



NoiStudiamo

DISPENSA DI TEORIA



Sistemi Operativi

Processi e thread, sincronizzazione, scheduling, gestione della memoria, file system e dischi.

A CURA DI

Damiano La Pusata

Ottobre 2026

Indice dei capitoli

I ventisette capitoli della dispensa, dai concetti introduttivi ai dischi a stato solido.

1. Che cos'è un sistema operativo	4
2. Richiami sull'architettura degli elaboratori	5
3. Lo zoo dei sistemi operativi	9
4. La struttura di un sistema operativo	10
5. Macchine virtuali	12
6. I processi	13
7. I thread	19
8. Comunicazione fra processi e corse critiche	24
9. Mutua esclusione con attesa attiva	26
10. Sleep e wakeup, semafori, mutex	29
11. Monitor e scambio di messaggi	34
12. Problemi classici di sincronizzazione	38
13. Lo scheduling: concetti di base	43
14. Scheduling nei sistemi batch	45
15. Scheduling nei sistemi interattivi	48
16. Scheduling dei thread e dei multiprocessori	53
17. Dalla memoria fisica allo spazio degli indirizzi	55

18. Memoria virtuale e paginazione	58
19. Gli algoritmi di sostituzione delle pagine	65
20. Allocazione dei frame e thrashing	70
21. Condivisione, ottimizzazioni e memoria del kernel	73
22. Il file system: file e directory	76
23. L'implementazione dei file	78
24. Le directory e la condivisione dei file	84
25. Spazio libero, affidabilità e prestazioni	86
26. Lo scheduling del disco	90
27. I sistemi RAID e gli SSD	93

Che cos'è un sistema operativo

Che cosa fa un sistema operativo, da quali punti di vista lo si può descrivere e come si è evoluto dalle prime macchine a valvole ai sistemi attuali.

1.1 Definizione

Un moderno calcolatore è formato da uno o più processori, dalla memoria centrale, dai dischi, dalle stampanti e da molte altre periferiche di I/O. I dettagli di basso livello di tutti questi componenti sono estremamente complessi, e scrivere un programma che li piloti direttamente sarebbe un'impresa quasi impossibile. Serve quindi uno **strato software intermedio** fra l'hardware e i programmi applicativi: il **sistema operativo**.

Una definizione più operativa identifica il sistema operativo con il programma che resta **sempre in esecuzione** sul calcolatore: il **kernel**. Il kernel è il cuore del sistema, ha controllo completo su tutto ciò che accade ed è il primo programma caricato in memoria durante l'avvio. Nei sistemi moderni, attorno al kernel si trova anche un **middleware**, cioè una collezione di ambienti software che offre servizi aggiuntivi per lo sviluppo delle applicazioni. È utile ricordare che la **GUI non è il sistema operativo**: è una delle possibili interfacce verso l'utente, e gira in modalità utente come qualunque altra applicazione.

1.2 Due punti di vista: macchina estesa e gestore delle risorse

Il sistema operativo si può guardare da due prospettive complementari.

Come **macchina estesa**, il sistema operativo nasconde la complessità dell'hardware dietro **astrazioni** più semplici e più potenti: il *file* al posto dei settori del disco, il *processo* al posto della CPU, lo *spazio di indirizzamento* al posto della RAM fisica. Il programmatore lavora su queste astrazioni e non ha bisogno di sapere, ad esempio, come si posiziona la testina di un disco.

Come **gestore delle risorse**, il sistema operativo deve permettere a **più programmi** e a **più utenti** di convivere, assegnando in modo ordinato e controllato processori, memoria, dispositivi di I/O e anche risorse non fisiche come i file. La tecnica fondamentale è il **multiplexing**, che può avvenire in due modi:

- **nel tempo**: la risorsa viene usata a turno: la CPU esegue un processo alla volta, la stampante stampa un documento alla volta;
- **nello spazio**: la risorsa viene suddivisa fra più utilizzatori contemporanei: la memoria centrale ospita più processi, il disco contiene i file di più utenti.

1.3 L'evoluzione dei sistemi operativi

La **prima generazione** (valvole e schede a spinotti) non conosceva sistemi operativi: il programmatore inseriva a mano la scheda con il programma, scritto in linguaggio macchina. Con i **transistor** della **seconda generazione** nacquero i **mainframe** e i sistemi **batch**: i job, scritti su schede perforate, venivano trasferiti su nastro da un calcolatore piccolo come l'IBM 1401, eseguiti in sequenza da uno potente come l'IBM 7094, e i risultati stampati di nuovo dal 1401, *off line*. La **terza generazione**, a circuiti integrati, portò la linea IBM System/360 con OS/360, enorme, complesso e pieno di errori, ma soprattutto introdusse la **multiprogrammazione**: mentre un job attende la fine di un'operazione di I/O, la CPU può eseguirne un altro. La **quarta generazione** è quella dei personal computer (MS DOS, poi le interfacce grafiche con il Macintosh); la **quinta** quella dei dispositivi mobili, con Android e iOS.

Richiami sull'architettura degli elaboratori

I concetti di architettura su cui poggia tutto il resto: processore, modalità di esecuzione, gerarchia di memoria, dispositivi di I/O e bus.

2.1 Il processore

Un calcolatore moderno è composto da una o più CPU e da un certo numero di **controllori di dispositivo**, collegati fra loro dal **bus di sistema**. La CPU recupera le istruzioni dalla memoria e le esegue ripetendo il ciclo di base **fetch, decode, execute**: preleva l'istruzione, la decodifica, la esegue. Oltre ai registri generici, la CPU dispone di alcuni **registri speciali**:

- il **Program Counter** (PC), che contiene l'indirizzo della prossima istruzione da eseguire;
- lo **Stack Pointer** (SP), che punta alla cima dello stack in memoria;
- la **Program Status Word** (PSW), che contiene bit di controllo: la modalità di esecuzione (kernel o utente), la priorità della CPU, i flag di stato dell'ultima operazione (zero, riporto, segno) e l'abilitazione degli interrupt.

Per aumentare le prestazioni le CPU usano la **pipeline**: mentre si esegue l'istruzione n , si decodifica la $n+1$ e si preleva la $n+2$, così da sovrapporre le fasi di istruzioni diverse. L'architettura **superscalare** va oltre, disponendo di più unità di esecuzione e prelevando più istruzioni per volta in un buffer. Queste tecniche **non sono del tutto trasparenti al sistema operativo**, che ne deve tenere conto, ad esempio, quando salva lo stato di un processo.

DA RICORDARE

Come si calcola il throughput di una CPU con pipeline?

Con una pipeline ideale, una volta riempita, si completa un'istruzione per ciclo di clock, quindi il throughput tende alla frequenza di clock. La **latenza** della singola istruzione invece non diminuisce: resta pari al numero di stadi. La pipeline non rende più veloce la singola istruzione, ma aumenta quante istruzioni si completano nell'unità di tempo.

2.2 Modalità kernel e modalità utente

Per garantire che un errore in un programma non comprometta l'intero sistema, l'hardware distingue almeno **due modalità di esecuzione**, indicate da un bit della PSW:

- in **modalità kernel** (o *supervisor*) il codice ha accesso completo all'hardware, può eseguire qualunque istruzione e accedere a qualunque indirizzo di memoria;
- in **modalità utente** il codice non può eseguire le istruzioni privilegiate né accedere direttamente all'hardware, e deve delegare queste operazioni al sistema operativo.

I processori Intel hanno in realtà più livelli di protezione, detti **protection ring**: il **ring 0** corrisponde alla modalità kernel e il **ring 3** alla modalità utente, codificati con due bit della PSW (00 e 11).

Quando un programma utente ha bisogno di un servizio che richiede privilegi (leggere un file, creare un processo, allocare memoria) esegue una **chiamata di sistema** (*system call*). La chiamata avviene tramite l'istruzione **TRAP**, che commuta la CPU da modalità utente a modalità kernel e salta al codice del sistema operativo che gestisce quella specifica richiesta, individuato attraverso una **tabella delle system call**. Terminata l'operazione, il kernel restituisce il controllo al programma in modalità utente. Nella pratica il programma non esegue la TRAP a mano:

chiama una funzione di libreria (ad esempio `write()`), che prepara i parametri ed esegue la TRAP al posto suo.

DA RICORDARE

Cosa succede quando un processo fa una system call: cambia il PID? Cambia lo spazio di indirizzamento?

Nessuno dei due. È sempre lo **stesso processo**, con la stessa memoria virtuale: cambia soltanto il **livello di privilegio** della CPU, che passa in modalità kernel per eseguire il codice della chiamata nel contesto dello stesso processo e poi torna in modalità utente. Questo passaggio di modalità non va confuso con la distinzione fra thread utente e thread kernel, che riguarda invece come sono implementati i thread.

2.3 Multithreading hardware, multiprocessori e multicore

Durante l'esecuzione di un programma capita spesso che parte della CPU resti inutilizzata, ad esempio mentre si attende un dato dalla memoria. Il **multithreading hardware** (chiamato *hyperthreading* da Intel) sfrutta questi momenti: la CPU mantiene al proprio interno **lo stato di due thread** e passa dall'uno all'altro in pochi nanosecondi. Va precisato che **non c'è un'esecuzione parallela vera e propria**: i due thread logici condividono le unità del core, e il sistema operativo deve tenerne conto, perché vede due CPU logiche dove fisicamente ce n'è una.

ESEMPIO SVOLTO

Si vuole calcolare il tempo complessivo di due thread T0 e T1 di 9 s e 5 s su una CPU hyperthreading a 2 thread che dedica loro il 100%. Il risultato è **9 + 5 = 14 s = 14 000 ms**: i due thread condividono le stesse unità di esecuzione e non avanzano davvero in parallelo.

I sistemi a **singolo processore** avevano comunque processori specializzati per compiti particolari, come i controller dei dispositivi. Oggi si usano sistemi **multiprocessore**, i cui vantaggi sono il maggior **throughput**, l'**economia di scala** (si condividono periferiche, memoria e alimentazione) e l'**affidabilità**. Il modello standard è la **multielaborazione simmetrica** (SMP): ogni processore può eseguire qualunque operazione di sistema e tutti condividono la memoria fisica attraverso il bus, pur avendo ciascuno i propri registri e la propria cache. Nei sistemi **multicore** le unità di elaborazione stanno sullo stesso chip; le **GPU** portano l'idea all'estremo con centinaia di core semplici.

Aggiungere CPU non fa crescere il throughput in modo lineare: con troppi processori il **bus condiviso** diventa un collo di bottiglia. Un'alternativa è dare a ogni CPU una propria memoria locale e collegare fra loro i processori: è l'architettura **NUMA** (*Non Uniform Memory Access*), in cui l'accesso alla memoria di un altro processore è possibile ma più lento.

CONCETTO	CHE COSA SIGNIFICA	PARALLELISMO
Multithreading HW	Un core mantiene lo stato di più thread e alterna fra loro	Non vero: le unità del core sono condivise
Multicore	Più core sullo stesso chip	Vero, un thread per core
Multiprocessore	Più CPU fisiche che condividono la RAM	Vero, ma il bus limita la scalabilità
NUMA	Ogni CPU ha una memoria locale; le altre sono raggiungibili ma più lente	Vero, con accessi non uniformi

2.4 La gerarchia di memoria

La memoria è organizzata in una **gerarchia di strati**: salendo verso la CPU le memorie diventano più veloci e più costose per bit, ma meno capienti. In cima ci sono i **registri**, veloci quanto la CPU. Seguono le **cache** (L1, L2, L3), gestite dall'hardware: quando un programma legge una parola, l'hardware controlla se si trova in cache (*cache hit*) e solo in caso contrario accede alla memoria principale. Poi viene la **RAM** (*Random Access Memory*), così chiamata perché qualunque indirizzo è accessibile con gli stessi tempi; è la memoria più bassa della gerarchia direttamente utilizzabile dalla CPU. Seguono i dischi elettronici (SSD) e i **dischi magnetici**, fatti di piatti su cui le informazioni sono scritte in cerchi concentrici: ogni testina legge un anello detto **traccia**, e l'insieme delle tracce sotto tutte le testine in una data posizione del braccio forma un **cilindro**.

Esistono poi memorie non volatili particolari: la **ROM**, che contiene tipicamente il programma di avvio (*bootstrap loader*), e la **CMOS**, alimentata da una batteria, che conserva data, ora e impostazioni del BIOS. Va ricordata infine la distinzione fra le unità di misura decimali (**kB, MB, GB**, potenze di 10) e binarie (**KiB, MiB, GiB**, potenze di 2), che negli esercizi va tenuta presente.

2.5 Protezione della memoria e MMU: un primo sguardo

Quando più programmi stanno in memoria contemporaneamente bisogna risolvere due problemi: **proteggere** i programmi l'uno dall'altro e gestire la **rilocazione**, perché al momento della compilazione non si sa in quale punto della memoria fisica il programma verrà caricato. La soluzione più semplice usa due registri speciali, il **registro base** e il **registro limite**: ogni indirizzo generato dal programma viene confrontato con il limite e poi sommato alla base. Il dispositivo che traduce gli **indirizzi virtuali** generati dal programma in **indirizzi fisici** si chiama **MMU** (*Memory Management Unit*). Riprenderemo questi concetti in dettaglio nella parte sulla memoria.

Per sincronizzare tutte le componenti la scheda madre usa un **clock di sistema**, un cristallo di quarzo che oscilla a frequenza costante e invia un segnale a tutti i componenti. Il **clock della CPU** segue lo stesso principio ma è interno al processore e scandisce il momento in cui i segnali fra i circuiti logici possono essere considerati validi.

2.6 Dispositivi di I/O: controller e driver

Un dispositivo di I/O è formato da due componenti: il **controller** e il **dispositivo** vero e proprio. Il dispositivo ha un'interfaccia elementare ma complicata da pilotare; il controller, un chip o un insieme di chip, accetta comandi dal sistema operativo e offre un'interfaccia molto più semplice. Un esempio sono i dischi SATA. Ogni controller ha un **buffer** interno e un insieme di **registri**, ed è responsabile del trasferimento dei dati fra i dispositivi collegati e il proprio buffer: quando si preme un tasto, ad esempio, il controller della tastiera memorizza nel buffer il codice del carattere.

Il sistema operativo comunica con i controller attraverso i **driver**, insiemi di procedure software specifici per ogni dispositivo (e spesso per ogni sistema operativo) che girano in **modalità kernel**. Il driver interagisce con i registri del controller attraverso le **porte di I/O**, in due modi possibili: con istruzioni dedicate di tipo **IN/OUT**, eseguibili solo in modalità kernel, oppure tramite **mappatura in memoria** (*memory mapped I/O*), in cui i registri del controller appaiono come normali indirizzi di memoria.

2.7 Tre modi di gestire l'I/O

BUSY WAITING (POLLING)

Il metodo più semplice: il programma invoca una system call, il kernel la traduce in una chiamata al driver, il driver avvia l'operazione e poi la CPU **interroga continuamente** il controller (*polling*) per sapere se l'operazione è terminata. È un metodo svantaggioso, perché la CPU resta occupata per tutto il tempo dell'operazione invece di fare lavoro utile.

I/O GUIDATO DAGLI INTERRUPT

Il driver scrive il comando nei registri del controller e la CPU passa ad altro. Quando il trasferimento è concluso, il controller segnala la fine con un **interrupt**: un segnale che l'hardware invia alla CPU tramite il bus di controllo in un momento qualsiasi. La CPU interrompe l'elaborazione corrente e salta alla **routine di servizio dell'interrupt** (ISR), che termina con un'istruzione di ritorno (tipo *return from interrupt*) con cui l'esecuzione interrotta riprende.

Poiché esistono molti dispositivi diversi, serve un modo per capire *quale* ha chiesto attenzione. L'**interrupt controller** dispone di più linee **IRQ** (*Interrupt ReQuest*): quando una periferica genera un interrupt, il controller mette sul bus dati il numero di IRQ, e la CPU lo usa come indice in una tabella di puntatori, il **vettore delle interruzioni** (*interrupt vector*), che contiene l'indirizzo della ISR corretta. Il vettore contiene soltanto questi **indirizzi**: lo stato del programma interrotto (PC e PSW) viene salvato **sullo stack**, come vedremo nella parte sui processi. Così nessun gestore deve cercare fra tutte le possibili sorgenti. Se i dispositivi sono più delle voci del vettore, si usa il **concatenamento delle interruzioni**: ogni voce punta a una lista di gestori, interrogati uno dopo l'altro finché uno riconosce la richiesta. Le CPU hanno in genere due linee di interrupt: una **non mascherabile**, riservata a eventi gravi come errori irreversibili della memoria, e una **mascherabile**, usata dai controller, che la CPU può disabilitare temporaneamente durante sequenze di istruzioni critiche.

I/O CON DMA

Il **DMA** (*Direct Memory Access*) permette di trasferire dati fra un controller e la memoria **senza coinvolgere la CPU** per ogni byte. La CPU programma il chip DMA indicando il dispositivo, gli indirizzi di memoria coinvolti e la quantità di dati da trasferire; il DMA esegue il trasferimento da solo e, alla fine, invia un interrupt. Durante il trasferimento la CPU è libera, salvo un leggero rallentamento dovuto al fatto che il DMA occupa a tratti il bus (*cycle stealing*).

Gli interrupt possono essere disabilitati dalla CPU su alcune linee. Se sono disabilitati, un processo in esecuzione può essere fermato soltanto quando termina o commette un errore. Esistono però situazioni in cui un programma termina comunque in modo forzato: un **errore** come la violazione di segmento, una **richiesta di I/O inesistente**, oppure un **kill** da parte di un altro programma.

DA RICORDARE

Quale di queste operazioni non richiede la modalità supervisor (kernel)?

Richiedono i privilegi tutte le operazioni che, se lasciate a un programma qualsiasi, gli permetterebbero di scavalcare la protezione: **modificare la mappatura della memoria** di un processo; **cambiare il registro PTBR** (*Page Table Base Register*, il registro che indica alla MMU dove si trova in RAM la tabella delle pagine del processo in esecuzione: cambiarlo significa cambiare la memoria visibile); **disabilitare gli interrupt**, che impedirebbe al sistema operativo di riprendere il controllo; **accedere alle porte di I/O** con le istruzioni IN/OUT. **Leggere la PSW** invece non richiede privilegi: la PSW può essere letta da chiunque, ma modificata solo in modalità kernel.

2.8 I bus

I componenti comunicano attraverso i **bus**: classicamente uno per i dati, uno per gli indirizzi e uno per il controllo, ma nei calcolatori moderni ne esistono molti altri, sia per l'I/O sia per velocizzare lo scambio fra CPU e memoria. Il bus principale oggi è il **PCIe** (*Peripheral Component Interconnect express*), un bus seriale ad alta velocità per GPU, schede di rete e SSD NVMe; il **DMI** (*Direct Media Interface*) collega la CPU al chipset, da cui dipendono USB, SATA e gli altri dispositivi; l'**USB** è nato per collegare dispositivi lenti secondo il principio *plug and play*. I bus si dividono in **paralleli**, dove i bit di una parola viaggiano contemporaneamente su linee distinte, e **seriali**, dove i bit viaggiano uno dopo l'altro su una connessione detta *lane*.

Lo zoo dei sistemi operativi

Le principali famiglie di sistemi operativi, distinte in base all'hardware su cui girano e agli obiettivi che devono raggiungere.

Esistono sistemi operativi molto diversi, ognuno progettato per un tipo di macchina e per un preciso insieme di vincoli. Le categorie non sono sempre esclusive: un sistema reale può avere caratteristiche di più classi.

CATEGORIA	CARATTERISTICHE PRINCIPALI
Mainframe e server	Gestiscono grandi moli di dati, molte CPU fisiche e un elevato carico di I/O, supportando la multiprogrammazione con straordinaria stabilità. Privilegiano throughput e affidabilità (ridondanza hardware, cluster, backup). Uno dei primi sistemi nati in questo ambito è UNIX.
Multiprocessore	Varianti dei sistemi per server con funzionalità specifiche di comunicazione e connessione fra le CPU; usati per grandi carichi batch.
Personal computer	Ambiente interattivo, reattivo e facile da usare, con GUI, supporto a molte periferiche, multiutenza e protezione dei dati.
Palmas e smartphone	Interfacce touch, processi isolati fra loro (<i>sandboxing</i>), gestione aggressiva dell'energia; il sistema è spesso sviluppato insieme all'hardware su cui gira.
Embedded (integrati)	Progettati per compiti specifici e definiti: i processi sono pochi e noti in anticipo agli sviluppatori, l'utente non installa programmi aggiuntivi.
Real time	La correttezza dipende anche dal momento in cui arriva il risultato. Si dividono in hard real time , dove mancare una scadenza compromette l'intero sistema, e soft real time , dove un ritardo degrada il servizio senza comprometterlo.

Sul real time c'è un punto da avere chiaro: **real time non significa veloce**, significa **rispettare le scadenze** e avere un comportamento **prevedibile**. Per riuscire questi sistemi usano in genere uno scheduling con prelazione, così da poter servire subito un evento urgente.

La struttura di un sistema operativo

I modi in cui si può organizzare il codice di un sistema operativo, dal kernel monolitico al microkernel, con vantaggi e limiti di ciascuno.

Si possono distinguere cinque strutture: **monolitica**, **a livelli**, **microkernel**, **a moduli** e **macchine virtuali**. Si tratta di categorie con intersezioni (la maggior parte dei sistemi reali è **ibrida**) e di una tassonomia non necessariamente completa né universalmente condivisa.

4.1 Sistemi monolitici

Nei sistemi monolitici l'intero sistema operativo è un **unico programma** che gira in modalità kernel: una raccolta di procedure, ciascuna delle quali può chiamarne liberamente qualunque altra. Il programma oggetto si ottiene compilando le singole procedure e collegandole in un unico file, senza alcun occultamento delle informazioni. Una struttura di base esiste comunque, perché le system call vengono richieste mettendo i parametri in posti prestabiliti ed eseguendo una TRAP: un **programma principale** invoca la **procedura di servizio** richiesta, che a sua volta si appoggia a **procedure di utilità** comuni. È una struttura semplice e veloce, perché i componenti si chiamano direttamente, ma **poco gestibile nel tempo** e difficile da comprendere. I primi sistemi monolitici, come MS DOS e il primo UNIX, non avevano nemmeno il supporto hardware alla doppia modalità, che arrivò in seguito.

4.2 Sistemi a livelli

Il sistema è organizzato come una **gerarchia di livelli**: ciascuno implementa una funzionalità usando soltanto i servizi del livello sottostante. Il vantaggio è una migliore progettazione, più semplice da sviluppare e da verificare, con un incapsulamento simile a quello della programmazione a oggetti; l'esempio storico è il sistema **THE** di Dijkstra. La separazione però è solo **logica**: un errore che espone le strutture dati di un livello resta possibile. Il limite principale sono le **prestazioni**, perché le chiamate nidificate attraverso molti livelli aggiungono overhead.

Una variante è la struttura ad **anelli concentrici** di **MULTICS**, in cui i livelli sono separati da meccanismi di protezione **hardware** e salendo verso l'esterno i privilegi diminuiscono; per interagire con un livello più interno si esegue una TRAP. La separazione è quindi forzata, ma i controlli aggiuntivi rendono l'overhead rilevante, e il sistema ebbe scarso successo.

4.3 Microkernel

L'idea del microkernel è mettere in modalità kernel **il meno possibile**. Il nucleo minimale si occupa soltanto delle funzioni essenziali (gestione degli interrupt e dei processi, scheduling, memoria di base, **comunicazione fra processi** (IPC) e clock) mentre tutto il resto (file system, driver dei dispositivi, servizi di rete) gira in **modalità utente** sotto forma di processi separati, detti **server**. I componenti comunicano fra loro tramite **messaggi**, che contengono mittente e destinatario, tipo di richiesta ed eventuali dati.

I vantaggi sono un **design migliore**, con componenti piccoli e comprensibili, e una maggiore **stabilità**: se un servizio in modalità utente fallisce, il sistema può terminarlo e riavviarlo automaticamente, un meccanismo chiamato **reincarnazione**. Lo svantaggio è l'**overhead** dello scambio di messaggi e dei cambi di contesto fra i servizi. Esempi tipici sono **MINIX 3**, **Mach**, **QNX**, **Mac OS (Darwin)** e **Windows NT**. Gli ultimi due nascono da un'impostazione a microkernel ma sono in realtà **sistemi ibridi**: il kernel XNU di macOS unisce il microkernel Mach a componenti BSD eseguiti nello stesso spazio kernel, e Windows NT esegue in modalità kernel molti servizi che un microkernel puro terrebbe fuori.

4.4 Struttura a moduli

È l'idea della programmazione a oggetti applicata al kernel. Un **kernel principale a funzionalità ridotte** carica **dinamicamente**, quando servono, dei **moduli** che implementano aspetti specifici come un file system o un driver; su Linux i moduli sono file con estensione `.ko` (*kernel object*). Il design è pulito ed **efficiente**, perché ogni modulo può invocare direttamente qualunque altro modulo senza scambio di messaggi, e i moduli caricati solo su richiesta riducono l'uso di RAM. Il rovescio della medaglia è che i moduli girano **in modalità kernel** con pieni privilegi, quindi un modulo difettoso può compromettere l'intero sistema. Esempi: Solaris, Linux e, come ibrido, Mac OS.

STRUTTURA	PUNTI DI FORZA	LIMITI	ESEMPI
Monolitica	Massime prestazioni, chiamate dirette	Poco gestibile; un errore coinvolge tutto il kernel	MS DOS, primo UNIX
A livelli	Progettazione ordinata, verifica per livelli	Overhead delle chiamate nidificate	THE, MULTICS (anelli)
Microkernel	Nucleo piccolo, servizi isolati e riavviabili	Overhead di messaggi e cambi di contesto	MINIX 3, Mach, QNX
A moduli	Estendibile, caricamento dinamico, niente messaggi	I moduli hanno privilegi kernel	Linux, Solaris

DA RICORDARE

Differenze fra kernel monolitico, a livelli, microkernel e a moduli in termini di overhead e manutenibilità.

Conviene ragionare lungo due assi. Sul piano dell'**overhead**, monolitico e moduli sono i più veloci perché tutto gira nello stesso spazio kernel con chiamate dirette; i livelli pagano l'attraversamento degli strati; il microkernel paga i messaggi. Sul piano dell'**isolamento dei guasti**, il microkernel è il migliore perché i servizi sono processi utente; i moduli migliorano la manutenibilità rispetto al monolitico (si caricano e scaricano senza ricompilare) ma non l'isolamento. È per questo che Linux viene descritto come un *monolitico a moduli*.

Macchine virtuali

Come un unico calcolatore fisico può ospitare più macchine virtuali, ognuna con il proprio sistema operativo, e quali tecniche lo rendono possibile.

Portando all'estremo il concetto di astrazione si arriva alla **virtualizzazione** di un intero calcolatore. Una **macchina virtuale** esegue software come se si trovasse su un computer fisico, compreso un sistema operativo diverso da quello della macchina ospitante. Le motivazioni sono molte: usare più sistemi operativi sulla stessa macchina, offrire server virtuali in affitto (**VPS**), **isolare i servizi** l'uno dall'altro. L'isolamento logico rispetto alla macchina ospitante rende le macchine virtuali molto usate anche in sicurezza informatica, ad esempio per analizzare malware.

Esistono diversi modi di realizzare la virtualizzazione:

- i **container** si appoggiano al kernel del sistema ospitante e contengono applicazioni, alcune API e servizi di sistema eseguiti in modalità utente;
- nella **virtualizzazione classica** la CPU della macchina ospitante esegue direttamente il codice delle macchine virtuali, garantendo velocità;
- nell'**emulazione** (o *simulazione*) il codice ospitato viene eseguito su una CPU emulata via software, con registri simulati in memoria: è molto meno efficiente, ma permette di eseguire codice scritto per un'architettura diversa;
- nella **paravirtualizzazione** il sistema ospitato sa di essere virtualizzato e dialoga direttamente con l'hypervisor, ottenendo prestazioni migliori e un uso più efficiente delle risorse.

Le macchine virtuali sono gestite dall'**hypervisor**, il componente software che intercetta e controlla tutte le loro operazioni. Ne esistono due tipi:

TIPO	DOVE GIRA	CARATTERISTICHE	ESEMPI
Tipo 1 (bare metal)	Direttamente sull'hardware	Strato software il cui unico compito è creare e gestire le VM; prestazioni elevate	VMware ESX/ESXi, Microsoft Hyper V
Tipo 2 (hosted)	Come processo in un sistema operativo ospitante	Meno efficiente ma più comodo, adatto allo sviluppo; spesso richiede moduli kernel dell'host	VMware Workstation, VirtualBox

Le CPU moderne offrono un **supporto hardware alla virtualizzazione**, che gestisce in modo efficiente il passaggio fra modalità utente e kernel sia nella macchina ospitante sia in quella ospitata. Dal punto di vista della sicurezza, l'attacco più grave è il **VM escape**, in cui codice eseguito nella macchina virtuale riesce a oltrepassarne i confini e a raggiungere l'host o le altre VM, vanificando proprio l'isolamento che giustifica la virtualizzazione.

I processi

Il processo come astrazione fondamentale: come nasce e termina, in quali stati si trova e come il sistema operativo lo rappresenta e lo alterna agli altri.

6.1 Che cos'è un processo

Un **processo** è un'istanza di esecuzione di un programma. La distinzione fra i due termini è il primo punto da avere chiaro: il **programma** è un'entità **passiva**, un file sul disco che contiene istruzioni; il **processo** è un'entità **attiva**, che occupa memoria, usa la CPU e interagisce con i dispositivi. Lo stesso programma può dare origine a più processi contemporaneamente: due finestre dello stesso editor sono due processi distinti, ciascuno con i propri dati.

DEFINIZIONE

Un **processo** è un programma in esecuzione: comprende il codice, i dati, lo stack, i valori correnti del program counter e dei registri e le risorse che gli sono assegnate, come i file aperti. Il programma è un'entità passiva; il processo è un'entità attiva, e lo stesso programma può dare origine a più processi.

Tanenbaum usa un'analogia efficace. Un pasticcere prepara una torta seguendo una ricetta: la **ricetta** è il programma, il **pasticcere** è la CPU, gli **ingredienti** sono i dati di ingresso, e il **processo** è l'intera attività di leggere la ricetta, prendere gli ingredienti e cuocere la torta. Se a metà lavoro il figlio del pasticcere viene punto da un'ape, il pasticcere segna il punto della ricetta a cui era arrivato, prende il manuale di pronto soccorso (un altro programma, quindi un altro processo, con priorità più alta) e, finita la medicazione, riprende la torta esattamente da dove l'aveva lasciata. È esattamente ciò che fa il sistema operativo quando passa da un processo all'altro.

A ogni processo il sistema operativo associa un insieme di risorse. La principale è lo **spazio degli indirizzi**, cioè la memoria che il processo può usare, organizzata in più parti:

- il **codice** (*text*), cioè le istruzioni del programma, di dimensione fissa;
- i **dati**, cioè le variabili globali e statiche, anch'essi di dimensione fissa;
- lo **heap**, la memoria allocata dinamicamente durante l'esecuzione (con `malloc`), che cresce verso gli indirizzi alti;
- lo **stack**, che contiene un *record di attivazione* per ogni chiamata di funzione in corso (parametri, variabili locali, indirizzo di ritorno) e cresce verso gli indirizzi bassi. Il sistema operativo deve garantire che stack e heap non si sovrappongano.

Oltre alla memoria, al processo sono associati una **copia dei registri della CPU** (compreso il program counter), l'elenco dei **file aperti**, gli **allarmi pendenti** (timer impostati dal processo) e le informazioni sui **processi imparentati** (padre e figli). Tutte queste informazioni sono raccolte nella **tabella dei processi**, che contiene una voce, il **Process Control Block** (PCB), per ogni processo esistente.

6.2 Il modello dei processi

Con la **multiprogrammazione** la CPU passa continuamente da un processo all'altro, eseguendo ciascuno per qualche decina di millisecondi. In ogni istante una CPU esegue **un solo processo**, ma nell'arco di un secondo ne fa avanzare molti, dando l'illusione che girino tutti insieme: si parla di **pseudoparallelismo**, per distinguerlo dal parallelismo vero dei sistemi con più CPU o più core.

Ragionare su questo continuo alternarsi sarebbe complicato. Per questo si adotta il **modello dei processi**: si immagina che ogni processo sia un programma **sequenziale** che dispone di una propria **CPU virtuale** dedicata. Ogni processo ha quindi un proprio **program counter logico**; quando il processo va in esecuzione, il suo program counter logico viene caricato nel **program counter fisico** della CPU, e quando viene sospeso il valore del program counter fisico viene salvato nel suo PCB. Una conseguenza importante è che la velocità con cui un processo avanza **non è uniforme né riproducibile**: dipende da quanti altri processi ci sono e da come lo scheduler li alterna. Un programma non deve quindi mai fare ipotesi sui tempi di esecuzione.

6.3 Creazione e terminazione dei processi

QUANDO NASCE UN PROCESSO

Un processo può essere creato in tre circostanze. La prima è l'**inizializzazione del sistema**: all'avvio vengono creati diversi processi, alcuni in primo piano che interagiscono con l'utente, altri in background che svolgono servizi (posta, stampa, rete) e sono detti **demoni** (*daemon*). La seconda è la **richiesta di un altro processo**, che tramite una system call crea un processo **figlio** per farsi aiutare in un lavoro. La terza è un'**azione dell'utente**, come un doppio clic su un'icona o il nome di un programma digitato nel terminale; anche in questo caso, tecnicamente, è un processo già esistente (la shell o l'interfaccia grafica) a chiedere la creazione.

UNIX: FORK ED EXEC

Nei sistemi UNIX esiste una sola system call per creare un processo: `fork()`. La `fork` **sdoppia il processo chiamante**: crea un figlio che è una **copia esatta** del padre, con la stessa immagine di memoria, gli stessi file aperti e gli stessi valori dei registri. Da quel momento i due processi proseguono indipendentemente, ciascuno con il **proprio spazio degli indirizzi**: una modifica a una variabile fatta dal figlio non è visibile al padre, e viceversa. Per distinguere chi è chi, la `fork` restituisce il **PID del figlio al padre e 0 al figlio** (e `-1` al padre se la creazione fallisce).

Per eseguire un programma diverso, il figlio chiama una system call della famiglia `exec` (ad esempio `execve`), che **sostituisce** il codice e i dati del processo con quelli del nuovo programma. La `exec` **non crea un nuovo processo**: il PID resta lo stesso, cambia solo il programma eseguito. Separare la creazione (`fork`) dal caricamento del programma (`exec`) ha un vantaggio preciso: fra le due chiamate il figlio può sistemare il proprio ambiente, ad esempio redirigere lo standard input o output. È così che la shell esegue un comando: fa una `fork`, e il figlio fa una `exec` del comando richiesto. Il padre può attendere la fine del figlio con `wait/waitpid`, che restituisce anche il codice di uscita.

SCHEMA TIPICO FORK, EXEC E WAIT

```
pid_t pid = fork();           // da qui in poi i processi sono due
if (pid < 0) {
    perror("fork");           // creazione fallita: c'è solo il padre
} else if (pid == 0) {
    // ramo del FIGLIO: fork ha restituito 0
    execlp("ls", "ls", NULL); // sostituisce il programma con ls
    perror("exec");           // si arriva qui solo se exec fallisce
    exit(1);
} else {
    // ramo del PADRE: pid contiene il PID del figlio
    int stato;
    waitpid(pid, &stato, 0);   // attende la terminazione del figlio
}
```

I sistemi UNIX tengono traccia delle parentele: ogni processo ha un padre, e l'insieme dei processi forma un **albero** con radice nel primo processo creato all'avvio (su Linux si visualizza con `ps tree`). Nei sistemi moderni la `fork` non copia davvero subito tutta la memoria: padre e figlio condividono le pagine finché uno dei due non prova a

modificarne una, e solo allora quella pagina viene duplicata. È la tecnica **copy on write**, che vedremo nella parte sulla memoria.

WINDOWS: CREATEPROCESS

In Windows una sola chiamata, `CreateProcess`, crea il nuovo processo e vi carica **subito** il programma da eseguire: non c'è la fase di copia del padre. Inoltre in Windows **non esiste una gerarchia** di processi: tutti i processi sono uguali. Il padre riceve un **handle**, un identificatore con cui può controllare il figlio, ma può cederlo a un altro processo, cosa che in UNIX, dove la parentela è fissa, non è possibile.

COME TERMINA UN PROCESSO

Le cause di terminazione sono quattro, due volontarie e due involontarie:

- **uscita normale** (volontaria): il processo ha finito il suo lavoro e chiama `exit` in UNIX o `ExitProcess` in Windows;
- **uscita su errore** (volontaria): il processo rileva un problema, ad esempio un file di input inesistente, e termina di sua iniziativa con un codice di errore;
- **errore critico** (involontaria): il processo esegue un'istruzione illegale, come una divisione per zero o un accesso a memoria non consentito, e il sistema operativo lo termina;
- **terminazione da parte di un altro processo** (involontaria): con `kill` in UNIX o `TerminateProcess` in Windows, purché il processo che la richiede abbia i permessi necessari.

Per convenzione il codice di uscita è **0** se tutto è andato bene e **diverso da 0** in caso di errore; quando un processo viene terminato da un segnale, la shell riporta un codice pari a **128 più il numero del segnale**. Un processo terminato il cui padre non ha ancora raccolto il codice di uscita con `wait` resta nella tabella dei processi come **zombie**: non esegue più nulla, ma la sua voce occupa spazio finché il padre non la legge. Se invece è il padre a terminare per primo, il figlio rimasto **orfano** viene adottato dal processo iniziale del sistema, che si occupa di raccogliergli la terminazione.

DA RICORDARE

Che cosa fanno `fork` ed `exec`, e perché sono due chiamate separate?

La `fork` crea un processo figlio duplicando il padre: il figlio ha un **nuovo PID** e una copia dello spazio degli indirizzi, dei file aperti e dei registri; la chiamata restituisce il PID del figlio al padre e 0 al figlio. La `exec` non crea processi: **sostituisce il programma** del processo che la chiama, mantenendone il PID. Tenerle separate permette al figlio di preparare il proprio ambiente (ad esempio redirigere input e output o chiudere file) prima di caricare il nuovo programma: è il meccanismo con cui la shell esegue i comandi e realizza le redirezioni.

ESEMPIO SVOLTO: CONTARE I PROCESSI GENERATI

Il primo passo è ricordare che dopo una `fork` **entrambi** i processi proseguono dall'istruzione successiva. Con un ciclo semplice come `for (i = 0; i < n; i++) fork();` ogni processo esistente si sdoppia a ogni giro, quindi alla fine i processi sono **2ⁿ**, compreso quello iniziale.

Si vuole sapere quanti processi genera, compreso quello avviato dalla shell, questo codice:

```
for (int i = 0; i <= 3; i++) {
    pid = fork();
    if (pid == 0 && i != 2)    // i figli escono subito, tranne quello nato con i = 2
        exit(0);
}
exit(0);
```

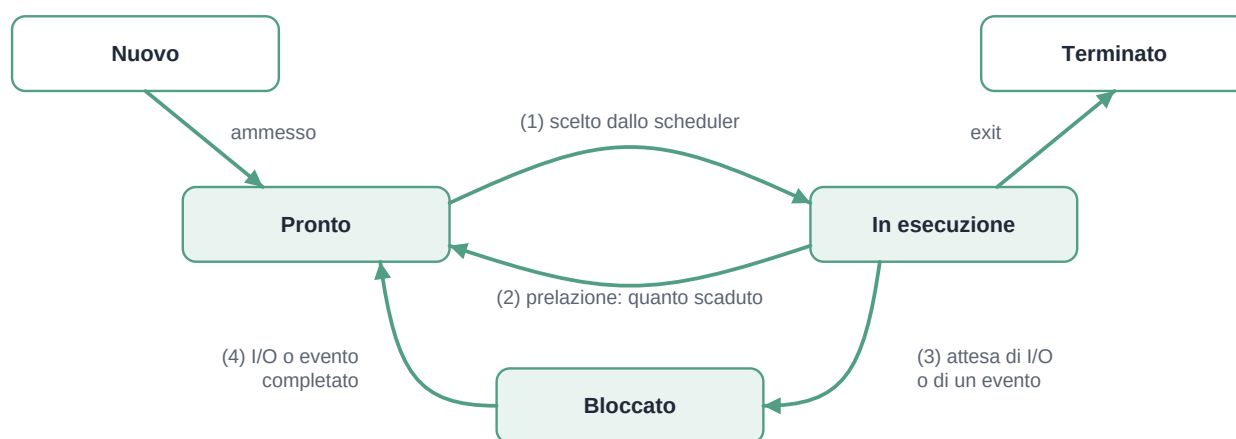
Si segue il ciclo giro per giro. Il processo iniziale P0 fa una fork a ogni giro, quindi crea **4 figli** (con $i = 0, 1, 2, 3$). I figli nati con $i = 0, 1$ e 3 escono subito. Il figlio nato con $i = 2$ invece prosegue il ciclo: al giro $i = 3$ fa a sua volta una fork, e il suo figlio esce subito perché i vale 3. In totale: P0, i suoi 4 figli e 1 nipote, cioè **6 processi**.

6.4 Gli stati di un processo

Durante la sua vita un processo cambia **stato**. Gli stati principali sono tre:

- **in esecuzione** (*running*): il processo sta usando la CPU in quell'istante;
- **pronto** (*ready*): il processo potrebbe eseguire, ma è temporaneamente fermo perché la CPU è occupata da un altro processo;
- **bloccato** (*blocked* o *waiting*): il processo non può proseguire finché non si verifica un evento esterno, tipicamente il completamento di un'operazione di I/O. Anche se la CPU fosse libera, non potrebbe usarla.

A questi si aggiungono due stati **addizionali**, che riguardano l'inizio e la fine: **nuovo** (*new*), mentre il processo viene creato e il sistema ne prepara le strutture, e **terminato** (*terminated*), quando ha finito di eseguire e il sistema ne sta recuperando le risorse.



Gli stati di un processo: i tre stati principali (riquadri pieni) e i due stati addizionali (riquadri bianchi).

Le transizioni fra i tre stati principali sono quattro:

1. **Da pronto a in esecuzione:** lo **scheduler** sceglie il processo fra quelli pronti e gli assegna la CPU (si dice che il processo viene *dispatched*).
2. **Da in esecuzione a pronto:** il processo perde la CPU senza averne colpa, perché lo scheduler decide di passarla a un altro processo, tipicamente alla scadenza del **quanto di tempo** segnalata da un interrupt del clock. È la **prelazione**.
3. **Da in esecuzione a bloccato:** il processo chiede qualcosa che non è ancora disponibile, ad esempio legge da un file o attende un dato dalla tastiera, e non può andare avanti. Invece di tenere la CPU ferma, il sistema lo blocca e la assegna a un altro processo.
4. **Da bloccato a pronto:** l'evento atteso si verifica (l'I/O termina, arriva il dato). Il processo torna **pronto**, non in esecuzione: dovrà aspettare che lo scheduler lo scelga di nuovo.

Una transizione diretta **da pronto a bloccato non può esistere**: un processo si blocca soltanto perché, mentre esegue, chiede qualcosa che deve attendere, e un processo pronto non sta eseguendo nulla. Per lo stesso motivo un processo bloccato non passa mai direttamente in esecuzione.

DA RICORDARE

Fra Pronto, Bloccato e In esecuzione, quale transizione diretta non può mai avvenire?

La transizione **da Pronto a Bloccato**. Un processo si blocca solo come conseguenza di una sua richiesta (un'operazione di I/O, l'attesa di un evento), e per fare una richiesta deve essere **in esecuzione**. Le transizioni possibili sono: da Pronto a In esecuzione (scelta dello scheduler), da In esecuzione a Pronto (prelazione), da In esecuzione a Bloccato (attesa di un evento), da Bloccato a Pronto (evento avvenuto). Da notare che quando l'evento arriva il processo torna Pronto, e non direttamente In esecuzione.

6.5 La tabella dei processi e il PCB

Per realizzare il modello dei processi il sistema operativo mantiene la **tabella dei processi**, con una voce per ogni processo: il **Process Control Block (PCB)**. Il PCB è la «carta d'identità» del processo e contiene tutto ciò che serve per sospenderlo e poi riprenderlo come se non fosse mai stato interrotto:

CATEGORIA	INFORMAZIONI CONTENUTE NEL PCB
Identità e stato	Il numero del processo (PID), lo stato (pronto, in esecuzione, bloccato...), i riferimenti al processo padre e ai figli.
Contesto della CPU	Il program counter , lo stack pointer, la PSW e i valori di tutti gli altri registri al momento della sospensione.
Memoria	I limiti di memoria o il puntatore alla tabella delle pagine: tutto ciò che serve per riprogrammare la MMU quando il processo torna in esecuzione.
File e I/O	L'elenco dei file aperti e dei dispositivi in uso.
Scheduling e contabilità	La priorità, i puntatori alle code in cui il processo è inserito, il tempo di CPU già consumato.

Il PCB non è il processo: è la **struttura dati** con cui il sistema operativo lo rappresenta. Il componente che sceglie quale processo pronto mandare in esecuzione è lo **scheduler**, che lavora proprio consultando la tabella dei processi.

DA RICORDARE

Che cosa contiene il PCB, e perché va salvato e ripristinato a ogni cambio di contesto?

Contiene il PID e lo stato del processo, il program counter, i registri e la PSW, le informazioni sulla memoria (registri base e limite o puntatore alla tabella delle pagine), i file aperti, la priorità e i dati di contabilità, i riferimenti al padre. Va salvato perché i **registri fisici della CPU sono uno solo** e vengono riutati dal processo successivo: se il sistema non copiasse il loro contenuto nel PCB prima di cambiare processo, quei valori andrebbero persi e il processo non potrebbe più riprendere dal punto in cui era. Ripristinandoli dal PCB, il processo riparte esattamente da dove si era fermato. È questo salvataggio continuo a rendere possibile lo pseudoparallelismo.

6.6 Interrupt e cambio di contesto

Il passaggio della CPU da un processo all'altro si chiama **cambio di contesto** (*context switch*) e consiste nel salvare lo stato del processo corrente nel suo PCB e ripristinare quello del processo successivo. Il cambio di contesto avviene in risposta a un **interrupt** (ad esempio quello del clock che segnala la fine del quanto, o quello di un disco che ha terminato un trasferimento) o a una system call. La sequenza di operazioni, riprendendo quanto visto nei richiami di architettura, è la seguente:

1. l'hardware salva **sullo stack attuale** il program counter e la PSW del processo interrotto;

2. l'hardware carica dal **vettore degli interrupt** l'indirizzo della procedura di servizio associata a quell'interrupt;
3. una breve procedura in assembly **salva i registri** del processo e imposta un **nuovo stack** per il gestore;
4. viene eseguita la **procedura di servizio** dell'interrupt (scritta in C), che gestisce l'evento: ad esempio segna come pronto il processo che attendeva quell'I/O;
5. viene interrogato lo **scheduler**, che decide con quale processo proseguire;
6. si **ripristina dal PCB** lo stato del processo scelto: registri e mappa della memoria;
7. riprende l'esecuzione del processo scelto, che può essere lo stesso di prima o un altro.

Il cambio di contesto è **puro overhead**: mentre lo esegue, la CPU non fa avanzare nessun processo utente. Il costo maggiore non sta tanto nel copiare i registri, quanto nel cambiare la **mappa della memoria**: bisogna riprogrammare la MMU e le cache contengono ancora i dati del processo precedente, che diventano inutili. Per questo il sistema deve trovare un equilibrio fra reattività (cambiare spesso processo) ed efficienza (cambiarlo il meno possibile), come vedremo nella parte sullo scheduling.

6.7 Code e accodamento

I processi pronti sono organizzati nella **coda dei processi pronti** (*ready queue*); esistono poi le **code dei dispositivi**, una per ogni dispositivo, con i processi bloccati in attesa di quel dispositivo. Queste code non contengono copie dei processi: sono **liste collegate di PCB**, in cui la testa della coda punta al primo PCB e ogni PCB punta al successivo.

Il percorso di un processo si descrive con il **diagramma di accodamento**. Un processo entra nella ready queue e attende di essere scelto; quando ottiene la CPU, mentre esegue può accadere una di queste cose: emette una **richiesta di I/O**, e viene spostato nella coda del dispositivo finché l'operazione non termina; il suo **quanto di tempo scade**, e torna in fondo alla ready queue; **crea un processo figlio** e ne attende la terminazione. In tutti i casi, quando l'attesa finisce il processo rientra nella ready queue. Il ciclo si ripete finché il processo non termina; a quel punto il suo PCB viene rimosso da tutte le code e deallocato.

I thread

I thread come flussi di esecuzione all'interno di un processo: perché servono, che cosa condividono e come vengono realizzati.

7.1 Perché servono i thread

Nel modello dei processi ogni processo è un'entità indipendente che **raggruppa delle risorse** (spazio degli indirizzi, file aperti...) e possiede **un unico flusso di esecuzione**. Spesso però è utile avere **più flussi di esecuzione che condividono lo stesso spazio degli indirizzi**, e quindi gli stessi dati: questi flussi si chiamano **thread**. Quando un processo contiene più thread si parla di **multithreading**; ogni processo nasce con un solo thread, che può poi crearne altri.

Gli esempi classici aiutano a capire quando conviene. Un **programma di videoscrittura** deve leggere l'input dell'utente, reimpaginare il documento e salvarlo periodicamente su disco. Con un solo flusso di esecuzione, durante il salvataggio automatico il programma non risponderebbe alla tastiera. Con tre thread (uno per l'input, uno per l'impaginazione, uno per il salvataggio) ognuno svolge il suo compito e, se uno si blocca in attesa del disco, gli altri continuano. I tre thread devono lavorare **sullo stesso documento**: per questo non basterebbero tre processi, che avrebbero memorie separate.

Un **web server** usa invece lo schema **dispatcher e worker**: un thread *dispatcher* riceve le richieste dalla rete e le assegna a un thread *worker* libero, che cerca la pagina nella cache condivisa o sul disco. Se un worker si blocca leggendo dal disco, il dispatcher e gli altri worker continuano a servire richieste. Lo stesso vale per un **browser**, che scarica le immagini di una pagina mentre l'utente la scorre.

In sintesi i thread offrono tre vantaggi: permettono di **sovrapporre calcolo e I/O** all'interno della stessa applicazione (il guadagno è grande soprattutto per i programmi I/O bound); sono **più leggeri** dei processi, perché crearli e distruggerli costa molto meno; e su un sistema con più core permettono un **parallelismo reale** all'interno di un'unica applicazione.

7.2 Il modello a thread

Ogni thread ha soltanto ciò che serve a descrivere un flusso di esecuzione: il proprio **program counter**, i propri **registri**, il proprio **stack** (perché ogni thread chiama funzioni diverse in momenti diversi) e il proprio **stato**. Tutto il resto è **condiviso** fra i thread dello stesso processo: lo spazio degli indirizzi con codice, variabili globali e heap, i file aperti, i processi figli, gli allarmi. Un'analogia utile: i thread sono **collegati dello stesso ufficio**, che condividono archivio e strumenti ma hanno ciascuno la propria scrivania e il proprio compito.

DEFINIZIONE

Un **thread** è un flusso di esecuzione all'interno di un processo. Ogni thread ha un proprio program counter, propri registri, un proprio stack e un proprio stato; condivide con gli altri thread dello stesso processo lo spazio degli indirizzi, le variabili globali, i file aperti e le altre risorse.

PER OGNI PROCESSO (CONDIVISO DAI SUOI THREAD)	PER OGNI THREAD (PRIVATO)
Spazio degli indirizzi: codice, dati globali, heap	Program counter
File aperti	Registri
Processi figli, allarmi pendenti, segnali	Stack
Informazioni di contabilità	Stato (in esecuzione, pronto, bloccato, terminato)

La condivisione ha un rovescio importante: **fra i thread di un processo non c'è protezione della memoria**. Un thread può leggere e sovrascrivere lo stack o i dati di un altro thread dello stesso processo. Non è una mancanza del sistema: i thread appartengono alla stessa applicazione, scritta dallo stesso programmatore, e si suppone che cooperino. I thread sono quindi flussi **indipendenti ma non isolati**: un errore in un thread può corrompere i dati usati da tutti gli altri, e gli accessi alle variabili condivise vanno coordinati con gli strumenti che vedremo nella parte sulla sincronizzazione.

Come i processi, anche i thread cambiano stato: un thread può essere **in esecuzione, pronto, bloccato** (ad esempio in attesa di I/O o di un altro thread) o **terminato**, con le stesse transizioni viste per i processi.

ASPETTO	PROCESSO FIGLIO (FORK)	THREAD
Memoria	Spazio degli indirizzi proprio, copia di quello del padre	Condivide lo spazio degli indirizzi del processo
Identità	Nuovo PID	Stesso processo, quindi stesso PID
Creazione e cambio di contesto	Più costosi	Più leggeri
Isolamento	Alto: un errore resta nel processo	Nessuna protezione fra i thread
Comunicazione	Serve un meccanismo di IPC (pipe, messaggi...)	Diretta, tramite le variabili condivise

7.3 Scheduling e cambio di contesto dei thread

Ogni thread è un'unità di esecuzione indipendente e viene schedato con logiche simili a quelle dei processi: se un thread si blocca, gli altri thread dello stesso processo possono continuare. Il vero guadagno di prestazioni sta nel **cambio di contesto**: passare da un thread a un altro **dello stesso processo** è molto più veloce che passare da un processo all'altro, perché lo spazio degli indirizzi non cambia e quindi **la MMU non va riprogrammata**. Se invece il thread successivo appartiene a un **altro processo**, il cambio comporta anche il cambio di spazio degli indirizzi, e costa quanto un normale cambio di contesto fra processi.

DA RICORDARE

Perché il cambio di contesto fra thread dello stesso processo è più leggero di quello fra processi diversi?

In entrambi i casi vanno salvati e ripristinati program counter, registri e stack pointer, perché sono propri di ogni thread. La differenza sta nello **spazio degli indirizzi**: fra thread dello stesso processo resta identico, quindi non bisogna cambiare la tabella delle pagine né riprogrammare la MMU, e le traduzioni degli indirizzi già memorizzate e il contenuto delle cache restano validi. Fra processi diversi invece la mappa della memoria cambia completamente: va caricato il riferimento alla nuova tabella delle pagine e le informazioni di traduzione del processo precedente non servono più. Cambiare la mappa della memoria è l'operazione più pesante di un cambio di contesto, ed è proprio quella che i thread dello stesso processo evitano.

7.4 Operazioni sui thread

Le operazioni tipiche, indipendenti dalla libreria usata, sono quattro:

- **thread_create**: un thread ne crea un altro, indicando come parametro la **procedura** che il nuovo thread dovrà eseguire. Di solito fra i thread non c'è gerarchia: sono tutti uguali;
- **thread_exit**: il thread chiamante termina. A differenza di `exit`, che termina l'intero processo, gli altri thread continuano a esistere;
- **thread_join**: un thread si sincronizza con la fine di un altro. Se il thread A chiama la `join` sul thread B, A resta **bloccato** finché B non termina, poi torna pronto;
- **thread_yield**: il thread chiamante **rilascia volontariamente** la CPU per permettere a un altro thread di eseguire.

L'esistenza della `yield` dice molto sulla natura dei thread. Fra **processi** il rapporto è **competitivo**: appartengono magari a utenti diversi e si contendono la CPU, e il sistema deve poterla togliere loro con il clock. Fra i **thread** di un processo il rapporto è **collaborativo**: lavorano per la stessa applicazione e possono cedere la CPU a vicenda. Nella libreria POSIX, che userai in laboratorio, queste operazioni si chiamano `pthread_create`, `pthread_exit`, `pthread_join` e `sched_yield`.

7.5 Programmazione multicore

I thread permettono di sfruttare al meglio le CPU con hyperthreading e soprattutto i sistemi **multicore**. Su un sistema con un solo core, quattro thread T1...T4 avanzano in modo **interlacciato** (*interleaved*): il core esegue un pezzo di T1, poi di T2, di T3, di T4, e così via. Su un sistema con due core si ha invece **parallelismo vero**: il core 0 alterna T1 e T3 mentre il core 1 alterna T2 e T4, e due thread avanzano davvero nello stesso istante. Quando i thread sono più dei core, il sistema usa comunque il time sharing su ciascun core.

Scrivere programmi che sfruttino davvero più core non è banale. Si possono indicare cinque principi:

- **separazione dei task**: individuare nel problema attività distinte che possano procedere contemporaneamente;
- **bilanciamento**: fare in modo che le attività abbiano un carico di lavoro paragonabile, altrimenti un core finisce presto e resta inattivo mentre un altro lavora ancora;
- **suddivisione dei dati**: dividere anche i dati su cui lavorano i task, così che ogni core lavori sulla propria parte;
- **dipendenze dei dati**: se un task usa i risultati di un altro, i due vanno sincronizzati, e la sincronizzazione ha un costo;
- **test e debugging**: con più flussi di esecuzione le possibili sequenze di eventi sono moltissime, e un errore può presentarsi solo in alcune esecuzioni, il che rende test e correzione molto più difficili.

Un esempio è il **merge sort parallelo**: le due metà dell'array possono essere ordinate da due thread su due core diversi (task separati, dati suddivisi, carico bilanciato), ma la fusione finale **dipende** da entrambi i risultati e può partire solo quando tutti e due i thread hanno finito.

7.6 Thread a livello utente: il modello uno a molti

Esistono tre modi di realizzare i thread, che differiscono per **chi li gestisce**: una libreria nello spazio utente, il kernel, o una combinazione dei due. Nel primo, i **thread a livello utente**, il kernel non sa nulla dei thread: vede soltanto un normale processo con un unico flusso di esecuzione. I thread sono realizzati da una **libreria** che implementa un **sistema run time**; questo mantiene, all'interno del processo stesso, una **tabella dei thread** con program counter, registri, stack e stato di ciascuno. Il modello si chiama **uno a molti** perché a un'unica entità schedulabile del kernel corrispondono molti thread dell'utente. Storicamente è nato per avere i thread anche sui sistemi operativi che non li supportavano.

Quando un thread chiama una funzione della libreria, ad esempio `thread_yield`, il sistema run time salva il suo stato nella tabella, sceglie un altro thread pronto e ne carica lo stato nei registri. I **vantaggi** sono due:

- il **dispatching non richiede trap nel kernel**: il cambio di thread è una semplice chiamata di procedura locale, molto più veloce di una system call, senza cambio di modalità e senza toccare la memoria;
- ogni processo può avere un **algoritmo di scheduling personalizzato**, scelto in base alle esigenze dell'applicazione.

Gli **svantaggi** però sono pesanti:

- **chiamate bloccanti**: se un thread esegue una system call bloccante, ad esempio una lettura da tastiera, il kernel blocca **l'intero processo**, perché è l'unica cosa che conosce, e con esso tutti gli altri thread, anche quelli che potrebbero lavorare. Si può aggirare il problema con `select`, che permette di verificare **prima** se una chiamata si bloccherebbe: in quel caso il thread cede la CPU con una `yield` e riprova più tardi. Ma è una soluzione macchinosa e non vale per tutte le chiamate;
- **page fault**: se un thread accede a una parte del programma che non è in memoria, il kernel blocca il processo mentre la carica dal disco, ancora una volta fermando tutti i thread. Qui la `select` non serve, perché il page fault non è una chiamata che il thread possa controllare in anticipo;
- **mancato rilascio della CPU**: all'interno di un processo non arrivano interrupt del clock destinati ai singoli thread, quindi il sistema run time non può togliere la CPU a un thread. Se un thread non chiama mai la `yield` e non si blocca, gli altri thread dello stesso processo non vengono mai eseguiti;
- infine, poiché il kernel vede un solo flusso di esecuzione, i thread utente di un processo **non possono girare in parallelo su più core**.

7.7 Thread a livello kernel: il modello uno a uno

Se il kernel supporta i thread, e oggi praticamente tutti i sistemi operativi moderni lo fanno, è il kernel stesso a gestirli. Esiste allora un'**unica tabella dei thread nel kernel**, spesso fusa con la tabella dei processi, e ogni thread dell'utente corrisponde a un thread del kernel: è il modello **uno a uno**. Creare, distruggere o bloccare un thread richiede una system call.

Il **vantaggio** principale risolve proprio il difetto del modello precedente: poiché il kernel conosce i singoli thread, **un thread che fa una chiamata bloccante o causa un page fault non intralcia gli altri**. Il kernel blocca solo quel thread e può mandare in esecuzione un altro thread dello stesso processo o di un altro. Inoltre lo scheduler del kernel può togliere la CPU a un thread con la prelazione, e su un sistema multicore può eseguire più thread dello stesso processo in parallelo.

Gli **svantaggi** riguardano i costi. Il **cambio di contesto è più lento**, perché richiede una trap nel kernel. Anche la **creazione e la distruzione** dei thread sono **più costose**, per lo stesso motivo; per questo il numero di thread kernel è tipicamente limitato, e alcuni sistemi li **riciclano**: quando un thread termina, la sua struttura nel kernel non viene distrutta ma riutilizzata per il thread successivo.

7.8 Il modello ibrido: molti a molti

Il **modello ibrido**, detto **molti a molti**, cerca di prendere il meglio degli altri due. Il sistema mette a disposizione un certo numero di **thread kernel**, e su ciascuno di essi vengono multiplexati più **thread utente**, gestiti dalla libreria come nel modello uno a molti; un thread kernel può anche ospitare un solo thread utente. L'**assegnazione è decisa dal programmatore**: le attività che rischiano di bloccarsi o devono girare in parallelo vengono legate a thread kernel distinti, mentre quelle leggere e numerose possono condividere un thread kernel e scambiarsi la CPU senza entrare nel kernel. Si ottiene così la massima flessibilità.

	THREAD UTENTE (UNO A MOLTI)	THREAD KERNEL (UNO A UNO)	IBRIDO (MOLTI A MOLTI)
Chi li gestisce	Libreria run time nello spazio utente	Kernel	Kernel e libreria insieme
Tabella dei thread	Nel processo	Nel kernel	Entrambe
Cambio di thread	Veloce, senza trap	Più lento, con trap	Veloce fra thread utente dello stesso thread kernel
Chiamata bloccante o page fault	Blocca tutti i thread del processo	Blocca solo il thread interessato	Blocca i thread utente di quel thread kernel
Prelazione e più core	No	Sì	Sì, a livello dei thread kernel
Scheduling	Personalizzabile	Quello del kernel	Personalizzabile entro ogni thread kernel

DA RICORDARE

Differenze fra thread a livello utente e thread a livello kernel.

I thread utente sono gestiti da una **libreria** nello spazio del processo: il kernel vede un solo flusso di esecuzione. Il cambio di thread è velocissimo perché non richiede trap, e lo scheduling è personalizzabile; ma una chiamata bloccante o un page fault **bloccano tutto il processo**, un thread che non cede la CPU blocca gli altri, e non si sfruttano più core. I thread kernel sono gestiti dal **sistema operativo**, con la tabella dei thread nel kernel: un thread bloccato non ferma gli altri e si può avere parallelismo vero, ma creazione e cambio di contesto costano di più perché passano dal kernel. Il modello ibrido multiplexa più thread utente su alcuni thread kernel.

Nel modello uno a uno, con uno scheduler con prelazione, quale affermazione è errata?

Le alternative proposte: un page fault di un thread non blocca gli altri thread (vera: il kernel blocca solo quel thread); per manipolare la tabella dei thread serve la modalità supervisor (vera: la tabella sta nel kernel); il modello si può usare solo se supportato nativamente dal sistema operativo (vera); il cambio di contesto è meno costoso fra thread dello stesso processo (vera: non cambia lo spazio degli indirizzi). L'affermazione **errata** è che «un thread che non rilascia spontaneamente la CPU può comunque bloccare gli altri thread dello stesso processo»: con i thread kernel e la prelazione, lo scheduler gli toglie la CPU allo scadere del quanto. Quel problema esiste solo con i thread utente.

7.9 I thread nei sistemi operativi reali

Quasi tutti i sistemi operativi attuali supportano i **thread a livello kernel**: Windows, Linux, Solaris, macOS. I thread utente restano disponibili attraverso apposite librerie: i **green threads** su Solaris, le **GNU Portable Threads** sui sistemi UNIX e le **fiber** su Windows. Avendo entrambi, questi sistemi possono realizzare anche il modello ibrido.

Indipendentemente dal modello sottostante, il programmatore accede ai thread tramite **librerie**. La più importante è **Pthreads** di POSIX, disponibile su Solaris, Linux, macOS e, tramite librerie aggiuntive, anche su Windows: va notato che Pthreads è una **specifica**, cioè un insieme di funzioni e comportamenti standard che ogni sistema implementa a modo suo. Esistono poi i thread nativi di **Windows** e i thread di **Java**, che sono un **wrapper** sulle API del sistema sottostante: la stessa classe Java usa i thread Windows su Windows e Pthreads su Linux.

Comunicazione fra processi e corse critiche

Che cosa accade quando più processi lavorano su dati condivisi, e quali condizioni deve rispettare una soluzione corretta al problema.

8.1 Il problema della comunicazione

Spesso i processi hanno bisogno di **cooperare**. L'esempio più semplice è la **pipe** della shell: nel comando `ls | grep .c` l'output di `ls` diventa l'input di `grep`. Già qui si vedono i problemi tipici: serve un **mezzo** attraverso cui passare i dati, e serve **coordinamento**, perché `grep` deve aspettare che `ls` produca qualcosa e, se il canale è pieno, `ls` deve aspettare che `grep` lo svuoti. I meccanismi che permettono ai processi di scambiarsi dati e coordinarsi prendono il nome di **InterProcess Communication** (IPC).

I problemi da risolvere sono tre: **come scambiarsi i dati**; come evitare l'**accavallamento delle operazioni su dati comuni**, cioè che due processi si intralcino mentre lavorano sulla stessa risorsa; come garantire il **coordinamento** fra le operazioni, cioè la **sincronizzazione**, ad esempio che chi consuma un dato aspetti chi lo produce. Per i thread il primo problema è banale, perché condividono la memoria; gli altri due si presentano esattamente allo stesso modo. Per questo, in tutta questa parte, «processo» e «thread» sono interscambiabili: le soluzioni valgono per entrambi.

8.2 Le corse critiche

Quando più processi leggono e scrivono dati condivisi, il risultato può dipendere dall'**ordine** in cui vengono eseguiti. Un esempio classico è lo **spooler di stampa**. Chi vuole stampare scrive il nome del file in una directory di spool, in cui gli slot sono numerati; due variabili condivise indicano il prossimo file da stampare (out) e il prossimo slot libero (in). Supponiamo che in valga 7. Il processo A legge in e si memorizza il valore 7, ma proprio in quell'istante arriva l'interrupt del clock e la CPU passa al processo B. Anche B legge in, trova 7, scrive il nome del suo file nello slot 7 e aggiorna in a 8. Quando A riprende, usa il valore 7 che aveva letto: scrive il proprio file nello slot 7, **cancellando quello di B**, e porta in a 8. La directory sembra del tutto coerente, ma il file di B non verrà mai stampato.

DEFINIZIONE

Si ha una **corsa critica** (*race condition*) quando due o più processi leggono e scrivono dati condivisi e il risultato finale dipende dall'ordine preciso in cui le loro istruzioni vengono eseguite. La parte di codice in cui un processo accede ai dati condivisi si chiama **sezione critica**.

Lo stesso accade con i **versamenti su un conto corrente**: due versamenti di 50 € su un conto che ne contiene 100 vengono eseguiti in parallelo; entrambi leggono il saldo 100, entrambi calcolano 150 e lo scrivono. Il saldo finale è 150 invece di 200, e uno dei due versamenti è andato perso. Situazioni come queste, in cui il **risultato finale dipende dalla temporizzazione** con cui i processi vengono eseguiti, si chiamano **corse critiche** (*race condition*). Sono particolarmente insidiose perché il programma funziona quasi sempre, e l'errore si manifesta solo quando l'interruzione cade nel punto sbagliato. Si verificano anche **nel codice del kernel**, ad esempio se due core manipolano contemporaneamente la lista dei processi pronti.

La soluzione è la **mutua esclusione** nell'accesso ai dati condivisi: se un processo sta usando una variabile condivisa, gli altri devono essere esclusi dal farlo finché non ha finito.

8.3 Sezioni critiche

Il problema si può astrarre. Ogni processo alterna **sezioni non critiche**, in cui lavora su dati propri, e **sezioni critiche**, cioè le parti di codice in cui accede a dati condivisi. Basta allora che due processi non si trovino mai contemporaneamente nelle rispettive sezioni critiche. Una **buona soluzione** deve però rispettare **quattro condizioni**:

1. **mutua esclusione**: due processi non possono trovarsi contemporaneamente nelle rispettive sezioni critiche;
2. **nessuna assunzione** sulla velocità di esecuzione o sul numero di CPU;
3. **nessun processo che si trova fuori dalla propria sezione critica può bloccare** un altro processo;
4. **nessun processo deve restare all'infinito in attesa** di entrare nella propria sezione critica.

La prima condizione garantisce la correttezza, le altre tre garantiscono che la soluzione sia generale e non blocchi il sistema. Nei paragrafi che seguono valuteremo ogni tentativo proprio rispetto a queste quattro condizioni. Per comodità useremo due procedure: `enter_region`, chiamata prima di entrare nella sezione critica, e `leave_region`, chiamata all'uscita.

DA RICORDARE

Che cos'è una corsa critica e quali condizioni deve rispettare una buona soluzione?

Si ha una corsa critica quando più processi o thread accedono a dati condivisi e il risultato finale dipende dall'ordine, imprevedibile, in cui vengono eseguite le loro operazioni (ad esempio due versamenti che leggono lo stesso saldo e uno dei due va perso). La parte di codice che accede ai dati condivisi è la **sezione critica**. Una buona soluzione deve garantire che due processi non siano mai insieme nelle sezioni critiche, non fare ipotesi sulla velocità o sul numero di CPU, impedire che un processo fuori dalla sezione critica blocchi gli altri, ed evitare che un processo attenda all'infinito di entrare.

Mutua esclusione con attesa attiva

Le prime soluzioni alla mutua esclusione, basate su variabili condivise o su istruzioni hardware e accomunate dall'attesa attiva.

9.1 Disabilitare gli interrupt

La soluzione più semplice: ogni processo **disabilita gli interrupt** appena entra nella sezione critica e li riabilita appena esce. Senza interrupt, e in particolare senza quello del clock, la CPU non può passare a un altro processo, quindi nessuno può intromettersi. L'idea però ha due difetti gravi. Il primo è che non è saggio dare a un processo utente il potere di disabilitare gli interrupt: se non li riabilitasse più, per errore o di proposito, il sistema sarebbe finito. Il secondo è che su un sistema con **più CPU** la disabilitazione riguarda **solo la CPU** che esegue l'istruzione: le altre continuano a lavorare e possono accedere alla stessa memoria condivisa.

La tecnica resta invece utile **dentro il kernel**, che spesso disabilita gli interrupt per poche istruzioni mentre aggiorna variabili o liste delicate: un interrupt arrivato mentre, ad esempio, la lista dei processi pronti è in uno stato incoerente potrebbe causare proprio una corsa critica.

DA RICORDARE

Perché disabilitare gli interrupt non basta a garantire la mutua esclusione su un sistema multiprocessore?

Perché la disabilitazione è **locale alla singola CPU** che la esegue. Su un solo core, senza interrupt non possono avvenire cambi di contesto, e quindi nessun altro processo può entrare nella sezione critica. Con più core, mentre il core 1 ha gli interrupt disabilitati il core 2 continua a eseguire normalmente e può accedere agli stessi dati condivisi. Serve quindi un meccanismo che agisca sulla memoria condivisa e sia visibile a tutti i core, come le istruzioni atomiche TSL e XCHG. In più, anche su un solo core tenere gli interrupt disabilitati a lungo è pericoloso, perché si rischia di perdere eventi dei dispositivi.

9.2 Variabili di lock

Una soluzione software: una variabile condivisa `lock`, inizialmente 0. Chi vuole entrare controlla la variabile: se vale 0 la porta a 1 ed entra, se vale 1 aspetta che torni a 0. Il difetto è che **la variabile di lock è a sua volta un dato condiviso**, e il problema si ripresenta identico a quello dello spooler: un processo legge `lock` e trova 0, ma prima di riuscire a scrivere 1 perde la CPU; un secondo processo legge anch'esso 0, scrive 1 ed entra; quando il primo riprende, scrive 1 ed entra anche lui. Rileggere la variabile una seconda volta prima di scrivere non serve: la corsa critica si sposta semplicemente fra la seconda lettura e la scrittura.

9.3 Alternanza stretta

Un terzo tentativo usa una variabile condivisa `turn` che indica **di chi è il turno** di entrare. Con due processi numerati 0 e 1:

ALTERNANZA STRETTA

```
int N = 2
int turn // di chi è il turno: 0 oppure 1

function enter_region(int process)
    while (turn != process) do // finché non tocca a me...
        nothing // ...continuo a controllare (busy waiting)

function leave_region(int process)
    turn = 1 - process // passo il turno all'altro processo
```

La soluzione garantisce la mutua esclusione e si generalizza facilmente a N processi, ma ha due problemi. Il primo è che il processo in attesa **controlla continuamente** la variabile in un ciclo: è l'**attesa attiva** (*busy waiting*), che spreca tempo di CPU; un lock realizzato in questo modo si chiama **spin lock**. Il secondo è più grave: impone **turni rigidi**. Se il processo 0 esce dalla sezione critica e vorrebbe rientrare subito, non può farlo finché il processo 1 non è entrato e uscito a sua volta; se il processo 1 è lento, o è impegnato a lungo nella sua parte non critica, il processo 0 resta bloccato pur essendo la risorsa libera. Viene quindi violata la **condizione 3**: un processo fuori dalla sezione critica ne blocca un altro.

9.4 La soluzione di Peterson

Nel 1981 Peterson ha proposto una soluzione software corretta, che combina l'idea del turno con una dichiarazione di **interesse**. Si usano una variabile *turn* e un vettore *interested*, con una casella per processo:

SOLUZIONE DI PETERSON

```
int N = 2
int turn // chi è arrivato per ultimo
int interested[N] // inizialmente tutti false

function enter_region(int process)
    other = 1 - process // numero dell'altro processo
    interested[process] = true // dichiaro che voglio entrare
    turn = process // scrivo che sono l'ultimo arrivato
    while (interested[other] == true and turn == process) do
        nothing // aspetto solo se l'altro è interessato
        // e sono stato io l'ultimo a scrivere turn

function leave_region(int process)
    interested[process] = false // non sono più interessato
```

Se un processo vuole entrare e l'altro **non è interessato**, entra subito. Se invece entrambi chiamano *enter_region* quasi insieme, entrambi scrivono in *turn* il proprio numero, ma **conta solo l'ultima scrittura**, che sovrascrive la prima. Supponiamo che l'ultimo a scrivere sia il processo 1, quindi *turn* vale 1: per il processo 0 la condizione *turn == process* è falsa ed entra; il processo 1 invece resta nel ciclo finché il processo 0 non esce e pone *interested[0]* a false. In sintesi: **chi scrive per ultimo in *turn* aspetta**. Nessun processo può essere bloccato da uno che non è interessato, quindi le quattro condizioni sono rispettate.

Restano due limiti. La soluzione fa ancora **busy waiting**. Inoltre può avere problemi sui moderni **multiprocessori**, che per efficienza possono **riordinare gli accessi alla memoria**: se la scrittura di *interested* diventa visibile agli altri core dopo la lettura successiva, la logica dell'algoritmo salta. Per impedirlo servono apposite istruzioni dette **memory barrier**. Anche Peterson, come l'alternanza stretta, può essere generalizzato al caso di N processi.

DA RICORDARE

Come funziona la soluzione di Peterson? Quali strutture dati condivise usa?

Usa due variabili condivise: il vettore `interested[]`, in cui ogni processo dichiara di voler entrare nella sezione critica, e la variabile `turn`, in cui ogni processo che vuole entrare scrive il proprio numero. Un processo aspetta, in busy waiting, solo se **l'altro è interessato e lui è stato l'ultimo a scrivere turn**; in tutti gli altri casi entra. Se i due arrivano insieme, la scrittura successiva in turn sovrascrive la precedente, quindi esattamente uno dei due entra. All'uscita il processo azzera il proprio interesse e sblocca l'altro. Rispetta tutte e quattro le condizioni, ma spreca CPU con l'attesa attiva e sui multiprocessori moderni richiede memory barrier, per questo nella pratica si preferiscono le istruzioni hardware.

9.5 Le istruzioni TSL e XCHG

Molte architetture, soprattutto multiprocessore, offrono un aiuto hardware. L'istruzione **TSL** (*Test and Set Lock*), nella forma `TSL REGISTRO, LOCK`, copia nel registro il valore della variabile `lock` in memoria e contemporaneamente vi scrive un valore diverso da zero. Le due azioni avvengono in un'**unica operazione atomica**: nessun altro può intromettersi fra la lettura e la scrittura. Per garantirlo anche con più CPU, durante l'esecuzione la TSL **blocca il bus della memoria**, così nessun altro core può accedere alla memoria finché l'istruzione non è conclusa. È proprio ciò che mancava alla variabile di lock del paragrafo 9.2.

CON TSL

```
enter_region:
    TSL REGISTER,LOCK    ; copia lock nel registro
                        ; e mette lock a 1
    CMP REGISTER,#0      ; lock era 0, cioè libero?
    JNE enter_region    ; no: riprova (spin lock)
    RET                  ; sì: entro

leave_region:
    MOVE LOCK,#0        ; libero il lock
    RET
```

CON XCHG

```
enter_region:
    MOVE REGISTER,#1    ; metto 1 nel registro
    XCHG REGISTER,LOCK  ; scambio atomico
    CMP REGISTER,#0     ; lock era 0, cioè libero?
    JNE enter_region    ; no: riprova (spin lock)
    RET                  ; sì: entro

leave_region:
    MOVE LOCK,#0        ; libero il lock
    RET
```

Un'istruzione equivalente è **XCHG** (*eXCHanGe*), disponibile su tutte le CPU Intel x86, che **scambia atomicamente** il contenuto di un registro con quello di una locazione di memoria: mettendo 1 nel registro e scambiandolo con il lock, si ottiene nel registro il vecchio valore del lock e nel lock il valore 1, esattamente come con la TSL. Entrambe le istruzioni sono eseguibili in modalità utente, senza system call, ma da sole **non bloccano nulla**: si limitano a leggere e scrivere un valore, e il processo che trova il lock occupato continua a riprovare. Siamo quindi **ancora in busy waiting**.

DA RICORDARE

Che cosa significa che un'operazione è atomica?

Che è **indivisibile**: viene eseguita tutta insieme, senza che nessun altro processo, thread o core possa intromettersi a metà o osservare uno stato intermedio. Per la prelazione, un'istruzione atomica viene eseguita o tutta prima o tutta dopo un'eventuale interruzione. La TSL ne è l'esempio: lettura del vecchio valore del lock e scrittura del nuovo avvengono in un'unica operazione, con il bus della memoria bloccato, così due processi non possono mai leggere entrambi il valore «libero».

Sleep e wakeup, semafori, mutex

Gli strumenti che permettono di attendere senza consumare CPU: sleep e wakeup, semafori, mutex e futex, applicati al problema del produttore e consumatore.

10.1 Il problema dell'attesa attiva

Tutte le soluzioni viste finora fanno **spin lock**: il processo che non può entrare resta in esecuzione a controllare una variabile. Oltre a sprecare CPU, l'attesa attiva può causare il **problema dell'inversione di priorità**. Supponiamo che lo scheduler esegua sempre il processo pronto a priorità più alta, e che due processi, H ad alta priorità e L a bassa priorità, condividano un lock. L entra nella sezione critica; a quel punto H diventa pronto, lo scheduler gli dà subito la CPU e H inizia il busy waiting sul lock. Ma L, che detiene il lock, non verrà mai più eseguito, perché H è sempre pronto e ha priorità maggiore. H cicla all'infinito aspettando un lock che nessuno rilascerà: il sistema è bloccato.

La soluzione è permettere a un processo di **bloccarsi in modo passivo**: invece di girare a vuoto, il processo viene **tolto dai processi pronti** e risvegliato solo quando può proseguire. Servono due primitive: **sleep**, che blocca il processo chiamante, e **wakeup**, che risveglia un processo indicato.

10.2 Produttore e consumatore con sleep e wakeup

Il banco di prova di ogni meccanismo di sincronizzazione è il **problema del produttore e consumatore**. Due processi condividono un **buffer limitato** di N posizioni: il **produttore** genera elementi e li inserisce nel buffer, il **consumatore** li preleva e li utilizza. Il produttore deve fermarsi se il buffer è **pieno**, il consumatore se è **vuoto**. Un esempio concreto è lo streaming video: il server produce pezzi del video nel buffer, il lettore li consuma, e il buffer permette di continuare a guardare anche se la rete rallenta per un attimo.

Un primo tentativo usa una variabile condivisa count con il numero di elementi nel buffer:

PRODUTTORE

```
function producer()
  while (true) do
    item = produce_item()
    if (count == N) sleep()    // buffer pieno
    insert_item(item)
    count = count + 1
    if (count == 1)          // era vuoto:
      wakeup(consumer)      // lo sveglio
```

CONSUMATORE

```
function consumer()
  while (true) do
    if (count == 0) sleep()   // buffer vuoto
    item = remove_item()
    count = count - 1
    if (count == N - 1)      // era pieno:
      wakeup(producer)      // lo sveglio
    consume_item(item)
```

La soluzione **non funziona**, perché l'accesso a count non è protetto. Il buffer è vuoto e il consumatore legge count, trovando 0; prima che riesca ad addormentarsi, lo scheduler passa al produttore. Il produttore inserisce un elemento, porta count a 1 e, vedendo che il buffer era vuoto, chiama wakeup(consumer). Ma il consumatore **non sta ancora dormendo**, quindi la sveglia **va persa**. Quando il consumatore riprende, ricorda di aver letto 0 e si addormenta. Il produttore riempie il buffer e si addormenta a sua volta: entrambi dormono per sempre.

Una pezza è aggiungere un **bit di attesa wakeup** (*wakeup waiting bit*): una wakeup inviata a un processo sveglia accende il bit, e la successiva sleep di quel processo, trovando il bit acceso, lo spegne e non blocca. Funziona con due processi, ma con più produttori e consumatori un solo bit non basta: servirebbe ricordare **quante** sveglie sono pendenti. È esattamente l'idea del semaforo.

10.3 I semafori

Il **semaforo**, introdotto da Dijkstra, generalizza sleep e wakeup. È una **variabile intera condivisa**, che non può mai diventare negativa, su cui si eseguono soltanto due operazioni, **down** e **up** (dette anche *wait* e *signal*, o P e V):

- **down(S)**: se S è maggiore di 0, lo decrementa e il processo prosegue; se S vale 0, il processo **si blocca** sul semaforo, senza completare per ora la down;
- **up(S)**: incrementa S; se c'erano processi bloccati su quel semaforo, il sistema ne sceglie uno e gli fa completare la sua down. In questo caso il valore torna subito a 0: la up ha di fatto «passato» la risorsa al processo risvegliato. Una up non blocca mai.

DEFINIZIONE

Un **semaforo** è una variabile intera condivisa, mai negativa, accessibile soltanto tramite due operazioni atomiche: **down**, che decrementa il valore se è positivo e altrimenti sospende il chiamante, e **up**, che risveglia uno dei processi sospesi oppure, se non ce ne sono, incrementa il valore.

A differenza della sveglia del paragrafo precedente, una up eseguita quando nessuno aspetta **non va persa**: resta memorizzata nel valore del semaforo, e la prossima down la «consumerà» senza bloccarsi. Il semaforo è quindi un **contatore di sveglie pendenti**.

Down e up devono essere **operazioni atomiche**: controllare il valore, modificarlo ed eventualmente bloccare il processo devono avvenire come un'unica azione indivisibile, altrimenti il semaforo stesso sarebbe soggetto a corse critiche. Poiché i semafori sono realizzati dal sistema operativo come system call, l'atomicità si ottiene nel kernel: su un sistema con **una sola CPU** basta **disabilitare gli interrupt** per le poche istruzioni necessarie; con **più CPU** si protegge il semaforo con un lock realizzato con **TSL o XCHG**. In questo secondo caso c'è ancora un brevissimo spin lock, ma dura solo le poche istruzioni che servono ad aggiornare il semaforo, non l'intera sezione critica. Il semaforo vero e proprio è implementato **senza busy waiting**: a ogni semaforo è associata una **lista dei processi bloccati** su di esso.

I semafori hanno **due usi** distinti:

- **semaforo mutex**, per la **mutua esclusione**: inizializzato a 1, ogni processo esegue down prima della sezione critica e up dopo. Il primo che fa down porta il valore a 0 ed entra; chi arriva dopo si blocca finché il primo non fa up. Un semaforo usato così assume solo i valori 0 e 1 e si dice **binario**;
- **conteggio delle risorse**, per la **sincronizzazione**: il semaforo è inizializzato al numero di risorse disponibili (ad esempio le posizioni libere di un buffer); ogni down ne prende una e blocca il processo se sono finite, ogni up ne restituisce una.

DA RICORDARE

Un semaforo mutex coordina alcuni processi su strutture condivise. Al termine delle operazioni di tutti i processi, che valore ci aspettiamo per il semaforo?

1. Il mutex parte da 1 (sezione critica libera), e ogni processo esegue una down all'ingresso e una up all'uscita. Ogni coppia down e up lascia il valore invariato, quindi quando tutti hanno finito il semaforo è tornato al valore iniziale, che indica la sezione critica di nuovo libera.

ESEMPIO SVOLTO: SEGUIRE L'ESECUZIONE CON I SEMAFORI

Tre processi condividono la variabile x, inizialmente 1; i semafori valgono S = 1, R = 0, T = 0. Che cosa stampa P3?

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)		

Il metodo è cercare chi può partire. P2 si blocca subito su wait(R) perché R vale 0, e P3 su wait(T) perché T vale 0. Solo P1 può avanzare: wait(S) porta S a 0, x diventa $1 - 2 = -1$, signal(T) porta T a 1. Poi P1 esegue di nuovo wait(S), ma S vale 0 e nessuno farà mai signal(S): **P1 resta bloccato per sempre** e l'istruzione $x = x - 1$ non verrà mai eseguita.

Ora T vale 1, quindi P3 supera wait(T) (T torna 0). Poiché $x = -1$ è negativo, P3 esegue signal(R), che risveglia P2. P3 si blocca sulla seconda wait(T). P2 esegue $x = -1 + 2 = 1$ e signal(T), che risveglia P3; poi P2 si blocca su wait(R). P3 stampa il valore attuale di x: l'output è **1**. Non esistono altri ordini possibili, perché ogni passo è forzato dai semafori.

10.4 Produttore e consumatore con i semafori

Con i semafori il problema si risolve in modo pulito. Ne servono tre: mutex, inizializzato a 1, garantisce che produttore e consumatore non accedano al buffer nello stesso momento; empty, inizializzato a N, conta le **posizioni vuote**; full, inizializzato a 0, conta le **posizioni piene**. Gli ultimi due servono alla sincronizzazione: il produttore si blocca su empty quando il buffer è pieno, il consumatore su full quando è vuoto.

```
int N = 100
semaphore mutex = 1    // accesso esclusivo al buffer
semaphore empty = N    // posizioni vuote
semaphore full = 0     // posizioni piene
```

PRODUTTORE

```
function producer()
  while (true) do
    item = produce_item()
    down(empty) // un posto libero?
    down(mutex) // entro in sez. critica
    insert_item(item)
    up(mutex)   // esco dalla sez. critica
    up(full)    // un posto pieno in più
```

CONSUMATORE

```
function consumer()
  while (true) do
    down(full) // c'è un elemento?
    down(mutex) // entro in sez. critica
    item = remove_item()
    up(mutex)   // esco dalla sez. critica
    up(empty)   // un posto vuoto in più
    consume_item(item)
```

DA RICORDARE

L'ordine delle down è fondamentale.

Le due down vanno fatte **prima sul semaforo di conteggio e poi sul mutex**. Se il consumatore facesse prima down(mutex) e poi down(full) con il buffer vuoto, si bloccherebbe su full **tenendo il mutex**: il produttore, per inserire un elemento, si bloccherebbe a sua volta su down(mutex), e nessuno dei due potrebbe più proseguire. È uno **stallo (deadlock)**. L'ordine delle up invece non crea problemi, perché una up non blocca mai.

DA RICORDARE

Nella soluzione con i semafori mutex, full ed empty, qual è l'esatta sequenza di operazioni del produttore per l'inserimento?

`down(empty)`, `down(mutex)`, `insert(item)`, `up(mutex)`, `up(full)`. Il produttore prima si assicura che ci sia un posto libero (e se non c'è si blocca senza tenere il mutex), poi entra nella sezione critica, inserisce, esce, e infine segnala che c'è un elemento in più. Le alternative che fanno `down(mutex)` per prima o che usano `up(mutex)` al posto della `down` sono sbagliate: la prima rischia lo stallo, la seconda non garantisce la mutua esclusione.

10.5 Mutex e thread a livello utente

Quando serve solo la mutua esclusione, senza contare risorse, si usa una versione semplificata del semaforo: il **mutex**, una variabile che può essere in due stati, **libero** (0) o **bloccato** (diverso da 0), con due operazioni, `mutex_lock` e `mutex_unlock`. Fra i **thread utente** di un unico processo (modello uno a molti) i mutex si possono realizzare in modo molto efficiente, senza passare dal kernel, usando la TSL:

ACQUISIZIONE

```
mutex_lock:
    TSL REGISTER,MUTEX ; copia e mette a 1
    CMP REGISTER,#0    ; il mutex era libero?
    JZE ok             ; sì: ho preso il lock
    CALL thread_yield  ; no: cedo la CPU
    JMP mutex_lock     ; quando riparto, riprovo
ok:
    RET
```

RILASCIO

```
mutex_unlock:
    MOVE MUTEX,#0      ; libero il mutex
    RET
```

Il codice è quasi identico a `enter_region`, con una differenza decisiva: se il lock è occupato, invece di ripetere subito il test il thread chiama `thread_yield` e **cede la CPU**. Con i processi lo spin lock è solo uno spreco, perché prima o poi il clock scade e il processo che detiene il lock torna in esecuzione e lo rilascia. Con i **thread utente**, invece, il clock non interviene fra i thread di uno stesso processo (lo abbiamo visto nella parte sui thread): un thread che facesse spin lock **girerebbe per sempre**, impedendo proprio al thread che detiene il lock di essere eseguito e liberarlo. Con la `yield` il problema sparisce. Le chiamate sono velocissime, perché non entrano mai nel kernel.

DA RICORDARE

Con thread utente (modello uno a molti) le soluzioni usate per processi e thread kernel hanno problemi. Scrivere in pseudoassembly `mutex_lock` e `mutex_unlock` basati su TSL.

Il codice è quello del riquadro qui sopra: `mutex_lock` esegue la TSL sul mutex e confronta con 0 il valore letto; se il mutex era libero salta a `ok` e ritorna con il lock acquisito, altrimenti chiama `thread_yield` e poi ripete dall'inizio. `mutex_unlock` rimette il mutex a 0. Il punto centrale è il perché della `yield`: fra thread utente non c'è prelazione, quindi uno spin lock non terminerebbe mai, mentre cedendo la CPU si permette al thread che detiene il lock di proseguire e rilasciarlo. Senza la `yield` il thread non viene messo in pausa: continua a girare a vuoto consumando tutta la CPU del processo.

10.6 I futex

I mutex nello spazio utente sono molto efficienti, ma se il lock resta occupato a lungo l'attesa, anche con la `yield`, continua a consumare cicli. D'altra parte un semaforo del kernel non spreca CPU, ma **ogni** `down` e `up` è una system call, anche quando il lock è libero e non ci sarebbe nulla da attendere. Linux risolve il dilemma con il **futex** (*fast user space mutex*), formato da due componenti:

- una **libreria utente**, che gestisce la **variabile di lock** e la **contesa in modalità utente**, con istruzioni atomiche tipo TSL o XCHG;
- un **servizio del kernel**, che mantiene una **coda di thread bloccati** in attesa del lock.

Quando un thread vuole il lock, la libreria prova ad acquisirlo con un'istruzione atomica, **senza entrare nel kernel**. Solo se il lock è occupato il thread esegue una system call per **bloccarsi** davvero nella coda del kernel; chi rilascia il lock chiama il kernel solo se c'è qualcuno da svegliare. Poiché nella maggior parte dei casi il lock è libero, il costo del kernel si paga solo quando c'è effettivamente contesa. I mutex della libreria Pthreads su Linux sono realizzati proprio con i futex.

DA RICORDARE

Quali sono i metodi moderni per realizzare la mutua esclusione con basso overhead? Perché il futex non ha costi quando il lock è libero?

Le istruzioni atomiche come **TSL** e i **futex**. La TSL permette di prendere un lock libero con una sola istruzione, senza system call, ma se il lock è occupato porta al busy waiting. Il futex combina i due mondi: il tentativo di acquisizione avviene **in spazio utente** con un'istruzione atomica, quindi se il lock è libero non c'è alcuna system call; solo se il lock è occupato il thread chiede al kernel di bloccarlo in una coda, senza consumare CPU durante l'attesa. È veloce come la TSL nel caso comune ed efficiente come un semaforo quando bisogna aspettare.

Monitor e scambio di messaggi

Due meccanismi di sincronizzazione di livello più alto: il monitor, che porta la mutua esclusione nel linguaggio, e lo scambio di messaggi.

11.1 I monitor

I semafori funzionano, ma sono strumenti di **basso livello**: basta sbagliare l'ordine di due down, o dimenticare una up, per ottenere uno stallò o una corsa critica, ed errori del genere sono difficili da trovare. Per questo alcuni linguaggi offrono un costrutto ad alto livello, il **monitor**. Un monitor è un **tipo di dato astratto**, cioè un insieme di variabili e di procedure raccolte in un modulo, con tre garanzie:

- **mutua esclusione**: in ogni istante al più un thread può essere attivo all'interno del monitor, cioè eseguire una delle sue procedure. Chi chiama una procedura mentre un altro thread è dentro viene messo in una **coda di attesa interna** al monitor;
- **vincolo di accesso ai dati**: le variabili interne al monitor sono accessibili solo dalle sue procedure, e le procedure non accedono a dati esterni;
- la mutua esclusione è realizzata **dal compilatore**, non dal programmatore: è il compilatore a inserire il codice di lock e unlock (appoggiandosi alle primitive del sistema operativo, ad esempio i futex su Linux), quindi non si può dimenticare.

DEFINIZIONE

Un **monitor** è un costrutto di linguaggio che raccoglie variabili e procedure in un modulo: i processi possono accedere alle variabili solo attraverso le procedure, e al più un processo alla volta può essere attivo nel monitor. La mutua esclusione è quindi garantita dal compilatore; per le attese si usano le **variabili condizione** con le operazioni wait e signal.

La mutua esclusione da sola non basta: serve anche un modo per **aspettare** che una condizione si verifichi, ad esempio che il buffer non sia più pieno. A questo servono le **variabili condizione**, su cui si eseguono due operazioni. Con **wait(c)** il thread si sospende sulla variabile c e, cosa fondamentale, **rilascia il monitor**, così che un altro thread possa entrare e rendere vera la condizione attesa. Con **signal(c)** si risveglia uno dei thread sospesi su c. A ogni variabile condizione è associata una **coda di attesa**.

Le variabili condizione **non sono contatori**: se si esegue una signal su una variabile su cui nessuno aspetta, la signal **va persa**, a differenza della up di un semaforo. Non è un problema, perché grazie alla mutua esclusione del monitor nessuno può inserirsi fra il controllo della condizione e la wait: un thread decide di sospendersi solo dopo aver controllato lo stato, e nessuno può modificarlo nel frattempo.

11.2 Le semantiche della signal

Quando un thread esegue una signal e risveglia un altro thread, si avrebbero due thread attivi nel monitor, cosa vietata. Bisogna quindi decidere chi prosegue. Le soluzioni sono tre:

- **monitor di Hoare**, semantica *signal and wait*: chi esegue la signal si sospende subito e cede il monitor al thread risvegliato, riprendendo quando questo esce. È la proposta teorica originale: ha il pregio che il thread risvegliato trova la condizione esattamente come l'ha lasciata chi l'ha segnalata, ma è più complessa da realizzare;

- **monitor Mesa**, semantica *signal and continue*, usata ad esempio da **Java**: chi esegue la signal continua la sua esecuzione, e il thread risvegliato viene solo reso pronto e rientrerà nel monitor più tardi, quando sarà libero;
- **compromesso** di Brinch Hansen, adottato dal Concurrent Pascal, semantica *signal and return*: la signal deve essere l'**ultima istruzione** della procedura, quindi chi la esegue esce subito dal monitor. È la più semplice, anche concettualmente.

DA RICORDARE

Con la semantica Mesa la condizione va ricontrollata.

Con *signal and continue* fra il momento della signal e quello in cui il thread risvegliato rientra nel monitor possono essere entrati altri thread, che magari hanno reso di nuovo falsa la condizione. Per questo con i monitor Mesa la condizione si controlla con un **while** e non con un **if**: al risveglio il thread ricontrolla e, se serve, torna in wait. Gli pseudocodici di questa parte usano l'if, valido con la semantica di Hoare. In laboratorio, dove mutex e variabili condizione di Pthreads seguono la semantica Mesa, va usato il while.

11.3 Produttore e consumatore con i monitor

IL MONITOR

```
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)
    remove = remove_item()
    count = count - 1
    if count == N-1 then signal(full)
```

PRODUTTORE E CONSUMATORE

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

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

// nessuna down/up esplicita: la mutua
// esclusione su count e sul buffer è
// garantita dal monitor stesso
```

La struttura ricorda la soluzione con sleep e wakeup, ma qui funziona: il controllo di count e l'eventuale wait avvengono dentro il monitor, in mutua esclusione, quindi la sveglia non può perdersi fra i due. Se una signal viene eseguita quando nessuno aspetta, non succede nulla di male: vuol dire semplicemente che nessuno ne aveva bisogno.

11.4 Semafori e monitor a confronto

Il **vantaggio** dei monitor è che rendono la programmazione concorrente molto meno soggetta a errori: la mutua esclusione è automatica e il codice è più compatto. Gli **svantaggi** sono due. Il monitor è un **costrutto del linguaggio**: richiede un compilatore che lo supporti (Java, C#), mentre C e C++ non lo hanno; i semafori invece sono system call, usabili da qualunque linguaggio. Soprattutto, il monitor si basa su dati **condivisi in memoria**, quindi serve a coordinare i **thread di uno stesso processo**: due processi distinti hanno spazi degli indirizzi separati e non possono condividere un monitor. I semafori, che vivono nel kernel fuori dai processi, possono invece coordinare anche processi indipendenti, purché sulla stessa macchina. Né i semafori né i monitor funzionano invece fra macchine diverse, che non hanno memoria in comune: per questo serve lo scambio di messaggi.

DA RICORDARE

Qual è la differenza principale fra semafori e monitor?

Il semaforo è un meccanismo di **basso livello** offerto dal sistema operativo: un contatore con le operazioni atomiche down e up, utilizzabile da qualsiasi linguaggio e anche fra processi indipendenti; la mutua esclusione dipende dal programmatore, che deve chiamare down e up al posto giusto. Il monitor è un costrutto di **alto livello** del linguaggio: la mutua esclusione è garantita automaticamente dal compilatore, e la sincronizzazione si ottiene con le variabili condizione. Funziona fra i thread di uno stesso processo. Un'altra differenza: una up su un semaforo viene **ricordata** nel contatore anche se nessuno aspetta, mentre una signal su una variabile condizione senza thread in attesa **va persa**, anche se ripetuta più volte.

Fra queste affermazioni su semafori e monitor, qual è vera?

Le alternative errate sostenevano che i semafori sono più ad alto livello dei monitor, che le signal multiple su una variabile condizione vengono preservate come per i semafori, che i semafori si possono usare solo nei linguaggi che li supportano e che i monitor garantiscono la mutua esclusione su memoria condivisa fra processi indipendenti: sono tutte false per quanto visto sopra. L'affermazione vera è che **le variabili condizione non sono necessarie per garantire la mutua esclusione**: quella la garantisce già il monitor, e le variabili condizione servono solo alla sincronizzazione.

DA RICORDARE

Che cosa succede quando un thread prova a entrare in un monitor occupato, e che cosa succede se, una volta dentro, deve aspettare un dato?

Nel primo caso il thread viene **bloccato** automaticamente nella coda di ingresso del monitor: il codice inserito dal compilatore usa una primitiva bloccante, quindi il thread dorme senza consumare CPU, e viene risvegliato quando il monitor si libera. Nel secondo caso il thread esegue `wait` su una variabile condizione: si sospende nella coda di quella variabile e **rilascia il monitor**, così un altro thread può entrare e produrre il dato. Quando questo esegue `signal` sulla stessa variabile, il thread sospeso viene risvegliato e rientra nel monitor riprendendo subito dopo la `wait`. È questo che il monitor offre in più rispetto a un semplice mutex: la possibilità di aspettare una condizione senza tenere occupata la sezione critica.

11.5 Lo scambio di messaggi

Lo **scambio di messaggi** usa due primitive di più alto livello, offerte come system call:

- `send(destinazione, messaggio)` invia un messaggio a un destinatario;
- `receive(sorgente, messaggio)` riceve un messaggio da una sorgente. Se non ci sono messaggi la chiamata è **bloccante** per il chiamante, oppure può restituire subito un errore.

Il grande vantaggio è che il modello si **estende al caso di più macchine**, perché non richiede memoria condivisa: è lo stesso modello usato nelle reti, e la libreria **MPI** (*Message Passing Interface*) lo usa per il calcolo parallelo e distribuito sui cluster. I messaggi si possono indirizzare in modo **diretto**, specificando il processo destinatario, oppure tramite una **mailbox**, una casella intermedia in cui i messaggi vengono depositati e da cui vengono prelevati, così mittente e destinatario non devono conoscersi direttamente.

Realisticamente i messaggi in transito vengono conservati in un **buffer di capienza finita N**: finché c'è posto la `send` deposita il messaggio e prosegue, ma quando il buffer è pieno anche **la send diventa bloccante**, finché il destinatario non ne preleva uno. Al limite, senza alcun buffer, mittente e destinatario devono incontrarsi: la `send` blocca finché la `receive` non avviene (*rendezvous*). Produttore e consumatore diventa semplicissimo:

PRODUTTORE

```
function producer()
while (true) do
    item = produce_item()
    build_msg(msg, item) // impacchetto
    send(consumer, msg) // lo spedisco
```

CONSUMATORE

```
function consumer()
while (true) do
    receive(producer, msg) // aspetto
    item = extract_msg(msg) // estraggo
    consume_item(item)
```

Il prezzo è l'**efficienza**: ogni messaggio richiede una system call per l'invio, una **copia** dallo spazio del mittente al buffer del kernel e un'altra copia nello spazio del destinatario. Fra macchine diverse si aggiungono i problemi tipici delle reti: messaggi **persi** (da gestire con conferme e ritrasmissioni), duplicati, e l'**autenticazione** dei processi che comunicano.

Problemi classici di sincronizzazione

Due problemi di sincronizzazione che fanno da banco di prova per ogni meccanismo: i cinque filosofi e i lettori e scrittori.

12.1 Il problema dei cinque filosofi

Cinque filosofi siedono a un tavolo circolare, ciascuno davanti a un piatto di spaghetti; fra ogni coppia di piatti c'è **una forchetta**, cinque in tutto. Ogni filosofo alterna periodi in cui **pensa** e periodi in cui **mangia**, e per mangiare ha bisogno di **entrambe** le forchette ai lati del proprio piatto. Il problema modella l'**accesso esclusivo a un numero limitato di risorse** da parte di processi in concorrenza, in cui ogni processo ha bisogno di più risorse insieme. Con cinque forchette possono mangiare al massimo **due filosofi alla volta**, purché non siano vicini. Numeriamo filosofi e forchette da 0 a 4: il filosofo i usa le forchette i e $(i+1) \bmod N$.

SOLUZIONE 1: LA PIÙ OVVIA

```
int N = 5
function philosopher(int i)
    while (true) do
        think()
        take_fork(i)           // prende la prima forchetta (bloccante)
        take_fork((i+1) mod N) // prende la seconda forchetta (bloccante)
        eat()
        put_fork(i)
        put_fork((i+1) mod N)
```

La **soluzione 1** non funziona: se tutti e cinque i filosofi prendono contemporaneamente la prima forchetta, nessuno troverà libera la seconda, e tutti resteranno bloccati per sempre. È uno **stallo** (*deadlock*): un insieme di processi, ciascuno in attesa di una risorsa posseduta da un altro processo dell'insieme. Si possono considerare altri tre tentativi:

- **soluzione 2**, *controlla con rilascio e riprova*: dopo aver preso la prima forchetta, il filosofo controlla se la seconda è libera; se non lo è posa la prima, aspetta un tempo fissato e riprova. Se tutti partono insieme, tutti prendono la prima forchetta, tutti trovano la seconda occupata, tutti posano, aspettano lo stesso tempo e ricominciano, all'infinito. Nessuno è bloccato, tutti continuano a eseguire, ma **nessuno fa progressi**: è un *livelock*, che porta i filosofi all'inedia (*starvation*);
- **soluzione 3**, *attesa casuale*: come la 2, ma il tempo di attesa prima di riprovare è **casuale**. La probabilità che tutti si risincronizzino diventa piccolissima, ed è la stessa idea usata da Ethernet dopo una collisione. Funziona quasi sempre, ma non dà **garanzie**, e in certe applicazioni non è accettabile;
- **soluzione 4**, *un semaforo mutex*: si protegge l'intera sequenza da `take_fork` a `put_fork` con un unico mutex. È corretta, ma permette a **un solo filosofo alla volta** di mangiare, sprecando una forchetta che potrebbe servire a un secondo filosofo.

La soluzione corretta e con il massimo parallelismo tiene traccia dello **stato** di ogni filosofo: THINKING, HUNGRY (vuole mangiare) o EATING. Un filosofo può passare a EATING solo se nessuno dei due vicini sta mangiando. Si usano un vettore `state`, un mutex che lo protegge e un **semaforo privato** `s[i]` per ogni filosofo, su cui il filosofo si blocca se le forchette non sono disponibili.

SOLUZIONE CON I SEMAFORI

```
int N = 5; int THINKING = 0; int HUNGRY = 1; int EATING = 2
int state[N] // stato di ogni filosofo
semaphore mutex = 1 // protegge il vettore state
semaphore s[N] = {0, ..., 0} // un semaforo privato per filosofo

function left(int i) = (i + N - 1) mod N // vicino di sinistra
function right(int i) = (i + 1) mod N // vicino di destra

function philosopher(int i)
    while (true) do
        think()
        take_forks(i) // prende entrambe le forchette o si blocca
        eat()
        put_forks(i) // posa le forchette

function take_forks(int i)
    down(mutex) // entro in sezione critica su state
    state[i] = HUNGRY // dichiaro che ho fame
    test(i) // provo a prendere le forchette
    up(mutex) // esco dalla sezione critica
    down(s[i]) // mi blocco se test non ha avuto successo

function put_forks(int i)
    down(mutex)
    state[i] = THINKING // ho finito di mangiare
    test(left(i)) // il vicino di sinistra ora può mangiare?
    test(right(i)) // e quello di destra?
    up(mutex)

function test(int i)
    if state[i] == HUNGRY and state[left(i)] != EATING
        and state[right(i)] != EATING
        state[i] = EATING // le due forchette sono libere: mangio
        up(s[i]) // sblocco (o non blocca) il filosofo i
```

Il funzionamento ruota attorno a test. Quando un filosofo ha fame, take_forks lo segna HUNGRY e chiama test su di lui: se i vicini non stanno mangiando, lo segna EATING ed esegue up(s[i]), così la successiva down(s[i]) passa senza bloccare. Se invece un vicino sta mangiando, s[i] resta a 0 e il filosofo si blocca sulla down. Quando un vicino finisce, put_forks esegue test su entrambi i vicini: se uno di loro era HUNGRY e ora ha le forchette libere, viene segnato EATING e risvegliato dalla up. Da notare che la down(s[i]) sta **dopo** up(mutex): se il filosofo si bloccasse tenendo il mutex, nessun altro potrebbe più modificare state e quindi svegliarlo.

La **versione con i monitor** è identica nella logica, ma senza mutex: take_forks, put_forks e test sono procedure di un monitor, che garantisce già la mutua esclusione, e i semafori privati diventano variabili condizione self[i].

SOLUZIONE CON I MONITOR

```
monitor dp_monitor
int state[N]
condition self[N]          // una variabile condizione per filosofo

function take_forks(int i)
    state[i] = HUNGRY
    test(i)
    if state[i] != EATING then wait(self[i]) // forchette occupate: aspetto

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[i] == HUNGRY
        and state[right(i)] != EATING
        state[i] = EATING
        signal(self[i])          // risveglia il filosofo i, se aspetta

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

Quando test è chiamata da take_forks dello stesso filosofo, la sua signal non trova nessuno in attesa e va persa senza conseguenze: il filosofo non farà la wait, perché il suo stato è già EATING. La stessa signal diventa invece essenziale quando test è chiamata da put_forks di un vicino, perché risveglia un filosofo che stava davvero aspettando.

DA RICORDARE

Perché la soluzione più semplice al problema dei filosofi va in stallo, e come lo evita la soluzione con i semafori?

Nella soluzione semplice ogni filosofo prende una forchetta alla volta e si blocca se la seconda è occupata: se tutti prendono la prima forchetta nello stesso istante, ciascuno aspetta la forchetta tenuta dal vicino e nessuno può proseguire. La soluzione con i semafori evita lo stallo perché un filosofo **non prende mai una sola forchetta**: dentro la sezione critica protetta dal mutex controlla se **entrambe** sono libere (i vicini non sono EATING) e solo in quel caso passa a EATING, altrimenti si blocca sul proprio semaforo privato **senza tenere risorse**. Quando un filosofo posa le forchette, controlla i due vicini e risveglia quello che ora può mangiare. Inoltre consente il massimo parallelismo: due filosofi non vicini mangiano insieme.

12.2 Il problema dei lettori e scrittori

Il secondo problema classico modella l'accesso a un **database** (o a una qualsiasi struttura dati condivisa) da parte di molti processi, alcuni dei quali leggono e altri scrivono. Più **lettori** possono accedere **contemporaneamente** senza alcun problema, perché nessuno modifica i dati; uno **scrittore** invece deve avere l'accesso **esclusivo**, senza altri scrittori né lettori. Un semplice mutex sull'intero database sarebbe corretto ma inutilmente lento, perché bloccherebbe anche le letture fra loro. Si pensi a un database di previsioni meteo, consultato da migliaia di utenti e aggiornato di rado.

LETTORE

```
semaphore mutex = 1 // protegge rc
semaphore db = 1    // accesso al database
int rc = 0          // lettori attivi

function reader()
  while (true) do
    down(mutex)
    rc = rc + 1
    if (rc == 1) down(db) // primo lettore
    up(mutex)
    read_database()
    down(mutex)
    rc = rc - 1
    if (rc == 0) up(db)   // ultimo lettore
    up(mutex)
    use_data_read()
```

SCRITTORE

```
function writer()
  while (true) do
    think_up_data()
    down(db)        // accesso esclusivo
    write_database()
    up(db)           // rilascio il database
```

Il **primo lettore** che entra esegue `down(db)`: se c'è uno scrittore al lavoro si blocca, altrimenti «chiude la porta» agli scrittori. I lettori successivi entrano direttamente, incrementando soltanto il contatore `rc`, protetto dal `mutex`. L'**ultimo lettore** che esce esegue `up(db)` e riapre la porta. Lo scrittore usa semplicemente `db` come un `mutex`. Il difetto è che **lo scrittore può attendere per un tempo indefinito**: se nuovi lettori continuano ad arrivare mentre altri stanno ancora leggendo, `rc` non torna mai a zero e lo scrittore resta bloccato. La soluzione dà quindi **priorità ai lettori**.

Con i **monitor** si possono realizzare politiche diverse. Il database resta **fuori** dal monitor (altrimenti anche le letture sarebbero in mutua esclusione): il monitor contiene solo le procedure che regolano l'ingresso e l'uscita, il contatore `rc`, un flag `busy_on_write` e due variabili condizione, `read` e `write`. Lettori e scrittori racchiudono il proprio accesso fra `start_read/end_read` e `start_write/end_write`. La funzione `in_queue(c)` dice se ci sono thread in attesa sulla variabile `c`.

SOLUZIONE N.1 CON I MONITOR

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

function start_read()
    if (busy_on_write) wait(read)          // c'è uno scrittore: aspetto
    rc = rc + 1
    signal(read)                            // risveglio a catena degli altri lettori

function end_read()
    rc = rc - 1
    if (rc == 0) signal(write)             // ultimo lettore: può entrare uno scrittore

function start_write()
    if (rc > 0 or busy_on_write) wait(write) // c'è qualcuno dentro: aspetto
    busy_on_write = true

function end_write()
    busy_on_write = false
    if (in_queue(read)) signal(read)       // precedenza ai lettori in attesa
    else signal(write)

function reader()                          function writer()
while (true) do                             while (true) do
    rw_monitor.start_read()                think_up_data()
    read_database()                        rw_monitor.start_write()
    rw_monitor.end_read()                  write_database()
    use_data_read()                        rw_monitor.end_write()
```

La `signal(read)` alla fine di `start_read` produce un **risveglio a catena**: quando uno scrittore termina e risveglia un lettore, questo entra e risveglia il successivo, che a sua volta risveglia il successivo, finché la coda dei lettori non si svuota. Così tutti i lettori in attesa entrano insieme. Le tre soluzioni differiscono in due soli punti:

SOLUZIONE	UN NUOVO LETTORE ASPETTA SE...	LO SCRITTORE CHE ESCE RISVEGLIA...	EFFETTO
n.1	c'è uno scrittore al lavoro	prima i lettori in coda, poi uno scrittore	Priorità ai lettori : gli scrittori possono attendere all'infinito
n.2	c'è uno scrittore al lavoro o uno scrittore in coda (<code>in_queue(write)</code>)	prima i lettori in coda, poi uno scrittore	Bilanciata : nessuno dei due attende all'infinito
n.3	c'è uno scrittore al lavoro o uno scrittore in coda	prima gli scrittori in coda, poi i lettori	Priorità agli scrittori : i lettori possono attendere all'infinito

Lo scheduling: concetti di base

Che cosa deve decidere lo scheduler, in quali momenti interviene e con quali criteri si valuta la qualità delle sue scelte.

13.1 Scheduler e comportamento dei processi

Quando più processi sono pronti e la CPU è una sola, il sistema operativo deve decidere quale eseguire. La parte del sistema che prende questa decisione è lo **scheduler**, e la regola che usa è l'**algoritmo di scheduling**. Non esiste un algoritmo migliore in assoluto: ognuno è adatto a certi ambienti e a certi obiettivi.

Per scegliere bene bisogna capire come si comportano i processi. Quasi tutti **alternano** fasi di calcolo, i **CPU burst**, in cui usano la CPU, a fasi in cui aspettano un'operazione di I/O, gli **I/O burst**, in cui sono bloccati. In base a questo alternarsi si distinguono due tipi di processi:

- **CPU bound** (orientati al calcolo): passano la maggior parte del tempo a calcolare, con burst di CPU **lunghi** e richieste di I/O rare;
- **I/O bound** (orientati all'I/O): calcolano poco fra una richiesta di I/O e l'altra, con burst di CPU **brevi**. Si chiamano così non perché le loro richieste di I/O siano lunghe, ma perché calcolano poco fra l'una e l'altra.

Un principio guida è che conviene dare la CPU **presto** ai processi I/O bound: usano la CPU per poco, avviano subito la successiva operazione di I/O e la lasciano libera, così CPU e dispositivi lavorano in parallelo. Se invece un processo I/O bound aspetta a lungo la CPU, resta fermo in memoria senza motivo e anche il disco resta inattivo. La distinzione non è rigida: un processo può cambiare comportamento durante la sua vita, e lo scheduler dovrebbe accorgersene.

13.2 Quando interviene lo scheduler

Lo scheduler viene attivato in alcune situazioni precise:

- alla **creazione** di un processo, quando bisogna decidere se far proseguire il padre o il figlio, e alla sua **terminazione**, quando bisogna scegliere chi esegue al suo posto;
- quando un processo esegue una **chiamata bloccante** (ad esempio un'operazione di I/O) e si blocca, e quando arriva l'**interrupt** che segnala il completamento di quell'operazione, che rende di nuovo pronto un processo;
- in corrispondenza di **interrupt periodici** del clock.

Proprio sull'ultimo punto si distinguono due famiglie di algoritmi. Negli algoritmi **senza prelazione** (*non preemptive*) un processo, una volta ottenuta la CPU, la tiene finché non termina, non si blocca o non la rilascia volontariamente: gli interrupt del clock non causano cambi di processo. Negli algoritmi **con prelazione** (*preemptive*) un processo può eseguire al massimo per un intervallo di tempo fissato, il **quanto**; allo scadere, segnalato dall'interrupt del clock, lo scheduler gli toglie la CPU e può sceglierne un altro. La prelazione richiede quindi un clock hardware.

Lo scheduler si limita a **scegliere**. Chi esegue materialmente il passaggio al processo scelto è il **dispatcher**: effettua il cambio di contesto, passa alla modalità utente e salta al punto giusto del programma. Il tempo che il dispatcher impiega per fermare un processo e avviarne un altro si chiama **latenza di dispatch**, e deve essere il più breve possibile, perché è tempo in cui nessun processo utente avanza.

13.3 Obiettivi e metriche

Gli obiettivi dipendono dall'ambiente. Alcuni sono **comuni a tutti i sistemi**: l'**equità**, cioè dare a processi comparabili una quota comparabile di CPU, e il **bilanciamento** nell'uso delle risorse, cioè tenere occupate tutte le parti del sistema, CPU e dispositivi, il più possibile. Gli altri cambiano a seconda del tipo di sistema:

AMBIENTE	CARATTERISTICHE	OBIETTIVI TIPICI
Batch	Lavori non interattivi, nessun utente in attesa davanti al terminale (paghe, elaborazioni periodiche, calcoli)	Massimizzare il throughput ; minimizzare il tempo di turnaround e il tempo di attesa
Interattivi	Utenti che interagiscono e aspettano una reazione; anche i server	Minimizzare il tempo di risposta
Real time	Processi legati a eventi esterni con scadenze	Rispettare le scadenze e garantire un comportamento prevedibile

Le metriche da conoscere sono queste:

- **throughput** (produttività): il numero di lavori completati nell'unità di tempo;
- **tempo di turnaround** o **tempo di completamento**: il tempo che passa dall'**arrivo** di un lavoro alla sua **terminazione**, cioè istante di fine meno istante di arrivo;
- **tempo di attesa**: il tempo complessivo che il lavoro trascorre **nella coda dei pronti**, senza eseguire. Per lavori che usano solo la CPU è il turnaround meno la durata;
- **tempo di risposta**: il tempo che passa dall'arrivo di una richiesta alla prima reazione, cioè, negli esercizi, dall'arrivo alla **prima volta** in cui il processo ottiene la CPU.

Massimizzare il throughput non basta. Uno scheduler che scegliesse sempre i lavori più brevi completerebbe moltissimi lavori all'ora, ma un lavoro lungo potrebbe non essere mai eseguito, con un turnaround enorme. Per questo si guardano sempre più metriche insieme, quasi sempre come **medie** su tutti i processi.

DA RICORDARE

Come si calcolano i tempi negli esercizi.

Si disegna prima il **diagramma di Gantt**, cioè la sequenza temporale di chi usa la CPU, e poi per ogni processo si ricava:

- **turnaround** = istante in cui termina – istante di arrivo;
- **attesa** = turnaround – durata. Con gli algoritmi con prelazione un processo può attendere più volte: questa formula le somma automaticamente tutte, ed è il modo più sicuro per non dimenticarne nessuna;
- **risposta** = istante in cui ottiene la CPU per la prima volta – istante di arrivo. Negli algoritmi senza prelazione coincide con l'attesa.

Infine si fanno le medie. A parità di condizioni, quando due processi sono equivalenti per l'algoritmo (ad esempio stessa durata in SJF), si sceglie quello arrivato prima.

DA RICORDARE

In base a che cosa si sceglie un algoritmo di scheduling piuttosto che un altro?

In base al tipo di sistema, quindi alla metrica che conta di più, e alle informazioni disponibili. Nei sistemi **batch** contano throughput, turnaround e attesa, e se le durate dei lavori sono note si possono usare SJF o SRTN. Nei sistemi **interattivi** conta il tempo di risposta, quindi servono algoritmi con prelazione come Round Robin, che serve tutti i processi a turno e regolarmente. Nei sistemi **real time** contano il rispetto delle scadenze e la prevedibilità, anche a costo di non ottimizzare le medie. Non esiste un algoritmo migliore in assoluto.

Scheduling nei sistemi batch

Gli algoritmi pensati per i sistemi batch, dove conta il rendimento complessivo: FCFS, SJF e SRTN, con il calcolo dei tempi di attesa e di turnaround.

14.1 First Come First Served

L'algoritmo più semplice è **FCFS** (*First Come First Served*), cioè **per ordine di arrivo**. È **senza prelazione** e si realizza con una semplice **coda FIFO** dei processi pronti: il primo arrivato viene eseguito fino alla fine, poi il secondo, e così via; un processo che si blocca, quando torna pronto, va in fondo alla coda. È semplice da capire e da realizzare ed è **equo** nel senso dell'ordine di arrivo, ma può dare tempi medi molto alti.

Esempio: tre processi arrivano insieme all'istante 0, nell'ordine P1 (durata 24), P2 (3) e P3 (3). P1 viene eseguito da 0 a 24, P2 da 24 a 27, P3 da 27 a 30. Il **tempo medio di attesa** è $(0 + 24 + 27) / 3 = 17$ e il **tempo medio di completamento** è $(24 + 27 + 30) / 3 = 27$. Se gli stessi processi fossero arrivati nell'ordine P2, P3, P1, l'attesa media sarebbe stata $(0 + 3 + 6) / 3 = 3$. Un processo lungo in testa alla coda fa aspettare tutti quelli brevi che stanno dietro: è l'**effetto convoglio**, come un camion lento su una strada a una corsia.

14.2 Shortest Job First

Con **SJF** (*Shortest Job First*), detto anche scheduling **per brevità**, quando la CPU si libera si sceglie il lavoro pronto con la **durata minore**. È **senza prelazione** e **presuppone di conoscere in anticipo** la durata di ogni lavoro. Nei sistemi batch l'ipotesi è ragionevole, perché spesso gli stessi tipi di lavoro vengono eseguiti di continuo (transazioni bancarie, elaborazioni periodiche) e la loro durata si può stimare dalle esecuzioni precedenti.

SJF è **ottimale** per il tempo medio di attesa e di turnaround, ma **solo se i lavori sono tutti disponibili contemporaneamente**. L'intuizione è semplice: la durata del primo lavoro eseguito si somma all'attesa di tutti quelli che vengono dopo, quella del secondo all'attesa di tutti i successivi, e così via; mettendo per primi i lavori più brevi si minimizza la somma. Se però i lavori **arrivano in momenti diversi**, SJF non è più ottimale. Un controesempio: P1 e P2 arrivano a 0 con durate 2 e 4, mentre P3, P4 e P5 arrivano a 3, ciascuno con durata 1. SJF esegue P1 (da 0 a 2), poi P2 (da 2 a 6), che all'istante 2 è l'unico disponibile e non può essere interrotto, e infine P3, P4, P5: attesa media $(0 + 2 + 3 + 4 + 5) / 5 = 2,8$. Eseguendo invece prima P2 (da 0 a 4), poi P3, P4, P5 (da 4 a 7) e per ultimo P1 (da 7 a 9), l'attesa media è $(7 + 0 + 1 + 2 + 3) / 5 = 2,6$, migliore. SJF ha deciso all'istante 0 senza sapere che stavano per arrivare tre lavori brevissimi.

DA RICORDARE

Quale di queste affermazioni su SJF è falsa?

Le affermazioni vere sono: non usa la prelazione, ma può essere adattato per usarla (diventando SRTN); è difficile da impiegare nei sistemi operativi reali; è pensato per lavori non interattivi; richiede di conoscere in anticipo la durata dei lavori. L'affermazione **falsa** è che «il tempo di turnaround è ottimale in presenza di job con tempi di arrivo non nulli»: SJF è ottimale solo quando tutti i lavori sono disponibili fin dall'inizio, come mostra il controesempio con attesa media 2,8 contro 2,6.

14.3 Shortest Remaining Time Next

SRTN (*Shortest Remaining Time Next*) è la **versione con prelazione** di SJF. Lo scheduler sceglie sempre il processo con il **tempo residuo** minore, e ogni volta che arriva un nuovo processo confronta la sua durata con il tempo che manca al processo in esecuzione: se il nuovo arrivato è più breve, il processo corrente viene **interrotto** e rimesso fra i pronti, e parte il nuovo. Anche SRTN richiede di conoscere in anticipo le durate. Rispetto a SJF gestisce bene gli arrivi in momenti diversi, perché può correggere una scelta fatta quando i lavori brevi non erano ancora arrivati. Il prezzo è il rischio di **starvation** per i lavori lunghi, che possono essere scavalcati di continuo dall'arrivo di lavori brevi.

14.4 Un esempio completo

L'esempio seguente confronta i tre algoritmi sugli stessi quattro processi: P1 arriva a 0 e dura 7, P2 arriva a 2 e dura 4, P3 arriva a 4 e dura 1, P4 arriva a 5 e dura 4.



Diagrammi di Gantt per FCFS, SJF e SRTN sugli stessi quattro processi.

Con **FCFS** i processi vengono eseguiti nell'ordine di arrivo. Con **SJF** P1 parte da solo e, non essendoci prelazione, arriva fino a 7; a quel punto sono arrivati tutti, e si sceglie il più breve, P3 (durata 1); fra P2 e P4, entrambi di durata 4, si sceglie P2, arrivato prima. Con **SRTN** a 2 arriva P2 (4), più breve dei 5 che mancano a P1: P1 viene interrotto. A 4 arriva P3 (1), più breve dei 2 che mancano a P2: P2 viene interrotto. A 5, finito P3, arriva P4 (4), ma a P2 mancano solo 2, quindi riprende P2 fino a 7; poi P4 (4) batte P1 (5), e P1 termina per ultimo.

ALGORITMO	ATTESA P1, P2, P3, P4	ATTESA MEDIA	COMPLETAMENTO P1, P2, P3, P4	COMPLETAMENTO MEDIO
FCFS	0, 5, 7, 7	4,75	7, 9, 8, 11	8,75
SJF	0, 6, 3, 7	4	7, 10, 4, 11	8
SRTN	9, 1, 0, 2	3	16, 5, 1, 6	7

Si osserva il compromesso tipico: SRTN ottiene le medie migliori, ma lo fa a spese di P1, il processo più lungo, che con FCFS non aspettava affatto e con SRTN aspetta 9 unità di tempo.

DA RICORDARE

Perché SRTN usa la prelazione e SJF no, pur seguendo lo stesso principio?

Entrambi privilegiano il lavoro più breve; cambia il **momento** in cui scelgono. SJF sceglie solo quando la CPU si libera, perché il processo in esecuzione ha terminato: una volta partito, un lavoro arriva fino in fondo anche se nel frattempo ne arriva uno più breve. SRTN rivaluta la scelta **a ogni arrivo**: confronta la durata del nuovo processo con il **tempo residuo** di quello in esecuzione e, se il nuovo è più breve, interrompe quello corrente. È la versione con prelazione di SJF, e proprio per questo gestisce meglio gli arrivi scaglionati, dove SJF non è ottimale.

ESEMPIO SVOLTO: QUALE PROCESSO VIENE ESEGUITO PER ULTIMO CON SRTN?

Cinque processi: P1 arriva a 0 e dura 7; P2 arriva a 3 e dura 5; P3 arriva a 4 e dura 2; P4 arriva a 6 e dura 4; P5 arriva a 9 e dura 4. Si ragiona a ogni arrivo e a ogni terminazione, confrontando i tempi residui:

- a 0 c'è solo P1, che parte. A 3 arriva P2 (5), ma a P1 mancano 4: P1 continua;
- a 4 arriva P3 (2), più breve dei 3 che mancano a P1: **prelazione**, P3 esegue da 4 a 6;
- a 6 sono pronti P1 (3), P2 (5) e il nuovo P4 (4): vince P1, che termina a 9;
- a 9 arriva P5 (4); fra P2 (5), P4 (4) e P5 (4) si sceglie P4, arrivato prima di P5, che esegue da 9 a 13; poi P5 da 13 a 17 e infine P2 da 17 a 22.



Diagramma di Gantt con SRTN: P2 viene eseguito per ultimo.

Il processo eseguito per ultimo è **P2**, che pur essendo arrivato presto viene scavalcato da tutti i processi più brevi: un esempio concreto di come SRTN penalizzi i lavori lunghi.

Scheduling nei sistemi interattivi

Gli algoritmi per i sistemi interattivi, dove conta il tempo di risposta: Round Robin, priorità, code multiple e le loro varianti.

15.1 Round Robin

L'algoritmo di base dei sistemi interattivi è il **Round Robin** (RR), che si può vedere come la **versione con prelazione di FCFS**. I processi pronti stanno in una coda FIFO; a ognuno, a turno, viene assegnata la CPU per un **quanto di tempo** (*timeslice*). Se allo scadere del quanto il processo è ancora in esecuzione, gli viene tolta la CPU e viene rimesso **in fondo alla coda**. Se invece si blocca o termina prima della fine del quanto, la CPU passa subito al processo successivo; quando tornerà pronto, il processo rientrerà in fondo alla coda con un quanto nuovo.

Con n processi e un quanto di q millisecondi, ogni processo ottiene circa **$1/n$ della CPU** e, nel caso peggiore, aspetta **$(n - 1) \cdot q$ millisecondi** prima di tornare in esecuzione, cioè il tempo in cui tutti gli altri usano il loro quanto per intero. Il problema centrale è quanto debba essere lungo il quanto. Un quanto **troppo breve** causa troppi cambi di contesto, e ognuno costa tempo di CPU sprecato: l'**overhead** cresce. Un quanto **troppo lungo** peggiora il tempo di risposta, perché un utente può dover aspettare che tutti gli altri processi abbiano usato il loro lungo quanto; al limite, se il quanto è più lungo di tutti i CPU burst, il Round Robin degenera in FCFS. I valori tipici sono **da 20 a 50 ms**.

DA RICORDARE

Perché un quanto troppo piccolo aumenta l'overhead? Con un cambio di contesto di 5 ms e un quanto di 50 ms, qual è la percentuale di overhead?

Ogni scadenza del quanto comporta un cambio di contesto, che ha un costo fisso: salvare lo stato del processo uscente, caricare quello del processo entrante, eventualmente cambiare la mappa della memoria. In un ciclo completo la CPU lavora per un quanto e poi spende il tempo di un cambio di contesto, quindi la frazione sprecata è:

overhead = tempo di cambio di contesto / (quanto + tempo di cambio di contesto)

Con i dati dell'esercizio: $5 / (50 + 5) = 5 / 55 \approx 0,091$, cioè circa il **9%** del tempo di CPU va perso nei cambi di contesto. Più il quanto si avvicina al costo del cambio di contesto, più questa percentuale cresce: con un quanto di 5 ms sarebbe il 50%.

ESEMPIO SVOLTO: ORDINE DI ESECUZIONE CON ROUND ROBIN

Quattro processi in coda nell'ordine P1, P2, P3, P4, con durate 30, 15, 60 e 45 ms, e un quanto di 20 ms. In che ordine vengono eseguiti fino alla terminazione?

Si scorre la coda: ogni processo usa al massimo 20 ms; chi non ha finito torna in fondo con il tempo residuo. Primo giro: P1 (resta 10), P2 (finisce, gli servivano solo 15 ms), P3 (resta 40), P4 (resta 25). Secondo giro: P1 (finisce), P3 (resta 20), P4 (resta 5). Terzo giro: P3 (finisce, usa esattamente il suo quanto), P4 (finisce).

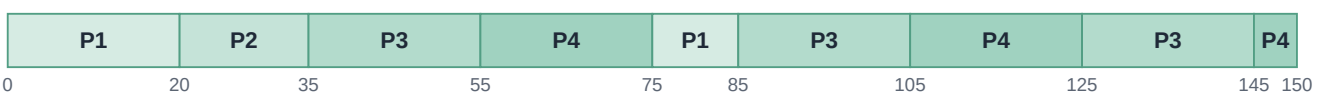
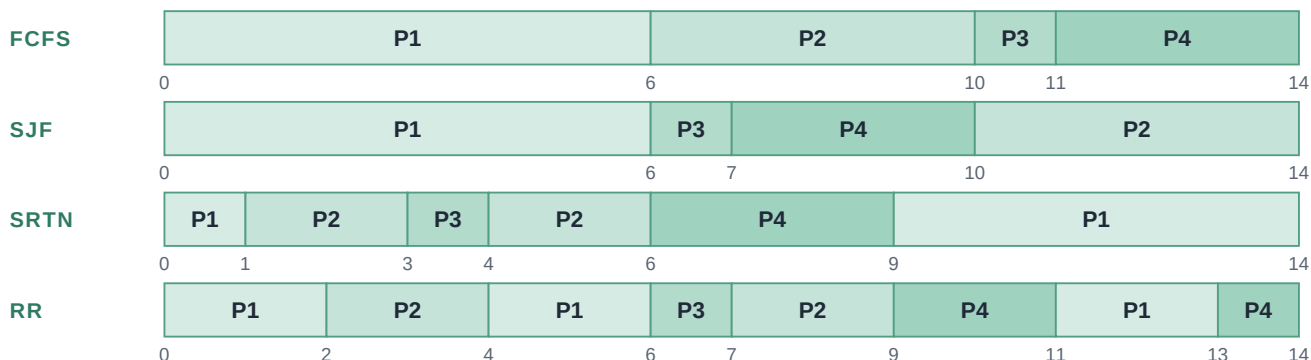


Diagramma di Gantt con Round Robin e quanto di 20 ms.

Ordine: **P1, P2, P3, P4, P1, P3, P4, P3, P4**. I processi terminano negli istanti 35 (P2), 85 (P1), 145 (P3) e 150 (P4).

ESEMPIO RIASSUNTIVO: FCFS, SJF, SRTN E ROUND ROBIN

Quattro processi: P1 arriva a 0 e dura 6; P2 arriva a 1 e dura 4; P3 arriva a 3 e dura 1; P4 arriva a 5 e dura 3. Calcolare attesa, risposta e turnaround medi con FCFS, SJF, SRTN e Round Robin con quanto 2.



Diagrammi di Gantt per FCFS, SJF, SRTN e Round Robin con quanto 2 sugli stessi quattro processi.

- **FCFS:** ordine di arrivo, nessuna sorpresa.
- **SJF:** P1 è solo a 0 e arriva fino a 6; a 6 sono pronti P2 (4), P3 (1) e P4 (3), e si eseguono in ordine di durata: P3, P4, P2.
- **SRTN:** a 1 P2 (4) interrompe P1 (residuo 5); a 3 P3 (1) interrompe P2 (residuo 2); a 4 riprende P2, e a 5 l'arrivo di P4 (3) non lo interrompe perché a P2 manca 1; a 6 P4 (3) batte P1 (5).
- **Round Robin, quanto 2:** chi arriva durante un quanto entra in coda prima del processo che sta per essere interrotto. A 2 la coda è P2, P1; a 4 è P1, P3, P2; a 6 è P3, P2, P4, P1; dopo P3 e P2 (che terminano a 7 e 9) restano P4 e P1, che si alternano fino a 14.

ALGORITMO	ATTESA DA P1 A P4	MEDIA	RISPOSTA DA P1 A P4	MEDIA	TURNAROUND DA P1 A P4	MEDIA
FCFS	0, 5, 7, 6	4,5	0, 5, 7, 6	4,5	6, 9, 8, 9	8
SJF	0, 9, 3, 2	3,5	0, 9, 3, 2	3,5	6, 13, 4, 5	7
SRTN	8, 1, 0, 1	2,5	0, 0, 0, 1	0,25	14, 5, 1, 4	6
RR (q = 2)	7, 4, 3, 6	5	0, 1, 3, 4	2	13, 8, 4, 9	8,5

Il confronto riassume le caratteristiche degli algoritmi: SRTN ha le medie migliori, ma richiede di conoscere le durate e penalizza P1; il Round Robin ha un turnaround peggiore, perché alterna continuamente i processi, ma ha un **tempo di risposta** basso e uguale per tutti, che è proprio ciò che serve in un sistema interattivo.

15.2 Scheduling a priorità

Il Round Robin tratta tutti i processi come ugualmente importanti, ma spesso non è così. Nello **scheduling a priorità** a ogni processo è associata una **priorità**, e la regola di base è semplice: **la CPU va al processo pronto con la priorità più alta**. Un processo a bassa priorità esegue solo quando non ci sono processi pronti a priorità maggiore. Le priorità possono essere assegnate in due modi:

- **statiche:** fissate alla creazione del processo e mai modificate, ad esempio in base all'importanza dell'utente o del servizio. In UNIX e Linux il comando `nice` permette di avviare un programma con una priorità modificata, con valori da -20 (più importante) a 19 (meno importante);
- **dinamiche:** ricalcolate dal sistema in base al comportamento del processo.

Un uso classico delle priorità dinamiche è **favorire i processi I/O bound**, per le ragioni viste all'inizio di questa parte. Un modo semplice è porre la priorità pari a $1/f$, dove f è la **frazione dell'ultimo quanto** effettivamente usata dal processo. Con un quanto di 50 ms, un processo che ha usato solo 1 ms prima di bloccarsi per I/O ha $f = 1/50$ e priorità 50; uno che ha usato 25 ms ha $f = 1/2$ e priorità 2; uno che ha usato tutto il quanto ha priorità 1. Anche **SJF** si può vedere come uno scheduling a priorità, in cui la priorità è tanto più alta quanto più breve è il lavoro.

Lo scheduling a priorità può essere **con o senza prelazione**: nel primo caso, se diventa pronto un processo con priorità maggiore di quello in esecuzione, quest'ultimo viene interrotto. Il rischio principale è la **starvation** (inedia): se continuano ad arrivare processi ad alta priorità, quelli a bassa priorità potrebbero non eseguire mai. Il rimedio è l'**aging** (invecchiamento): la priorità di un processo **aumenta gradualmente** man mano che attende, finché prima o poi diventa abbastanza alta da farlo eseguire. In alternativa si può diminuire la priorità del processo in esecuzione a ogni tick del clock.

ESEMPIO SVOLTO: SCHEDULING A PRIORITÀ CON PRELAZIONE

Cinque processi con arrivo, durata e priorità: A (0, 2, 1), B (0, 7, 2), C (3, 4, 3), D (4, 2, 5), E (5, 5, 4). In questo esercizio **a un numero più alto corrisponde una priorità più alta**: il testo dell'esercizio lo specifica sempre, ed è la prima cosa da controllare. Lo scheduling è con prelazione.

- A 0 sono pronti A (1) e B (2): esegue **B**. A 3 arriva C (3), più importante di B: B viene interrotto e esegue **C**.
- A 4 arriva D (5), la priorità massima: C viene interrotto ed esegue **D**, che termina a 6. L'arrivo di E (4) a 5 non interrompe D, che ha priorità maggiore.
- A 6 sono pronti A (1), B (2), C (3) ed E (4): esegue **E** fino a 11, poi **C** (gli mancano 3 unità) fino a 14, poi **B** (gli mancano 4) fino a 18, e infine **A** fino a 20.



Diagramma di Gantt dello scheduling a priorità con prelazione.

	A	B	C	D	E	MEDIA
Turnaround	20	18	11	2	6	11,4
Attesa	18	11	7	0	1	7,4

A, il processo meno importante, pur essendo arrivato per primo ed essendo il più breve, aspetta 18 unità: è la situazione che, se gli arrivi ad alta priorità continuassero, porterebbe alla starvation, e che l'aging serve a evitare.

DA RICORDARE

Differenza fra priorità statica e dinamica: quale rischio ha la statica?

La priorità **statica** viene fissata alla creazione del processo (ad esempio con nice) e non cambia più, qualunque cosa faccia il processo. Il rischio è la **starvation**: se continuano ad arrivare processi a priorità più alta, quelli a priorità bassa non ottengono mai la CPU, a meno di aggiungere un correttivo come l'aging. La priorità **dinamica** viene ricalcolata dal sistema in base al comportamento osservato: ad esempio un processo che usa poco del proprio quanto prima di bloccarsi per I/O riceve una priorità più alta (con la regola $1/f$), mentre uno che consuma sempre tutto il quanto viene penalizzato. Così il sistema si adatta al comportamento reale dei processi.

15.3 Code multiple

Una variante molto usata raggruppa i processi in **classi di priorità**, ognuna con la propria coda. Lo scheduling lavora su due livelli: fra le classi si sceglie sempre la **coda non vuota con priorità più alta**; all'interno della classe si usa il **Round Robin**. Finché c'è almeno un processo pronto nella classe più alta, le altre aspettano; quando quella si svuota, si passa alla successiva. Nella versione a **priorità fisse** ogni processo resta sempre nella sua classe, e le code basse rischiano la starvation; un rimedio è riservare a ciascuna coda una certa percentuale del tempo di CPU.

Nella versione **con feedback** (o **retroazione**) i processi **cambiano coda** in base al loro comportamento. Un processo che usa **tutto il quanto** senza bloccarsi si comporta da CPU bound e viene **spostato in una coda a priorità più bassa**; un processo che si blocca prima della fine del quanto si comporta da I/O bound e può salire verso le code più alte. Di solito alle code più basse si assegnano **quantità più lunghi**: i processi CPU bound vengono eseguiti meno spesso ma, quando eseguono, lo fanno più a lungo, con meno cambi di contesto. Le code alte, con quanti brevi, garantiscono la reattività ai processi interattivi. Nella coda più bassa si può usare anche FCFS.

DA RICORDARE

Come fanno le code multiple con feedback a capire se un processo è CPU bound o I/O bound?

Osservano **come il processo usa il proprio quanto** ogni volta che esegue. Se lo consuma tutto e viene interrotto dalla prelazione, il sistema deduce che è CPU bound e lo sposta in una coda a priorità più bassa, dove riceverà quanti più lunghi ma meno spesso. Se invece si blocca per I/O prima che il quanto scada, il sistema deduce che è I/O bound e lo mantiene o lo sposta in una coda a priorità più alta, con quanti brevi, per renderlo reattivo. Così la classificazione non va fatta a priori: si aggiorna in continuazione, e segue il processo anche se cambia comportamento nel tempo.

15.4 Shortest Process Next

SJF dà buoni risultati, e sarebbe utile applicarlo anche ai processi interattivi, scegliendo quello con il **prossimo CPU burst più breve**. È l'idea di **SPN** (*Shortest Process Next*). Il problema è che la durata del prossimo burst **non è nota**: dipende da eventi esterni, come l'input dell'utente. La soluzione è **stimarla** a partire dai burst precedenti dello stesso processo, con una media pesata detta **invecchiamento** (*exponential smoothing*):

$$S(n+1) = a \cdot T(n) + (1 - a) \cdot S(n)$$

dove $T(n)$ è la durata **misurata** dell'ultimo burst, $S(n)$ la stima precedente e $S(n+1)$ la nuova stima. Il parametro a , compreso fra 0 e 1, decide quanto peso dare al passato recente. Con a vicino a 1 la stima segue subito l'ultimo burst, ma si lascia ingannare da un valore isolato fuori norma; con a vicino a 0 la stima è stabile, ma si adatta troppo lentamente se il processo cambia comportamento. Il valore più comune è **$a = 1/2$** , con cui la nuova stima è semplicemente la **media** fra l'ultima misura e la stima precedente, ed è facilissima da calcolare. Espandendo la formula si vede che i burst più vecchi pesano sempre meno: con $a = 1/2$, quello di quattro passi fa pesa solo $1/16$.

	1	2	3	4	5	6	7	8
Burst misurato T	6	4	6	4	13	13	13	?
Stima S	10	8	6	6	5	9	11	12

Nell'esempio ($a = 1/2$, stima iniziale 10) ogni stima è la media della colonna precedente: $(6 + 10)/2 = 8$, $(4 + 8)/2 = 6$, e così via. Quando il processo passa a burst di 13, la stima lo insegue gradualmente (9, 11, 12) senza saltare subito al nuovo valore.

ESEMPIO SVOLTO: CALCOLARE LE STIME DI SPN

Su un processo si sono osservati, dal più vecchio al più recente, burst di 65, 70, 60, 80, 55 e 70 ms. Con $a = 1/2$, e assumendo che la stima iniziale coincida con il primo burst, calcolare le stime successive fino a quella del prossimo burst.

Con $a = 1/2$ la formula diventa $S(n+1) = (T(n) + S(n)) / 2$. La stima iniziale è 65; poi a ogni misura si fa la media fra la misura e la stima corrente:

- $(70 + 65) / 2 = 67,5$;
- $(60 + 67,5) / 2 = 63,75$;
- $(80 + 63,75) / 2 = 71,875$;
- $(55 + 71,875) / 2 = 63,4375$;
- $(70 + 63,4375) / 2 = 66,71875 \approx 66,72 \text{ ms}$, che è la stima del prossimo CPU burst.

Conviene portarsi dietro i valori esatti e arrotondare solo il risultato finale, per non accumulare errori.

DA RICORDARE

Qual è la differenza fra SPN e SJF, e perché SPN deve stimare invece di conoscere la durata?

SJF lavora nei **sistemi batch**, dove la durata dei lavori è nota o stimabile in anticipo, perché si tratta di lavori ricorrenti sottomessi con tutte le informazioni. SPN porta la stessa idea nei **sistemi interattivi**, dove la durata del prossimo CPU burst non si può conoscere, perché dipende da eventi esterni come l'input dell'utente o l'arrivo di dati dalla rete. Per questo SPN la **stima** con la media pesata $S(n+1) = a \cdot T(n) + (1 - a) \cdot S(n)$, basata sui burst precedenti dello stesso processo, e aggiorna la stima a ogni nuova misura.

15.5 Scheduling garantito, a lotteria e fair share

Lo **scheduling garantito** fa una promessa precisa sulla quota di CPU, ad esempio che con n processi ognuno riceverà $1/n$ della CPU, e la mantiene. Per farlo il sistema tiene traccia, per ogni processo, del tempo di CPU effettivamente consumato e di quello che gli spetterebbe, e calcola il rapporto fra i due: un rapporto di 0,5 significa che il processo ha avuto metà di quanto gli spettava, uno di 2 il doppio. Esegue poi il processo con il **rapporto più basso**, cioè quello più «in arretrato», finché non supera gli altri. I processi I/O bound, che consumano meno della loro quota, risultano naturalmente favoriti.

Lo **scheduling a lotteria** ottiene risultati simili in modo molto più semplice. Ogni processo riceve un certo numero di **biglietti**; a ogni decisione si estrae un biglietto **a caso**, e la CPU va al processo che lo possiede. Un processo con il 20% dei biglietti otterrà, a lungo andare, circa il 20% della CPU. Il criterio è **semplice e chiaro**, e le priorità si realizzano dando più biglietti ai processi più importanti. È anche molto **reattivo**: un nuovo processo con dei biglietti ha subito una probabilità di vincere proporzionale ai biglietti. Inoltre permette ai **processi cooperanti** di scambiarsi i biglietti: un client che si blocca in attesa di un server può cedergli i propri, così che il server esegua più spesso e gli risponda prima. Un server video che serve flussi a 10, 20 e 25 fotogrammi al secondo può dare ai rispettivi processi 10, 20 e 25 biglietti, e la CPU verrà divisa approssimativamente nelle proporzioni giuste.

Lo **scheduling fair share** sposta l'attenzione dai processi agli **utenti**. Se l'utente A ha un processo e l'utente B ne ha nove, uno scheduler che tratta tutti i processi allo stesso modo darebbe il 10% della CPU ad A e il 90% a B. Il fair share garantisce invece un uso **equo fra gli utenti**: ognuno riceve, ad esempio, il 50%, e B divide la propria metà fra i suoi nove processi.

Scheduling dei thread e dei multiprocessori

Come cambia lo scheduling quando le entità da schedulare sono thread e quando i processori sono più di uno, con gli esempi dei sistemi reali.

16.1 Lo scheduling dei thread

Con i thread lo scheduling dipende dal modello. Con i **thread utente** lo scheduler del kernel li **ignora**: sceglie un processo e gli assegna il quanto, e all'interno del processo è lo **scheduler del sistema run time** a decidere quale thread eseguire. Per il sistema run time vanno bene tutti gli algoritmi **senza prelazione** visti finora, perché non c'è un clock che possa interrompere un thread; in compenso si può usare uno scheduling **personalizzato**, scelto in base all'applicazione.

Con i **thread del kernel** è il kernel a scegliere direttamente il thread da eseguire, con due possibilità: considerare **tutti i thread uguali**, indipendentemente dal processo a cui appartengono, oppure **tenere conto dell'appartenenza al processo**. La seconda scelta ha senso perché passare a un thread di un **altro processo** richiede anche di **riprogrammare la MMU** e, in alcuni casi, di **svuotare la cache** della CPU, mentre passare a un thread dello stesso processo no. A parità di altre condizioni, conviene quindi preferire un thread dello stesso processo.

16.2 Lo scheduling sui multiprocessori

Con più CPU lo scheduler deve decidere non solo **quale** processo eseguire, ma anche **su quale CPU**. Gli approcci possibili sono due:

- **multielaborazione asimmetrica**: una sola CPU, il **master server**, esegue il codice del sistema operativo e prende tutte le decisioni di scheduling, mentre le altre eseguono solo codice utente. È semplice, perché solo il master accede alle strutture del kernel e non ci sono problemi di accesso concorrente; ma al crescere del numero di CPU il master diventa un **collo di bottiglia**;
- **multielaborazione simmetrica** (SMP): **ogni CPU** esegue il proprio scheduling e opera sulle strutture interne del kernel. È l'approccio dei sistemi moderni. I processi pronti possono stare in una **coda unificata**, comune a tutte le CPU, oppure in **code separate**, una per ogni processore o core.

La **coda unica** bilancia il carico in modo naturale, perché ogni CPU libera prende il primo processo pronto; ma la coda è una **struttura condivisa**, il cui accesso è una sezione critica da proteggere con un lock. Con molte CPU la contesa su quel lock diventa un collo di bottiglia, e una CPU libera può dover aspettare che un'altra abbia finito di consultare la coda. Le **code separate** eliminano questa contesa e scalano bene, ma introducono due nuove questioni.

La prima è la **predilezione per il processore** (*processor affinity*). Un processo che ha appena eseguito su una CPU ha riempito la **cache** di quella CPU con i propri dati; se torna in esecuzione sulla stessa CPU li ritrova, mentre se viene spostato su un'altra deve ricaricarli dalla memoria. Conviene quindi tenerlo sulla stessa CPU. La predilezione è **debole** quando il sistema cerca di mantenere il processo sulla stessa CPU senza garantirlo, ed è **forte** quando il processo viene **vincolato** a una CPU o a un gruppo di CPU specifico (su richiesta del programma o dell'amministratore). Con le code separate la predilezione debole viene da sé: un processo torna nella coda della CPU che lo stava eseguendo.

La seconda è il **bilanciamento del carico**, necessario **solo con code distinte**: senza, una CPU potrebbe restare inattiva con la coda vuota mentre un'altra ha molti processi in attesa. Le tecniche sono due:

- **migrazione guidata (push)**: un'attività del kernel controlla **periodicamente** il carico delle code e, se trova squilibri marcati, sposta processi dalle CPU più cariche a quelle più scariche. Poiché questa attività modifica le code degli altri processori, ogni coda ha il proprio lock; ma le migrazioni sono rare, quindi la contesa è minima;
- **migrazione spontanea (pull)**: una CPU che si ritrova con la **coda vuota** va a **prendersi** un processo dalla coda di un'altra CPU, scegliendo fra le più cariche.

Le due tecniche si usano **insieme**, come in Linux e FreeBSD: la migrazione guidata previene gli squilibri, quella spontanea copre i casi che si creano fra un controllo periodico e l'altro, ad esempio quando molti processi di una CPU si bloccano tutti insieme. C'è però un **conflitto con la predilezione**: ogni migrazione sposta un processo lontano dalla sua cache. Il sistema deve trovare un compromesso fra bilanciare il carico e rispettare la predilezione.

DA RICORDARE

Su un sistema multicore, come si può organizzare la coda dei processi pronti, e quale problema dà la coda unica?

Si può usare una **coda unica** condivisa da tutti i core, oppure **una coda per ogni core**. La coda unica bilancia automaticamente il carico, ma è una struttura condivisa: se due core vogliono estrarre un processo nello stesso momento si ha una corsa critica, quindi l'accesso va protetto con un lock, come il buffer del produttore e consumatore. Più core ci sono, più cresce la contesa su quel lock, che diventa un collo di bottiglia e limita la scalabilità. Con le code separate questo problema sparisce, ma bisogna bilanciare il carico con la migrazione guidata e quella spontanea.

Migrazione guidata e spontanea: perché si usano insieme senza entrare in conflitto?

Perché agiscono in **momenti diversi** e con logiche diverse. La **guidata** è periodica e preventiva: il sistema controlla tutte le code e sposta processi dai core più carichi a quelli più scarichi prima che gli squilibri diventino gravi. La **spontanea** è reattiva: scatta solo quando un core si accorge di avere la coda vuota, e va a prendere lavoro da un altro. Sono complementari: la prima previene, la seconda gestisce i casi che si creano fra un controllo periodico e il successivo.

16.3 Lo scheduling nei sistemi operativi reali

Gli elementi comuni ai sistemi attuali sono lo scheduling dei **thread** (a livello kernel), il supporto alla **SMP**, la gestione delle **priorità** e la **predilezione per i processi I/O bound**.

Windows usa uno scheduler basato su **code di priorità**, con euristiche che migliorano il servizio dei processi **interattivi**, in particolare di quello in **primo piano** (la finestra attiva riceve un quanto più lungo): la priorità di un processo viene alzata entro certi limiti se interagisce con i dispositivi, e abbassata se consuma molta CPU. Altre euristiche servono a evitare il problema dell'**inversione di priorità**. Anche **macOS** usa code di priorità con euristiche, nello scheduler ereditato da **Mach**.

Linux non distingue internamente processi e thread: schedula dei **task**, una generalizzazione di entrambi. Un task è un flusso di esecuzione a cui sono associate certe risorse (spazio degli indirizzi, file aperti, gestione dei segnali); sia fork sia la creazione di un thread sono realizzate con la stessa system call, `clone`, che specifica quali risorse il nuovo task condivide con chi lo crea. Per i task ordinari Linux ha usato a lungo il **Completely Fair Scheduler** (CFS), uno **scheduler garantito**; dalla versione 6.6 lo ha sostituito **EEVDF** (*Earliest Eligible Virtual Deadline First*), che ne conserva l'idea di fondo. Ogni task ha un **virtual runtime**, che misura quanto ha usato la CPU e cresce mentre esegue; lo scheduler favorisce i task rimasti indietro, tenuti ordinati in un **albero rosso nero**. La priorità si realizza facendo crescere il virtual runtime più in fretta per i task meno importanti, che ottengono meno CPU senza mai esserne esclusi. I task I/O bound, che usano poca CPU, restano indietro e sono favoriti in modo naturale.

Dalla memoria fisica allo spazio degli indirizzi

Come il sistema operativo gestisce la memoria centrale, dal caso senza astrazioni allo spazio degli indirizzi, allo swapping e alla gestione delle zone libere.

17.1 La memoria centrale e i processi

Nella gerarchia di memoria vista nei richiami di architettura, la **memoria centrale**, formata dalla RAM, è il livello più basso **direttamente utilizzabile dalla CPU**: un programma, per essere eseguito, deve trovarsi lì. La parte del sistema operativo che la amministra è il **gestore della memoria**: tiene traccia di quali parti sono in uso, assegna memoria ai processi e la recupera quando terminano. Il suo compito principale è offrire ai processi un'**astrazione** della memoria, e nel tempo si è passati a livelli di astrazione via via più complessi, che seguiremo in ordine.

17.2 Nessuna astrazione

Sui primi mainframe (anni '60) e sui primi personal computer (primi anni '80) non c'era alcuna astrazione: i programmi usavano direttamente gli **indirizzi fisici**. Un'istruzione come «carica il contenuto della cella 1000» accedeva proprio alla cella fisica 1000. In queste condizioni è difficile tenere due programmi in memoria contemporaneamente, perché ognuno potrebbe scrivere sopra l'altro. La memoria veniva organizzata in pochi modi: il sistema operativo nella parte bassa della RAM e il programma utente sopra; il sistema operativo in ROM nella parte alta e il programma in RAM sotto; oppure i driver dei dispositivi in ROM in alto, il sistema operativo in RAM in basso e il programma in mezzo (il modello dei primi PC con MS DOS, dove la parte in ROM era il BIOS).

Si può comunque ottenere una forma di multiprogrammazione mantenendo **più processi in memoria**, ma bisogna risolvere due problemi. Il primo è la **rilocazione**: un programma compilato per partire dall'indirizzo 0, se caricato a partire da 16 384, salterebbe agli indirizzi sbagliati. Si può risolvere al momento della compilazione, indicando al compilatore la zona di memoria in cui il programma verrà caricato (**rilocazione a compile time**), oppure con la **rilocazione statica in fase di caricamento**: il loader, mentre carica il programma, somma a ogni indirizzo l'indirizzo di partenza effettivo. Quest'ultima soluzione **rallenta il loader**, che deve riconoscere e correggere tutti gli indirizzi del programma (distinguendoli dalle costanti numeriche). Il secondo problema è la **protezione**: l'IBM 360 usava il meccanismo **lock & key**, dividendo la memoria in blocchi a ciascuno dei quali era associata una **chiave di protezione** di 4 bit; la **PSW** conteneva la chiave del processo in esecuzione, e l'hardware bloccava ogni accesso a un blocco con una chiave diversa.

17.3 Lo spazio degli indirizzi

Una soluzione migliore introduce un'astrazione vera: lo **spazio degli indirizzi**, cioè l'insieme degli indirizzi che un processo può usare, **proprio di ogni processo**. L'indirizzo 28 di un processo e l'indirizzo 28 di un altro processo corrispondono a celle fisiche diverse. Il modo più semplice per realizzarlo è la **rilocazione dinamica** con due registri hardware: il **registro base**, con l'indirizzo fisico da cui inizia la zona del processo, e il **registro limite**, con la sua lunghezza. A **ogni accesso** l'hardware confronta l'indirizzo generato dal programma con il limite (se lo supera, genera un'eccezione) e poi gli **somma la base**. Il programma usa solo indirizzi da 0 in su e non sa dove si trova realmente; se viene spostato, basta cambiare il registro base.

DEFINIZIONE

Lo **spazio degli indirizzi** è l'insieme degli indirizzi che un processo può usare per accedere alla memoria. Ogni processo ha il proprio spazio degli indirizzi, indipendente da quello degli altri: lo stesso indirizzo in due processi diversi corrisponde, in generale, a due celle fisiche diverse.

Nei primi sistemi era la CPU stessa a fare questo controllo; nei sistemi più evoluti se ne occupa la **MMU** (*Memory Management Unit*). Poiché ogni processo ha la propria zona, i due registri vanno **aggiornati a ogni cambio di contesto**, e solo il sistema operativo può modificarli. Il **CDC 6600** aveva entrambi i registri; l'**Intel 8088**, il processore del primo PC IBM, aveva solo registri base, senza registro limite. Lo svantaggio della tecnica è il costo di una somma e di un confronto a ogni accesso alla memoria.

DA RICORDARE

Che cos'è l'offset nella rilocalizzazione dinamica e nella paginazione? Cambia concetto fra i due?

Nella rilocalizzazione dinamica l'offset è l'indirizzo generato dal processo, cioè la distanza dall'inizio della sua zona di memoria: l'indirizzo fisico è **registro base + offset**. Nella paginazione, che vedremo fra poco, l'offset è la posizione **all'interno di una pagina**, da 0 alla dimensione della pagina meno 1: l'indirizzo fisico è **numero di frame × dimensione della pagina + offset**. Il concetto è lo stesso, la distanza dall'inizio di un blocco di riferimento; cambia il blocco, che nella rilocalizzazione dinamica è l'intero spazio del processo e nella paginazione è la singola pagina.

17.4 Lo swapping e la frammentazione

Anche con base e limite, il **livello di multiprogrammazione** resta limitato dalla dimensione della memoria centrale: se i processi non entrano tutti in RAM, alcuni non possono essere eseguiti. La prima soluzione è lo **swapping**: un processo viene caricato **per intero** in memoria, eseguito per un po', e poi **ricopiato per intero sul disco** per fare posto a un altro; più tardi verrà riportato in memoria, non necessariamente nella stessa posizione (la rilocalizzazione dinamica lo permette). La scelta di quali processi spostare è compito dello **swapper**, detto anche **scheduler di medio termine** per distinguerlo da quello visto nella parte sullo scheduling. Un problema delicato sono gli **I/O pendenti**: se un processo viene spostato su disco mentre un trasferimento DMA sta scrivendo nel suo buffer, i dati finirebbero nella memoria del processo che ha preso il suo posto. Si evita non spostando i processi con I/O in corso, oppure facendo l'I/O solo in buffer del kernel.

La memoria si può assegnare in **partizioni di dimensione fissa**, stabilite in anticipo, oppure di **dimensione dinamica**, grandi esattamente quanto serve. Se un processo può crescere (heap e stack), conviene assegnargli un po' di spazio in più. In entrambi i casi nasce la **frammentazione**, cioè memoria libera che non si riesce a usare:

- la **frammentazione interna** è lo spazio sprecato **dentro** una zona assegnata, perché la zona è più grande di quanto serva al processo: un processo di 700 KB in una partizione fissa da 1 MB ne spreca 300;
- la **frammentazione esterna** è lo spazio libero **fra** le zone assegnate, diviso in tanti buchi troppo piccoli: con tre buchi da 100 MB, un processo da 250 MB non entra, anche se la memoria libera in totale sarebbe 300 MB. Con le partizioni dinamiche, a forza di processi che entrano ed escono, la memoria si riempie di buchi.

La frammentazione esterna si elimina con la **compattazione della memoria** (*memory compaction*), spostando tutti i processi verso un'estremità per riunire i buchi in un unico spazio libero. È però un'operazione lentissima: copiando un byte ogni nanosecondo, compattare 16 GB richiede circa 16 secondi, durante i quali il sistema è quasi fermo. Per questo di solito non si fa.

17.5 Gestire l'allocazione: bitmap e liste

Il sistema deve sapere in ogni momento quali parti della memoria sono libere. I metodi sono due. Con la **bitmap** la memoria è divisa in **unità di allocazione** di dimensione fissa, e a ognuna corrisponde un bit: 0 se è libera, 1 se è occupata. La scelta della **dimensione del blocco** è un compromesso: blocchi piccoli producono una bitmap grande (con unità di 4 byte, cioè 32 bit, la bitmap occupa 1/33 della memoria), blocchi grandi producono più frammentazione interna, perché la dimensione dei processi difficilmente è un multiplo esatto del blocco. Il difetto principale è la lentezza: per caricare un processo che richiede k blocchi bisogna cercare nella bitmap una sequenza di k zeri consecutivi.

In alternativa si usa una **lista dei blocchi liberi e occupati**: ogni elemento descrive un segmento contiguo di memoria, che può essere un processo (P) o un buco (H, *hole*), con l'indirizzo di inizio e la lunghezza. La lista è **ordinata per indirizzo**, così quando un processo termina è facile fondere il suo spazio con gli eventuali buchi adiacenti in un unico buco più grande: è la **coalescenza**. Conviene che la lista sia **doppiamente collegata**, per trovare subito sia il segmento precedente sia il successivo.

Quando arriva una richiesta, c'è da scegliere il buco in cui collocarla. Gli **algoritmi di ricerca** classici sono quattro:

- **first fit**: si scorre la lista dall'inizio e si usa il **primo buco abbastanza grande**, che viene diviso in una parte per il processo e una che resta libera. È veloce, perché la ricerca si ferma al primo successo;
- **next fit**: come first fit, ma ogni ricerca **riparte dal punto** in cui si era fermata la precedente invece che dall'inizio. Le simulazioni mostrano prestazioni leggermente peggiori di first fit;
- **best fit**: si sceglie il **buco più piccolo** fra quelli abbastanza grandi. È più lento, perché deve scorrere tutta la lista, e paradossalmente spreca più memoria, perché lascia tanti buchi minuscoli che non si riusciranno più a usare;
- **worst fit**: si sceglie il **buco più grande**, così che il resto sia ancora abbastanza grande da servire. Neanche questo, nelle simulazioni, si comporta bene.

Si possono anche tenere **liste separate** per processi e buchi, con la **lista dei blocchi liberi ordinata per dimensione**: best fit diventa veloce quanto first fit, perché il primo buco sufficiente è anche il più piccolo, ma la coalescenza diventa costosa, perché i buchi vicini in memoria non sono vicini nella lista. Una variante è **quick fit**, con liste separate per le dimensioni richieste più spesso (una per i buchi da 4 KB, una per quelli da 8 KB e così via): trovare un buco è velocissimo, ma la coalescenza resta costosa.

ESEMPIO SVOLTO: FIRST FIT, NEXT FIT, BEST FIT E WORST FIT

Segmenti di dimensione fissa disponibili, nell'ordine: 100, 95, 80, 65, 40, 60, 55, 50, 85, 45 MiB. Richieste, nell'ordine: 33, 16, 68, 59, 46 MiB. Lo spazio residuo di un segmento non è riutilizzabile ed è uno spreco (frammentazione interna). Indicare il segmento scelto per ogni richiesta e lo spreco.

RICHIESTA	FIRST FIT	NEXT FIT	BEST FIT	WORST FIT
33 MiB	100 (spreco 67)	100 (spreco 67)	40 (spreco 7)	100 (spreco 67)
16 MiB	95 (spreco 79)	95 (spreco 79)	45 (spreco 29)	95 (spreco 79)
68 MiB	80 (spreco 12)	80 (spreco 12)	80 (spreco 12)	85 (spreco 17)
59 MiB	65 (spreco 6)	65 (spreco 6)	60 (spreco 1)	80 (spreco 21)
46 MiB	60 (spreco 14)	60 (spreco 14)	50 (spreco 4)	65 (spreco 19)
Spreco totale	178 MiB	178 MiB	53 MiB	203 MiB

Con **first fit** ogni richiesta riparte dall'inizio della lista: la 46 MiB salta il segmento da 40, troppo piccolo, e finisce nel 60. Con **next fit** la ricerca riparte da dove si era fermata la precedente: dopo il 65 si prova il 40 (troppo piccolo) e poi il 60. Qui first fit e next fit coincidono, mentre best fit spreca molto meno.

Memoria virtuale e paginazione

La memoria virtuale e la paginazione: traduzione degli indirizzi, tabelle delle pagine, TLB e tempo effettivo di accesso.

18.1 L'idea della memoria virtuale

Tutte le tecniche viste finora richiedono che ogni processo occupi una zona **contigua** di memoria, e che vi stia **per intero**. La **memoria virtuale** abbandona entrambi i vincoli e passa a un'**allocazione non contigua** della memoria fisica. Lo **spazio di indirizzamento virtuale** del processo viene diviso in blocchi di uguale dimensione, le **pagine**; la memoria fisica viene divisa in blocchi della stessa dimensione, i **frame** (o pagine fisiche). Ogni pagina può essere collocata in un frame qualsiasi, e pagine consecutive per il processo possono stare in frame lontanissimi fra loro. Inoltre **alcune pagine possono non risiedere affatto in memoria**: restano sul disco e vengono **caricate all'occorrenza** dalla memoria secondaria, quando il processo ne ha bisogno.

DEFINIZIONE

La **memoria virtuale** è la tecnica che permette di eseguire un processo anche se solo una parte del suo spazio degli indirizzi si trova in memoria centrale. Lo spazio virtuale è diviso in **pagine**, la memoria fisica in **frame** della stessa dimensione; la MMU traduce ogni indirizzo virtuale in un indirizzo fisico e, se la pagina non è in memoria, genera un **page fault**, con cui il sistema operativo la carica dal disco.

I vantaggi sono notevoli. Sparisce la **frammentazione esterna**, perché qualunque frame libero è utilizzabile. Un processo può essere **più grande della RAM**. Si adatta perfettamente a un sistema **multiprogrammato**, perché tenendo in memoria solo le pagine che servono ci stanno molti più processi; mentre un processo aspetta che una sua pagina venga caricata, la CPU può eseguirne un altro. Infine la **protezione fra processi è implicita**: ogni processo genera solo indirizzi del proprio spazio virtuale, e non ha modo di nominare i frame di un altro.

Resta invece la **frammentazione interna**, legata alla **dimensione delle pagine**: l'ultima pagina di un processo in media è piena solo a metà. Di solito la dimensione è fissa (tipicamente 4 KB), ma alcuni sistemi, come **Solaris**, supportano pagine di dimensione variabile. Da non confondere con lo swapping: lo swapping sposta sul disco un processo intero, la paginazione sposta singole pagine.

18.2 La paginazione e la traduzione degli indirizzi

La paginazione è gestita dalla **MMU**, che traduce ogni indirizzo virtuale generato dalla CPU nel corrispondente indirizzo fisico, consultando la **tabella delle pagine** del processo, con una voce per ogni pagina virtuale. Ogni voce contiene un **bit di presenza**: se la pagina è in memoria, la voce indica il frame che la contiene; se non lo è, la MMU genera un'eccezione, l'**errore di pagina** (*page fault*). Il page fault non è un errore del programma: è una **trap** con cui la MMU chiede aiuto al sistema operativo. Il sistema sceglie un frame (liberandone uno, se necessario), vi carica la pagina dal disco, aggiorna la tabella e **fa ripartire l'istruzione** che aveva causato il fault, che stavolta va a buon fine. Il processo, nel frattempo, è bloccato.

Poiché la dimensione delle pagine è una **potenza di 2**, la traduzione si fa semplicemente separando i bit dell'indirizzo. Con indirizzi virtuali di m bit (uno spazio virtuale di 2^m byte) e pagine di 2^n byte:

- gli $m - n$ bit più significativi dell'indirizzo virtuale sono il **numero di pagina**, quindi le pagine sono 2^{m-n} ;
- gli n bit meno significativi sono l'**offset**, cioè la posizione all'interno della pagina, che **non viene tradotto**.

La MMU usa il numero di pagina come indice nella tabella, ne ricava il numero di frame e lo affianca all'offset. Se gli indirizzi fisici sono di t bit (una memoria fisica di 2^t byte), i frame sono 2^{t-n} e il numero di frame occupa $t - n$ bit. Di solito lo spazio virtuale è più grande di quello fisico, cioè $m > t$. In decimale l'operazione equivale a una divisione: numero di pagina = indirizzo / dimensione pagina, offset = resto. Ad esempio, con pagine da 4 KB (4096 byte) l'indirizzo virtuale 8196 ha numero di pagina $8196 / 4096 = 2$ e offset 4; se la tabella mappa la pagina 2 sul frame 6, l'indirizzo fisico è $6 \times 4096 + 4 = 24\ 580$.

ESEMPI SVOLTI: DIMENSIONI, PAGINE E FRAME

Questi esercizi si risolvono sempre con le stesse relazioni: dimensione della pagina = 2^n , numero di pagine = spazio virtuale / dimensione pagina, numero di frame = memoria fisica / dimensione pagina. Conviene scrivere tutto in potenze di 2.

- **Spazio virtuale di 1 GB, numero di pagina virtuale di 22 bit, indirizzo fisico di 20 bit: quanti frame ci sono?** Lo spazio virtuale è 2^{30} byte e le pagine sono 2^{22} , quindi ogni pagina è grande $2^{30} / 2^{22} = 2^8 = 256$ byte. La memoria fisica è 2^{20} byte, quindi i frame sono $2^{20} / 2^8 = 2^{12} = 4096$.
- **Memoria virtuale di 4 MB, tabella delle pagine con 2^{13} voci, numeri di frame di 8 bit: quanto è grande la RAM?** Le pagine sono $2^{22} / 2^{13} = 2^9 = 512$ byte; con 8 bit si numerano 2^8 frame, quindi la RAM è $2^8 \times 2^9 = 2^{17}$ byte = **128 KB**.
- **Indirizzi virtuali di 48 bit e fisici di 32 bit, pagine da 4 KB: quante voci avrebbe una tabella a un solo livello?** L'offset richiede 12 bit ($2^{12} = 4096$), restano $48 - 12 = 36$ bit per il numero di pagina: **2^{36} voci**, cioè circa 64 miliardi. Una tabella del genere non si può tenere in memoria, ed è il motivo delle soluzioni del paragrafo 18.8.
- **Numero di pagina e offset degli indirizzi 20000, 32768 e 60000, con pagine da 4 KB e da 8 KB.** Si divide l'indirizzo per la dimensione della pagina: il quoziente intero è il numero di pagina, il resto è l'offset.

INDIRIZZO	PAGINE DA 4 KB (4096 BYTE)	PAGINE DA 8 KB (8192 BYTE)
20000	pagina 4, offset 3616	pagina 2, offset 3616
32768	pagina 8, offset 0	pagina 4, offset 0
60000	pagina 14, offset 2656	pagina 7, offset 2656

Ad esempio $60000 = 14 \times 4096 + 2656$. L'offset è lo stesso nei due casi perché, raddoppiando la pagina, l'offset guadagna un bit e il numero di pagina ne perde uno.

18.3 La voce della tabella delle pagine

Ogni voce della tabella delle pagine contiene diverse informazioni, usate in parte dall'hardware e in parte dal sistema operativo:

- il **numero del frame**, l'informazione principale, con cui la MMU costruisce l'indirizzo fisico;
- il **bit presente/assente**: se vale 1 la pagina è in memoria e la voce è valida per la traduzione, se vale 0 un accesso provoca un page fault;
- i bit di **protezione**, che indicano i tipi di accesso consentiti: **lettura** e **scrittura**, ed eventualmente **esecuzione**. Le pagine di codice sono tipicamente in sola lettura ed esecuzione; quelle di dati in lettura e scrittura, ma non eseguibili, il che ostacola gli attacchi che iniettano codice nello stack o nello heap. Una violazione genera un'eccezione;
- il **bit modificato** (*dirty bit*), che l'hardware imposta a 1 quando la pagina viene scritta. Quando bisogna liberare il frame, una pagina **pulita** (bit a 0) si può semplicemente scartare, perché la copia sul disco è ancora valida; una pagina **sporca** va prima riscritta sul disco;

- il **bit referenziato**, che l'hardware imposta a 1 a **ogni riferimento** alla pagina, in lettura o in scrittura, e che il sistema operativo azzerava periodicamente. Serve agli algoritmi di sostituzione per capire quali pagine sono usate;
- un bit per **disabilitare la cache** su quella pagina, utile per le pagine che corrispondono ai registri di un dispositivo (I/O mappato in memoria): se il valore venisse letto dalla cache, il sistema non vedrebbe mai i cambiamenti fatti dal dispositivo;
- nella pratica anche un **bit di validità** (o di **allocazione**), che indica se la pagina fa parte dello spazio effettivamente usato dal processo. È diverso dal bit di presenza: una pagina assente esiste ma è sul disco, e il suo accesso causa un page fault normale; una pagina **non valida** non è mai stata allocata (ad esempio lo spazio fra heap e stack), e il suo accesso è un errore fatale, come il *segmentation fault*.

Da notare che la voce **non contiene l'indirizzo sul disco** della pagina assente: quell'informazione è gestita dal sistema operativo in strutture proprie, perché l'hardware non ne ha bisogno.

DA RICORDARE

Quando si verifica un page fault?

Quando una pagina referenziata da un processo non si trova in alcun frame della memoria fisica, cioè il bit di presenza della sua voce vale 0. Non è un page fault l'accesso a un indirizzo fuori dallo spazio del processo o con offset fuori dai limiti (sono errori di protezione), né un cache miss (riguarda la cache, non la paginazione), né la necessità di riscrivere su disco una pagina sporca prima di sostituirla (è una conseguenza della gestione del fault, non la sua causa).

18.4 La tabella dei frame

Oltre alle tabelle delle pagine, una per processo, il sistema operativo mantiene un'unica **tabella dei frame**, con una voce per ogni frame fisico che ne indica lo **stato**, **occupato o libero**, e, se occupato, **da quale processo**. Viene consultata ogni volta che si crea un nuovo processo, per costruirne la tabella delle pagine, e ogni volta che un processo chiede di allocare nuove pagine o che un page fault richiede un frame libero.

18.5 Dove tenere la tabella delle pagine

Nella progettazione di una tabella delle pagine vanno curati due aspetti: la **velocità di consultazione**, perché la traduzione avviene a ogni accesso alla memoria (anche una semplice addizione con un operando in memoria richiede almeno due traduzioni, una per prelevare l'istruzione e una per l'operando), e la **dimensione**. Per la velocità ci sono due idee estreme:

- tenere l'intera tabella in un insieme di **registri** veloci dentro la MMU. La traduzione non richiede accessi alla memoria, ma a ogni cambio di contesto bisogna ricaricare tutti i registri, e il metodo funziona solo con tabelle piccole;
- tenere la tabella **interamente in memoria** e usare un solo registro, il **PTBR** (*Page Table Base Register*), che contiene l'indirizzo fisico di inizio della tabella del processo corrente. Il **cambio di contesto è velocissimo**, perché basta cambiare il PTBR, ma per prelevare un dato servono **due accessi alla memoria**: uno alla tabella per la traduzione e uno al dato vero e proprio. La memoria apparirebbe lenta il doppio.

ESEMPIO SVOLTO: IL COSTO DI CARICARE LA TABELLA NEI REGISTRI

Una macchina ha indirizzi virtuali di 32 bit e pagine da 8 KB. La tabella delle pagine è tenuta in registri hardware, con una parola di 32 bit per voce, e quando un processo va in esecuzione viene copiata dalla memoria nei registri al ritmo di una parola ogni 100 ns. Se ogni processo esegue per 100 ms (compreso il caricamento della tabella), che frazione del tempo di CPU è dedicata a caricare tabelle?

Le pagine da 8 KB = 2^{13} byte richiedono 13 bit di offset, quindi il numero di pagina ha $32 - 13 = 19$ bit e la tabella ha $2^{19} = 524\,288$ voci. Caricarle tutte richiede $524\,288 \times 100\text{ ns} \approx 52,4\text{ ms}$, su 100 ms di esecuzione: circa il **52%** del tempo di CPU va perso a caricare tabelle. Ecco perché le tabelle grandi si tengono in memoria.

18.6 La TLB

La soluzione al doppio accesso parte da un'osservazione: in genere un programma fa **un gran numero di riferimenti a un piccolo numero di pagine** (è il principio di località). Conviene allora tenere le traduzioni più usate in una piccola cache dentro la MMU: la **TLB** (*Translation Lookaside Buffer*), detta anche **memoria associativa** o registri associativi. È formata da un numero ridotto di registri, **fra 64 e 1024**, ognuno dei quali contiene una voce con:

- il **numero di pagina virtuale**, che fa da chiave di ricerca (nella tabella delle pagine non serve, perché lì il numero di pagina è l'indice della voce);
- un **bit di validità** della voce della TLB, che dice se il registro è in uso;
- il **codice di protezione**, perché la MMU deve continuare a far rispettare i permessi;
- il **dirty bit**, aggiornato localmente nella TLB e ricopiato nella tabella delle pagine quando la voce viene rimossa;
- il **numero di frame**.

DEFINIZIONE

La **TLB** (*Translation Lookaside Buffer*) è una piccola memoria associativa interna alla MMU che conserva le traduzioni da pagina virtuale a frame usate più di recente. Se la traduzione cercata è nella TLB (**hit**) non serve leggere la tabella delle pagine in memoria; altrimenti (**miss**) la si legge e la si inserisce nella TLB.

Quando la CPU genera un indirizzo, la MMU confronta il numero di pagina **contemporaneamente con tutte le voci** della TLB: la **ricerca è parallelizzata in hardware** e richiede un tempo costante. Se la pagina c'è, si ha un **TLB hit** e il frame si ottiene senza consultare la tabella in memoria. Altrimenti si ha un **TLB miss**: la MMU consulta la tabella delle pagine in memoria e copia la voce nella TLB, al posto di una già presente, così che i successivi accessi alla stessa pagina diano hit. Alcune voci possono essere **vincolate** (*wired*), cioè mai rimosse, ad esempio quelle del codice del kernel. Su alcune architetture RISC la TLB è gestita **via software**: in caso di miss interviene il sistema operativo, che carica la voce mancante.

C'è un problema con i cambi di contesto: le voci della TLB sono etichettate solo con il numero di pagina, ma la pagina 5 di un processo e la pagina 5 di un altro sono pagine diverse. La soluzione più semplice è **svuotare la TLB** (*flush*) a ogni cambio di contesto, ma così ogni processo riparte con una serie di miss. In alternativa si aggiunge a ogni voce un **ASID** (*address space identifier*), un identificatore dello spazio di indirizzamento a cui appartiene la traduzione: la TLB può contenere insieme voci di processi diversi senza ambiguità, e il flush non serve più. Si parla di spazio di indirizzamento e non di processo perché i thread dello stesso processo condividono lo stesso ASID.

DA RICORDARE

Che cos'è l'ASID in una TLB, e che cosa permette di evitare?

È un identificatore aggiunto a ogni voce della TLB che indica a quale **spazio di indirizzamento**, cioè a quale processo, appartiene quella traduzione. Senza ASID la TLB può contenere in modo affidabile le traduzioni di un solo processo, perché la stessa pagina virtuale corrisponde a frame diversi in processi diversi: a ogni cambio di contesto bisogna svuotarla, e il nuovo processo parte con una raffica di miss. Con l'ASID la MMU considera valida una voce solo se coincidono sia il numero di pagina sia l'ASID, quindi la TLB può contenere le traduzioni di più processi insieme e **non va svuotata** al cambio di contesto, che diventa più rapido.

18.7 Il tempo effettivo di accesso

L'effetto della TLB si misura con il **tempo effettivo di accesso** (EAT, *Effective Access Time*). Esempio: accesso alla memoria 100 ns, accesso alla TLB 20 ns. In caso di **hit** servono 20 ns per la TLB e 100 ns per il dato, cioè **120 ns**; in caso di **miss** servono 20 ns per la TLB, 100 ns per la tabella delle pagine e 100 ns per il dato, cioè **220 ns**. Con una percentuale di hit (*TLB ratio*) dell'80%, il tempo medio è $0,8 \times 120 + 0,2 \times 220 = \mathbf{140 \text{ ns}}$. In generale, con tempo di accesso alla memoria α , tempo di accesso alla TLB β e TLB ratio ϵ :

$$\text{EAT} = \epsilon \cdot (\alpha + \beta) + (1 - \epsilon) \cdot (2\alpha + \beta)$$

ESEMPI SVOLTI: CALCOLARE L'EAT

- **Memoria 160 ns, TLB 25 ns, TLB ratio 75%**, contando anche il prelievo del dato: $\text{EAT} = 0,75 \times (160 + 25) + 0,25 \times (2 \times 160 + 25) = 0,75 \times 185 + 0,25 \times 345 = 138,75 + 86,25 = \mathbf{225 \text{ ns}}$.
- **Memoria 150 ns, TLB 20 ns, TLB ratio 70%**: $\text{EAT} = 0,7 \times 170 + 0,3 \times 320 = 119 + 96 = \mathbf{215 \text{ ns}}$.
- **Con i page fault**. TLB 150 ns, memoria 400 ns, gestione di un page fault 8 ms, TLB hit 70% e page fault rate 2% (sul totale degli accessi). Un page fault può avvenire solo in caso di TLB miss, perché una pagina presente nella TLB è sicuramente in memoria. I casi sono tre: **hit** (70%): $150 + 400 = 550 \text{ ns}$; **miss senza fault** (30% - 2% = 28%): TLB, tabella e dato, $150 + 400 + 400 = 950 \text{ ns}$; **miss con fault** (2%): TLB, tabella, gestione del fault e dato, $150 + 400 + 8\,000\,000 + 400 = 8\,000\,950 \text{ ns}$. $\text{EAT} = 0,7 \times 550 + 0,28 \times 950 + 0,02 \times 8\,000\,950 = 385 + 266 + 160\,019 \approx \mathbf{160\,670 \text{ ns} \approx 0,16 \text{ ms}}$. Anche solo il 2% di page fault rende la memoria circa trecento volte più lenta: il tempo del disco domina tutto.

Il terzo esercizio mostra perché i sistemi fanno di tutto per ridurre i page fault: il loro costo è di vari ordini di grandezza superiore a quello di un accesso alla memoria.

ESEMPIO SVOLTO: QUALE TLB RATIO SERVE PER UN CERTO DEGRADO?

Una memoria paginata ha la tabella delle pagine in RAM e un tempo di accesso alla memoria $\alpha = 30 \text{ ns}$, quindi senza TLB ogni accesso costa $2\alpha = 60 \text{ ns}$. Si aggiunge una TLB con tempo di accesso $\beta = 1 \text{ ns}$. Quale TLB ratio ϵ serve perché il tempo effettivo peggiori al massimo di una certa percentuale rispetto ad α ?

Con la formula generale: $\text{EAT} = \epsilon \cdot (\alpha + \beta) + (1 - \epsilon) \cdot (2\alpha + \beta) = 31\epsilon + 61(1 - \epsilon) = 61 - 30\epsilon$. Il valore più basso si ha con $\epsilon = 1$ (tutti hit): $\text{EAT} = 31 \text{ ns}$, cioè **3,33% in più** di α . Quindi un degrado del 3% ($\text{EAT} = 30,9 \text{ ns}$) è **impossibile**: anche con il 100% di hit si paga comunque il nanosecondo della TLB. Con un degrado ammesso del 5% ($\text{EAT} = 31,5 \text{ ns}$) invece: $61 - 30\epsilon = 31,5$, da cui $\epsilon = 29,5 / 30 \approx \mathbf{98,3\%}$.

Con una tabella a **due livelli** un miss costa tre accessi alla memoria: $\text{EAT} = \epsilon \cdot (\alpha + \beta) + (1 - \epsilon) \cdot (3\alpha + \beta) = 31\epsilon + 91(1 - \epsilon) = 91 - 60\epsilon$. Il minimo è ancora 31 ns, quindi il 3% resta impossibile; per il 5% serve $91 - 60\epsilon = 31,5$, cioè $\epsilon = 59,5 / 60 \approx \mathbf{99,2\%}$. Con più livelli il miss costa di più, e serve una TLB ancora più efficace.

DA RICORDARE

Un sistema ha una TLB da 32 voci e pagine da 4 KB. Un programma ha il codice in una sola pagina e legge in sequenza gli elementi interi di un array che occupa migliaia di pagine. La TLB è efficace?

Sì. La pagina del codice resta sempre nella TLB. Per i dati, l'array è letto in sequenza, quindi si accede a una pagina per volta: il primo elemento di ogni pagina causa un miss, e tutti gli altri della stessa pagina danno hit. Se un intero lungo occupa 8 byte, una pagina ne contiene $4096 / 8 = 512$, quindi si ha **un miss ogni 512 letture**, un hit ratio di circa il 99,8%. Il fatto che l'array occupi molte più pagine delle voci della TLB non conta, perché le pagine vengono usate una dopo l'altra e non insieme.

18.8 Tabelle delle pagine multilivello

Resta il problema della **dimensione**. Con indirizzi virtuali di 32 bit (come sul Pentium) e pagine da 4 KB, le pagine sono 2^{20} , circa **un milione**; con voci da 32 bit la tabella di ogni processo occupa circa **4 MB**, che devono per giunta essere **contigui** in memoria fisica, perché il PTBR indica solo l'inizio. Con indirizzi a 64 bit le pagine da 4 KB sarebbero 2^{52} : una tabella impossibile da memorizzare.

La soluzione è **non mantenere l'intera tabella in memoria**, con le **tabelle multilivello** (o **paginazione gerarchica**). Con due livelli e indirizzi di 32 bit, il numero di pagina di 20 bit si divide in due campi di 10 bit: il primo (PT1) indicizza la **tabella di primo livello**, di 1024 voci, ognuna delle quali punta a una **tabella di secondo livello**; il secondo (PT2) indicizza la tabella di secondo livello, di altre 1024 voci, che contiene il numero di frame; gli ultimi 12 bit restano l'offset. Il vantaggio è che le tabelle di secondo livello corrispondenti a zone **non allocate** dello spazio virtuale (ad esempio l'enorme spazio vuoto fra heap e stack) **non vengono create**: la voce di primo livello è marcata come non valida. Un processo che usa poca memoria ha quindi una tabella di pochi kilobyte, e ogni tabella occupa al più una pagina, eliminando il problema della contiguità.

Il prezzo è il **numero di accessi**: in caso di TLB miss, con due livelli **possono essere necessari tre accessi alla memoria**, uno alla tabella di primo livello, uno a quella di secondo livello e uno al dato. Si possono usare anche **più livelli**, ma senza esagerare, perché ogni livello aggiunge un accesso a ogni miss. I processori x86 a 64 bit usano in pratica solo 48 bit di indirizzo e **quattro livelli** di 512 voci ciascuno: $(2^9)^4 \times 2^{12} = 2^{48}$ byte, cioè 256 TB di spazio virtuale.

18.9 La tabella delle pagine invertita

Un approccio opposto è la **tabella delle pagine invertita**: invece di una voce per ogni pagina virtuale di ogni processo, c'è **una voce per ogni frame fisico**, che riporta la coppia (**identificatore del processo, pagina virtuale**) ospitata in quel frame. C'è **una sola tabella per tutto il sistema**, e la sua dimensione dipende dalla memoria fisica e non dallo spazio virtuale: è **molto piccola** a paragone delle tabelle delle pagine, anche con indirizzi a 64 bit. In pratica è la tabella dei frame arricchita con il numero di pagina, e usata per tradurre.

Il problema è che la traduzione diventa una **ricerca**: la tabella è indicizzata per frame, mentre la MMU conosce il processo e la pagina, quindi dovrebbe scorgerla tutta, una ricerca lentissima. Si risolve con una **tabella hash indicizzata sull'indirizzo virtuale**, in cui le voci con lo stesso hash sono collegate in una lista, e la si accoppia con una **TLB**, così che la tabella si consulti solo in caso di miss. Servono comunque ancora le **tabelle per processo**, con le informazioni necessarie per **gestire i page fault** (ad esempio dove si trovano sul disco le pagine assenti e i loro permessi); queste però si consultano solo durante un fault, e possono essere **paginate a loro volta**.

DA RICORDARE

Che cos'è la tabella delle pagine invertita, perché si usa e come si cerca un indirizzo?

È un'unica tabella per tutto il sistema, con **una voce per ogni frame fisico** che indica quale processo e quale pagina virtuale lo occupano. Si usa perché la sua dimensione è proporzionale alla RAM, e non allo spazio virtuale di tutti i processi, che con indirizzi a 64 bit sarebbe enorme. Poiché è indicizzata per frame, non si può accedere direttamente alla voce di una pagina: bisognerebbe cercare in tutta la tabella la riga con quel processo e quella pagina. Per questo si usa una **tabella hash** sull'indirizzo virtuale, che porta direttamente alla voce giusta, e una **TLB** davanti, per evitare la ricerca nella maggior parte dei casi.

Che differenza c'è fra la tabella dei frame e la tabella delle pagine invertita?

Entrambe hanno una voce per frame. La **tabella dei frame** dice solo se il frame è libero o occupato e da quale processo: serve al sistema per amministrare la memoria fisica. La **tabella invertita** aggiunge anche il numero di pagina virtuale contenuta nel frame, e così diventa un vero strumento di **traduzione** degli indirizzi.

18.10 Cache della memoria e memoria virtuale

La cache della CPU e la MMU sono entrambe fra la CPU e la RAM, e bisogna decidere in che ordine. Una cache **basata sugli indirizzi fisici** sta dopo la MMU: **non va invalidata** al cambio di contesto, perché gli indirizzi fisici sono univoci, ma il caching è **poco efficace**, perché ogni accesso deve prima attraversare la MMU. Una cache **basata sugli indirizzi virtuali** sta prima della MMU ed è **più efficace**, perché risponde subito alla CPU; ma lo stesso indirizzo virtuale ha significati diversi in processi diversi, quindi **servono gli ASID** per non doverla invalidare a ogni cambio di contesto, e la soluzione **scala male** su cache grandi come la L2. In pratica si usano entrambe: la **cache L1**, piccola e vicinissima alla CPU, è basata su **indirizzi virtuali**; la **L2** e le successive sono basate su **indirizzi fisici**.

DA RICORDARE

Che differenza c'è fra la cache della memoria e la memoria virtuale? A che livello agiscono?

Risolvono problemi diversi a livelli diversi della gerarchia. La **cache** (L1, L2, L3) è una memoria hardware piccola e velocissima vicino alla CPU, che tiene copie dei dati **già presenti in RAM** usati di recente, per ridurre gli accessi alla RAM; è gestita interamente dall'hardware. La **memoria virtuale** è un meccanismo hardware e software (MMU, tabella delle pagine, sistema operativo) che dà a ogni processo uno spazio di indirizzi proprio, anche più grande della RAM, e decide **quali pagine stanno in RAM e quali sul disco**, spostandole quando serve. In breve: la cache velocizza l'accesso a ciò che è in RAM, la memoria virtuale decide che cosa ci sta. Le due interagiscono nella scelta fra cache basate su indirizzi virtuali o fisici.

Gli algoritmi di sostituzione delle pagine

Quale pagina togliere dalla memoria quando serve un frame libero: l'algoritmo ottimo e le sue approssimazioni realizzabili.

19.1 Il problema e l'algoritmo ottimo

Quando avviene un page fault e **non ci sono frame liberi**, bisogna scegliere una **pagina vittima** da togliere dalla memoria (se è sporca va prima riscritta sul disco). Il problema è simile a quello della gestione delle cache, con la differenza che qui la scelta è fatta dal software e che un errore costa un accesso al disco. L'**obiettivo** è **minimizzare il numero di page fault futuri**.

La soluzione **ottimale** è l'algoritmo **OPT**: si sceglie la pagina che **verrà referenziata nel futuro più lontano**, rimandando il più possibile il prossimo fault. È dimostrabile che è il migliore possibile, ma è **difficilmente realizzabile**, anzi impossibile, perché il sistema non conosce i riferimenti futuri. Rappresenta però un **termine di paragone**: simulando un programma due volte, si può misurare quanto un algoritmo reale si avvicina all'ottimo.

19.2 NRU

Gli algoritmi reali raccolgono statistiche sull'uso delle pagine attraverso i bit **R** (referenziato) e **M** (modificato) della tabella delle pagine, **aggiornati tipicamente in hardware** a ogni accesso e **azzerati dal sistema operativo**. In particolare il bit R viene **azzerato periodicamente**, ad esempio a ogni interrupt del clock, così da distinguere le pagine usate di recente da quelle che non lo sono. L'algoritmo **NRU** (*Not Recently Used*) divide le pagine in quattro classi:

CLASSE	R	M	SIGNIFICATO
0	0	0	Non referenziata di recente, non modificata: la candidata ideale
1	0	1	Non referenziata di recente, ma modificata: va riscritta sul disco prima di toglierla
2	1	0	Referenziata di recente, non modificata
3	1	1	Referenziata di recente e modificata

Si sceglie a caso una pagina dalla **classe non vuota di numero più basso**. La classe 1 può sembrare impossibile, ma nasce dall'azzeramento periodico: una pagina modificata tempo fa ha $M = 1$, mentre il suo R è stato azzerato dal clock. L'ordine delle classi riflette l'idea che sia meglio togliere una pagina modificata ma non usata di recente piuttosto che una pagina pulita ma usata da poco. NRU è semplice ed efficiente, ma è un'approssimazione rozza.

ESEMPIO SVOLTO: NRU

In memoria ci sono le pagine A ($R = 1, M = 0$), B ($R = 1, M = 1$), C ($R = 0, M = 1$), D ($R = 1, M = 1$), E ($R = 1, M = 0$). Quale pagina sceglie NRU?

Si assegnano le classi: A ed E sono in classe 2, B e D in classe 3, C in classe 1. Non ci sono pagine in classe 0, quindi si sceglie dalla classe 1: viene rimossa **C**, pur essendo sporca, perché è l'unica non usata di recente.

19.3 FIFO, seconda chance e clock

L'algoritmo **FIFO** (*First In First Out*) tiene le pagine in una coda in ordine di caricamento e rimuove la **pagina più vecchia**, quella in memoria da più tempo. È semplicissimo, ma la scelta non è sempre felice: la pagina più vecchia può essere anche una delle **più usate** (si pensi a una pagina con variabili usate da tutto il programma).

L'algoritmo della **seconda chance** corregge FIFO **tenendo conto dello stato attuale del bit R**: si guarda la pagina più vecchia; se il suo bit R è 0, la pagina è vecchia e non usata di recente, e viene rimossa. Se invece R vale 1, la pagina riceve una **seconda possibilità**: il bit viene azzerato e la pagina viene spostata in fondo alla coda, come se fosse appena arrivata; poi si ripete il controllo sulla nuova testa. Viene quindi rimossa **la pagina più vecchia fra quelle non usate di recente**. Se tutte le pagine hanno R = 1, l'algoritmo le scorre tutte azzerandone il bit e torna alla prima, comportandosi esattamente come FIFO.

L'idea è buona, ma spostare continuamente le pagine in fondo alla coda è inefficiente. L'algoritmo dell'**orologio** (*clock*) fa la stessa cosa in modo più efficiente: le pagine sono disposte in una **lista circolare**, come le ore di un orologio, e una **lancetta** punta alla pagina più vecchia. Al page fault si esamina la pagina indicata: se R = 0 viene rimossa, la nuova pagina prende il suo posto e la lancetta avanza; se R = 1, il bit viene azzerato e la lancetta avanza alla pagina successiva, finché non se ne trova una con R = 0. Nessuna pagina viene spostata: si muove solo la lancetta.

ESEMPIO SVOLTO: SECONDA CHANCE

La coda FIFO, dalla pagina più vecchia, è: A (R = 1, M = 0), B (R = 1, M = 1), C (R = 0, M = 0), D (R = 0, M = 1), E (R = 1, M = 0). Quale pagina sceglie la seconda chance?

Si parte dalla testa. A ha R = 1: le si azzerava il bit e la si sposta in fondo. B ha R = 1: stessa sorte. C ha R = 0: viene rimossa **C**, la prima pagina incontrata con R = 0. Il bit M non conta per questo algoritmo.

DA RICORDARE

Come funziona l'algoritmo clock?

Le pagine sono disposte in un **cerchio** e una **lancetta** indica la candidata alla rimozione. Al page fault si controlla il bit R della pagina indicata: se vale 1, le si dà una seconda possibilità azzerando il bit e si avanza alla successiva; se vale 0, la pagina viene rimossa e sostituita con quella nuova, e la lancetta avanza. La lancetta non riparte mai dall'inizio: continua a girare da dove si era fermata. Fa le stesse scelte della seconda chance, ma senza spostare le pagine nella coda, quindi è più efficiente.

19.4 LRU e le sue approssimazioni: NFU e aging

Un'osservazione utile per avvicinarsi a OPT è che le pagine **usate molto di recente** probabilmente lo saranno anche **in futuro**, mentre quelle inutilizzate da molto tempo probabilmente resteranno inutilizzate. Da qui l'algoritmo **LRU** (*Least Recently Used*): rimuovere la pagina **usata meno di recente**. È un'idea eccellente, vicina all'ottimo (OPT guarda al futuro, LRU al passato), ma **non semplice e dispendiosa** da realizzare, perché bisogna conoscere l'ordine esatto dei riferimenti e aggiornarlo a **ogni** accesso alla memoria. Con un **supporto hardware** si può fare: la CPU mantiene un **contatore** che aumenta a ogni istruzione, e a ogni riferimento il suo valore viene copiato in un **campo** della voce della pagina; al page fault si sceglie la pagina con il valore più basso. Via software, invece, si può solo approssimare LRU.

L'algoritmo **NFU** (*Not Frequently Used*) è una prima **approssimazione software** di LRU. Ogni voce della tabella delle pagine ha un **contatore**, inizialmente 0. A ogni interrupt del clock il sistema **somma al contatore il valore del bit R**, prima di azzerarlo. Il contatore misura così in quanti intervalli la pagina è stata usata, e al page fault si rimuove la pagina con il **contatore più basso**. Il problema è che NFU **non dimentica mai**: può erroneamente privilegiare pagine **molto usate in passato** ma ormai **scarsamente usate di recente**, che con buona probabilità lo saranno poco anche nel prossimo futuro. Ad esempio, le pagine della prima fase di un compilatore conservano

un contatore alto anche quando il compilatore è passato alle fasi successive.

L'algoritmo di **aging** (invecchiamento) corregge NFU con una piccola modifica. A ogni scadenza del clock, per ogni pagina: il contatore viene **spostato di un bit a destra** (shift), e il bit R viene **inserito a sinistra, come bit più significativo**. Così un riferimento recente pesa molto (è nel bit più alto), mentre i riferimenti vecchi scendono verso i bit bassi e, dopo tanti cicli quanti sono i bit del contatore, escono del tutto. Al page fault si rimuove la pagina con il **contatore più basso**. Una pagina non usata negli ultimi quattro cicli avrà i primi quattro bit a zero, e il suo valore sarà sicuramente più basso di quello di una pagina usata anche una sola volta di recente. Il numero di bit del contatore fissa l'orizzonte temporale: con 8 bit, l'algoritmo non distingue fra una pagina usata 9 cicli fa e una usata 1000 cicli fa.

DA RICORDARE

Qual è l'aggiornamento periodico del contatore nell'algoritmo di aging?

Shift a destra del contatore e inserimento del bit di referenziamento R come bit più significativo. Le alternative sbagliate (shift a sinistra, somma di R, uso del bit M) non realizzano l'invecchiamento: è lo shift a destra a far pesare sempre meno i riferimenti vecchi.

Come funziona NFU e qual è il suo difetto rispetto ad aging?

A ogni interrupt del clock si somma il bit R al contatore della pagina, senza shift, e si rimuove la pagina con il contatore più basso. Il difetto è che **non dimentica il passato**: una pagina usata moltissimo tempo fa conserva un contatore alto e rischia di non essere mai scelta, anche se non viene più usata. Aging invece, con lo shift a destra, fa pesare sempre meno i riferimenti vecchi.

ESEMPIO SVOLTO: AGGIORNARE I CONTATORI DI AGING

Cinque pagine hanno, all'istante t , i contatori $A = 11100011$, $B = 10110011$, $C = 11011000$, $D = 11100111$, $E = 01011100$. All'istante $t + 1$ i bit R sono $A 0$, $B 1$, $C 1$, $D 1$, $E 0$; all'istante $t + 2$ sono $A 0$, $B 0$, $C 0$, $D 0$, $E 1$. Riportare i contatori dopo i due aggiornamenti e la pagina da rimuovere a $t + 3$.

A ogni aggiornamento si sposta il contatore di un posto a destra (il bit più a destra si perde) e si scrive R come nuovo bit più a sinistra. Ad esempio per B: 10110011 diventa 11011001 ($R = 1$), poi 01101100 ($R = 0$).

PAGINA	ISTANTE T	DOPO $T + 1$	DOPO $T + 2$
A	11100011	01110001	00111000
B	10110011	11011001	01101100
C	11011000	11101100	01110110
D	11100111	11110011	01111001
E	01011100	00101110	10010111

Il contatore più basso è quello di **A** (00111000), che viene rimossa: non è stata usata negli ultimi due cicli. E invece, usata nell'ultimo ciclo, ha il contatore più alto, anche se prima era stata poco usata.

19.5 Il confronto delle prestazioni

Per confrontare gli algoritmi si sceglie come metrica il **numero di page fault**, a parità di condizioni: stesso numero di frame (qui **3 frame**, inizialmente vuoti) e stessa sequenza di indirizzi virtuali. Della sequenza di indirizzi interessa solo il numero di pagina, e i riferimenti consecutivi alla stessa pagina si possono compattare in uno solo, perché il secondo non causa mai un fault. La sequenza classica è **7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1**. Nelle tabelle ogni colonna mostra il contenuto dei frame dopo il riferimento indicato in alto, e la F segnala un page fault.

OPT: 9 PAGE FAULT

RIF.	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame 1	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
Frame 2		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
Frame 3			1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
Fault	F	F	F	F		F		F			F			F				F		

I primi tre riferimenti riempiono i frame. Al riferimento 2 si guarda il futuro: 0 verrà usata subito, 1 al quattordicesimo riferimento, 7 solo al diciottesimo; si rimuove 7. Al riferimento 3 si rimuove 1, usata più tardi di 2 e 0, e così via.

FIFO: 15 PAGE FAULT

RIF.	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame 1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
Frame 2		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
Frame 3			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
Fault	F	F	F	F		F	F	F	F	F	F			F	F			F	F	F

FIFO rimuove sempre la pagina caricata da più tempo, indipendentemente dall'uso: ad esempio al riferimento 3 rimuove 0, che serve subito dopo, causando un altro fault.

LRU: 12 PAGE FAULT

RIF.	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame 1	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
Frame 2		0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
Frame 3			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
Fault	F	F	F	F		F		F	F	F	F			F		F		F		

LRU rimuove la pagina il cui ultimo uso è più lontano: al riferimento 4 (ottava colonna) le pagine in memoria sono 2, 0 e 3, usate per l'ultima volta rispettivamente al quarto, settimo e sesto riferimento, quindi si rimuove 2. LRU si colloca fra OPT e FIFO, molto più vicino a OPT.

L'ANOMALIA DI BELADY

Ci si aspetterebbe che, aumentando i frame disponibili, i page fault possano solo diminuire. Con FIFO non è così. Con la sequenza **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5** FIFO causa **9** page fault con 3 frame e **10** con 4 frame:

RIF.	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	4	4	4	5	5	5	5	5	5
Frame 2		2	2	2	1	1	1	1	1	3	3	3
Frame 3			3	3	3	2	2	2	2	2	4	4
Fault	F	F	F	F	F	F	F			F	F	

RIF.	1	2	3	4	1	2	5	1	2	3	4	5
Frame 1	1	1	1	1	1	1	5	5	5	5	4	4
Frame 2		2	2	2	2	2	2	1	1	1	1	5
Frame 3			3	3	3	3	3	3	2	2	2	2
Frame 4				4	4	4	4	4	4	3	3	3
Fault	F	F	F	F			F	F	F	F	F	F

È l'**anomalia di Belady**: aumentando i frame disponibili, aumentano i page fault. **LRU non ne soffre**, perché gode della **proprietà di inclusione**: in ogni istante, l'insieme delle pagine in memoria avendo n frame è **incluso** in quello che si avrebbe avendo n + 1 frame, cioè $B_t(n) \subseteq B_t(n + 1)$ per ogni t e n. Con LRU le pagine in memoria sono sempre le n usate più di recente; con un frame in più sono le stesse n più quella usata ancora prima. Un frame in più non può quindi far perdere una pagina che si sarebbe tenuta con meno frame, e i fault non possono aumentare. Anche OPT gode della proprietà di inclusione. FIFO no, perché le sue scelte dipendono solo dal momento del caricamento, e con un frame in più quel momento cambia per tutte le pagine.

Fra gli altri algoritmi, **NFU** e **aging** godono della proprietà di inclusione, perché approssimano LRU; **seconda chance** e **clock** soffrono dell'anomalia, perché quando tutte le pagine hanno R = 1 si riducono a FIFO; anche **NRU** ne soffre, perché all'interno di una classe la scelta si riduce a FIFO.

ALGORITMO	GIUDIZIO	ANOMALIA DI BELADY
OPT	Non implementabile, ma utile come termine di paragone	No
NRU	Approssimazione rozza di LRU	Sì
FIFO	Può portare all'eliminazione di pagine importanti	Sì
Seconda chance	Netto miglioramento rispetto a FIFO	Sì
Clock	Come la seconda chance, ma più efficiente	Sì
LRU	Eccellente idea, vicina all'ottimo, ma difficilmente realizzabile se non in hardware	No
NFU	Approssimazione software abbastanza rozza di LRU	No
Aging	Buona approssimazione di LRU con un'implementazione software efficiente	No

Allocazione dei frame e thrashing

Quanti frame assegnare a ogni processo, come evitare il thrashing e quali scelte di progetto influenzano le prestazioni della paginazione.

20.1 Quanti frame dare a ogni processo

Nella forma più pura, la **paginazione su richiesta** (*pure demand paging*), all'avvio di un processo **nessuna pagina** viene caricata in memoria: anche la prima istruzione causa un page fault, e il processo costruisce gradualmente in memoria le pagine che gli servono. Dopo una fase iniziale con molti fault, le pagine usate restano in memoria e i fault diventano rari. Resta da decidere **quanti frame** assegnare a ogni processo:

- c'è un **minimo strutturale**, sotto il quale il processo non può eseguire nemmeno un'istruzione, che dipende dal **set di istruzioni** e dai **livelli di indirizzamento indiretto**. Un'istruzione con un operando in memoria ha bisogno almeno di due frame, uno per l'istruzione e uno per l'operando: con un solo frame, caricare l'operando toglierebbe l'istruzione e viceversa, all'infinito. Se gli operandi sono due e usano l'indirizzamento indiretto, ognuno può toccare due pagine, e i frame necessari salgono; un'istruzione a cavallo fra due pagine ne aggiunge un altro;
- il **massimo** è la memoria libera disponibile.

Fra questi estremi si possono seguire diverse **strategie**:

- **allocazione equa**: con m frame e n processi, ognuno riceve m/n frame. È semplice, ma tratta allo stesso modo un piccolo programma e un grande database;
- **allocazione proporzionale**: al processo i , di dimensione s_i , si assegnano $a_i = s_i / S \times m$ frame, dove $S = \sum s_i$ è la somma delle dimensioni di tutti i processi. Con 62 frame e due processi di 10 e 127 pagine, il primo riceve $10/137 \times 62 \approx 4$ frame e il secondo $127/137 \times 62 \approx 57$. Le assegnazioni vanno **adeguate al livello di multiprogrammazione**: quando entra un nuovo processo, gli altri ne perdono un po';
- **allocazione per priorità**: la proporzione tiene conto anche della priorità del processo, non solo della sua dimensione.

Un'altra scelta riguarda **quali pagine considerare** come possibili vittime al page fault. Con l'**allocazione locale** si sceglie solo **fra le pagine dello stesso processo**, che conserva sempre lo stesso numero di frame. Con l'**allocazione globale** si considerano **tutte le pagine**, anche di altri processi: il numero di frame di un processo cambia nel tempo, e chi ne ha più bisogno finisce per averne di più. L'allocazione globale si adatta meglio, ma il comportamento di un processo dipende anche da quello degli altri.

DA RICORDARE

Che differenza c'è fra la paginazione su richiesta pura e il caricamento tradizionale? Perché all'inizio ci sono tanti page fault?

Nel caricamento tradizionale il programma viene caricato in memoria, almeno in parte, prima di iniziare l'esecuzione. Con la paginazione su richiesta pura **nessuna pagina** è in memoria all'avvio: anche la prima istruzione causa un page fault, e ogni pagina toccata per la prima volta (codice, dati, stack) ne causa un altro. Per questo la fase iniziale ha molti fault: il processo sta costruendo in memoria l'insieme delle pagine che usa. Una volta formato, quasi tutti gli accessi cadono su pagine già presenti e i fault diventano rari. In cambio non si caricano mai pagine che il processo non userà.

20.2 Il thrashing e il working set

Che cosa succede se un processo ha **pochi frame**? Se è **sotto il minimo strutturale**, non può eseguire: viene **sospeso** e le sue pagine vengono spostate su disco con lo swapping. Se è **poco sopra** il minimo, il processo può eseguire, ma ogni pagina che carica ne toglie un'altra che gli servirà subito dopo: causa page fault in continuazione e passa più tempo ad aspettare il disco che a eseguire. È il **thrashing**. Quando il thrashing riguarda **tutti i processi** si parla di **sistema in sovraccarico**, dovuto a un **eccessivo livello di multiprogrammazione**: i processi si rubano i frame a vicenda, la CPU resta inattiva e un sistema ingenuo, vedendo la CPU poco usata, potrebbe persino ammettere altri processi, peggiorando la situazione.

DEFINIZIONE

Si ha **thrashing** quando un processo, o l'intero sistema, passa più tempo a gestire page fault che a eseguire istruzioni utili, perché le pagine che servono vengono continuamente tolte dalla memoria. Il **working set** di un processo è l'insieme delle pagine che ha usato negli ultimi riferimenti: se a ogni processo si garantisce un numero di frame sufficiente a contenerlo, il thrashing si evita.

Bisognerebbe quindi assegnare a ogni processo un numero di frame **commisurato alle sue necessità**. Lo strumento per stimarle è il **modello di località**: in ogni fase della sua esecuzione un processo usa attivamente solo un certo insieme di pagine, la sua **località** (ad esempio il codice di una funzione con le sue variabili e una parte di un array), e durante l'esecuzione passa da una località all'altra. La **strategia di base** è dare a ogni processo abbastanza frame da contenere la località corrente: in quel caso i fault avvengono solo ai cambi di località.

Il **working set** rende misurabile la località. Per ogni processo si considerano le pagine usate negli ultimi Δ accessi alla memoria: questo insieme è il working set, e il numero di pagine che contiene è la sua dimensione, **WSS** (*working set size*). Ad esempio, con $\Delta = 5$, se gli ultimi cinque riferimenti sono 1, 2, 3, 4, 5 il working set ha dimensione 5; se sono 3, 4, 3, 3, 4 ha dimensione 2. La scelta di Δ è delicata: deve adattarsi alla località corrente, perché un Δ troppo piccolo non copre l'intera località e uno troppo grande ne copre più d'una.

Conoscendo il working set di ogni processo, il sistema calcola la **richiesta globale di frame**, $D = \sum WSS_i$, e la confronta con la **memoria disponibile** m . Se D supera m , qualche processo andrà in thrashing: il sistema **previene il thrashing** sospendendo uno dei processi e liberandone i frame. Se invece avanzano frame, può avviare un nuovo processo. Calcolare il working set esattamente sarebbe troppo costoso, ma lo si può **approssimare** con **interrupt periodici** e il **bit di referenziamento R**: a ogni interrupt si registra lo stato di R di ogni pagina e lo si azzerà, conservando un **log** con la storia degli ultimi bit R , lungo in base al parametro Δ . Una pagina con almeno un 1 nella sua storia fa parte del working set (i contatori dell'algoritmo di aging contengono proprio questa storia).

Un modello più diretto per affrontare il thrashing è il monitoraggio della **frequenza dei page fault** (PFF, *Page Fault Frequency*) di ogni processo. Si fissano un **limite superiore** e uno **inferiore**: se un processo supera il limite superiore, ha troppo pochi frame e gliene vengono dati altri; se scende sotto quello inferiore, ne ha più del necessario e gliene vengono tolti. Se molti processi superano il limite superiore e non ci sono frame liberi, il sistema è **in sovraccarico** e bisogna sospendere qualche processo. I due modelli sono in relazione: i picchi di frequenza dei fault corrispondono ai **cambi di località**, cioè ai cambi di working set, e si riassorbono quando il nuovo working set è stato caricato.

DA RICORDARE

Che cos'è il working set e come si collega al thrashing?

Il working set di un processo è l'insieme delle pagine usate negli **ultimi Δ accessi** alla memoria: approssima la località corrente, cioè le pagine che il processo sta usando attivamente. Il **thrashing** si verifica quando la memoria non basta a contenere i working set dei processi: ogni processo, per caricare una pagina, ne toglie una che fa parte del working set suo o di un altro, e il sistema passa la maggior parte del tempo a gestire page fault invece di fare lavoro utile. Il working set permette di prevenirlo: se la somma delle dimensioni dei working set supera i frame disponibili, si **riduce il grado di multiprogrammazione** sospendendo qualche processo, finché i working set non tornano a stare in memoria.

20.3 La politica di pulitura

La gestione dei page fault è efficiente soprattutto se ci sono **sempre frame liberi**: in quel caso il fault richiede solo di leggere la pagina, senza scegliere una vittima e magari riscriverla sul disco. Molti sistemi, ad esempio **Linux** e **Windows**, usano allora un processo di servizio, il **paging daemon**, che controlla periodicamente lo stato di occupazione globale dei frame. Se i frame liberi sono troppo pochi, il daemon **seleziona** delle pagine con l'algoritmo di sostituzione, le **libera** e, se sono sporche, le **pulisce** riscrivendole sul disco, mantenendo così un **pool di frame liberi**. I frame del pool conservano il loro vecchio contenuto finché non vengono riassegnati: se nel frattempo il processo a cui sono stati tolti ne ha di nuovo bisogno, la pagina viene **ripescata dal pool** senza alcun accesso al disco.

20.4 La dimensione della pagina

La scelta della dimensione di base della pagina è un compromesso. Una pagina **grande** ha tre vantaggi: la **tabella delle pagine è più piccola**, perché le pagine sono meno; il **trasferimento da e verso il disco è più efficiente**, perché il tempo di posizionamento della testina viene ammortizzato su più dati; e tende a **ridurre il numero di page fault**, perché è più probabile che il prossimo accesso cada in una pagina già caricata (come camminare su piastrelle grandi: il passo successivo cade più spesso sulla stessa piastrella). Una pagina **piccola** ha invece **minore frammentazione interna**, perché l'ultima pagina spreca meno, e permette una **migliore risoluzione** nel definire il working set: si caricano solo le parti davvero usate, con meno memoria sprecata. C'è infine una **relazione con la dimensione del blocco su disco**: conviene che la pagina sia grande quanto un blocco, o un suo multiplo.

Condivisione, ottimizzazioni e memoria del kernel

Le tecniche che sfruttano la paginazione per condividere memoria e risparmiare copie, e il modo in cui il kernel alloca la propria memoria.

21.1 Le pagine condivise

I processi possono **condividere** alcune pagine in vari modi. In **sola lettura**, tipicamente il **codice eseguibile**: se cento utenti eseguono lo stesso editor, il codice può stare in memoria una volta sola, purché sia **codice rientrante**, cioè che non modifica se stesso. In **lettura e scrittura**, per la **comunicazione fra processi tramite memoria condivisa**: due processi chiedono al sistema un segmento comune, e ciò che uno scrive l'altro lo vede subito. Con una tabella delle pagine ordinaria o multilivello la realizzazione è **semplice ed efficiente**: basta che le voci delle pagine condivise, nelle tabelle dei diversi processi, puntino agli **stessi frame**.

Le difficoltà nascono altrove. La prima è la **gestione della cache**: con una cache basata su indirizzi virtuali, anche con gli ASID, lo stesso dato condiviso può comparire in due linee diverse (con indirizzi virtuali o ASID diversi), e se un processo modifica la propria copia l'altro continua a vedere quella vecchia. Le soluzioni sono **disabilitare la cache sulle pagine condivise**, oppure usare cache con **ricerca basata su indirizzi virtuali e tag fisici**: la cache cerca le linee candidate in parallelo con la TLB, usando l'indirizzo virtuale, e per capire se si tratta di un duplicato aspetta che la TLB fornisca l'indirizzo fisico, che è univoco. La seconda difficoltà riguarda la **tabella delle pagine invertita**, che ha una sola voce per frame e quindi può associare un frame a **una sola** coppia (processo, pagina). Con un solo core si può **modificare la voce** a ogni cambio di contesto, o a ogni page fault causato dall'altro processo; con più core, dove i processi che condividono il frame possono eseguire contemporaneamente, il problema è **difficilmente gestibile**, se non con teoriche tabelle invertite con corrispondenze molti a uno.

21.2 Copy on write e zero fill on demand

La condivisione si può spingere oltre con il **copy on write**, che **condivide finché è possibile tutti i tipi di pagine**, codice e dati. L'esempio tipico è la fork: invece di copiare subito tutta la memoria del padre, il sistema fa puntare la tabella delle pagine del figlio agli **stessi frame** del padre e marca tutte quelle pagine come **sola lettura** in entrambe le tabelle. Finché i due processi leggono, nessuna copia è necessaria. Quando uno dei due prova a **scrivere**, la MMU rileva la violazione del permesso e genera un'eccezione; il sistema operativo riconosce che si tratta di una pagina in copy on write, ne fa una **copia** in un nuovo frame, la rende scrivibile per il processo che scriveva e fa ripartire l'istruzione. Si copiano così solo le pagine davvero modificate; se il figlio fa subito una exec, non si copia quasi nulla.

Lo **zero fill on demand** si basa su un principio simile: le **nuove pagine** richieste da un processo (heap, stack, dati non inizializzati) devono essere **vuote**, cioè piene di zeri, e vengono **allocate su richiesta**, solo quando il processo le usa davvero. Azzerare le pagine è necessario per la **sicurezza**: un frame riciclato contiene ancora i dati del processo che lo usava prima, e senza azzeramento un nuovo processo potrebbe leggerli. L'**azzeramento** è gestito **in modo efficiente dal kernel**, che mantiene un **pool di pagine vuote**, azzerate in anticipo nei momenti di inattività. Inoltre, quando un processo chiede nuove pagine, il sistema può farle puntare tutte a un'unica **pagina di zeri statica in sola lettura** (*read only static zero page*), applicando il **copy on write**: un frame vero viene assegnato solo quando il processo scrive.

DA RICORDARE

Come fa il sistema a far scattare la copia nel copy on write? Che ruolo ha la maschera dei permessi?

Se ne accorge grazie alla **MMU**, che controlla i permessi a ogni accesso. Dopo la fork tutte le pagine condivise fra padre e figlio vengono marcate **in sola lettura** nella maschera dei permessi delle due tabelle delle pagine. Quando uno dei processi tenta di scriverci, la MMU rileva una violazione e genera un'eccezione; il sistema operativo verifica che la pagina è in sola lettura per via del copy on write (e non per un vero errore), la copia in un nuovo frame, dà alla copia il permesso di scrittura per il processo che stava scrivendo e fa ripartire l'istruzione. Da quel momento le due pagine sono indipendenti. Il copy on write sfrutta quindi un meccanismo di protezione già esistente per intercettare la prima scrittura.

Che cos'è lo zero fill on demand, e perché serve anche alla sicurezza?

È la tecnica per cui le nuove pagine di un processo vengono riempite di zeri solo quando vengono usate per la prima volta. Per le prestazioni evita di azzerare pagine che il processo non userà mai. Per la **sicurezza** è indispensabile perché i frame liberati da un processo contengono ancora i suoi dati, magari riservati: se un altro processo ricevesse un frame senza azzeramento, potrebbe leggerli. Azzerando sempre le pagine prima del primo uso, nessun processo vede mai i dati di un altro.

DA RICORDARE

Che cos'è l'overbooking della RAM (memory overcommit)?

È la pratica per cui il sistema promette ai processi, in totale, **più memoria virtuale** di quanta RAM (e area di swap) ci sia davvero, scommettendo sul fatto che raramente tutti i processi useranno tutta la memoria richiesta nello stesso momento. È resa possibile proprio dalla paginazione su richiesta e dallo zero fill on demand: quando un processo chiede memoria, il sistema riserva solo lo spazio virtuale, e i frame vengono assegnati solo quando le pagine vengono toccate. Grazie a questo, la paginazione permette di eseguire molti più processi di quanti ne entrerebbero in memoria per intero. Il rischio è che, se troppi processi usano davvero la memoria promessa, la memoria finisca: Linux gestisce questo caso con l'**OOM killer** (*Out Of Memory killer*), che termina forzatamente alcuni processi per liberare memoria.

21.3 Librerie condivise e file mappati

I programmi usano comunemente grandi **librerie**, come quella standard del C o quelle grafiche. Con il **linking statico** il codice delle funzioni usate viene **incluso nell'eseguibile** in fase di linking: ogni programma ne contiene una copia, sul disco e in memoria. Con il **linking dinamico** l'eseguibile contiene solo dei **riferimenti** alle funzioni, e la **libreria condivisa** viene **collegata e caricata a run time**, al primo utilizzo; su Windows sono le **DLL** (*Dynamic Link Libraries*). I vantaggi sono due: un forte **risparmio di spazio**, sia su disco sia in RAM, perché la libreria è presente una sola volta e le sue pagine sono condivise da tutti i processi che la usano; e lo **sviluppo indipendente**, con la **facilità di aggiornamento**: correggere un errore nella libreria non richiede di ricompilare i programmi, purché l'interfaccia delle funzioni resti la stessa.

I **file mappati** in memoria sono un **modello alternativo di I/O su file**. Un processo chiede al sistema di mappare un file in una zona del proprio spazio virtuale; da quel momento legge e scrive il file accedendo a quella zona come a un normale array, senza read e write. Al momento della mappatura non viene caricato nulla: le pagine della zona sono marcate come assenti, e il primo accesso a ciascuna causa un page fault, con cui il sistema carica la parte corrispondente del file. Le pagine modificate vengono riscritte sul file quando vengono rimosse o quando il file viene chiuso. È la paginazione su richiesta applicata ai file, e la stessa chiamata mmap la ritroverai in laboratorio. I file mappati offrono la **possibilità di condivisione**: se due processi mappano lo stesso file, le loro pagine puntano agli stessi frame, e le scritture di uno sono visibili all'altro. Con questo meccanismo il sistema gestisce **automaticamente le librerie condivise**, il **caricamento del codice eseguibile** (l'eseguibile viene mappato e le sue pagine caricate solo quando servono) e il **caricamento dei dati statici**.

DA RICORDARE

Che cosa significa mappare un file in memoria, e che cosa succede nella tabella delle pagine?

Significa collegare una zona dello spazio di indirizzamento virtuale del processo al contenuto di un file, così che il processo lo legga e lo scriva come un array in memoria. Alla chiamata `mmap` il sistema non carica nulla: riserva la zona virtuale e crea nella tabella delle pagine una voce per ogni sua pagina, marcata **non presente**. Quando il processo accede a una di quelle pagine si ha un `page fault`: il sistema trova un frame libero, individua sul disco la parte corrispondente del file, la carica nel frame e aggiorna la voce. Si carica quindi solo ciò che viene usato, e più processi che mappano lo stesso file in modo condiviso usano gli stessi frame.

21.4 L'allocazione della memoria per il kernel

La memoria dei **processi utente** è **paginata**, e accetta la **frammentazione interna** dell'ultima pagina. La **memoria interna al kernel** ha esigenze diverse. Il kernel alloca e libera continuamente strutture dati piccole e di **dimensione fissa per tipo**: un PCB ogni volta che nasce un processo, un semaforo, un inode quando si apre un file. Allocare una pagina intera per ognuna sarebbe uno spreco enorme, quindi si mira a una **frammentazione interna minima o nulla**. Inoltre in alcuni casi **non si può paginare** l'allocazione: certi dispositivi, come i controller DMA, lavorano direttamente con indirizzi fisici, scavalcando la MMU, e richiedono memoria **fisicamente contigua**.

Linux usa lo **slab allocator**. Uno **slab** è una **sequenza di pagine fisicamente contigue**; una **cache** è formata da uno o più slab, e c'è **una cache per ogni tipo di struttura dati interna omogenea**: una per i PCB, una per gli inode, una per i semafori, e così via. Ogni slab di una cache è diviso in **oggetti** della dimensione esatta di quella struttura: ad esempio, con pagine da 4096 byte e PCB da 320 byte, uno slab di una pagina contiene $4096 / 320 = 12$ PCB ($12 \times 320 = 3840$ byte, il resto è troppo piccolo per un tredicesimo). Lo **stato** di uno slab può essere **pieno** (tutti gli oggetti in uso), **vuoto** (tutti liberi) o **parziale**. La gestione è **dinamica**: una richiesta viene servita prima da uno slab parziale, poi da uno vuoto, e solo se non ce ne sono si crea un nuovo slab; quando una struttura viene liberata, il suo oggetto torna libero nello slab, pronto per la prossima richiesta dello stesso tipo, e gli slab vuoti possono essere restituiti al sistema se serve memoria. I **vantaggi** sono **niente spreco**, perché ogni oggetto ha esattamente la dimensione richiesta e non c'è frammentazione, ed **efficienza**, perché allocare significa solo prendere un oggetto libero già pronto, senza cercare spazio con first fit o best fit.

DA RICORDARE

Perché l'allocazione della memoria per il kernel è un caso particolare, e perché si usa lo slab allocator?

Perché il kernel alloca **in continuazione** strutture **piccole e di dimensione fissa per tipo** (PCB, inode, semafori), e in alcuni casi ha bisogno di memoria **fisicamente contigua**, ad esempio per il DMA, che non passa dalla MMU. Un allocatore generico a pagine sprecherebbe molto spazio, e uno con first fit o best fit sarebbe lento e soggetto a frammentazione. Lo slab allocator sfrutta proprio la regolarità delle richieste: per ogni tipo di struttura tiene una **cache di slab**, cioè gruppi di pagine contigue divise in oggetti della dimensione esatta. Allocare vuol dire prendere un oggetto libero, liberare vuol dire marcarlo di nuovo libero. Non c'è spreco e le operazioni sono velocissime.

Che cosa sono le cache dello slab allocator, e in che stati può trovarsi uno slab?

Una cache è l'insieme degli slab dedicati a un tipo di oggetto: c'è una cache per i PCB, una per gli inode, e così via. Ogni slab può essere **pieno**, con tutti gli oggetti in uso; **parziale**, con alcuni oggetti liberi, ed è da qui che si prende per primo il prossimo oggetto; oppure **vuoto**, con tutti gli oggetti liberi, che può essere usato per nuove richieste o restituito al sistema se serve memoria per altro.

Il file system: file e directory

Il file system visto dall'utente: file, directory, accesso condiviso, strutture per i file aperti e organizzazione del disco.

22.1 Il problema e le astrazioni

I processi hanno bisogno di memorizzare **grandi quantità di informazioni**, in modo **persistente**, cioè che sopravviva alla terminazione del processo e allo spegnimento della macchina, e **condiviso**, cioè accessibile da più processi. La memoria centrale non soddisfa nessuno dei tre requisiti; il disco sì, ma visto dall'hardware è solo una lunga sequenza di **blocchi numerati**, che non sa nulla di nomi o cartelle. Il sistema operativo costruisce sopra il disco due **astrazioni**: il **file**, un'unità logica di informazione con un nome, e la **directory**, una raccolta di file e di altre directory che forma una struttura ad albero. L'insieme dei dettagli di gestione e di implementazione di file e directory costituisce il **file system**, termine che indica sia la parte del sistema operativo che se ne occupa sia l'organizzazione dei dati su un disco.

DEFINIZIONE

Un **file** è un'unità logica di informazione, identificata da un nome, che il sistema operativo conserva in modo persistente. Una **directory** è una raccolta di file e di altre directory, che organizza il file system in una struttura ad albero. Il **file system** è la parte del sistema operativo che gestisce file e directory.

22.2 I file

Le caratteristiche di un file system che l'utente vede sono diverse:

- la **nomenclatura**: lunghezza massima dei nomi, caratteri ammessi, presenza di un'estensione e distinzione fra maiuscole e minuscole. MS DOS ammetteva 8 caratteri più 3 di estensione; i sistemi UNIX distinguono `test.txt` da `Test.txt`, mentre Windows li considera lo stesso file;
- i **tipi di file**: i **file regolari**, che contengono dati; le **directory**, file di sistema che mantengono la struttura del file system; nei sistemi UNIX anche i **file speciali di dispositivo**, che rappresentano un dispositivo fisico (un disco, una scheda audio) e permettono di leggerlo e scriverlo come un file qualsiasi. I file system moderni trattano tutti i file regolari come semplici sequenze di byte, lasciando alle applicazioni il compito di interpretarli;
- i **tipi di accesso**: **sequenziale**, in cui i byte si leggono uno dopo l'altro dall'inizio, come su un nastro magnetico, oppure **casuale**, in cui ci si può posizionare in un punto qualsiasi del file (con un'operazione di seek) e leggere o scrivere da lì. Oggi tutti i dischi permettono l'accesso casuale;
- i **metadati** (o **attributi**): dimensione, proprietario, permessi, date di creazione, ultima modifica e ultimo accesso;
- le **operazioni supportate**: creare e cancellare un file, aprirlo e chiuderlo, leggere, scrivere, aggiungere in fondo (*append*), posizionarsi, leggere e modificare gli attributi, rinominarlo.

22.3 L'accesso condiviso e i lock

Più processi possono aprire lo stesso file contemporaneamente, e si ripresentano gli stessi problemi di corse critiche visti per la memoria condivisa. I file system offrono quindi dei **lock** sui file, o su parti di essi, di due tipi:

- **shared** (condiviso) ed **exclusive** (esclusivo): un lock condiviso, tipicamente richiesto per leggere, è compatibile con altri lock condivisi, quindi più lettori possono lavorare insieme; un lock esclusivo, richiesto per scrivere, non

è compatibile con nessun altro lock. È la stessa logica dei lettori e scrittori;

- **mandatory** (obbligatorio) e **advisory** (consultivo): con un lock obbligatorio è il **sistema operativo** a impedire l'accesso a chi non rispetta il lock; con un lock consultivo il sistema si limita a registrare chi ha il lock, ed è compito dei processi controllarlo e rispettarlo, come un accordo fra programmi cooperanti. Windows usa in genere lock obbligatori, i sistemi UNIX lock consultivi.

22.4 Le strutture dati per i file aperti

Quando un processo apre un file, il sistema usa due livelli di **strutture dati in memoria**. Una tabella **globale** dei file aperti, unica nel sistema, contiene una voce per ogni file aperto, con le informazioni che non dipendono dal processo: il riferimento all'inode o alla posizione del file sul disco, il numero di processi che lo hanno aperto, i lock attivi. Una tabella **per processo**, collegata al PCB, contiene una voce per ogni file aperto da quel processo, identificata da un piccolo intero, il **descrittore di file** (in UNIX 0, 1 e 2 sono standard input, output ed error), con le informazioni proprie del processo, come la **posizione corrente** nel file e la modalità di apertura; ogni voce punta alla voce corrispondente della tabella globale. Così due processi possono leggere lo stesso file da punti diversi.

22.5 La struttura del disco

Un disco può essere diviso in **partizioni**, zone indipendenti ognuna delle quali può contenere un file system diverso (ad esempio per installare due sistemi operativi). Nello schema tradizionale, il settore 0 del disco è il **Master Boot Record** (MBR), che contiene il codice di avvio e, alla fine, la **tabella delle partizioni**, con l'inizio e la fine di ciascuna (al massimo quattro); una partizione è marcata come **attiva**. All'accensione il BIOS esegue il codice dell'MBR, che individua la partizione attiva e ne legge il primo blocco, il **boot block** (o *boot record*), il quale a sua volta carica il sistema operativo contenuto nella partizione. Ogni partizione inizia con un boot block, anche se non contiene un sistema avviabile.

Dopo il boot block c'è il **superblocco**, che contiene i parametri fondamentali del file system: il tipo, il numero di blocchi, la loro dimensione e altre informazioni amministrative. Viene letto in memoria quando il file system viene montato, e se si danneggia il file system diventa inaccessibile; per questo molti file system ne tengono più copie. Seguono le strutture per la **gestione dello spazio libero**, gli **inode** (nei file system che li usano), la **directory radice** e infine i file e le directory. Il layout moderno alternativo all'MBR è la **GPT** (*GUID Partition Table*), definita dallo standard **EFI** (UEFI), che supera i limiti dell'MBR sul numero e sulla dimensione delle partizioni; l'avvio è affidato a una partizione apposita, la partizione EFI, formattata in FAT, che contiene i programmi di avvio dei sistemi operativi.

L'implementazione dei file

Come il file system registra quali blocchi appartengono a ogni file: allocazione contigua, liste collegate, FAT e inode.

Il problema centrale è tenere traccia di **quali blocchi del disco appartengono a ogni file**, e in che ordine. Le soluzioni sono quattro, di complessità crescente.

23.1 Allocazione contigua

Ogni file è memorizzato in una sequenza di **blocchi consecutivi**: un file di 4 blocchi che inizia al blocco 20 occupa i blocchi 20, 21, 22 e 23. I vantaggi sono notevoli. Per ogni file basta ricordare **il primo blocco e il numero di blocchi**; l'**accesso casuale** è immediato, perché il blocco i -esimo è semplicemente primo blocco + i ; e la **lettura dell'intero file** è velocissima, con un solo posizionamento della testina all'inizio. Il problema è la **frammentazione esterna**: cancellando file si creano buchi, e un nuovo file può non entrare in nessuno di essi; compattare il disco sarebbe lentissimo. Inoltre bisogna **conoscere in anticipo la dimensione massima** del file, perché se in seguito il file cresce e il blocco successivo è occupato, non c'è modo di estenderlo senza spostarlo. L'allocazione contigua è però perfetta per i supporti in **sola lettura** come CD ROM, DVD e Blu ray, dove le dimensioni dei file sono note al momento della scrittura.

23.2 Allocazione con liste collegate

Nell'**allocazione concatenata** ogni file è una **lista collegata di blocchi**: la prima parola di ogni blocco contiene il numero del blocco successivo, e il resto contiene dati. La directory registra per ogni file solo il **primo blocco**. Non c'è frammentazione esterna, perché qualunque blocco libero è utilizzabile, e la **lettura sequenziale** è semplice. Restano però tre problemi: l'**accesso casuale è lentissimo**, perché per arrivare al blocco n bisogna leggere dal disco tutti gli $n - 1$ precedenti, seguendo i puntatori; parte di ogni blocco è occupata dal puntatore, quindi i dati in un blocco **non sono più una potenza di 2** (con blocchi da 1024 byte e puntatori da 4 byte, ogni blocco contiene 1020 byte di dati), cosa scomoda perché i programmi leggono e scrivono di solito in multipli di potenze di 2; e un solo puntatore danneggiato fa perdere tutto il resto del file. Resta naturalmente la **frammentazione interna** nell'ultimo blocco.

ESEMPI SVOLTI: ALLOCAZIONE CONCATENATA

- **Con blocchi da 4 KB e numeri di blocco di 32 bit, quanti blocchi servono per un file di 40 KB?** Ogni blocco riserva 4 byte al puntatore, quindi contiene $4096 - 4 = 4092$ byte di dati. Il file occupa $40 \times 1024 = 40960$ byte, e $40960 / 4092 \approx 10,01$: dieci blocchi non bastano per pochi byte, quindi ne servono **11**.

Esercizio con la directory. I blocchi sono da 1 KB e la prima parola (32 bit) di ogni blocco contiene il numero del blocco successivo (0 indica la fine). La directory contiene fract.c (primo blocco 233, dimensione 3056 byte) e hello.c (primo blocco 123, dimensione 1036 byte). Del disco sono note le prime parole di alcuni blocchi:

PAROLA	BLOCCO 321	BLOCCO 123	BLOCCO 200	BLOCCO 233	BLOCCO 821
0	0	200	0	821	321
1	544	288	233	822	212
2	20	201	322	915	999
3	555	280	530	50	0
4	922	399	742	51	843

Ogni blocco contiene **1020 byte di dati**, dopo la parola del puntatore. **Word di offset 2048 di fract.c:** $2048 / 1020 = 2$ con resto 8, quindi si trova nel **terzo blocco** del file, 8 byte dopo l'inizio dei dati. La catena è $233 \rightarrow 821 \rightarrow 321$ (la prima parola di 233 vale 821, quella di 821 vale 321). Nel blocco 321 i dati iniziano al byte 4, quindi l'offset 8 dei dati è il byte 12, cioè la **parola 3**: il valore è **555**. Il blocco 321 ha puntatore 0, e infatti è l'ultimo: tre blocchi da 1020 byte contengono $3060 \geq 3056$ byte.

Ultima word di hello.c: 1036 byte occupano un primo blocco intero (1020 byte, blocco 123) e 16 byte del secondo, il blocco 200 (puntatore di 123). Nel blocco 200 i 16 byte di dati occupano le parole da 1 a 4, quindi l'ultima word del file è la **parola 4** del blocco 200: **742**.

23.3 La FAT

Il difetto principale delle liste collegate si elimina **togliendo i puntatori dai blocchi** e raccogliendoli in una tabella in memoria, la **FAT** (*File Allocation Table*, o **allocazione tabellare**). La FAT ha **una voce per ogni blocco** del disco; la voce i contiene il numero del blocco che segue il blocco i nel suo file, oppure un valore speciale che indica la **fine del file**, oppure un valore che indica che il blocco è **libero**. La directory registra per ogni file il **primo blocco**, e da lì si segue la catena nella FAT. Ad esempio, un file nei blocchi 4, 7, 2, 10 e 12 è rappresentato da $FAT[4] = 7$, $FAT[7] = 2$, $FAT[2] = 10$, $FAT[10] = 12$ e $FAT[12] = \text{fine}$.

I blocchi ora sono interamente dedicati ai dati. La FAT risiede sul disco, ma quando il file system viene montato viene **caricata in memoria**: seguire la catena diventa una serie di accessi in RAM, e l'**accesso casuale** richiede solo di scorrere la catena in memoria, senza leggere i blocchi intermedi dal disco. Il prezzo è che **l'intera tabella deve stare in memoria**, e la sua dimensione cresce con il disco: **dimensione della FAT = numero di blocchi × dimensione di una voce**. Blocchi più grandi riducono la FAT ma aumentano la frammentazione interna. Poiché perdere la FAT significa perdere tutti i file, sul disco se ne tengono di solito **due copie**. Infine, con la FAT **non serve una struttura separata per i blocchi liberi**: lo stato di ogni blocco è già scritto nella sua voce. Le versioni storiche si distinguono per la dimensione delle voci: FAT12, FAT16, FAT32; **exFAT** è la versione moderna per le unità rimovibili.

DA RICORDARE

Che cos'è la FAT e da cosa dipende la sua dimensione? Chi fornisce il primo blocco di un file?

È una tabella con **una voce per ogni blocco** del disco, in cui ogni voce indica il blocco successivo del file, la fine del file o il fatto che il blocco è libero: i file sono catene di blocchi descritte nella tabella. La sua dimensione è **numero di blocchi del disco × dimensione di una voce**. Risiede sul disco, ma viene caricata **interamente in RAM** al montaggio, perché seguire la catena leggendo il disco a ogni passo sarebbe troppo lento. Il primo blocco di un file è fornito dalla **directory**, che contiene la coppia (nome del file, primo blocco); da lì si segue la catena fino al valore di fine file.

ESEMPI SVOLTI: SEGUIRE LA CATENA NELLA FAT

Il procedimento è sempre lo stesso: si divide l'offset per la dimensione del blocco, e il quoziente intero dice **quale blocco del file** (contando da 0) contiene il byte; poi si segue la catena nella FAT dal primo blocco indicato dalla directory, facendo tanti passi quanto vale il quoziente.

Esercizio 1. Blocchi da 4 KB; la directory indica che pippo.txt inizia al blocco 7; la FAT contiene $1 \rightarrow 4, 2 \rightarrow 3, 3 \rightarrow 15, 4 \rightarrow 5, 5 \rightarrow 10, 6 \rightarrow 12, 7 \rightarrow 1, 8 \rightarrow 2, 9 \rightarrow 3, 10 \rightarrow \text{fine}$. In quale blocco si trova l'offset 10100, e qual è la dimensione minima del file? La catena di pippo.txt è $7 \rightarrow 1 \rightarrow 4 \rightarrow 5 \rightarrow 10$, cinque blocchi. $10100 / 4096 = 2$ (resto 1908), quindi è il **terzo blocco** del file (indice 2): il blocco **4**. Il file occupa cinque blocchi, quindi i primi quattro sono sicuramente pieni e il quinto contiene almeno un byte: la dimensione minima è $4 \times 4096 + 1 = 16\ 385$ byte (la massima sarebbe $5 \times 4096 = 20\ 480$).

Esercizio 2. Blocchi da 4 KB; la FAT contiene $30 \rightarrow 31, 31 \rightarrow 37, 32 \rightarrow 36, 33 \rightarrow 32, 34 \rightarrow 35, 35 \rightarrow 30, 36 \rightarrow 40, 37 \rightarrow 41$. Il byte 6758 di un file che inizia al blocco 33 è nel blocco del file di indice $6758 / 4096 = 1$, cioè $FAT[33] = 32$. Il byte 8212 di un file che inizia al blocco 34 è nel blocco di indice 2: $34 \rightarrow 35 \rightarrow 30$. Il byte 4094 dello stesso file è nel blocco di indice 0: **34**, il primo, senza consultare la FAT.

Esercizio 3. Blocchi da 3 KB (3072 byte); FAT: $0 \rightarrow 7, 1 \rightarrow 2, 2 \rightarrow 11, 3 \rightarrow 5, 4 \rightarrow 3, 5 \rightarrow 20, 6 \rightarrow 13, 7 \rightarrow 1, 8 \rightarrow 0$. Il byte 6758 del file che inizia al blocco 8 è nel blocco di indice $6758 / 3072 = 2: 8 \rightarrow 0 \rightarrow 7$. Il byte 4017 del file che inizia al blocco 3 è nel blocco di indice 1: **5**. Il byte 2044 del file che inizia al blocco 4 è nel blocco di indice 0: **4**.

ESEMPI SVOLTI: DIMENSIONI DELLA FAT

- **Blocchi da 1 KB, voci della FAT di 3 byte.** Con 24 bit si numerano 2^{24} blocchi, quindi la capacità massima del disco è 2^{24} blocchi $\times 2^{10}$ byte = 2^{34} byte = **16 GiB**. La FAT ha 2^{24} voci da 3 byte: 3×2^{24} byte = **48 MiB**. Se il file pippo occupa nell'ordine i blocchi 15, 30, 16, 64 e 40, lo descrivono **5 voci**: $FAT[15] = 30, FAT[30] = 16, FAT[16] = 64, FAT[64] = 40, FAT[40] = \text{fine file}$.
- **Blocchi da 1 KB, voci di 32 bit.** Capacità massima 2^{32} blocchi $\times 2^{10}$ byte = 2^{42} byte = **4 TiB**; la FAT occupa $2^{32} \times 4$ byte = 2^{34} byte = **16 GiB**. In memoria, con pagine da 4 KB, sono $2^{34} / 2^{12} = 2^{22} = 4\ 194\ 304$ pagine: con voci così grandi la FAT è ingestibile, ed è il motivo per cui sui dischi grandi si usano gli inode.
- **FAT16 con blocchi da 2 KB, FAT a partire dal blocco 0.** Le voci da 16 bit indirizzano 2^{16} blocchi di dati. La FAT occupa $2^{16} \times 2 = 2^{17}$ byte, cioè $2^{17} / 2^{11} = 64$ blocchi, quindi il primo blocco di dati è il **64**. Per contenere tutti i blocchi indirizzabili più la FAT il disco deve avere $2^{16} + 64$ blocchi, cioè $2^{27} + 2^{17}$ byte.
- **Disco da 1024 MB con blocchi da 8 KB.** I blocchi sono $2^{30} / 2^{13} = 2^{17}$, quindi per numerarli servono 17 bit: fra 1, 2 e 4 byte, servono **4 byte** per voce. La FAT occupa $2^{17} \times 4 = 2^{19}$ byte = **512 KB**, cioè $512 / 8 = 64$ blocchi sul disco. Un file nei blocchi 3, 6, 0 e 10 (in quest'ordine) è descritto da $FAT[3] = 6, FAT[6] = 0, FAT[0] = 10, FAT[10] = \text{fine file}$.
- **Disco da 512 MB, blocchi da 16 KB, numeri di blocco di 16 bit.** I blocchi sono $2^{29} / 2^{14} = 2^{15}$, tutti indirizzabili con 16 bit. La FAT occupa $2^{15} \times 2 = 2^{16}$ byte = 64 KB, cioè **4 blocchi**. Per trovare il blocco che contiene il byte 125 384 di un file: $125\ 384 / 16\ 384 = 7$, quindi è il blocco di indice 7 del file, e partendo dal primo blocco servono **7 accessi alla FAT** per seguire la catena.

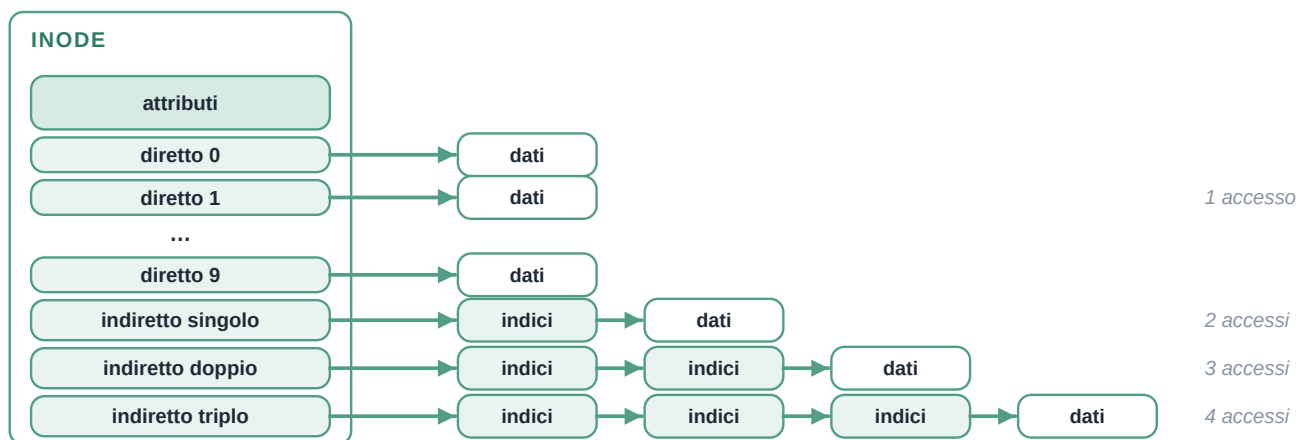
23.4 Gli inode

L'ultima soluzione, usata dai sistemi UNIX, associa a ogni file una piccola struttura, l'**inode** (*index node*, nodo indice), che contiene gli **attributi** del file e gli **indirizzi dei suoi blocchi** sul disco: è l'**allocazione indicizzata**. Ogni inode è identificato da un numero, e gli inode di tutti i file sono raccolti in un'area del disco. Il grande vantaggio rispetto alla FAT è che in memoria serve solo l'inode dei **file aperti**: se un inode occupa n byte e i file aperti sono al massimo k, servono $k \times n$ byte, indipendentemente dalla dimensione del disco.

DEFINIZIONE

L'**inode** (*index node*) è la struttura associata a ogni file nei sistemi UNIX: contiene gli attributi del file e gli indirizzi dei suoi blocchi sul disco, con dieci puntatori diretti e tre puntatori indiretti (singolo, doppio e triplo). Ogni inode è identificato da un numero, e le voci di directory associano ai nomi dei file i numeri dei loro inode.

Il problema è che un inode ha un numero fisso di voci, mentre i file possono essere grandissimi. Il blocco indice si può estendere in più modi, detti **varianti**: nella variante **collegata** i blocchi indice di un file sono collegati in una lista, e l'ultima voce di ognuno punta al successivo; nella variante **multilivello** un blocco indice di primo livello punta a blocchi indice di secondo livello, che puntano ai dati, come nelle tabelle delle pagine multilivello; la variante **ibrida** è quella dell'inode UNIX. Nell'inode classico ci sono **13 voci**: le prime **10** sono **indirizzi diretti** dei primi blocchi del file; l'undicesima punta a un **blocco indiretto singolo**, cioè un blocco che contiene solo indirizzi di blocchi di dati; la dodicesima a un **blocco indiretto doppio**, che contiene indirizzi di blocchi indiretti singoli; la tredicesima a un **blocco indiretto triplo**. I file piccoli, che sono la maggioranza, usano solo gli indirizzi diretti; solo i file grandi pagano i livelli di indirezione.



La struttura di un inode UNIX, con gli accessi al disco necessari per raggiungere un blocco di dati quando l'inode è già in memoria.

Con blocchi da 4 KB e indirizzi di 32 bit (4 byte), un blocco indiretto contiene $4096 / 4 = 1024$ indirizzi. La dimensione massima di un file è quindi $(10 + 1024 + 1024^2 + 1024^3)$ blocchi $\times$ 4 KB, circa **4 TB**. Il numero di accessi per raggiungere un blocco qualsiasi resta piccolo: con l'inode già in memoria basta **un accesso** per un blocco diretto, mentre nel caso peggiore, un blocco raggiunto tramite l'indiretto triplo, servono **quattro accessi** (i tre blocchi indice e il blocco di dati); se anche l'inode va letto dal disco, gli accessi diventano **cinque**. Mentre la capacità cresce di un fattore 1024 a ogni livello, il numero di accessi cresce di uno solo: si dice che gli inode **scalano in modo logaritmico**.

Per trovare il blocco che contiene un certo offset si divide l'offset per la dimensione del blocco, ottenendo l'**indice del blocco nel file**, e si guarda in quale zona cade: con 10 diretti, gli indici da 0 a 9 sono diretti, da 10 a $10 + 1023$ passano per l'indiretto singolo, i successivi per il doppio, e così via, sottraendo di volta in volta i blocchi coperti dalle zone precedenti.

DA RICORDARE

Perché l'inode conviene rispetto alla FAT, e a cosa servono gli indiretti singolo, doppio e triplo?

Perché in memoria serve solo l'inode dei file **aperti**, mentre la FAT deve stare **tutta** in RAM, anche se il disco contiene milioni di file non usati, e cresce con il disco. Inoltre con l'inode l'accesso a un blocco arbitrario è quasi diretto, mentre con la FAT bisogna scorrere la catena dall'inizio. Gli indiretti permettono di gestire file molto grandi con un inode di dimensione fissa: i file piccoli usano solo gli indirizzi diretti, quelli grandi i blocchi indice, e ogni livello moltiplica la capacità per il numero di indirizzi contenuti in un blocco (ad esempio 1024).

Che cosa vuol dire che con l'inode si scala «in modo logaritmico», con al più tre passaggi in più?

Che quando la dimensione massima del file cresce enormemente, passando dai diretti all'indiretto singolo, doppio e triplo, il numero di accessi per raggiungere un blocco qualsiasi cresce solo di **uno per livello**: la capacità cresce in modo esponenziale (moltiplicata per 1024 a ogni livello), il costo in modo lineare nel numero di livelli. Anche per il file più grande bastano tre accessi ai blocchi indice più uno al blocco di dati, quattro in tutto con l'inode già in memoria.

ESEMPI SVOLTI: DIMENSIONE MASSIMA CON GLI INODE

Si calcola prima quanti indirizzi entrano in un blocco (dimensione del blocco / dimensione di un indirizzo, arrotondata per difetto), poi si sommano i blocchi raggiungibili da ogni tipo di voce: un blocco per ogni diretto, p per l'indiretto singolo, p^2 per il doppio, p^3 per il triplo. La capacità massima del disco dipende invece solo da quanti blocchi si possono numerare con un indirizzo.

CONFIGURAZIONE	INDIRIZZI PER BLOCCO (P)	DIMENSIONE MASSIMA DEL FILE	CAPACITÀ MASSIMA DEL DISCO
12 diretti + singolo + doppio; blocchi da 1 KB, indirizzi di 32 bit	$1024 / 4 = 256$	$12 + 256 + 256^2 = 65\,804$ blocchi = 65 804 KB	2^{32} blocchi
10 diretti + 3 indiretti; blocchi da 1 KB, indirizzi di 4 byte	256	$10 + 256 + 256^2 + 256^3 = 16\,843\,018$ blocchi ≈ 16 GiB	$2^{32} \times 1 \text{ KB} = 4 \text{ TiB}$
8 diretti + singolo + doppio; blocchi da 1 KB, indirizzi di 3 byte	$1024 / 3 = 341$	$8 + 341 + 341^2 = 116\,630$ blocchi ≈ 114 MiB	$2^{24} \times 1 \text{ KB} = 16 \text{ GiB}$
12 diretti + singolo, doppio, triplo; indirizzi di 4 byte; blocchi da 512 byte	128	$12 + 128 + 128^2 + 128^3 = 2\,113\,676$ blocchi ≈ 1 GiB	2^{32} blocchi
come sopra, blocchi da 1024 byte	256	$16\,843\,020$ blocchi ≈ 16 GiB	2^{32} blocchi
come sopra, blocchi da 4096 byte	1024	$1\,074\,791\,436$ blocchi ≈ 4 TiB	2^{32} blocchi

Con indirizzi di 3 byte il blocco da 1024 byte contiene solo 341 indirizzi interi: l'ultimo byte avanza. Si nota anche quanto pesa la dimensione del blocco: quadruplicandola, il file massimo cresce di circa mille volte, perché ogni livello di indirizzione contiene quattro volte più indirizzi.

ESEMPIO SVOLTO: INDIVIDUARE I BLOCCHI TRAMITE L'INODE

L'inode 54 ha 13 voci: 10 dirette (321, 322, 239, 234, 235, 236, 14, 21, 233, 12), poi l'indiretto singolo 112, il doppio 333 e il triplo -1 (non usato). Le prime parole di alcuni blocchi sono: blocco **112**: 16, 544, 20, 555...; blocco **333**: 233, 322, 444, 530...; blocco **233**: 821, 822, 915, 50... I blocchi sono da 4 KB e gli indirizzi di 32 bit, quindi ogni blocco indice contiene 1024 indirizzi. In quali blocchi si trovano gli offset 4096, 11 000 e 44 321?

- $4096 / 4096 = 1$: blocco del file di indice 1, cioè il secondo diretto: **322**.
- $11\,000 / 4096 = 2$ (resto 2808): indice 2, il terzo diretto: **239**.

- **44 321** / 4096 = 10 (resto 3361): l'indice 10 supera i 10 diretti (indici da 0 a 9), quindi è il primo blocco raggiunto dall'indiretto singolo: la prima parola del blocco 112, cioè **16**.

Per esercizio si può spingere il ragionamento all'indiretto doppio: gli indici da 10 a 1033 passano per l'indiretto singolo, quindi il primo blocco del doppio è quello di indice 1034, all'offset $1034 \times 4096 = 4\,235\,264$. Si legge il blocco 333 (indiretto doppio), la sua prima parola è 233, si legge il blocco 233, la sua prima parola è 821: il byte si trova nel blocco **821**, dopo tre letture di blocchi indice (inode a parte).

ESEMPIO SVOLTO: ALLOCAZIONE INDICIZZATA A DUE LIVELLI

File system con indici a due livelli, blocchi da 512 byte e indirizzi da 4 byte. Un blocco indice contiene $512 / 4 = 128$ indirizzi: il blocco indice di primo livello punta a fino a 128 blocchi indice di secondo livello, ognuno dei quali punta a 128 blocchi di dati.

- **File di 1 MB**: servono $2^{20} / 2^9 = 2048$ blocchi di dati. Ogni blocco di secondo livello ne indirizza 128, quindi servono $2048 / 128 = 16$ blocchi di secondo livello, più il blocco di primo livello: 17 blocchi indice, **2065 blocchi** in tutto.
- **Il 400° blocco** del file ha indice 399 (contando da 0): $399 / 128 = 3$ con resto 15, quindi si segue la **voce 3** del blocco di primo livello e la **voce 15** del blocco di secondo livello così trovato.
- **Il byte 236 448**: $236\,448 / 512 = 461$ con resto 416, quindi è nel blocco di indice 461, all'offset 416; $461 / 128 = 3$ con resto 77: **voce 3** del primo livello, **voce 77** del secondo livello, byte 416 del blocco di dati.
- **Dimensione massima**: $128 \times 128 = 16\,384$ blocchi di dati $\times 512$ byte = **8 MiB**. Un file di dimensione massima occupa complessivamente $16\,384 + 128 + 1 = 16\,513$ blocchi, contando anche i blocchi indice.

DA RICORDARE

Con un inode completo (diretti, indiretto singolo, doppio e triplo), qual è il numero massimo di blocchi da leggere su disco per portare in memoria un blocco qualsiasi del file, con il record della directory già in memoria?

Cinque: un accesso per leggere l'**inode** (la directory dà solo il suo numero), tre accessi per i blocchi indice del **triplo indiretto** (primo, secondo e terzo livello) e un accesso per il **blocco di dati**. Se anche l'inode fosse già in memoria, gli accessi sarebbero quattro.

Fra queste affermazioni sulle tecniche di allocazione dei file, qual è falsa?

Sono vere: nell'allocazione contigua bisogna conoscere a priori la dimensione massima del file; l'allocazione concatenata soffre di frammentazione interna; l'allocazione contigua richiede meno RAM e meno accessi al disco per trovare il blocco che contiene un dato qualsiasi; con la FAT non serve una bitmap separata per i blocchi liberi. È **falsa** quella secondo cui nella FAT «la capacità del singolo blocco può essere solo parzialmente sfruttata, dovendo memorizzare il numero del blocco successivo»: questo vale per le liste collegate, mentre la FAT ha tolto proprio i puntatori dai blocchi.

Le directory e la condivisione dei file

Come si realizzano le directory e come un file può comparire in più directory, con gli hard link e i soft link.

24.1 L'implementazione delle directory

Quando un file viene aperto, il sistema usa il **percorso** (*path name*) per trovare la voce corrispondente nella directory, e da lì le informazioni per trovare i blocchi del file: l'indirizzo del primo blocco con la FAT, il numero di inode con gli inode. Una questione è **dove memorizzare i metadati**. Nella soluzione della **FAT** la directory è una lista di voci di dimensione fissa, ognuna con il **nome**, gli **attributi** e il **primo blocco** del file. Nei file system con **inode** gli attributi stanno nell'inode, e ogni voce di directory contiene soltanto il **nome** e il **numero di inode**: è proprio questo che rende possibili gli hard link.

Per trovare un file a partire dal percorso, ad esempio `/usr/ast/mbox`, il sistema parte dall'inode della **directory radice**, che si trova in una posizione nota del disco; legge la radice e vi cerca `usr`, ottenendone il numero di inode; legge quell'inode per sapere dove si trova il contenuto di `usr`, lo legge e vi cerca `ast`; e così via fino al file finale. Ogni componente del percorso costa quindi la lettura di un inode e di una directory.

ESEMPIO SVOLTO: ACCESSI AL DISCO PER APRIRE UN FILE

In un file system in cui ogni inode e ogni directory occupano un blocco, quanti accessi al disco servono per aprire `/usr/carlo/documenti/compito`, se l'inode della radice è già in memoria e tutto il resto è sul disco?

Per ogni componente del percorso bisogna leggere la directory che lo contiene e poi il suo inode: directory radice (1), inode di `usr` (2), directory `usr` (3), inode di `carlo` (4), directory `carlo` (5), inode di `documenti` (6), directory `documenti` (7), inode di `compito` (8). Aprire il file significa avere in memoria il suo inode, quindi servono **8 accessi**. Per leggerne i dati servirebbero poi altri accessi, uno per ogni blocco.

I file system moderni ammettono **nomi lunghi** e di lunghezza variabile, anche di centinaia di caratteri, e riservare a ogni voce lo spazio per il nome più lungo sarebbe uno spreco. Una prima soluzione sono le voci di **lunghezza variabile**: ogni voce inizia con un campo che ne indica la lunghezza, seguito dagli attributi (di dimensione fissa) e dal nome, terminato da un carattere speciale e allineato alla parola. Quando un file viene cancellato resta però un buco di dimensione variabile, cioè frammentazione esterna dentro la directory, che si può risolvere compattandola, operazione accettabile perché una directory è piccola. Una soluzione migliore è separare le parti: le voci hanno **dimensione fissa** e contengono, al posto del nome, un **riferimento** alla sua posizione in uno **heap** in fondo alla directory, dove i nomi sono memorizzati uno dopo l'altro; un buco fra le voci fisse è sempre riutilizzabile.

Per le **prestazioni**, nelle directory molto grandi la ricerca lineare di un nome è lenta. Si può usare una **tabella hash**, indicizzata sul nome, che porta direttamente alla voce; oppure, più semplicemente, una **cache** delle ricerche recenti, perché gli stessi percorsi vengono cercati spesso. Per directory di dimensioni normali la differenza è piccola, perché una volta letta in memoria la directory si scorre rapidamente.

24.2 La condivisione di un file: hard link e soft link

Capita che due utenti vogliano condividere un file, che deve comparire in due directory diverse, senza essere duplicato. Con la **FAT** si potrebbe **duplicare la lista dei riferimenti** ai blocchi, cioè copiare nella seconda directory la voce con il primo blocco; ma se uno dei due utenti **aggiunge dati in fondo** al file (*append*), i nuovi blocchi vengono registrati solo nella sua voce, e l'altro non li vede.

Con gli **inode** la soluzione è naturale: entrambe le directory contengono una voce con lo **stesso numero di inode**. È l'**hard link**: i due nomi sono del tutto equivalenti, nessuno dei due è «l'originale», e le modifiche fatte attraverso uno si vedono attraverso l'altro. L'inode contiene un **contatore dei link**, che conta quante voci di directory puntano a lui: creare un link lo incrementa, cancellare un nome (con la system call `unlink`) lo decrementa, e l'inode e i blocchi vengono liberati **solo quando il contatore arriva a 0**. C'è però un'**anomalia con l'accounting**: se l'utente A, proprietario del file, lo cancella mentre B ha un hard link, il file resta in vita, ma il proprietario registrato nell'inode è ancora A, al quale continua a essere addebitato lo spazio.

L'alternativa è il **soft link** (o **link simbolico**): un file speciale che contiene il **percorso** del file a cui si riferisce. Quando si accede al soft link, il sistema legge il percorso e accede al file indicato. È **universale**: permette di fare riferimento anche a **oggetti fuori dal file system**, su un'altra partizione o su un'altra macchina in rete, e anche a directory, perché un percorso ha senso ovunque, mentre un numero di inode vale solo all'interno del suo file system. In cambio **appesantisce la gestione**: ogni accesso richiede di leggere il link e poi ripercorrere il percorso, con più accessi al disco, e il soft link occupa un proprio inode. Inoltre, se il file viene cancellato o spostato, il soft link resta, ma punta a qualcosa che non esiste più (*dangling link*).

Entrambi i tipi di link possono creare **problemi negli attraversamenti** del file system e nei **backup**: un programma che visita l'albero può trovare lo stesso file più volte, o copiarlo due volte nel backup. Con i soft link il problema è gestibile, perché il sistema li riconosce come tali e può decidere di non seguirli; per questo gli hard link alle directory sono vietati agli utenti.

DA RICORDARE

Come sono realizzati hard link e soft link, e perché il soft link funziona anche fra file system diversi?

L'**hard link** è una voce di directory che contiene un nome e il **numero di un inode già esistente**: più nomi puntano allo stesso inode, e un contatore nell'inode conta quanti sono. Funziona solo **all'interno dello stesso file system**, perché la numerazione degli inode è locale a ciascun file system. Il **soft link** è un file speciale, con un proprio inode, che contiene il **percorso testuale** del file a cui si riferisce: poiché non dipende da un numero di inode, può puntare a file su altri file system o su altre macchine.

Qual è il problema principale del soft link, e quale quello dell'hard link? Che cosa conta il contatore?

Il soft link non è legato al file a livello di inode: se il file viene cancellato o spostato, il link resta e punta al vuoto, ed è anche più lento, perché richiede di leggere il link e poi risolvere il percorso. L'hard link funziona solo nello stesso file system. Il suo contatore (*link count*), memorizzato nell'inode, conta quanti nomi di directory sono associati a quell'inode: il file esiste finché il contatore è maggiore di zero, e viene eliminato solo quando si cancella l'ultimo nome.

Perché gli hard link non possono puntare a directory, mentre i soft link sì, e perché un ciclo di soft link non è un problema irrisolvibile?

Con un hard link a una directory si potrebbe creare un **ciclo** nell'albero, ad esempio una sottodirectory che punta a una directory più in alto, e poiché un hard link è indistinguibile da un nome normale, i programmi che attraversano il file system (copie ricorsive, backup, ricerche) non saprebbero di girare in tondo e andrebbero in loop. Il soft link invece è un **tipo di file riconoscibile**: chi attraversa l'albero può decidere di non seguirlo, e il sistema può contare quanti soft link ha seguito nella risoluzione di un percorso e fermarsi oltre un limite, segnalando un errore.

Spazio libero, affidabilità e prestazioni

La gestione dello spazio libero, i controlli di consistenza, il journaling e la cache del disco, con uno sguardo ai file system attuali.

25.1 La gestione dei blocchi liberi

Il file system deve sapere quali blocchi sono liberi, per creare nuovi file o allungare quelli esistenti. I metodi sono gli stessi visti per la memoria. Con una **bitmap** ogni blocco del disco corrisponde a un bit; la bitmap è **relativamente piccola** (con blocchi da 4 KB, un disco da 1 TB ne richiede 32 MB) e ha dimensione fissa. Per usarla in modo efficiente bisogna averla in memoria, con diverse **strategie**: tenerla **tutta in memoria**, oppure **un blocco alla volta**, caricando un altro blocco della bitmap quando quello corrente è esaurito; oppure lasciarla **paginata dalla memoria virtuale**, che terrà in RAM le parti usate. Più i blocchi del disco sono grandi, più la bitmap è piccola, ma più cresce la frammentazione interna.

In alternativa si usa una **lista concatenata dei blocchi liberi**. Richiede **più spazio** della bitmap, perché per ogni blocco libero bisogna scrivere un intero numero di blocco invece di un bit, ma lo spazio usato è quello dei **blocchi liberi stessi**: un blocco libero contiene i numeri di altri blocchi liberi e il puntatore al blocco successivo della lista, quindi la lista non sottrae spazio utile, e si accorcia man mano che il disco si riempie. Si possono anche inserire dei **contatori per i blocchi contigui**: invece di elencare uno per uno i blocchi da 100 a 150, si registra «blocco 100, seguito da 50 blocchi liberi». Un disco appena formattato è descritto da una sola voce. Nella pratica la bitmap è più usata, per la sua semplicità: i file system con inode usano una bitmap per i blocchi e un'altra per gli inode liberi, mentre la FAT, come visto, non ha bisogno di nessuna struttura aggiuntiva.

ESEMPIO SVOLTO: EVOLUZIONE DI UNA BITMAP

Dopo la formattazione la bitmap (1 = occupato) vale **1000 0000 0000 0000**: il blocco 0 contiene la directory radice. Il sistema alloca sempre il blocco libero con l'indice più piccolo. Operazioni: crea A di 6 blocchi, crea B di 5 blocchi, cancella A, crea C di 8 blocchi, cancella B.

- A occupa i blocchi da 1 a 6: **1111 1110 0000 0000**.
- B occupa i blocchi da 7 a 11: **1111 1111 1111 0000**.
- La cancellazione di A libera da 1 a 6: **1000 0001 1111 0000**.
- C prende i primi otto liberi, cioè da 1 a 6, 12 e 13: **1111 1111 1111 1100**. Il file C non è contiguo, ma con FAT o inode non è un problema.
- La cancellazione di B libera da 7 a 11: stato finale **1111 1110 0000 1100**.

25.2 I controlli di consistenza

Le strutture di un file system contengono informazioni **ridondanti**: lo stesso blocco compare negli inode (o nella FAT) del file che lo usa e, se è libero, nella bitmap. Un **crash** durante una serie di modifiche, ad esempio un'interruzione di corrente fra l'aggiornamento di un inode e quello della bitmap, lascia queste strutture in uno stato **inconsistente**, e continuare a lavorare in quelle condizioni è pericoloso: un blocco in uso ma marcato libero verrebbe assegnato a un altro file, distruggendo il primo. Per questo all'avvio, se il file system non è stato smontato correttamente (un flag sul disco lo segnala), si eseguono apposite utility di **controllo della consistenza**, come fsck nei sistemi UNIX. I controlli sono di due tipi.

Il **controllo sui blocchi** costruisce due tabelle con un contatore per ogni blocco: la prima conta quante volte il blocco compare **in uso** nei file (scorrendo tutti gli inode), la seconda quante volte compare **fra i blocchi liberi**

(nella bitmap o nella lista). In un file system consistente ogni blocco ha un 1 in **esattamente una** delle due tabelle. Le anomalie e le correzioni sono:

SITUAZIONE	IN USO	LIBERO	CORREZIONE
Consistente	1	0	Nessuna
Consistente	0	1	Nessuna
Blocco mancante	0	0	Il blocco non è di nessuno ma non è disponibile: si aggiunge ai liberi . È solo spazio perso, non un danno
In uso e libero	1	1	Il blocco appartiene a un file, quindi si toglie dai liberi , prima che venga assegnato a un altro
Libero duplicato	0	2 o più	Può accadere solo con la lista: si corregge la lista in modo che il blocco compaia una volta
In uso duplicato	2 o più	0	Due file condividono per errore lo stesso blocco: si copia il blocco in un blocco libero e si assegna la copia a uno dei due file, segnalando all'utente che il contenuto potrebbe essere danneggiato

Il **controllo sui riferimenti agli inode** scorre tutto l'albero delle directory e conta, per ogni inode, quante voci lo nominano; il valore va confrontato con il contatore dei link scritto nell'inode. Se il contatore è **più alto** del conteggio reale, il file non verrà mai liberato, nemmeno quando saranno stati cancellati tutti i nomi: è solo spazio sprecato. Se è **più basso**, è grave: cancellando un nome il contatore arriverebbe a zero mentre esiste ancora un riferimento, e i blocchi liberati potrebbero finire a un altro file. In entrambi i casi si **corregge il contatore** con il valore calcolato.

ESEMPI SVOLTI: CORREGGERE I VETTORI DI FSCK

Esercizio 1. Blocchi da 0 a 15, con i vettori «in uso» 1 1 0 1 0 0 0 1 1 0 0 1 0 0 0 1 e «liberi» 0 0 1 0 0 0 1 0 0 1 1 0 1 2 1 0. Si controlla colonna per colonna: i blocchi **4 e 5** hanno 0 in entrambi i vettori, quindi sono blocchi mancanti e si **aggiungono ai liberi**; il blocco **13** compare due volte fra i liberi, quindi si **corregge la lista** in modo che il suo contatore valga 1. Tutti gli altri blocchi hanno un solo 1 e sono a posto.

Esercizio 2. Blocchi da 0 a 17, con «liberi» 1 1 1 1 0 0 0 1 1 1 0 0 0 0 2 2 1 e «in uso» 1 2 1 1 0 0 1 1 1 0 0 0 0 0 0 1 1 2.

- I blocchi **0, 1, 2, 3, 7, 8, 15, 16 e 17** risultano sia in uso sia liberi: si **tolgono dai liberi** (anche dove il contatore dei liberi vale 2).
- I blocchi **4, 5, 10, 11, 12, 13 e 14** non compaiono in nessun vettore: si **aggiungono ai liberi**.
- I blocchi **1 e 17** sono usati da due file: fsck ne fa una **copia** in due blocchi liberi (i più bassi disponibili, 4 e 5), assegna le copie a uno dei due file e avverte l'utente del possibile danno.

Alla fine i blocchi liberi sono 9, 10, 11, 12, 13 e 14, e tutti gli altri risultano in uso una sola volta.

DA RICORDARE

Come si verifica la consistenza fra la mappa dei blocchi liberi e l'allocazione effettiva dei file?

Con una scansione preliminare che produce **due vettori** di contatori: per ogni blocco, quante volte compare fra i blocchi in uso nei file e quante volte fra quelli liberi. In un file system consistente ogni blocco compare esattamente una volta in uno dei due. Le correzioni sono: blocco **sia in uso sia libero**, lo si toglie dai liberi; blocco **in nessuno dei due**, lo si aggiunge ai liberi; blocco **libero più volte**, si corregge la lista; blocco **in uso più volte**, si copia il blocco e si assegna la copia a uno dei file, segnalando l'errore. Un secondo controllo confronta il contatore dei link di ogni inode con il numero di voci di directory che lo nominano.

25.3 Il journaling

Su dischi grandi un controllo di consistenza completo richiede molto tempo. Il **journaling** affronta il problema in modo preventivo. Le operazioni sui **metadati** (allocare un blocco, aggiornare un inode, aggiungere una voce di directory) vengono **preliminarmente annotate in un log**, il **journal**, che risiede sul disco; poi vengono eseguite, e quando sono state completate la loro registrazione viene **cancellata** dal log, subito dopo. Se avviene un crash, al riavvio il sistema guarda il journal: se è vuoto non c'era nulla in corso; se contiene delle operazioni, queste potrebbero essere state eseguite in parte, e il sistema le **ripete**. I benefici sono una **maggiore robustezza dei metadati** e un **ripristino veloce** dopo un crash o un riavvio, perché basta rieseguire le poche operazioni del journal invece di controllare tutto il disco.

DEFINIZIONE

Il **journaling** è la tecnica con cui un file system annota in un registro sul disco, il **journal**, le operazioni sui metadati prima di eseguirle, e le cancella dal registro dopo averle completate. Dopo un crash basta ripetere le operazioni rimaste nel journal per riportare il file system in uno stato coerente, senza un controllo completo del disco.

Perché tutto funzioni le operazioni annotate devono essere **idempotenti**, cioè ripeterle più volte deve dare lo stesso risultato che eseguirle una volta sola: il sistema non sa a che punto fosse arrivato, e potrebbe rieseguire operazioni già completate (o subire un secondo crash durante il ripristino). «Segna il blocco 25 come occupato» è idempotente; «incrementa il contatore dei link» non lo è. Di solito il journal registra **solo i metadati** e non i dati dei file: annotare ogni scrittura di dati raddoppierebbe il lavoro del disco, e la priorità è che la **struttura** del file system resti integra. Usano il journaling, ad esempio, NTFS ed ext4; non lo usano i file system più vecchi come FAT ed ext2.

DA RICORDARE

Che cos'è il journaling, e perché le operazioni registrate devono essere idempotenti?

È una tecnica in cui il file system annota nel **journal**, un'area dedicata del disco, le operazioni sui metadati **prima** di eseguirle, e le cancella dal journal dopo averle completate. Dopo un crash, al riavvio, le operazioni rimaste nel journal vengono **rieseguite**, riportando rapidamente il file system in uno stato consistente senza scansionare tutto il disco. Devono essere **idempotenti** perché il sistema non sa se, al momento del crash, un'operazione era già stata applicata del tutto, in parte o per niente, e potrebbe anche subire un nuovo crash durante il ripristino: rieseguire un'operazione idempotente produce lo stesso risultato finale, senza introdurre nuovi errori.

Sul journaling, quale affermazione è errata?

È **errata** l'affermazione che «il journal tiene nota delle ultime operazioni completate con successo»: il journal registra le operazioni **da eseguire o in corso**, e le cancella proprio quando sono completate. Sono invece vere: il journal permette di correggere velocemente le incongruenze dopo un crash; se non ci sono operazioni pendenti e il file system è consistente, il journal è vuoto; le operazioni annotate devono essere idempotenti; i file system di MS DOS ed ext2 non supportano il journaling.

25.4 La cache del disco

Per migliorare le prestazioni si usa una **cache del disco** (o **buffer cache**): una parte della RAM conserva i blocchi letti di recente, così che una nuova richiesta dello stesso blocco non richieda di accedere al disco. La struttura è basata su una **tabella hash**, indicizzata sul numero di dispositivo e di blocco, per sapere rapidamente se un blocco è in cache, e su una **lista doppiamente collegata** che tiene i blocchi in ordine di utilizzo, dal meno al più usato di recente. A differenza della memoria virtuale, qui un **LRU** vero si può realizzare, perché i riferimenti ai blocchi sono relativamente rari e passano tutti dal sistema operativo.

Le strategie di gestione sono più raffinate di un LRU puro. Con un **LRU modificato** si tiene conto della natura dei blocchi: un blocco che difficilmente servirà di nuovo a breve (ad esempio un blocco di dati appena scritto per intero) viene messo subito fra i candidati alla rimozione, mentre i blocchi **critici** per la consistenza, come inode e directory, vengono riscritti sul disco **appena modificati**, senza aspettare che escano dalla cache. Con le letture **sequenziali** si usano due ottimizzazioni: il **free behind**, che toglie dalla cache un blocco appena il processo passa al successivo, perché in una lettura sequenziale non verrà riletto; e il **read ahead**, che carica in anticipo i blocchi successivi a quello richiesto, perché probabilmente serviranno a breve. Su un disco meccanico leggere qualche blocco in più costa poco, perché il grosso del tempo è il posizionamento della testina.

Per le **scritture** ci sono due possibilità. Con la scrittura **sincrona** (*write through*) ogni modifica viene subito riportata sul disco: è sicura, ma lenta. Con la scrittura **asincrona** (*write back*) la modifica resta in cache e viene scritta più tardi: è più veloce, perché scritture ripetute dello stesso blocco si accorpano e il sistema può ordinarle in base alla posizione della testina, ma in caso di crash le modifiche non ancora scritte vanno perse. Per questo i sistemi UNIX scrivono comunque sul disco i blocchi modificati a intervalli regolari (ad esempio ogni 30 secondi), e mettono sempre in scrittura sincrona i metadati.

25.5 I file system dei sistemi attuali

Windows usa **exFAT** sulle unità rimovibili e **NTFS** sui dischi fissi: un file system moderno e molto complesso, con journaling, cifratura, compressione, **copie shadow** dei volumi (quando un blocco viene modificato se ne conserva prima la versione originale, con un meccanismo di **copy on write**) e supporto a dischi multipli (RAID). **Linux** usa di norma **ext4**, con journaling e allocazione efficiente, e supporta **Btrfs**, un file system evoluto che al posto del journaling usa il **copy on write** per tutti gli aggiornamenti, con **checksum** sui dati e sui metadati (per accorgersi di corruzioni silenziose), compressione, gestione di volumi, snapshot, clonazione dei file e dischi multipli. **macOS** usava **HFS+**, sostituito dal 2017 da **APFS**.

Lo scheduling del disco

In quale ordine servire le richieste di accesso al disco per ridurre i movimenti della testina: gli algoritmi e il loro confronto.

26.1 Obiettivi e tempi di accesso

In un sistema multiprogrammato si formano molte richieste di I/O sul disco, ma il controller ne può ricevere **una alla volta**: si crea una **coda di richieste pendenti**, e il sistema operativo può scegliere in che ordine inviarle. Gli obiettivi sono **massimizzare il throughput**, cioè il numero di richieste soddisfatte nell'unità di tempo, e **minimizzare il tempo medio di accesso**. Su un disco magnetico il tempo per servire una richiesta è la somma di tre componenti: il **tempo di posizionamento** (*seek time*), per spostare il braccio sul cilindro giusto, che è la componente dominante; la **latenza di rotazione**, l'attesa che il settore giusto passi sotto la testina; e il **tempo di trasferimento**, quasi costante. Si può quindi ottimizzare soprattutto il seek time, e in secondo luogo la latenza di rotazione, servendo insieme le richieste che cadono sullo stesso cilindro. Negli esercizi conta solo la **distanza percorsa** dalla testina, misurata in tracce (o cilindri): con un tempo di 1 ms per traccia, la distanza in tracce coincide con il tempo in millisecondi.

26.2 Gli algoritmi

Esempio: la coda contiene, in ordine di arrivo, le richieste per i cilindri **98, 183, 37, 122, 14, 124, 65, 67**, e la testina parte dal cilindro **53**.

- **FCFS** (*First Come First Served*): le richieste vengono servite in ordine di arrivo. Distanza totale: $|53 - 98| + |98 - 183| + |183 - 37| + |37 - 122| + |122 - 14| + |14 - 124| + |124 - 65| + |65 - 67| = 45 + 85 + 146 + 85 + 108 + 110 + 59 + 2 = \mathbf{640 \text{ tracce}}$. È semplice da realizzare ed **equo**, ma inefficiente: la testina va a zigzag da un capo all'altro del disco.
- **SSTF** (*Shortest Seek Time First*): si serve sempre la richiesta **più vicina** alla posizione attuale della testina. Ordine 65, 67, 37, 14, 98, 122, 124, 183, distanza totale **236 tracce**. Le prestazioni sono buone, ma l'algoritmo **non è equo**: una richiesta lontana può attendere all'infinito se continuano ad arrivare richieste vicine alla testina (**starvation**). Come SJF nello scheduling della CPU, non è comunque ottimale.
- **Scheduling per scansione**, detto **algoritmo dell'ascensore**: la testina **mantiene un verso** finché ci sono richieste in quella direzione, servendole man mano, e poi inverte il senso, proprio come un ascensore che non cambia direzione finché ha piani da servire. Partendo verso l'alto l'ordine è 65, 67, 98, 122, 124, 183, poi 37, 14: distanza totale $130 + 169 = \mathbf{299 \text{ tracce}}$, più di SSTF. In compenso la scansione è **uniforme** e garantisce comunque un'**attesa massima** per ogni richiesta: nel caso peggiore, una richiesta appena superata dalla testina aspetta che questa vada fino in fondo e torni indietro.
- **Scansione circolare**: le posizioni vengono considerate **collegate in modo circolare**: arrivata alla fine, la testina torna sul primo cilindro **senza servire alcuna richiesta** durante il ritorno, e riprende a servirle nello stesso verso. L'ordine è 65, 67, 98, 122, 124, 183, poi 14, 37. Contando anche il ritorno da 183 a 14, la distanza è $130 + 169 + 23 = \mathbf{322 \text{ tracce}}$. Il vantaggio è un **tempo di attesa più uniforme** e, in presenza di molte richieste, più basso in media: con la scansione semplice le richieste agli estremi del disco aspettano mediamente più di quelle al centro, mentre con quella circolare tutte vengono trattate allo stesso modo.

Nello scheduling per scansione si distinguono due varianti: quella descritta qui, in cui la testina inverte il senso (o torna indietro) appena **servita l'ultima richiesta** in quella direzione, detta anche **LOOK** (e **C LOOK** la versione circolare), e quella in cui la testina arriva comunque **fino all'ultimo cilindro** del disco prima di invertire, detta **SCAN** in senso stretto (e **C SCAN** quella circolare). La prima evita di percorrere tratti senza richieste ed è quella

usata in pratica. **Cosa scegliere?** Ad **alto carico** conviene la **scansione circolare**; a **basso carico** la **scansione** o **SSTF**.

ESEMPIO SVOLTO: TUTTI GLI ALGORITMI SULLA STESSA CODA

Disco con cilindri da 0 a 199, testina sul cilindro **50** in movimento verso l'alto; coda: **82, 170, 43, 140, 24, 16, 190**. Calcolare ordine e distanza totale con ogni algoritmo.

ALGORITMO	ORDINE DI SERVIZIO	DISTANZA TOTALE
FCFS	82, 170, 43, 140, 24, 16, 190	$32 + 88 + 127 + 97 + 116 + 8 + 174 = \mathbf{642}$
SSTF	43, 24, 16, 82, 140, 170, 190	$7 + 19 + 8 + 66 + 58 + 30 + 20 = \mathbf{208}$
Ascensore (LOOK)	82, 140, 170, 190, poi 43, 24, 16	$(190 - 50) + (190 - 16) = \mathbf{314}$
Circolare (C LOOK)	82, 140, 170, 190, poi 16, 24, 43	$(190 - 50) + (190 - 16) + (43 - 16) = \mathbf{341}$
SCAN fino al bordo	82, 140, 170, 190, 199, poi 43, 24, 16	$(199 - 50) + (199 - 16) = \mathbf{332}$
C SCAN fino al bordo	82, 140, 170, 190, 199, 0, poi 16, 24, 43	$(199 - 50) + (199 - 0) + (43 - 0) = \mathbf{391}$

Nelle versioni circolari la distanza include il tratto di ritorno, in cui la testina si sposta senza servire richieste. SSTF qui è il migliore, ma serve per ultime le richieste più lontane dalla zona di partenza.

ESEMPIO SVOLTO: ASCENSORE CON RICHIESTE CHE ARRIVANO DURANTE IL SERVIZIO

Disco con 200 tracce (da 0 a 199), 1 traccia per ms. A $t = 0$ la testina serve la traccia **100**, e in coda ci sono 50, 115 e 180. Arrivano poi la richiesta **150** a $t = 70$ e la **90** a $t = 130$. Calcolare il tempo di ricerca complessivo con l'algoritmo dell'ascensore, partendo in ordine ascendente.

Il metodo è seguire la testina nel tempo, ricordando che una richiesta può essere servita solo dopo essere arrivata e che l'ascensore serve le richieste che incontra lungo il percorso:

- salendo da 100 serve **115** a $t = 15$ e **180** a $t = 80$. Sopra 180 non ci sono altre richieste (la 150 è arrivata a $t = 70$, ma sta sotto), quindi la testina **inverte il senso**;
- scendendo serve **150** a $t = 110$. In quel momento l'unica richiesta in attesa è la 50: la 90 non è ancora arrivata. La testina scende verso 50 e a $t = 130$, quando arriva la richiesta 90, si trova alla traccia 130: la 90 è lungo il percorso, quindi viene servita a $t = 170$;
- infine serve **50** a $t = 210$.

Sequenza: **100, 115, 180, 150, 90, 50**; spostamenti $15 + 65 + 30 + 60 + 40 = 210$ tracce, cioè un tempo di ricerca complessivo di **210 ms**.

ESEMPIO SVOLTO: TEMPI DI ATTESA CON L'ASCENSORE

Disco con tracce da 0 a 100, testina sul cilindro 40 in salita, 2 ms per spostarsi di una traccia. Arrivano richieste per i cilindri 90, 45, 40, 60 e 55, rispettivamente agli istanti 0, 20, 30, 40 e 80 ms. Il tempo di attesa di una richiesta va dall'arrivo all'istante in cui viene servita. Trovare l'ordine di servizio e il tempo di attesa medio.

La testina parte da 40 verso l'alto con la sola richiesta 90. A $t = 20$ (cilindro 50) arriva la 45, a $t = 30$ (cilindro 55) la 40: entrambe sono alle spalle. A $t = 40$ la testina è proprio sul cilindro 60 quando arriva la richiesta 60, che viene servita subito. Continua fino a **90**, servita a $t = 100$; nel frattempo, a $t = 80$, è arrivata la 55. Con l'algoritmo dell'ascensore, sopra 90 non c'è altro e la testina inverte: serve **55** a $t = 170$, **45** a $t = 190$ e **40** a $t = 200$.

Ordine: **60, 90, 55, 45, 40**. Attese: $60 \rightarrow 0$, $90 \rightarrow 100$, $55 \rightarrow 90$, $45 \rightarrow 170$, $40 \rightarrow 170$ ms; media $(0 + 100 + 90 + 170 + 170) / 5 = \mathbf{106 \text{ ms}}$. Se il testo chiede di arrivare **fino al bordo** del disco (SCAN in senso stretto), la testina sale fino a 100 ($t = 120$) prima di invertire: 55 a $t = 210$, 45 a $t = 230$, 40 a $t = 240$, con attese 0, 100, 130, 210, 210 e media **130 ms**. È un buon esempio del costo di andare fino in fondo senza motivo.

ESEMPIO SVOLTO: SSTF CON ARRIVI NEL TEMPO

Disco con 200 tracce, 1 traccia per ms, testina sulla traccia 140 a $t = 0$. Richieste: 145, 120, 90 e 30 all'istante 0; 175 a $t = 105$; 40 a $t = 110$; 170 a $t = 135$. Usare SSTF.

Con SSTF la prossima richiesta si sceglie **quando quella corrente è stata servita**, fra quelle già arrivate; durante lo spostamento la destinazione non cambia.

- Da 140 la più vicina è **145** (servita a $t = 5$); poi **120** ($t = 30$), **90** ($t = 60$) e **30** ($t = 120$).
- A $t = 120$, sulla traccia 30, sono arrivate la 175 e la 40: la più vicina è **40** ($t = 130$).
- A $t = 130$ l'unica in attesa è la 175 (la 170 arriverà solo a $t = 135$): la testina parte verso **175** e la raggiunge a $t = 265$; infine serve **170** a $t = 270$.

Sequenza **145, 120, 90, 30, 40, 175, 170**; distanza totale $5 + 25 + 30 + 60 + 10 + 135 + 5 = \mathbf{270 \text{ tracce}}$, cioè 270 ms. Attese (istante di servizio meno arrivo): 5, 30, 60, 120, 160 (la 175), 20 (la 40), 135 (la 170), con media $530 / 7 \approx \mathbf{75,7 \text{ ms}}$. La 175, arrivata prima della 40, aspetta molto di più: SSTF non è equo.

DA RICORDARE

Differenze fra FCFS, SSTF e lo scheduling per scansione (ascensore): quale rischia la starvation?

FCFS serve le richieste in ordine di arrivo: è equo, nessuna richiesta va in starvation, ma la testina fa movimenti ampi e casuali. **SSTF** serve sempre la richiesta più vicina alla testina: riduce molto la distanza percorsa, ma **rischia la starvation** delle richieste lontane, se continuano ad arrivarne di vicine. La **scansione** (ascensore) muove la testina in un verso servendo tutte le richieste incontrate e poi inverte: percorre un po' più di SSTF, ma **non c'è starvation**, perché prima o poi la testina passa ovunque, e c'è un limite massimo all'attesa. La variante che si ferma all'ultima richiesta (LOOK) evita di arrivare inutilmente fino al bordo del disco; quella circolare rende i tempi di attesa più uniformi.

I sistemi RAID e gli SSD

Come più dischi possono lavorare insieme per ottenere prestazioni e affidabilità maggiori, e come funzionano i dischi a stato solido.

27.1 L'idea del RAID

Un altro modo per aumentare le prestazioni è sfruttare il **parallelismo** anche per l'I/O su disco. Servono **più dischi indipendenti**, e i dati di un'unità logica (un file o, in generale, un volume) vengono suddivisi su più dischi, in modo **trasparente** all'utente: è lo **striping**. Il volume viene diviso in **strip** (porzioni di dimensione fissa, multipla del blocco, dette anche *stripe unit*) assegnate a dischi diversi; l'insieme delle strip che stanno alla stessa altezza su tutti i dischi forma una **stripe**, cioè una «riga». Se un file grande è distribuito su quattro dischi, i quattro possono leggerne ciascuno un pezzo **contemporaneamente**, e il trasferimento è circa quattro volte più veloce. Il rapporto fra il tempo con un disco e il tempo con più dischi è lo **speedup**.

DEFINIZIONE

Un **RAID** (*Redundant Array of Independent Disks*) è un insieme di dischi che il sistema vede come un'unica unità logica. I dati vengono distribuiti sui dischi con lo **striping**, per leggerli e scriverli in parallelo, e quasi tutti i livelli aggiungono **ridondanza** (copie o informazioni di parità) per sopravvivere al guasto di un disco.

C'è però un **problema**: con più dischi aumenta la probabilità che si verifichi un **guasto** sul volume logico. Se un disco si guasta in media una volta ogni cinque anni, con venti dischi ci si aspetta un guasto ogni tre mesi circa, e con lo striping puro un solo guasto rende inutilizzabile l'intero volume. La soluzione è aggiungere **ridondanza**, per ottenere una migliore **affidabilità**; spesso si aggiungono anche dischi di riserva (*spare*) per la **sostituzione automatica** del disco guasto. Un insieme di dischi organizzato così si chiama **RAID** (*Redundant Array of Inexpensive Disks*, noto anche come *Redundant Array of Independent Disks*). Esistono vari schemi, i **livelli RAID**, che bilanciano in modo diverso prestazioni e affidabilità. Il RAID può essere realizzato **via hardware**, da un controller dedicato che presenta i dischi al sistema operativo come un unico disco, in modo trasparente, oppure **via software**, dal sistema operativo, con un carico aggiuntivo sulla CPU.

27.2 I livelli RAID

RAID 0 (*striping*) distribuisce le strip sui dischi in modalità **Round Robin**: la strip 0 sul disco 0, la 1 sul disco 1, e così via, ricominciando dal primo. È **semplice**, con **prestazioni ottimali** nelle letture di grandi volumi, perché tutti i dischi lavorano in parallelo, ma **non ha alcuna ridondanza**: è il livello più vulnerabile, perché il guasto di un solo disco fa perdere i dati di tutti i file che vi avevano una strip.

RAID 1 (*mirroring*) **duplica** ogni strip su un altro disco: ogni dato esiste in due copie identiche. Può essere usato anche **senza striping**, con un disco e la sua copia. Le **letture** possono essere distribuite fra le due copie, con un **raddoppio delle prestazioni in lettura**; le **scritture** invece vanno fatte su entrambe le copie, quindi non sono più veloci. Ha la **migliore tolleranza ai guasti**, perché se un disco si rompe si usa la copia, ma un **alto overhead di memorizzazione**: metà dello spazio serve per le copie.

RAID 2 (*striping a livello di bit con ECC*) lavora sulle singole parole, distribuendone i bit su dischi diversi e aggiungendo un **codice di correzione degli errori** (ECC), ad esempio un **codice di Hamming** che corregge errori su un singolo bit: con **4 bit di dati e 3 bit di ridondanza** servono 7 dischi, ognuno dei quali contiene un bit di ogni parola. Ha un'**ottima tolleranza ai guasti**, perché il codice permette di ricostruire il contenuto di un disco qualsiasi, ma richiede che i dischi abbiano le **rotazioni sincronizzate**, perché ogni lettura coinvolge tutti i dischi

nello stesso istante.

RAID 3 (*striping a livello di bit con bit di parità*) semplifica il RAID 2: usa **un solo disco** in cui sono raccolti i **bit di parità** di ogni parola. Un bit di parità da solo permette solo di **rilevare** un errore, ma in un RAID si sa sempre **quale disco** si è guastato, e conoscendo la posizione del bit mancante la parità basta a **ricostruirlo**: il RAID 3 offre quindi la **stessa tolleranza ai guasti** del RAID 2 con un solo disco in più. Serve però **ancora la sincronia** delle rotazioni. In generale, fare striping a livello di bit è pesante se non è gestito in hardware.

RAID 4 (*striping a livello di blocchi con XOR sull'ultimo disco*) torna allo striping **a blocchi**, come il RAID 0, e aggiunge un **disco extra** che contiene, per ogni stripe, lo **XOR** di tutte le strip della stessa riga. Poiché ogni disco lavora su blocchi interi, **non serve la sincronia**. Ha un'**ottima tolleranza ai guasti**: se un disco si guasta, ogni sua strip si ricostruisce facendo lo XOR delle strip rimaste nella stessa riga, compresa la parità. Il problema è l'**aggiornamento** quando si modifica un solo blocco: in apparenza bisognerebbe rileggere tutti gli altri blocchi della riga per ricalcolare la parità. C'è però un'**ottimizzazione**: il nuovo blocco di parità si può calcolare dal **vecchio blocco di parità**, dal **vecchio contenuto del blocco sovrascritto** e dal suo nuovo contenuto, cioè $P_{\text{nuova}} = P_{\text{vecchia}} \text{ XOR } D_{\text{vecchio}} \text{ XOR } D_{\text{nuovo}}$. Bastano quindi due letture e due scritture. Resta il fatto che **ogni** scrittura, su qualunque disco, aggiorna anche il **disco di parità**, che diventa il collo di bottiglia del sistema e si usura prima degli altri.

RAID 5 (*striping a livello di blocchi con parità distribuita*) è identico al RAID 4, ma i **blocchi di parità vengono distribuiti su tutti i dischi**, a rotazione: la parità della prima riga sta su un disco, quella della seconda su un altro, e così via. Le scritture di parità si dividono fra tutti i dischi e il collo di bottiglia sparisce, con la stessa tolleranza ai guasti; inoltre in lettura lavorano tutti i dischi, perché tutti contengono dati. Di fatto il RAID 5 **sostituisce il RAID 4**.

RIGA	DISCO 0	DISCO 1	DISCO 2	DISCO 3	DISCO 0	DISCO 1	DISCO 2	DISCO 3
0	strip 0	strip 1	strip 2	P da 0 a 2	strip 0	strip 1	strip 2	P da 0 a 2
1	strip 3	strip 4	strip 5	P da 3 a 5	strip 3	strip 4	P da 3 a 5	strip 5
2	strip 6	strip 7	strip 8	P da 6 a 8	strip 6	P da 6 a 8	strip 7	strip 8
3	strip 9	strip 10	strip 11	P da 9 a 11	P da 9 a 11	strip 9	strip 10	strip 11

Nella tabella, a sinistra la disposizione del RAID 4, con tutta la parità (P) sul disco 3; a destra quella del RAID 5, con la parità che ruota fra i dischi.

ESEMPIO SVOLTO: CALCOLARE E USARE LA PARITÀ XOR

La parità di una riga è lo **XOR** bit a bit delle strip di dati: ogni bit di parità vale 1 se il numero di 1 nella colonna è dispari. Lo XOR ha una proprietà utile: facendo lo XOR di tutte le strip di una riga, parità compresa, si ottiene 0, quindi **una strip mancante è lo XOR di tutte le altre**.

Calcolo della parità. Il disco 1 contiene le strip 0010, 1001 e 0110, il disco 2 le strip 1001, 0101 e 0100, e il disco 3 contiene la parità delle due, di cui è ignoto l'ultimo bit (101x, 110x, 001x). Prima riga: $0010 \text{ XOR } 1001 = 1011$, quindi $x = 1$. Seconda: $1001 \text{ XOR } 0101 = 1100$, $x = 0$. Terza: $0110 \text{ XOR } 0100 = 0010$, $x = 0$. I tre bit cercati sono 1, 0, 0.

Ricostruzione. Una riga di un RAID 4 con quattro dischi di dati contiene 101, 011, 110 e 000, con parità $101 \text{ XOR } 011 \text{ XOR } 110 \text{ XOR } 000 = 000$. Se si guasta il disco con 011, la strip si ricostruisce con lo XOR delle altre e della parità: $101 \text{ XOR } 110 \text{ XOR } 000 \text{ XOR } 000 = 011$. Funziona per un solo disco guasto per riga: con due strip mancanti, lo XOR non basta più.

DA RICORDARE

Perché con il RAID 0 si ottiene maggiore velocità? Che cos'è lo speedup? Che cosa succede se un disco si rompe?

Perché i dati sono divisi in strip su più dischi, che lavorano **in parallelo**: una lettura grande viene servita contemporaneamente da tutti i dischi, e richieste diverse possono andare su dischi diversi, senza overhead di parità o di copie. Lo **speedup** è il rapporto fra il tempo dell'operazione con un solo disco e il tempo con più dischi in parallelo: con RAID 0 e n dischi, nel caso migliore (richieste distribuite equamente) si avvicina a **n**; nel caso peggiore (tutte le richieste sullo stesso disco) è 1. Se un disco si rompe si perdono i dati di **tutto il volume**, perché ogni file grande ha strip su tutti i dischi e non c'è ridondanza.

Che cos'è la stripe unit e come se ne sceglie la dimensione?

È la quantità di dati scritta su un disco prima di passare al successivo. Se è **piccola** (al limite un bit o un byte, come nei RAID 2 e 3), ogni operazione coinvolge tutti i dischi anche per richieste piccole, che quindi non possono essere servite in parallelo fra loro; se è **grande**, un file piccolo sta tutto su un disco, e il parallelismo si ottiene solo fra richieste diverse. Va scelta in base al tipo di carico.

Perché nel RAID 1 lo speedup in lettura è diverso da quello in scrittura?

Perché ogni dato esiste in due copie: le letture si possono **dividere fra le copie**, servendo due richieste contemporaneamente, con uno speedup fino a 2; le scritture invece vanno eseguite **su tutte le copie**, per tenerle identiche, quindi non sono più veloci di un disco singolo, e anzi bisogna aspettare che entrambe siano completate.

DA RICORDARE

Come funziona il RAID 2 e che cosa ha in più del RAID 0? È ancora praticabile con dischi da terabyte?

Come il RAID 0 distribuisce i dati su più dischi, ma a livello di **bit**, e aggiunge dischi che contengono i bit di un **codice di Hamming**: con 4 bit di dati e 3 di ridondanza, combinando quali controlli di parità falliscono si individua e si corregge il bit errato. In più rispetto al RAID 0 ha quindi la **tolleranza ai guasti**. Oggi non si usa: richiede la **sincronizzazione** delle rotazioni di tutti i dischi, difficile con dischi meccanici reali, e i dischi moderni hanno già una propria correzione degli errori interna, che rende ridondante il codice a livello di RAID; la protezione dal guasto di un intero disco si ottiene con molto meno overhead con la parità dei RAID 4 e 5.

RAID 4 o RAID 5: perché si preferisce il RAID 5?

Entrambi fanno striping a blocchi con una strip di parità XOR per riga e tollerano il guasto di un disco. Nel RAID 4 la parità sta tutta su un **disco dedicato**: ogni scrittura, su qualunque disco dati, deve aggiornare anche quel disco, che diventa un **collo di bottiglia** e si usura prima degli altri. Il RAID 5 **distribuisce la parità** su tutti i dischi a rotazione: il carico delle scritture di parità si bilancia, il collo di bottiglia sparisce e in lettura lavorano tutti i dischi.

DA RICORDARE

Se interessa unicamente migliorare le prestazioni rispetto a un disco singolo, quale livello RAID si sceglie?

Il **RAID 0**: è l'unico che non spende nulla per la ridondanza, né spazio né scritture aggiuntive, e quindi offre il massimo parallelismo. Gli altri livelli migliorano l'affidabilità a scapito di una parte delle prestazioni, soprattutto in scrittura.

27.3 I dischi a stato solido

Gli **SSD** (*Solid State Disk*) sono dispositivi basati su **memorie flash** di tecnologia **NAND**, senza parti meccaniche. Funzionano in modo molto diverso dai dischi magnetici:

- le **letture** sono **molto più veloci delle scritture**;
- lo spazio è organizzato in **blocchi** (unità di **cancellazione**) composti da **pagine** (unità di **allocazione**), e **letture e scritture avvengono per pagine**;
- una pagina già scritta **non si può sovrascrivere**: un blocco **va cancellato prima di poter essere riscritto**, e la cancellazione riguarda sempre l'intero blocco;
- il **numero di cancellazioni è limitato** per ogni blocco: oltre un certo numero il blocco si usura e non è più affidabile.

I **file system classici** sono basati su principi diversi, pensati per i dischi magnetici, dove si sovrascrive un settore quando si vuole. Le soluzioni sono due. Si possono usare **file system ad hoc**, come **F2FS** (*Flash Friendly File System*) o i **file system log based**, che non sovrascrivono mai i dati ma scrivono sempre in aree nuove, in sequenza. Oppure, ed è la soluzione più comune, il **controller** dell'SSD **rimappa i blocchi** tramite un **Flash Translation Layer** (FTL): quando il sistema chiede di riscrivere un blocco logico, il controller scrive il nuovo contenuto in una pagina libera, aggiorna la propria tabella di corrispondenza fra blocchi logici e pagine fisiche, e marca come **invalida** (*stale*) la pagina vecchia. Così il sistema operativo continua a vedere un normale disco a blocchi, e il controller distribuisce le scritture su tutto il dispositivo, evitando di usurare sempre gli stessi blocchi.

Le pagine invalide vanno prima o poi recuperate con la **garbage collection**: il controller sceglie un blocco con molte pagine invalide, copia altrove le pagine ancora valide e cancella il blocco, rendendolo di nuovo scrivibile. C'è però un'altra forma di «spazzatura» che il controller non conosce da solo: quando il sistema operativo **cancella un file**, si limita ad aggiornare le proprie strutture (marcando i blocchi come liberi), ma per l'SSD quei blocchi contengono ancora dati validi, che la garbage collection continua a copiare inutilmente. Per questo esiste l'operazione **TRIM**, con cui il sistema operativo comunica all'SSD quali blocchi non contengono più dati utili. Senza TRIM le prestazioni **degradano nel tempo**; il suo uso **richiede l'adeguamento del sistema operativo** e del file system (ad esempio Windows supporta TRIM su NTFS).