.}$$

**Osservazione.** Se fosse presente anche l'indiretto *triplo*, i blocchi indirizzabili diventerebbero  $10 + 1024 + 1024^2 + 1024^3$ , con una dimensione massima di  $(10 + 1024 + 1024^2 + 1024^3) \cdot 4096 \text{ byte}$ .

## 5.5 Frame fisici

### 5.5.1 Allocazione uguale e proporzionale

**Dati.** Sono disponibili  $F = 14$  frame di memoria fisica (quelli del SO sono già riservati). Tre processi hanno ampiezza del working set (o pagine attive) pari a:

$$P_1 : 10 \text{ pagine}, \quad P_2 : 20 \text{ pagine}, \quad P_3 : 6 \text{ pagine.}$$

Totalle dichiarato:  $10 + 20 + 6 = \boxed{36}$  pagine.

**Richieste.**

1. Calcolare l'allocazione **uguale** (equal allocation).
2. Calcolare l'allocazione **proporzionale** alle pagine attive.

**Soluzione.** 1) **Equal allocation.** Quota base:  $q = \left\lfloor \frac{F}{3} \right\rfloor = \left\lfloor \frac{14}{3} \right\rfloor = 4$ . Restano  $r = 14 - 3 \cdot 4 = 2$  frame da distribuire (ad es. ai primi due processi).

$$\boxed{P_1 = 5, \quad P_2 = 5, \quad P_3 = 4} \quad (\text{somma} = 14).$$

2) **Allocazione proporzionale.** Formula:  $f_i = \text{round}\left(F \cdot \frac{\text{pagine}_i}{\sum \text{pagine}}\right)$ .

$$\begin{aligned} P_1 : 14 \cdot \frac{10}{36} &= 3.89 \Rightarrow 4, \\ P_2 : 14 \cdot \frac{20}{36} &= 7.78 \Rightarrow 8, \\ P_3 : 14 \cdot \frac{6}{36} &= 2.33 \Rightarrow 2. \end{aligned}$$

La somma è già  $4 + 8 + 2 = 14$ , quindi non servono aggiustamenti.

$$\boxed{P_1 = 4, \quad P_2 = 8, \quad P_3 = 2}.$$

**Nota.** Generalmente si dovrebbe, per calcolare il numero di frame per processo, calcolare il massimo tra il minimo architetturale e quello di cui effettivamente il processo ha bisogno, in questo esercizio è presente una semplificazione.