NoiStudiamo

APPUNTI

Sistemi Operativi

Processi, thread, comunicazione tra processi, scheduling, gestione della memoria, memoria virtuale e filesystem

A CURA DI

Paolo Volpini

Novembre 2026

INDICE

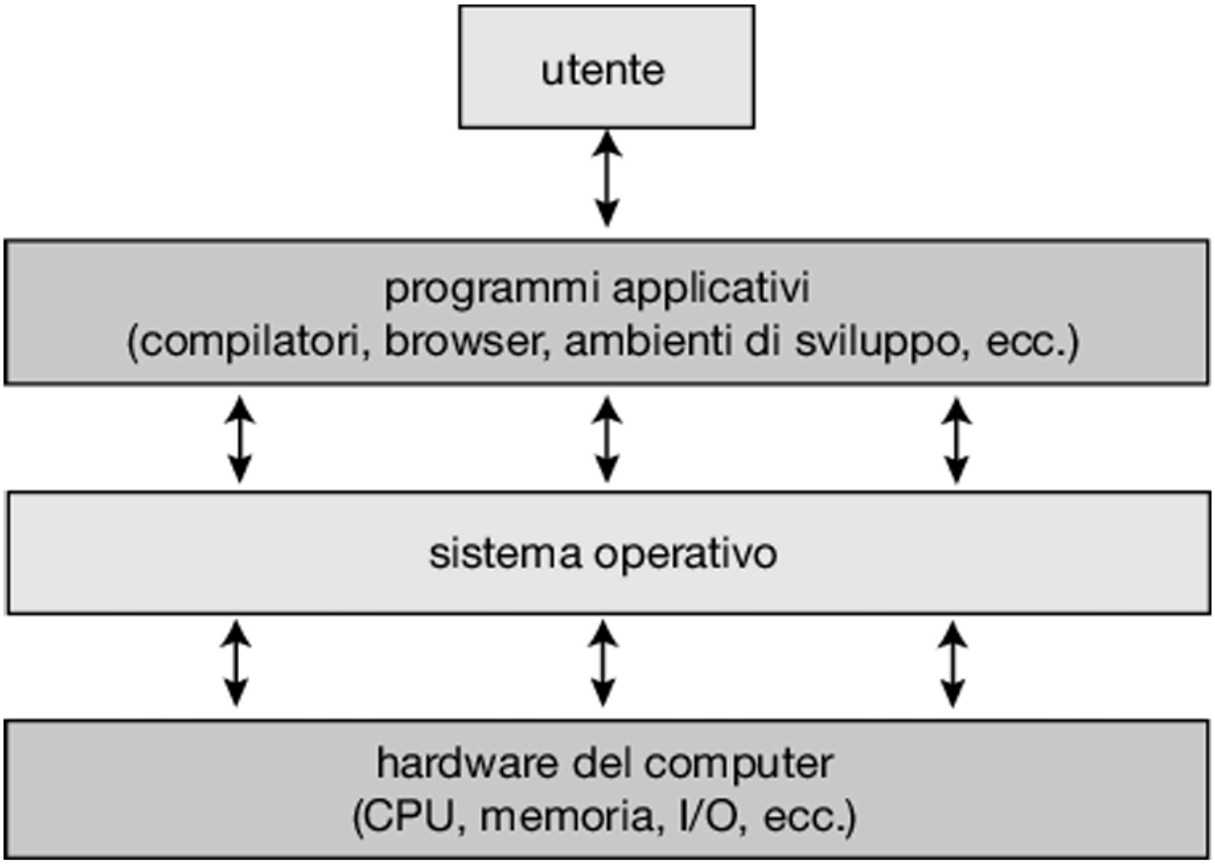
Indice degli argomenti

Gli argomenti principali degli appunti e la pagina in cui iniziano.

1. Introduzione ai sistemi operativi 3
2. Quarta lezione 15
3. Processi 25
4. Threads 33
5. Interprocess Communication 42
6. TSL, Sleep e Awake, Semafori 49
7. Futex 57
8. Problema dei cinque filosofi 65
9. Scheduling, sistemi batch 74
10. Scheduling in sistemi interattivi 79
11. Gestione della memoria 86
12. Memoria virtuale 94
13. Gestione della tabella delle pagine 103
14. Algoritmi di sostituzione delle pagine 111
15. Allocazione dei frame 118
16. Pagine condivise 126
17. Allocazione della memoria per il kernel, filesystem 132
18. Inode, directory, link e consistenza 140
19. Caching nel disco, Scheduling del disco 151

00,01,02 - Introduzione ai sistemi operativi

Un sistema operativo è un insieme di programmi che gestisce gli elementi fisici di un calcolatore, fornendo una piattaforma alle applicazioni e agendo da intermediario tra l'utente e la struttura fisica del calcolatore.
Per comprende al meglio come opera un sistema operativo, cerchiamo di identificare i componenti di un **sistema di elaborazione**:



L'hardware è composto dalla **CPU**, dalla **memoria** e dai **dispositivi I/O**. I programmi applicativi (le applicazioni quindi) definiscono il modo in cui le risorse fornite dall'hardware vengono usate. Il sistema operativo allora è quell'entità intermedia che controlla l'hardware e ne coordina l'utilizzo da parte delle applicazioni, fornendo gli strumenti per impiegare al meglio le risorse.

Definizione di sistema operativo

Non vi è una definizione realmente completa ed esauriente di cosa sia un sistema operativo, in quanto essi esistono come soluzione della realizzazione di un sistema elaborativo facile da impiegare e che aiuti l'utente a risolvere problemi, senza la difficoltà di dover imparare ad utilizzare direttamente l'hardware del calcolatore.

Una definizione comune è quella secondo cui il sistema operativo è **il programma che funziona sempre all'interno del calcolatore**: tale programma è spesso chiamato **kernel**.

> **Kernel**
>
> Il **kernel** è il programma che costituisce il cuore del sistema operativo. Ha controllo completo su tutto quello che accade nel sistema, ed è il primo programma che viene caricato in memoria durante l'avvio.

Per i sistemi operativi moderni, oltre al kernel è anche presente un **middleware**, ossia una collezione di ambienti software che fornisce servizi aggiuntivi per lo sviluppo di applicazioni.

Storia dei sistemi operativi

Possiamo percorrere la storia dei sistemi operativi analizzando le generazioni di questi.

Prima generazione: valvole e schede a spinotti

I primi calcolatori erano macchine enormi realizzate inizialmente con relè meccanici, ma erano molto lente. Successivamente i relè vennero sostituiti da valvole termoioniche. I calcolatori occupavano intere stanze ed erano milioni di volte più lenti rispetto agli attuali pc.
La programmazione avveniva in linguaggio macchina, spesso utilizzando schede particolari per controllare funzioni più elementari della macchina. Non si parlava di sistemi operativi, e generalmente il programmatore inseriva la scheda a spinotti con il programma. Tali programmi erano semplici calcoli numerici.
Le procedure migliorarono con l'introduzione di schede perforate.

Seconda generazione: transistor e sistemi batch

L'introduzione dei transistor cambiò la situazione, permettendo la creazione di calcolatori affidabili, che prendono il nome di **mainframe**. I mainframe erano tenuti in speciali sale macchine. Per avviare un **job**, ossia un programma o insieme di programmi, il programmatore prima scriveva il programma su carta e poi lo copiava sulle schede perforate; il pacchetto di schede veniva portato nella stanza dove venivano raccolti i programmi e consegnato ad un operatore. Alla fine del job, l'operatore andava alla stampante e prelevava quanto prodotto.
Dato che il costo dei mainframe era alto (infatti erano sistemi utilizzati da grosse aziende o da università), si cercava di ridurre il tempo sprecato, dovuto principalmente al fatto che gli operatori dovevano andare a raccogliere le schede e portarle nella sala macchine. I job quindi venivano eseguiti in **batch**, ossia a lotti. Quello che si faceva era quindi:

1. prendere le schede perforate e trasferirle su un nastro magnetico, usando un calcolatore relativamente piccolo come l'IBM 1401;

2. portare il nastro nel calcolatore più potente, come gli IBM 7094. Il calcolatore eseguiva i programmi nel nastro uno dopo l'altro, e produceva un nastro magnetico contenente i risultati;
3. i risultati venivano portati al 1401 per stampare i dati **off line**, ossia senza usare il calcolatore principale.

I programmi erano scritti in **FORTRAN** e linguaggio assembler, quindi spesso veniva chiesto al sistema operativo di caricare il compilatore FORTRAN dal nastro di sistema.

Terza generazione: circuiti integrati e multiprogrammazione

I calcolatori visti si dividevano in due gruppi incompatibili tra loro:

- i calcolatori usati per calcoli scientifici e ingegneristici, come l'IBM 7094;
- i calcolatori usati per ordinare dati su nastro o per le stampe, come l'IBM 1401.

Sviluppare due linee di prodotti era molto dispendioso. L'IBM tentò di risolvere introducendo il **System 360**, una serie di macchine compatibili tra loro a livello software, che utilizzava la tecnologia a **circuiti integrati** e tale che sia macchine di piccolo calibro sia quelle di grosso calibro potessero avere lo stesso software e sistema operativo, che era **OS/360**. Il problema è che scrivere un software che potesse sia supportare sistemi molto piccoli, usati solo per copiare schede su nastro, sia sistemi molto complessi, usati per calcoli avanzati, comportò la creazione di un sistema operativo che era:

- enorme;
- straordinariamente complesso;
- contenente migliaia di errori.

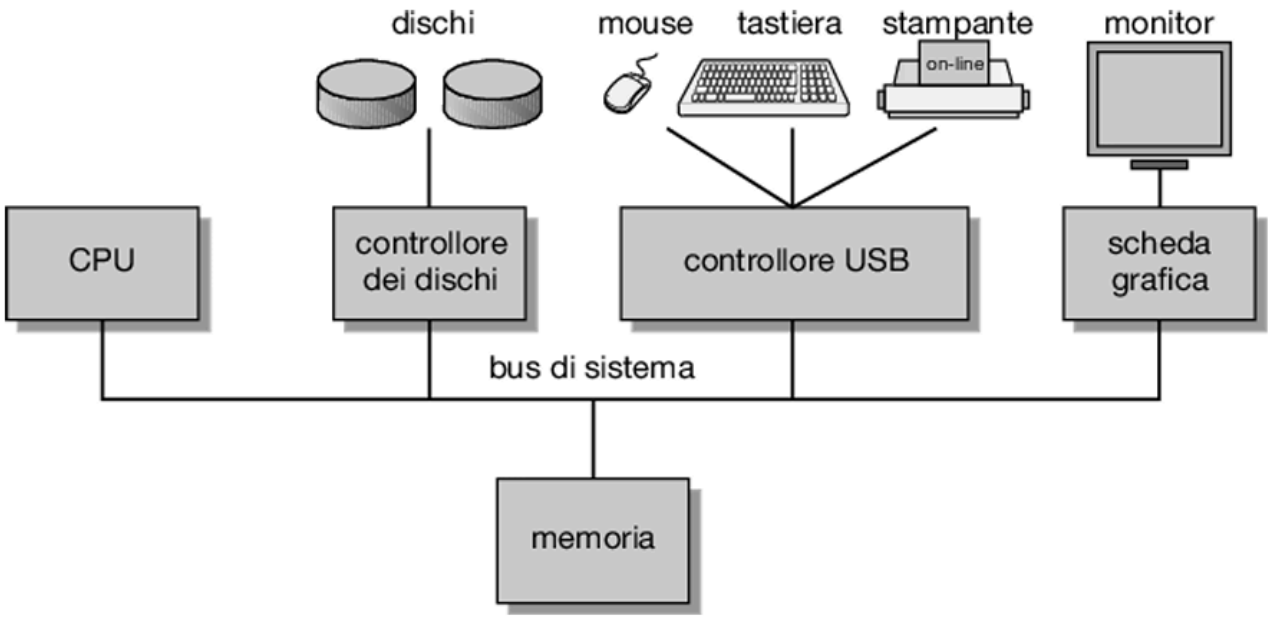
Nonostante questi problemi, in questa generazione fu introdotta la tecnica di **multiprogrammazione**, specialmente per i calcolatori come il 7094, in modo tale che la CPU potesse essere utilizzata per eseguire altri job quando uno di questi rimaneva in attesa del completamento di un'attività I/O.

Quarta e quinta generazione: Personal computer

Nella quarta generazione nasce il sistema operativo **MS-DOS**, ampiamente utilizzato dopo che aziende come Intel misero sul mercato CPU di tipo generale.
I primi sistemi operativi operavano con inserimento di comandi da tastiera. Successivamente nacquero le **GUI**. Steve Jobs fu il primo che vide il potenziale delle GUI, e così nacque **Macintosh**.
Più in avanti, alla quinta generazione, nacquero anche i sistemi operativi per dispositivi mobile, come **Android** e **iOS**.

Organizzazione di un sistema di elaborazione

Un moderno calcolatore è composto da una o più CPU e da un certo numero di **controllori di dispositivi** connessi tra loro tramite il **bus di sistema**. Ciascuno dei controllori si occupa di un particolare tipo di dispositivo fisico, e gestisce una o più unità ad esso connesse. Ad esempio, la porta usb può connettersi ad un hub, a cui possono essere diversi dispositivi.



CPU

> **CPU**
>
> La CPU è l'unita centrale di elaborazione del calcolatore. Essa recupera le istruzioni dalla memoria e le esegue.

Il ciclo di base eseguito da ogni CPU è il ciclo **fetch-decode-execute**: si recupera l'istruzione dalla memoria, la si decodifica e la si esegue.
La CPU è costituita da registri generici e da alcuni registri speciali:

1. il **program counter**, contenente l'indirizzo di memoria della successiva istruzione da eseguire;
2. lo **stack pointer**, che punta alla cima dello stack in memoria;
3. il **Program Status Word**, contenente bit di controllo (modalità kernel o utente, CPU priority, system calls, ecc.).

Molte CPU oggi sono progettate per eseguire più rapidamente il ciclo **fetch-decode-execute**:

- molte CPU utilizzano la **pipeline**, ossia mentre si esegue l'istruzione n, si decodifica

l'istruzione $n+1$, aumentando il throughput di sistema;
- il modello **superscalare** permette l'utilizzo di più unità di esecuzione, con il recupero di più istruzioni per volta memorizzate in un **buffer**.

Modalità di esecuzione dei programmi

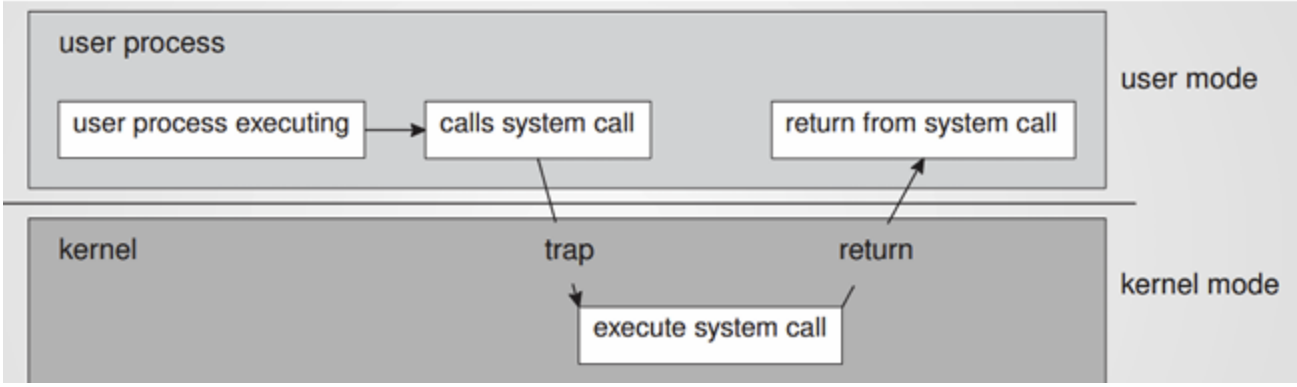
Per garantire la corretta esecuzione del sistema operativo, bisogna distinguere l'esecuzione del codice del sistema operativo e l'esecuzione del codice utente. Oggi i sistemi operativi presentano almeno due modalità per l'esecuzione dei programmi:

- la **modalità kernel**, in cui il codice in esecuzione ha accesso completo all'hardware sottostante il sistema operativo, può eseguire qualsiasi istruzione CPU e può accedere a qualsiasi indirizzo di memoria;
- la **modalità utente**, in cui il codice in esecuzione non ha accesso all'hardware e delega alle API di sistema accessi all'hardware e alla memoria.

Per ottenere servizi dal sistema operativo il programma esegue una **chiamata di sistema** o **system call**, che esegue una `TRAP` al kernel e invoca il sistema operativo.

> **Trap**
>
> L'istruzione `TRAP` è un'istruzione che cambia la modalità da utente a kernel.

In generale una chiamata di sistema permette ad un programma utente di chiedere al kernel di eseguire un'operazione privilegiata, eseguendo una `trap`. La chiamata di sistema viene gestita da una **tabella delle system call**. Dopo l'esecuzione di quest'operazione privilegiata, il kernel restituisce il controllo all'utente.



Questa duplice modalità di funzionamento protegge il sistema operativo dagli errori dell'utente. Inoltre, processori moderni come i processori Intel hanno più livelli di protezione, detti **protection ring**, dove **ring 0** indica la modalità kernel e **ring 3** la modalità utente. Nei processori Intel, la Program Status Word indica con due bit il protection ring, con 00 che indica il ring 0 e 11 che indica il ring 3.

Multithreading

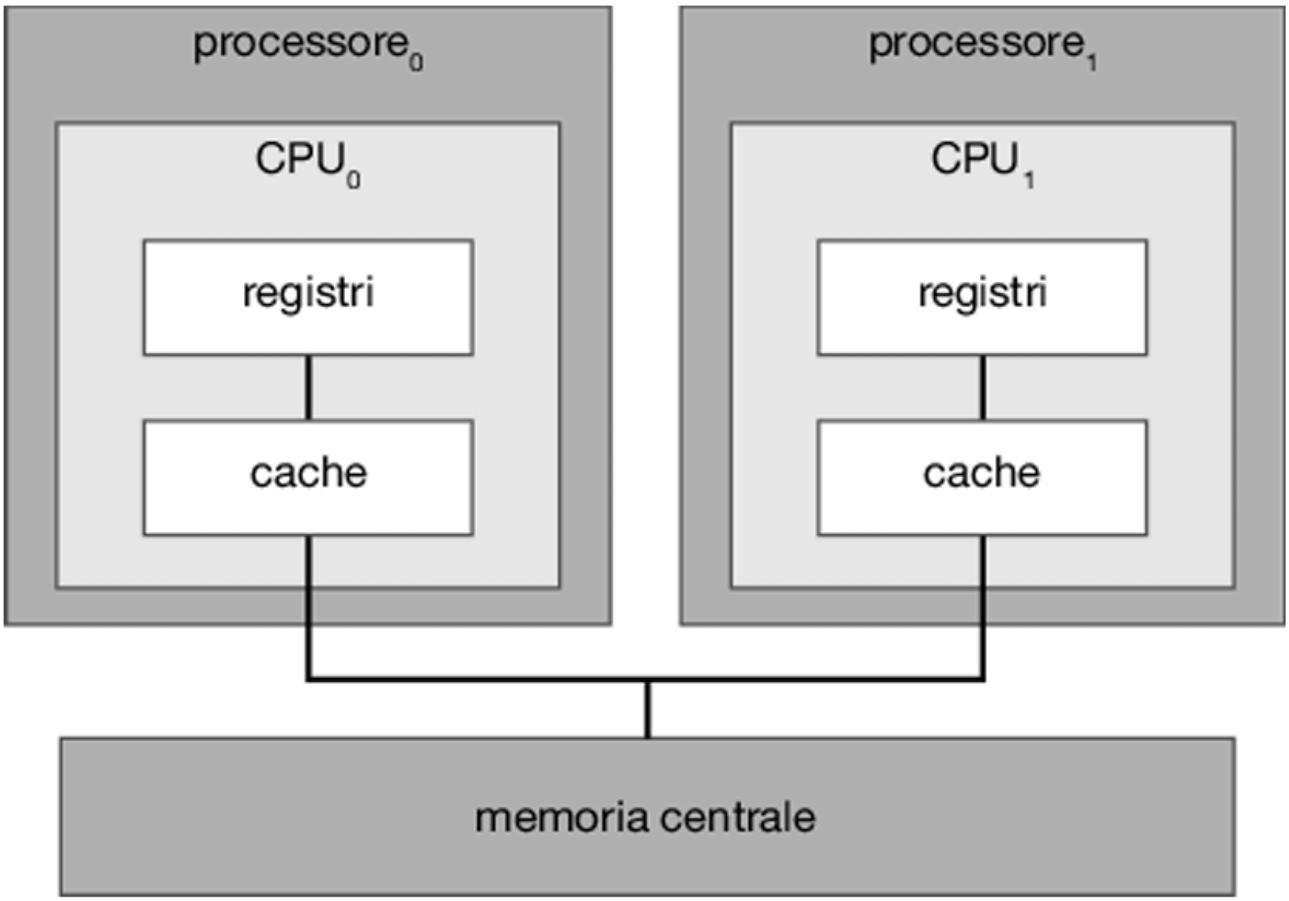
Nel ciclo di esecuzione di uno o più programmi, è possibile che la CPU sia inutilizzata. L'ideale sarebbe eseguire un altro processo o un altro thread. Questo dà origine ad una tecnica nota come **multithreading**.

> **Multithreading**
>
> Il multithreading è una tecnica che consente a più **thread**, ossia a più flussi di esecuzione di uno stesso processo, di essere gestiti dalla CPU quasi simultaneamente.

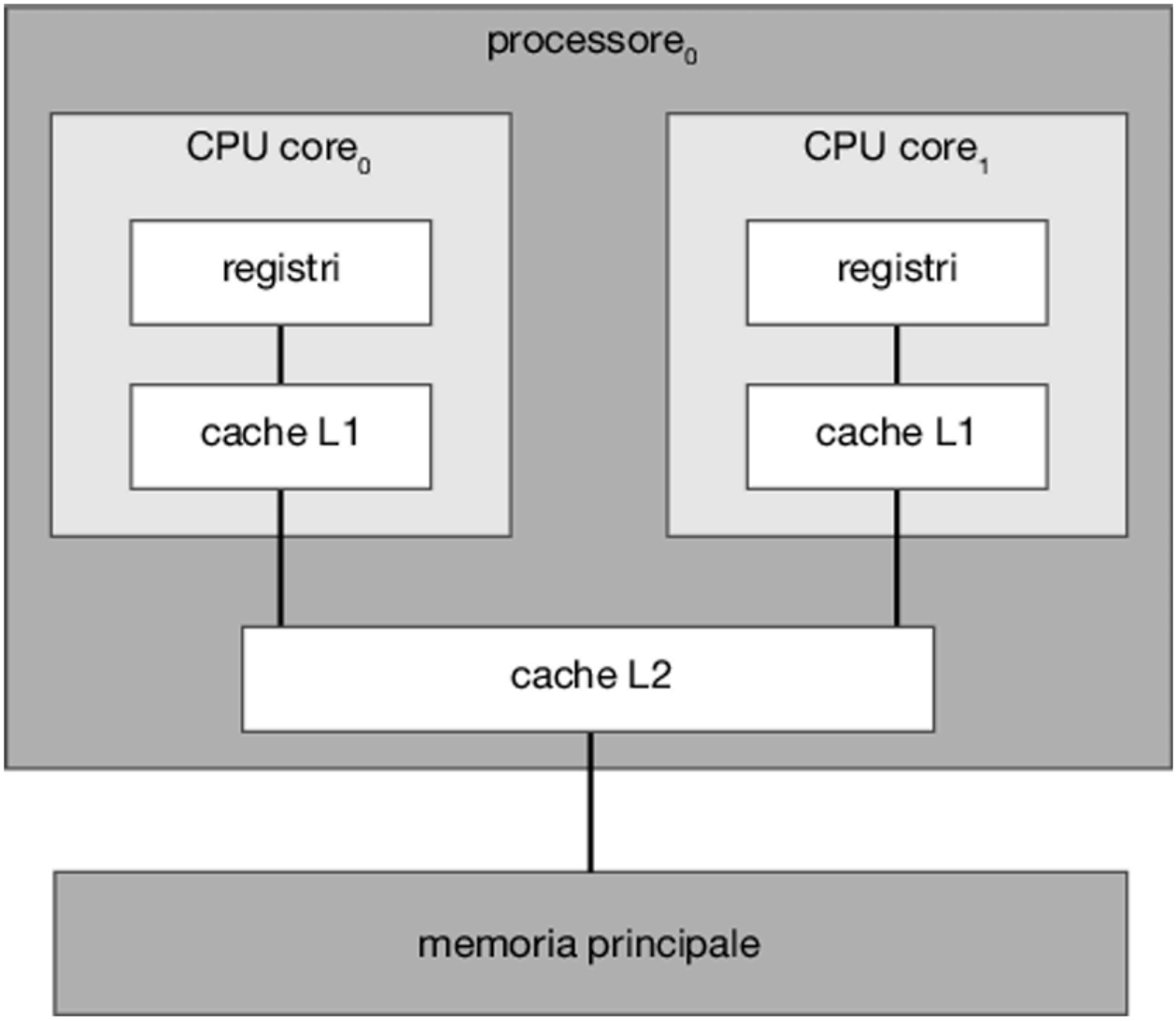
Da sottolineare che la tecnica riguarda thread appartenenti allo **stesso processo**. Questa tecnica viene eseguita con meccanismi di **time-sharing**.
Il sistema operativo dà l'illusione che i processi vengano eseguiti in parallelo usando il **context-switch** tra thread, ossia il **cambio di contesto** da un thread ad un altro, salvando lo stato del thread corrente per poterlo riprendere in seguito. Le informazioni vengono salvate in un **Thread Control Block**, una struttura dati per tracciare le informazioni dei thread in esecuzione. Questo è possibile gestendo **indirizzi virtuali** di memoria relativi ai singoli programmi, come spiegato successivamente.

Sistemi monoprocessore, multiprocessore

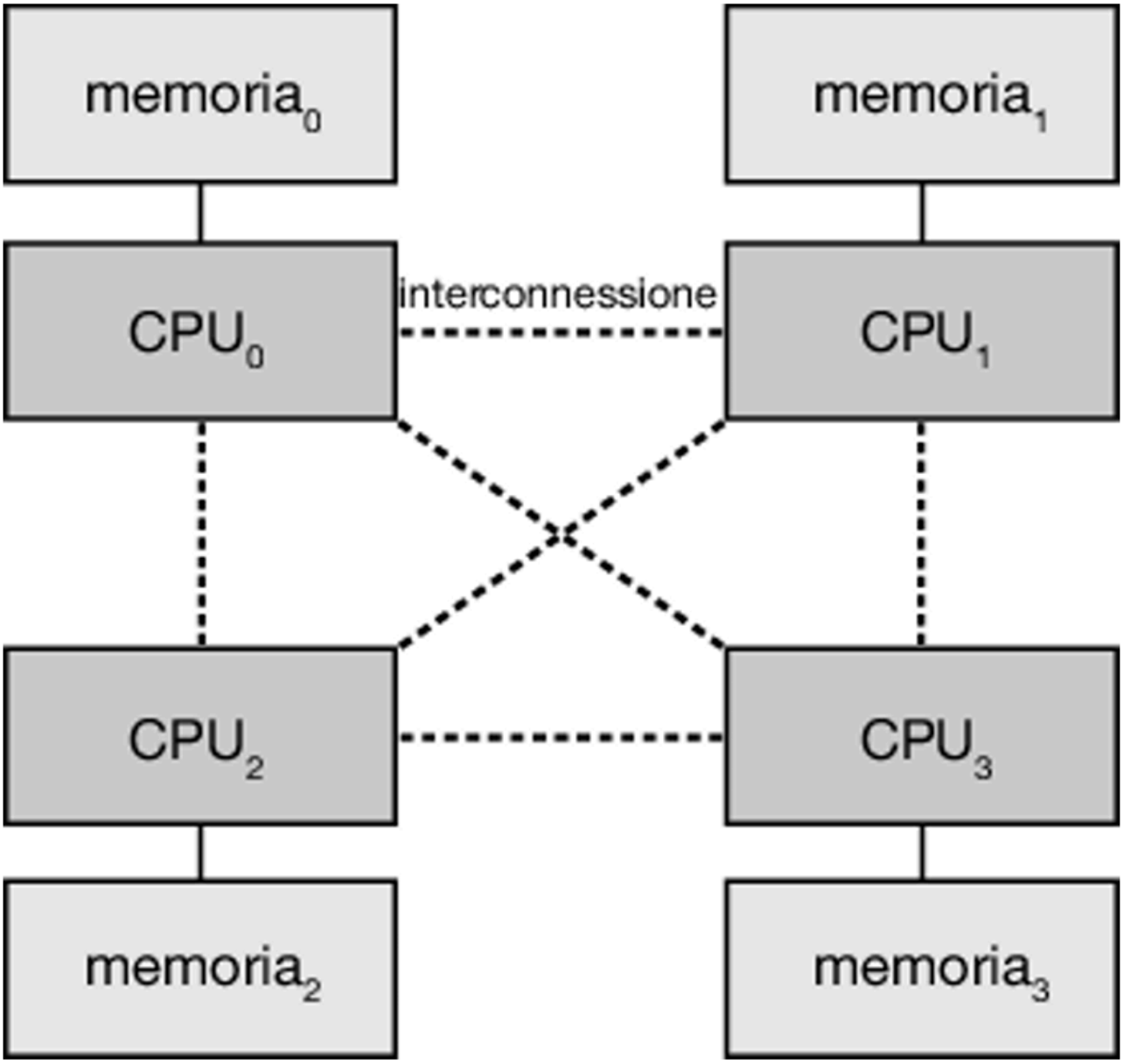
Inizialmente la maggior parte dei sistemi utilizzava un solo processore contenente un'unica CPU con un unico **core**. I **sistemi monoprocessore** possiedono altri processori specializzati, deputati a compiti particolari e che assumono la forma di processori specifici per un dispositivo. Tuttavia, i moderni computer non utilizzano sistemi a singolo processore per motivi di performance. Oggi si utilizzano sistemi **multiprocessore** e **multicore** (nel tempo il termine "multiprocessore" ha inglobato anche il termine "multicore"). Questi sistemi utilizzano la **multielaborazione simmetrica** (**symmetric multiprocessing, SMP**), in cui ogni processore è abilitato all'esecuzione di tutte le operazioni di sistema, condividendo la memoria fisica sul bus di sistema. Ogni processore è dotato dei suoi registri e di una sua cache:



Tale modello permette l'esecuzione di più processi contemporaneamente senza diminuire le prestazioni. L'unica riduzione di performance in questo caso risiede nel fatto che le unità di elaborazione sono separate. Pertanto oggi sono più comuni **sistemi multicore**, in cui vi è anche un solo processore, dotato tuttavia di più unità di elaborazione. Ad esempio, il processore in figura è un processore **dual core**:



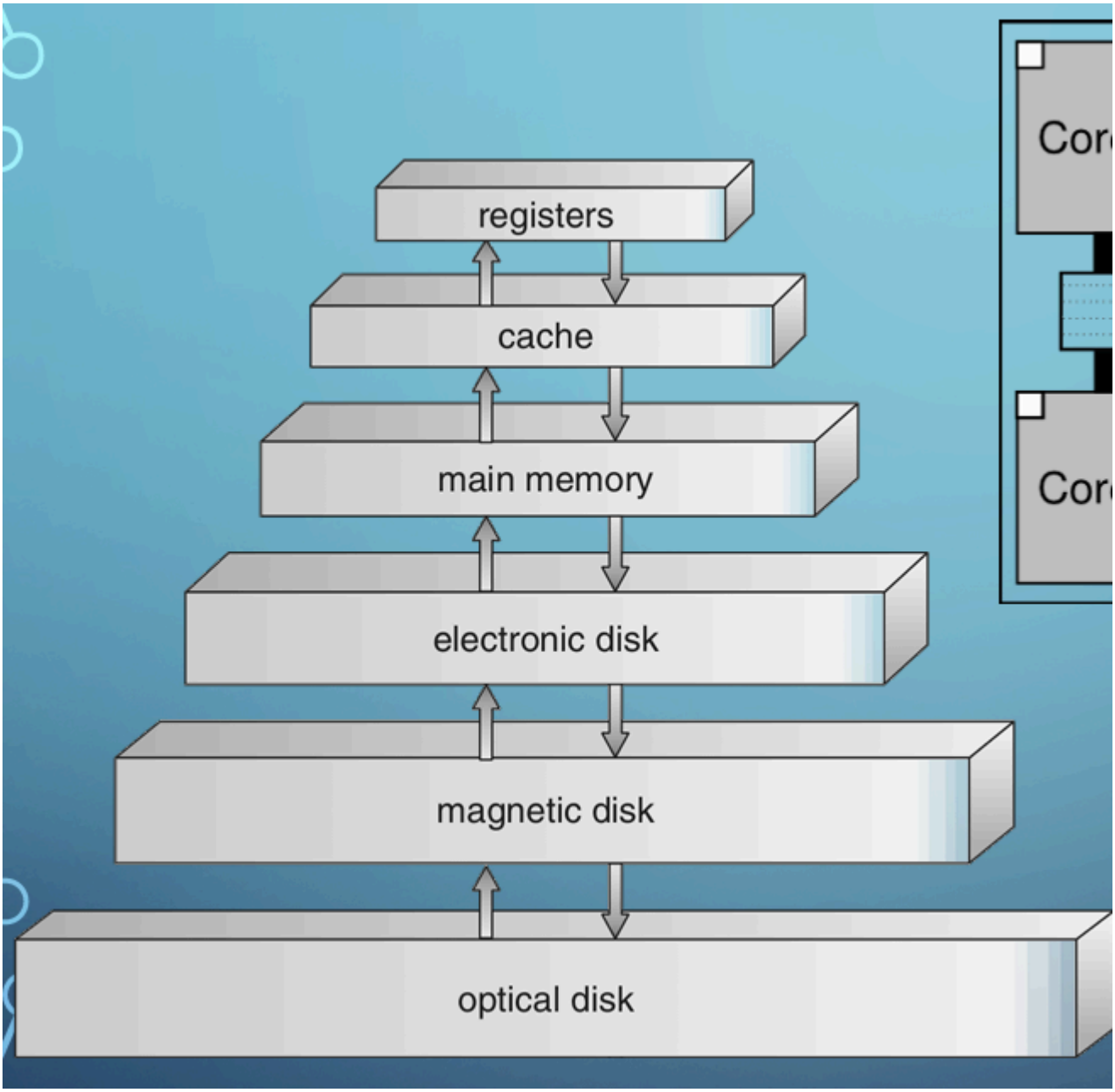
Bisogna sottolineare che aggiungere più CPU non permette di scalare in maniera lineare il throughput di sistema, in quanto con troppe CPU il **conflitto per il bus di sistema** diventa un collo di bottiglia. Un approccio alternativo sarebbe quello di fornire a ciascuna CPU la propria memoria locale e collegare tra loro le CPU. Questo approccio prende il nome di **accesso non uniforme alla memoria** (**non-uniform memory access, NUMA**) e prevede un sistema come quello sottostante:



L'accesso alle altre memorie tuttavia può subire latenze.

Memoria

La memoria in un calcolatore è costruita come una **gerarchia di strati**:



Lo strato più in alto è costituito dai registri della CPU. Sono veloci quant la CPU e non c'è ritardo nell'accesso a questi.
Successivamente si ha la memoria cache, controllata dall'hardware. Generalmente se un programma deve leggere una parola di memoria, l'hardware della cache controlla se la word richiesta è presente in cache: si ha in tal caso un **cache hit**. Altrimenti, bisogna eseguire un accesso in memoria principale.
La memoria principale è anche nota come **Random Access Memory** (**RAM**). La terminologia "accesso casuale" fa riferimento al fatto che si può accedere a qualunque indirizzo di memoria con gli stessi tempi.
A seguire ci sono i dischi elettronici e poi i dischi magnetici. I dischi magnetici costituiscono gli odierni HDD. Ogni disco è costituito da più piatti metallici, e su ogni disco una puntina legge le informazioni sul disco similmente ad un vinile. Le informazioni sono scritte sul disco come una serie di cerchi concentrici. Ogni testina legge una regione a forma di anello nota come **traccia**,

e tutte le tracce per una posizione del braccio formano un **cilindro**. I dati cercati vengono letti semplicemente aspettando che la testina legga il settore di una traccia specifica.

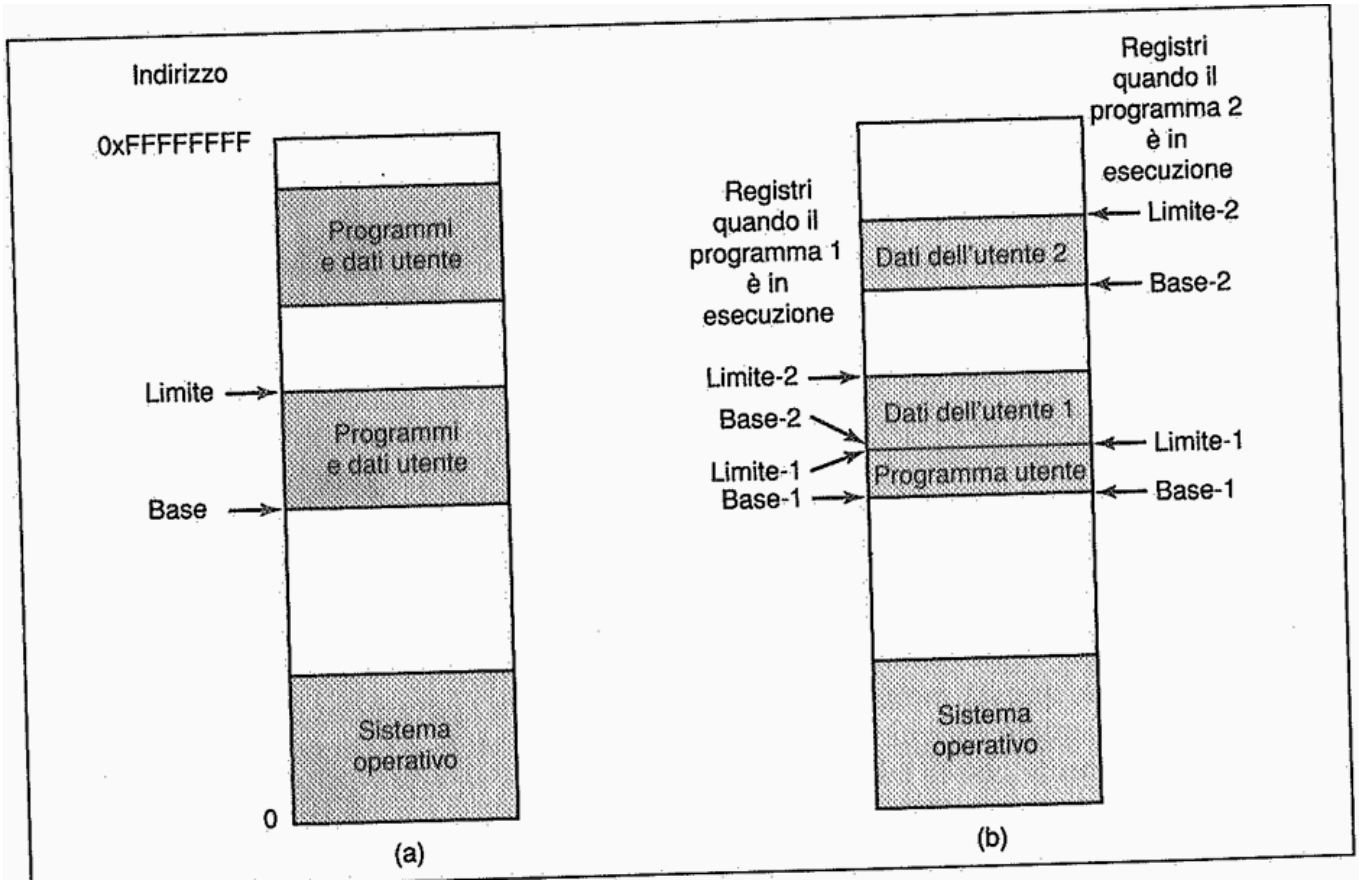
Molti elaboratori presentano un'area di memoria non volatile a random access nota come **ROM** (**Read-Only Memory**), che tendenzialmente è quella che contiene il **bootstrap loader**, ossia il programma di avvio per l'elaboratore stesso. Esiste anche la memoria **CMOS**, utilizzata per memorizzare ora e data corrente. Il circuito CMOS è alimentato da una batteria, e oggi viene usato per le schede madri per memorizzare le informazioni del **BIOS**.

Gestione della memoria per l'esecuzione dei programmi: MMU

Spesso si mantengono più programmi in memoria nello stesso istante. In tal caso bisogna risolvere i seguenti problemi:

1. proteggere i programmi l'uno dall'altro;
2. gestire la rilocazione.

Il problema risiede nel fatto che quando un programma è compilato e linkato, non si sa dove verrà caricato in memoria fisica quando esso sarà eseguito. Se ad esempio la prima istruzione recupera una parola dall'indirizzo 10000, ma il programma è caricato a partire dall'indirizzo 50000, chiaramente ci sarà un errore. Bisogna quindi rilocare il programma quando viene caricato. Ciò può essere fatto utilizzando **due registri speciali**: il **registro base**, impostato al punto di inizio del codice del programma eseguito, e il **registro limite**, che contiene il totale dello spazio occupato dal codice e dai dati del programma. L'hardware controlla che il program counter sia minore del registro limite, sommando il contenuto del registro base all'istruzione che dev'essere recuperata. Riprendendo l'esempio precedente, la parola viene realmente recuperata all'indirizzo 60000.
Il dispositivo che fa corrispondere gli **indirizzi virtuali**, ossia gli indirizzi generati dal programma, agli **indirizzi fisici**, ossia gli indirizzi utilizzati in memoria, è chiamato **Memory Management Unit** (**MMU**).
Il modello descritto è un modello molto semplice, ma in realtà MMU più avanzate permettono anche di definire più coppie registro base-registro limite, permettendo a più utenti di condividere lo stesso programma senza che esso sia caricato due volte in memoria:



Clock di sistema, clock di CPU

Per sincronizzare tutte le componenti del sistema operativo le schede madri utilizzano un **clock di sistema**. Il clock di sistema è un piccolo cristallo di quarzo localizzato sulla scheda madre. Il cristallo funge da **oscillatore**: quando riceve energia, il cristallo vibra ad una frequenza costante, inviando quindi un segnale a tutte le altre componenti dell'elaboratore. La velocità di clock è impostata in modo tale che il segnale del clock arrivi prima che altri segnali siano pronti, mantenendo tutto sincronizzato.
Il clock di sistema è diverso dal **clock della CPU**: l'idea è sempre quella, ma il quarzo è posto all'interno della CPU e viene utilizzato dal chip. Il motivo della presenza del processore risiede nel fatto che se circuiti logici devono scambiarsi informazioni, è necessario che a ciascuno di questi sia indicato il momento esatto in cui si possa ritenere valido il segnale in ingresso. Il clock è condiviso da tutti i circuiti della CPU.

03 - Quarta lezione

Dispositivi di I/O

Le attività che svolgiamo richiedono l'utilizzo di risorse esterne. Una di queste risorse esterne può essere anche un hard disk.
I dispositivi di I/O sono costituiti da due componenti:

- il **controller**;
- il dispositivo stesso.

Controller

Il controller fornisce un'interfaccia più semplice. Fondamentalmente è un chip o un insieme di chip che controlla il dispositivo, accettando comandi dal sistema operativo, che sono lettura e scrittura di dati sul dispositivo. Il controller presenta un'interfaccia che sia la più semplice possibile per dialogare con il sistema operativo.
Ogni controllore di dispositivo ha un suo **buffer**, ossia una **memoria interna**, e un insieme di **registri**. Il controllore è responsabile del trasferimento dei dati tra i dispositivi periferici a esso connessi e la propria memoria interna. Ad esempio, digitare tasti sulla tastiera implica che l'interfaccia hardware della periferica memorizzi nel buffer il codice ascii corrispondente al carattere inserito.
I sistemi operativi gestiscono i controllori dei dispositivi tramite i **driver**.

> **Driver**
>
> Un driver è un insieme di procedure software che permette al sistema operativo di utilizzare l'hardware tramite un'interfaccia standard e i registri del controllore.

Il driver è **specifico**, e potrebbe essere differente a seconda del tipo di sistema operativo utilizzato. Inoltre, i driver agiscono in modalità **kernel**.
I controller vengono programmati attraverso le porte I/O (come porte USB). Le porte sono programmabili solo dal kernel in modalità supervisor.

Gestione I/O

L'I/O viene gestita in tre principali modi:

- **busy waiting**;
- uso di **interrupt**;

- con uso di **DMA**.

Busy waiting

Il metodo più semplice per gestire l'interazione I/O consiste semplicemente in una chiamata a sistema, che il kernel traduce in una chiamata di procedura al driver appropriato. Il driver quindi dà inizio all'operazione I/O. A questo punto l'unità di elaborazione esegue un **polling** (una richiesta di Load all'indirizzo di periferica) continuo per verificare l'esito della richiesta I/O. La CPU quindi chiede costantemente se la risposta è arrivata. Questo metodo è detto **busy waiting**, ed è una tecnica svantaggiosa in quanto la CPU viene tenuta occupata nel polling finché l'attività di I/O non è terminata. Si potrebbe infatti occupare l'unità di elaborazione per eseguire un'altra attività, invece di aspettare.

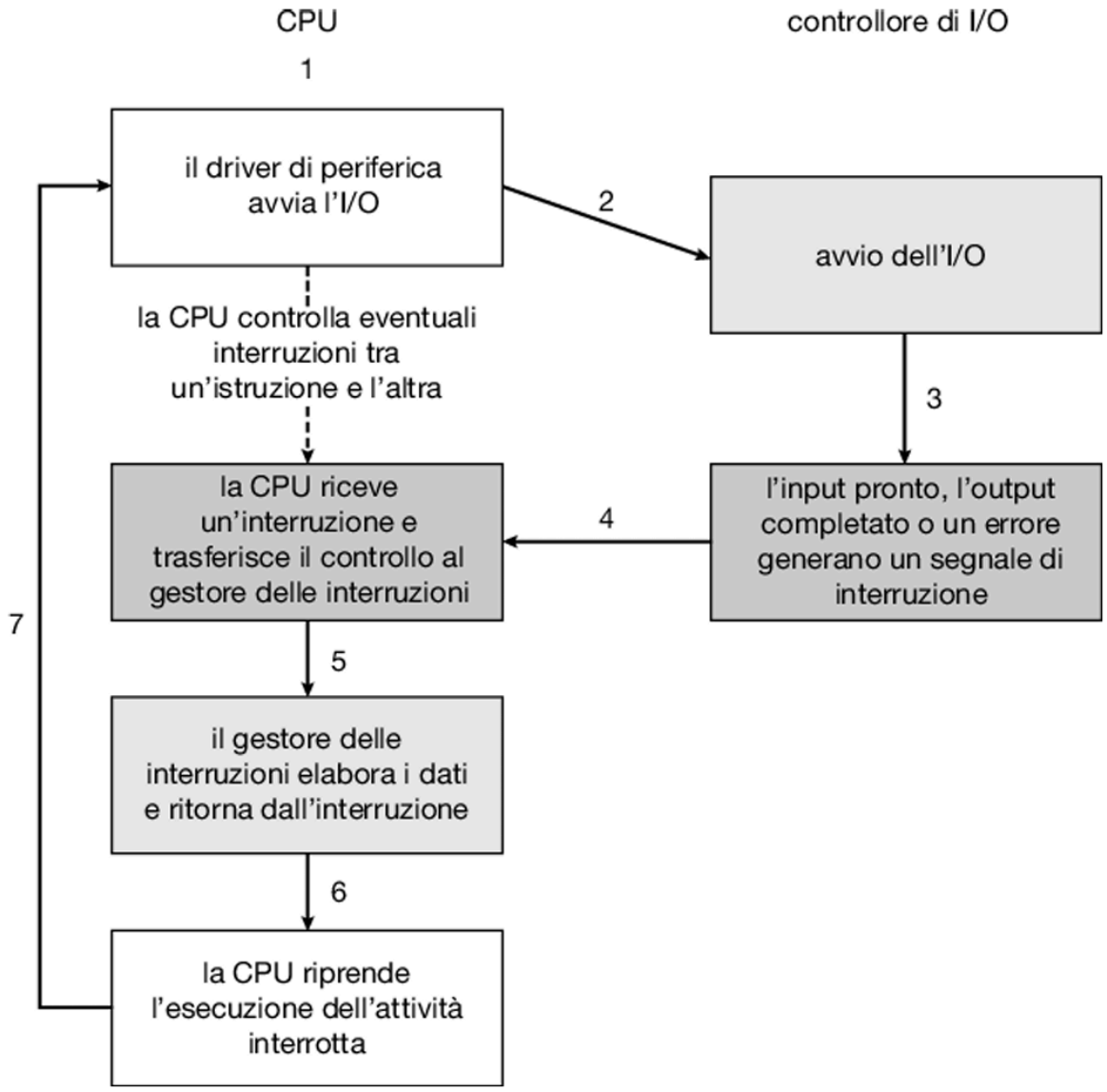
Uso di interrupt

Per gestire le operazioni di I/O, il driver del dispositivo scrive nei registri all'interno del controllore. Quest'ultimo esamina i contenuti dei registri per dedurre l'operazione da compiere. I dati allora vengono trasferiti dal dispositivo al buffer. A fine trasferimento, il controllore comunica al driver tramite un'**interruzione** di avere terminato l'operazione. Il driver:

1. restituisce i dati se l'operazione era di lettura;
2. restituisce informazioni di stato se l'operazione è di scrittura.

L'hardware può generare un'interruzione in un qualsiasi momento inviando un segnale alla CPU tramite il **Control Bus**. La CPU allora interrompe l'elaborazione corrente, trasferendo l'esecuzione a una locazione fissa di memoria che contiene l'indirizzo iniziale della **Interrupt Service Routine**, ossia della routine di servizio dell'interruzione. Tale routine termina sempre con un'istruzione `return_from_interrupt`, che indica alla CPU di riprendere l'elaborazione precedentemente interrotta.
Poiché vi sono molteplici tipologie di dispositivi, anche le richieste di interruzione sono diverse. Le routine di gestione pertanto non possono sempre essere le stesse. Per gestirne più di una, nei calcolatori viene utilizzato l'**Interrupt Controller**, un controllore dotato di più linee **IRQ** (**Interrupt ReQuest**). Nel momento in cui una periferica genera un'interruzione, l'interrupt Controller invia sul Data Bus il numero di IRQ. La CPU, nel momento in cui gestisce l'interrupt, recupera dal Data Bus il numero di IRQ e lo utilizza come **offset** in una **tabella di puntatori**, l'**Interrupt Vector**. L'elemento indicizzato corrisponde all'indirizzo di memoria della corretta ISR.

Lo schema seguente riassume quanto esposto:



Inoltre, la maggior parte delle CPU ha due linee di richiesta di interruzione:

1. una **non-maskable interrupt line**, riservata a eventi come errori irreversibili di memoria;
2. una **maskable interrupt line**, che può essere disattivata prima dell'esecuzione di sequenze di istruzioni critiche che non devono essere interrotte. I controllori dei dispositivi utilizzano quest'ultima.

Lo scopo del meccanismo di interruzione tramite tabella di vettori di interruzione è quello di evitare a un singolo gestore di interruzioni di dover cercare tra tutte le possibili fonti per determinare quale ha bisogno di assistenza. Nella realtà però i computer hanno anche più dispositivi di quanti siano i campi nel vettore di interruzioni. Per ovviare a ciò, si utilizza il **concatenamento delle interruzioni**. Ogni elemento della Interrupt Vector punta alla testa di un elenco di gestori, e quando viene sollevata un'interruzione, i gestori vengono chiamati uno ad uno fino a quando non viene trovato uno in grado di servire la richiesta.

Uso di DMA

Il **DMA** permette di controllare il flusso tra la memoria e controller, senza che intervenga l'unità di elaborazione. La CPU avvia il chip DMA specificando quanti bit devono essere trasferiti, il dispositivo e gli indirizzi di memoria coinvolti e ulteriori informazioni necessarie per l'operazione. Il dispositivo quindi accede direttamente alla memoria senza chiamare in causa l'unità di elaborazione, e alla conclusione dell'operazione viene inviato un segnale interrupt.

Precisazione sui segnali interrupt

I segnali di interrupt possono avvenire in qualsiasi momento, e indicano un interruzione dell'esecuzione per qualunque motivo, come richiesta di I/O o errore. Possono esserci delle circostanze particolari per cui un segnale di interrupt può compromettere considerevolmente le prestazioni del sistema. L'unità di elaborazione ha una sua autorità per disabilitare alcuni segnali di interrupt, disabilitando alcune **IRQ**.
Quando i segnali di interrupt vengono disabilitati, un processo in esecuzione potrebbe terminare la sua esecuzione quando viene terminato o quando c'è un errore. Nessun'altra circostanza permetterà l'interruzione del processo. Ci sono anche situazioni forzate per cui un programma termina l'esecuzione:

1. **errori**, quali segmentation fault;
2. richiesta I/O inesistente;
3. **kill** da un altro programma.

Bus

Per ottenere importanti prestazioni, è fondamentale che le comunicazioni tra i vari dispositivi avvengano in maniera efficiente. In particolare, la comunicazione tra la memoria e l'unità di elaborazione è la più importante.
In generale sappiamo che i dispositivi comunicano tramite il **bus**, uno per i dati, uno per il controllo e uno per gli address di memoria. Nei moderni calcolatori in realtà ci sono anche bus aggiuntivi, sia per facilitare l'interazione con i dispositivi di I/O sia per velocizzare la comunicazione tra CPU e memoria.
La maggior parte dei bus sono standardizzati. Il bus principale utilizzato oggi è il **PCIe**, ossia il **Peripheral Component Interconnect express**. Si tratta di un bus seriale ad alta velocità usato per collegare il calcolatore con schede di espansioni quali GPU, schede di rete e SSD NVMe.
Un altro bus importante è il **Direct Media Interface** (**DMI**), bus ad alta velocità che connette la CPU al **chipset** della scheda madre, permettendo l'interazione con porte USB, SATA e altri dispositivi I/O. Le porte **USB**, ossia **Universal Serial Bus**, nacquero per permettere a dispositivi più lenti di comunicare col calcolatore secondo il principio **plug-and-play**: poiché i dispositivi USB hanno un solo driver USB, non si ha la necessità di riavviare il calcolatore,

perché il kernel non deve cercare all'avvio un nuovo driver da avviare.
I bus inoltre si dividono in due macrogruppi:

- bus **paralleli**, in cui i bit della word da trasferire vengono trasferiti contemporaneamente, ciascuno su una linea dedicata;
- bus **seriali**, in cui i bit della word vengono trasmessi uno dopo l'altro. La connessione durante la quale avviene il trasferimento viene detta **lane**.

I vari sistemi operativi

Esistono diversi tipi di sistemi operativi, ognuno creato per gestire determinati scenari e calcolatori.

Sistemi per mainframe e server

I mainframe e i server gestiscono **grandi moli di dati** e **numerosi processi attivi**. Un sistema operativo per questo tipo di calcolatori dev'essere in grado di gestire più CPU fisiche in parallelo e un elevato carico di richieste I/O, supportando la multiprogrammazione pur mantenendo una straordinaria stabilità.
Uno tra i primi sistemi operativi nati per la gestione di mainframe fu **UNIX**.

Sistemi operativi per calcolatori multiprocessore

Altri sistemi operativi sono sviluppati per la gestione di sistemi **multiprocessore**. Questi sistemi vengono utilizzati per eseguire numerose attività batch, e necessitano di sistemi operativi particolari, generalmente varianti di sistemi operativi per i server, con caratteristiche particolari per comunicazione e connessione.

Sistemi operativi per personal computer

I sistemi operativi che usiamo ogni giorno sono sviluppati per i **personal computer**. Questi sistemi devono gestire la **multiutenza** e la **protezione dei dati**, semplificando anche l'uso tramite interfacce grafiche.

Sistemi operativi mobile

Ci sono anche **sistemi operativi mobile**. Non si ha una multiutenza, e le GUI sono progettate per il touchscreen. I processi, inoltre, sono isolati tra loro. Il sistema operativo molto spesso è anche sviluppato in virtù dell'hardware su cui è costruito, cosa che non è sempre vera per altri dispositivi (ad esempio, Windows può supportare molti calcolatori diversi tra loro. Molte **ROM di Android** sono cablate sullo smartphone o tablet che lo eseguirà).

Sistemi operativi per calcolatori embedded

I sistemi embedded sono sistemi progettati per eseguire dei compiti specifici e definiti. L'utente non dovrà eseguire processi aggiuntivi, e i processi che girano sono pochi e noti agli sviluppatori del sistema operativo.

Sistemi operativi real-time

Infine ci sono anche sistemi operativi **real-time**, sviluppati per gestire attività critiche. Questi si dividono in:

1. **hard real-time**, se l'operazione che non avviene in un certo istante compromette l'intero sistema;
2. **soft real-time**, se il sistema non viene compromesso nella sua interezza quando un'operazione non avviene in un certo istante.

In entrambi i casi, questi sistemi operativi sono **non preemptive**, cioè non interrompono un processo per eseguirne un altro, ma attendono che il processo si completi affinché il successivo possa cominciare.

Struttura di un sistema operativo

Il kernel di un sistema operativo può essere progettato in molteplici modi. La maggior parte degli OS odierni è **ibrido**, strutturando le varie componenti del sistema in modo diverso.

Sistema operativo con struttura monolitica

Il sistema operativo viene definito come un **singolo programma in modalità kernel**. Il sistema è una raccolta di procedure, in cui ogni procedura può richiamare tranquillamente una qualunque altra procedura senza alcuna restrizione. Si ha un sistema da un lato semplice, ma dall'altro lato poco gestibile e più difficile da comprendere.
Usando questo approccio, per costruire il programma oggetto del sistema operativo, prima si compilano le singole procedure e poi vengono linkate in un unico file oggetto. Vi è assenza, come intuibile, di occultamento di informazioni. Si può tuttavia intuire una struttura di base per il sistema operativo in virtù del fatto che le chiamate di sistema vengono richieste mettendo i parametri in luoghi ben definiti e poi eseguendo una `trap`. L'istruzione trasferisce così il controllo al sistema operativo, che esamina i parametri e determina la system call opportuna.
La struttura che si deriva è la seguente:

1. un programma principale chiama le procedure di servizio richieste;
2. un insieme di procedure di servizio eseguono le chiamate di sistema;
3. un insieme di procedure di utilità aiutano l'esecuzione delle procedure di servizio.

Sistemi di questo tipo sono **MS-DOS**, **UNIX** (nella sua versione iniziale).

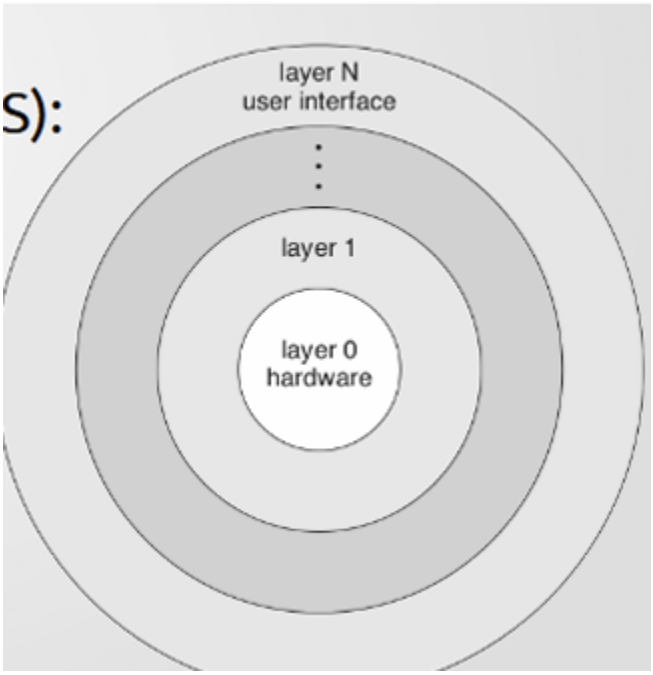
Sistema operativo con struttura a livelli

In questa struttura gerarchica ogni livello è costruito sopra il livello sottostante. L'idea è che ogni livello implementa una particolare funzione del sistema operativo attraverso insiemi di procedure. Questa stratificazione è puramente a livello logico, pertanto un bug che espone strutture dati in un livello può sempre verificarsi.
Ad esempio, il sistema operativo THE, sviluppato da Dijkstra, implementa la struttura a livelli come segue:

Struttura del sistema operativo THE.

Livello	Funzione
5	L'operatore
4	Programmi utente
3	Gestione del'input/output
2	Comunicazione operatore-processo
1	Gestione della memoria e del tamburo
0	Allocazione del processore e multiprogrammazione

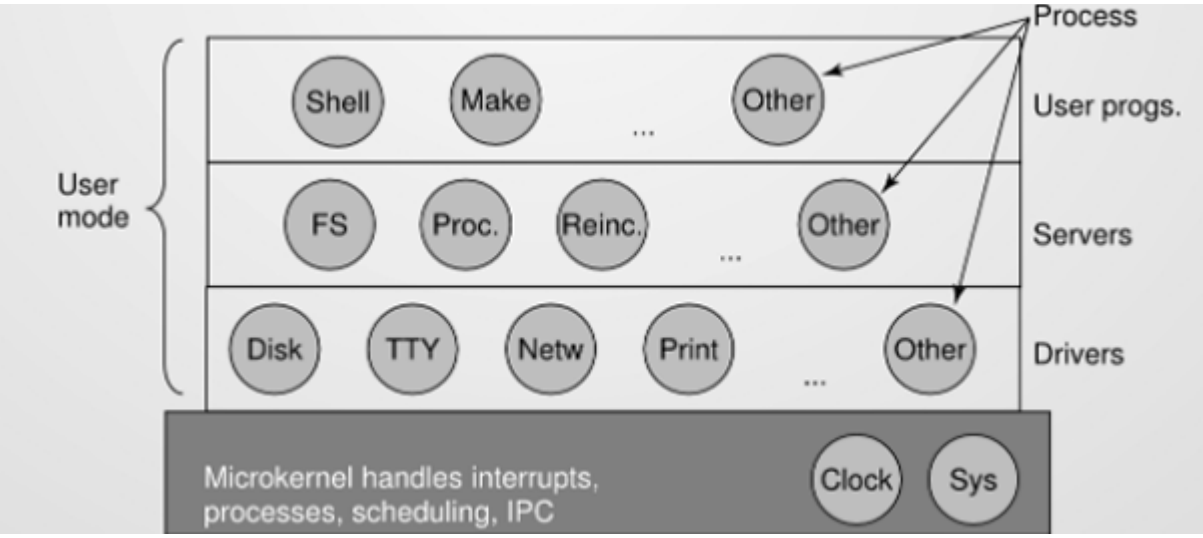
Il problema di questo tipo di sistema riguarda le prestazioni, dovuto alle chiamate nidificate che possono causare un aumento di **overhead** e quindi una diminuzione di efficienza.
Una variante della struttura a livelli è la **struttura a livelli con anelli concentrici**. I livelli in questo caso sono separati con meccanismi di protezione implementati a livello hardware, e più si sale in alto più i processi hanno meno privilegi. Se un processo di livello superiore deve interagire con uno di livello inferiore, esso deve eseguire una chiamata `trap`. Il sistema venne implementato in **MULTICS**, ma ebbe scarso successo proprio perché l'overhead in questo caso diventa rilevante a causa di ulteriori controlli e operazioni di gestione.



Sistema operativo microkernel

I sistemi **microkernel** cercano di collocare il meno possibile tutto ciò che va eseguito in modalità kernel. In questo modo si garantisce maggiore stabilità ed efficienza al sistema. In questa struttura, il sistema operativo viene diviso in piccoli moduli. Di questi piccoli moduli, uno soltanto sarà eseguito in modalità kernel, mentre i restanti in modalità utente. Inoltre, se servizi in modalità utente falliscono, **il sistema lo termina e lo riavvia automaticamente**: tale processo prende il nome di **reincarnazione**.
Il livello più basso, gestito dal kernel, gestisce gli interrupt, i processi, lo scheduling, la comunicazione tra processi, il **clock** e le chiamate di sistema. Nel livello superiore abbiamo i driver dei dispositivi. Successivamente incontriamo i **server**. Infine, si ha il livello dell'utente, che contiene tutti i programmi relativi all'utente.
Il kernel comunica con le altre componenti utilizzando dei messaggi `trap`, che includono:

1. **mittente e destinatario** (process ID);
2. **tipo di richiesta**;
3. ulteriori dati aggiuntivi per l'operazione.



Si tratta di una struttura più snella, efficiente e più stabile, isolando i componenti esterni al kernel in uno spazio utente separato. Un esempio di sistemi è Mac OS e Windows NT.

Sistema operativo con struttura a moduli

Il sistema operativo in questo caso è costituito da un **kernel modulare**. L'idea è utilizzare moduli caricati dinamicamente. Il kernel principale è a funzionalità ridotte. La struttura è molto flessibile, in quanto ogni modulo può invocare qualunque altro modulo direttamente. I moduli, inoltre, sono caricati **on demand**, riducendo l'uso della RAM.
Anche in questo caso il kernel è un nucleo ridotto. Più precisamente, è un **nucleo essenziale** con funzioni base. Se un determinato dispositivo hardware o una funzionalità viene richiesta, il kernel può caricare un modulo in RAM. Tali moduli sono dei file con estensione `.ko` (**Kernel Object**). Il vantaggio è quindi che la comunicazione tra le componenti del sistema non avviene con messaggi. Uno svantaggio, tuttavia, risiede nel fatto che **i moduli sono eseguiti in modalità kernel**.
Oggi Mac OS X e Linux utilizzano tale struttura. Tuttavia Mac OS X è ibrido tra il sistema a microkernel e la struttura a moduli.

Macchine virtuali

L'estremizzazione dell'astrazione viene raggiunta con la **virtualizzazione di un intero computer**. Ciò viene raggiunto con le **macchine virtuali**.
Una macchina virtuale è in grado di eseguire software come se fosse in un computer fisico ed eseguire un sistema operativo diverso da quello dell'host.
Le macchine virtuali sono utilizzate nell'ambito della sicurezza informatica in quanto **isolate a livello logico dalla macchina ospitante**.
Vi sono vari modi per implementare le macchine virtuali:

1. **container**: un container si basa sul kernel del sistema operativo ospitante e contiene app, alcune API e servizi del sistema operativo eseguiti in modalità utente;

2. **virtualizzazione classica**: la CPU della macchina ospitante esegue direttamente il codice eseguito nelle macchine virtuali, garantendo velocità;
3. **emulazione**: il codice del software ospitato viene eseguito da una **CPU** emulata, con registri emulati internamente tramite array. Si tratta di una soluzione meno efficiente rispetto alla virtualizzazione classica
4. **para-virtualizzazione**: il sistema operativo ospitato **riconosce di essere virtualizzato**, ed è in grado di interagire con l'**hypervisor**. Ciò garantisce prestazioni migliori rispetto alla virtualizzazione classica, in quanto il sistema operativo è creato ad hoc per essere virtualizzato, e permette un miglior uso delle risorse grazie alla comunicazione diretta tra guest e hypervisor.

Hypervisor

Le macchine virtuali vengono gestite tramite gli **hypervisor**, una componente software che intercetta e controlla tutte le operazioni della macchina virtuale. Gli hypervisor sono di due tipi:

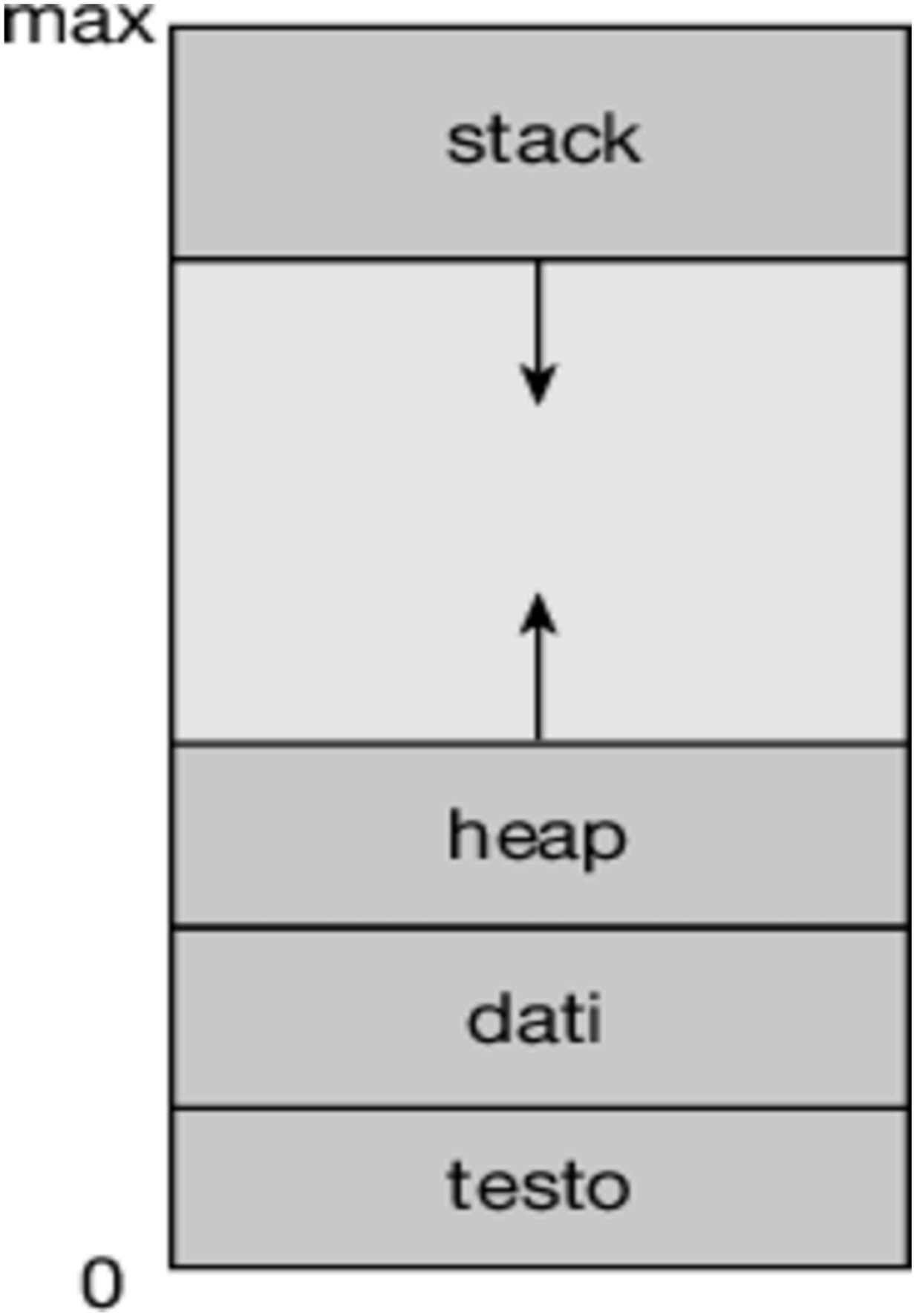
1. **hypervisor di tipo 1**: viene installato uno strato software **direttamente nell'hardware** della macchina principale, che ha come compito la creazione di macchine virtuali;
2. **hypervisor di tipo 2**: l'hypervisor viene installato su un sistema operativo ospitante. Risulta meno efficiente ma più adatto per lo sviluppo di software. Molto spesso questo tipo di hypervisor necessita di lavorare a contatto con il kernel della macchina ospitante per ottimizzare la resa della virtualizzazione, e per tale motivo potrebbero essere necessari dei **moduli kernel specifici**. Infine, molte CPU oggi offrono un **supporto nativo per la virtualizzazione** per gestire i passaggi da modalità utente a kernel e viceversa sia nella macchina host sia nella macchina guest.

04 - Processi

I primi sistemi elaborativi consentivano l'esecuzione di un solo programma alla volta, che aveva il completo controllo del sistema e accesso a tutte le sue risorse. Oggi è possibile avere più programmi caricati in memoria ed eseguiti in modo concorrente.
Abbiamo visto come un sistema batch esegua lavori. Un sistema **time-sharing** invece esegue programmi utenti o **task**. Questi task, eseguiti anche contemporaneamente, prendono il nome di **processi**.

> **Processo**
>
> Un processo è un'istanza di esecuzione di un programma.

Da sottolineare che un programma di per sé **non è un processo**: un programma infatti è un'entità **passiva**, mentre un processo è un'entità **attiva**. Il programma diventa processo quando l'eseguibile è caricato in memoria.
La struttura di un processo in memoria è suddivisa in più sezioni, come segue:



- la **sezione di testo** contiene il codice eseguibile;

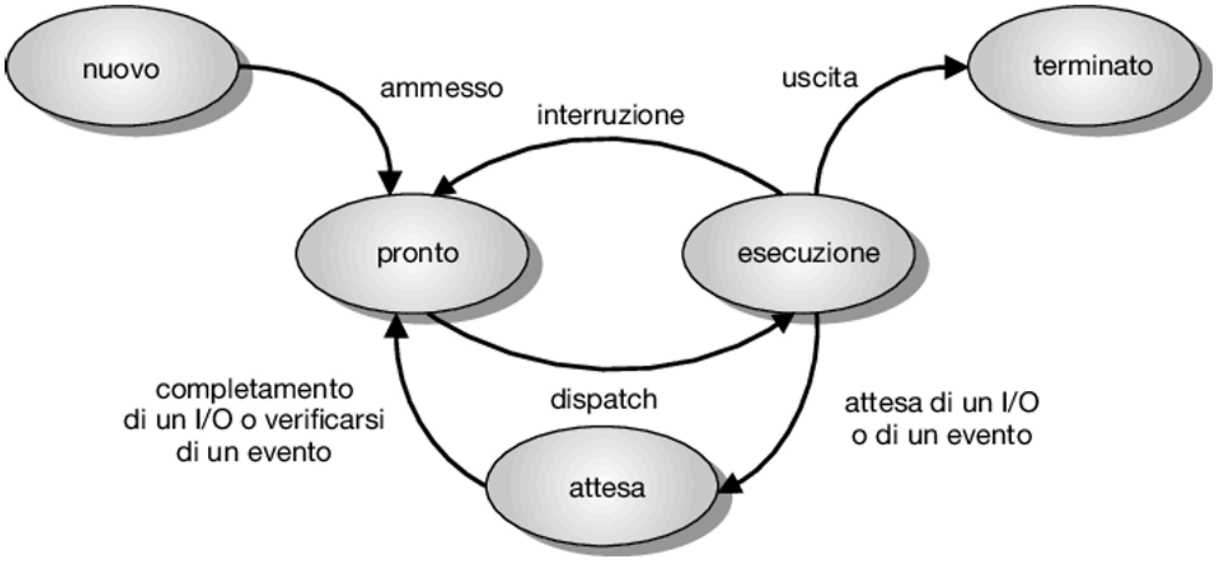
- la **sezione dati** contiene le variabili globali;
- l'**heap** è la memoria allocata dinamicamente durante l'esecuzione del programma;
- lo **stack** è la memoria usata durante le chiamate di funzione (variabili locali, parametri di funzione, indirizzi di ritorno).

Sezione testo e dati hanno dimensione fisse, mentre stack e heap variano durante l'esecuzione del programma. Ogni chiamata di funzione comporta l'inserimento sullo stack di un **record di attivazione**; quando viene eseguita l'istruzione `return`, il record viene rimosso dalla stack. Il sistema operativo deve garantire che stack e heap **non si sovrappongano**.

Stato del processo

Un processo è soggetto a cambiamenti del suo stato, definito in parte dall'attività del processo stesso. Un processo può trovarsi in uno di questi stati:

- **nuovo**: il processo viene creato;
- **in esecuzione** (**running**): le sue istruzioni vengono eseguite. Dallo stato di running si passa agli stati **ready** e **blocked**;
- **in attesa** (**blocked**): il processo attende che si verifichi qualche evento. Se il processo deve interagire con I/O, deve invocare una **syscall** per farsi aiutare dal sistema operativo. Alcune di queste syscall, catalogate come **lente** o **bloccanti**, sono fatte per questo. Prendono il nome di bloccanti perché sono lente rispetto alle altre istruzioni della CPU. Finché il processo non ha interagito con la periferica richiesta, la CPU **rimane ferma**, e l'idea per migliorare il sistema è specificare che il processo è bloccato e riassegnare la CPU ad un altro processo;
- **pronto** (**ready**): il processo attende di essere assegnato ad un'unità di elaborazione. La transizione da uno stato running a uno stato ready è legata ad un meccanismo di **prelazione**. I meccanismi di prelazione forniscono metriche che aiutano ad un processo di riprendersi la CPU quando questi, ad esempio, con una chiamata bloccante ha lasciato la CPU ad un altro processo. Una metrica può essere un'informazione temporale: se il processo non riprende il controllo della CPU dopo t millisecondi, allora la riprende automaticamente. Questo può essere raggiunto usando **interrupt dell'hardware**, mandati in **intervalli scanditi dal clock di sistema**. La CPU, tramite una subroutine, fa transitare allo stato di ready il processo in esecuzione se questo ha superato il suo **quanto di tempo assegnato**;
- **terminato**: il processo ha terminato l'esecuzione.



Ready queue, Scheduling

Lo stato "pronto" india che il processo è pronto ad utilizzare la CPU. In generale i processi pronti vengono inseriti in una coda di processi pronti nota come **ready queue**. La testa di questa coda contiene il puntatore al primo pcb della lista, e ciascun pcb contiene un puntatore al successivo pcb della coda.
Quando un nuovo processo viene collocato nella ready queue, esso attende fino a quando non viene selezionato per essere eseguito. In tal caso si dice che il processo è **dispatched**. Il processo viene selezionato dallo **scheduler della CPU**, che seleziona ripetutamente un processo nella ready queue. Quando il processo è assegnato alla CPU ed è in esecuzione, può accadere uno dei seguenti eventi:

- il processo emette una richiesta I/O. In tal caso, esso viene inserito in una coda di I/O di attesa;
- il processo genera un processo figlio, e ne attende la terminazione;
- il processo viene rimosso a causa di un interrupt, e viene inserito nuovamente nella ready queue.

Al termine dell'esecuzione, il pcb del processo viene deallocato.
Affinché un processo non prenda il controllo della CPU per troppo tempo, un **timer** basato sul clock di sistema viene stabilito. Il clock di sistema invia alla conclusione del timer un segnale di interrupt (detto **clock interrupt**).

> **Clock interrupt**
>
> Il clock interrupt è un segnale di interruzione generato dal clock di sistema.

Blocco di controllo del processo

Ogni processo viene rappresentato nel sistema operativo da un **process control block** (**pcb**), ossia da un **blocco di controllo del processo**. Questo blocco contiene informazioni relative a un processo specifico, tra cui:

- stato del processo;
- numero del processo (PID);
- contatore di programma (virtual PC);
- registri della CPU. Quando un processo viene interrotto, il sistema operativo memorizza lo stato dei suoi registri, così che quando viene ripresa l'esecuzione il sistema è in grado di ripristinare i valori originari;
- limiti di memoria, usati per la gestione degli indirizzi virtuali;
- elenco dei file aperti;
- processi imparentati.

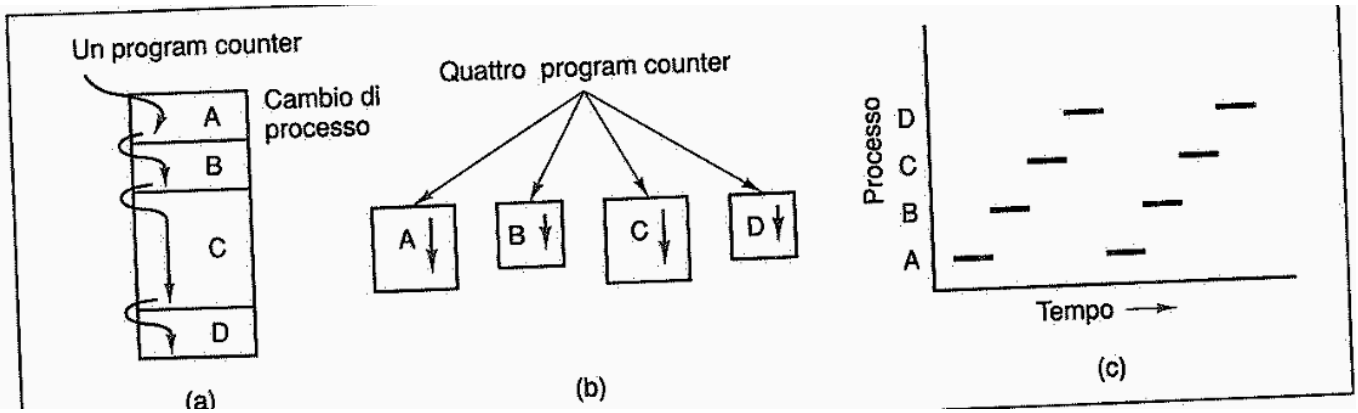
Tutti i PCB vengono memorizzati in una **tabella dei processi**.

Multiprogrammazione

Nel modello a processi il software viene eseguito dal sistema operativo tramite processi sequenziali. Concettualmente, ogni processo è dotato di una **CPU virtuale**, sebbene chiaramente nella realtà la CPU passi in continuazione da un processo all'altro. Il **cambio di contesto** continuo tra i vari processi prende il nome di **multiprogrammazione**.

> **Cambio di contesto**
>
> Il cambio di contesto è il passaggio della CPU a un nuovo processo. Esso implica il salvataggio dello stato del processo attuale e il ripristino dello stato del nuovo processo.

Ogni processo quindi ha un suo flusso di controllo tramite il contatore di programma, pensabile come un **program counter logico**. Ovviamente c'è un solo program counter fisico, pertanto il program counter logico di un processo viene caricato nel program counter fisico quando il processo è in esecuzione. Scaduto il tempo assegnato al processo, il program counter fisico viene memorizzato nel pcb. Pertanto la CPU cambia continuamente contesto, passando da un processo ad un altro:



Per il cambio di contesto risulta necessario gestire correttamente i segnali di interruzione. Vengono effettuate le seguenti operazioni:

- salvataggio del program counter e del program status word nello stack;
- caricamento dell'indirizzo della ISR dal vettore delle interruzioni;
- salvataggio dei registri e impostazione di un nuovo stack per la ISR;
- esecuzione della ISR;
- interrogazione dello scheduler per l'esecuzione del prossimo processo;
- ripristino dello stato del processo a partire dal pcb del processo stesso;
- ripresa del processo scelto.

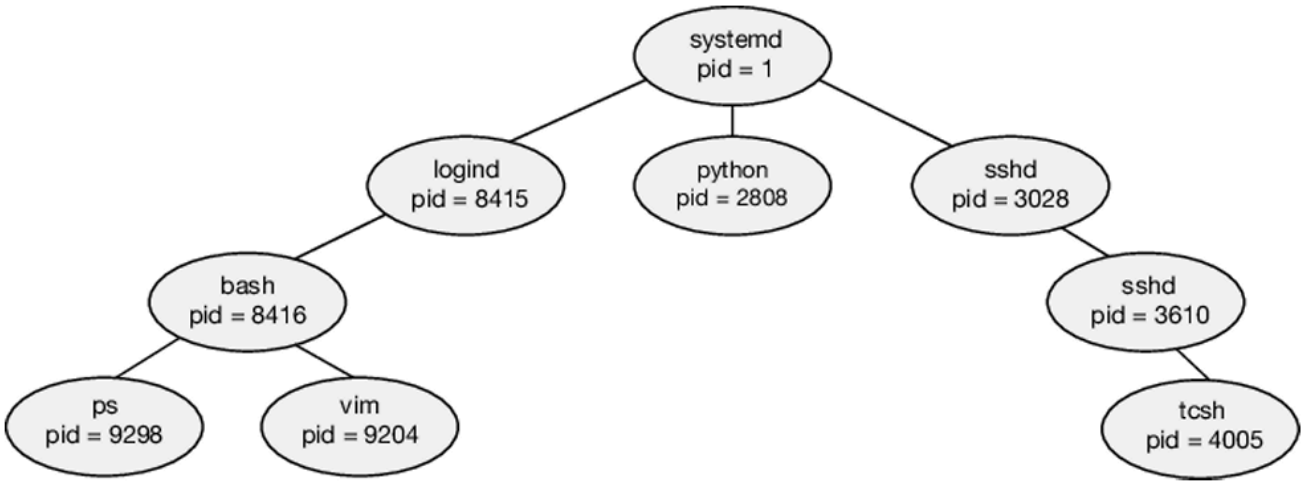
Creazione di un processo

A provocare la creazione di un processo sono generalmente 3 eventi:

1. l'inizializzazione del sistema;
2. l'esecuzione di una system call per la creazione di un processo da parte di un processo in esecuzione;
3. una richiesta da parte dell'utente per la creazione di un nuovo processo.

Generalmente quando un sistema viene lanciato vengono creati diversi processi, alcuni in primo piano e altri in background. I processi eseguiti in background vengono chiamati **demoni** (**daemon**).
Oltre ai processi creati a tempo di avvio, se ne possono creare di nuovi. Spesso è un processo in esecuzione che tramite una chiamata di sistema crea nuovi processi per completare dei lavori. Infine, l'utente può creare un nuovo processo, ad esempio con un doppio clic su sistemi a GUI o utilizzando un terminale e scrivendo il nome di un programma da eseguire.
In sistemi UNIX esiste una sola system call per creare un processo: la chiamata `fork`. La chiamata `fork` crea una copia esatta del processo chiamante, quindi distinguiamo un processo **genitore** e un processo **figlio**. Il processo figlio successivamente esegue una `execve` per cambiare la sua immagine di memoria o eseguire un nuovo programma. Ad esempio, se in `bash` eseguiamo il programma `sort`, allora la shell esegue una `fork`, crea un

processo figlio e tale processo figlio esegue `sort`. Ciò permette al figlio di manipolare i propri descrittori di file dopo `fork` ma prima di `execve`. La chiamata `execve`, invece, **modifica i segmenti codice e dati** così da eseguire un altro programma.
Sistemi UNIX e Linux tengono conto delle parentele tra i processi. Si crea così un **albero dei processi** come quello sottostante:



Su Linux la struttura ad albero è visualizzabile con il comando `pstree`. Per elencare tutti i processi senza visualizzare la struttura ad albero si utilizza `ps -el`.
Su sistemi Windows il meccanismo di creazione del processo consiste nella creazione del nuovo processo e dell'immediata specificazione relativa all'attività da svolgere. La chiamata a sistema per la creazione dei processi è `CreateProcess`. Inoltre, nei sistemi Windows **non vi è un concetto di gerarchia di processi**: ogni processo è uguale. Il solo caso in cui esiste una sorta di gerarchia di processi si verifica quando un processo viene creato e viene gestito dal genitore. In questo caso il genitore riceve un intero noto come **handle** del processo. Generalmente la chiamata di `CreateProcess` restituisce l'handle del nuovo processo. L'handle viene creato con diritto di accesso completo e può essere usato per qualsiasi funzione che prende in input un processo. L'handle può essere passato dal genitore a qualche altro processo invalidando la gerarchia, cosa che in UNIX e Linux è impossibile.
Da notare come il processo padre e il figlio hanno il **proprio spazio degli indirizzi**, pertanto una modifica nello spazio di indirizzo di uno non è visibile all'altro.

Terminazione di un processo

Un processo termina secondo una delle seguenti condizioni:

- normale terminazione del processo;
- terminazione con errore;
- errore **fatale** del processo;
- **uccisione** del processo da parte di un altro processo.

Le prime due terminazioni sono **volontarie**, mentre le ultime due sono **involontarie.** Nei sistemi UNIX e Linux la chiamata per la terminazione dei processi è `exit`, che fornisce un codice di terminazione:

- 0 se tutto è andato a buon fine;
- maggiore di 0 se ci sono stati errori;
- maggiore di 128 se si è verificato un errore fatale.

Sui sistemi Windows il processo viene terminato con `TerminateProcess`.

05 - Threads

Thread

Il thread è un **flusso di esecuzione** all'interno di un processo.

Essi permettono di suddividere l'attività dello stesso processo in mini-flussi tali che non siano in conflitto tra loro.
I thread permettono quindi esecuzione **parallela** tra entità che **condividono tra loro lo spazio di indirizzi del processo**, evitando un'esecuzione sequenziale e quindi rimuovendo la dipendenza tra le attività del processo. Un secondo vantaggio è che sono entità più **leggere**: il processo complesso è suddiviso in attività più semplici da creare e da eseguire. Si ha così un **considerevole guadagno di prestazioni**. Il grande guadagno non si ha se il processo è **CPU-bound**, ma se esso è **I/O-bound** o ha attività che svolgono attività I/O il guadagno è considerevole: si può avere infatti un thread che si occupa dell'esecuzione di operazioni e altri thread che si occupano dell'I/O. In questo modo se viene richiesto I/O l'intero processo non viene bloccato, ma solo quello specifico thread che non può proseguire la sua attività.
Il massimo vantaggio nell'utilizzo di thread si ha in un sistema multicore o multiprocessore.

Multithreading

Il termine **multithreading** è utilizzato per descrivere la situazione in cui ad un solo processo sono associati più thread.

Esempi

Un esempio che dimostra il vantaggio nell'utilizzo di thread è il **word processor** (ad esempio Word). Un word processor esegue fondamentalmente due attività:

- legge l'input dell'utente;
- impagina il contenuto digitato.

In un flusso sequenziale, se prima non ha ricevuto l'input dell'utente il word processor non può impaginare il contenuto. Impaginare il contenuto in realtà può richiedere del tempo. I tempi possono essere ottimizzati se dividiamo il programma in due thread:

- il primo si occupa di leggere il contenuto dell'utente;
- il secondo si occupa dell'impaginazione.

Se ci sono più thread di impaginazione questi possono impaginare il contenuto in maniera parallela. Il guadagno risiede se il thread è **passivo**.

> **Thread passivo**
>
> Un thread si dice passivo se lavora in background.

Un altro thread potrebbe occuparsi del salvataggio automatico. Notiamo che nel momento in cui non usassimo i thread, il processo si blocca interamente nel caso in cui abbiamo il salvataggio automatico per scrivere sul disco.
Un altro esempio potrebbe essere un processo per un web server. Nei server di siti web una cache memorizza le pagine visualizzate più recentemente. Generalmente un web server quando riceve una richiesta per una pagina web, il server controlla nella cache se la pagina è memorizzata, altrimenti controlla la memoria del server. Supponiamo l'attività sia svolta da un solo flusso di esecuzione. La prima cosa che viene eseguita è la ricezione della richiesta di pagina. Successivamente, avviene la ricerca in cache: durante tale operazione nessun'altra attività può essere eseguita. Se la pagina richiesta non è in cache, allora si va avanti procedendo con una ricerca nel main storage.
Si può utilizzare l'attività in thread:

- un **thread dispatcher** gestisce le richieste ricevute dal web server;
- le richieste gestite dal dispatcher vanno inviate ad un **thread worker** (così il dispatcher torna disponibile), che elabora la richiesta cercando la web page richiesta;
- una volta che l'informazione è trovata dal thread worker, il worker restituisce la pagina al dispatcher.

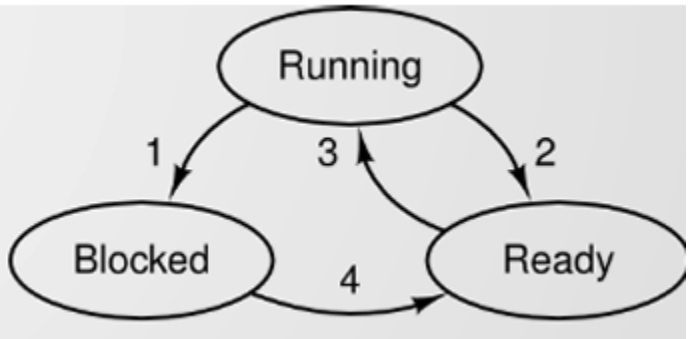
Notare che le prestazioni sono considerevolmente migliori utilizzando più thread worker, rispetto ad eseguire tali operazioni in maniera sequenziale.

Modello a thread

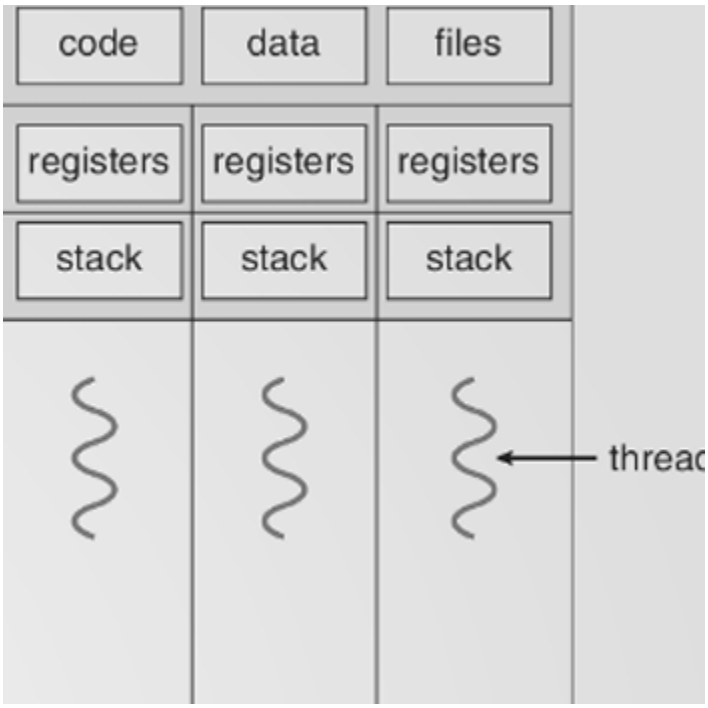
Di default ogni processo viene istanziato con un unico thread. Successivamente se ne possono aprire altri per attività diverse.
A differenza dei processi, ogni thread è caratterizzato da:

- un program counter;
- registri;
- stack;
- stato, che può essere:
  - **blocked**, cioè **in attesa**;
  - **ready**: in attesa di essere schedulato;

- **running**: occupa un'unità di elaborazione.



I thread condividono tra loro lo spazio di indirizzamento, l'heap, la memoria globale, il segmento del codice e i file aperti. Il sistema operativo (o la libreria dei thread, vedremo successivamente il modello utente) alloca e traccia le regioni per evitare sovrapposizioni. Dato tuttavia che lo spazio di indirizzamento è condiviso, **ogni thread può operare su zone di memoria usate da altri thread**, in quanto **non esiste alcun meccanismo standard di protezione delle aree di memoria dei singoli thread**. Quindi i thread sono flussi di esecuzione **indipendenti**, ma **non sono isolati**.



Scheduling

Lo scheduling dei thread segue logiche simili a quelle dei processi, poiché ogni thread viene considerato un'unità indipendente di esecuzione. Se un thread è bloccato, gli altri thread possono continuare l'esecuzione in quanto indipendenti.
Possiamo immaginare che lo scheduler del sistema operativo selezioni prima un processo, e poi un thread.

Efficienza del context switch

Il vero guadagno di performance nell'utilizzo del modello a thread risiede nel **cambio di contesto**. Generalmente il cambio di contesto tra un thread e un altro è molto più veloce del cambio di contesto tra processi, per il fatto che l'**MMU non dev'essere riprogrammata** in quanto il processo è sempre lo stesso: non abbiamo bisogno di aggiornare nuovamente i registri relativi ai limiti dello spazio di memoria associato al processo, perché il processo è sempre lo stesso.
Chiaramente cambiare thread di un processo con il thread di un altro processo **non porta alcun vantaggio** perché il cambio dei thread in questo caso richiede il cambio di contesto di processo.

Operazioni su thread

In un sistema multithreading i processi iniziano con un singolo thread, che ha a sua volta un compito di creare nuovi thread attraverso specifiche procedure.
In particolare, la procedura di libreria `thread_create` permette la creazione di un nuovo thread e specifica la procedura che deve eseguire (si tratta di un **parametro necessario**). Ogni processo parte con un thread, che ha il compito di creare nuovi thread. Usualmente tra i thread non c'è una gerarchia, quindi i thread sono considerati uno uguale all'altro a livello gerarchico.
Un thread può terminare usando la chiamata alla procedura `thread_exit`. A differenza della chiamata `exit`, che termina il processo, gli altri thread continueranno ad esistere.
Spesso e volentieri è necessario che un thread si sincronizzi con la fine di un altro thread, rimanendo quindi in attesa della sua fine. Questo è possibile utilizzando la chiamata a funzione `thread_join`. Se un thread A chiama `thread_join` per sincronizzarsi con il thread B, rimarrà in uno stato **blocked** fino a quando B non termina la sua esecuzione. Alla terminazione di B, il thread chiamante viene spostato nella ready queue dei thread.
Infine, `thread_yield` è una funzione tramite la quale un thread rilascia volontariamente la CPU. In questo caso **non si hanno interruzioni di clock**. Da qui l'idea che il modello a thread è un **modello collaborativo**, a differenza del modello a processi, che è **competitivo** (si ha "competizione" per avere la CPU).

Programmazione multicore

Avere un programma con più thread permette di sfruttare le potenzialità di un calcolatore multicore o multiprocessore, usufruendo eventualmente anche del meccanismo di multithreading. Prima dell'utilizzo dei thread, sistemi multicore necessitavano tanti processi che occupassero tutti i core. Ora ciò è possibile anche avendo un solo processo costituito da molti thread.
Su un sistema multicore, inoltre, è possibile raggiungere il **parallelismo**, garantendo prestazioni migliori in base al bilanciamento di lavoro dei thread, la sincronizzazione di questi e il tipo di operazioni eseguite. Più precisamente:

- nei sistemi a core singola i thread condividono il tempo di esecuzione in modo interlacciato. Si tratta quindi di un meccanismo di **time-sharing**, dove il processore **dà l'idea di eseguire i thread in parallelo**, passando nella realtà da un thread ad un altro;
- nei sistemi multicore i thread vengono **realmente eseguiti in parallelo**. Se ci sono molti thread, tuttavia, il sistema operativo userà comunque dei meccanismi di time-sharing per alternare tra i thread (con "molti thread" si intende chiaramente il fatto che il numero di thread è superiore rispetto al numero di unità di elaborazione).

Sviluppare programmi che sfruttino le architetture multi-core tuttavia non è semplice. Risulta necessario rispettare alcuni **principi** per la progettazione di tali programmi.

Principio di separazione dei task

Abbiamo detto che i thread sono flussi di esecuzione di un processo per eseguire attività diverse che non agiscono in conflittualità affinché agiscano in parallelo. Bisogna sapere allora **individuare delle aree per delle attività separate e simultanee**.

Principio di bilanciamento

Le aree separate per lo svolgimento delle attività devono essere identificate in maniera tale da essere **bilanciate** a livello di carico di lavoro.

Principio di suddivisione dei dati

Creare delle attività separate, distinte ma che alla fine lavorano su gran parte delle stesse variabili richiede una **suddivisione dei dati** affinché i task possano essere eseguiti su core separati.

Dipendenze dei dati

Nell'analisi dei dati, risulta necessario individuare le **dipendenze dei dati**: se esse sono presenti, le attività devono essere sincronizzate tra loro. Ciò introduce costi aggiuntivi per garantire accessi corretti.

Test e debugging

Se un processo viene diviso in diversi thread, ci possono essere diversi flussi di esecuzione. Questo comporta complicazioni nel momento di test e debugging.

Tipologie di thread

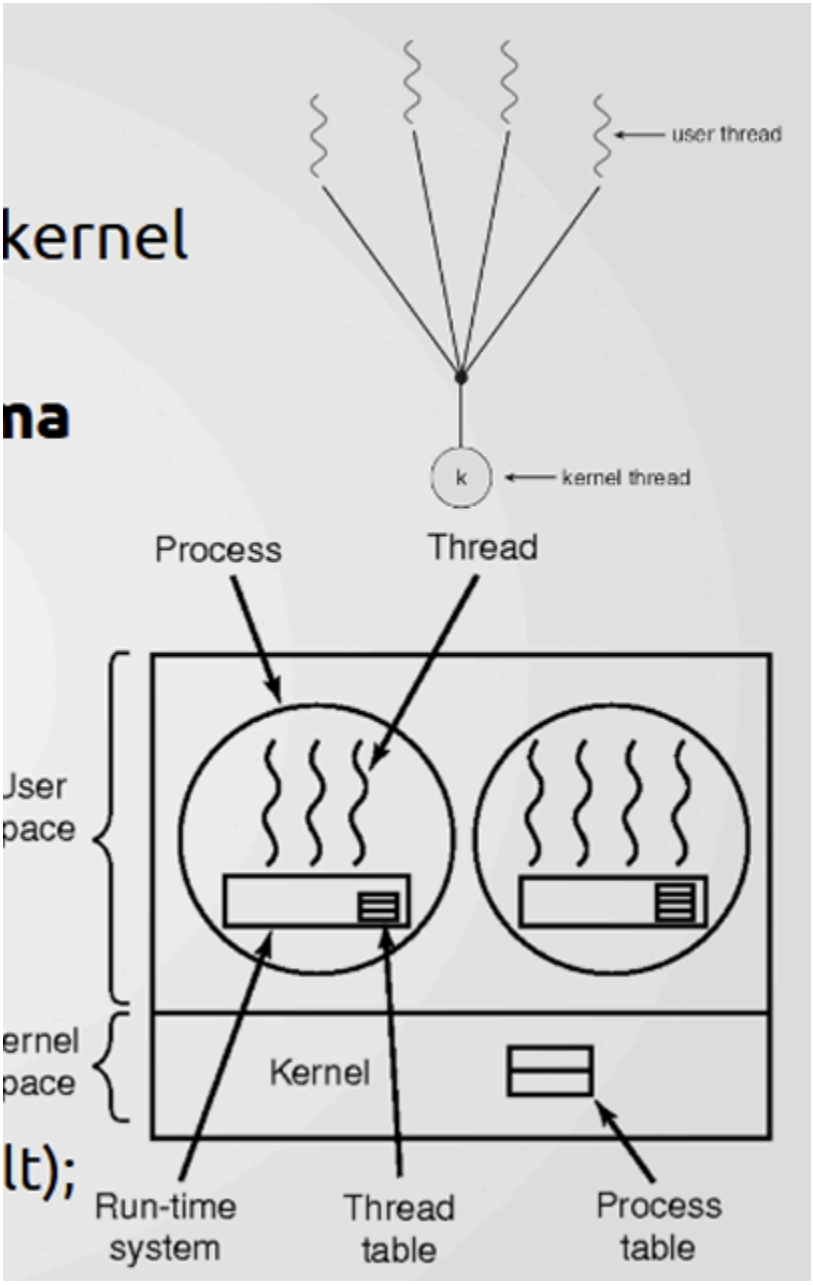
Possiamo implementare i thread in maniere differenti:

- thread a livello **kernel**;

- thread a livello **utente**;
- thread con modalità **ibrida**.

Thread a livello utente

In questo modello i thread vengono implementati a livello utente. Inizialmente **non tutti i sistemi operativi supportavano i thread**, pertanto si cercò di aggirare tale limitazione. Il modello del thread a livello utente è anche detto modello **1-a-molti** in quanto il sistema operativo crea un solo flusso di esecuzione, ma il tempo di esecuzione per i thread viene **artificialmente creato** con un meccanismo di scheduling: il quanto di tempo assegnato al programma viene suddiviso in modo da eseguire i thread in tempi ritagliati. Il sistema operativo quindi fa corrispondere ad un thread del kernel più thread dell'utente.
In assenza di un meccanismo supportato dal sistema operativo per lo switch dei thread, le librerie utente che implementano un **sistema run-time** gestiscono una **tabella dei thread**, memorizzata nel processo stesso. Per fare lo switch tra i thread del processo, si utilizza la funzione `thread_yield`, memorizzando le informazioni del thread in tabella ed effettuando il cambio. In questo modo il dispatching del thread avviene in modalità utente, riducendo l'overhead causato da un'eventuale chiamata di sistema.
Un problema che tuttavia si presenta in tale modello a thread risiede nell'**incapacità del sistema operativo di riconoscere i differenti thread**. Quando un thread esegue una syscall bloccante, transiterà l'**intero processo** nello stato **blocked**. Tutti i thread quindi verranno bloccati. Per mitigare ciò, si può usare la syscall `select`, la quale verifica se la syscall è bloccante; in tal caso, invoca una `thread_yield` per passare ad un altro thread. Se la CPU ritorna a quel thread, quest'ultimo esegue una `select` per verificare se la syscall è ancora bloccante. Questo meccanismo tuttavia non funziona con tutte le syscall.
Un altro caso in cui si verifica questo problema è quando si usa la paginazione e si verifica un `page_fault`: la CPU non trova il dato richiesto e dà al sistema operativo il compito di recuperarla. Ciò comporta un blocco del processo, che non sa tuttavia di esserne responsabile e quindi non può controllare a priori l'operazione tramite una `select`.
Dopo l'esecuzione del `thread_yield`, il sistema runtime deve eseguire una scelta per il successivo thread. Ciò permette l'implementazione di un meccanismo di **scheduling personalizzato**, definendo opportune euristiche per la selezione del thread più "appropriato".

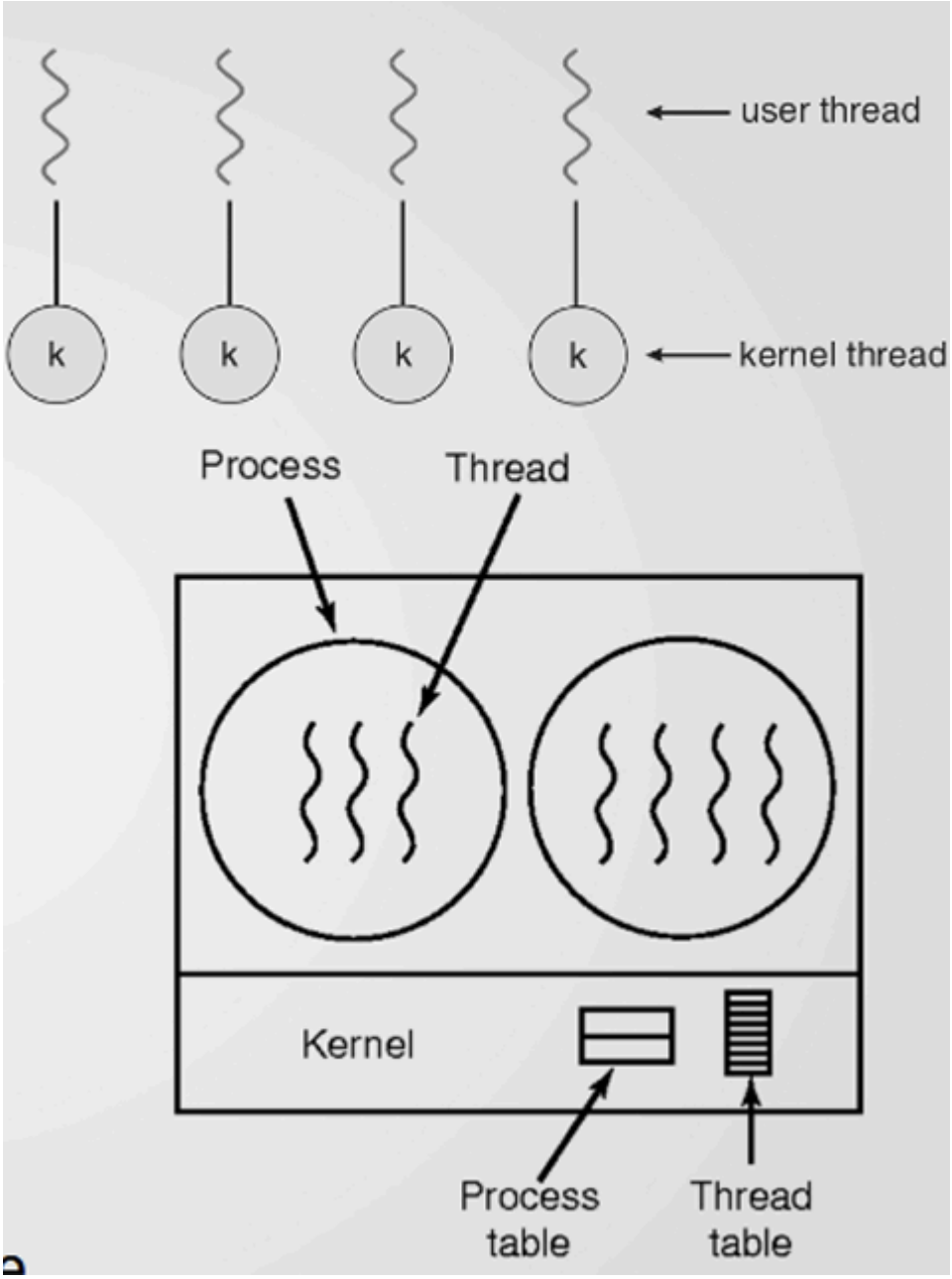


Thread a livello kernel

Se il sistema operativo supporta il meccanismo dei thread e quindi ha **apposite chiamate di sistema per gestire i thread** è possibile implementare i thread a livello kernel. La tabella dei thread quindi è presente all'interno del kernel e (spesso) fusa con quella dei processi. Il modello in questo caso prende il nome di **modello 1-a-1** in quanto fa corrispondere ad un thread dell'utente un thread del kernel.
Il vantaggio è che se un thread esegue una chiamata bloccante, essa non blocca l'intero processo, ma solo il singolo thread, in quanto stavolta il sistema operativo **sa dell'esistenza dei thread**. Abbiamo tuttavia due svantaggi:

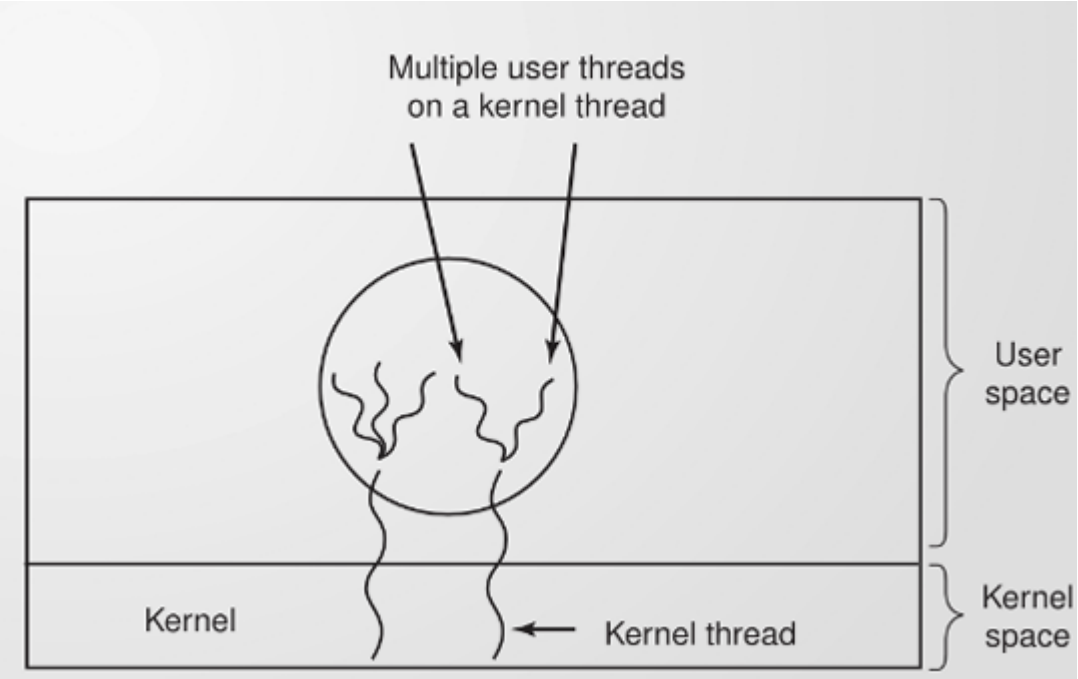
- il **context switch** è più **lento**, in quanto ogni thread è gestito dal kernel, richiedendo così una syscall, che è più costosa rispetto ai cambiamenti di contesto tra thread a livello utente;

- **creare e distruggere thread richiede syscall**, e ciò comporta **maggiori costi** in quanto le operazioni sono **più lente** e **più costose**. Per ridurre l'overhead, i thread kernel possono essere riutilizzati, utilizzando tecniche quali **riciclo dei thread**, ma ciò non elimina i costi associati alla gestione a livello kernel. Inoltre, creare e distruggere thread continuamente non è buona prassi.



Modello ibrido

I vantaggi dei modelli descritti precedenti possono essere combinati in un **modello ibrido** detto **modello molti-a-molti**. L'idea è quella di scegliere quali dei task del programma vengono gestiti come thread a livello kernel (ad esempio, possono richiedere funzionalità di gestione avanzata) e quali no. Si può quindi usare i thread del kernel ed effettuare un **multiplexing dei thread a livello utente** su alcuni thread del kernel, come segue:



Su un thread kernel ci sono quindi più thread utente gestiti con le modalità del modello 1-a-molti. Questo approccio permette di combinare i vantaggi dei modelli visti precedentemente in una soluzione flessibile per il programmatore.

Thread nei sistemi operativi

Ormai quasi tutti i sistemi operativi supportano i thread a livello kernel, permettendo l'implementazione del modello ibrido. Apposite librerie, quali **GNU Portable Threads** per **UNIX**, **Fibers** su **Windows** e **Green Threads** su **Solaris**, garantiscono il supporto per i thread a livello utente.
Ogni sistema operativo chiaramente risponde con delle syscall specifiche all'accesso ai thread. Per garantire portabilità, esiste una libreria chiamata `pthreads` (**POSIX threads**) che astrae le differenze tra i vari sistemi operativi, garantendo l'esecuzione della corretta syscall.
Windows non supporta tuttavia pthreads in maniera del tutto nativa. Il progetto `pthreads4w` garantisce allora portabilità con i sistemi Windows, permettendo anche di eseguire porting di software UNIX che utilizza pthreads con piccole (o anche inesistenti) modifiche. Anche **Mingw-w64** contiene un'implementazione di pthreads per Windows, ossia `winpthreads`, che utilizza syscall più native rispetto a `pthreads4w`.

06 - Interprocess Communication

Spesso i processi necessitano di comunicare tra loro. Un esempio può essere utilizzando il **pipe in shell**.
Il carattere pipe (`|`) è una funzionalità a riga di comando dei sistemi Linux e UNIX. Tramite la pipe, l'output di un comando viene reindirizzato e inviato ad un altro comando. Ad esempio:

```
~:$ ls | grep ".cpp"
```

stampa solo i file contenenti `.cpp`. Quello che avviene è che l'output del comando `ls` viene reindirizzato come input per il comando `grep`.
Il meccanismo di pipelining, che è un meccanismo **unidirezionale**, all'interno di shell è un esempio di comunicazione tra processi, che semplifica tuttavia alcuni aspetti:

- il mezzo in cui i processi comunicano, che implicitamente utilizza i canali I/O `stdin`, `stdout` e `stderr`, caratteristici di ogni processo;
- il coordinamento e la sincronizzazione tra i processi, in quanto `grep` deve attendere che `ls` restituisca l'output. Inoltre, il processo di `grep` potrebbe essere occupato, quindi il processo di `ls` deve attendere che `grep` li riceva prima di terminare.

A livello implementativo i canali I/O hanno un buffer di dimensione limitata in cui i dati vengono memorizzati, quindi il processo `ls` può liberarsi facilmente. Tuttavia ciò non elimina la necessità di coordinamento, perché non sempre tutti i dati possono essere memorizzati nel buffer.
Il meccanismo del pipelining è più semplice in quanto, come anticipato, è **unidirezionale**.
Dobbiamo gestire tre possibili problemi:

1. come passare informazioni da un processo ad un altro;
2. come evitare che i processi si intralcino durante **attività critiche**;
3. come mettere in sequenza i processi in maniera adeguata.

Nei sistemi operativi i processi che lavorano insieme possono condividere una **memoria comune** su cui possono leggere e scrivere. Lo spazio condiviso può essere in memoria principale, gestito ad esempio come struttura dati, oppure un file condiviso. In generale, qualunque sia il tipo di memoria condivisa, **si verifica un problema di concorrenza**: due o più processi possono accedere contemporaneamente alla stessa area di memoria. Per vedere praticamente qual è il problema, consideriamo un esempio.

Esempio: spooler di stampante

Quando un processo vuole stampare un file, aggiunge il nome del file in una directory detta **directory di spool** o **spooler directory**; il **daemon** della stampante controlla periodicamente se ci sono file da stampare, e in tal caso li stampa e li rimuove dalla spooler directory. Supponiamo di numerare gli elementi presenti nella directory con $0, 1, 2, \ldots$ e supponiamo ci siano due variabili condivise:

- `out`, che punta al prossimo file da stampare;
- `in`, che punta al prossimo posto libero in directory.

Supponiamo il processo A e il processo B vogliano accodare un file per la stampa. Potrebbe succedere che il processo A legge `in` e ne memorizza il valore in una variabile, ad esempio `primo_slot_libero`; in quell'istante arriva un'interrupt dal clock di sistema, quindi la CPU passa al processo B, che legge `in` e ottiene 7. Anche B memorizza in una sua variabile `primo_slot_libero` e continua la sua esecuzione memorizzando il nome del suo file nello slot 7 e aggiorna `in` a 8. Quando il processo A torna in stato di esecuzione, accede alla sua variabile `primo_slot_libero` e vi trova 7, pertanto se scrive il nome del suo file cancella il nome assegnato dal processo B. Quindi B non otterrà mai niente in uscita, sebbene la spooler directory sia consistente.

Race condition

Situazioni in cui il risultato finale dipende dall'ordine in cui più processi vengono eseguiti vengono dette **race condition** o **corse critiche**.

> **Race condition**
>
> La race condition è un fenomeno che si presenta nei sistemi concorrenti quando, in presenza di una sequenza di processi multipli, il risultato finale dell'esecuzione dei processi dipende dalla **temporizzazione** o dalla **sequenza** con cui questi vengono eseguiti.

Nell'esempio precedente abbiamo mostrato un caso semplice con la condivisione della variabile `in`. Le situazioni in realtà possono anche essere più complicate:

- una struttura dati complessa potrebbe essere condivisa tra processi;
- situazioni simili potrebbero sorgere **all'interno del kernel**.

Evitare le corse critiche

La chiave per evitare le race condition risiede chiaramente nell'evitare che più di un processo alla volta legga o scriva i dati condivisi contemporaneamente. Tale problema può essere formulata in maniera più astratta. Un processo generico può eseguire operazioni che non

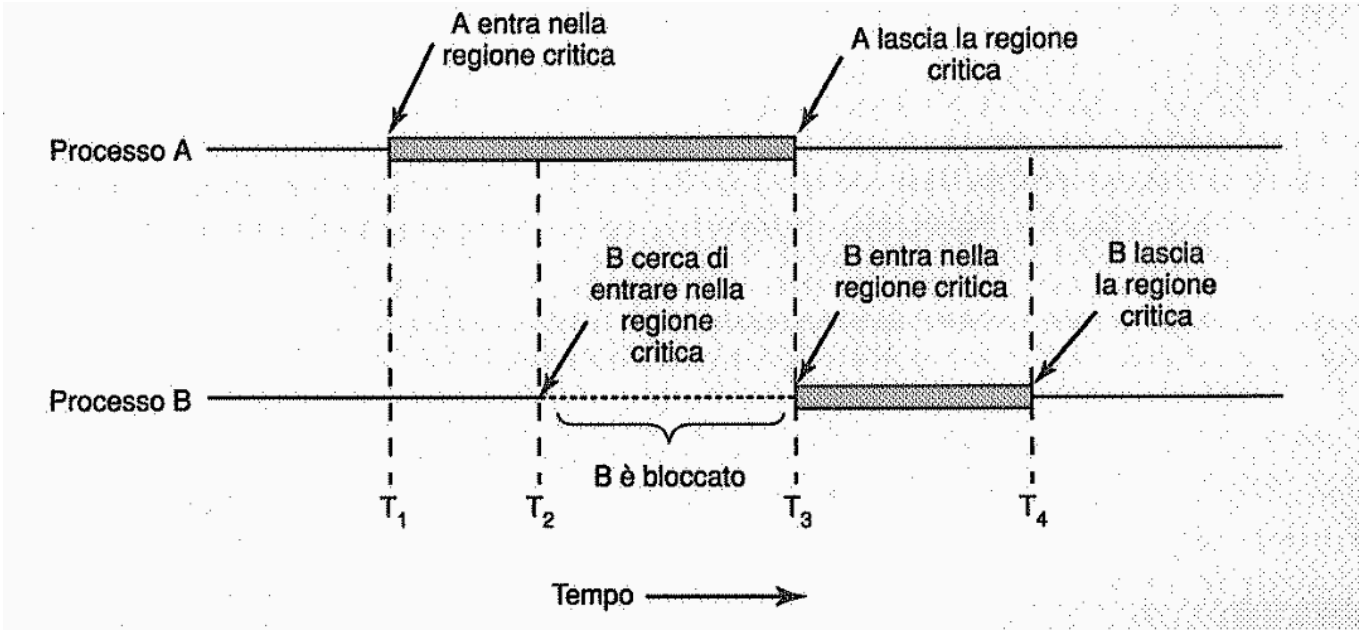
portino a corse critiche, ma qualche volta può svolgere azioni che possono portare a race condition.

> **Sezione critica**
>
> La parte di programma in cui il processo accede alla memoria condivisa prende il nome di **regione critica** o **sezione critica**.

In pratica abbiamo bisogno di un meccanismo di **mutua esclusione** che ci assicuri che se un processo usa una sezione di memoria condivisa, gli altri processi non potranno fare la stessa cosa; in altre parole, organizziamo l'esecuzione in modo tale che due programmi non possano mai essere nelle loro sezioni critiche nello stesso istante. Ciò tuttavia non è sufficiente per garantire la cooperazione e l'efficienza nell'utilizzo di memoria condivisa. Risulta necessario:

1. che due processi non si trovino contemporaneamente nelle sezioni critiche;
2. che non si facciano ipotesi sulle velocità e sul numero di CPU;
3. che nessun processo in esecuzione blocchi altri processi quando non si trova nella sua sezione critica;
4. che nessun processo attenda indefinitamente per entrare in una sezione critica.

In pratica il meccanismo che dobbiamo implementare è simile al seguente:



Indicheremo con `enter_region()` e con `leave_region()` le sezioni di codice di un processo in cui si hanno le sezioni critiche.

Disabilitazione degli interrupt

La soluzione più semplice è permettere a ciascun processo di **disabilitare le interruzioni** nel momento in cui il processo esegue `enter_region()` (ossia entra in regione critica) e riabilitarle quando esegue `leave_region()` (ossia esce in regione critica). Poiché le interruzioni sono disabilitate non può arrivare nessun clock interrupt, interrompendo il meccanismo di prelazione in quanto la CPU passa da un processo all'altro solo come il risultato di un'interruzione da clock o da altre periferiche.
L'approccio non è ideale per due aspetti:

- dare il potere ad un processo di disabilitare le interruzioni potrebbe portare alla **fine del sistema se il processo non le riabilitasse più**;
- in un sistema multi-core disabilitare le interrupt ha effetto **esclusivamente sulla CPU che esegue** `disable`, mentre le altre continuerebbero l'esecuzione e potrebbero entrare in una regione critica.

D'altro canto, spesso è conveniente che il kernel disabiliti le interruzioni per poche istruzioni mentre aggiorna variabili o liste. Se si verifica un'interruzione mentre, ad esempio, la lista dei processi pronti è in uno stato inconsistente, potrebbero verificarsi race condition.

Variabile di lock

Un secondo approccio consiste nell'introduzione di una variabile condivisa di **lock** con valore iniziale 0. Se un processo vuole accedere ad una sezione critica, prima controlla la variabile di lock:

- se la variabile di lock è pari a 0, allora imposta il valore della variabile di lock a 1 e entra nella sezione critica;
- se la variabile di lock è pari ad 1, allora **attende che diventi** 0.

In sostanza, se la variabile di lock è pari ad 1, significa che qualche processo sta eseguendo una sezione critica. Alla fine dell'esecuzione di tale sezione critica, la variabile di lock sarà posta a 0.
Anche questo approccio, tuttavia, presenta problematiche:

- la variabile di lock è condivisa, pertanto **può dare vita a race condition**. Può accadere (come nell'esempio della spooler directory) che un processo legga la variabile di lock, impostata a 0, e prima che possa metterla a 1 un altro processo viene mandato in esecuzione, impostando il lock a 1. Al ritorno dell'esecuzione del primo processo, questo eseguirà la sezione critica del suo codice. Si potrebbe pensare ad un secondo controllo prima della scrittura nella variabile, ma questo non funziona, perché si potrebbe avere una race condition se il secondo processo modifica la variabile dopo che il primo ha effettuato la seconda lettura.

- il meccanismo di controllo della variabile di lock è spesso implementato con **busy-waiting**, ossia con test continuo della variabile.

> **Busy waiting**
>
> L'**attesa attiva** (o **busy waiting**) è il testare continuamente che una certa condizione venga verificata.

Alternanza stretta

Un terzo approccio consiste in un'**alternanza stretta** dell'esecuzione dei processi. Si utilizza una variabile condivisa `turn` per **memorizzare il PID del processo autorizzato ad accedere alla sezione critica**:

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

function enter_region(int process)
    while(turn != process) do nothing; // attende che turn sia pari a
"process"

function leave_region(int process)
    turn = 1 - process
```

In questo caso si assume che il numero di processi sia pari a 2. Quindi i PID saranno 0 e 1. Allora:

- `enter_region(int process)` verifica il valore di `turn`: se esso è pari al PID del processo che esegue il codice, allora si può accedere alla sezione critica. Altrimenti, il processo entra in **busy waiting**;
- `leave_region(int process)` aggiorna il valore di `turn` all'uscita della regione critica del processo.

L'approccio proposto evita qualsiasi race condition, ma **non è candidato alla soluzione del problema** in quanto viola la terza condizione illustrata precedentemente, ossia che nessun processo in esecuzione blocchi altri processi quando non si trova nella sua sezione critica.

Soluzione di Peterson

Nel 1981 G.L. Peterson ideò un algoritmo semplice per garantire la mutua esclusione e che utilizza due variabili condivise:

- una variabile `turn` per indicare quale processo può accedere alla sezione critica;
- un vettore `interested[N]`, dove `N` indica il numero di processi. Ogni elemento del vettore è pari ad 1 se il processo corrispondente allo slot del vettore è interessato ad entrare nella sezione critica. Altrimenti, lo slot è pari a `0`.

Lo pseudo-codice della soluzione di Peterson è il seguente:

```
int N = 2 // numero processi
int turn
int interested[N]

function enter_region(int process) {
    int other; // numero dell'altro processo
    other = 1 - process;
    interested[process] = 1;
    turn = process;
    while(turn == process and interested[other] == 1) do nothing;
}

function leave_region(int process) {
    interested[process] = 0;
}
```

Inizialmente nessun processo è in sezione critica. Il processo 0 chiama la procedura `enter_region()` indicando il suo interesse per entrare nella regione critica, impostando il valore nel vettore a 1 e ponendo `turn` uguale a 0. Poiché 1 non è interessato, il processo 0 termina immediatamente. Se il processo 1 chiamasse ora `enter_region()`, allora rimarrebbe in attesa del ciclo `while`: deve verificarsi, infatti, che `interested[0] = 0`, cosa che accade solo se il processo 0 chiama `leave_region()`.
Se entrambi i processi chiamano la procedura `enter_region()` quasi contemporaneamente, entrambi memorizzeranno il proprio PID in `turn` (chiaramente solo il secondo valore conta: il primo viene sovrascritto). Ad esempio, se `turn = 1`, quando entrambi i processi arrivano all'istruzione `while`, il processo 0 non esegue alcuna iterazione: infatti `turn = 1`. Il processo 1 invece cicla e non entra in sezione critica fino a quando 0 non invoca `leave_region()`.
L'unico problema che ha questa soluzione, oltre all'introduzione del meccanismo di busy waiting, è che si possono verificare delle problematiche su sistemi multi-core a causa del **riordinamento degli accessi in memoria centrale**, cosa che accade per migliorare l'efficienza durante l'esecuzione. I processori moderni infatti possono in qualche modo forzare un ordinamento nello stream di accessi in memoria, tipicamente attraverso un istruzione di

memory barrier. Da notare che i riordinamenti degli accessi in memoria accadono anche in processori che **non riordinano istruzioni**.

07 - TSL, Sleep-Awake, Semafori

Le architetture più recenti, soprattutto multi-core, risolvono il problema della concorrenza con una variabile di lock gestita via hardware. L'idea è fornire istruzioni specializzate che rendono possibile la corretta gestione della variabile di lock.

Istruzione TSL

L'istruzione **TSL** (**Test and Set Lock**) permette contestualmente in un'unica operazione in modo atomico di controllare il valore di una variabile di lock e contemporaneamente di impostare un nuovo valore.
L'istruzione TSL fa riferimento ad una **locazione di memoria** (`lock`) e un registro, e ha una sintassi del tipo:

```
TSL registro, lock
```

Le azioni sono due:

- ricopia il valore della variabile `lock` in `registro` ;
- imposta il valore della variabile `lock` ad un valore predefinito **diverso da** 0 (generalmente 1).

Questo equivale ad eseguire due operazioni, ossia una operazione `MOV registro, lock` e poi `MOV lock, 1` . Ci sono tuttavia alcune osservazioni da fare:

- ci sono due operazioni in questo caso, mentre `TSL` è un'unica istruzione. Le due azioni **non saranno mai scisse**. Questo garantisce un'**atomicità** nei confronti della prelazione: o la prelazione arriva prima di `TSL` o arriva dopo;
- in sistemi multi-core se abbiamo due core distinti C_1, C_2 con due processi distinti P_1, P_2, se supponiamo che i processi eseguono una `TSL` sulla stessa variabile di lock, allora fondamentalmente quello che succede è che nell'accedere nella RAM bisogna inserire delle richieste di fetch e di store per recuperare e modificare il valore nell'indirizzo di memoria di `lock` . Sul bus per l'accesso alla memoria, le operazioni della TSL verranno accodate. Se gli accodamenti delle operazioni della TSL arrivano in modo "mischiato", ossia dal processo di un core arriva `MOV registro, lock` , poi in coda si ha `MOV registro, lock` dell'altro processo, si presenta un problema. L'istruzione `TSL` tuttavia garantisce che il **bus di memoria viene bloccato a tutti gli altri core**. Potenzialmente `TSL` può avere tempi morti, ma si tratta di un tempo infinitesimale che non provoca veri e propri ritardi, e anche se succedesse garantisce tuttavia **atomicità**.

Pur trattandosi di un'operazione hardware, qualunque processo senza alcuna syscall può eseguire la `TSL`. Di suo la `TSL` **non implementa alcun blocco o controllo**: essa **modifica solamente un registro**. Essa va implementata all'interno di un codice come segue nell'esempio in pseudo-assembly:

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

leave_region
    MOV LOCK, #0
    RET
```

Questo meccanismo è un **busy-waiting**, perché all'interno di `REGISTER` se si avrà un valore pari ad 1 si esegue un jump alla funzione `enter_region`, quindi cicla mentre lock è diverso da 0.
Non tutte le architetture supportano TSL, ma sono presenti istruzioni equivalenti, come `XCHG`, che si trova nelle architetture Intel X86. La semantica `XCHG` tuttavia è leggermente diversa. Prende tipicamente un registro e una variabile di lock, ma l'istruzione è più semplice in quanto scambia i valori. Lo pseudo-codice precedente quindi diventa:

```
enter-region:
    MOV register, 1
    XCHG register, lock
    ...
```

Anche `XCHG` gode delle proprietà di atomicità.

Sleep e Awake

L'unico problema da risolvere adesso è il **busy-waiting**. Tutte le soluzioni viste infatti usano uno **spin lock**.

> **Spin lock**
>
> Uno **spin lock** è un lock che utilizza il meccanismo di busy waiting.

Il problema del busy waiting è che ci possono essere processi più importanti degli altri. I sistemi operativi reali permettono di assegnare **priorità** a processi che permettono al sistema operativo di garantire precedenza di risorse. Supponiamo di avere 3 processi P_h, P_m, P_l, con priorità rispettivamente alta (high), media (medium) e bassa (low). Supponiamo i processi utilizzino una variabile di lock implementata con meccanismo busy-waiting. Il meccanismo di priorità prevede (nella sua strategia più semplice) di considerare tutti i processi attivi e scegliere quello con priorità più alta, e più precisamente finché P_h non viene bloccato, la CPU viene impegnata per P_h. P_l quindi può usare la CPU quando gli altri due processi sono bloccati. Supponiamo in questo scenario che si utilizzi la priorità del processo come regola di scheduling: se P_h è nello stato "ready", allora viene mandato in esecuzione. Supponiamo inoltre che P_h e P_l utilizzano la variabile di lock per gestire una struttura dati condivisa e che P_l sia il primo ad accedere alla variabile di lock. P_h comincerà la sua attesa attiva, ma il busy-waiting **non terminerà mai**, in quanto il sistema operativo riconosce che il processo è in esecuzione. Poiché le regole di scheduling impongono che il processo a priorità più alta sia quello che deve andare in esecuzione, si ha uno stallo del sistema. Questa situazione viene chiamata **problema dell'inversione delle priorità** (**priority inversion problem**).
Il sistema operativo deve quindi aiutare a gestire le attese passive. Risulta necessario introdurre delle chiamate di sistema per questo scopo: **sleep** e **wakeup**. La prima fa **addormentare il thread o il processo chiamante**, mentre la wakeup **sblocca il processo o il thread dallo stato di sleep**.
Proviamo ad applicarle ad un problema classico dell'IPC: il problema del **produttore-consumatore**. (breve parentesi: il problema del produttore-consumatore appartiene ad una classe di problemi su cui vengono applicate nuove primitive di sistema quando queste devono essere introdotte).

Produttore-consumatore

Supponiamo di avere un processo produttore e un processo consumatore, con un buffer **condiviso** su cui possono essere inseriti elementi (**item**). Il produttore produce item a cui è interessato il consumatore (**non è di importanza la natura dell'item prodotto**). Quando il produttore produce l'item, lo inserisce nel buffer (che può essere una coda, una stack o un albero con priorità. Nella versione classica del problema il buffer è **limitato**); il consumatore durante la sua esecuzione consuma un prodotto nel buffer.
Ci sono situazioni in cui sia il consumatore sia il produttore possono bloccarsi (che modella il problema della **sincronizzazione**). Tali situazioni sono le seguenti:

- il consumatore si trova davanti ad un buffer vuoto;
- il produttore trova un buffer **pieno**, ossia tutti gli slot del buffer sono occupati.

Notare che il problema si presenta anche quando ci siano più produttori e più processi. Per semplicità tratteremo il problema solo con un processo produttore e un processo consumatore.

All'interno della nostra logica, inseriamo una variabile `count` che tiene conto del numero di elementi presenti all'interno del buffer. In pseudo-codice avremo allora le seguenti logiche:

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

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

Le operazioni chiaramente sono astratte e non ci interessa cosa rappresentano le funzioni `produce_item(), consume_item(), remove_item()`. Se il thread consumatore o produttore non sta dormendo, l'istruzione `wakeup()` non fa niente.
Il meccanismo illustrato non funziona tuttavia. Il problema risiede proprio nel fatto che la variabile `count` crea una **race condition**. Infatti supponiamo che il consumatore legga il valore contenuto in `count` e trovi 0. Supponiamo che in questo istante la CPU tramite il meccanismo di prelazione sospenda momentaneamente l'esecuzione del processo consumatore e mandi in esecuzione il produttore. Il produttore allora produce un item, lo inserisce nel buffer e aumenta `count` portandolo ad 1. A questo punto esegue la syscall `wakeup()`. Tuttavia il consumatore non è logicamente sospeso, quindi **il segnale di sveglia va perso**. Quando il consumatore torna in esecuzione, poiché il valore letto di `count` era pari a 0 entra in uno stato di sleep. Il produttore allora continuerà ad aggiungere oggetti fino a quando non si addormenta e i processi rimangono sospesi per sempre.
Il problema sembra essere che le sveglie vengono perse. Una soluzione può essere l'aggiunta di un meccanismo: se viene lanciata una wakeup su un thread che non sta dormendo, **si accende una bit flag** che indica una **sveglia pendente**. Questa sveglia pendente va "consumata". La bit flag prende il nome di **wakeup waiting bit**.
Apparentemente la soluzione sembra valida, ma in realtà il problema si innesca quando i

processi consumatori e produttori sono di più. Non è sufficiente ricordarci una sola sveglia, ma ci vuole una memoria più estesa.

Semafori

L'evoluzione del meccanismo sleep-wakeup è il concetto dei **semafori**. Il semaforo può essere pensato come una variabile intera condivisa S che può assumere un qualunque valore degli interi. C'è solo una condizione: **il valore del semaforo non può essere negativo**. Definiamo due operazioni (il nome cambia in base al contesto, ma classicamente sono denominate così): `down` e `up` (o `wait` e `post`, `wait` e `signal`). Il decremento intrinseco all'operazione `down` deve rispettare l'invariante definito precedentemente. Se si ha un decremento con semaforo pari a 0, si ha una chiamata bloccante. `up` invece incrementa semplicemente il semaforo.
In effetti si potrebbe considerare il fatto che le operazioni `up` e `down` causino una race condition, o in generale ci potrebbero essere race condition sulla gestione del semaforo. Per evitare ciò, implementiamo `up` e `down` come **operazioni atomiche**: si garantisce così che nessun processo possa accedere al semaforo fino a quando l'operazione viene completata.
Poiché è il sistema operativo che implementa il meccanismo dei semafori, esso deve implementare la mutua esclusione.
Una prima idea potrebbe essere quella di disabilitare gli interrupt, strategia che funziona in sistemi single-core. La strategia è accettabile poiché:

- siamo in modalità kernel;
- il bloccaggio degli interrupt va bene per un piccolo lasso di tempo, ossia il tempo necessario per controllare la variabile S.

Una seconda idea è usare una variabile di lock sul semaforo S e utilizzare l'operazione `TSL`. Tuttavia la `TSL` utilizza il meccanismo di busy-waiting; sarebbe così un'incoerenza. Il meccanismo di atomicità tuttavia copre solamente il lasso di tempo piccolissimo necessario per controllare e modificare il valore contenuto in S. Se la wait viene invocata per un semaforo $S = 0$, allora diventa una chiamata bloccante.
Quindi:

- in sistemi con un solo core si può disabilitare l'interrupt;
- in sistemi multi-core si utilizza la strategia con `TSL` o `XCHG` con un meccanismo di busy-waiting per una frazione di tempo minimale.

Semaforo mutex

Il **semaforo mutex** è un uso particolare per garantire la mutua esclusione (mutex = mutual exclusion). Si tratta di una soluzione utilizzata quando bisogna gestire la mutua esclusione di qualche risorsa o pezzo di codice condivisi e quando **non è di interesse la capacità di**

contare le risorse.
Il semaforo mutex viene implementato come una variabile intera:

- il valore 0 indica uno stato di **non bloccato**;
- ogni altro valore diverso da 0 indica uno stato di **bloccato**.

Con i mutex si utilizzano due procedure: se un processo accede ad una sezione critica, richiama `mutex_lock()` e se il mutex si trova nello stato di non bloccato, la chiamata ha successo e il thread chiamate entra in regione critica. Se il mutex è bloccato, il thread viene bloccato finché il thread in regione critica chiama `mutex_unlock()`: tale procedura modifica il valore del semaforo mutex, impostandolo a 0.
I mutex sono molto semplici, pertanto possono essere implementati facilmente nello spazio utente sfruttando l'istruzione `TSL`:

```
mutex_lock:
    TSL registro, mutex
    CMP registro, #0
    JEQ ok
    CALL thread_yield
    JMP mutex_lock

ok:
    RET

mutex_unlock:
    MOV mutex, #0
    RET
```

La logica della chiamata sembra identica alla `enter_region()`. Tuttavia, `enter_region()` implementa un busy waiting. Con i processi non c'è problema, perché prima o poi il clock scade e un altro processo viene schedulato: il processo che detiene il blocco andrà in esecuzione e lo rilascerà. Ma con i thread la situazione è diversa: **non esiste un clock per fermare i thread che sono stati in esecuzione troppo a lungo**. Il thread che prova ad acquisire un blocco starà in ciclo per sempre, in quanto non permette ad altri thread di essere in esecuzione.
Con `mutex_lock` è possibile risolvere il problema: se il mutex è diverso da zero, il thread esegue la chiamata a `thread_yield`, **cedendo la CPU ad un altro thread**. Non c'è busy waiting, ma semplicemente se torna in esecuzione effettuerà nuovamente il test su blocco.
Da notare che `thread_yield` permette di utilizzare il meccanismo mutex **nello spazio utente**, permettendo quindi a thread di livello utente di sincronizzarsi completamente nello spazio utente. Sono inoltre chiamate molto veloci, pertanto **l'overhead è molto basso**.

Conteggio risorse

Esiste un altro modo per utilizzare i semafori che consiste nell'utilizzare i semafori per contare risorse. Questo permette di risolvere il problema usando la **sincronizzazione**.
I semafori possono essere utilizzati come contatori di risorse. Consideriamo il problema produttore consumatore e utilizziamo le seguenti variabili:

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

Dove:

- `full` conta i posti occupati del buffer;
- `empty` conta i posti vuoti del buffer;
- `mutex` controlla che produttore e consumatore non accedano al buffer contemporaneamente;
- `N` indica i posti del buffer.

Semafori come il `mutex` prendono il nome di **semafori binari**.

> **Semaforo binario**
>
> Un semaforo si dice **binario** se viene utilizzato da due o più processi per assicurarsi che di volta in volta solo uno di essi possa entrare in una regione critica.

Facciamo una sequenza di operazioni precise:

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

function consumer()
    while(true) do
```

```
        down(full)
        down(mutex)
        item = remove_item()
        up(mutex)
        up(empty)
        consume_item(item)
```

Il produttore genera qualcosa da inserire nel buffer con `produce_item()` e decrementa il contatore delle posizione vuote con `down(empty)`. Per garantire la mutua esclusione, si utilizza la coppia di chiamate `down(mutex)` e `up(mutex)`. Generalmente, **se ogni processo chiama una `down` prima di entrare in regione critica e una `up` dopo esserne uscito, si garantisce la mutua esclusione**. Infatti se un secondo processo esegue una chiamata a `down(mutex)`, se `mutex` è uguale a 0 il processo viene sospeso senza completare la `down`.
Dopo che il produttore inserisce l'item nel buffer, si chiama `up(mutex)` e `up(full)`.
Le chiamate bloccanti per il produttore avvengono quando si vuole inserire un elemento all'interno del buffer (`down(empty)`); analogamente, le chiamate sono bloccanti per il consumatore quando si cerca di rimuovere oggetti su un buffer vuoto (`down(full)`). Notiamo quindi che le chiamate `down` e `up` sono **accoppiate** e l'ordine dei semafori **non è opinabile**: ad una chiamata di `down(empty)` del produttore corrisponde una chiamata `up(empty)` del consumatore dopo che il consumatore rimuove un oggetto dal buffer. Si garantisce così la **sincronizzazione**: l'accesso corretto richiede un corretto ordine di esecuzione e coordinazione.

08 - Futex

Futex

In alcuni sistemi operativi esiste un meccanismo ibrido per avere meccanismi di lock efficienti. Questo strumento è il **futex** (**fast userspace mutex**). Si tratta di un mutex la cui implementazione garantisce la sua efficienza superiore rispetto a quanto visto. Il sistema viene utilizzato esclusivamente su sistemi Linux. Ad esempio, le pthread su Linux utilizzano i futex, mentre in altri sistemi operativi utilizza altri tipi di mutex compatibili.

L'efficienza del futex rispetto al mutex risiede nel meccanismo di contesa, ossia il meccanismo di controllo relativo all'acquisizione del lock. Il futex combina l'aspetto positivo della TSL, ossia la possibilità di eseguire controlli di bloccaggio sul lato utente, con un meccanismo di bloccaggio **gestito dal kernel**: quando è necessario, il thread / processo chiede aiuto al sistema operativo e si blocca con una chiamata di sistema, chiedendo una sleep in cui la richiesta viene eseguita **solo se necessaria**.
Questo non significa che i semafori non sono efficienti, ma hanno un overhead per il fatto che `down` e `up` sono comunque chiamate di sistema. Sebbene siano costi piccoli, se iterati hanno un costo anche considerevole.

Struttura Futex

Il futex è costituito da una libreria utente e da un servizio del kernel. Il servizio del kernel fornisce una coda di attesa che permette a più processi di attendere un lock: i processi non verranno eseguiti se il kernel non li sblocca esplicitamente. La componente dello spazio utente, ossia la libreria, è coinvolta nel testing della variabile di lock, eseguendo operazioni TSL. Sebbene sia richiesta una chiamata di sistema, vi sono numerosi vantaggi:

- facciamo uso della chiamata di sistema solo nel momento in cui bisogna bloccare il processo / thread;
- il blocco è implementato in maniera **passiva** in quanto il processo viene tolto dalla coda dei processi attivi e inserito in una coda di attesa;
- la componente kernel è chiamata in situazioni di alta contesa. Nei casi di assenza di contesa, il controllo della variabile di lock non richiede passaggi al kernel.

Monitor

Un'alternativa all'utilizzo dei semafori risiede nell'utilizzo di **monitor**.
Utilizzare i semafori richiede che il programmatore sappia gestire bene la mutua esclusione utilizzando correttamente le chiamate `down` e `up`. Per rendere il programmatore più "rilassato",

e quindi avere una soluzione più comoda, esiste una soluzione più ad alto livello, ossia il **monitor**.

Il monitor è un tipo di dato astratto contenente dati e metodi, un particolare "oggetto" con alcuni vincoli sui dati. Molto spesso il linguaggio stesso di programmazione definisce il monitor. Non si tratta di qualcosa offerto dal sistema operativo: chi fornisce il monitor è il compilatore (nel caso di un linguaggio compilato) che crea del codice che rende concrete le proprietà dello strumento, e nel farlo sfrutta l'infrastruttura del sistema operativo (ad esempio, futex su Linux).

Il fatto che il monitor gestisca in maniera elegante aspetti come la mutua esclusione rende l'utilizzo di questi vantaggioso. Possiamo immaginare il monitor come un oggetto con una serie di strutture dati e una serie di metodi. Le strutture dati definite nel monitor sono **interne** (quindi non accessibili dall'esterno). I metodi possono accedere ai dati delle strutture interne, **ma non possono accedere ai dati delle strutture esterne**. Inoltre, **tutti i metodi del monitor sono eseguiti in mutua esclusione**: se un thread ha invocato un metodo del monitor (in gergo si dice che *il thread è dentro il monitor*), nessun altro thread può farlo. Quindi viene garantita la **mutua esclusione nell'esecuzione dei metodi di un monitor**. Questo meccanismo di mutua esclusione tuttavia è garantita dal linguaggio, quindi nessun programmatore deve aggiungere istruzione per far funzionare l'esclusione. In questo modo:

- il codice diventa minimale;
- non vi è rischio di eseguire istruzioni parziali per l'esclusione (ad esempio, dimenticando una chiamata `down` o `up`), garantendo correttezza del codice.

La gestione della mutua esclusione tuttavia non è l'unico aspetto analizzato di questi strumenti: da sola la mutua esclusione non garantisce **sincronizzazione**, facendo riferimento con questo termine al fatto che in alcune condizioni il sistema si deve bloccare in attesa di qualcosa. I monitor infatti hanno altre funzionalità per garantire sincronizzazione.
I monitor vengono principalmente utilizzati per garantire sincronizzazione tra thread. Linguaggi come C# e Java permettono le attivazioni di funzionalità dei monitor alle classi (in java basta utilizzare la keyword `synchronized`). Per garantire la sincronizzazione vengono aggiunte le **variabili condizione**, etichette su cui il thread stesso dall'interno del monitor può esercitare una `wait` e una `signal`. La prima provoca il **bloccaggio**, testando una condizione e verificando che essa sia vera: se la condizione è vera, il thread si blocca. Questo meccanismo è stato già visto con il controllo nel produttore e consumatore: se il buffer è pieno, il produttore si deve fermare. La seconda, invece, è intesa come un **risveglio**. Un altro processo che ha accesso alla regione critica può risvegliare un processo usando `signal` sulla variabile condizione su cui il processo stava attendendo.
Le variabili condizione non sono implementate con semafori inseriti in un monitor. Le variabili condizione:

- **non hanno uno stato**, ma sono etichette per indicare quali sono le condizioni da verificare. In questo caso non è necessario uno stato, cosa necessaria per i semafori per indicare quante sono i messaggi di sveglia da consumare;
- **non hanno memoria**, perdendo quindi i messaggi di sveglia se nessun processo è in stato sleep. Le variabili condizioni non hanno bisogno di memoria in quanto le variabili condizione vengono usate solo dentro il monitor, e il monitor **garantisce la mutua esclusione**. In altri termini, **non c'è nulla che si possa inserire tra la fase di controllo della condizione bloccante e l'azione conseguente di sleep**. Non ci sono prelazioni e race condition, perché tutto ciò che agisce nel monitor agisce in mutua esclusione;

Semantiche signal

Ci sono diverse semantiche per definire il concetto di `signal`. Più precisamente, bisogna definire una semantica per specificare cosa accade dopo una chiamata `signal`.

La prima semantica è la semantica di **Hoare** e prende il nome di **signal and wait**. Come suggerisce il nome stesso, signal and wait significa che se un thread invia una signal, esso si **autosospende**, permettendo all'altro thread in sleep di risvegliarsi. Quando l'altro thread ha completato, il thread che ha chiamato `signal` si sveglia.
Un problema di questo approccio riguarda l'**interlacciamento dei metodi all'interno del monitor**. L'idea che un codice viene eseguito in mutua esclusione potrebbe essere negata, in quanto i codici dei metodi potrebbero mischiarsi, interlacciando metodi che in teoria dovrebbero essere eseguiti in modo esclusivo. La semantica di Hoare pertanto non viene considerata una buona soluzione.

L'altra soluzione che prende il nome di **signal and continue** è una soluzione reale, e i monitor che utilizzano questa soluzione prendono il nome di **Mesa**. Si tratta di una semantica che utilizza un approccio leggermente diverso che non nega mutua esclusione.
L'idea è che se un thread lancia una `signal`, il sistema ne prende nota, **ma il thread continua la sua esecuzione**. Quando il thread ha completato l'esecuzione, il thread che ha ricevuto la `signal` si risveglia. Questa semantica viene anche utilizzata per Java.
Una critica a questo tipo di approccio risiede nel fatto che non risvegliare subito il thread da alcuni punti di vista potrebbe essere visto come un aspetto negativo. Il rischio potrebbe essere che si lancia una `signal` perché la condizione di bloccaggio potrebbe non essere più presente; tuttavia, tutto ciò che segue la signal non deve invalidare il motivo per cui è stata eseguita la chiamata `signal`.

Esiste una terza semantica ibrida, che consiste nell'associare ad una `signal` un `return`. Questo significa che dopo la `signal` non vi è alcuna istruzione. L'interlacciamento tra i processi diventa minimale. Questa soluzione è la soluzione di **Brinch Hansen** ed è la più facile da implementare e anche concettualmente semplice.

Monitor e thread

I monitor si usano per coordinare thread e non processi per il modo in cui sono integrati nel linguaggio. Linguaggi che implementano i monitor implementano anche i thread: Java integra i monitor e i thread in maniera elegante, definendo più thread, e lo stesso in C#. Non si può dire la stessa cosa per due processi distinti.
Se due processi distinti, ognuno col proprio flusso di esecuzione, devono lavorare in maniera condivisa, poiché sono entità separate non hanno lo stesso spazio di indirizzamento, per cui non possono condividere il monitor senza un mezzo di comunicazione definito. Per questo tipi di scenari si utilizzano i semafori in quanto sono oggetti fuori dai processi (infatti sono definiti nel sistema operativo con le relative chiamate di sistema).

Esempio: problema del produttore e consumatore con i monitor

Utilizziamo ora i monitor per risolvere il problema del produttore e consumatore. Definiamo il monitor:

```
monitor pc_monitor
    condition full, empty;
    integer count = 0;
    buffer[N]; // nascosto nelle azioni quali insert_item()

    function insert(item)
        if count = N then wait(full);
        insert_item(item); // riferisce a buffer[N]
        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);

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

function consumer()
```

```
    while(true) do
        item = pc_monitor.remove();
        consume_item(item);
```

L'azione di inserimento `insert()` controlla prima se c'è spazio. Se `count = N`, allora bisogna bloccare il thread su un evento. Abbiamo definito le variabili condizione `full` e `empty`: sono delle condizioni su cui rispettivamente chi inserisce e chi estrae può addormentarsi o risvegliare qualcun altro. Nella funzione di inserimento, se `count = 1` viene lanciata una `signal(empty)`, che risveglia chi è in stato di sleep a causa di una chiamata `wait(empty)`. Se `signal(empty)` viene chiamata ma nessun thread è in stato di sleep per la condizione `empty`, allora viene persa, ma come illustrato non vi è alcun problema.

Scambio messaggi tra processi

Nello scenario multi-processo si può utilizzare un approccio basato sullo **scambio di messaggi**. L'idea è abbastanza intuitiva: si comunica tra processi tramite uno scambio di messaggi, impacchettando le informazioni in messaggi e inviando questi da un processo ad un altro. Si tratta del modello usato anche per lo scambio di messaggi in rete. Lo scambio dei messaggi può essere implementato con due primitive ad alto livello `send(destinazione, messaggio)` e `receive(sorgente, messaggio)`.
La chiamata `receive(sorgente, messaggio)` è una chiamata bloccante per il chiamante, in quanto si blocca per ricevere il messaggio, in maniera simile con cui nello shell un processo attende l'output del processo precedente con il meccanismo della pipeline. Allo stesso modo una `send` può essere bloccante se il messaggio non viene ricevuto: poiché non si utilizzano **buffer**, se un messaggio non viene memorizzato esso viene perso. Questa strategia prende il nome di strategia **rendezvous**.
Alternativamente è possibile utilizzare una **mailbox**, ossia un buffer per messaggi. Se il buffer è pieno, un processo che esegue `send` viene bloccato fino a quando non vi è posto per un altro messaggio.

Il problema produttore-consumatore può essere quindi modellato come segue:

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

function consumer()
    while(true) do
        receive(producer, msg)
```

```
        item = extract_msg(msg)
        consume_item(item)
```

Il modello a messaggi, sebbene sia molto semplice, è **molto inefficiente**. L'idea di prendere un item, incapsularli e usare la `send` e `receive` e usare la mailbox consiste nell'utilizzo di chiamate di sistema; la **mailbox** è un buffer memorizzato in un sistema operativo. Inoltre, porre i messaggi in mailbox o riceverli richiede una copia delle risorse all'interno dello spazio di indirizzamento dei processi. Risulta quindi essere un procedimento molto costoso l'invio del messaggio, in quanto:

- si ha una chiamata di sistema per mandare il messaggio;
- si copia il messaggio nel buffer;
- tramite la receive si copia il messaggio sullo spazio di indirizzamento del ricevente.

Infine, sistemi che si basano sullo scambio di messaggi presentano problemi relativi alla **perdita** di questi durante la comunicazione, nonché a problemi di **autenticazione**. I benefici tuttavia di tale tipo di comunicazione risiede nel fatto che tale tipo di scambio opera anche su processi su macchine separate; infatti un modello simile viene usato per le reti.

Librerie MPI

Per risolvere un problema computazionalmente complesso utilizzando un calcolo distribuito è possibile organizzare il lavoro utilizzando le **librerie MPI** (**Message-Passing Interface**).
Tramite la libreria è possibile modellare l'elaborazione tramite uno scambio di messaggi. La libreria fornisce un paradigma unico, gestendo le differenze che ci sono tra vari scenari, quali l'utilizzo di un cluster distribuito sulla stessa LAN, o su cluster che lavorano su datacenter distanti ma che vengono visti come un unico pool, o ancora come processi che lavorano sulla stessa macchina.
MPI è diventato uno standard per il calcolo distribuito e la programmazione parallela.
Il vantaggio di tale approccio risiede nel fatto che tale metodologia permette di modellare ogni elaborazione tramite scambio di messaggi, rendendo qualsiasi scenario adattabile a qualsiasi macchina.

Problema dei cinque filosofi

Il problema dei cinque filosofi è un problema classico che modella l'accesso esclusivo ad un numero limitato di risorse.
Si hanno cinque filosofi seduti su un tavolo circolare. Davanti a ciascun filosofo è presente un piatto. A lato dei piatti vi è una forchetta, per un totale di cinque forchette. Quando un filosofo deve mangiare, deve prendere entrambe le forchette ai lati del piatto.
Si modella un problema più generale, in cui per poter passare ad una fase particolare di un task bisogna prendere **risorse esclusive** utilizzabili da un processo alla volta.

Con cinque forchette ci possono essere due filosofi che possono mangiare contemporaneamente, in quanto acquisiscono insiemi disgiunti di forchette (ossia i due filosofi non sono vicini). Il motivo per cui ci sono cinque filosofi e non quattro si riconduce al fatto che il numero cinque permette di generalizzare la soluzione del problema. Alcune soluzioni per il problema quattro possono causare stalli quando si hanno n filosofi.

Prima soluzione

Un primo approccio al problema può essere il seguente:

```
int N = 5
function philosopher(int i)
    think()
    take_fork(i)
    take_fork((i+1) mod N)
    eat()
    put_fork(i)
    put_fork((i+1) mod N)
```

Con la forchetta `i` e `(i+1) mod N` indichiamo la forchetta che sta rispettivamente alla sua destra e alla sua sinistra.
Il meccanismo funziona con un ciclo di base basato su una chiamata bloccante. La chiamata `take_fork()` tenta di ottenere l'i-esima forchetta e la seconda tenta di ottenere la successiva.

La soluzione proposta non funziona in quanto se tutti e cinque i filosofi provano ad acquisire la forchetta alla propria destra nello stesso momento, nessuno può avere la forchetta a sinistra, e tutti si bloccheranno sulla chiamata `take_fork((i+1) mod N)`. Si tratta di una condizione limite, ma è possibile. La natura subdola del problema risiede proprio nell'utilizzo di entrambe le forchette.
Una situazione del genere, in cui tutti i processi continuano ad essere eseguiti in maniera indefinita ma senza fare progressi, prende il nome di **starvation**.

Seconda soluzione

Un secondo tentativo potrebbe essere di utilizzare un tempo di attesa nel momento in cui non si può prendere la seconda forchetta. Quindi si prende la forchetta, e nel momento in cui la seconda forchetta non è accessibile si posa la forchetta e si aspetta un tempo t. Dopo questo tempo, si riprende la forchetta e si esegue un nuovo tentativo.

Anche in questo caso si può rimanere in stallo: se tutti tentano di prendere la forchetta nello stesso tempo, tutti aspettano lo stesso tempo t e la perfetta sincronia pone in stallo il sistema.

Terza soluzione

Si può risolvere la soluzione precedente mettendo un timer t randomico. La soluzione è anche utilizzata in applicazioni reali, come Ethernet in LAN: se due computer inviano un pacchetto nello stesso momento, ognuno di questi aspetta un timer randomico e riprova. Tuttavia, in alcune applicazioni questa soluzione non è accettabile, ad esempio in applicativi che richiedono una soluzione **sempre** funzionante e senza errori.

09 - Problema dei cinque filosofi

Problema dei filosofi, problema lettore-scrittore, scheduling

Soluzione basata sui semafori

Possiamo applicare una soluzione basata sull'utilizzo dei semafori:

```
THINKING = 0, HUNGRY = 1, EATING = 2;
int state[N];           // stato di ogni filosofo
semaphore mutex = 1;    // mutua esclusione per le regioni critiche
semaphore s[N];         // un semaforo per filosofo

function philosopher(i) {
    while(TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}

function take_forks(i) {
    down(mutex);
    state[i] = HUNGRY;
    test(i);        //test per acquisire le forchette
    up(mutex);
    down(s[i]);     //blocca se le forchette non sono prese
}

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

function test(i) {
```

```
    if(state[i] == HUNGRY and state[left(i)] != EATING and
    state[right(i)] != EATING) {
        state[i] = EATING;
        up(s[i]);
    }
}

function left(i) { return (i+N-1) % N; }
function right(i) { return (i+1) % N; }
```

Stati dei filosofi

Identifichiamo gli stati dei filosofi con le variabili `THINKING`, `HUNGRY`, `EATING`. Useremo un vettore `state[N]` per indicare lo stato del relativo filosofo. Le numerazioni partono da 0 a 4 e sono numerate in senso orario, partendo dal filosofo 0 in ore 12. Quindi `N = 5`.

Utilizziamo un `mutex` per garantire il meccanismo di mutua esclusione per l'accesso al vettore `state`. Utilizzeremo un array di semafori `s[N]`, permettendo al filosofo di auto-addormentarsi accedendo alla cella corrispondente.

Analisi funzioni

La funzione `philosopher(i)` controlla se l'i-esimo filosofo è in grado di prendere entrambe le forchette. Successivamente mangia e pone giù le forchette.

La funzione `take_forks(i)` consente all'i-esimo filosofo di prendere entrambe le forchette. La prima istruzione utilizza l'istruzione `down(mutex)` per accedere alla sezione critica dell'esecuzione, ossia l'accesso alla risorsa condivisa `state`.

`state[i] = HUNGRY` modifica lo stato dell'i-esimo filosofo in uno stato "affamato", senza però prendere le forchette.

`test(i)` esegue un test per verificare se il filosofo è in grado di accedere alle risorse, ossia prendere entrambe le forchette. All'interno della funzione `test(i)` troviamo l'istruzione `up(s[i])`: se il test ha successo, è possibile eseguire l'istruzione `up(s[i])`. Quando si ritorna dalla funzione `test(i)`, si esegue una `up(mutex)` e di `down(s[i])`. Notiamo che eseguiamo `down(s[i])` solamente quando dopo essere usciti dalla funzione `test(i)` in quanto potrebbe essere una chiamata bloccante all'interno della funzione `test(i)`, non permettendo quindi a nesusn filosofo di mangiare: infatti se scrivessimo `down(s[i])` all'interno di un ipotetico `else` della funzione `test`, si può avere una chiamata bloccante e non potremmo più eseguire `up(mutex)` poiché non si ritorna alla funzione `take_forks(i)`. In questo modo anche se

`down(s[i])` fosse bloccante, gli altri processi potrebbero accedere in ogni caso alla sezione critica.

La funzione `put_forks(i)` comincia chiamando `down(mutex)`. Successivamente si modifica lo stato `state[i]` in `THINKING`. Successivamente si chiama la stessa funzione `test` per `left(i)` e `right(i)`, verificando se i filosofi a sinistra e a destra di i possono entrare in stato di `EATING`. Il motivo è per il fatto che il filosofo è entrato in uno stato bloccato dopo `down(s[i])` perché almeno uno tra il filosofo a sinistra e il filosofo a destra è in uno stato di `EATING`.

Soluzione basata sui monitor

La soluzione è simile ma ha meno problematiche relative ai blocchi con l'utilizzo dei semafori.:

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

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

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

    function test(int i) {
        if (state[left(i)] != EATING and state[i] == HUNGRY
        and state[right(i)] != EATING) {
            state[i] = EATING;
            signal(self[i])
        }
    }

function philosopher(int i) {
    while(true) do
```

```
        think();
        dp_monitor.take_forks(i);
        eat();
        dp_monitor.put_forks(i);
}
```

Definiamo `N` variabili condizioni `self`, una per filosofo e necessarie per il filosofo per auto-addormentarsi. L'i-esimo filosofo si addormenterà sull'i-esima variabile `self[i]` quando non ha le condizioni necessarie per mangiare.

Il codice del filosofo è molto simile, con l'unica variante che `take_forks` e `put_forks` sono definite nel monitor e pertanto non vengono eseguite in maniera concorrente.

Il codice di `take_forks` transita lo stato dell'i-esimo filosofo in `HUNGRY` ed esegue un test sull'i-esimo filosofo. `test(i)` controlla se a destra e a sinistra i filosofi sono in fase di `EATING`.
Notiamo che il controllo `state[i] = HUNGRY` è un controllo ridondante.
Se il test dà esito negativo, quindi `state[i]` alla terminazione è diverso da `EATING`, il filosofo si blocca, chiamanto `wait(self[i])`. Altrimenti, `state[i]` è uguale a `EATING` e si esegue `signal(self[i])`.

Problema dei lettori - scrittori

Il problema dei lettori - scrittori è un problema abbastanza ricorrente che, come quelli visti precedenti, modella un problema reale e implementato nei sistemi operativi per risolvere problemi simili. Il problema modella l'accesso a un database da parte di molteplici entità. Il database enfatizza la complessità dello scenario, ma può essere benissimo una struttura dati condivisa.
Gli accessi concorrenti creano race condition se c'è un agente che vuole modificare la struttura nel momento in cui ci sono letture. In altri termini, la mutua esclusione non è sempre necessaria quando si parla in generale di accesso concorrente, ma solo quando si hanno accessi concorrenti e uno di essi è in scrittura. In generale, l'azione di uno scrittore preclude ogni altra concorrenza. In questo modo, evitiamo di bloccare thread o processi che lavorano sulla stessa struttura dati quando non è realmente necessario.

Pensiamo ad un database che rappresenta le previsioni meteo, e supponiamo molti utenti accedano a tale database in lettura. Non avremmo motivo di bloccare questi accessi: usare mutex o meccanismi con lock rallenta gli accessi. Se però noi modifichiamo il database, è necessario bloccare l'accesso concorrente.

```
semaphore mutex = 1;
semaphore db = 1;
```

```
int rc = 0;

function reader() {
    while(true) do {
        down(mutex);
        rc = rc+1;
        if (rc = 1) down(db);
        up(mutex);
        read_database();
        down(mutex);
        rc = rc-1;
        if (rc = 0) up(db);
        up(mutex);
        use_data_read();
    }
}
function writer() {
    while(true) do {
        think_up_data();
        down(db);
        write_database();
        up(db);
    }
}
```

Individuiamo due procedure `reader()` e `writer()`, cicli infiniti che eseguono rispettivamente lettura di dati e scrittura dei dati. Utilizziamo due semafori, apparentemente entrambi di tipo mutex: `db` e `mutex`. `db` è associato per garantire mutua esclusione all'accesso al database. `mutex` invece è un altro semaforo che protegge la mutua esclusione nei confronti di un'altra struttura dati, un contatore `rc`, ossia `reader_counter`, che conta quanti lettori stanno leggendo il database. `rc` è inizializzato a zero.

`reader()` esegue una `down(mutex)` per aggiornare il contatore `rc`, e controlla se è il primo tra i lettori. Se `rc` è uguale ad `1`, ciò significa che è il primo dei lettori.: in tal caso, esegue `down(db)`, permettendo quindi di bloccare lo scrittore se vuole operare durante la lettura del reader o di bloccarsi se vi è già uno scrittore. Successivamente esegue un `up(mutex)` e legge il database. Dopo la lettura del database, esegue una `down(mutex)` per accedere a `rc` e ridurne di 1 il valore. Se `rc = 0`, e quindi è l'ultimo lettore, il reader permette ad eventuali scrittori di accedere al database. Infine esegue `up(mutex)` ed elabora i dati letti con `use_data_read()`.

`writer()` semplicemente scrive nel database se è possibile accedervi, ossia eseguendo `down(db)`, scrivendo nel database ed eseguendo `up(db)`. Se vi sono altri reader, la chiamata è bloccante.

Il problema di questo approccio è che lo scrittore potrebbe attendere un tempo indefinito. Infatti fintanto che un flusso di lettori è continuo, essi entrano appena arrivano: se uno scrittore vuole entrare dovrà aspettare che tutti i lettori siano usciti.

Implementazioni tra monitor

Utilizziamo un monitor con operazioni che riferiscono ad un database esterno. All'interno del monitor implementiamo le operazioni di lettura e scrittura nel database. Inserire il database infatti all'interno del monitor implicherebbe mutua esclusione anche per la lettura. Utilizziamo le variabili condizioni `read`, `write`:

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

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

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

    function start_write() {
        if (rc > 0 or busy_on_write) wait(write);
        busy_on_write = true;
    }

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

```
function reader() {
    while(true) do {
        rw_monitor.start_read();
        read_database();
        rw_monitor.end_read();
        use_data_read();
    }
}

function writer() {
    while(true) do {
        think_up_data();
        rw_monitor.start_write();
        write_database();
        rw_monitor.end_write();
    }
}
```

Il lettore comincia l'operazione di lettura con `rw_monitor.start_read()`. `start_read()` prima verifica se il database è occupato in operazioni di scrittura: in tal caso, si blocca con `wait(read)`. Altrimenti, aumenta `rc` di 1 ed esegue `signal(read)`. `end_read()` invece diminuisce `rc` e se `rc = 0` chiama `signal(write)`, risvegliando un possibile writer bloccato.

`start_write()` individua due scenari in cui uno scrittore si blocca:

- c'è almeno un lettore;
- c'è un altro scrittore.

Se uno di questi due scenari si presenta, si blocca con `wait(write)`. Altrimenti, notifica che il database è impegnato in una scrittura imponendo `busy_on_write = true`. Con `end_write()` la variabile booleana `busy_on_write` viene impostata a `false`, e se ci sono lettori in coda viene risvegliato il primo lettore in coda. Altrimenti risveglia uno scrittore.

Questa soluzione sembra avere lo stesso problema del metodo precedente. Possiamo modificare la funzione `start_read()`, in particolare controllando non solo se `busy_on_write = true`, ma anche se `in_queue(write)`. Ossia, se è presente in coda uno scrittore, il lettore viene mandato in attesa:

```
...
function start_read() {
```

```
    if(busy_on_write or in_queue(write)) wait(read);
    rc = rc + 1;
    signal(read);
}
```

Oltre ad aggiungere la precedente modifica, possiamo modificare la `end_write()` permettendo di far risvegliare uno scrittore a presincedere dalla sua posizione in coda:

```
...
function end_write() {
    busy_on_write = false;
    if(in_queue(write)) signal(write);
    else signal(read);
}
```

In questo modo, gli scrittori hanno priorità e si favorisono a vicenda. La situazione tuttavia si ribalta, ponendo a sfavore i lettori.

Scheduling

La componente **scheduler** prevede di scegliere, quando serve al sistema operativo, tra tutti i processi uno tra quelli presenti nella coda dei processi pronti. Non esiste un solo algoritmo di scheduling, o un algoritmo migliore degli altri, ma algoritmi adatti ad alcuni scenari:

- scenari di sistemi batch, dove si eseguono una serie di job non interattivi;
- scenari di sistemi interattivi, come scenari di computer desktop vicini all'esperienza quotidiana con i calcolatori;
- scenari di sistemi real-time.

Il modo con cui un algoritmo prende una decisione dipende da che tipi di processi si ha davanti. L'algoritmo prende decisioni in virtù dell'uso del processore, uso di memoria e uso di periferiche, network ed elementi di differente natura. Distingueremo inoltre le fasi in cui un processo usa la CPU da quelle in cui utilizza altre risorse.

Tipologie di processi

Quasi ogni processo alterna **burst** di computazione (o **CPU burst**, ossia istanti di attività di CPU) con richieste I/O. Anche alcune attività I/O possono essere considerate come computazione, pertanto quando diciamo "attività I/O" intendiamo l'ingresso da parte di un processo in uno stato bloccato in attesa che un dispositivo esterno completi il suo lavoro. Possiamo raggruppare quindi i processi in due categorie:

- processi **CPU-bound**;
- processi **I/O-bound**.

I processi del primo tipo sono caratterizzati da CPU burst abbastanza lunghi. I processi del secondo tipo invece sono caratterizzati da CPU burst abbastanza brevi, attendendo o inviando dati a dispositivi I/O. Inoltre, i processi I/O-bound non eseguono molta computazione durante le richieste I/O, e non sono definiti in tale modo perché le richieste I/O sono lunghe.

La caratterizzazione dei processi chiaramente non è rigida, ossia ci sono dei processi che non sono strettamente CPU-bound o I/O-bound, ma può essere usata per capire come uno scheduler debba comportarsi con i tipi di processi. Il sistema operativo dev'essere capace di identificare il tipo di processo in esecuzione e capire l'evoluzione di questo nel tempo di esecuzione.

Interrogazione di uno scheduler

Uno scheduler viene interrogato generalmente in una di queste situazioni:

- terminazione e creazione di processi. Più precisamente, durante la creazione di processi bisogna decidere se eseguire il processo padre o il processo figlio. Se invece un processo viene terminato, un altro processo dev'essere scelto dalla ready queue;
- chiamata bloccante e arrivo di interrupt;
- interrupt periodici, ad esempio il clock interrupt. In questo modo si distinguono due tipi di sistemi:
  - sistemi **non-preemptive**, senza meccanismo di prelazione. In questo caso l'algoritmo di scheduling prende un processo e lo manda in esecuzione fino a quando si blocca o rilascia volontariamente la CPU, senza limiti temporali;
  - sistemi **preemptive**, con meccanismi di prelazione. L'algoritmo di scheduling sceglie un processo e lo manda in esecuzione per un quanto di tempo fissato. Dopo tale quanto di tempo, il processo viene sospeso e si prende un nuovo processo.

Scheduler e context switch

Fondamentalmente lo scheduler esegue la scelta. Chi si occupa invece di eseguire il context switch è il **dispatcher**. Il context switch ha un certo costo, e il tempo impiegato dal dispatching prende il nome di **latenza di dispatch**.

10 - Scheduling, sistemi batch

Scheduling

Ragioniamo ora su possibilità a livello algoritmico di scegliere processo. Tali scelte vanno cablate in virtù del sistema da gestire. In particolare, faremo una distinzione principale tra sistemi batch, interattivi e real-time:

- i primi gestiscono processi che non sono interattivi e non devono dare feedback. Il sistema deve massimizzare la produttività e minimizzare i tempi di attesa;
- i secondi gestiscono processi che hanno interfaccia, interagiscono con l'utente e sono spesso dotati di GUI ("spesso" perché anche i server vengono incorpati nell'insieme di sistemi interattivi, ma non hanno GUI nella maggior parte dei casi). In questo caso bisogna minimizzare i tempi di risposta;
- i terzi gestiscono processi che sono legati ad eventi esterni che possono arrivare in qualunque momento e azioni che vengono scatenate in reazione agli eventi. Caratteristica fondamentale è la **tempestività**, nonché la **prevedibilità** degli eventi.

Utile in questo caso è l'utilizzo di alcune **metriche**. In generale si seguono alcuni principi a prescindere dal sistema:

- **equità** nell'assegnamento della CPU. I processi si contendono la CPU, pertanto dovremmo cercare di fornire lo stesso servizio a tutti i nostri processi. Vedremo anche che non tutti i processi sono uguali, ma assumendo che questi lo siano ci aspetteremo di fornire lo stesso servizio a tutti, e quindi di fornire una frazione di utilizzo di CPU uguale per tutti. In generale, non deve scattare un meccanismo di **inequità** tra i processi;
- **bilanciamento** nell'uso di risorse. In questo ragionamento si possono includere risorse I/O, oltre alla CPU e alla memoria. L'ideale in processi CPU bound è mantenere le risorse tutte pienamente in uso. Si può rischiare di non massimizzare l'uso se si fanno determinate scelte. Una scelta per bilanciare l'uso di risorse è scegliere prima processi I/O bound, in quanto non monopolizzano la CPU.

Metriche per sistemi batch

Throughput

Il **throughput** del sistema batch è inteso come il **numero di job gestibili in unità di tempo**. L'idea è che vogliamo far lavorare al meglio il sistema batch, e per verificare ciò possiamo contare quanti job possono essere svolti in una giornata, ad esempio. L'ideale sarebbe massimizzare il throughput del sistema.

Non si tratta tuttavia dell'unica metrica da tenere in considerazione. Quando un job viene fornito al sistema, quest'ultimo elabora la richiesta e decide se immettere il job o meno. Il caso realistico è che il job viene messo in coda e verrà selezionato quando l'algoritmo di scheduling lo rimuoverà dalla coda.

Turnaround

Ha senso pensare al tempo di completamento di ogni singolo job, ma in generale vogliamo **minimizzare la media dei tempi di completamento** dei job che costituiscono un batch. Questo permette di fornire un ottimo servizio per tutti. Il rischio infatti di massimizzare solo il throughput risiede nella possibilità di creare un sistema che non consideri altri aspetti.
In una coda di job potrebbero entrare molteplici job durante l'esecuzione. Un sistema che cerchi di ottimizzare il throughput potrebbe selezionare sistematicamente job brevi, anche se sono appena arrivati in coda. Questo aumenta il throughput, perché tale approccio aumenta il numero di job svolti. Se però osserviamo il servizio fornito ai job più lunghi, stiamo fornendo un pessimo servizio: il **turnaround**, ossia il tempo di completamento dall'ingresso in coda, è molto grande per job più lunghi, in quanto non vengono selezionati.

Tempo di attesa

Nel tempo di attesa del completamento dell'operazione, possiamo realmente individuare due intervalli temporali:

- il tempo di completamento del job, ossia il tempo che passa dall'inizio alla conclusione dell'esecuzione. Questo tempo è intrinseco del job, a prescindere dal sistema di scheduling adottato;
- il **tempo di attesa**, ossia il lasso di tempo che passa dall'inserimento in coda allo scheduling del job. Esso è influenzato dall'algoritmo di scheduling.

Vogliamo allora minimizzare anche il tempo di attesa.

Scheduling nei sistemi batch

Criterio FCFS

Il primo algoritmo di scheduling analizzato è l'algoritmo FIFO: semplicemente viene estratto per primo il processo che è stato inserito meno recentemente in coda. L'algoritmo è tipico di sistemi non-preemptive, ossia senza prelazione.
Supponiamo di avere tre processi P_1, P_2, P_3, arrivati nell'ordine in cui sono scritti. Tracciamo un **diagramma di Gantt**, che marca l'utilizzatore di una risorsa (la CPU) in ogni istante.
Scheduliamo P_1, che userà la CPU fino a quando non ha concluso. Infine, scheduliamo P_2 e P_3.
Supponiamo P_1 abbia durata di esecuzione pari a 24, mentre P_2, P_3 hanno durata 3.
Lavoriamo adesso sulle metriche definite precedentemente. Il tempo di attesa medio è dato da:

$$\frac{0+24+27}{3} = 17$$

Il tempo di completamento medio è dato da:

$$\frac{24+27+30}{3} = 27$$

Ora come ora non ci interessa il significato di questi numeri (sono secondi? Millisecondi?), ma li usiamo come metrica di riferimento.

Shortest Job First (SJF)

Utilizziamo ora uno **scheduling per brevità**. Affinché sia applicabile, necessitiamo di un'informazione bonus: bisogna conoscere il tempo impiegato da ogni lavoro. La domanda che ci poniamo è se è possibile avere quest'informazione a priori. Su un sistema batch, quest'informazione può essere stimata. In altri termini, nei sistemi batch e nelle loro applicazioni moderne i tipi di job che un dato sistema si ritrova ad eseguire spesso sono di tipo ricorrente (transazioni e-commerce, bancarie, ecc.), e poiché sono dello stesso tipo, il loro tempo può essere categorizzato. Possiamo quindi fare delle stime in virtù delle esecuzioni di job precedenti dello stesso tipo.

L'idea di conoscere quindi a priori il tempo di esecuzione è un'ipotesi accettabile. Daremo quest'ipotesi per buona nei sistemi batch. Lo scheduler privilegia semplicemente i lavori che terminano prima. Avendo i processi P_1, P_2, P_3, P_4, P_5 di durata rispettivamente $2, 4, 1, 1, 1$ e tempo di arrivo rispettivamente $0, 0, 3, 3, 3$ si ottiene un tempo di attesa medio pari a:

$$\frac{0+2+3+4+5}{5} = 2.8$$

Questo perché i primi due processi sono arrivati prima. Il tempo di attesa del processo P_3 è pari a $6-3$, ossia al tempo totale di completamento dei primi due processi meno il tempo di ingresso in coda del processo. Analogamente si calcola per P_4, P_5.

Possiamo dimostrare che l'algoritmo SJF è un algoritmo **ottimale**, in quanto stiamo ordinando per durata di esecuzione i processi, riducendo in questo modo il tempo medio di attesa. Tuttavia, questo è corretto se **tutti i processi arrivano nello stesso momento**. Le condizioni tuttavia non sono sempre riscontranti nella realtà: non è detto infatti che i processi **arrivino tutti insieme**. Dobbiamo modellare situazioni in cui all'inizio la coda è vuota, e poi si riempie gradualmente. In particolare, se i tempi di arrivo non sono tutti uguali, l'algoritmo SJF non è ottimale. Lo dimostreremo con un controesempio, usando l'esempio precedente. Infatti, un ordine diverso di schedulazione sarebbe P_2, P_3, P_4, P_5, P_1. Si ottengono così i tempi di attesa, ordinati a partire da P_1, a P_5, come segue: $7, 0, 1, 2, 3$. La media dei tempi di attesa è

$$\frac{7+0+1+2+3}{5} = 2.6$$

che dimostra che l'algoritmo SJF non è ottimale.

Shortest Remaining Time Next (SRTN)

In sistemi preemptive è possibile scegliere un altro processo. Allora potremmo eseguire nuovamente la scelta SJF in virtù del tempo rimanente dei processi in coda.

Questo algoritmo per brevità introduce l'idea di applicare la prelazione. Se il processo attualmente in esecuzione è stato scelto, il processo è il più piccolo per tempo rimanente. Quando un nuovo processo arriva in coda, si controlla il tempo di esecuzione necessario per il nuovo candidato e lo si confronta con il processo in esecuzione.

Esempio

Supponiamo di avere i seguenti processi:

Processo	Arrivo	Durata
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Utilizzando l'algoritmo FCFS, i processo vengono eseguiti in ordine di indice a partire da 1. Quindi si ottiene un tempo di attesa medio pari a:

$$\frac{0+5+7+7}{4} = 4.75$$

e il tempo medio di completamento è:

$$\frac{7+9+8+11}{4} = 8.75$$

“../../pablo_material/!Images/10 - Scheduler/image.png” non trovato.

Utilizzando l'algoritmo SJF, prima viene eseguito P_1, poi P_3, poi P_2 e P_4. Si ha allora che il tempo d'attesa per ogni processo, rispettivamente da P_1 a P_4, sono $0, 6, 3, 7$, con media 4. I tempi di completamento, invece, sono $7, 10, 4, 11$, con media 8.

“../../pablo_material/!Images/10 - Scheduler/image-1.png” non trovato.

Utilizzando infine l'algoritmo SRTN, si hanno tempi di attesa rispettivamente $9, 1, 0, 2$ e di completamento $16, 5, 1, 6$. Si ha allora che la media dei tempi di attesa è pari a:

$$\frac{9+1+0+2}{4} = 3$$

mentre il tempo di completamento medio è:

$$\frac{16+5+1+6}{4} = 7$$

Per calcolare correttamente i tempi di attesa, basta sottrarre il tempo attuale con il tempo di arrivo (nel caso di SRTN, si sommano tutti i tempi di attesa dopo che il processo entra in coda). Per calcolare correttamente il tempo di completamento, basta sottrarre il tempo di completamento di esecuzione del processo al tempo di arrivo.

11 - Scheduling in sistemi interattivi

Scheduling in sistemi interattivi

Nei sistemi interattivi dobbiamo garantire che se un processo riceve un comando, deve nel più breve tempo possibile rispondere al comando. Bisogna quindi minimizzare i tempi di attesa.

Round Robin

L'algoritmo di base per questi sistemi è l'algoritmo di scheduling **round-robin** (**RR scheduling**). L'idea di base è quella di forzare la prelazione sul concetto di quanto di tempo. Fondamentalmente l'algoritmo RR utilizza il principio FCFS visto precedentemente, ma aggiunge la prelazione. Si assegna al processo in testa della queue la CPU per un timeslice. Valori tipici per il time slice sono tra i 20 e i 50 millisecondi.

Supponiamo che un processo che viene schedulato esegua una chiamata bloccante, ad esempio richieste I/O. Se il processo in generale si blocca spontaneamente sulla sua azione, e quindi non completa nel suo quanto di tempo, viene assegnata la CPU al prossimo processo in coda. Il processo allora viene inserito in una coda di processi bloccati, e verrà poi reinserito nella coda di processi ready.
Se il quanto di tempo per il processo è scaduto, non transita in uno stato di bloccato, pertanto viene direttamente reinserito nella coda dei processi ready.

Più è grande il quanto di tempo, più aumenta la probabilità che il processo si blocchi su I/O. Si rende infatti più probabile che venga catturato nel quanto di tempo il CPU burst. Se il quanto di tempo è sempre più grande, l'algoritmo RR si comporta come FCFS.
Usare invece quanti di tempi piccoli aumenta la possibilità di eventi di prelazione, ossia di eventi in cui il sistema operativo toglie al processo la CPU. Il costo della prelazione è dovuto al cambio di contesto. Se si usa un quanto di tempo eccessivamente piccolo, il numero di context switch aumenta, e questo aumenta l'overhead. Tuttavia, è necessario ragionare sulla seguente. Quando un processo lascia la CPU, viene rimesso in coda, e otterrà la CPU quando ritornerà a essere in testa. Supponendo di avere n processi in totale e un quanto di tempo q, allora il processo prima di poter ottenere nuovamente la CPU attende $(n-1)\cdot q$ millisecondi, ossia il tempo necessario affinché ogni processo utilizzi il suo quanto di tempo. Non è detto chiaramente che il processo utilizzi del tutto il quanto di tempo, ma può anche terminare prima; tuttavia, il caso peggiore è che ogni processo utilizza l'intero quanto q. Se q è più piccolo, il valore $(n-1)\cdot q$ è minore, ma se è troppo basso si eseguono troppi context-switch.

Scheduling a priorità

Lo scheduling RR assume che ogni processo sia di uguale importanza, ma questo non è sempre vero. Se per ogni processo si individua una **priorità**, si adotta lo **scheduling a priorità**. Tale algoritmo segue una regola di base: si assegna la CPU al processo con più alta priorità. La priorità viene specificata tramite un'etichetta. Dati più processi, l'idea è quella di scegliere i processi con priorità più alta. I processi a priorità più bassa potranno usare la CPU solo se in quel momento non sono presenti in coda processi con priorità maggiore. Questo non significa che i processi di maggior priorità sono terminati, ma questi potrebbero o non essere interessati a usare la CPU oppure sono in stato bloccato.
Per stabilire la priorità, possiamo adottare diverse vie:

- si possono etichettare i processi in maniera statica o dinamica. Questo ad esempio dipende dal contesto: in un contesto militare i processi avviati da un individuo più in alto in gerarchia avranno una priorità più alta rispetto ai processi inizializzati da un altro più in basso nella scala gerarchica. Linux e UNIX ad esempio hanno il comando `nice`, che permette l'esecuzione di un programma con una priorità di scheduling modificata in virtù di un parametro nel range $[-20, 19]$;
- si possono favorire i processi I/O bound, in quanto **schedulare i processi I/O conviene rispetto a preferire i processi CPU bound**. Questo è dovuto al fatto che processi che eseguono molte richieste I/O trascorrono molto tempo ad attendere che la richiesta I/O venga completata. Un processo di questo tipo dovrebbe avere subito la CPU, in quanto può cominciare la successiva richiesta I/O e lasciare la CPU per altre computazioni. Se invece facciamo attendere il processo I/O, esso aspetterà molto tempo in memoria senza motivo: al tempo di attesa per l'ottenimento della CPU si aggiunge anche il tempo di attesa dell'elaborazione della richiesta I/O. Le priorità a tali processi allora possono essere assegnate con il calcolo $\frac{1}{f}$, dove f è la frazione del quanto di tempo utilizzato dal processo. Se ad esempio il processo ha usato solo 1 millisecondo su 50 prima di andare in attesa per una richiesta I/0, allora avrà priorità 50, mentre se un processo è stato in esecuzione per 25 millisecondi avrà priorità 2 (ossia $\frac{25}{50}$);
- utilizzare la priorità con il meccanismo di Shortest Job First.

Risulta conveniente inoltre raggruppare i processi in **classi di priorità** e usare uno scheduling per le classi:

“../../pablo_material/!Images/11 - Scheduling in sistemi interattivi/image.png” non trovato.

L'algoritmo di scheduling in questo caso funziona nel seguente modo: fintanto che esiste un processo nella classe di priorità 4, si esegue un RR-scheduling. Quando non ci sono più processi nella classe 4, si esegue lo stesso approccio per la classe 3.
Notiamo che se le priorità non sono aggiustate occasionalmente, i processi di bassa priorità potrebbero andare in **starvation** in quanto non otterranno mai la CPU. Una tecnica che si utilizza per evitare la starvation è l'**aging**. Ci sono diversi modi per applicare l'aging, ma si

basano sempre sull'idea che la priorità di un processo aumenti man mano che quest'ultimo attende nella ready queue.
Poiché per ogni classe di priorità si utilizza l'approccio round-robin, è necessario specificare anche i quanti di tempo per coda. In generale, processi CPU-bound necessitano quanti di tempo più grandi, in modo tale da ridurre i context-switch. Tuttavia, quanti di tempi grandi significa un minor tempo di risposta, cosa non ideale per processi I/O-bound. Consideriamo nuovamente l'esempio precedente con quattro classi di priorità. Basta assegnare quanti di tempo q_i tali che:

$$q_4 < q_3 < q_2 < q_1$$

Idealmente nella coda di processo più bassa si potrebbe anche utilizzare un approccio FCFS.

Shortest Process Next (SPN)

Possiamo applicare lo stesso approccio SJF ai processi interattivi. Basta identificare la durata del prossimo burst di CPU. Si potrebbe così prevedere questa durata, stimando sui burst precedenti. Generalmente, i processi I/O bound presentano dei burst di CPU più brevi. Quindi, sapendo a priori quanto sarà il CPU burst successivo potremmo assegnare la CPU al prossimo processo. Stiamo applicando quindi l'idea precedente, utilizzando una priorità dinamica in virtù anche del comportamento del processo. Rimane solamente come prevedere il prossimo burst. Per prevedere il prossimo burst, possiamo applicare una **media pesata**.
Avendo delle misurazioni reali dei precedenti CPU burst, vogliamo analizzare le varie sequenze di CPU burst e le loro durate. Dato allora i burst T_i, per prevedere T_{i+1} dobbiamo formulare una stima, che chiameremo S_{i+1}. Si ha che:

$$S_{i+1} = S_i \cdot (1 - a) + T_i \cdot a$$

dove a è un parametro.
Essenzialmente si tratta di una media pesata tra la stima precedente e l'evento reale precedente. Il parametro a rappresenta il peso che viene dato a entrambi i valori T_i e S_i. Un valore di a maggiore dà più peso a T_i che ad S_i, permettendo alle stime di adeguarsi maggiormente ai recenti CPU burst. La rapidità ad adeguarsi tuttavia può essere rischiosa: i cambiamenti reali potrebbero essere disturbi, e se le misurazioni reali sono molto rumorose, un peso più piccolo permetterebbe alla misurazione di non dipendere dai disturbi. Un valore a medio quindi è meno soggetto a disturbi e cambiamenti fuori scala e locali, e quindi quasi immune a eventi singoli che non hanno influenza quindi a lungo termine. Generalmente il valore di a adottato è $\frac{1}{2}$, in quanto per trovare il nuovo valore S_{i+1} basta fare la media dei valori S_i, T_i.

Scheduling garantito

Un approccio diverso allo scheduling consiste nel promettere **equità** nell'utilizzo della CPU da parte dei processi. Nel caso di sistemi multi-utente, si garantisce che ogni utente abbia la stessa frazione di potenza di CPU. Per garantire ciò, il sistema deve tener traccia di quanta

CPU è stata usata da ogni processo, in particolare calcolando il rapporto:

$$\frac{\text{tempo di utilizzo CPU effettivo}}{\text{tempo di utilizzo CPU assegnato}}$$

Si sceglie così il processo che ha il rapporto **più basso**: se il rapporto è basso, significa che il processo ha usato la CPU in maniera minore rispetto agli altri, in virtù del tempo assegnatogli.

Scheduling a lotteria

Il sistema illustrato precedentemente potrebbe essere difficile da implementare. Esiste una soluzione più semplice: lo **scheduling a lotteria**. Si assegna un certo numero di biglietti ai processi. Lo scheduling a lotteria implementa le priorità tenendo conto della quantità di biglietti, attribuendo una priorità più alta ad un processo semplicemente dandogli più biglietti. Per schedulare il prossimo processo, la CPU fa un'estrazione.
Lo scheduling a lotteria ha alcune proprietà interessanti:

- è uno scheduling **reattivo**: un processo può essere selezionato in virtù dei biglietti che possiede;
- permette la **collaborazione tra processi**: un processo può cedere i suoi biglietti ad un altro, aumentando le probabilità che quest'ultimo venga scelto;
- può risolvere problemi più difficili. Ad esempio, un server di streaming di video può avere processi che forniscono video a framerate differenti, ad esempio due processi che trasmettono a 10 e 20 fps. Possiamo allora assegnare rispettivamente 10 e 20 biglietti: in questo modo, la CPU è divisa in maniera approssimativa con la corretta proporzione.

Scheduling fair-share

Lo **scheduling fair-share** attenziona lo scheduling in un sistema multiutente. Gli approcci focalizzati ai processi non sono focalizzati sugli utenti. Se suddividiamo l'assegnazione su tutti i processi la CPU, potrebbero esserci alcuni utenti che hanno più processi. Un utente con più processi può ricevere maggiore percentuale di CPU. Si ha allora uno scheduling assegnato non ai singoli processi, ma a **gruppi** di processi. Si raggruppano quindi i processi per utente, garantendo equità sugli utenti. Linux dà l'opportunità di raggruppare i processi in gruppi, e in questo caso l'amministratore può stabilire le proprie regole di assegnamento di CPU.

Scheduling nei thread

Per i thread utente, lo scheduler del kernel, poiché non ne conosce l'esistenza, ignora i thread utente. Per lo scheduler del sistema run-time possono essere utilizzati tutti gli algoritmi non-preemptive. Si possono anche implementare algoritmi di scheduling personalizzati.
Per i thread del kernel, o si considerano tutti i thread uguali, o si considera l'**appartenenza al processo**. Anche i thread possono avere priorità, e nulla vieta di avere priorità su thread di uno

stesso processo. Si esegue quindi una doppia scelta: si sceglie il processo, e del processo si sceglie un thread. Considerare l'appartenenza al processo è importante, in quanto il context switch tra due thread dello stesso processo è meno pesante dovuto all'assenza della riprogrammazione della MMU.

Scheduling in sistemi multicore

L'elaborazione in sistemi multicore può essere di due tipi:

- multielaborazione **asimmetrica**;
- multielaborazione **simmetrica**.

Nella prima, si hanno n core, e l'asimmetria risiede nel fatto che **non tutti i core hanno lo stesso ruolo**. Esiste un **core master** e tutti gli altri sono **core slave** (o **primario** e **secondari**). Il master gestisce il problema dello scheduling sulle code dei vari core, mentre i secondari eseguono i task. Il vantaggio di avere un unico core master è il fatto che **esso è l'unico core che accede alle code**. In un contesto parallelo, ci potremmo porre il problema dell'accesso concorrente alla ready queue dei processi, ma poiché è solo il master che lavora alla ready queue, non è necessario porsi il problema in questo caso. Lo svantaggio, tuttavia, risiede nel fatto che se il numero di core aumenta, il core master potrebbe diventare un **bottleneck**, in quanto il core master deve dirigere ogni core secondario. Proprio per questo motivo, nessun sistema operativo utilizza questo approccio.
Nella seconda tipologia di multielaborazione, si hanno due approcci:

- una **coda unica** dei processi pronti, comune a tutti i core;
- n **code separate**, una per core.

Avere una coda unica richiede l'applicazione del meccanismo di lock. Questo può causare problemi di scalabilità: al crescere del numero di CPU, un core potrebbe essere libero, ma non può scegliere il prossimo candidato, in quanto vi è un altro core che sta già eseguendo lo scheduling. Al crescere del numero di core, l'approccio diventa insostenibile. Avere code separate, invece, permette ad ogni core di avere un suo gruppo di processi, e quando un core diventa libero guarda nella propria ready queue. L'approccio permette chiaramente scalabilità. Infatti, se un processo viene schedulato, la prossima volta conviene che userà la stessa CPU, perché è un modo più efficace di eseguire quel processo. Fondamentalmente si può constatare che la cache di quella CPU con buona probabilità ha tracce di linee di cache utili per quel processo. La **predilezione** di un processo per una CPU permette di aumentare leggermente l'efficienza dell'esecuzione del processo, che è un processo naturale in quanto il processo viene reinserito nella coda della CPU che lo stava eseguendo.
Vogliamo anche garantire un **bilanciamento di carico** nell'utilizzo di code separate. Si utilizzano vari approcci, ma ne vedremo due:

- **migrazione guidata**. Si monitora lo stato delle code e si osserva se esistono sbilanciamenti marcati (un gruppo di processi molto pieno rispetto ad un altro), spostando processi da una coda ad un'altra. Si tratta di una migrazione "guidata", in quanto si attiva un task periodicamente che controlla lo stato delle code. Questo task consiste nell'ispezionare e modificare tutte le code: un core quindi può modificare la coda di altri. Il meccanismo di lock quindi è necessario, e si ha un lock per coda. L'unico caso in cui si ha accesso concorrente è chiaramente nel caso di migrazione, ma le migrazioni non sono molto frequenti. Non si tratta quindi di un meccanismo così bloccante, come l'utilizzo di una coda per tutti i core;
- **migrazione spontanea**. Si applica quando un core ha la ready queue vuota. Il core attiva un meccanismo simile a quello precedente, scansionando le code degli altri core, focalizzandosi su quelle più piene, e prendendo dei processi.

I due metodi agiscono in fasi differenti: pur avendo una sorta di bilanciamento tramite la migrazione guidata, si potrebbe comunque avere una serie di processi che si bloccano rapidamente, svuotando la ready queue di un core. In queste situazioni, il core può utilizzare la migrazione spontanea per essere in esecuzione. Entrambi i metodi vengono quindi usati insieme.
Il meccanismo è una **predilezione debole**, in quanto non vi è un meccanismo che forza la predilezione di un processo ad un core. Una **predilezione forte** invece consiste nel meccanismo per cui un processo può essere forzato ad essere eseguito su una data CPU, vincolando un processo ad un core o a un gruppo di core. Questo va in direzione opposta al bilanciamento, ma se questa forzatura viene applicato ad un ristretto gruppo di processi, questo approccio è accettabile.

Scheduling nei sistemi operativi attuali

Windows

Windows utilizza uno scheduler con code di priorità, dotato anche di un sistema di **priorità dinamiche**. In particolare, il sistema individua la natura del processo, ad esempio vedendo con quali dispositivi interagisce, declassando i processi che sono più CPU-bound e aumentando entro un certo range la priorità del processo più I/O-bound. A parità di classi di priorità, Windows dà più priorità ai processi che sono in primo piano. Al processo corrispondente alla finestra in primo piano viene assegnato un quanto di tempo più ampio. Ci sono anche euristiche per evitare il problema di inversione di priorità.

Linux

Lo scheduler di Linux non gestisce processi o thread, ma gestisce **task**. Il **task** è una generalizzazione dei processi e dei thread. I task sono dei flussi di esecuzione a cui è associata una certa quantità di risorse, quali spazio di indirizzamento, set di file aperti, variabili d'ambiente e la gestione dei segnali. Sono quindi informazioni che circondano il flusso di esecuzione. Ad

ogni flusso di esecuzione, il sistema decide quali di queste risorse devono essere associate. Si tratta di una estraneizzazione della differenza tra processo e thread. Thread e processo sono quindi casi particolari del task. La chiamata `fork` usata nei sistemi UNIX, o `thread_create()`, esistono su Linux, ma sono invocazioni wrapper su una chiamata di sistema che si chiama `clone`. Le due chiamate specificano più precisamente quali sono le risorse da condividere. La `clone` inoltre permette in modo granulare di definire quali sono gli elementi da condividere tra i task. Ad alto livello chiaramente Linux distingue tra thread e processo in virtù del fatto che Linux implementa POSIX.

Linux permette di **cambiare scheduler** in quanto lo scheduler è un modulo del kernel. Linux ha già uno scheduler di default, il **Completely Fair Scheduler** (**CFS**), ma esso può essere cambiato. Il CFS è un esempio di scheduler evoluto che muta delle idee provenienti dallo scheduler garantito (ossia che garantisce la stessa quantità di risorse a tutti i processi). Esso implementa il **virtual runtime**, ossia un modo di tracciare per quanto tempo il task ha usato la CPU. Il meccanismo di base è semplice: viene scelto il processo con il virtual runtime più piccolo. La struttura dati "coda dei processi" dev'essere idonea alla scelta di essere sempre quello più piccolo; per questo motivo, la coda dei processi è implementata con un **albero rosso-nero**. Quando un task è in esecuzione, il virtual runtime del task aumenta. Il meccanismo di prelazione allora avviene quando il processo in esecuzione ha un virtual runtime che non è più il minimo tra i task in esecuzione. Tale situazione determina il context switch. Il context switch di conseguenza è **forzato** in virtù del virtual runtime. Per evitare che il virtual runtime vada in overflow (contesto server), esso viene periodicamente **normalizzato**, sottraendo il minimo dei runtime a tutti i valori di virtual runtime. Per gestire inoltre nuovi processi che entrano in esecuzione, il suo virtual runtime viene assegnato con una etichetta pari al virtual runtime minimo tra tutti i processi.
Il modo in cui il virtual runtime viene usato per gestire la priorità è il seguente. Ad ogni processo viene assegnato un **fattore di decadenza** in virtù della sua priorità (più alto se la priorità è più bassa e viceversa). In questo modo quando viene aggiornato il virtual runtime, i processi con priorità più bassa vedono un virtual runtime che cresce più rapidamente rispetto agli altri. Se un fattore di decadenza quindi è pari a 2, rispetto a un altro processo che ha un fattore pari a 1, al processo viene assegnata la CPU per metà del tempo. Questo inoltre garantisce anche di evitare la **starvation** di un processo: se ci sono 10 processi ad altissima priorità e un processo a bassissima priorità, i primi ad essere attenzionati sono i 10, ma l'utilizzo della CPU aumenta il virtual runtime di questi processi. Pertanto **anche i processi a bassa priorità hanno la garanzia di utilizzo della CPU**.
Uno scheduler di questo tipo favorisce naturalmente i **task I/O-bound**. Inoltre, **il concetto di quanto di tempo non esiste in tali sistemi**. Infine, Linux implementa i **cgroups** (**control groups**) per garantire un fair share nell'utilizzo della CPU nel caso di multiutenza.

12 - Gestione della memoria

Gestione della memoria centrale

Il problema nella gestione della memoria riguarda l'esposizione della memoria centrale ai processi. Soluzioni moderne introducono un livello di astrazione di accesso alla memoria da parte del processo più alto rispetto alle prime soluzioni adottate.

Soluzione iniziale: nessun livello di astrazione

Inizialmente le esigenze erano semplici: si aveva un solo processo per volta in memoria. I sistemi operativi offrivano come interfaccia il prompt (tipico di MS-DOS). Inoltre, non si avevano scheduler visto che il processo era l'unico in memoria: se esso terminava, si apriva la shell. Il problema della gestione della memoria era quasi inesistente. Nei sistemi senza astrazione (MS-DOS) si aveva solo la necessita di ospitare in RAM il sistema operativo e il programma utente. La memoria quindi veniva suddivisa in modi semplici:

1. caricare nella parte bassa il codice e le strutture dati del sistema operativo, mentre nella parte alta il programma utente. Ogni programma poteva accedere agli indirizzi di memoria che contenevano il codice del sistema operativo, con rischi di corruzione;
2. caricare il codice del sistema operativo in ROM, mentre le strutture dati venivano caricate in RAM assieme al programma utente. Stavolta, tuttavia, gli indirizzi alti erano riservati al sistema operativo, mentre il processo utente usava indirizzi fisici bassi. Non si ha la possibilità di accedere al codice del sistema operativo, in quanto esso era caricato in ROM;
3. si usa un software presente in ROM, antenato del **BIOS** (si occupa del bootstrap e dell'uso dei primi dispositivi), che aiuta a caricare il sistema operativo in una regione bassa della RAM. La parte rimanente della memoria veniva usata per l'esecuzione del programma utente.

Esistono problemi più semplici e primordiali nell'includere un ulteriore programma all'interno di questo sistema:

- si ha un problema nella **gestione della rilocazione**. Supponiamo che in una RAM ci sia il sistema operativo caricato in parte alta, e i programmi nella restante parte. Supponiamo di avere due processi, identificati con due **partizioni di memoria**: una che parte dall'indirizzo fisico 0, e l'altra dall'indirizzo fisico $1.000.000$. Il problema risiede nel codice dei programmi dei processi. Quando il codice binario di un processo viene creato, il compilatore deve ricevere il **range di indirizzi su cui lavorare** affinché il programma lavori sul range di indirizzi della partizione di memoria associata al processo. In questo modo il compilatore riceve correttamente gli indirizzi fisici, in quanto il codice compilato fa riferimento a degli

indirizzi di memoria che potrebbero non combaciare se il processo viene eseguito su una partizione di memoria differente. Il problema della rilocazione può essere gestito come segue:

- a **compile-time**: si informa il compilatore della partizione di memoria su cui lavorare. La soluzione è **scomoda**: bisognerebbe stabilire per ogni programma in esecuzione la sezione di memoria su cui dev'essere eseguito;
- tramite **rilocazione statica** in fase di caricamento. Il compilatore quando crea il codice di programma immagina di lavorare su indirizzi canonici (detti **indirizzi logici**). Gli indirizzi lavorano in un range $[0, \text{max_indirizzo}]$ (dove max_indirizzo è determinato in base alle esigenze del programma). Quando poi il codice del programma viene caricato in una partizione di memoria, il loader piazza il codice nel segmento di memoria e corregge il codice. Il loader quindi si occupa di **manipolare il codice del processo**: quando un'istruzione utilizza un indirizzo assoluto, si traduce l'indirizzo assoluto del codice (un indirizzo logico) all'indirizzo corrispondente. Tale indirizzo corrispondente si ottiene aggiungendo un valore di **offset** in virtù dell'indirizzo a partire dal quale si sviluppa la partizione di memoria assegnata al processo. Questo chiaramente **rallenta la fase di loading**, ma è un processo che viene fatto una tantum.

I meccanismi proposti chiaramente non aggiungono nessun meccanismo di protezione della memoria: un processo potrebbe accedere alla regione di memoria di un altro processo, anche tramite la rilocazione statica (ad esempio, se il codice è malevolo o ci sono bug nella manipolazione del codice del processo). In generale, non è sufficiente stabilire che indirizzi di memoria al di fuori della partizione di memoria associata al processo non possano essere generati, ma bisogna implementare dei meccanismi di protezione di accessi.
Un primo meccanismo è il **lock and key**, implementato inizialmente su IBM 360. Si utilizzano delle chiavi di protezioni, sequenze di bit con cui venivano etichettate aree di memoria. La memoria fisica era suddivisa in aree di memoria nell'ordine di qualche kilobyte. La partizione di un processo poteva richiedere un certo numero di aree di memoria. Una tabella all'interno della MMU manipolata dal sistema operativo associava ad ognuna di queste aree una chiave a quattro bit che ne garantiva la unicità. La chiave poteva essere associata ad ognuno dei blocchi dell'area di memoria associata ad un processo. La CPU aveva al suo interno un registro apposito che conteneva una combinazione di quattro bit corrispondente all'attuale chiave utilizzata dalla CPU. Ogni volta che la CPU veniva assegnata ad un processo, il registro veniva popolato con la chiave del processo attuale. La CPU per eseguire fetch e store ad una determinata locazione di memoria confrontava la chiave con l'etichetta memorizzata nella MMU. Gli accessi in questo modo venivano protetti.

Astrazione tramite spazio degli indirizzi

In modo simile all'approccio di rilocazione statica, il codice dei programmi viene generato in un range canonico di indirizzi logici. Il problema della rilocazione viene gestito dalla MMU, e la

MMU viene pilotata attraverso una coppia di registri:

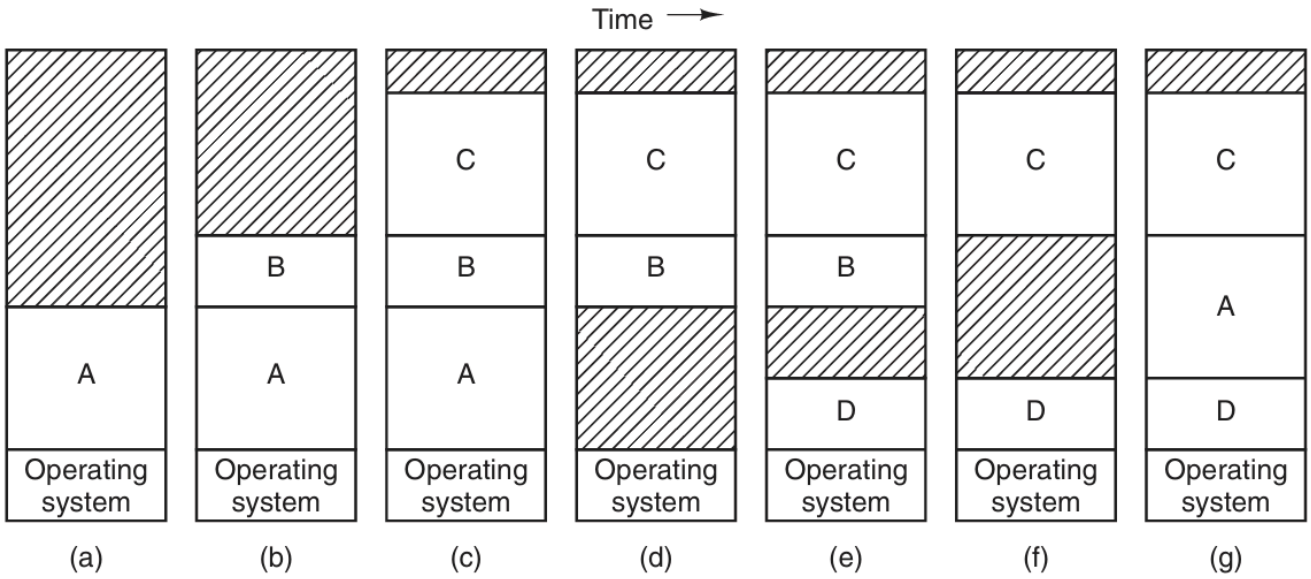
- un **registro base**, che indica la base di partenza della partizione di memoria dedicata al processo;
- un **registro limite**, che indica la dimensione della partizione di memoria dedicata al processo.

Tali registri devono essere aggiornati ad ogni context switch. Gli indirizzi logici dal codice vengono manipolati dalla MMU (in teoria dalla CPU, ma in realtà nei sistemi odierni la MMU gestisce questi accessi). Più precisamente, quando la CPU legge l'indirizzo logico, la MMU verifica se l'indirizzo è inferiore rispetto al registro limite: se lo è, la MMU somma a tale indirizzo il registro base per calcolare l'indirizzo fisico corrispondente. Se non lo è, viene generata una **eccezione**.
Il controllo tramite la MMU permette anche di gestire la protezione di memoria, in quanto la MMU è una componente hardware che corregge ogni volta gli indirizzi logici. Il sistema quindi **non può essere bypassato**: anche se il codice si corrompe o viene generato in modo malevolo, la MMU controlla gli indirizzi in ogni caso. Viene gestito quindi sia la rilocazione sia la protezione, evitando quindi l'accesso ad aree di memoria al di fuori della partizione di memoria dedicata al processo.

Il processore Intel 8088 metteva a disposizione il registro base, mentre il CDC 6600 metteva a disposizione anche il registro limite. Più precisamente, il CDC 6600 implementava i registri base e limite in modo tale che solo il sistema operativo poteva modificarli, mentre l'Intel 8800 era dotato di più registri base.

Swapping

Il livello di multiprogrammazione è seriamente limitato dalla dimensione della memoria centrale, una risorsa limitata all'interno del calcolatore. Per bypassare il problema sono state ideate varie strategie.
La strategia più semplice prende il nome di **swapping**, da non confondere con la partizione di memoria swap dei sistemi UNIX e linux (su Windows si usano esclusivamente i file di swap, e non le partizioni). Semplicemente quando si ha un sovraccarico di memoria si sposta il processo non in esecuzione nel **disco fisso**. In questo modo un processo che è in idle non occupa spazio in memoria. L'operazione di swapping può essere rappresentata nel seguente modo:



Notiamo come il processo di swapping crei dei **buchi in memoria**. Risulta possibile compattare questi buchi utilizzando una tecnica nota come **memory compaction**. La memory compaction semplicemente combina i tanti buchi che si creano con lo swapping in un unico buco semplicemente muovendo tutti i processi agli indirizzi più in basso in memoria (ove questo risulta possibile chiaramente). Tuttavia, questo non viene fatto in quanto è un processo che richiede un po' di tempo: ad esempio, in un sistema a 16 gigabyte di memoria che copia 1 byte in un nanosecondo, la compattazione della memoria richiede 16 secondi.
Da notare chiaramente che ogni processo può richiedere più o meno spazio in memoria. Ogni processo, inoltre, può richiedere più o meno memoria in virtù del fatto che possono accadere **allocazioni dinamiche** durante l'esecuzione del processo. D'altronde, se un processo richiede uno spazio fisso, il sistema operativo alloca solamente lo spazio necessario. Se invece il processo richiede più spazio in memoria:

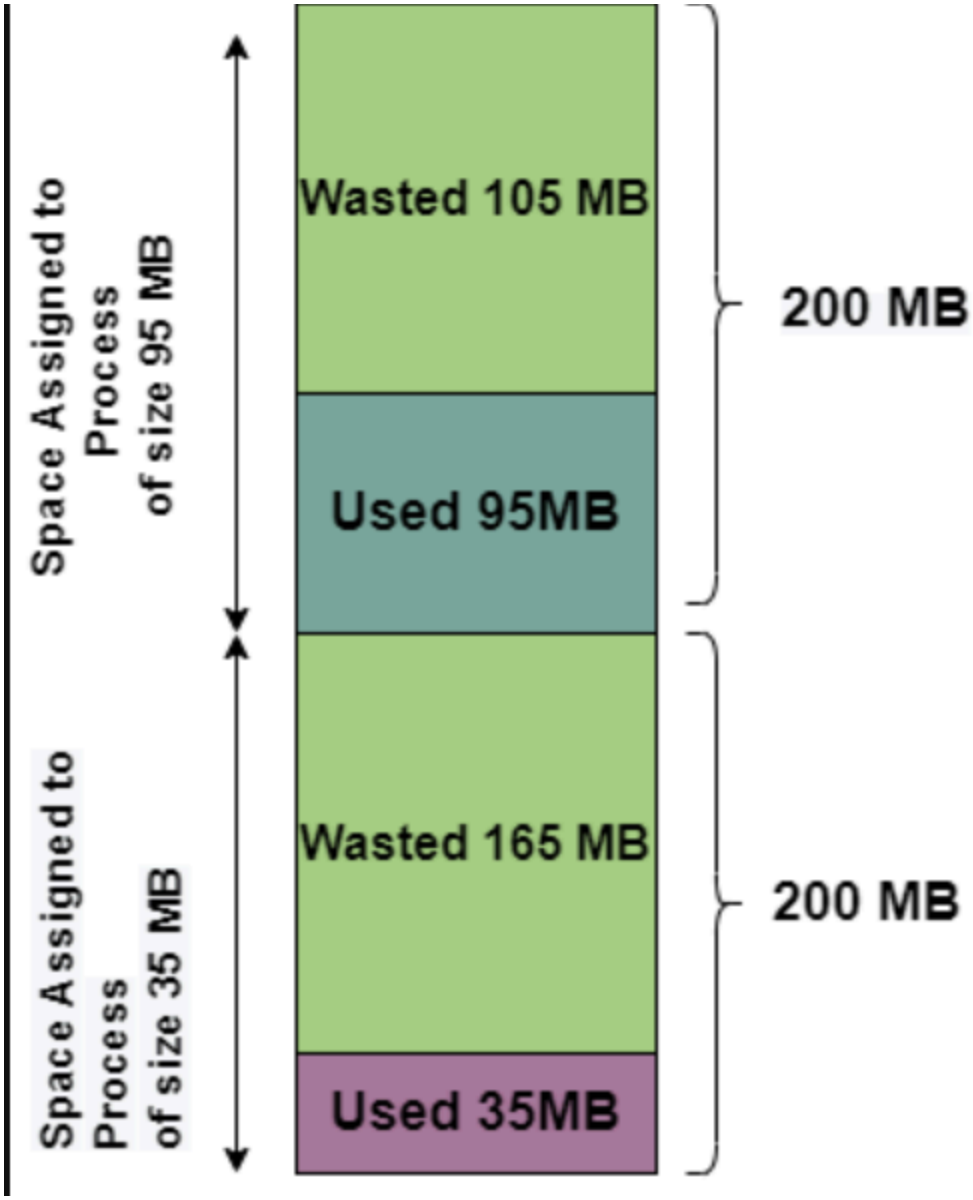
- se sono presenti buchi adiacenti allo spazio allocato per il processo, può essere allocata ulteriore memoria in questo buco;
- se il processo è adiacente ad un altro processo in memoria, allora deve:
  - aspettare che l'altro processo venga swappato;
  - entrare in sospesione, attendendo quindi che si liberi spazio.

Generalmente la scelta del processo da swappare viene compiuta da uno **scheduler di medio termine**, che si distingue dallo scheduler di breve termine, ossia quello visto fino ad ora.
Esistono anche gli **scheduler a lungo termine**, effettuando una scelta all'interno di un batch di job per tirar fuori un job da quelli da eseguire nel batch.

Frammentazione della memoria

Ripetute aggiunte o rimozioni di spazi all'interno della memoria causano la **frammentazione della memoria**. La frammentazione può essere di due tipi:

- **interna**, legata al non utilizzo dovuto ad una allocazione in memoria centrale più grossa rispetto alle esigenze, per motivi quali voler "abbondare" le esigenze di allocazioni dinamiche del processo o arrotondamenti da parte del sistema operativo:



- **esterna**, che si manifesta nel momento in cui un sistema dinamico si trova in situazioni insolite. Supponiamo di avere due aree di memoria libere, una di 4 MB e una di 1 MB. Se un processo richiede 5 MB, poiché il processo richiede memoria contigua è necessario utilizzare la memory compactation, ma si tratta di un processo inefficiente come visto precedentemente. In questo caso si ha un vincolo per la contiguità della memoria. Le tecniche più avanzate riescono a risolvere il problema della frammentazione esterna, ma finché siamo legati al concetto di allocazione contigua è un problema di cui tenere conto.

Gestione dell'allocazione

Il sistema operativo deve gestire nella RAM le parti occupate da quelle libere. Bisogna inoltre decidere come allocare lo spazio: per blocco oppure per misura precisa. Le misurazioni a byte tuttavia richiedono una granularità che è uno svantaggio. Si sceglie quindi di allocare per blocchi.
Le strutture dati che si possono utilizzare per tracciare i blocchi di memoria occupati sono tipicamente due: le **bitmap** e le **linked list**.

Bitmap

Supponiamo di dividere tutta la RAM in blocchi di dimensione k. Avremo allora una bitmap che ha tanti bit quanti i blocchi allocati. Più precisamente, la bitmap marca un blocco occupato con 1 e un blocco libero 0. La bitmap è di supporto nel momento in cui si ha l'esigenza di un processo che ha bisogno di x blocchi in memoria. Si cerca quindi nella bitmap una sequenza di x blocchi contigui da destinare alla richiesta.
Se si utilizza una bitmap è necessario tener conto della dimensione del blocco k, che funge da unità di allocazione ("unità" nel senso di sensibilità di misura): più è piccolo k, maggiore è la dimensione della bitmap. Tuttavia, se k è un valore grande, è più facile che si sprechi memoria quando la dimensione del processo non è un esatto multiplo di k. Inoltre, se un processo richiede n spazi di dimensione k, bisogna cercare all'interno della bitmap una sequenza di n zeri consecutivi, che è solitamente un'operazione lenta.

Linked list

In alternativa alla bitmap è possibile adottare una **lista concatenata**, dove ogni elemento rappresenta blocchi contigui occupati o blocchi contigui liberi. Si potrebbe quindi rappresentare la stessa informazione con una struttura più compatta. Ogni nodo rappresenta o un processo o un "buco", e presenta informazioni relative ai blocchi occupati (ad esempio $[P, 0, 5]$ rappresenta un processo che occupa i blocchi che vanno da 0 a 5). Solitamente, si utilizza l'etichetta P per indicare un processo e l'etichetta H per indicare uno spazio vuoto in memoria (H = hole). Questa struttura dati potrebbe usare meno memoria rispetto alla bitmap.
Nulla vieta di indicare nel nodo anche il PID del processo che è allocato in quella partizione di memoria, cosa che non si può fare nella bitmap. In generale, la linked list può essere **arricchita**. Se arriva una nuova richiesta per un processo Z, basta scorrere la lista alla ricerca di un nodo che rappresenta un buco in memoria sufficientemente grande. Se il buco è grande quanto la richiesta, allora l'etichetta "buco" viene sostituita con l'etichetta "processo"; altrimenti, se il buco è più grande dello spazio richiesto, basta scindere il nodo in due nodi. Se però si creano troppi piccoli buchi che causano frammentazione esterna, lo spreco aumenta.
Chiaramente ci rendiamo conto che è vantaggioso che la nostra lista sia **doppiamente concatenata** per facilitare lo sdoppiamento del nodo. Vogliamo inoltre mantenere la lista **ordinata per indirizzo**. Se due nodi rappresentano due buchi contigui, divetano un unico nodo che rappresenta un buco più grosso: in questo caso, utilizzare una lista doppiamente

concatenata permette di verificare facilmente se un merge è possibile.
Questo fenomeno di "merging" prende il nome di **coalescenza**. Mantenere la lista ordinata per indirizzo permette quindi di eseguire queste fusioni di nodi.

In linea di principio se la lista è ordinata per indirizzo si possono adottare diverse strategie di ricerca per buchi liberi al fine di soddisfare le richieste di allocazione:

- **first fit**, che ricerca il primo buco che permette l'inserimento del processo. Non appena trova un buco di dimensione sufficiente, il processo di ricerca si ferma. Il buco viene quindi diviso in due pezzi: uno per il processo, e uno per lo spazio di memoria non utilizzato. Questo potrebbe portare ad alcuni sprechi di memoria se si creano buchi di dimensione troppo piccola a seguito dell'allocazione di memoria per il processo, ma è un algoritmo molto rapido, perché la ricerca si ferma dopo il primo match;
- **next fit**, che applica il principio first fit ma riprende la successiva ricerca dal buco scoperto nell'ultima ricerca. Alcune simulazioni dimostrano che l'algoritmo next fit si comporta in maniera leggermente peggiore del first fit;
- **best fit**, che cerca il buco migliore per allocare il processo. Il buco migliore sarebbe uno spazio in memoria che ha le stesse dimensioni della richiesta, caso in cui la ricerca si ferma immediatamente. Si tratta di una ricerca però meno efficiente, perché richiede di scansionare l'intera lista se essa è ordinata per indirizzo. Dal punto di vista statistico, tuttavia, la ricerca di tale blocco porta alla creazione di buchi talmente piccoli che raramente potranno essere sfruttati;
- **worst fit**, che cerca invece il peggior blocco da allocare. Si cerca un blocco sufficientemente grande, ma ancora più grande, affinché non si creino buchi rimanenti che siano troppo piccoli per essere sfruttati, tamponando quindi l'effetto best fit.

Solitamente si sceglie l'approccio first fit o next fit in virtù di simulazioni che tengono conto della frammentazione esterna. Potremmo in alternativa tenere due liste separate:

- una lista per i processi;
- una lista per i buchi.

In questo modo, la lista per i buchi potrebbe essere **ordinata per dimensione di buco** e in tal caso l'algoritmo best fit è più efficiente: infatti, non appena si trova un buco sufficientemente grande per il processo la ricerca può fermarsi. First fit e best fit di conseguenza hanno circa la medesima rapidità, mentre next fit diventa senza senso. Inoltre, se la lista è ordinata per dimensione di buco, la coalescenza diventa un processo **dispendioso**.
Infine, nel caso di liste separate esiste anche un altro algoritmo, il **quick fit**. In questo caso si utilizzano liste separate per i buchi, dove ogni lista contiene gli indirizzi a buchi di memoria **di dimensioni tipiche** richieste dai processi, avendo così una lista per i buchi da 4 KB, una per

quelli da 8 e così via. La ricerca del buco diventa estremamente veloce, ma presenta gli stessi svantaggi della lista ordinata per dimensione nel caso di coalescenza.

13 - Memoria virtuale

Memoria virtuale

Per risolvere in maniera ottimale il problema della frammentazione, abbandoniamo l'idea di avere spazi contigui in memoria per i processi. Analizziamo ora la soluzione tramite **memoria virtuale**.
Vogliamo che il nostro processo abbia una CPU a lui dedicata, desiderio assecondato tramite time sharing e virtual CPU tramite salvataggio del virtual program counter e dei registri.
Vorremmo anche che ogni processo abbia uno spazio di memoria suo che non abbia limiti. Questo diventa virtualmente vero con la memoria virtuale. Abbiamo visto che lo spazio di indirizzamento è l'area di memoria che contiene codice e dati. Lo spazio di indirizzamento, che da ora in poi sarà uno spazio di indirizzamento virtuale, in realtà **non esiste**. Gli spazi di indirizzamento virtuali difatti hanno una dimensione che va oltre la dimensione fisica installata sulla macchina. Generalmente parliamo di sistemi a 32 o 64 bit, bit associati alla dimensione dei registri della CPU ma in modo indiretto anche alla dimensione dello spazio di indirizzamento virtuale che ogni processo gestisce. In altri termini, è la taglia degli indirizzi che la CPU può generare. La CPU utilizza l'indirizzo per eseguire call a procedure memorizzate in memoria. In sistemi a 64 bit, gli indirizzi sono numeri a 64 bit. Questo ci dà anche l'idea della dimensione dello spazio di memoria virtuale: con una CPU a 32 bit, ad esempio, si possono indirizzare 2^{32} byte, ossia 4 gigabyte. Ogni processo virtualmente ha uno spazio di 4 gigabyte con una CPU a 32 bit. In uno spazio a indirizzamento a 64 bit, ci sono 2^{64} byte. La memoria virtuale inoltre permette ad un programma di richiedere più memoria di quella fisicamente disponibile, obiettivo inizialmente raggiunto con gli **overlay**. Semplicemente un programma veniva diviso in overlay, e ogni volta che il programma doveva caricare un overlay:

- se esisteva uno spazio libero, caricava l'overlay;
- se non esisteva uno spazio libero, sovrascriveva l'overlay precedente.

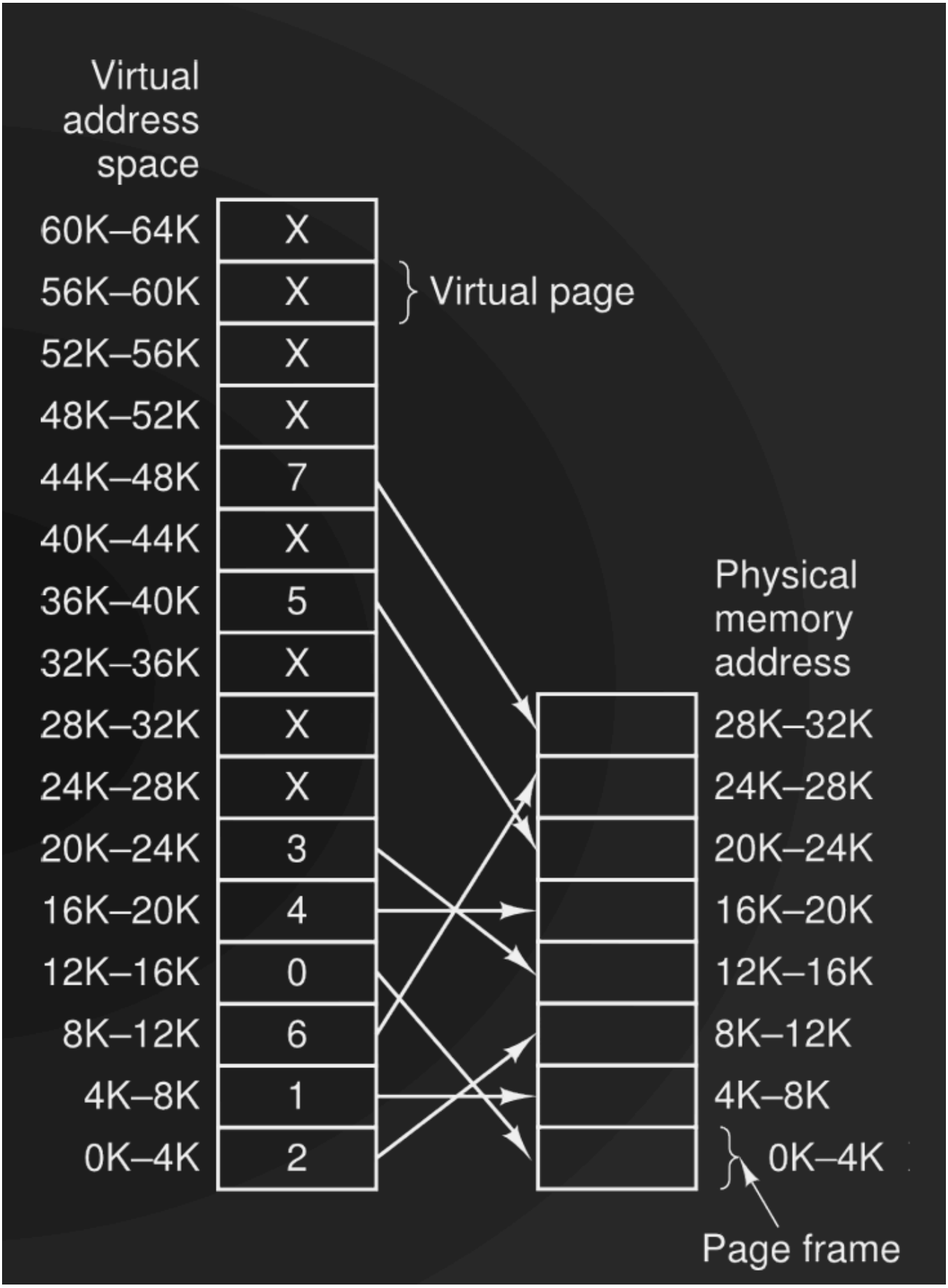
Il sistema era però gestito dal programmatore, ma era un antenato della **paginazione**.

Paginazione

In generale, gli spazi virtuali vanno moltiplicati in virtù del grado di multiprogrammazione. Per gestire lo spazio virtuale, possiamo utilizzare la **paginazione**.
Lo spazio virtuale viene diviso in **pagine**, o pagine virtuali, stabilendo una dimensione di pagina (che tipicamente è di 4 kilobyte). Lo spazio in memoria fisica, invece, viene diviso in **frame**, o frame fisici. L'idea è quella di memorizzare l'informazione delle pagine in RAM. Tuttavia, assumeremo che andremo a memorizzare l'informazione memorizzata nelle pagine **senza il vincolo di contiguità**. Ad esempio, le pagine virtuali $0, 1, 2, 3$ possono essere memorizzate in

frame di RAM differenti. Quando la CPU genera gli indirizzi virtuali, la MMU tradurrà gli indirizzi virtuali, dirottando le richieste sul frame corrispondente alla pagina. La traduzione indirizzo virtuale - indirizzo fisico permette alla CPU di eseguire un fetch corretto, anche se la CPU legge una struttura dati che in virtù della sua natura è contigua (un array, ad esempio). La CPU, quindi, non si rende conto del reindirizzamento. Inoltre, lo spazio di indirizzamento può essere più grande della RAM, in virtù dell'indirizzamento a x bit del sistema (ad esempio, a 64 bit lo spazio virtuale contiene 2^{64} indirizzi possibili, sebbene non si abbiano obbligatoriamente tutti questi byte di RAM).
La paginazione risolve il problema della **frammentazione esterna**. Infatti, anche se i buchi in memoria non sono contigui, essi possono essere utilizzati a prescindere. Tuttavia, non è detto che ogni pagina di un processo venga salvata in RAM. In tal caso, se non vi è spazio in RAM le pagine possono essere memorizzate nel disco. Questo richiama lo swapping, ma in realtà sono tecniche diverse: l'unico aspetto in comune è che si "parcheggia" qualcosa sul disco. Ma nello swapping in realtà transitiamo il processo interamente sul disco. L'approccio, tuttavia, può funzionare solo se il sistema operativo vede che un processo richiede informazioni memorizzate in disco. Se il sistema operativo riconosce che l'informazione viene memorizzata nel disco, si ha un **page fault**, che **non è un errore**. Il page fault è un evento che non permette di completare l'istruzione che ha richiesto l'informazione (un fetch per effettuare un'addizione, i cui operandi sono in una pagina memorizzata in memoria, ad esempio). Il sistema operativo gestisce questo evento prendendo l'informazione che si trova nel disco e portandola in RAM, predisponendo la CPU per eseguire nuovamente l'operazione che ha causato il page fault: l'istruzione, questa volta, verrà eseguita correttamente.

L'unico vincolo da rispettare è che la dimensione della pagina deve coincidere con la dimensione del frame. Sebbene inoltre la frammentazione esterna venga risolta, non viene risolta la frammentazione interna: vi è ancora uno spreco nella pagina finale, perché l'intero spazio non è sempre richiesto. Lo spreco è inoltre proporzionale alla dimensione della pagina finale. Sistemi come Solaris, inoltre, riescono a gestire pagine di dimensione differente, ma la maggior parte dei sistemi operativi utilizza una dimensione fissata delle pagine.



Overbooking della RAM

Il motivo per cui prendiamo più RAM di quanto sia necessario risiede nel fatto che aumentiamo il grado di multiprogrammazione rispetto ad un sistema che non usa la paginazione. Un sistema a 16 gigabyte di RAM che non utilizza la paginazione riuscirebbe a caricare all'incirca 20 processi. Un sistema che utilizza la paginazione può gestire anche 50 volte o pure 100 volte il numero precedente dei processi, in quanto in RAM sono memorizzate solo le informazioni necessarie per il processo. Questo permette di aumentare lo sfruttamento del calcolatore, sebbene l'esecuzione di ogni processo potrebbe essere leggermente più lenta in alcune fasi.

Ogni processo ha inoltre il proprio spazio di indirizzamento virtuale generando indirizzi virtuali tramite richieste fetch e store, ma accede solo a quello. Ogni processo quindi accede solo al suo spazio di indirizzamento virtuale, sebbene il range di indirizzi sia lo stesso per tutti. La protezione di memoria tra i processi tuttavia è **implicita**, in quanto ogni processo utilizza degli indirizzi virtuali, contestualizzati rispetto al proprio spazio di indirizzamento. Per un qualsiasi processo, non vi è modo per accedere alla memoria di altri processi.

Paginazione, MMU

La paginazione è possibile grazie alla **Memory Management Unit** (**MMU**). La MMU gestisce la mappatura degli indirizzi dello spazio virtuale nella memoria RAM, traducendo indirizzi virtuali in indirizzi fisici. Non è detto chiaramente che la MMU riesca a fare ciò per ogni indirizzo: se la MMU riconosce che la pagina non è mappata in RAM si ha un'eccezione ossia la **page fault**, che è una trap per richiedere al sistema operativo di recuperare la pagina dal disco (in quanto è il sistema operativo che gestisce l'area di swap).
La MMU effettua questa traduzione per ogni accesso alla RAM, per cui la MMU gestisce la RAM in maniera hardware. La MMU interviene anche nel caso di una semplice addizione. Si hanno infatti tre accessi:

- fetch per prelevare l'istruzione di addizione;
- fetch degli operandi;

Chiaramente i fetch vengono eseguiti con indirizzi virtuali, che pertanto devono essere tradotti. La MMU quindi esegue tre traduzioni, sebbene sia stata invocata un'unica istruzione.

La MMU utilizza le **paging table**. Ogni processo ha un suo spazio di indirizzamento virtuale. Per rappresentare lo stato di allocazione delle pagine che fanno riferimento a un processo, si utilizza una tabella della pagina per il processo. La paging table permette di capire se una pagina si trova in RAM o meno, e ha una voce per ciascuna pagina virtuale del processo. Per verificare se la pagina è presente una pagina in RAM, è presente per ogni voce un **bit di presenza**:

- se il bit è posto a 0, si ha un page fault;

- se il bit è posto a 1, si ha una voce relativa al frame in cui è memorizzata la pagina in memoria.

Ad esempio, supponiamo di avere una RAM a 32 kilobyte. Supponiamo di voler accedere all'indirizzo virtuale 8196. Per individuare il numero di pagina virtuale, basta dividere l'indirizzo virtuale per le dimensioni della pagina. Se la pagina è di 4 kilobyte, allora si ha che il numero di pagina pari a:

$$\frac{8196}{4096} \approx 2$$

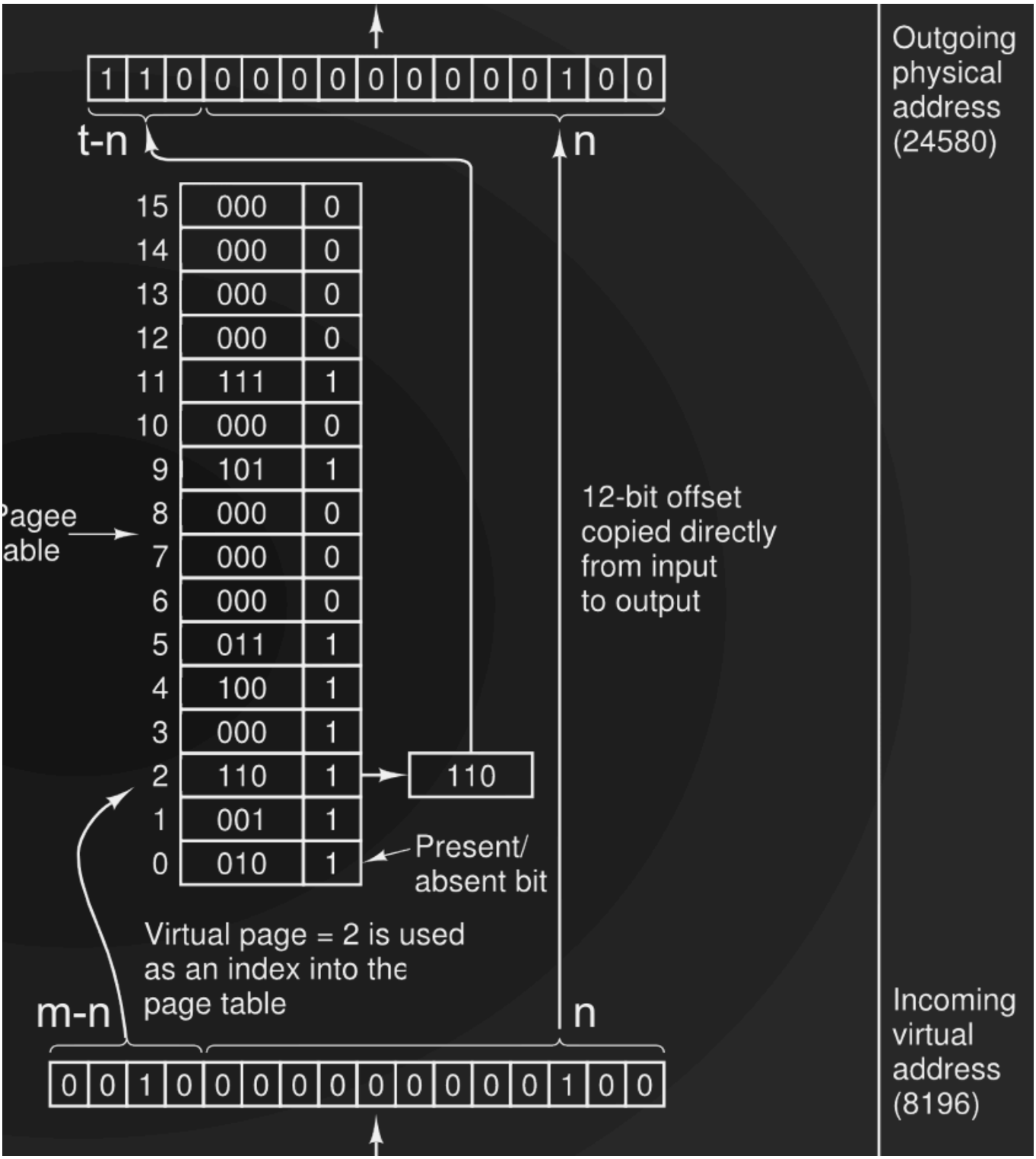
Ci interessa il quoziente e il resto. Il quoziente è 2, e il resto è 4. Il quoziente indica il **numero di pagina**, mentre il resto ci dà un **offset** relativo al numero di pagina. Il **numero di frame** è mappato nella tabella delle pagine. Supponiamo la tabella mappi la pagina 2 al frame 6. Moltiplicando la dimensione della pagina per il numero di frame si ottiene **l'indirizzo fisico del frame**. Poiché noi abbiamo calcolato l'offset, per trovare l'indirizzo fisico basta prendere l'indirizzo fisico del frame e aggiungere l'offset: si ottiene **l'indirizzo fisico** .

Il calcolo analizzato precedentemente può essere eseguito in maniera più intuitiva. Il calcolatore utilizza sempre dimensioni di pagina pari a potenze di due in modo tale che la divisione possa essere eseguita utilizzando shift di bit. Il numero dopo lo shift di bit è il quoziente, ossia il numero di pagina. Supponendo quindi che il nostro indirizzo possa essere rappresentato con m bit, e di shiftare di n posti a destra, si ha che:

- gli $m - n$ bit più significativi rappresentano il numero di pagina;
- gli n bit meno significativi rappresentano l'offset.

D'altronde n rappresenta la dimensione della pagina, ed è anche la taglia dei numeri per indicare univocamente ogni indirizzo della pagina. La dimensione quindi dell'offset è legata alla dimensione della pagina, ed è lo stesso motivo per cui se ho uno spazio a indirizzamento virtuale di 64 kilobyte, si ha bisogno di 16 bit per identificare univocamente ogni byte.
Per eseguire la moltiplicazione, basta shiftare il risultato di n posti a sinistra. Quindi prendiamo il numero di frame e lo shiftiamo a sinistra di n posti. Si ottiene un numero tale che si hanno n bit a destra tutti posti a 0. Di conseguenza, per eseguire la somma basta eseguire una operazione di OR logico con l'offset per trovare l'indirizzo fisico finale.

La parte del calcolo quindi viene eseguita dal calcolatore in modo efficiente.



Esempio

Supponiamo di avere una memoria virtuale di 4 megabyte, e una tabella di paging con 2^{13} voci. I numeri di frame sono a 8 bit. Vogliamo determinare quanto è grande la RAM.

Poiché la memoria virtuale è pari a 4 megabyte, 4 megabyte corrispondo a 2^{22} byte. La dimensione di ogni singola pagina è data da:

$$\frac{2^{22}}{2^{13}} = 2^9 \text{ byte}$$

La dimensione dell'offset è data quindi da 9 bit, perché la dimensione del campo offset è legata alla dimensione di ogni singola pagina. La dimensione di una pagina è uguale alla dimensione di un frame, per cui si evince che il numero totale di byte di RAM è dato da:

$$2^9 \cdot 2^8 = 128 \text{ kilobyte}$$

Struttura della tabella delle pagine

Quando si verifica un page fault, poiché il sistema operativo conosce dove si trova il dato in memoria, il processo deve chiedere aiuto. L'area di swap viene gestita dal sistema operativo ed è utilizzata per memorizzare i dati che vengono salvati nel disco.
La tabella delle pagine deve poter essere uniformata in modo tale che la MMU sappia come accedere ai dati salvati nel disco. La tabella quindi viene utilizzata sia dall'hardware sia dal software. Ovviamente è il sistema operativo che decide se la pagina viene salvata in memoria o meno, decidendo quindi il bit presente o assente. Ogni voce della tabella delle pagine contiene i seguenti campi:

- numero di frame;
- bit presenza;
- protezione;
- bit modificato (o bit dirty);
- bit referenziato (o bit R);
- bit per disabilitazione cache;
- bit di validità o allocazione

Campo protezione

Il sistema operativo può popolare il campo di maschera di protezione. Ogni singola pagina può essere marcata in virtù del fatto che può essere concessa la **lettura** e la **scrittura**. I permessi sono concessi in virtù dell'esecutore del processo. Ad esempio, una pagina per i dati richiesti con la `malloc` può essere letta e scritta, ma uno spazio del codice potrebbe solo essere letto. Il processo non ha accesso chiaramente all'informazione relativa al campo di protezione. Il sistema operativo può quindi marcare l'area in sola lettura. La forzatura del divieto viene imposta dall'MMU. La MMU verifica se l'operazione in esecuzione è una `LOAD` o una `STORE`, e in virtù del tipo di protezione per la pagina, il processo può eseguire l'operazione o viene terminato dal sistema operativo.
Il vincolo forzato di porre in sola lettura il codice eseguibile ha vantaggi nel rendere efficiente l'uso di memoria. Le tecniche fanno anche appoggio alla proprietà di sola lettura del codice di un programma.

Un ulteriore caso di protezione gestito è il caso di **esecuzione**. Il ciclo di esecuzione della CPU è fetch-decode-execute. La CPU può specificare se un fetch è destinato al recupero di un'istruzione. Questo è utile per la protezione contro **code injection**. Alcune tecniche, sebbene semplici, passano dall'idea di inserire all'interno di dati da far arrivare al programma del codice eseguibile. Il codice può essere quindi inserito nello stack o nell'heap. Si porta la CPU ad eseguire quelle porzioni di codice. Il diritto specifico di esecuzione permette di limitare questo tipo di attacchi, limitando che qualcosa che si trova nell'heap o nello stack venga interpretato come istruzioni da eseguire.

Campo bit modificato

Il bit dirty o bit modificato permette di capire se, supponendo che ci sia una pagina sul disco riportata in un secondo momento in RAM (attenzione: non si cancella la copia sul disco. Si perderebbe tempo nella cancellazione, e la pagina potrebbe tornare utile in un secondo momento), la pagina presente in RAM non combacia con la pagina presente sul disco. Se il bit è posto a 0, la pagina si dice pulita: in tal caso, se la pagina dev'essere posta in memoria non si ha la necessita di scrivere nuovamente il valore sul disco. In questo modo, si evita un'operazione di scrittura I/O. Se la pagina invece è stata modificata, il dirty bit è impostato a 1 e pertanto si segnala che si dovrà ricopiare la pagina su disco se questa va parcheggiata nuovamente.

Il bit viene impostato a 1 dalla MMU, mentre quando il sistema operativo copia la pagina il bit viene impostato a 0.

Campo bit referenziato

Il bit di referenziamento, o bit R, viene impostato a 1 non appena la pagina viene referenziata. Con il termine "referenziata" si intende un'operazione generica di fetch o store. Il bit viene posto a 0 quando la pagina viene copiato in RAM, mentre viene impostato a 1 quando viene referenziato. Il sistema operativo può azzerare i bit di referenziamento, ma è solo l'hardware che si occupa di tenere aggiornato quel bit. Il bit viene periodicamente resettato dal sistema operativo, in modo tale da capire quali pagine vengono referenziate di più.

Campo bit per disabilitazione cache

Il bit viene utilizzato per la disabilitazione della cache. In alcuni casi è utile inibire il meccanismo di cache in alcune pagine. In certe architetture esiste un bit per pagina per cui il sistema operativo istruisce l'hardware a non usare il meccanismo di caching, in quanto può essere controproducente in alcuni contesti, ad esempio, se si utilizza un I/O mappato sugli spazi di indirizzamento. Per accedere ad alcune porte I/O, il sistema operativo può mappare queste porte su alcuni indirizzi di memoria, in questo caso su alcune pagine. Fatta questa operazione di mappatura, può concedere al processo di leggere e scrivere sulle porte I/O. Si autorizza il

processo utente a interagire con il controller del disco. Il meccanismo di caching può rendere non funzionante l'interazione col dispositivo. Se si legge ripetutamente da una porta in attesa che spunti un valore predeterminato per capire se l'operazione è conclusa, l'uso della cache non permette una lettura rapida del valore, poiché si fa riferimento al valore in cache.

Campo bit di validità

Il campo bit di validità viene utilizzato per l'utilizzo dello spazio compreso tra heap e stack. Ogni volta che il processo deve allocare nuove strutture dati, deve invocare il sistema operativo per far sapere che la pagina è stata allocata dal processo. Se il bit di validità o di allocazione è impostato a 0, ogni accesso a questa regione causa un errore fatale (come errore di segmentazione).

14 - Gestione della tabella delle pagine

Progettazione di una tabella di pagine

Anche per semplici istruzioni come somma di addendi risiedenti memoria ci sono almeno tre accessi da fare alla memoria, che devono essere tradotti in quanto espressi in termini di memoria virtuale. Affinché gli indirizzi vengano tradotti, il processo ha bisogno di un accesso alla tabella delle pagine. Il modo in cui questa è memorizzata influenza la velocità di consultazione.

Nei primi sistemi la MMU era equipaggiata con un numero sufficiente di registri per la memorizzazione della tabella (tabelle piccole chiaramente, con al massimo qualche centinaio di campi). Il sistema operativo aveva anche il compito di riprogrammare la MMU per ogni processo. A livello di prestazioni, non è uno scenario negativo: la MMU con i registri implementati al suo interno aveva tutto ciò che era necessario per completare la traduzione, escludendo il caso page fault. Questo tuttavia **non è un approccio scalabile**: all'aumentare della memoria virtuale diventa sempre più oneroso a livello di numero di registri. Ma non è neanche scalabile in termini di lavoro da effettuare nel caso di context switch: più erano i registri da popolare, più era impegnativo il lavoro di riprogrammazione.

Un'approccio alternativo consiste nella memorizzazione delle tabelle di pagine in RAM. Si utilizza un registro che punta alla tabella delle pagine dell'attuale processo. Esiste un registro, che chiameremo **PTBR** (**Page-Table Base Register**), che contiene l'indirizzo fisico della tabella da consultare (puntando quindi al primo record della tabella). Da sottolineare che si tratta di un indirizzo fisico: il registro, e in generale la tabella, viene utilizzata per tradurre l'indirizzo logico in fisico. Questo implica inoltre che la memoria allocata per la tabella **dev'essere contigua**: nella memoria fisica bisogna trovare un pezzo di memoria per ospitare la tabella. Quando bisogna riprogrammare la MMU, basta quindi cambiare un solo registro, indicando dove si trova la tabella dell'attuale processo. Questo quindi semplifica il lavoro e consente scalabilità, permettendo un context switch più veloce. Tuttavia, l'idea di mantenere l'unica copia della tabella in RAM implica che ogni volta che la MMU ha bisogno di consultare la tabella essa dovrà eseguire un fetch alla memoria RAM utilizzando un indirizzo fisico. Quindi ogni accesso in memoria tramite indirizzo logico ha bisogno di accessi alla RAM per la traduzione: il **throughput** degli accessi viene **dimezzato**. In alcuni casi, inoltre, per la traduzione potrebbero essere necessari più accessi alla tabella: l'utilizzo della tabella di pagine in RAM potrebbe non essere vantaggioso. Bisogna anche considerare le dimensioni della tabella di pagine: sistemi a 32 bit richiedono 1 milione di voci.

Tabella dei frame

Esiste un'ulteriore tabella, la **tabella di frame**, gestita dal sistema operativo. La tabella fornisce una mappatura dei frame utilizzati. Ricordiamo che i frame fisici sono la ripartizione della memoria fisica in cui vengono memorizzate le pagine di memoria. La quantità di frame è legata alla memoria fisica utilizzata.
Per ogni frame si memorizza un record contenente:

- informazioni relative all'**occupazione** del frame (stato occupato / libero) Se bisogna allocare una pagina per un processo a causa di un page fault, bisogna trovare un frame libero. La tabella dei frame pertanto permette di trovare il frame, utilizzando solitamente un algoritmo first match.
- informazioni relative a quale processo occupa il frame.

La tabella dei frame è **unica** per l'intero sistema operativo.

Lo spazio fisico si riferisce alla memoria fisica massima gestibile dal sistema. Tuttavia non è detto che il sistema abbia a disposizione l'intera memoria fisica disponibile.

Utilizzo di TLB

All'interno della MMU si utilizza un set di registri noto come **Translation Lookaside Buffer** (**TLB**, o **memoria associativa**). I registri fungono da cache e sono in numero limitato (solitamente tra 64 e 1024) in quanto hanno un impatto sul costo del sistema da implementare, ma sono di vitale importanza per semplificare il lavoro della MMU.

I registri fungono da copie quasi identiche a informazioni che si trovano in alcune voci della tabella delle pagine (similmente alla cache che tiene copie di informazioni presenti in RAM per la CPU). In particolare, vengono memorizzate le voci strettamente utili per un processo. Le voci memorizzate in TLB saranno favorite per il processo di traduzione.
Una voce all'interno di una TLB contiene informazioni prese dalla tabella delle pagine:

- il numero di pagina virtuale a cui si riferisce la voce. Questa informazione non è presente nel contenuto di una voce della tabella delle pagine, ma questa informazione è la **chiave di ricerca** del record della tabella delle pagine: l'informazione infatti non prevede un campo in quanto è la chiave di ricerca che permette di accedere in modo diretto alla voce interessata. Poiché non si ha una tabulazione completa delle voci della tabella delle pagine, e quindi è una **mappatura sparsa**, si ha la necessità di capire qual è il numero del record mappato;
- il bit di validità della voce di TLB, che indica se la voce è usata o meno;
- il codice (o maschera) di protezione. La MMU garantisce che i permessi di lettura / scrittura / esecuzione vengano rispettati. La TLB necessita quest'informazione per garantire che il permesso venga ancora rispettato;
- dirty bit, il flag attivato dalla MMU per tracciare che il contenuto è modificato rispetto alla copia presente in memoria. Notare che se ci sono operazioni di scrittura, la modifica viene

fatta localmente alla TLB. Se la voce in un momento successivo viene ripulita per ospitare un'altra voce, allora il dirty bit dev'essere ricopiato nella tabella delle pagine;
- numero di frame. Per ogni pagina mappata nella TLB esiste l'implementazione della funzione ideale che permette l'associazione pagina-frame.

Ci sono alcune informazioni mancanti per ogni voce. Più precisamente, il bit di referenziamento e il bit di validità sono mancanti. Entrambi mancano in quanto se la pagina è citata nella TLB, è ovvio che ha un contenuto (in quanto è stato già referenziato in precedenza). Al primo TLB miss infatti i successivi accessi trovano giovamento quando l'informazione è copiata nella TLB.

Generalmente l'accesso alla TLB avviene in tempo **costante**. Quando viene generato l'indirizzo virtuale, esso viene presentato alla MMU e si verifica se all'interno della TLB è presente il numero di pagina virtuale dell'indirizzo. Questo controllo avviene verificando **in parallelo** tutte le voci della TLB. Se il numero è presente all'interno della TLB, si ha una **TLB hit**. Il numero di frame permette la traduzione in indirizzo fisico. Inoltre, nel caso di un TLB hit non si ha un page fault.
Nel caso di un **TLB miss**, la MMU esegue una ricerca nella tabella di pagina, rimuovendo una voce dalla TLB e sostituendola con la nuova pagina recuperata.

La gestione delle pagine nella TLB influisce sulla probabilità di un TLB miss rispetto ad un TLB hit, impattando l'efficienza di accesso alla memoria fisica. Un aspetto non trascurabile inoltre è il seguente. Un sistema con questo approccio ha un limite. Ogni processo ha un proprio spazio di indirizzamento: l'indirizzo virtuale è univoco, ma questo non si può dire se si passa da un processo ad un altro. Questo ha diverse implicazioni.
Se si riprogramma la MMU per passare da un processo ad un altro, bisogna modificare il PTBR (e possibilmente anche gli altri). La TLB dev'essere **azzerata** in occorrenza di ogni context switch. La TLB infatti contiene copie ridondanti di informazioni che stanno nella tabella delle pagine; se stiamo cambiando tabella delle pagine, le informazioni non sono più valide. Il fatto che le informazioni non siano più valide deriva dal fatto che queste sono etichettate con chiavi di ricerca che non sono più valide. L'unica etichettatura che si ha è l'informazione sul numero di pagine. Un processo potrebbe avere in una pagina con lo stesso numero contenuti diversi: si rischierebbe di effettuare traduzioni errate. Il problema risiede della univocità del sistema di etichettatura: se si adotta questo sistema, bisogna azzerare la TLB. Ad ogni context switch, quindi, la TLB perderebbe la sua efficienza.

Vorremmo allora riuscire a "mischiare" informazioni nella TLB. Aggiungiamo allora un'informazione relativa a qual è il processo a cui appartiene la pagina. Questo permetterebbe di distinguere la pagina 1 di un processo dalla pagina 1 di un altro processo. La CPU di conseguenza dovrà anche comunicare non solo l'informazione relativa all'indirizzo virtuale e alla natura dell'operazione, ma indicare anche l'appartenenza del processo. Questo prende il nome di **address-space identifier** (**ASID**).
Osserviamo che thread multipli dello stesso processo condividono la stessa tabella delle

pagine. Se la CPU esegue thread di uno stesso processo, le pagine etichettate con lo stesso ASID vanno bene anche nel caso del modello a thread. Per questo si parla di spazio di indirizzamento e non semplicemente di identificatore di processo.

Gestione TLB via software

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

- il **soft miss**, un miss che occorre quando la pagina ricercata risiede in memoria. Risolvere questo tipo di miss solitamente richiede una ventina di istruzioni e può essere risolto in un tempo dell'ordine dei nanosecondi;
- la **hard miss**, un miss che occorre quando la pagina è nel disco. In questo caso bisogna accedere al disco, e il processo di fetching è più lento rispetto al soft miss.

Non ci sono solamente soft miss e hard miss. Quando un programma va in page fault, ci sono tre possibilità:

- **minor page fault**, una situazione di page fault in cui la pagina non si trova nella tabella delle pagine del processo ma si trova in memoria;
- **major page fault**, page fault in cui la pagina dev'essere recuperata dal disco;
- **segmentation fault**, in cui il programma ha avuto accesso ad un indirizzo non valido. Il sistema operativo uccide il programma in questo caso, in quanto è il programma la sorgente dell'errore.

Effective Access Time

Supponiamo che un accesso in RAM costi 100 nanosecondi e un accesso in TLB ne costi 20. Un accesso alla RAM è comunque necessario, in quanto traducendo l'indirizzo logico in fisico è necessario comunque accedere all'indirizzo fisico in RAM. Si ha allora che:

- un TLB hit impiega 120 nanosecondi, in quanto si consulta la TLB, che fornisce tutte le informazioni utili alla traduzione. Dopo i 20 nanosecondi si hanno le informazioni fornite dalla TLB, e in 100 nanosecondi si accede all'indirizzo fisico in RAM;
- un TLB miss impiega 220 nanosecondi. Questo perché si consulta la TLB, che non ci fornisce le informazioni per la traduzione, costando 20 nanosecondi. In 100 nanosecondi si dovrà fare un accesso alla tabella delle pagine, e un ulteriore accesso all'indirizzo fisico calcolato costa altri 100 nanosecondi.

Se supponiamo che un TLB hit ha probabilità pari a 80%, si ha che il tempo medio per un accesso all'indirizzo fisico corretto è:

$$0.8 \cdot 120 + 0.2 \cdot 220 = 140\text{ns}$$

Il discorso può essere generalizzato. Sia α il tempo per un TLB hit e sia β il tempo per un TLB miss. Sia ϵ la probabilità che si verifichi un TLB hit. Si ha che:

$$t_{\text{medio}} = \epsilon \cdot \alpha + (1 - \epsilon) \cdot \beta$$

Tabelle multilivello

Il problema adesso risiede nella memorizzazione delle tabelle delle pagine. Abbiamo detto che per un sistema a 32 bit necessitiamo di 1 milione di voci se la pagina è di 4 kilobyte. Una tabella occuperebbe quindi 4 megabyte supponendo voci da 32 bit. Avendo, ad esempio, 50 processi, 200 megabyte dei 4 gigabyte di memoria massima disponibile non è ottimale come memorizzazione. Inoltre, si ha anche la necessità di trovare un blocco contiguo di memoria, e si ritorna così al problema precedente. Bisogna trovare una soluzione alternativa.

Abbiamo visto che con un indirizzo a 32 bit si hanno 12 bit per offset. Organizziamo allora le pagine in sottoinsiemi per utilizzare al meglio i restanti 20 bit. Invece di avere una tabella di 2^{20} voci, si creano tanti gruppi, ognuno dei quali consta di 2^{10} voci ciascuno. Si organizzano quindi le pagine in gruppi omogenei della stessa grandezza. Per individuare il gruppo, basta dividere per la dimensione del gruppo. Si separano quindi le informazioni dei 20 bit in due campi: $\mathrm{PT1}$, che specifica il gruppo, e $\mathrm{PT2}$, che identifica la pagina all'interno del gruppo (offset del gruppo).

La tabella allora diventa una **tabella multilivello**. La tabella ha 1024 voci (nel nostro esempio). Ognuna di queste voci corrisponde ad un gruppo. La tabella del secondo livello ha la stessa funzione della tabella delle pagine. Unendo tutte le foglie di un ipotetico albero, si ottiene la tabella originale.

La traduzione in questo caso funziona in maniera diversa. Il PTBR punta in questo caso alla tabella del primo livello. Si somma il PTBR al campo $\mathrm{PT1}$. Calcolata la somma, si conosce la posizione della tabella del secondo livello. Per calcolare la voce della tabella del secondo livello, si somma all'indirizzo della posizione della tabella del secondo livello il contenuto di $\mathrm{PT2}$.

In termini di capienza, il problema non viene risolto. Ma questo sarebbe vero se richiedessi che tutte le pagine siano rappresentate con foglie di secondo livello. In un esempio realistico, se si ha una mappa dello spazio di indirizzamento, potrei immaginare che la tabella di pagine di alto livello sia strutturata come la sezione di memoria assegnata ad un processo. I gruppi di pagine non allocate possono essere collocate centralmente, e non richiedono rappresentazione. Non sono pagine che sono parcheggiate sul disco, ma sono proprio spazio vuoto. L'idea quindi è quella di non rappresentare con delle apposite tabelle di secondo livello i gruppi vuoti, ossia gruppi di pagine non allocate. Risulta quindi importante il bit di validità, bit usato per segnalare

al sistema operativo l'allocazione di pagine in memoria. Se si prova ad accedere ad un gruppo non allocato, si ha un problema di accesso in memoria, non un page fault. In questo modo non si ha realmente la necessita di allocare 4 megabyte di spazio per la tabella delle pagine di un processo, ma solo alcuni kilobyte, ossia uno spazio che è **proporzionale all'uso di memoria da parte del processo**.

Lo svantaggio nell'utilizzo di tale approccio risiede nel fatto che è necessario eseguire due accessi: uno per la tabella di primo livello e uno per la tabella del secondo livello. Tuttavia il caso del TLB hit è invariato: la TLB permette lo stesso di evitare l'accesso alla tabella.

Se due livelli non sono sufficienti, è possibile aumentare la profondità dell'albero di tabelle. Aumenta così il numero di accessi, ma si facilita la ricerca di memoria contigua in RAM per l'occupazione delle tabelle. Sistemi a 64 bit solitamente utilizzano 4 livelli di tabelle, e poiché non si raggiunge mai la necessità di utilizzare tutti i 64 bit solitamente si lavora con 48 bit, rendendo più semplice la gestione. Più precisamente, i sistemi Intel a 32 bit utilizzavano una tabella a due livelli. Una **page directory** puntava alle tabelle di pagine contenenti gli indirizzi dei frame di 4 kilobyte. Il Pentium Pro introdusse poi un altro livello, la **page directory pointer table**, contenente solo quattro ingressi e modificando gli altri due livelli in modo tale da contenere ognuno 512 voci (prima si avevano 1024 voci). Successivamente AMD introdusse il supporto a 64 bit per sistemi x86, utilizzando una tabella a 4 livelli. Ogni livello aveva 512 voci, permettendo un indirizzamento di $(2^9)^4 \cdot 2^{12} = 2^{48}$ byte possibili (in totale, 256 terabyte di RAM).

Tabella delle pagine invertita

Un approccio interessante riesce a risolvere lo stesso problema risolto dalle tabelle multilivello. L'idea della **tabella delle pagine invertita** risiede nell'arricchimento della tabella dei frame, ossia la tabella dei frame fisici monitorata dal sistema operativo. Più precisamente, si arricchisce la tabella dei frame in modo tale da indicare all'interno tutto ciò che riguarda la pagina ospitata nel frame fisico. La dimensione della tabella dei frame è chiaramente proporzionale alla dimensione della memoria fisica. Se un frame è occupato, vogliamo indicare non solo qual è il processo occupante, ma anche la pagina virtuale che la occupa. L'idea è molto semplice: si utilizza la tabella dei frame per effettuare le traduzioni.

L'indirizzo virtuale calcolato dalla CPU è diviso sempre da numero di pagina e da offset. Si utilizza il numero di pagina assieme al PID per ricercare la voce nella tabella delle pagine invertita. Trovata la voce, si sa qual è il frame associato, ossia il numero della voce corrispondente.

Chiaramente ci sono alcuni svantaggi. Innanzitutto la ricerca all'interno della tabella delle pagine invertita è una ricerca lineare, pertanto è poco efficiente. Si potrebbe utilizzare una tabella hash indicizzata sull'indirizzo virtuale. Più precisamente, si utilizza una tabella hash in cui le collisioni sono risolte per concatenazione. Il numero di frame sarà in questo caso

memorizzato nel record. Se scorrendo la tabella non si trova la pagina, si ha un page fault.
Il beneficio nell'utilizzo di una tabella hash tuttavia risiede nel fatto che si ha una sola struttura dati che sostituisce tutte le tabelle delle pagine. Sistemi reali inoltre non utilizzano solamente la tabella hash ma anche la TLB come cache per evitare di accedere alla tabella invertita.

Ci sono alcune informazioni che non vengono memorizzate nella tabella. Una tra queste potrebbe essere la maschera dei permessi. Alcune pagine inoltre possono essere nel disco, e non ci basta portare il contenuto del disco in RAM, ma bisogna inserire il record nella tabella delle pagine invertita. Alcune delle informazioni con cui dobbiamo popolare il record non sono deducibili (bit di modifica e di referenziamento sono deducibili e sono entrambi posti a 0 quando la pagina è caricata in RAM). La maschera dei permessi non è deducibile, e il sistema operativo ha ancora bisogno delle tabelle classiche per capire dove si trova la pagina nel disco. Ma questo è solo necessario solo quando si verifica un page fault. Non è corretto quindi dire che le tabelle classiche non servono, ma il loro uso è molto limitato e tutto sommato possono anche essere memorizzate sul disco.

Cache della memoria e memoria virtuale

La MMU prende indirizzi virtuali e li traduce in fisici. La immaginiamo quindi a metà tra RAM e CPU. Ma se lavoriamo sulle cache, dobbiamo capire se la cache va posta prima o dopo la MMU.

Cache dopo la MMU

Se la cache è posta dopo la MMU, la cache lavora su indirizzi fisici: la CPU infatti genera indirizzi logici tradotti dalla MMU e passati alla cache. La cache cerca di evitare l'accesso alla memoria RAM. In questo caso non abbiamo ambiguità nel caso di context switch: gli indirizzi infatti sono fisici, per cui la cache non dev'essere azzerata. Questo ha implicazioni positive: le linee di cache di un processo possono essere memorizzate sulla cache anche quando si cambia processo in esecuzione. Il caching tuttavia è poco efficace per il fatto che la MMU deve sempre tradurre ogni indirizzo generato dalla CPU.

Cache prima della MMU

Se la cache è posta prima della MMU, la cache lavora su indirizzi logici. In questo caso servono gli ASID per non azzerare la cache ogni volta quando la CPU produce indirizzi per lo stesso processo. Tuttavia, si garantisce maggiore efficenza, in quanto la cache lavora direttamente con la CPU. La scalabilità tuttavia soffre al crescere della dimensione di cache.

Ibrido

Si utilizza in realtà un approccio ibrido. In particolare, le cache L1, più piccolo e a stretto contatto con la CPU, utilizzano indirizzi virtuali. Le cache L2 e successive, più economice e

poste dopo la MMU, lavorano con indirizzi fisici.

15 - Algoritmi di sostituzione delle pagine

Algoritmi di sostituzione delle pagine

Quando l'indirizzo generato dalla CPU, che viene tradotto dalla MMU, non si trova in RAM, si verifica un **page fault**. Il controllo viene passato al sistema operativo, commutando la CPU dalla modalità utente alla modalità kernel.
La gestione del page fault include l'individuazione del contenuto da caricare dall'area di swap in memoria RAM. Affinchè ciò possa avvenire, è necessario individuare un frame libero che ospita tale pagina. Il processo che ha causato page fault è fondamentalmente bloccato: il processo infatti viene bloccato sull'istruzione che ha causato l'eccezione. L'istruzione viene rieseguita quando la pagina viene caricata in memoria.

Il caso più semplice corrisponde alla situazione in cui in RAM è presente un frame libero. La pagina viene allora caricata in RAM. Il caso più problematico invece è quello in cui tutti i frame sono occupati; tale evento è tuttavia ordinario e frequente. Bisogna allora sviluppare una soluzione che riduce al minimo l'overhead. L'overhead in questo caso è misurato in virtù del numero di page fault che si verificano: si cerca allora di ridurre al minimo il numero di page fault che accadranno in futuro.
In questo caso la scelta del frame da liberare è determinata in virtù di un **algoritmo di sostituzione di pagina**. Il problema è inoltre abbastanza vicino al problema della gestione delle cache. Le cache gestiscono infatti una situazione simile: la cache è di dimensione medio-piccola e mantiene un numero finito di informazioni presenti in RAM. Di fronte ad un cache miss, la linea che ha causato il page miss dev'essere portata in cache e bisogna eliminare qualche contenuto dalla cache. L'idea che sta dietro la scelta mira a creare un disservizio minimo se non trascurabile. Ci sono tuttavia differenze con l'algoritmo di sostituzione di pagina, ad esempio la cache gestisce le linee via hardware, mentre le pagine vanno gestite via software.

Soluzione ottimale

L'algoritmo ideale deve scegliere la pagina che verrà referenziata in un futuro più lontano. Si possono etichettare in questo caso le pagine con un numero, che può essere il numero di istruzioni che la CPU eseguirà prima di fare riferimento a qualsiasi informazione di quella pagina. Avere un'etichettatura di questo genere permette di selezionare la pagina tale che il numero è il più grande. Il numero di page fault totali viene minimizzato, in quanto viene rimandato ad un futuro sempre più lontano l'evento di page fault.
Chiaramente non è possibile ottenere tale metrica facilmente senza fare delle simulazioni. La soluzione ottimale tuttavia rappresenta un ottimo termine di paragone per lo sviluppo di altri algoritmi.

Algoritmo NRU: Not Recently Used

L'algoritmo **Not Recently Used** (**NRU**) si basa su alcune informazioni delle voci della tabella delle pagine. In particolare, si considerano il bit di referenziamento e il bit di modifica. Ricordiamo che il bit di modifica indica se la pagina in questione è potenzialmente differente rispetto alla copia su disco. La MMU pone il bit di modifica a 1 quando si ha un'operazione di store qualsiasi. Il bit di referenziamento invece indica se è stata eseguita una lettura o scrittura sulla pagina. I bit vengono direttamente gestiti dalla MMU.
In alcune occasioni il sistema operativo azzera il bit di referenziamento. Una prima strategia sarebbe quella di azzerare tutti i bit di referenziamento periodicamente, stabilendo, ad esempio, ogni mezzo secondo l'azzeramento dei bit. Questo trasforma il meccanismo in una **statistica a breve termine**. Questo perché l'aggiornamento delle voci viene eseguito dall'hardware. Tenere aggiornato un flag via software infatti sarebbe oneroso per il sistema operativo, in quanto si ha la necessità di accedere alla RAM per ogni istruzione di aggiornamento. Il fatto che sia l'hardware se ne occupi in termini di attivazione è un grosso vantaggio. Questa statistica è a breve termine in quanto prima di azzerare i bit lo stato attuale dei vettori di bit (uno per pagina) ha un significato ben preciso sull'uso che i processi hanno fatto delle pagine prima dell'ultimo azzeramento. Nel tempo è quindi una statistica limitata; è anche limitata dal fatto che non si monitora il numero di accessi alle pagine, ma solo se esso è avvenuto. Nel suo essere a costo zero tuttavia può essere una buona soluzione.

L'algoritmo NRU utilizza entrambi i bit citati. Una volta chiamato dal sistema operativo, l'algoritmo analizza i record della tabella delle pagine di tutti i processi e osserva la combinazione di bit referenziamento e modifica relativa alle pagine, individuando quattro classi:

- **classe 0**: non referenziato, non modificato;
- **classe 1**: non referenziato, modificato;
- **classe 2**: referenziato, non modificato;
- **classe 3**: referenziato, modificato.

Individuata la classificazione, si può individuare la classe non vuota più in alto. Ad esempio, se esiste almeno una pagina in classe 0, allora si elimina una pagina di questa classe; altrimenti, si passa alla classe successiva. Si individua allora una classe più alta non vuota.
L'ordine è ben ponderato in questo caso:

- la classe 0 è individuata dai bit 0, 0. Prima dell'ultimo azzeramento, tutte le pagine di questa classe non sono state attenzionate. Questa è un'indicazione precisa: ci piacerebbe scegliere infati un bit di referenziamento a zero, che fornisce un'indicazione sull'utilizzo della pagina da parte del processo. Alla pagina dobbiamo preferire una pagina che è usata;
- la classe 1 è individuata dai bit 0, 1. Chiaramente tra due pagine non referenziate, preferiamo una pagina non modificata, in modo tale da evitare l'overhead della copia della

pagina in RAM quando la sua voce viene eliminata;

- la classe 2 è individuata dai bit 1, 0. Tutte le pagine in questo caso sono referenziate (sintomo di un lasso di tempo troppo lungo). Anche in questo caso preferiamo l'eliminazione di una pagina pulita piuttosto di una pagina sporca;
- la classe 3 è individuata dai bit 1, 1. Le pagine di questa classe non sono quelle preferite per l'eliminazione.

L'algoritmo è molto semplice da implementare. Tuttavia ci poniamo una domanda: una volta scelta la classe, qual è la pagina da scegliere? Anche in questo caso dobbiamo eseguire una scelta. Una prima idea è tramite **scelta FIFO**, considerando l'**età della pagina**, ossia quando la pagina è stata portata in RAM. Si sceglie la pagina più vecchia, ossia la pagina che è stata più tempo in RAM. Questo potrebbe non essere una scelta universale e non è correlato con l'utilizzo della pagina, ma è un principio di scelta accettabile. Inoltre è un meccanismo semplice da gestire. Il tracciamento dell'età del processo può essere fatto via software: gli eventi che influiscono questo ordine sono eventi poco frequenti. Si possono etichettare le pagine con un timestamp. Nella sua semplicità è stato utilizzato in alcune varianti di sistemi operativi esistenti.

Algoritmo FIFO

La scelta FIFO può essere utilizzata come criterio principale, mantenendo le pagine in ordine in virtù dell'ingresso in RAM, tenendo in testa le pagine più vecchie (caricate per più tempo in RAM). Questa tecnica non tiene conto dell'utilizzo della pagina da parte del processo. Inoltre, non è detto che se la pagina è più vecchia allora non viene utilizzata. La pagina infatti potrebbe essere di molto interesse per un processo. Ad esempio, la tabella dei simboli di un compilatore è pesantemente referenziata dal processo del compilatore. La tabella è utile in molte fasi, e questo significa che la pagina è riferita molto, sebbene sia vecchia.

Algoritmo della seconda chance

Sull'ordinamento FIFO viene sviluppato l'algoritmo della **seconda chance**. Sulla testa è presente la pagina più vecchia, e l'algoritmo mira come sua scelta a questa pagina. Tuttavia, **può risparmiare la pagina**, trattandola come **una pagina appena caricata in memoria** (nella coda, è come se ponessimo la pagina in coda). La possibilità di risparmiare la pagina dipende dallo stato del bit di referenziamento.
Se la pagina viene risparmiata, si esegue la medesima operazione con la successiva pagina, fino a quando non si trova la pagina con bit di referenziamento posto a zero. Da notare come questa scelta **esista concretamente al 100%**: quando una pagina infatti viene risparmiata, il bit di referenziamento viene azzerato. L'azzeramento periodico del bit può essere quindi evitato. Il caso limite è che tutte le pagine hanno bit di referenziamento pari a 1: tutte le pagine vedranno il bit azzerato, quindi la scelta ricadrà sulla pagina che inizialmente era in testa.

Algoritmo clock

Un'implementazione efficiente dell'algoritmo di seconda chance è **l'algoritmo clock**. In virtù della struttura dati che si utilizza, infatti, risparmiare una pagina ha un suo costo, in quanto si ha un'estrazione e un successivo inserimento. Una variante dell'algoritmo di seconda chance, ossia l'algoritmo clock, implementa la coda come una lista singolarmente linkata ma circolare. Viene mantenuta una struttura con un puntatore `next` che è chiusa, ossia l'ultimo elemento punta al primo. Si fa scorrere allora il puntatore all'elemento successivo se l'elemento attuale ha bit di referenziamento posto a 1; in questo caso, si imposta il bit a 0 e si scorre al successivo elemento. Si riduce quindi il numero di operazioni da effettuare.

Algoritmo LRU: Least Recently Used

Tra tutti gli algoritmi noti che hanno un'idea riconducibile all'idea di algoritmo ottimale, l'**algoritmo Least Recently Used** (**LRU**) è il migliore. Si scarta la pagina meno usata di recente. Si traccia temporalmente ogni pagina in virtù dell'ultimo utilizzo. Si sceglie allora la pagina più lontana temporalmente, che rappresenta infatti la pagina meno recentemente utilizzata. Questo si avvicina molto all'idea dell'algoritmo ottimale, con l'unica differenza che LRU guarda al passato. Si osserva tuttavia una certa simmetria: si osserva infatti che informazioni che si utilizzano nel recente passato, probabilmente si utilizzano nel recente futuro. Questo è simile al **principio di località temporale**.

Il problema ora è il tracciamento temporale. Via hardware questo è possibile: si utilizza un contatore che aumenta asintoticamente all'interno della CPU (esistono i contatori che contano i cicli della CPU). All'interno della pagina a cui si fa accesso si copia il valore del contatore in quel momento. Si ha allora una sorta di timestamp in virtù dell'ultimo accesso. Fare qualcosa del genere via software tuttavia è impossibile. Se l'aggiornamento viene fatto via hardware, l'implementazione della LRU è molto semplice.
Non esiste tuttavia nessun hardware che può fare un lavoro del genere. Possiamo provare a vedere due possibili approssimazioni dell'algoritmo LRU per avere una soluzione gestita dal sistema operativo.

Algoritmo NFU: Not Frequently Used

Una prima approssimazione è l'**algoritmo Not Frequently Used** (**NFU**). Periodicamente il bit di referenziamento viene azzerato dal sistema operativo, ma quest'ultimo viene aggregato in un contatore per pagina. Si aggiorna il contatore tramite il bit di referenziamento. Non si fa ad ogni referenziamento della pagina, ma ad ogni intervallo temporale prestabilito. A questo punto la scelta risiede nella pagina con il contatore più basso.
Chiaramente il sistema operativo non conta gli utilizzi, ma il numero di cicli in cui la pagina è stata utilizzata. Inoltre il contatore non dà chiare informazioni su quante volte esattamente la pagina sia stata utilizzata. Una pagina utilizzata frequentemente in passato potrebbe essere privilegiata sebbene sia scarsamente utilizzata di recente.

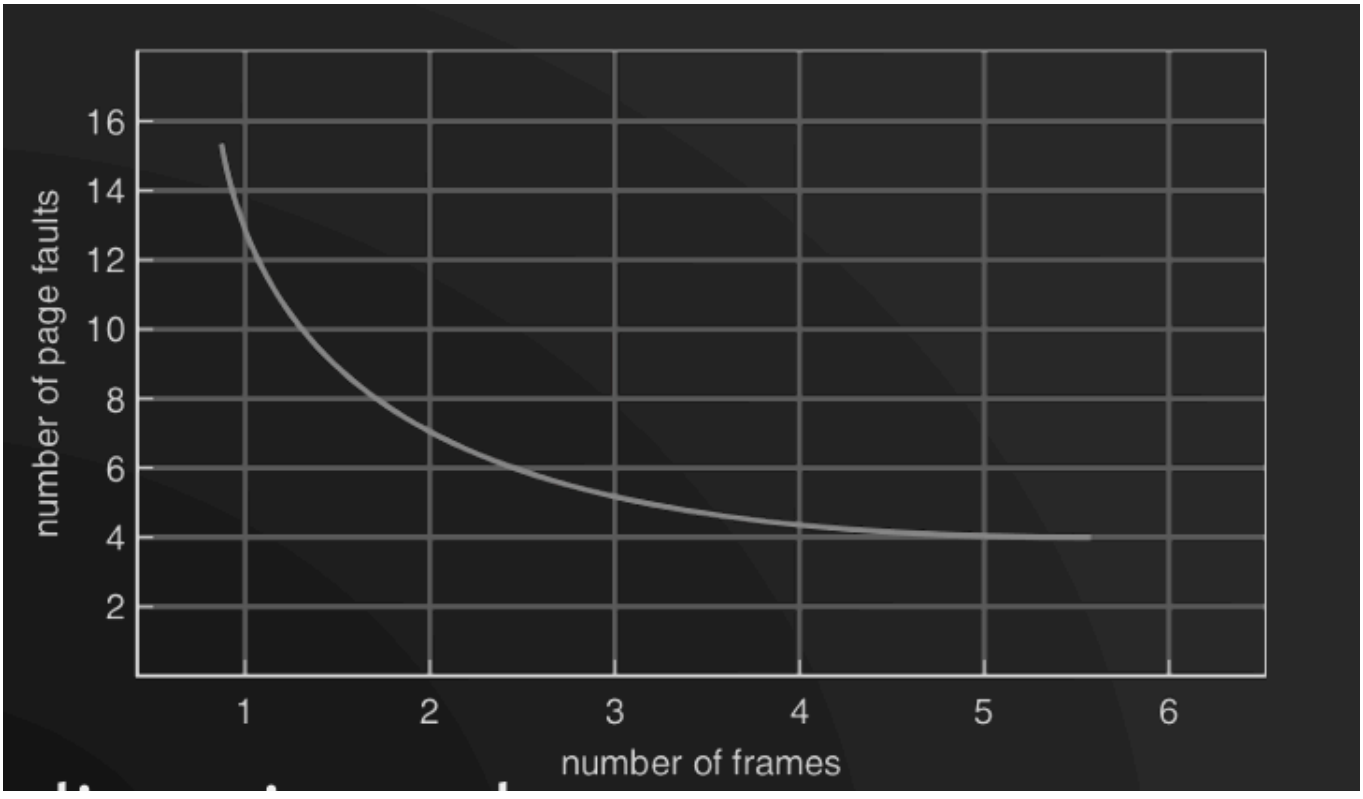
Algoritmo di aging

Un tentativo migliore di approssimare l'algoritmo LRU è l'**algoritmo di aging**. L'aggiornamento dei contatori prevede uno shift a destra del contatore e si pone il bit di referenziamento come bit più significativo del contatore. La scelta del bit più significativo ha un impatto importante: il contatore slitta verso destra butta via il bit di referenziamento in relazione al ciclo più vecchio tracciato.
Si sceglie il contatore più piccolo. Il bit più significativo chiaramente dà un contributo maggiore al contatore, seguendo lo stesso principio dell'algoritmo LRU: infatti anche a parità dei bit meno significativi, la scelta del contatore più piccolo ci permette di dare peso alla pagina usata anche in momenti passati, eliminando una pagina usata meno in passato.

Ci sono tuttavia alcuni aspetti da considerare riguaradnte la **taglia del contatore**. La taglia del contatore esprime la "profondità temporale". Se il contatore è di n bit, non si può guardare oltre gli n bit più recenti. Il discorso quindi è che non è detto che la LRU potrebbe non eseguire la stessa scelta dell'algoritmo di aging. Il numero di bit del contatore funge quindi da "barriera temporale".

Confronto delle prestazioni

Utilizziamo come metrica di confronto il numero di page fault. Possiamo inoltre studiare il numero di page fault in virtù del numero di frame disponibili: all'aumentare dei frame si ha maggiore RAM per il sistema. Allora si presenta una situazione simile:

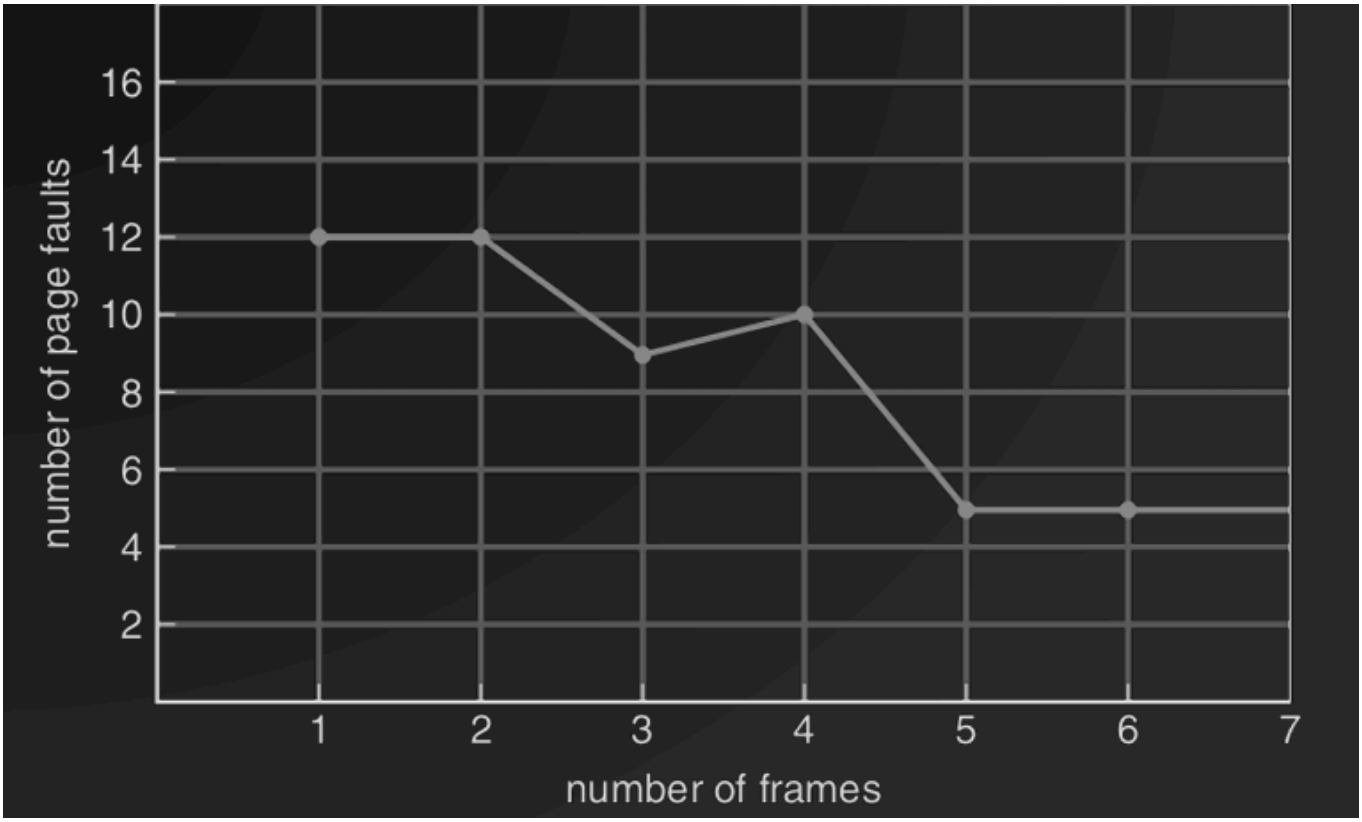


A parità di condizioni, fissiamo:

- 3 frame da amministrare;
- sequenza di indirizzi virtuali a cui accedere, da cui possiamo estrapolare la sequenza di numeri di pagina a cui accedere. La sequenza può essere compattata rimuovendo le ripetizioni contigue (ad esempio, `23332` diventa `232`).

Consideriamo allora la sequenza: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1. Partendo dai tre frame vuoti, si caricheranno le pagine 7,0,1 a seguire di tre page fault. La richiesta della pagina 2 causa un ulteriore page fault. Per ora utilizziamo l'**algoritmo ottimale**: andremo a vedere gli usi delle pagine nel futuro (praticamente è impossibile... è solo una simulazione teorica). Notiamo che la pagina 7 sarà quella scartata. Si hanno quindi le pagine 2,0,1. La richiesta della pagina 0 non causa page fault, ma la tre sì, portando alla sostituzione della pagina 1 con la richiesta: si ha quindi 2,0,3. Compllletata la sequenza, si hanno in totale 9 page fault. Questo 9 è il **numero minimo di page fault**.

A conferma di ciò confrontiamo il comportamento con l'algoritmo FIFO. A parità di sequenza, dopo 7,0,1 si ha un page fault per 2: si scarta 7 e si ottiene 2,0,1 come configurazione delle pagine caricate. Successivamente 3 viene richiesta, e si ha 2,3,1, e la richiesta di 0 causa un page fault, portando a 2,3,0. Continuando così si ottengono 15 page fault.
Tuttavia con alcune sequenze accade qualcosa di strano (ad esempio, 1,2,3,4,1,2,5,1,2,3,4,5). Generalmente ci aspettiamo, come mostra il grafico illustrato precedentemente, che aumentando la quantità di RAM si abbia una miglioramento del numero di page fault. L'algoritmo FIFO, di fronte ad una situazione reale, può presentare il seguente comportamento anomalo:



Aumentando la RAM, aumentano i page fault, cambiando il numero di frame da 3 a 4,

aumentano i page fault, che è controintuitivo. Questa anomalia prende il nome di **anomalia di Belady**.

Confrontiamo adesso le performance analizzando come opera l'algoritmo LRU con la medesima sequenza che abbiamo visto per l'algoritmo ottimale e l'algoritmo FIFO. Si ottengono in totale 12 page fault. L'algoritmo LRU non soffre dell'anomalia di Belady in quanto gode della **proprietà di inclusione**: l'insieme di pagine caricate avendo n frame è incluso in quello che si avrebbe avendo $n+1$ frame. Sia quindi t un istante temporale, e sia $B_t(n)$ l'insieme di n pagine in memoria all'istante t. In ogni istante allora vale:

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

L'algoritmo quindi è stabile, nel senso che avendo un frame in più si hanno i vecchi frame più un altro frame. Questo garantisce che con l'aumento dei frame si possono solo avere meno page fault. La LRU offre intuitivamente questa proprietà in quanto colloca nella linea temporale del passato le pagine da scegliere. Sceglie di volta in volta il gruppo di n pagine più recenti se sono a disposizione n frame. Quindi avendo $n+1$ frame, si ha sempre lo stesso insieme di n pagine, ma si aggiunge un'ulteriore pagina che precede questo insieme nella linea temporale.

L'algoritmo NFU e di aging godono pure della proprietà di inclusione dovuto al fatto che approssimano il comportamento dell'algoritmo LRU. L'algoritmo di seconda chance e di clock soffrono dell'anomalia di Belady in quanto in alcuni casi si comportano come l'algoritmo FIFO. Infine, anche l'algoritmo NRU soffre dell'anomalia di Belady in quanto dopo la selezione di una classe si utilizza un principio FIFO; poiché ci si riduce al caso FIFO, si può ripresentare lo stesso problema.

Implementazione in sistemi operativi

I sistemi operativi spesso implementano l'algoritmo di seconda chance con una scelta arricchita. Non si considera solo il bit di referenziamento ma anche il bit di modifica.

16 - Allocazione dei frame

Allocazione dei frame

Ci sono una serie di considerazioni di cui tener conto. Una di queste è l'**allocazione dei frame**: come possiamo distribuire la RAM in virtù dei processi?
La quantità di frame da assegnare ad un processo ha già dei range di funzionamento. Il numero massimo di frame che si potrebbero assegnare ad un processo è chiaramente pari al numero massimo di frame disponibili per la RAM. Esiste anche un minimo teorico per ogni processo. Un frame per processo è troppo poco come minimo. Basta ragionare sul numero di riferimenti che l'esecuzione di un'istruzione può generare.
Esiste un minimo strutturale sotto il quale non si può andare. Questo minimo è legato all'**architettura del modo in cui vengono eseguite le istruzioni**. Supponiamo di avere un'istruzione semplice, come `ADD`, in cui un operando è un registro e un operando è in memoria. Avere un solo frame per un processo che esegue l'istruzione porta all'impossibilità di eseguire l'istruzione. Se infatti nella fase di fetch ho la pagina in RAM, nell'unica pagina in memoria, l'istruzione non causa page fault. Se la pagina non è in RAM, si ha un page fault. Tuttavia, l'**operando con molta probabilità ricadrà su una pagina diversa**, in quanto il codice e i dati sono in aree di memoria diversa, e le aree in memoria contengono dati omogenei (ossia una regione contiene solo codice, un'altra contiene solo dati). Quindi è possibile che ci sia una pagina per il codice e una pagina per la memoria. Quindi:

- un primo page fault porta al caricamento della pagina contenente il dato, che rimpiazza la pagina contenente il codice;
- un ulteriore page fault porta alla situazione diametralmente opposta rispetto alla precedente.

Un solo frame per processo quindi porterebbe ad un ciclo infinito di page fault: non si ha un numero minimo di frame utili per eseguire una singola istruzione. Potremmo allora dire che due frame bastano. Ma in realtà dipende dall'istruzione da eseguire. In generale il numero minimo potrebbe essere 3, ad esempio se si pensa ad un'istruzione con 2 operandi. Ma in realtà se un'operando richiede un riferimento in memoria di tipo **indiretto** (nota: un accesso in memoria indiretto funziona nel seguente modo. Un'istruzione dice di accedere alla memoria dove si trova un indirizzo; in quell'indirizzo si trova l'indirizzo per trovare l'operando in memoria), un singolo operando potrebbe richiedere accessi a due pagine distinte. Ad esempio, `ADD L1, L2` potrebbe avere `L1, L2` riferimenti indiretti. Il minimo strutturale diventa 5. I livelli di accesso inoltre potrebbero essere di un livello superiore al primo, portando non ad un solo reindirizzamento, ma anche due, tre e così via. Per alcune istruzioni inoltre la fase di fetch può richiedere più pagine, aumentando di un'unità il minimo strutturale.

Il minimo strutturale, in definitiva, dipende dall'architettura. Questo minimo strutturale funge da soglia minima sotto il quale non ha senso eseguire una qualunque istruzione, in quanto quest'ultima causa un blocco su una singola istruzione. Non è il caso tuttavia di accesso che causa page fault: qui si ha un problema relativo al fatto che l'esecuzione si blocca su una data istruzione **e non procede**. Non si ha quindi la garanzia che tutte le istruzioni vengano eseguite. Nella pratica, se l'assegnazione soddisfa il minimo strutturale ma si supera il minimo strutturale di poco, si ha un'esecuzione del processo con prestazioni pessime: il numero di page fault è alto e l'overhead è eccessivo.

Paginazione su richiesta (pure demand paging)

Se di un processo si possono tenere solo alcune pagine, quali pagine dovrebbero essere memorizzate in memoria durante la fase iniziale? Se il sistema operativo dedica solo alcune pagine a un processo, quali sono le pagine scelte da dover portare dal disco alla RAM? Risposte a queste domande si trovano nella **paginazione su richiesta**. Il sistema operativo **non carica alcuna pagina all'avvio del processo**. Nel momento in cui il processo si avvia concretamente, nei suoi primi istanti di vita la prima istruzione crea uno o più page fault. Ci sarà un'inpennata di page fault legati alla necessità di mantenere il processo in esecuzione. Si decide quindi di non caricare nulla in RAM, e si lascia al processo stabilire quali pagine caricare. Il meccanismo di page fault permette quindi di capire quali pagine devono essere caricate in memoria, a costo di un avvio lento del processo (in quanto la quantità di page fault causa molti overhead).

In alcuni sistemi pratici (ad esempio su Windows è implementato a livello kernel) è possibile ottimizzare l'avvio di un'applicazione osservando quali sono le pagine utili nella fase di avvio del programma facendo riferimento alle vecchie esecuzioni: le pagine più frequentemente utilizzate sono quelle caricate.

Strategie di suddivisione dei frame

Una prima strategia per l'allocazione dei frame è la strategia di **allocazione equa**. Si fornisce un numero uguale di frame per ogni processo. Avendo quindi n frame ed m processi, ogni processo ha un numero di frame pari a $\frac{n}{m}$. Se il grado di multiprogrammazione è alto, ogni processo riceve meno pagine.
Non è detto che l'equità fornisca benefici concreti ad ogni processo. Sebbene tutti i processi abbiano lo stesso numero di frame, i processi possono essere di dimensioni differenti (a livello di informazioni che risiedono in spazio di indirizzamento, ossia codici e dati): in questo caso l'assegnazione equa non favorisce i processi di dimensione maggiore. Processi più piccoli potrebbero avere anche più frame di quelli necessari, ma la stessa qualità di servizio non viene garantita a processi con maggiori esigenze.

Si potrebbe assegnare un numero di frame che è proporzionale alla dimensione del processo. Sia s_i la dimensione di un processo i (la dimensione per ora è assunta) e sia S la dimensione totale di tutti i processi. Vorremmo allora assegnare un numero di frame a_i al processo tale che:

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

L'approccio di **allocazione proporzionale** fornisce un numero maggiore di frame a processi di dimensioni maggiori. Tuttavia, questo non è sufficiente: processi con maggiore priorità non sono obbligatoriamente di maggiore dimensione. In generale, vogliamo fornire un miglior servizio a processi con più alta priorità. Quest'ultimo tipo di approccio prende il nome di **allocazione per priorità**.

In base a come opera l'algoritmo di sostituzione, possiamo anche distinguere due tipi di allocazione differenti:

- **allocazione locale**, ossia il set di pagine da cui scegliere la vittima è costituito dai frame appartenenti ad un processo x durante la cui esecuzione si è verificato un page fault. L'approccio è funzionale in virtù del numero di frame associati ad un processo, in quanto se si parte con 40 pagine, sostituendo una pagina si rimane comunque a 40 pagine per il processo;
- **allocazione globale**, ossia il set di pagine da cui scegliere la vittima non è limitato alle pagine del processo. In questo caso la scelta globale per la sostituzione della pagina varia il numero di frame disponibili per un processo: un processo può vedere il suo numero di frame aumentare dopo la sostituzione di una pagina di un altro processo. L'utilizzo di un approccio globale, tuttavia, consente in maniera dinamica di bilanciare il numero di frame necessari per il processo. Un algoritmo di sostituzione delle pagine efficiente assegnerebbe col tempo una quantità di frame maggiore ad un processo che lavora con molti dati rispetto ad un processo che lavora con pochi dati e quindi necessita pochi frame. Questo è alla base di un modello di allocazione dei frame noto come **modello di località**.

Insufficienza di frame per processo

Se ad un processo sono assegnati pochi frame, abbiamo diversi scenari:

- **il numero di frame è sotto il minimo strutturale**. In questo caso il processo viene sospeso e si fa swapping su disco;
- **il numero di frame è leggermente superiore rispetto al minimo strutturale**. Il processo va in **thrashing**.

Thrashing

Il **thrashing** è un picco di eventi di page-fault da parte di un processo. Se il sistema operativo riesce ad accorgersi del problema, eventualmente assegnando più frame al processo, la situazione è in grado di stabilizzarsi. Il sistema quindi dev'essere in grado di adattarsi al thrashing del processo. Tipicamente adottando una strategia di allocazione globale se si riesce a trovare dei frame per il processo il numero di page-fault si normalizza, stabilizzando la situazione. Tuttavia, se l'allocazione è locale, il processo va in thrashing e il processo generalmente lavora male.
In situazioni in cui tuttavia il thrashing riguarda tutti i processi, ossia si ha una mancanza di frame, il sistema è in **sovraccarico**. Il sistema operativo deve quindi swappare alcuni processi o direttamente ucciderli per stabilizzare il sistema.

Modello di località

Il concetto di località è un concetto legato al modo in cui la CPU esegue le istruzioni di un processo. Possiamo definire una località come un insieme di locazioni di memoria, che riguardano codice e dati. La località attuale di un processo in un determinato stato del processo è l'insieme di locazioni utili al processo in quell'istante. La località attuale muta nel tempo, e il processo può avere una certa località in un istante x e avere una località molto diversa anche dopo una piccola frazione di tempo. Osserviamo che anche la dimensione della località non è prefissata. La dimensione è legata infatti a cosa fa il processo in un determinato istante: la località attuale può avere una certa taglia, ma poco dopo può anche cambiare di dimensione. Se la località non riesce a essere memorizzata nei frame, è logico pensare che il processo abbia un numero maggiore di page fault.

Se riuscissimo a tenere traccia della località (qual è e quant'è grande) avremmo uno strumento chiave per l'assegnazione esatta delle risorse.

Working set

Il **working set** è un concetto simile al concetto di località, ma lavora per pagine. Il working set può essere definito a partire da un parametro Δ, parametro che misura le **istruzioni**. Il working set allora è l'insieme delle pagine referenziate dalle ultime Δ istruzioni. Data allora la linea temporale e date le ultime Δ pagine referenziate dal codice, se stabiliamo un istante $t+1$ basta allora considerare le precedenti Δ pagine referenziate a partire dall'istante t (incluso).

Il working set può cambiare di dimensione in virtù di quali sono le pagine e quante volte sono state referenziate. Ad esempio, con $\Delta = 5$ posso avere la sequenza 1,2,3,4,5 e ottenere un working set di dimensione 5, oppure la sequenza 3,4,3,3,4 e ottenere un working set di dimensione 2. Questo si avvicina molto al concetto di località definito precedentemente.

Conoscendo il working set, è possibile assegnare al processo un numero di frame che è proporzionale alla dimensione del working set. Possiamo quindi calcolare una **richiesta**

globale di frame, ossia capire qual è l'esigenza del sistema: basta sommare le dimensioni di tutti i working set e confrontarle con la memoria disponibile. Se questo numero è maggiore del numero di frame disponibili, si ha una probabile situazione di **thrashing**. In tal caso è possibile sospendere alcuni processi e in generale svolgere un lavoro attivo per prevenire una situazione di sovraccarico del sistema.

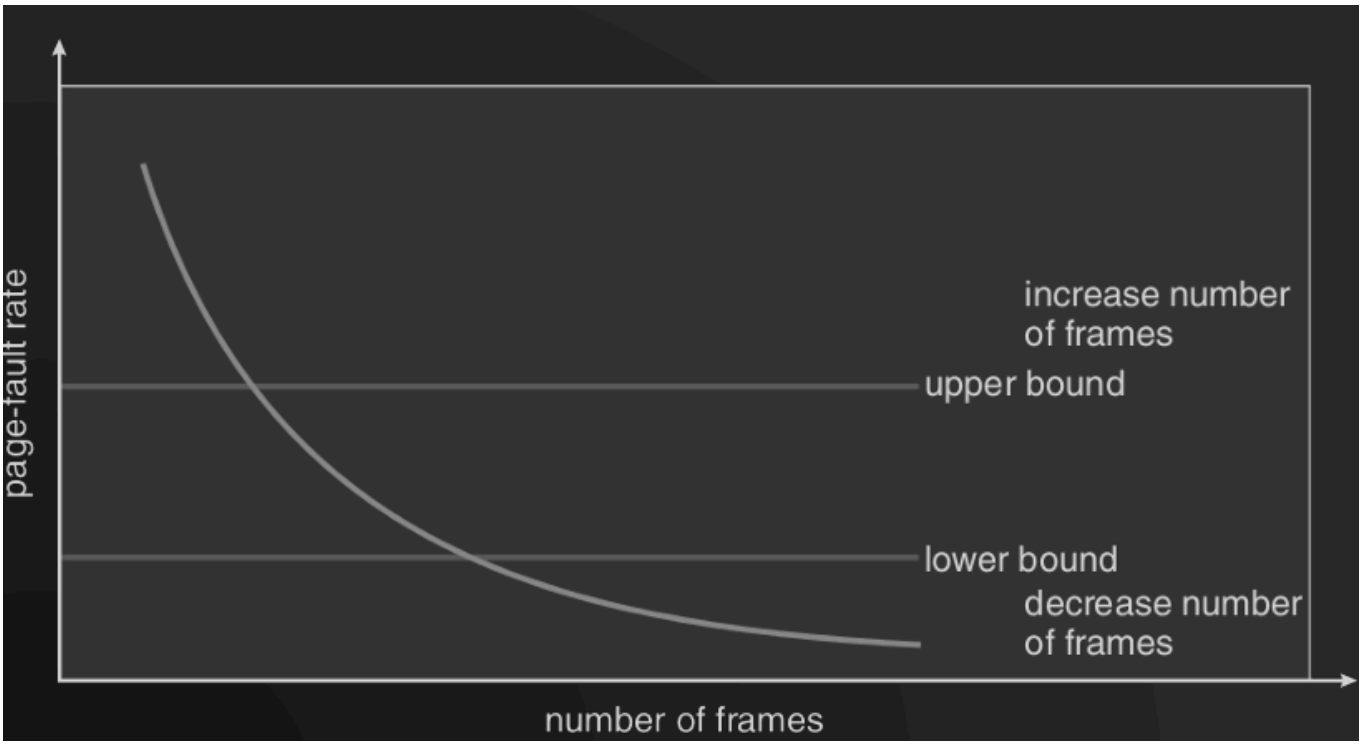
Calcolo del working set

Possiamo approssimare il working set utilizzando degli **interrupt periodici**.
Supponiamo di adottare un meccanismo di aging per la sostituzione delle pagine, e quindi di avere un contatore per una pagina dove ogni bit rappresenta un ciclo di referenziamento alla pagina. In base al valore Δ, possiamo guardare i Δ bit più significativi del contatore e vedere se è presente almeno un bit nel contatore. In tal caso, significa che almeno una volta la pagina è stata referenziata negli ultimi Δ cicli. Questo ragionamento può essere applicato ad ogni frame, e mettendo assieme i risultati possiamo capire quali sono le pagine toccate negli ultimi trenta cicli, ottenendo un tracciamento efficiente del working set del processo.

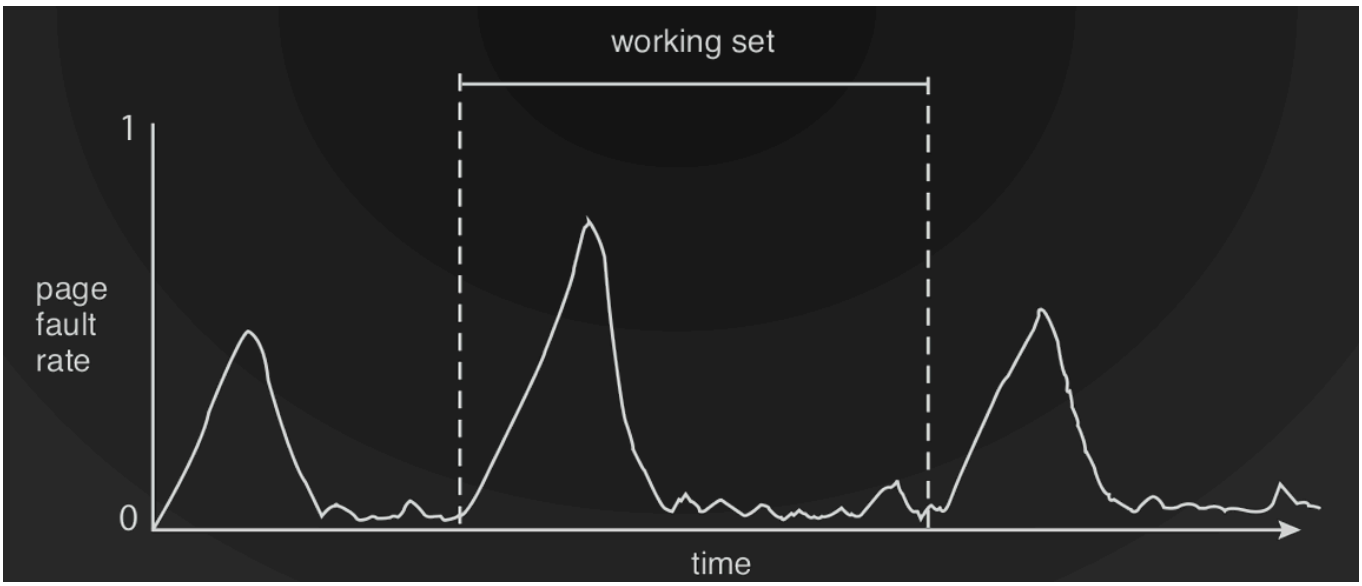
Page fault frequency

Possiamo monitorare il thrashing dei processi non basandoci sui working set, ma semplicemente in base alla **frequenza dei page fault**. Per ogni processo si considera qual è la frequenza dei page fault. Alla luce di quanto abbiamo detto, il numero di page fault può far trasparire dei problemi a livello di assegnazione dei frame.

Generalmente si può affermare che una scarsa quantità dei frame porta ad un alta frequenza di page fault; il viceversa è altrettanto vero. Sarebbe praticabile l'idea di monitorare la frequenza di page fault e di prendere un'azione di conseguenza.
Si stabiliscono degli **upper bound** e **lower bound** per la frequenza di page fault, in modo tale da monitorare la frequenza e stabilizzarla all'interno del range. Se la frequenza si innalza oltre l'upper bound, probabilmente il processo sta andando in thrashing. Andiamo allora a controllare i processi che presentano una frequenza più bassa di page fault: in questi processi è possibile che il numero di frame assegnati sia maggiore del working set. Questi processi presentano una frequenza che è più bassa del lower bound impostato:



Il modello del page fault frequency può essere messo in relazione con il working set. Notiamo infatti che se ci sono picchi di page fault, questi sono dovuti a cambi di località. Gli innalzamenti della frequenza di page fault sono naturali e mostrano un cambiamento del working set. Chiaramente è auspicabile che questi picchi si normalizzino nella fascia identificata dai lower bound e upper bound stabiliti. Inoltre, l'altezza di tali picchi dipende dalle modifiche di working set: se il cambio è radicale i picchi saranno più alti, ma se il cambiamento è minimo i picchi saranno più bassi. Ma picchi che perdurano per un po' di tempo e si manifestano su più processi sono sintomi di una situazione che non si può stabilizzare in maniera naturale.



Politica di cleaning

Il meccanismo di gestione dei page fault è efficiente se ci sono frame liberi sempre disponibili. Si adotta allora una **politica di pulitura delle pagine** (**page cleaning policy**) per velocizzare la gestione dei page-fault.
L'aspetto computazionale legato alla selezione della pagina può sembrare trascurabile, ma al crescere del numero di processi l'invocazione dell'algoritmo di sostituzione delle pagine può essere un aspetto da non sottovalutare. Per ottimizzare l'overhead, se esiste un frame libero si ha una situazione migliore: basta scegliere il frame libero per l'allocazione della pagina. Se invece i frame sono occupati bisogna:

- sostituire la pagina;
- salvare la pagina nel disco se essa è sporca;
- leggere dal disco la pagina per caricarla in RAM;
- aggiornare la tabella.

Sarebbe comodo allora avere sempre una **pool** di frame liberi. Un servizio apposito, il **paging daemon**, ha questo scopo. L'idea è quella di assicurarsi un set di frame liberi in ogni istante; il servizio mantiene il set in modo tale da far sì che ci sia sempre una certa quantità minima, così da poter gestire anche un numero grande di page fault. Un servizio del genere deve occuparsi di liberare i frame, sottraendoli così ad un processo proprietario. Questo potrebbe essere visto come uno svantaggio, in quanto si toglie una pagina quando non è strettamente necessario. Con opportuni accorgimenti però il servizio si rivela vantaggioso, e comunque l'algoritmo di sostituzione delle pagine verrà chiamato in futuro quando i frame esauriscono. Periodicamente se si scende sotto la soglia critica si invoca un certo numero di volte l'algoritmo di sostituzione in maniera globale.
Il lavoro di liberazione dei frame per inserirli nella pool permette che il successivo page fault accadrà in minor tempo. Si potrebbe controbattere, dicendo che la pagina che viene pulita potrebbe essere richiesta successivamente dal processo che l'ha vista sottratta. In realtà, quando la pagina viene messa nel pool dei frame liberi ha un contenuto se il frame non viene toccato, per cui il page fault può essere gestito senza la necessità di una richiesta I/O. Lo spostare una pagina nel pool di frame liberi mi permette di anticipare la parte algoritmica della scelta e di anticipare aggiornamenti della tabella delle pagine senza che il contenuo del frame venga alterato, "allegerendo" il page fault. La tecnica con ripescaggio ottimizza la stragrande maggioranza dei page fault che si possono verificare.

Dimensione di pagina

Un altro aspetto legato alla paginazione riguarda la dimensione delle pagine. Un sistema basato su paginazione sceglie una taglia per le pagine (e quindi per i frame) e su quella tutto il sistema lavora. Cerchiamo di capire come viene scelta la dimensione.

Alla luce di come viene gestito l'indirizzamento e la traduzione degli indirizzi, conviene lavorare con potenze di due. La misura, inoltre, dev'essere supportata dall'hardware.

In generale, se optiamo per una pagina di maggiore dimensione, si osservano le seguenti conseguenze:

- **tabelle più piccole**. Le pagine sono di maggiori dimensioni, per cui le voci sono in minor quantità;
- **efficienza nel trasferimento I/O**. Il trasferimento I/O viene fatto per blocchi più grandi, e questo è un efficientamento legato a dispostivi di storage elettromeccanico. Questi dispositivi hanno un meccanismo di accesso legato principalmente alla posizione della testina nel braccio meccanico. Leggere una pagina grossa ammortizza il tempo di posizionamento e la latenza di posizionamento per trasferire la stessa quantità di dati. Queste logiche cambiano su dischi elettronici;
- **ottimizzazione del numero di page fault**. Questo perché avendo pagine di dimensione maggiore, un singolo processo può far riferimento ad un indirizzo che possibilmente appartiene ad una pagina già caricata in RAM (per capire perché questo accade, basta pensare ad un pavimento costituito da piastrelle grandi. Se utilizziamo piastrelle grandi, è più probabile che il successivo passo che facciamo ricade nella piastrella in cui ci troviamo. Allo stesso modo funziona il meccanismo delle pagine: avendo una pagina di dimensioni maggiori, è più probabile che la successiva istruzione si trovi in una pagina già caricata in RAM).

Utilizzare una pagina più piccola tuttavia presenta i seguenti vantaggi:

- **riduzione della frammentazione interna**. Ricordiamo che la frammentazione interna è lo spreco di memoria dovuto ad una maggiore allocazione di memoria per un processo. Poiché le pagine sono più piccole, si evita di eccedere con l'allocazione dello spazio per un processo;
- **risoluzione del working set**. Avere pagine più piccole permette di migliorare il modo in cui approssimiamo la località, poiché la locazione di memoria presenta una maggiore precisione. Questo chiaramente richiede un valore di Δ più ampio.

La dimensione è anche legata alla memorizzazione delle pagine nel disco. Risulta strettamente utile che la pagina abbia una dimensione che sia un multiplo della dimensione di un blocco sul disco, o ancora meglio sia grande quanto la dimensione di un blocco nel disco. Se la dimensione di un blocco è maggiore rispetto alla dimensione di una pagina, allora si ha che per ogni scrittura di una pagina bisogna scrivere più informazioni sul disco.

17 - Pagine condivise

Pagine condivise

Si può risparmiare memoria utilizzando la **condivisione delle pagine**. Se istanziamo più processi a partire dallo stesso programma, vengono creati altrettanti processi che contengono nello stesso spazio di indirizzamento una parte legata al codice e una parte legata ai dati. Supponiamo che tre pagine siano dedicate al codice dei tre processi. Poiché però i processi sono istanze dello stesso programma, il contenuto delle tre pagine coincide. Ossia, abbiamo delle pagine che hanno contenuti duplicati. In termini di astrazioni, è sempre necessario fornire lo spazio di indirizzamento virtuale, pur condividendo le stesse pagine per i codici. Intuitivamente quello che si dovrebbe fare è arrangiare le voci della tabella delle pagine in modo tale che le pagine virtuali del codice vengano mappate ai frame contenenti il codice.

Se tuttavia un processo scrive nelle pagine condivise, gli altri processi che condividono la stessa pagina vedono le modifiche. Tuttavia in questo scenario questo non è un problema: le pagine del codice solitamente sono di **sola lettura**. In generale, si può ipotizzare che la pagina condivisa sia **in sola lettura**, e questa è un'ipotesi accettabile quando si parla di pagine che contengono codice. Esistono tecniche avanzate che prevedino che il codice possa essere cambiato, ma chiaramente sono tecniche di nicchia che in alcuni casi complicano lo sviluppo del codice. Si hanno quindi più benefici quando si imposta la pagina del codice in sola lettura.

Non è ancora un problema se le pagine sono condivise ma la condivisione è spontanea. I processi possono richiedere di condividere un segmento di memoria. Essenzialmente, due processi possono richiedere al sistema operativo di avere un segmento di memoria condiviso. Il sistema operativo accorda la richiesta, e i processi sono consci che un certo intervallo di indirizzi virtuali (gli intervalli possono anche non coincidere) identifica lo spazio condiviso. Il sistema operativo allora si assicura che le pagine condivise corrispondenti siano mappate al segmento di memoria condiviso. In questo caso le modifiche alle pagine condivise non sono anomalie ma sono strumenti di condivisione.

La tecnica di modifica delle tabelle funziona sia in tabelle ordinarie sia in tabelle multilivello. Tuttavia, il meccanismo non funziona con le tabelle di pagine invertite. Ad esempio, se un processo mappa la pagina 5 al frame 11 e un altro processo mappa la pagina 8 al frame 11, si ha un problema di **aliasing**: lo stesso contenuto è visto con due alias. In generale, poiché la tabella ha tante voci quanti sono i frame, il frame condiviso è identificato dall'ASID del primo processo con pagina 5. Ma il secondo processo appena esegue la ricerca **non trova la pagina**. Questa è una conseguenza della condivisione. In questo caso, il sistema operativo deve sistemare l'etichetta del frame. In questo modo il secondo processo può accedere al frame condiviso. Ogni volta che la CPU passa quindi ad un altro processo, si ha un "falso page-fault"

che richiede la sistemazione dell'etichetta del frame. In sistemi multicore la complessità del problema è ancora più grande.

Gestione della cache

Abbiamo già visto che in base al modo in cui la cache viene inserita nella pipeline CPU - MMU essa può lavorare con indirizzi logici o fisici. In particolare, se la cache lavora prima della MMU si abbassa la latenza perché la MMU non funge da collo di bottiglia. Se invece si utilizzano le cache con indirizzi fisici, non bisogna azzerare le linee nella memoria tampone.

Possono sorgere dei problemi quando si lavora con una cache a indirizzi virtuali a causa dell'aliasing. Se la cache contiene una linea che riguarda la pagina condivisa perché un processo vi ha fatto accesso, allora la cache memorizza la coppia pagina - ASID del processo che ha fatto accesso. Se un altro processo però vuole accedere allo stesso dato memorizzato in linea di cache, si ha un cache miss: la cache infatti non riconosce l'alias, caricando due copie della stessa linea. Questo non è solo uno spreco, ma diventa un problema di **coerenza e correttezza dei dati**: se un processo modifica il dato, l'altro processo farà riferimento all'altra linea di cache. La cache basata su indirizzi virtuali ha bisogno di riconoscere che si parla dello stesso contenuto ma con alias diversi. Se la cache lavora con indirizzi fisici, questo problema non si pone chiaramente, in quanto l'indirizzo fisico è univoco e non si ha la necessità di utilizzare gli ASID.

Si arricchisce allora il funzionamento della cache. Si basa sempre su indirizzi virtuali e utilizza dei **tag fisici**, ossia informazioni provenienti dalla collocazione fisica del contenuto per disambiguare la situazione precedente. Semplicemente la cache eseguirà una ricerca preliminare per identificare se ci sono delle possibili corrispondenze, ossia che potremmo riconoscere se ci sono linee che non corrispondono totalmente, quindi ottenere più candidati. La risposta finale verrà confermata dalla TLB, che darà in output l'indirizzo fisico.
Il meccanismo di caching illustrato prende il nome di caching **Virtually Indexed Physically Tagged** (**VIPT**). Ogni indirizzo in cache è costituito da un tag fisico e da un indice preso dall'indirizzo virtuale. Questo indice viene scelto con i bit di offset dell'indirizzo virtuale, in quanto questi bit vengono copiati all'interno dell'indirizzo fisico. Dopo la generazione dell'indirizzo virtuale, un lookup iniziale utilizza l'indirizzo virtuale per trovare se esistono dei match iniziali. Intanto, l'indirizzo virtuale viene mandato alla TLB per la traduzione dell'indirizzo, in modo tale da recuperare le informazioni relative al tag fisico. Si verifica tramite la TLB se il dato in cache è presente o meno. La ricerca in questo caso è in grado di risolvere il caso di aliasing in quanto si utilizza anche l'informazione relativa all'indirizzo fisico in memoria.
Da notare che se ad esempio una cache lavora con linee a 64 byte e in un indirizzo virtuale sono presenti 12 bit per l'offset, i 6 bit meno significativi vengono ignorati, in quanto la ricerca procede con l'identificazione della linea in cache, che dipende in questo caso dai 6 bit più significativi.

Copy-on-write, Zero-fill-on-demand

Possiamo applicare la stessa ottimizzazione applicata precedentemente anche copiando i dati. Alcuni scenari rendono applicare questo ragionamento facile. Uno tra questi è la `fork`.

Quando un processo viene forkato, il figlio ha una copia del codice, dello stack e dell'heap in un proprio spazio di indirizzamento. Subito dopo la fork, abbiamo due processi che per tutte le pagine hanno lo stesso contenuto. Questo significa che il sistema operativo può fare un'operazione che giustifica l'apparente mole di lavoro dietro il `fork`. Per evitare che il sistema operativo copi tutto il contenuto delle pagine, basta modificare la tabella delle pagine del figlio in modo tale che la tabella delle pagine del figlio punti agli stessi frame del padre. Se il figlio fa `exec` abbiamo quindi evitato l'allocazione di dati che poi venivano cancellati immediatamente. Tuttavia se il processo padre dopo `fork` scrive sulle pagine condivise, il figlio può leggere le modifiche.

La tecnica di **copy-on-write** consiste nel rimandare la copia dei dati nel processo solo se è necessaria. Supponiamo che le pagine dei processi siano condivise. Se il processo padre vuole scrivere su una pagina, la pagina viene duplicata, in modo tale che il figlio possa avere la sua copia. In questo modo il sistema operativo intercetta la scrittura.
Resta solo da capire come il sistema operativo possa intercettare le scritture. Semplicemente le pagine condivise hanno le maschere dei permessi rivisitate. Il sistema operativo pone nella tabella delle pagine del processo padre e del processo figlio che le pagine sono solo in lettura. In questo modo il sistema operativo manipolando i permessi delle pagine condivise può assicurarsi che qualunque tentativo di scrivere nelle pagine sia intercettato dall'hardware, creando un'eccezione e passando il controllo al sistema operativo. Il meccanismo di manipolazione dei permessi permette quindi al sistema operativo di intercettare la scrittura, ma il sistema operativo ha chiaramente tutti gli strumenti per capire se si tratta di un vero errore di permessi, assicurandosi che solo la pagina coinvolta venga duplicata. Dopo la duplicazione, i permessi delle pagine interessate vengono ripristinati.
Se il padre esegue più `fork`, ossia ha più figli, allora il sistema operativo non crea tante copie per quanti sono i figli, ma semplicemente crea una copia della pagina per il padre, ma gli altri figli rimangono con la stessa copia e non vedono i loro permessi modificati nelle proprie tabelle delle pagine. Questo approccio lazy limita quindi gli sprechi.

File system moderni cercano di supportare la clonazione di contenuti. La possibilità di clonare un file significa creare una copia, ma il sistema operativo può risparmiare andando a condividere gli stessi blocchi tra i due cloni. Clonazione significa creare due file indipendenti, ma ci aspettiamo quindi di modificare liberamente ogni singolo file. La copy-on-write su disco allora segue la stessa logica, implementando quindi la clonazione di un contenuto con costo quasi nullo in quanto lo spazio viene occupato quando le due copie divergono.

Per ottimizzare invece le pagine allocate in un processo si utilizza la **zero-fill-on-demand**. Questa tecnica è legata ad uno scenario abbastanza ordinario. Supponiamo di avere un processo che vuole eseguire un'allocazione di memoria, facendo crescere l'heap (nota: la gestione dell'heap viene gestito dal linguaggio di programmazione. Solitamente si utilizza l'impaginazione, cercando sempre di utilizzare dei buchi.). La libreria del linguaggio di programmazione utilizzato può richiedere al sistema operativo uno spazio in memoria (se non ci sono buchi). Lo spazio in memoria concesso dal sistema operativo allora viene inizializzato con sequenze di zeri. Tuttavia le pagine che vengono assegnate ad un processo potrebbero avere già del contenuto: bisogna quindi svuotare la pagina, e questo ha un costo.
L'idea allora è di giocare in anticipo. Invece di svuotare le pagine quando vengono chieste, basta svuotarle prima. Se ci sono dei frame vuoti, allora possiamo avere un pool di pagine vuote e cominciarle a svuotarle in momenti di idle. Si potrebbe utilizzare anche la DMA per sovrascrivere il frame libero interamente con zero.
In ogni sistema operativo inoltre è presente una **read only static zero page**, una pagina piena di zeri. Il sistema operativo allora può utilizzare una tecnica di copy-on-write sulla read only static zero page: basta infatti che quando un processo richiede n pagine a quel frame restituire n riferimenti a quella pagina. Nel momento in cui il processo scrive, si utilizza la copy-on-write, eseguendo effettivamente la ricerca di frame disponibili.

Librerie condivise

Le librerie condivise sono comunemente utilizzate. Abbiamo diversi modi di eseguire il linking:

- linking **statico**;
- linking **dinamico**.

Nel linking statico il codice farà riferimento alla libreria, aggiungendo il codice sorgente della libreria e generando un codice composto dal codice sorgente del file e dal codice sorgente della libreria. In larga scala tuttavia si possono identificare degli sprechi. Se parliamo di librerie molto utilizzate (quindi la stessa libreria è usata da tante applicazioni), il codice binario di due programmi hanno dei codici in comune. Si ha una situazione simile a quella delle pagine condivise. Sarebbe quindi utile condividere le porzioni di codice binario corrispondente alle funzioni. Tuttavia:

1. non è facile individuare quali sono le parti di codice condivise;
2. ammesso di avere indicazioni sui pezzi comuni, la tecnica di condivisione opera su pagine, ma i pezzi sono inseriti in codice e non è detto che in termini di dimensioni siano corrispondenti alle pagine, né sono obbligatoriamente allineati alla disposizione delle pagine.

In modo funzionale vogliamo mantenere una sola copia delle librerie condivise in memoria in modo tale da rendere un eseguibile più snello. Una libreria condivisa permetterebbe di fare un

linking dinamico, per cui il compilatore in fase di compilazione ritroverà all'interno del codice sorgente i riferimenti alle varie funzionalità della libreria (spesso il compilatore ha degli header e il codice pre-compilato in binario della libreria stessa). Il compilatore creerà un codice binario che corrisponde alla traduzione del codice scritto nel programma, e dove ci sono riferimenti in libreria ci sono riferimenti a dei segnaposti che si occuperanno di caricare la libreria in questione in memoria se non è presente in memoria e di far sì che si esegua il salto alla funzione corretta in libreria. Nel dettaglio il linker quando vede la chiamata a funzione di una libreria utilizza dei **riferimenti non risolti** a delle librerie precompilate (in Windows prendono il nome di **DLL**, **Dynamic Link Libraries**, librerie a collegamento dinamico). Il loader poi si occuperà di risolvere questi riferimenti e di caricare la libreria se non presente in memoria. Questo permette di caricare la libreria in memoria una tantum, per cui il primo utilizzatore carica la libreria in memoria e da quel momento in poi la libreria sarà utilizzabile dai processi che fanno riferimento a questa. In questo modo:

- gli eseguibili sono più piccoli, poiché fanno riferimento solamente al codice del sorgente compilato;
- essendo oggetti separati, il discorso della condivisione di codice diventa semplice;
- il binario viene creato in modo tale da funzionare con una libreria, ma non è vincolato a funzionare con quella copia, ma semplicemente alle specifiche delle funzioni esposte (parametri, tipi di ritorno). La libreria quindi può essere aggiornata indipendentemente dal codice del programma utente.

Se le librerie vogliono aggiungere funzionalità o cambiare alcune funzioni allora bisogna cambiare interfaccia. Si gestisce questa situazione con il versioning. Si dichiara fondamentalmente che una certa versione della libreria è compatibile con le applicazioni vecchie.

File mappati in memoria

Il sistema operativo può permettere di **mappare un file in memoria**. Nessun contenuto del file viene portato in RAM, ma si fa virtualmente corrispondere (mappare) il file in un'area di memoria virtuale. Se il processo accede a quell'area di memoria virtuale, allora accede al file nello stesso modo con cui accede ad una stringa, una serie di record o un vettore.

Consideriamo ad esempio un file di vari centinaia di megabyte. Un modo per accedere al database è di mappare il file ed eseguire letture e scritture dove ci interessa senza toccare tutto il resto. La stragrande maggioranza del file non viene portata in RAM.

Internamente un fetch agli indirizzi mappati sul file vengono gestiti con un **page fault**. L'MMU lo vuole tradurre come un indirizzo fisico, ma le pagine vengono segnate con bit di presenza pari a 0. Allora il sistema operativo si accorge che il contenuto è presente sul disco e lo porterà in memoria. Le modifiche vengono apportate sulla pagina portata in RAM e la pagina viene

marcata come presente ma sporca. In questo modo le modifiche vengono caricate solamente quando è necessario. Riutilizziamo quindi il meccanismo della paginazione anche per gestire i file. Questo è un modello di accesso ai file diventato standard, nonché più comodo in alcuni scenari.

Il meccanismo di mappatura dei file permette la gestione automatica delle librerie condivise. Un processo può avere la libreria mappata, e i salti saranno eseguiti in maniera coerente alla mappatura. Questo permette di:

- eseguire i salti in maniera del tutto user-space;
- migliorare l'efficienza nell'utilizzo di **librerie condivise**. Si possono portare in RAM solamente le parti che ci interessano, anche se il file della libreria è usato da più processi. Se due o più processi richiedono la stessa porzione di file, le voci delle tabelle delle pagine puntano alle stesse parti caricate in RAM;
- gestire la scrittura da parte di più processi. Due processi indipendenti che vogliono comunicare tra di loro possono farlo (su windows si ha la possibilità di farlo) creando un file e mappando lo stesso file. Questo porterebbe di fare la stessa cosa che sui segmenti UNIX consiste nella condivisione di un segmento di memoria. Qualunque scrittura fatta da un processo viene vista dall'altro.

Possiamo anche utilizzare la mappatura dei file per il caricamento del codice eseguibile. Ogni processo infatti presenta uno spazio di indirizzamento composto da codice e dati statici. Invece di caricare l'intero eseguibile in RAM, si mappa l'eseguibile e i dati statici allo spazio di indirizzamento del processo. Una volta mappato il codice può essere eseguito direttamente.

18 - Allocazione della memoria per il kernel, filesystem

Allocazione della memoria per il kernel

Abbiamo visto che la memoria per i processi utente prevede un sistema a paginazione con frammentazione interna. Vediamo ora l'allocazione per il **kernel**.

Il kernel, con i suoi molteplici thread, ha bisogno di allocare spazio in RAM per strutture dati per obiettivi riguardanti la gestione. Ad esempio, quando si crea un processo si ha bisogno di un PCB, ossia il blocco per il controllo della tabella dei processi. La tabella dei blocchi di controllo dei processi è una struttura dinamica, così come la coda dei processi. Allo stesso modo, il semaforo è allocato in maniera dinamica, allocandolo ogni volta che un processo lo richiede e disallocandolo quando non si ha più bisogno. Anche il kernel quindi ha bisogno di allocare memoria continuamente.

La memoria del sistema operativo dev'essere bloccata dall'esterno. Si potrebbe pensare che il sistema operativo abbia un suo spazio di indirizzamento, gestendo lo spazio di indirizzamento tramite la paginazione. Questo è parzialmente vero, in quanto i kernel moderni gestiscono una parte delle proprie esigenze utilizzando un meccanismo riconducibile a quello dei processi con tutti i vantaggi visti, come allocare ingenti quantità di memoria senza preoccuparsi della contiguità. Si hanno tuttavia dei limiti: in alcuni casi infatti bisogna utilizzare degli indirizzi fisici, in quanto alcune componenti hardware bypassano il meccanismo di indirizzi virtuali. Ad esempio, il DMA permette di trasferire i dati direttamente dalla memoria centrale bypassando la CPU, e quindi anche la MMU. Gli indirizzi quindi devono essere fisici.
Una conseguenza di questo è che le strutture dati devono essere obbligatoriamente contigue, in quanto si lavora con indirizzi hardware. Lavorando a basso livello, il sistema operativo può avere difficoltà ad utilizzare il suo spazio di indirizzamento. Inoltre vi è anche un problema nell'efficienza delle strutture dati. Le strutture dati per contenere i PCB e altre informazioni sono più piccole delle pagine.

In definitiva si utilizza:

- l'allocazione virtuale, con la stessa logica dei processi;
- l'allocazione fisica specifica del kernel.

Il kernel in base alle esigenze delle richieste utilizza uno dei due metodi:

- per strutture dati piccole e di taglia prestabilita, si utilizza il meccanismo di allocazione fisica del kernel;
- per allocazioni grandi utilizza la paginazione.

Ogni sistema operativo adotta una sua strategia di allocazione fisica. Linux adotta la **slab allocation**.

Soluzione linux: slab allocation

Per capire come funziona la **slab allocation**, definiamo prima cos'è uno **slab**.

Uno **slab** non è altro che una sequenza contigua di frame della RAM. Le **cache** invece sono costituiti da un numero finito di slab. In virtù del dato da memorizzare, la slab può avere dimensioni differenti (ad esempio, si può avere delle cache per memorizzare PCB che occupano 3 KB), permettendo una maggiore efficienza di allocazione.

Da notare che questo tipo di operatività non è possibile per i processi, in quanto non si sa a prescindere qual è la dimensione del dato dinamico da allocare. Invece, poiché il kernel sa qual è la dimensione del dato da memorizzare, possiamo utilizzare questa tecnica di memorizzazione.

L'idea è definire la dimensione della slab in modo tale che essa sia un multiplo della dimensione del dato da memorizzare. Supponiamo di avere uno slab di 3 frame, e che all'interno di questo slab ci entrano 6 strutture dati da 3 KB. Allora possiamo gestire lo slab con slot da 3 kb. Se arriva una richiesta da 3kb, si può piazzare in un qualunque slot dello slab. Ci si assicura inoltre che quando uno slot si libera esso possa essere riutilizzato per soddisfare altre richieste. Si evita quindi la frammentazione interna, in quanto si conosce già la dimensione del dato da ospitare, pertanto non si eccede a livello di spazio da allocare. Infine, la frammentazione esterna non è presente: ogni slot dello slab è utilizzabile allo scopo.
Se si esauriscono gli slab con degli slot liberi della cache, basta aggiungere un altro slab dove c'è spazio in memoria fisica, mappando un numero sufficiente per rispettare i vincoli visti (dimensione e contiguità dei frame).

Risulta un meccanismo molto efficiente, perché a parte la saltuaria allocazione degli slab non si hanno particolari overhead. Se ci sono slab liberi inoltre, si possono deallocare gli slab per i processi che necessitano della memoria. L'efficienza è anche presente in allocazione. Uno dei contro della gestione a pagine è che vi è un overhead dovuto alla traduzione degli indirizzi. Osserviamo che la gestione diretta della memoria fisica da parte del kernel ha overhead nullo, in quanto non si hanno meccanismi di traduzione.

File system

Per gestire grandi quantità di informazioni in maniera persistente e condivisa con vari processi si utilizzano i **file** e le **directory**. I file sono delle **unità logiche di informazione create dai processi**. La parte del sistema operativo che gestisce l'implementazione dei file è nota come **file system**.

Qualunque processo necessita di gestire informazioni in modo persistente. Anche le directory associate al concetto di file system sono astrazioni, in quanto sono collezioni di file o altre directory, creando un albero del file system partendo dalla directory radice.
Ogni sistema operativo implementa uno o più file system. Il termine stesso file system è anche ambiguo, in quanto il termine indica sia la componente del sistema operativo che si occupa di implementare i file sia l'insieme di file. Il fatto che il file system sia modulare permette al sistema operativo di lavorare con più file system.

Generalmente ogni sistema operativo ha una sorta di standard, supportando uno o due file system. Windows utilizza NTFS e per ragioni storiche anche FAT. Il sistema operativo o il singolo standard definisce anche alcuni dettagli relativi al file. I dettagli che possono differire tra i file e le directory (esistono anche altri oggetti astratti con i file, per ora tratteremo questi) sono i seguenti:

- **la nomenclatura**. Storicamente i file system per ragioni di efficienza avevano vincoli sulla dimensione del nome del file. MSDOS prevedeva 7 caratteri per il nome e 3 per l'estensione, mentre UNIX prevedeva 12 caratteri totali. Vi erano anche precisazioni sulla case sensitivity. Solitamente in sistemi UNIX file con la differenza di una sola lettera, in una maiuscola e in una minuscola, sono file diversi, mentre Windows non considera la case sensitivity (`test.txt` e `Test.txt` sono lo stesso file);
- **tipi di file**. Un dispositivo fisico (disco, scheda audio o video) su un sistema UNIX è tipicamente rappresentato come un file di dispositivo generale. Qualunque processo può leggere o scrivere come se fosse un file normale. Leggendo su un file dispositivo associato ad un dispositivo audio, si campiona, mentre se si scrive si riproduce un audio. Leggere e scrivere su un disco opera come lettura e scrittura grezza sul disco. Alcuni file system storici inoltre distinguevano i file di testo dai file binari, distinguendo chiamate di sistema differenti. Nei file system moderni si gestisce tutto come file binari che non hanno una struttura predeterminata, e la specializzazione viene gestita in user space (si hanno chiamate di librerie standard per la gestione dei file di testo, ma questo avviene a livello di applicazione: il processo utente usa le stesse chiamate di sistema che si utilizzano per un file binario). Infine, distinguiamo le **directory** dai file normali come dei file di sistema utilizzati per mantenere la struttura del file system;
- **tipi di accesso**. Alcuni file possono avere un accesso sequenziale o casuale:
  - in un accesso casuale si possono eseguire operazioni puntuali di lettura e scrittura in punti predeterminati;
  - in un accesso sequenziale si può solo leggere e scrivere a partire dall'inizio del file. Questo tipo di accesso è legato alla struttura del dispositivo utilizzato, come nastri magnetici, che necessitano un accesso sequenziale. Oggi qualunque dispositivo utilizzato permette l'accesso casuale;
- **metadati**. Il file system associa ad ogni file un timestamp basato sull'operazione eseguita sul file (come l'ultima modifica), la maschera dei permessi e il proprietario di un file. Ogni

utente in un contesto multi-utente può avere il diritto di avere uno spazio personale sul disco, gestire i propri file o anche condividerli. Il file system dev'essere in grado di permettere ciò. Ogni sistema operativo inoltre può indicare l'utente o i permessi in maniera diversa;

- **accesso condiviso**. Più processi potrebbero aprire lo stesso file, e ciò comporta lo stesso tipo di problematiche che si hanno con strutture dati presenti in memoria centrale o condivisa, quali race condition e sincronizzazione. Le problematiche sono del tutto simili, pertanto gli strumenti di lock sono fondamentali anche parlando di file. Questo problema viene gestito dai file system offrendo dei meccanismi di lock. Ogni sistema operativo inoltre gestisce i lock in maniera diversa, ma si possono sempre distinguere:
  - lock **shared** e **exclusive**. Un processo può richiedere un lock di tipo shared per la lettura, permettendo quindi ad altri processi di poter leggere sul file. Un processo invece richiede un lock di tipo exclusive per la scrittura sul file, precludendo l'accesso a qualunque altro processo. Il lock shared è compatibile invece con altri shared lock e non è compatibile con altri exclusive lock;
  - lock **mandatory** e **advisory**. Un lock mandatory è **obbligatorio**: i sistemi operativi che implementano questa strategia obbligano i processi a rispettare i lock. La richiesta ad esempio di una scrittura da parte di un processo per un file che presenta un lock shared viene bloccato dal sistema operativo. Il sistema operativo quindi si impegna per far rispettare i lock. Il meccanismo di lock è gestito dal sistema operativo e viene anche applicato da questo. Un lock advisory invece è tale che il sistema operativo permette ai processi di decidere chi possiede il lock, ma non fa nulla per vietare l'accesso. In altri termini, la gestione dell'accesso al file dev'essere un accordo tra i processi.

Tutto ciò che viene memorizzato nel file system dev'essere fatto in maniera **persistente**. La gestione da parte del sistema operativo quindi non si ferma alla gestione della rappresentazione, ma anche al modo con cui i processi hanno accesso a questi oggetti. Si deve allora inquadrare il concetto di **tabella globale dei file aperti** (sappiamo già che esiste per un PCB la tabella dei file aperti), ossia dei file aperti da qualche processo. Avere una tabella globale permette di gestire informazioni che vanno oltre al concetto di processo.

Struttura di un file system

La gestione delle partizioni non è concettualmente legata ad un file system. All'interno della partizione c'è un file system, ma la gestione della partizione di un disco non è vincolata al file system.

La partizione di un disco permette la divisione del disco in più parti. All'interno di una partizione possiamo specificare l'utilizzo di un preciso file system. Questo permette di utilizzare lo stesso disco per più file system (utile in Dual Boot).

Il meccanismo di partizionamento è legato ad uno standard noto come **Master Boot Record** (**MBR**). Il sistema di partizionamento prevede:

1. la definizione della partizione, includendo i limiti della partizione (dimensione, inizio, fine, file system);
2. gestione della fase di avvio del calcolatore.

Il primo blocco di partizione del disco prende il nome di **Master Boot Record** e viene utilizzato per accendere il computer. La fine della MBR contiene la **tabella di partizione**, che fornisce gli indirizzi di inizio e di fine di ogni partizione; una tra queste partizioni è marcata come attiva e viene utilizzata dal programma Master Boot Record. Si prevedeva inoltre che MBR supportasse al più quattro partizioni (nota: il limite era legato alla capienza dei floppy disk e dei primi hard disk. Oggi come oggi lo standard permette di creare partizioni fino a 2 terabyte e non viene più utilizzato).
Quando il calcolatore viene avviato, il **BIOS** (**Basic Input Output System**) esegue un programma per localizzare la MBR. Il programma del Master Boot Record localizza la partizione attiva e legge il **boot block**, che contiene le istruzioni necessarie per caricare il sistema operativo contenuto nella partizione.

Il sistema operativo di turno ha un suo file system principale, ed è anche il file system che si occupa di memorizzare all'interno tutti i file di sistema necessari per l'avvio del sistema.
All'interno di ogni partizione è presente:

- **boot block**, un refuso storico che non viene più sfruttato. Il boot record in MBR poteva scegliere di caricare un altro pezzo di codice che stava nel boot block, in quanto il codice del boot record doveva capire qual era la partizione attiva, e ogni altro sistema doveva capire solo quali programmi avviare per il sistema operativo. Il fatto che sia presente come primo blocco è solo come ragione storica: tutte le partizioni hanno il blocco, ma nessuno lo utilizza;
- **super block**, che conserva metadati sulla partizione. Contiene indicazioni sul tipo di file system utilizzato, e sulla dimensione del **cluster** in virtù della dimensione del blocco del disco. Un **cluster** è la più piccola unità di spazio che il file system del sistema operativo supporta. Solitamente il cluster è espresso in virtù della dimensione del blocco sul disco. Ad esempio, un file che occupa 1 KB in un file system il cui cluster size è 32 KB occupa 32KB sul disco.
- relativi blocchi per la memorizzazione delle strutture dati e dei file. Si hanno strutture dati per gestire **lo spazio libero**, strutture dati per la gestione delle **directory** e per la memorizzazione delle informazioni relative ai file salvati.

Attualmente si utilizza un layout moderno, noto come **GPT** (**GUID Partition Table**), definito dallo standard **EFI**. Esiste sempre un blocco centrale per definire il partizionamento del disco, senza limiti su dimensioni e numero. Una partizione prestabilita, la partizione EFI, ha un

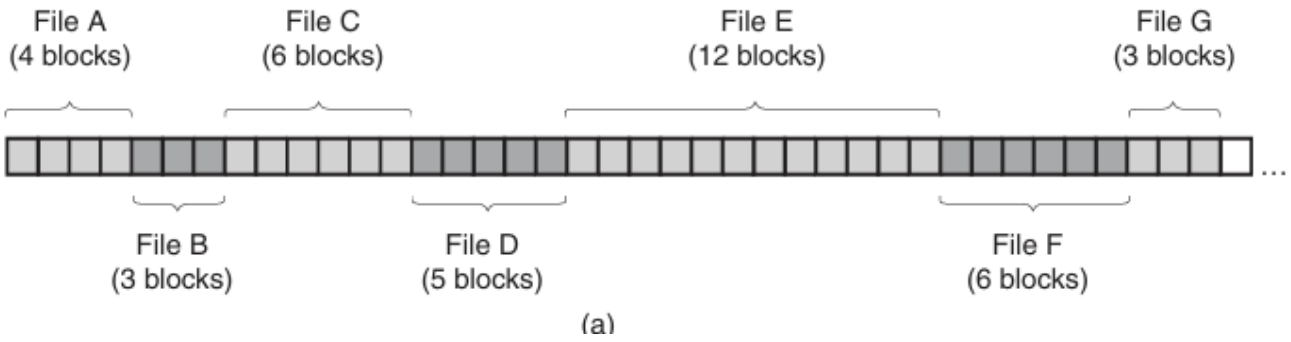
compito dedicato per l'avvio del sistema. La partizione EFI è formattata in FAT e all'interno del file system esistono vari file di configurazione e gli eseguibili EFI da lanciare in un ambiente minimale. Questi binari si occuperanno dell'avvio dei sistemi operativi. Linux solitamente pone in EFI il suo bootloader, supportando anche l'avvio di altri sistemi operativi, mentre Windows inserisce solo il bootloader per se stesso. La gestione dell'avvio multiplo è spostata all'interno del BIOS, che decide quale sistema avviare.

Implementazione dei file

Avendo una certa capienza, vogliamo implementare una moltitudine dei file. I file vengono creati, distrutti e modificati, così come accade nei processi. Le operazioni di allungamento dei file possono essere non banali quando implementate dal sistema operativo. Partiremo quindi da gradi di complessità più semplici a più complicate per la gestione dello spazio.

Allocazione contigua

La strategia più semplice di memorizzazione è la strategia di **allocazione contigua**, ossia i file sono allocati in blocchi contigui di memoria sul disco:



(a)

Se la dimensione del file è inferiore a quella di un blocco, si verifica una **frammentazione interna**. Il problema della frammentazione interna nel caso del file system è irrisolvibile, in quanto ce ne sarà sempre un po'. Se abbiamo allora dei file, la cui dimensione è rappresentabile in blocchi, possiamo memorizzare i file in blocchi contigui di memoria. Se un file ha bisogno di quattro blocchi, il blocco 1 dev'essere seguito dai blocchi 2,3 e 4. La directory inoltre sarà una struttura dati, una sequenza di record, in cui si memorizza il file, il blocco di inizio e il numero di blocchi usati per memorizzare il file. L'accesso diretto in questo caso è possibile tramite un calcolo basato su offset.

Per il concetto di efficienza, parlando di file system non ha senso parlare di efficienza computazionale. Banalmente in questo caso il numero di accessi in RAM per un accesso diretto ad un blocco è pari ad uno: per accedere al blocco i-esimo del file `E`, basta leggere il blocco di inizio di `E` e sommare i.
In generale l'allocazione contigua è vantaggiosa per i seguenti motivi:

1. per tenere traccia di un file basta memorizzare il numero di blocchi di un file e l'indirizzo del primo blocco. Questo quindi permette, come visto, di eseguire accessi diretti rapidamente;
2. la lettura di un intero file può avvenire in una singola operazione, eseguendo solamente un seek al primo blocco del file.

L'allocazione contigua sarebbe molto efficiente ma diventa problematica in una situazione dinamica, un po' come succede con i processi. Se si cancellano file e se ne creano di nuovi, i blocchi diventano liberi, ma vorremmo riciclare questi blocchi. Sorgono problemi legati alla frammentazione, e questi problemi si verificano non solo con la creazione e la rimozione di file, ma anche con l'allungamento di un file: se vogliamo aumentare le dimensioni di un file e c'è un file adiacente, dobbiamo spostare i blocchi per poter estendere tutto. Una memory compactation in questo caso è **impensabile**: spostare un blocco ha un suo costo, e farlo per migliaia e migliaia di blocchi è impossibile.
L'allocazione contigua quindi sarebbe piacevole ma è impossibile da utilizzare. Tuttavia questa allocazione è una soluzione adatta a file system **in sola lettura** (DVD-ROM e Blu-Ray ROM). In questi sistemi infatti le dimensioni dei file sono già noti a posteriori, pertanto utilizzare l'approccio di allocazione contigua permette di godere dei vantaggi analizzati precedentemente.

Allocazione concatenata

Se vogliamo gestire anche scritture, bisogna spostarsi su una strategia non contigua. Una prima strategia è l'**allocazione con liste concatenate**, o **allocazione concatenata**.

Si ha semplicemente una lista concatenata in cui ogni nodo rappresenta un blocco di un file e una parte del blocco funge da puntatore al blocco successivo. Ad esempio, se un blocco ha dimensione pari a 1024 byte, 4 byte sono dedicati al puntatore al blocco successivo (da notare che questa dimensione, ossia 4 byte, non è casuale: si tratta di una **word di memoria**). Sebbene non si soffra del problema di frammentazione esterna, si avranno problemi di frammentazione interna dovuti alla dimensione del blocco. Inoltre, gli accessi diretti sono impossibili: se vogliamo accedere all'i-esimo blocco, dobbiamo utilizzare gli $i-1$ blocchi e caricarli in RAM. La lettura sequenziale di un file, invece, è abbastanza semplice: basta seguire la lista concatenata.

Allocazione concatenata tramite FAT

Nella pratica si utilizza un'implementazione leggermente diversa dell'allocazione concatenata. La dimensione del blocco è sempre una potenza di 2, ma nell'esempio precedente utilizziamo solo 1020 byte. Questo può creare un disallineamento nel modo in cui il processo che utilizza il file lo vede e il modo in cui esso viene memorizzato. Risulta prassi eseguire letture e scritture per potenze di due. Se leggiamo il primo kilobyte di un file, dobbiamo leggere in realtà due kilobyte: i primi 1024 kilobyte infatti sono 1020 + 4 di puntatore, e quindi dobbiamo leggere 4 byte dall'altro blocco.

Possiamo spostare l'informazione relativa ai puntatori nella **File Allocation Table** (**FAT**). Questa tabella ha un record per ogni blocco allocabile dai file, ed è una collezione di puntatori al successivo blocco. Fondamentalmente, l'i-esima voce della tabella FAT dice qual è il successore del blocco. In altri termini, supponiamo di cercare un file `D` memorizzato nei blocchi 3,5,1,2. Questo file è rappresentato andando a valorizzare i campi della tabella FAT: nel posto 3 scriviamo 5, nel posto 5 scriviamo 1, nel posto 1 scriviamo 2 e nel posto 2 c'è un terminatore che indica che non è presente alcun blocco successivo. Serve quindi solo il nome del file e da dove inizia.

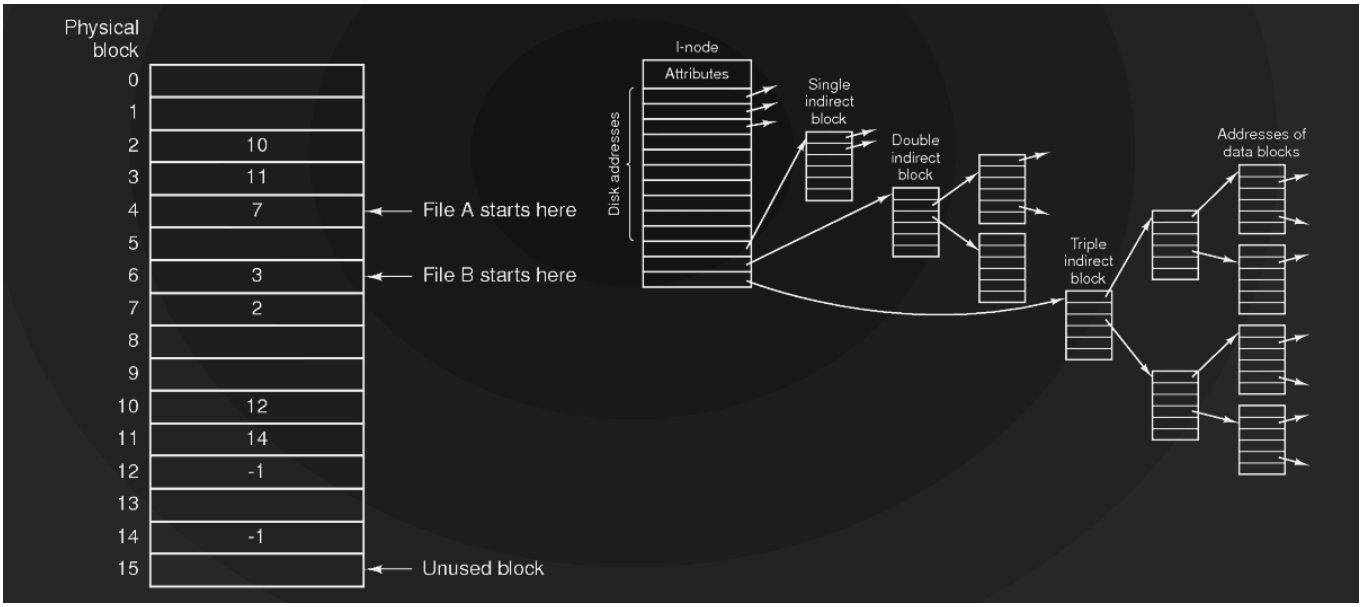
La FAT è memorizzata all'interno della partizione. Nei file system che utilizzano la FAT, se si perde la FAT **si perdono tutti i file**. Nella pratica veniva memorizzata in doppia copia e se si danneggia una copia si ha la speranza di avere una seconda copia.
Per ragioni di efficienza, sebbene la FAT sia salvata nel disco, quando il file system viene **montato** (si apre la possibilità di accedere al file system), la FAT viene caricata in RAM. Questo permette di risolvere il problema di efficienza dell'accesso diretto. In altri termini, se ho relativamente ad un file il suo blocco d'inizio, gli accessi multipli alla FAT costituiscono un costo trascurabile. Il problema tuttavia richiede che la FAT dev'essere caricata in RAM, e questo dipende da quanto è grande la FAT:

- più è grande il blocco, più piccola è la FAT ma maggiore è la frammentazione interna;
- più è piccolo il blocco, maggiore è la FAT e l'efficienza non è massimizzata in quanto può non essere caricata totalmente in RAM.

19 - I-nodes, directory, link e consistenza

Allocazione con nodi indice

Possiamo tenere traccia di quali blocchi sono utilizzati per allocare il file associandolo ad una struttura dati nota come **i-node** (**index-node**). L'i-node elenca gli attributi (ossia i metadati) del file e gli indirizzi dei blocchi del fil.e Ad ogni i-node è associato un numero, ossia l'**i-number**, che lo identifica. Dato l'i-node allora è possibile trovare tutti i blocchi del file. Il vantaggio di questo schema risiede nel fatto che ogni i-node dev'essere in memoria solo quando il file è aperto. Se ogni i-node occupa n byte e si aprono k file, allora la memoria occupata dagli i-node è $k \cdot n$. Da notare inoltre che l'**i-node è specifico del file ed è memorizzato sul disco**.



Tuttavia se vogliamo accedere all'i-esimo blocco del file dell'i-node non siamo ancora in grado di eseguire accessi diretti, in quanto dobbiamo leggere le voci dell'i-node. Vediamo tuttavia che i tempi richiesti sono minori rispetto al caso della lista concatenata.

Solitamente un i-node contiene 13 voci (per ragioni storiche). Per estendere la lista di blocchi associabili al file, si utilizza un'estensione della lista tramite una **struttura multilivello**. Fondamentalmente le prime voci dell'i-node punteranno ai primi blocchi del file. Quando le voci si esauriscono, il terz'ultimo blocco punta ad un indirizzo singolo, che punta ad un blocco del disco riempito di indirizzi di blocchi. All'interno di questo blocco, la prima voce punta ad un blocco in memoria usato dal file.
La penultima voce punta ad un blocco contenente sempre indirizzi verso blocchi, ma i blocchi puntati rappresentano un'ulteriore estensione della lista. Quindi non contengono numeri di blocco che contengono parti del file, ma sono numeri di blocco che estendono la lista.
Supponiamo che un blocco sia grande 4KB e che lavoriamo con indirizzi a 32 bit. La dimensione massima del file che possiamo gestire per ora è:

$$(10 + 1024 + 1024^2) \cdot 4\text{kilobyte}$$

Possiamo estendere ulteriormente utilizzando l'ultima voce dell'i-node. Questo contiene l'indirizzo di un blocco contenente un blocco di indirizzi. Ogni indirizzo tuttavia è un indirizzo ad un blocco nel disco che contiene altri indirizzi sul disco. Quindi il principio è lo stesso per la penultima voce, ma è applicato ad ogni indirizzo nel blocco puntato dall'ultima voce.

Cerchiamo ora di capire come accedere all'i-esimo blocco di un file. Il numero dipende da alcuni fattori:

- la dimensione del file;
- dove ricade il blocco.

Supponiamo di avere un file di massima dimensione, espandendo al massimo l'albero dei blocchi. Nel caso peggiore, che coincide con il caso in cui il blocco cercato ricade nel blocco di indirizzi più profondo, si ha un totale di quattro accessi, supponendo che l'i-node sia caricato in RAM:

- il primo accesso è richiesto per caricare il blocco puntato dall'ultima voce dell'i-node;
- il secondo accesso è richiesto per caricare il blocco di secondo livello;
- il terzo accesso è richiesto per caricare il blocco di terzo livello;
- il quarto accesso è richiesto per caricare il blocco cercato.

Da notare che il vantaggio di questo approccio risiede nel fatto che **il tempo è costante**.

Esempio esercizi

Supponiamo di avere un FAT e che il primo blocco in un file sia memorizzato nel blocco 7. Supponiamo che la FAT sia:

Indice	Next
1	4
2	3
3	15
4	5
6	12
7	1
8	2
9	3
10	-1 (null)

Capire qual è il blocco dove si trova l'offset 10100, supponendo che si lavori con un blocco di dimensione 4KB.

Dividiamo l'offset per la dimensione di un blocco del disco. Si ha:

$$\frac{10100}{4096} = 2, \dots$$

Non ci interessa chiaramente il numero preciso, ma sapere che si trova nel terzo blocco. Seguendo la FAT si ha che il blocco è 7->1->4. Quindi 10100 si trova nel blocco 4.

Allo stesso modo se abbiamo degli i-node, dividiamo l'offset per la dimensione del blocco per sapere qual è il blocco in cui si trova l'offset. Se cerchiamo qual è l'undicesimo blocco, basta consultare la prima voce della blocco a cui punta la tabella puntata dall'undicesima voce dell'i-node. Se vogliamo capire dove si trova il 3000esimo blocco del file, basta sottrarre prima 10, poi 1024, poi 1024 ancora e in generale continuare fino a quando non si trova un numero minore di 1024, tenendo conto della profondità in cui ci ritroviamo.

Implementazione delle directory

Quando un file viene aperto, il sistema operativo utilizza il path name fornito dall'utente per localizzare l'entry della directory sul disco.
Ogni entry della directory fornisce le informazioni necessarie per trovare i blocchi del disco associati ad un determinato file. Un problema legato all'implementazione delle directory è la memorizzazione di queste informazioni, nonché dei metadati.

Una prima possibilità, adottata da FAT, consiste nel memorizzare le informazioni all'interno della directory. In questo caso una directory è una lista di record (uno per file) di dimensione fissa che contengono:

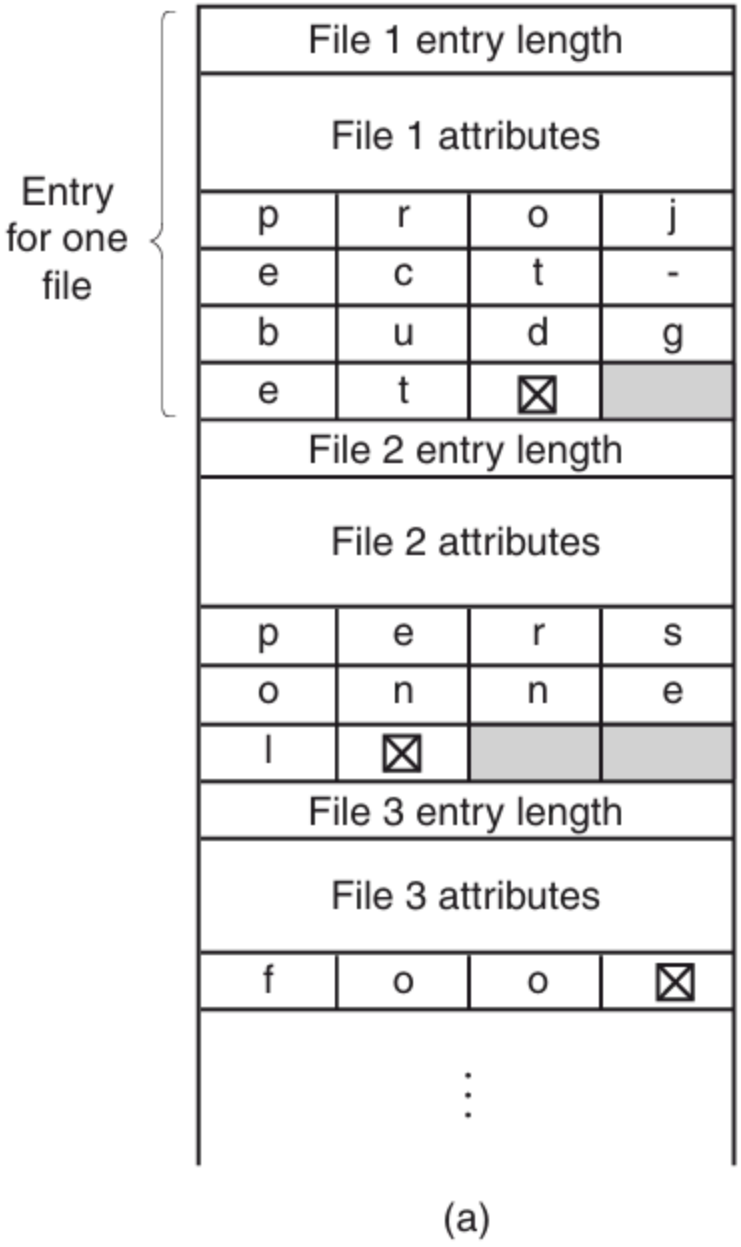
- nome del file;
- attributi del file;
- indirizzo del primo blocco.

In file system che adottano gli i-node si possono direttamente memorizzare i metadati negli i-node. In questo caso ogni entry della directory è il nome del file e il numero di i-node.

File system moderni oggi gestiscono nomi di dimensione medio grandi, anche di diverse centinaia di byte e in generale di **lunghezza variabile**. Si pensa allora ad una gestione dinamica per il nome nelle directory. Infatti una prima idea potrebbe essere quella di avere una lunghezza fissa per il nome di un file. Tuttavia, questo non è una bella idea se molti nomi non raggiungono il limite fissato: si ha uno spreco di risorse.

Supponiamo di generalizzare, e di avere nella voce una parte dedicata agli attributi e una parte dedicata al nome (la parte degli attributi nel caso degli i-node non c'è, ma consideriamo un caso generico). Per gestire ogni record nella directory, possiamo includere come prima voce la dimensione del record. Questo permette di saltare alla successiva voce senza leggere i byte in mezzo: infatti basta leggere il campo lunghezza di byte e si può andare avanti sommando all'indirizzo di inizio record il campo lunghezza. Il secondo campo sono gli attributi del file, un campo a dimensione fissa. Infine, si ha il nome del file. Il nome viene gestito in maniera differente:

- il nome prevede l'utilizzo di un delimitatore, il byte nullo, per la terminazione della stringa;
- c'è un padding per l'allineamento dei record alla word di memoria.



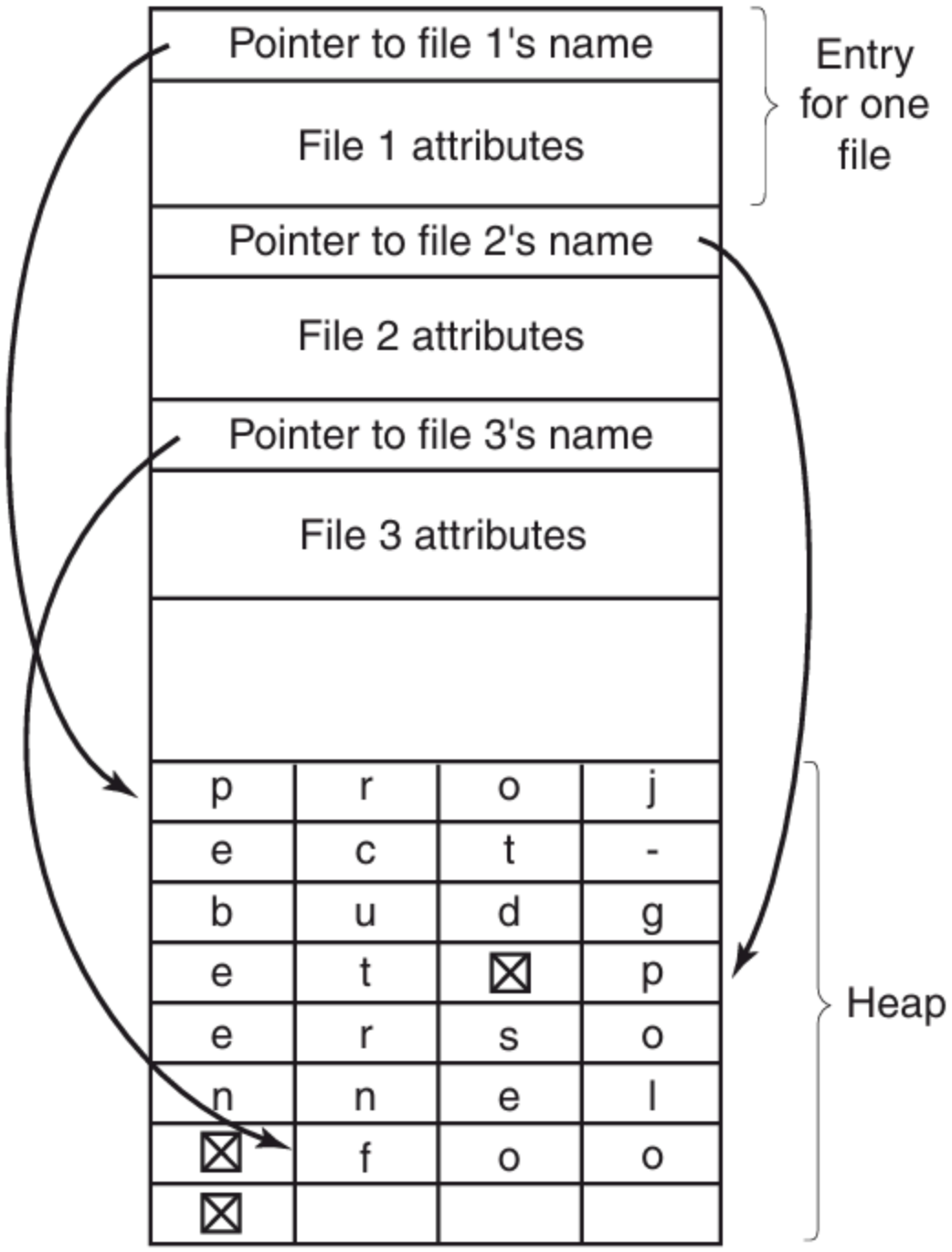
(a)

Il problema in questo caso risiede nella dinamicità dello scenario. Un po' come nel caso dell'allocazione dei blocchi ai file, specialmente utilizzando una strategia di allocazione

contigua, anche qui abbiamo dei record di dimensione variabile ma allocati in maniera contigua. Se eliminiamo un file, si può avere un buco. Col campo length si potrebbero gestire i buchi, ma non è detto che si possa memorizzare un altro file. Sarebbe quindi una frammentazione esterna: il vincolo di continuità impedisce l'aggiunta di un qualunque file. Quello che si può fare è compattazione: una directory contiene al più migliaia di file, e portarla in RAM per compattarla e riscriverla è una soluzione accettabile.

Si può tuttavia fare di meglio. Un approccio alternativo è quello di scindere le parti fisse dalle parti di dimensione variabile del record di directory. Nella parte fissa si ha un puntatore al nome del file, un indirizzo contestuale al contenuto del file, quindi contiene una distanza tra l'inizio della directory e dove inizia il nome del file all'interno dell'heap. Inoltre, nella parte fissa sono memorizzati i metadati.

Il problema della frammentazione esterna nella parte fissa non è presente: qualunque buco che si viene a creare è buono per ogni file. Nell'heap invece sono memorizzati i nomi del file, in quanto è una zona più semplice da gestire.



(b)

A livello di prestazioni, la ricerca di un file sarebbe lineare. Si potrebbe utilizzare una tabella hash, che permette di eseguire ricerce più veloci. Considerando anche il fatto che la questione dell'efficienza è legata alla dimensione delle directory, per directory medio-piccole una struttura dati vale l'altra. Nel fare più ricerche all'interno della stessa directory, a prescindere dalla struttura dati con cui si rappresentano le voci di questa, tutti i blocchi che coinvolgono la rappresentazione della directory saranno portate in RAM. Quindi anche se è una directory complicata, il fatto che si porta in RAM la rappresentazione alleggerisce il tutto. Tuttavia, la

hash table non è realmente necessaria in sistemi odierni che non richiedono particolari performance.

Condivisione di un file su un file system

Se due o più utenti vogliono condividere un file, abbiamo diverse alternative.
Usando la FAT, possiamo duplicare la lista con i riferimenti ai blocchi. Il problema risiede nel mantenere questi riferimenti. La FAT in realtà non offre ciò, ma si basa sull'utilizzo dei **soft link**.

I soft link sono degli oggetti universali che permettono di fare riferimenti anche al di fuori del file system. Il soft link, o link simbolico, contiene al suo interno il percorso del file referenziato e viene gestito dal sistema operativo. Ogni operazione applicata al soft link viene applicata sull'oggetto riferito dal link.

I limiti del link simbolico sono legati dal fatto che sono oggetti distinti che non gestiscono situazioni in cui l'oggetto puntato potrebbe venire a mancare o potrebbe essere spostato: il path name non viene infatti aggiornato, rendendo il link inconsistente. Se uno dei due utenti cancella il file condiviso, allora il soft link si rompe e l'altro utente non potrà usarlo.

Si può allora utilizzare l'**hard link**. L'approccio è basato su come funzionano gli i-node e la relazione tra directory e i-node. In un file system che fa uso di i-node, nella directory si trova il nome del file e il numero identificativo dell'i-node. L'idea è quella di sfruttare come meccanismo voci multiple che puntano al medesimo i-node. La differenza è che **non vi è un problema di mantenere coerenza**: l'unica cosa che viene duplicata, oltre all'i-number, è il nome. Il nome tuttavia è indipendente: l'hard link può avere un nome diverso dal nome originale, ma l'oggetto puntato continua ad essere il medesimo. Se si apre un file e si fanno modifiche, le modifiche saranno disponibili anche se si apre un collegamento. La gestione dei link multipli richiede che all'interno dell'i-node sia rappresentato un **contatore** (noto come **n-links**) che tiene traccia di quanti sono i link presenti. La situazione normale è che si ha un i-node e un solo riferimento. Nel momento in cui si crea un secondo riferimento, i metadati rimangono uguali (tra cui il proprietario), ma si aumenta il contatore.
La gestione dev'essere propedeutica anche in situazioni in cui bisogna cancellare la voce. La chiamata di sistema `unlink()` permette di cancellare un file se il contatore di riferimenti è posto a 1. Tuttavia, se si elimina un riferimento, si decrementa solo il count e l'i-node non viene eliminato. Tuttavia, se consideriamo un file il cui proprietario è di un utente C, esistono riferimenti per B e C ed eliminiamo il riferimento C, allora B ha l'unico riferimento al file, ma l'ownership è sempre di C.

Gli hard link non creano alcun overhead: accedere tramite un hard link richiede sempre l'accesso all'i-node, ma questo è come aprire un file. Tuttavia, se utilizziamo un soft link, prima bisogna leggere il soft link, scoprire il path name e poi leggere il file. Quindi i soft link sono

meno efficienti. Tuttavia, ci sono cose che non si possono fare con gli hard link e che si possono bypassare con i soft link.

Una prima caratteristica degli hard link è che proprio perché sfruttiamo lo stesso meccanismo di creazione di un file, il sistema può creare dei problemi osservando che applicando lo stesso principio per gli i-node delle directory possiamo creare link a directory. Nel caso dei soft link non si hanno complicazioni: basta indicare il path della directory e questo gestirà il problema. Con gli hard link si può fare la stessa cosa: la directory ha un suo i-node, per cui dentro una directory si può creare una voce che punta al medesimo i-number della directory. Si può tuttavia creare una situazione in cui si crea un riferimento ad una directory che sta più sopra nella struttura ad albero del file system. Questo porta alla trasformazione dell'albero in un grafo con un ciclo. Questo può essere un problema con operazioni ordinarie che si aspettano un albero, come una copia ricorsiva. Un semplice algoritmo di questo genere potrebbe andare in loop, in quanto non si avrebbe modo di saltare indietro. Sistemi che supportano hard link quindi vietano l'utilizzo di hard link per le directory.
Un'altra cosa che gli hard link non possono fare è **cross file system link**. In pratica, file system multipli possono essere accessibili tra loro. UNIX ad esempio ha un file system montato su root, e in altri **mount point** si possono montare altri file system. Quindi si monta tutto come si fosse un unico albero. Qualunque oggetto può raggiungere un altro oggetto partendo dalla radice. Windows invece tiene alberi separati etichettati con lettere dell'alfabeto. Pathname assoluti avranno un'etichetta prima. Se abbiamo un file e creiamo un link `F2` a quel file, possiamo supporre che il file stia su un file system e `F2` su un altro file system. Un soft link può essere creato per `F2`, ma per gli hard link no. Lo scenario è ovvio se il link e il file hanno diversi file system. Ma supponendo che abbiamo lo stesso file system, l'hard link è impossibile in quanto **ogni file system ha una propria numerazione per gli i-node**: la numerazione è univoca nell'ambito di quel file system. Quindi è come se in un processo si ha un puntatore ad una variabile in uno spazio di indirizzamento diverso: i puntatori hanno senso solo nello stesso spazio e la numerazione non ha senso perché non coincide.

Il discorso dell'attraversamento di directory nel soft link non è problematico perché qualunque tool può ignorare i soft link nell'attraversamento.

Gestione dei blocchi liberi

Abbiamo visto che per la gestione della RAM abbiamo bisogno di tracciare i blocchi di RAM liberi. Anche per il disco dobbiamo ricordare quali sono i blocchi liberi per la creazione di un nuovo file o per il suo allungamento. Abbiamo visto:

- le bitmap;
- le liste dinamiche.

Qui abbiamo una situazione simile. Possiamo utilizzare una bitmap che fondamentalmente occupando una quantità di memoria dipendente dalle caratteristiche del disco rappresenta con un singolo bit lo stato di allocazione di un blocco. La bitmap ha una dimensione fissa, e la sua dimensione dipende dalla dimensione di un blocco: blocchi più grandi portano ad una bitmap più piccola ma ad una maggiore frammentazione interna e il viceversa è altrettanto vero. La bitmap risiede sul disco, ma può essere portata in ram per questioni di efficienza. Se le bitmap sono troppo grandi, si potrebbero paginare in memoria virtuale.

Alternativamente si può utilizzare una lista concatenata che elenca i blocchi liberi. Questo elenco nella sua massima estensione cita tutti i blocchi del disco (caso in cui abbiamo appena formattato tutti i blocchi). Se una partizione ha poco spazio libero, inoltre, la lista concatenata è molto piccola. Da questo punto di vista la lista concatenata non ruba spazio e dinamicamente si restringe man mano che la partizione si riempie.
Oltre a indicare in ogni voce il numero di blocco nel disco, possiamo anche dare un'informazione relativa al **numero di blocchi liberi contigui**. In una sola voce si ha un range di blocchi liberi. Quindi se abbiamo l'intero disco libero, l'informazione è racchiusa in una sola voce. La voce andrà quindi a frammentarsi quando si creano i file, ma potenzialmente si raggruppano le zone libere con record più compatti.

Sebbene la lista concatenata sia un'implementazione intelligente, per via della sua semplicità si utilizza la bitmap. FAT, inoltre, non utilizza né la bitmap né la lista concatenata: non utilizza... niente. Questo perché l'informazione relativa ai blocchi liberi è ridondante. Il fatto che nella FAT ci sia una voce indica che il blocco è allocato. I file system basati su i-node utilizzano le bitmap, e in particolare è in una partizione che gestisce lo spazio libero. Inoltre, esiste una bitmap per gli i-node che dice fondamentalmente quali i-node sono liberi o meno. Quando bisogna creare un file, basta cercare un i-node, vedere qual è libero e popolarlo.

Controlli di consistenza

Un problema concreto è quello che succede quando si crea un'**inconsistenza** nella struttura dati del file system. Situazioni non gestibili via software legati a crash del sistema per driver instabili o mancanza di voltaggio richiedono particolare attenzione. I danni di tipo permanente possono essere arrecati al disco, proprio perché magari il crash ha interrotto una sequenza di operazioni applicate al file.

Si tratta di una situazione che non può essere ignorata: lavorare su inconsistenze è pericoloso, perché possibilmente blocchi indicati come liberi possono essere indicati come occupati su i-node. Se un blocco a causa del fatto che la bitmap lo segnala come libero lo si alloca ad un altro file, si crea una situazione in cui, se il file che usava quel blocco non era danneggiato, cominciandolo ad usare si può perdere il contenuto di file. Risulta necessario avere strutture dati **coerenti**.
Parliamo di coerenza in quanto c'è una certa ridondanza di dati tra tutte le strutture. I sistemi

operativi quando montano un file system effettuano una sorta di controllo di massima. Ossia, l'ultima volta che il file system è stato usato, si verifica se esso è stato chiuso correttamente.

All'interno del file system si ha un flag che traccia se il file system è stato montato: viene attivato se un sistema operativo lo utilizza, e viene posto a 0 quando in fase di spegnimento tutto va bene. In seguito ad un crash, il sistema operativo nota che il flag è posto a 1, per cui segnala un'anomalia. Si fa partire una utility di controllo della struttura del file system, analizzando la presenza di informazioni che sono coerenti in maniera automatica e rapida.

Motli sistemi operativi, pertanto, hanno utilità per controllare la consistenza del file system. Ad esempio, UNIX ha lo strumento `fsck` mentre Windows ha `sfc`. L'idea di `fsck`, che in realtà segue il principio di come diverse utility lavorano, si basa sull'utilizzo di due tabelle per controllare lo stato dei blocchi. Ogni tabella contiene un contatore per singolo blocco:

- una tabella traccia quante volte un blocco è presente in un file;
- l'altra tabella traccia quante volte un blocco è rappresentato nella lista di blocchi liberi. Nel caso di una bitmap, coincide con essa.

Poiché UNIX usa i-nodes, si leggono tutte le voci degli i-node, e ogni volta che si ha un blocco utilizzato, il relativo contatore viene aumentato di uno nella tabella che traccia gli utilizzi del blocco. Quello che ci aspettiamo è la creazione di due vettori di valori binari tali che uno è il complementare dell'altro. Anomalie di questo approccio possono essere:

- **un blocco che è segnato come usato e non usato contemporaneamente**. In questo caso basta segnare il blocco come non usato, e tipicamente è la situazione di **blocco mancante**;
- **un blocco che è segnato due volte come blocco libero**. In questo caso quando si utilizza una lista collegata il fix è ovvio: basta rimuovere il nodo e ricostruire la lista;
- **un blocco che è segnato due volte come blocco occupato**. Questo indica che nel file system esistono due file che utilizzano lo stesso blocco. In questo caso bisogna correggere l'incongruenza creando un blocco per uno dei due file. Uno dei due file risulterà come danneggiato, ma non si rischia di creare ulteriori danni.

Un ulteriore controllo viene fatto sugli i-node. Si scansiona il file system e ogni volta che si ha un riferimento all'i-node la voce associata ad esso è aumentata di uno. Il valore dovrà essere uguale al contatore presente all'interno dell'i-node. In questo caso il contatore potrebbe essere non sincronizzato:

- nel caso in cui il contatore sia più alto, il danno è ridotto, in quanto è solamente uno spreco nel disco;
- nel caso in cui il contatore sia più basso, si possono avere potenziali problemi. Infatti rimuovendo link al file, il contatore potrebbe azzerarsi prima, e un link potrebbe far

riferimento a questo i-node inutilizzato. Se un altro file utilizza i blocchi puntati dall'i-node, allora il link potrebbe far riferimento a questi dati.

In entrambi i casi, basta porre il contatore dell'i-node uguale al valore calcolato dal controllo.

Journaling

Risulta molto utile tracciare le operazioni eseguite nel sistema operativo. Un file system può avere un **journal**, e ogni volta che viene eseguita un'operazione sul file system si segna l'operazione. Si separa tuttavia l'operazione in micro operazioni che fanno parte dell'implementazione dell'operazione considerata. Se questa operazione viene interrotta, si ha un problema.

Il journal è una struttura dati residente sul disco che si trova in una partizione. Ogni volta che si esegue un'operazione, si descrive nel journal cosa si fa. Il sistema operativo si assicura di scrivere sul journal e di ripulirlo quando la macro operazione viene completata correttamente. Se c'è un crash, allora nel journal c'è una sequenza di operazioni. Bisogna pensare che questo non ci permette di capire **quale operazione è stata svolta o meno**, ma se c'è la sequenza di operazioni vuol dire che c'è un errore. Se si trova una sequenza di passi, allora si rieseguono i passi, assicurandoci che i passi vengano applicati. Questo può creare dei problemi: se alcune operazioni sono state già fatte, è possibile che reiterandole creino danni. Le operazioni in un journal devono essere **idempotenti**, ossia se reiterate devono garantire un medesimo effetto.

Le operazioni presenti nel journal riguardano solo quelle che influenzano i meta-dati. Alcune operazioni inoltre non vengono dettagliate. Ad esempio, inserire la descrizione dei blocchi scritti su un file sarebbe dispendioso. Le operazioni sui meta-dati sono tracciate in quanto garantire l'integrità dei meta-dati è più importante di memorizzare il contenuto. Il journal inoltre favorisce il tipo di controllo di consistenza. Nei sistemi moderni i controlli di consistenza non vengono eseguiti a tappeto su tutto il disco, ma sono mirati sulle directory segnalate nel journal.

20 - Caching nel disco, Scheduling del disco

Cache nel disco

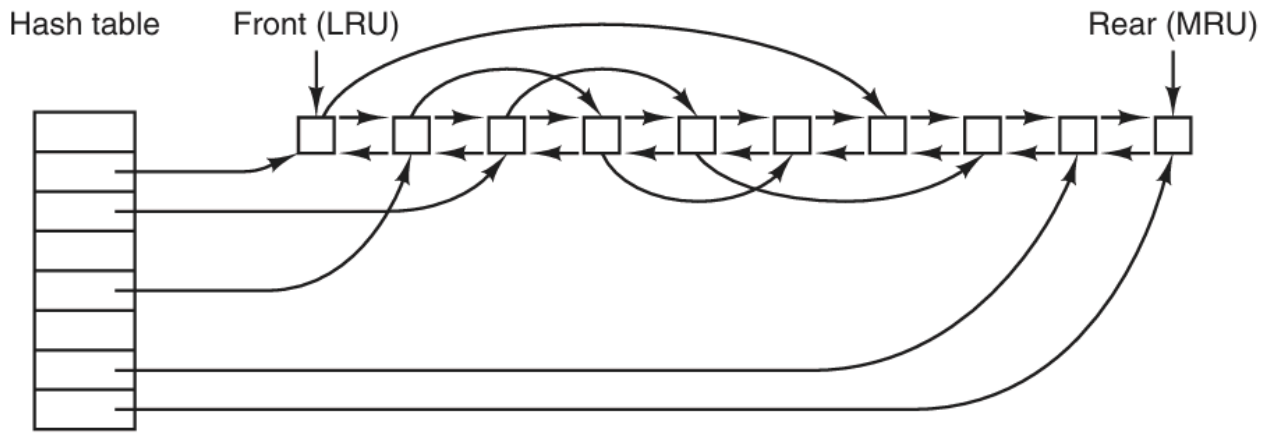
Per migliorare le prestazioni dei processi il sistema operativo migliora l'accesso ai dischi implementando via software una cache utilizzando porzioni di RAM.

Abbiamo già visto che la memoria virtuale ha problematiche simile alle cache hardware lì dove deve decidere quali pagine tenere in memoria. Il sistema operativo deve scegliere quali blocchi del disco mantenere e quali scartare perché la porzione di RAM dedicata al caching si è riempita. L'idea è sempre quella di utilizzare il principio LRU. Il vantaggio qui tuttavia risiede nella possibilità di **poter applicare LRU senza problemi**. Basta infatti mantenere un timestamp dell'ultima richiesta fatta di quel blocco: si guardano i timestamp e si capisce qual è quello che è stato usato di meno.

Per gestire la cache è necessario allora:

- avere un meccanismo che ci permette di eseguire **lookup veloci**;
- avere un meccanismo che ci permette di gestire il **blocco meno recentemente utilizzato**.

Si gestisce il lookup tramite una **tabella hash** e si utilizza una **lista doppiamente concatenata** per gestire qual è la pagina meno recentemente utilizzata. In questo modo quando si fa un riferimento al blocco, esso viene posto in coda e rappresenta il blocco MRU, ossia più recentemente utilizzato. Invece, l'elemento in testa rappresenta il blocco LRU. Le collisioni nella hash table sono risolte per concatenazione:



Una cache gestita in questo modo permette di agevolare sia operazioni di lettura sia di scrittura. Il sistema operativo può accettare chiamate di sistema che chiedono scritture sul disco ma mantenere la modifica in memoria, ritardando il momento in qui questa viene riversata sul disco. In linea di principio questo permette di evitare di scrivere due volte la stessa

informazione, ma il principale vantaggio si ha quando si utilizza un **disco elettromeccanico**, in quanto possiamo ordinare le scritture in virtù della posizione della testina sul disco.

Il problema che ora ci poniamo riguarda l'integrità delle informazioni. Ritardare la scrittura può portare a perdere dei dati perché le modifiche rimarebbero in cache (ossia in RAM). Il principio della LRU potrebbe inoltre causare problemi. Quando i blocchi vengono rimossi dalla cache, se sono sporchi i dati vengono ricopiat isul disco. Ma a livello di dati i blocchi che sono i più recentemente utilizzati e si trovano sempre nelle aree della coda della cache non saranno mai oggetto di effettiva scrittura. Quindi è vero che si evitano tante scritture, ma le zone più recentemente usate vengono perse in caso di crash.

Quando le scritture riguardano metadati del file system il sistema operativo non le ritarda: perdere i metadati (o averli danneggiati) è più rischioso in quanto si conservano le informazioni anche relative al file system stesso. Le scritture riguardanti i dati vengono invece ritardate, facendo però riferimento ad un tempo specifico entro il quale il dato dev'essere sincronizzato nel disco indipendentemente dal suo stato di utilizzo (LRU o MRU).

Si utilizzano anche altre due ottimizzazioni nella gestione della cache: il **free-behind** e il **read-ahead**.
Se si ha una scrittura su blocchi contigui della lista, probabilmente i blocchi verranno anche letti a breve. Se un blocco è stato scritto ed è stato individuato un comportamento di scrittura sequenziale, allora i blocchi che stanno dietro la scrittura vengono rimossi dalla cache. In quest'ottica la lettura può essere ottimizzata andando a leggere non solo il blocco richiesto ma anche i blocchi a seguire (questo è un meccanismo potente nel caso di letture su dischi magnetici). In questo modo oltre a leggere un blocco, si leggono anche i blocchi a seguire. Il costo di trasferimento è proporzionale ai blocchi letti, ma è una parte minore del costo totale di lettura: la parte principale del costo risiede nel posizionamento della testina.

Applicazioni nei sistemi operativi moderni

Windows permette l'utilizzo di partizioni exFAT per unità removibili, che è un FAT con la possibilità di gestire dischi più grandi. Tuttavia Windows incoraggia l'utilizzo di NTFS, il file system principale obbligatorio per qualunque partizione che deve supportare Windows. NTFS è un sistema abbastanza complesso e presenta alcune funzionalità viste:

- journaling;
- cifratura a blocco e a livello di file singolo. Associa la chiave di cifratura ad un utente e ad una password;
- compressione per i dati che devono essere scritti sul disco, risparmiando spazio sul disco;
- copy on write per la clonazione del file. Quando un file viene clonato, se non avviene nessuna scrittura i due file fanno riferimento agli stessi blocchi nel disco. Questo può essere

utile per la creazione degli **snapshot**, ripristinando interamente il sistema semplicemente clonando gruppi di file (**copia shadow**);
- supporto per dischi multipli (RAID).

Linux supporta molti file system, ma di standard utilizza **ext4**, che supporta journaling e tecniche che ottimizzano scritture. Inoltre, supporta BTRFS, un file system evoluto che include:

- journaling;
- **check-sum** per i dati e metadati. Questo assicura che quando il file system ricostruisce i dati essi siano effettivamente corretti;
- gestione di volumi e clonazione;
- supporto per dischi multipli (RAID).

Infine macOs utilizza APFS.

Scheduling nel disco

Ci concentreremo ora sul tempo necessario per le testine di posizionarsi tra i vari anelli del disco.
Il sistema operativo in un contesto di multiprogrammazione può vedere multiple richieste I/O sul disco che però possono essere inviate al controller solo uno alla volta. Si crea quindi una **coda di richieste pending**, chiedendo al controller di elaborarle. Il sistema operativo allora può migliorare la gestione della coda, scegliendo quale richiesta passare. Il modo infatti in cui le richieste vengono inserite in coda influenza la velocità di esecuzione e di elaborazione delle richieste. Il tempo di accesso relativo ad una singola richiesta è la cosa più importante, ma bisogna fare anche un ragionamento globale guardando tutte le richieste e vedendo nell'ottica di ottimizzare il throughput del disco (ossia **quante richieste si gestiscono per unità di tempo**).

Nell'ottica del disco elettromeccanico, il costo globale si può scindere in tre tempi:

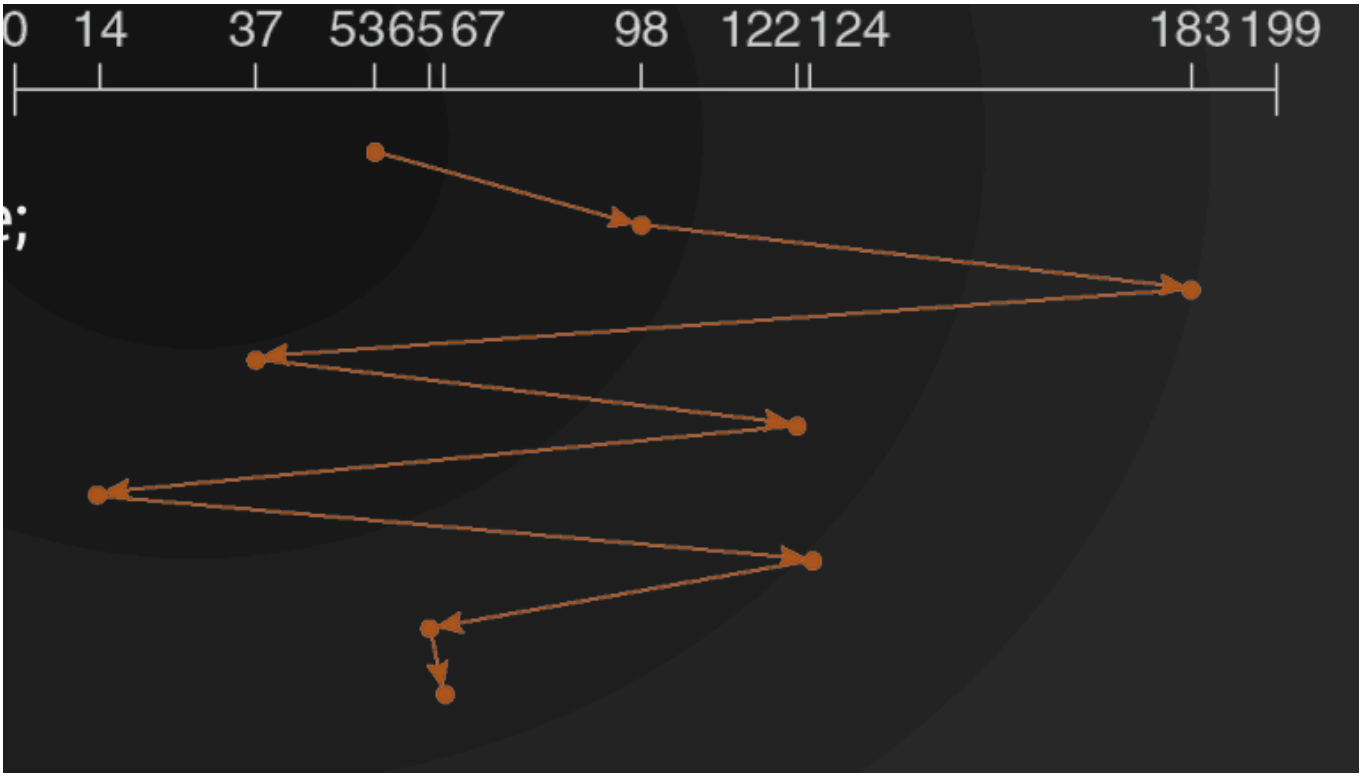
- tempo di trasferimento, una sorta di costante. Scrivere un blocco ha un costo fisso indipendentemente dalla sua posizione nel disco;
- tempo di posizionamento della testina, noto come **seek-time**;
- latenza di rotazione.

Il tempo di posizionamento è predominante, ed è il tempo necessario per posizionare la testina sul cilindro corretto. Cercheremo di ottimizzare il tempo di posizionamento in virtù delle richieste.
Un discorso simile può essere fatto anche per la latenza di rotazione. Posizionata la testina sul cilindro corretto, ci potrebbero essere richieste multiple su questo cilindro: cerchiamo allora di soddisfare queste richieste multiple.

Quando si parla di scheduling di richieste I/O verso il disco e cerchiamo di minimizzare il tempo di posizionamento della testina, parliamo di **scheduling di posizionamento della testina sul disco**. Consideriamo allora un insieme di richieste e consideriamo alcune metriche. La lista di richieste di riferimento è la lista: **98, 183, 37, 122, 14, 124, 65, 67**. Partiamo inoltre dal cilindro **53**. Non andremo a guardare la richiesta proveniente da processi o su file precisi, ma tradurre le richieste in termini di **coordinate dei cilindri sul disco**, identificando con numeri crescenti i cilindri in virtù della loro distanza dal perno (ossia il cilindro 0 è il più vicino al perno). Inoltre, non ci interessa qual è il piatto, ma solamente il cilindro.
In termini di seek time, pagheremo un tempo in virtù della **distanza tra i cilindri**. Ci concentreremo allora sulla **distanza percorsa**, ossia la somma di tutte le distanze tra i cilindri riferite da richieste consecutive.

First Come First Serve (FCFS)

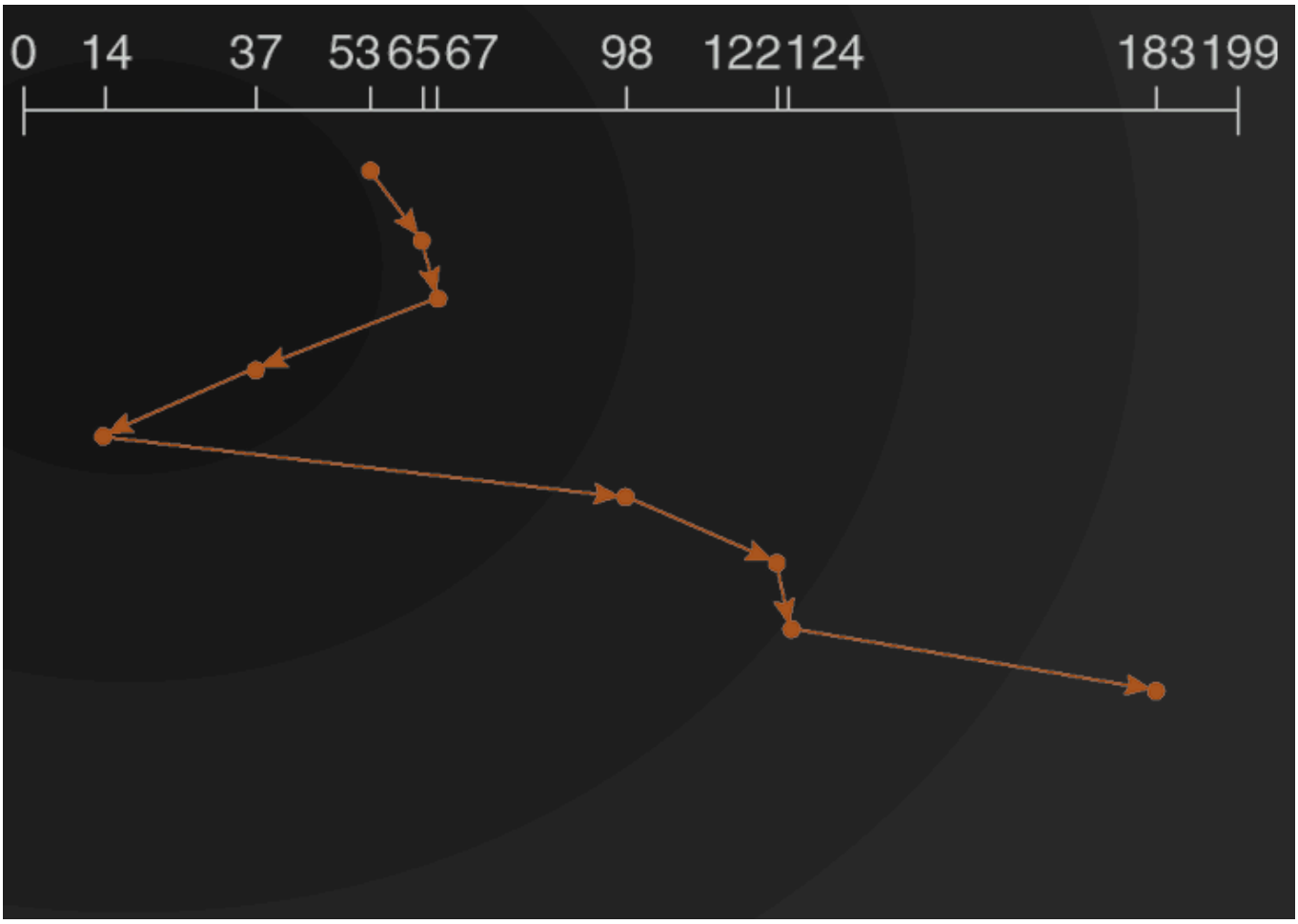
Il primo algoritmo che analizziamo in virtù della sequenza vista è l'**algoritmo FIFO** o **FCFS**. Si paga in questo caso un seek-time globale pari a 640 tracce di distanza tra richieste successive. Si ha un zig-zag continuo come mostra il grafico sottostante:



Sebbene sia semplice, si hanno prestazioni mediocri.

Shortest Seek Time First (SSTF)

In questo caso ordiniamo i seek in virtù della distanza minima tra le richieste. L'ordine usato in questo caso è il **65, 67, 37, 14, 98, 122, 124, 183**. In questo caso si osserva una distanza totale di 236 tracce, fornendo ottime prestazioni ma causando **starvation** per le richieste che si trovano molto distanti. La tecnica quindi non viene utilizzata in quanto non è equa.

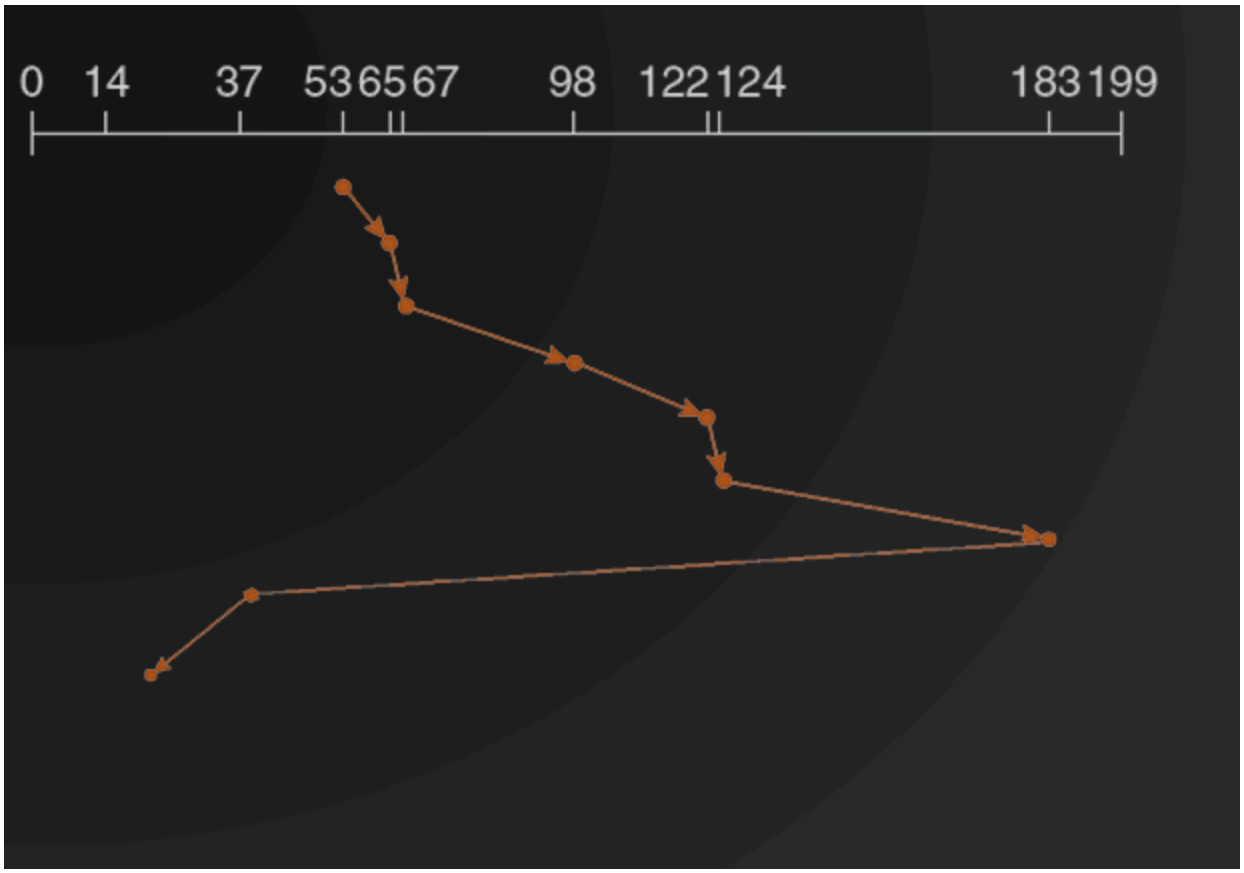


Scheduling per scansione (o algoritmo dell'ascensore o LOOK)

L'idea è quella di servire le richieste, ma nello scegliere la prossima richiesta l'algoritmo di scansione non solo ha un dato relativo a dove si trova attualmente, ma anche un dato relativo ad un **verso**, andando verso numeri di cilindro più grandi o più piccoli. L'ordine in questo caso è il seguente: **65, 67, 98, 122, 124, 183, 37, 14**. Pertanto l'algoritmo analizza se non ci sono richieste pendenti che hanno numeri di cilindro più grandi, l'algoritmo cambia stato interno e procede con richieste con numero di cilindro più basso.

La distanza totale in questo caso è pari a 299. Tuttavia **non soffre di starvation**: si può sempre dare un upper bound oltre il quale la richiesta già in coda attende. Il caso peggiore infatti è che la richiesta risiede nella posizione immediatamente precedentemente a quella della testina, e che la testina si muova in una direzione opposta.

L'algoritmo prende anche il nome di **algoritmo dell'ascensore** per il modo in cui gli ascensori si muovono nei grattacieli. L'algoritmo dell'ascensore è quella più utilizzata oggi dai dischi.



Scheduling per scansione circolare

Esiste una variante dello scheduling per scansione, ossia la **variante circolare**. La variante circolare immagina di ritornare dall'altra parte una volta raggiunto il cerchio più interno o più esterno. In questo modo la sequenza diventa: **65, 67, 98, 122, 124, 183, 14, 37**. La circolarità è legata al fatto che le richieste vengono servite in una sola direzione (a livello di crescenza/decrescenza del numero di cilindro).

Il motivo dietro questa implementazione risiede nel fatto che in una situazione ad alto carico si garantisce un tempo medio di attesa più basso. Nella situazione in cui si cambia direzione, se si sono aggiunte richieste in cilindri precedentemente visitati, il ritorno è più lento, poichè bisogna soddisfare le richieste aggiunte. Per questo motivo il caso peggiore illustrato nello scheduling per scansione viene gestito più rapidamente, per il fatto che richieste intermedie non vengono gestite, trattando prima le richieste aggiunte in precedenza.

Esempio

Supponiamo di avere un disco con 200 tracce numerate da 0 a 199 la cui velocità di seek è di 1 traccia per ms. All'istante $t = 0$ il sistema operativo sta servendo una richiesta sulla traccia 100 e in coda ci sono già le richieste per le tracce 50, 115 e 180. Successivamente arrivano altre richieste all'istante $t = 70$ per la traccia 150 e all'istante $t = 130$ per la traccia 90. Calcolare il tempo di ricerca complessivo in millisecondi per servire tutte le richieste secondo la politica

LOOK, iniziando in ordine ascendente e trascurando la latenza rotazionale e il tempo di trasferimento. Indicare anche la sequenza di scheduling considerata.

La sequenza di scheduling chiaramente è: **100, 115, 180, 150, 50, 90**. Le distanze sono, in ordine: **15, 65, 30, 100, 40**. Più precisamente:

- all'istante $t = 15$ viene soddisfatta la richiesta 115;
- all'istante $t = 80$ viene soddisfatta la richiesta 180;
- all'istante $t = 110$ viene soddisfatta la richiesta 150;
- all'istante $t = 170$ viene soddisfatta la richiesta 90;
- all'istante $t = 210$ viene soddisfatta la richiesta 50;

Sistemi RAID (Redundand Array of Inexpensive/Independent Disks)

Sistemi operativi per server potrebbero avere la necessità di gestire più dischi. I **sistemi RAID** sono un'ulteriore astrazione offerta dal sistema operativo. I sistemi prendono un certo numero di dischi e implementano un disco logico usandolo come un disco ordinario, ma effettuando letture e scritture su vari dischi. Questo permette di sviluppare sistemi che riesca a **resistere ai guasti**, ossia sistemi che **non perdono dati in caso di crash**.

Si può ottenere un vantaggio in termini di trasferimento di dati effettuando lo **striping** delle informazioni. In generale invece di lavorare su file lavoreremo su entità logiche. Immaginiamo allora di avere un file medio grosso, e di doverlo memorizzare su quattro dischi. L'idea è di suddividere il file in parti, che prendono il nome di **stripe**. Uno stripe è tale da avere dimensione che è multiplo della dimensione di un blocco. Ogni stripe viene memorizzato su un disco diverso. Se utilizziamo ad esempio 4 dischi e memorizziamo il file, esso viene diviso in 4 stripe, e ogni disco memorizza $\frac{1}{4}$ del file. Il file verrà letto in un tempo ridotto di un fattore 4, dovuto al fatto che **ogni disco lavora in parallelo sulla lettura**.
Lo striping chiaramente mira al miglioramento del throughput, ma **non comporta la protezione dei dati**.

L'implementazione chiaramente è nascosta dall'astrazione. Chi si occupa di pilotare i dischi è l'implementazione dell'astrazione. Questa implementazione può essere via software o via hardware:

- solitamente sistemi server che devono pilotare un certo numero di dischi implementano queste tecniche in hardware. Il controller si occupa di esporre i suoi n dischi fisici al sistema operativo in un unico disco;
- sistemi operativi moderni permettono la gestione di molteplici dischi in modalità RAID.

In passato si supponeva che i controller fossero più veloci. I file system moderni, come BTRFS, NTFS e altri preferiscono (ossia hanno meno controindicazioni) pilotare una serie di dischi. Non si hanno grossi svantaggi rispetto ad un controller hardware; anzi, file system moderni come NTFS sono preferiti rispetto a controller hardware, che si rivelano controproducenti.

Il problema in questo tipo di sistemi riguarda il **guasto di un disco**. I sistemi RAID tuttavia permettono di gestire situazioni come il guasto e la sostituzione del disco. Quando un disco si guasta, esso viene sostituito e il sistema lo flagga come inutilizzabile. Un operatore sostituirà fisicamente il disco, e il sistema RAID nota il cambiamento e il livello di guasti al disco viene ripristinato.
Quando un disco si guasta, il sistema viene detto **degradato**. Spesso i sistemi RAID utilizzano dei dischi di riserva, ossia degli **spare disks**. Il sistema potrebbe totalmente in automatico escludere quello guasto e cominciare ad utilizzare un disco di riserva.

Livelli RAID

Quelli che sono in gergo noti come **livelli RAID** sono le configurazioni dei RAID.

RAID 0

Il RAID 0 è il livello dello striping. Il disco virtuale viene distribuito su n dischi in modalità round-robin, dividendo una certa capienza in stripe e assegnando queste stripe con modalità round robin con i dischi. Ad esempio, se dividiano 4 terabyte in 12 stripe e avendo 4 dischi, il primo stripe è nel primo disco, il secondo nel secondo, il terzo nel terzo, il quarto nel quarto e il quinto nuovamente nel primo disco, e così via.

Ragioniamo adesso a livello di richieste. Supponiamo di avere 100 richieste, e cerchiamo di capire il tempo necessario per gestire queste richieste in 4 dischi. Se siamo fortunati, le 100 richieste sono divise equamente in tutti e 4 i dischi. Ogni disco allora si fa carico di $\frac{1}{4}$ delle richieste. Il throughput quindi giova di un fattore 4. In generale, in queste situazioni in cui le richieste sono distribuite equamente su n dischi, il throughput aumenta di un fattore n.
Nel caso peggiore le richieste sono tutte indirizzate su un disco singolo. Quindi qui non si hanno realmente aumenti di throughput.
Nel caso medio se abbiamo un numero sufficiente di dischi potremmo riuscire ad avvicinarci ad un aumento di throughput pari ad un fattore n.

RAID 0 tuttavia è un sistema ad **alta vulnerabilità**. I dati sono più esposti a guasti se distribuiti su più unità. Sorge la necessità di avere ridondanza.

RAID 1

La ridondanza più semplice è la tecnica del **mirroring**. La tecnica del mirroring introduce l'idea di creare un duplicato dei dati, ed è applicabile su un qualsiasi numero pari di dischi. Ad

esempio, su un disco posso avere il file system, e su un altro disco posso avere la copia del file system.

Le richieste di scrittura che stavolta arrivano sui dischi devono essere replicate su tutti i dischi che fanno mirroring. Tuttavia, in letttura anche in casi in cui non stiamo usando lo striping abbiamo un vantaggio. Se abbiamo un gruppo di richieste per il volume logico, possiamo suddividere le richieste sempre in due sottogruppi:

- una prima metà sui dischi originali;
- la seconda metà sui dischi mirror.

Questo è un vantaggio di un fattore 2 netto, ed è sempre costante. Se si usa lo striping, RAID 1 migliora di un fattore 2 le performance di RAID 0.

In questo livello di RAID abbiamo una migliore **fault tolerance**, dovuto al fatto che se perdiamo il contenuto del disco abbiamo una copia. Tuttavia, qui abbiamo un **overhead di storage**.

RAID 2

Si potrebbe risparmiare spazio sul disco utilizzando una strategia differente. Si applica l'idea dello striping aggiungendo ridondanza, aggiungendo degli **Error Correction Codes** (**ECC**). Più precisamente, **si utilizza uno striping a livello di bit con ECC**. Invece di utilizzare un checksum, che permette solo di identificare un errore, un ECC permette di individuare ed entro certi limiti di **correggere** un errore.

Un esempio di ECC è il **codice di Hamming**, che a partire da 4 bit di dati ne crea 3 di ridondanza. In particolare, di questi bit potremmo salvare un bit su ogni disco. Se si guasta il disco relativo, ad esempio, al quinto bit, allora siamo in grado di ricostruire le informazioni interamente.
Il sistema RAID 2 tuttavia funziona bene solo se **i dischi lavorano in estrema sincronia**. Questo non è presente negli altri livelli. Infatti, se dobbiamo fare una richiesta ad un disco, questo la elabora non appena è possibile farlo. Il sistema RAID 2 invece richiede sincronia, in quanto per ricostruire un blocco si ha la necessità di attivare tutti e 7 i dischi. Non si può quindi ricostruire il blocco originale senza interpellare tutti i dischi. Se un disco ha problemi in lettura, allora sta rallentando il processo di lettura del blocco logico.

RAID 3

In realtà avere uno striping a livello di bit con ECC potrebbe essere eccessivo. Potremmo infatti utilizzare un solo disco che mantiene tutti i singoli **bit di parità**. Il bit di parità permette di identificare al più un bit di errore, ma se c'è non sa dire dove. Tuttavia, spesso vorremmo anche ripristinare il blocco nel suo stato originale. Il bit di parità in sé non ci permette di ricostruire l'informazione. Tuttavia, se sappiamo a priori dov'e il problema, un codice di individuazione

degli errori ci permette anche di correggerlo. Supponiamo di avere quattro dischi, e di utilizzare un quinto disco per memorizzare i bit di parità. Allora se leggendo l'informazione `1011` perdessi l'informazione relativa al terzo bit, il bit di parità mi permette di dedurre il terzo bit. Quello che abbiamo guadagnato quindi è che abbiamo quasi lo stesso livello di capienza: su cinque dischi, un disco è impiegato per il bit di parità.

Il problema della sincronia tuttavia rimane per lo striping del bit.

RAID 4

Il RAID 4 applica la stessa idea del RAID 3 ma applica uno striping a livello di blocchi con uno XOR sull'ultimo disco. Partendo da un volume logico e avendo n dischi, si esegue un RAID zero sui primi $n-1$ dischi e sull'n-esimo disco si calcola il bit di controllo. In particolare, si esgue uno XOR di tutti gli stripe della stessa riga. Si sostituisce quindi il bit di parità con lo XOR.

Con RAID 4 siamo ancora in grado di ricostruire l'informazione. Se si perde uno stripe in un disco, siamo in grado di ricostruirlo facendo uno XOR tra tutti gli stripe che abbiamo e il blocco di parità. Quindi anche se un qualunque disco si guasta, siamo in grado di ricostruirlo con lo XOR, sfruttando il fatto che **XOR è commutativo**. Inoltre, non si hanno problemi di sincronia grazie allo striping che non avviene a livello di bit.

A livello di lettura si hanno gli stessi vantaggi che si hanno in un RAID 0, ma non si può sfruttare la conoscenza del blocco di parità: poter estrapolare l'informazione nel blocco di parità richiede la conoscenza degli stripe usati per il calcolo dello XOR. Per la scrittura abbiamo degli svantaggi. Infatti notiamo che per una scrittura necessitiamo di $n-2$ letture e di 2 scritture:

- $n-2$ letture perché per il calcolo dello XOR dobbiamo conoscere gli stripe dello stesso livello presenti negli altri dischi;
- 2 scritture perché bisogna fare la scrittura sullo stripe interessato e una scrittura nel disco che memorizza lo XOR.

Si può fare un'ottimizzazione. Infatti, notiamo che se facciamo uno XOR tra uno stripe e se stesso otteniamo uno stripe nullo. Da cui capiamo che lo XOR tra lo stripe di parità e lo stripe interessato alla scrittura è uguare allo XOR tra gli $n-2$ stripe. Il nuovo stripe di parità si può ottenere come lo XOR tra gli $n-2$ stripe e lo stripe modificato, ossia tra **il vecchio blocco di parità e il blocco sovrascritto**.

RAID 5

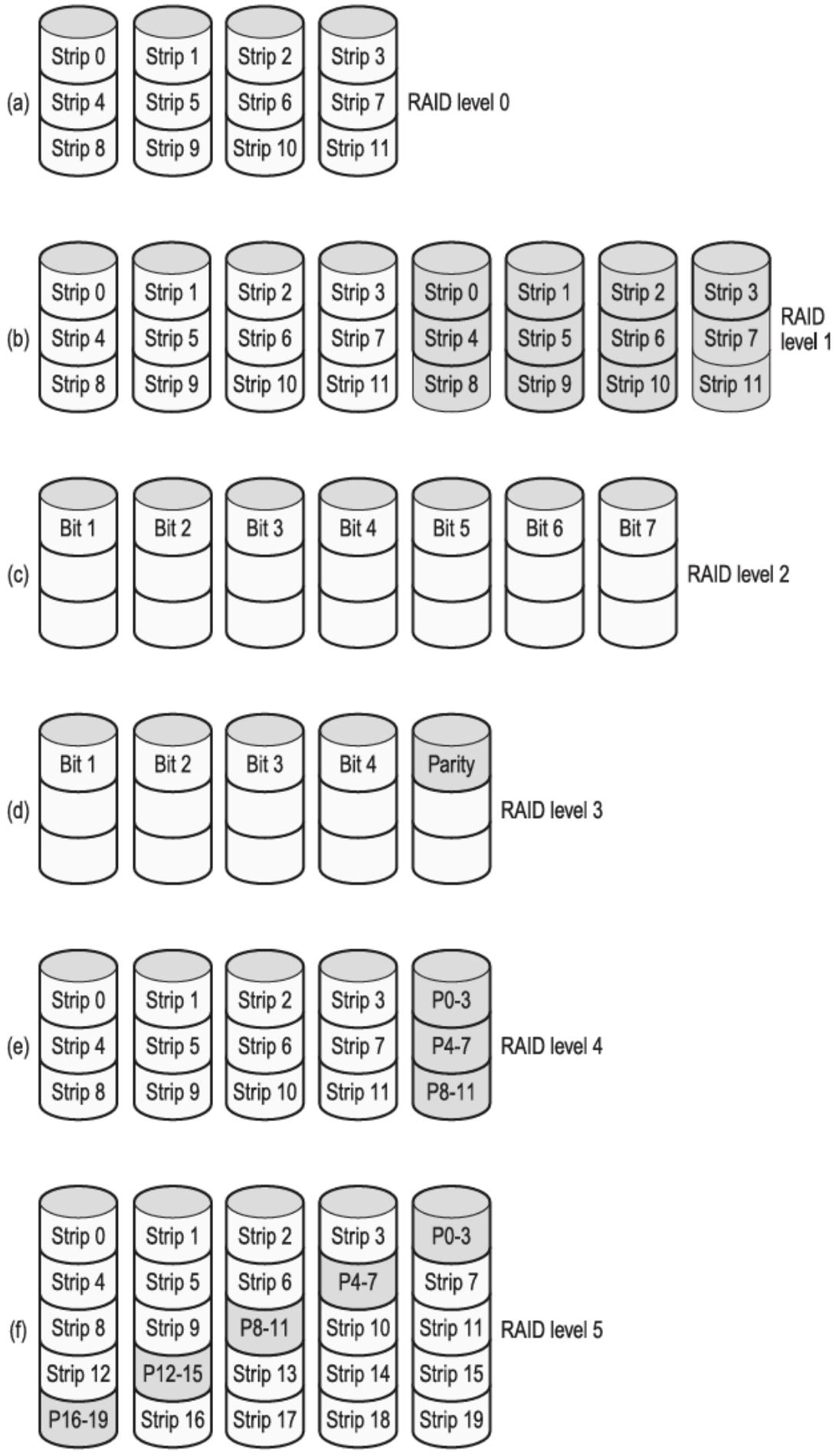
Il problema del RAID 4 è che l'ultimo disco è usato più degli altri. Qualunque scrittura su qualunque stripe infatti richiede una lettura e una scrittura sul disco su cui si memorizzano le

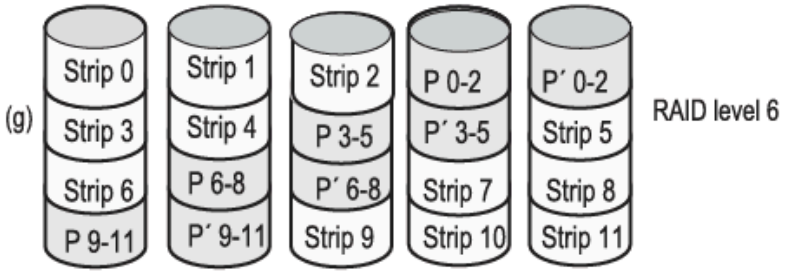
informazioni di parità. Quindi **è più probabile che il disco di parità si guasti prima a causa dell'usura**.

L'idea di RAID 5 è di mantenere la stessa configurazione, ma cambiare di posto gli stripe sulla stessa riga. L'idea quindi è che nella prima riga il blocco di ridondanza si mette in un disco, poi nella seconda riga in un altro disco, nella terza riga in un altro disco ancora eccetera. Essenzialmente si segue uno striping di bit con **stripe di XOR distribuito in modalità round robin**. Questo permette di avere la stessa resistenza ai guasti ma con operazioni più semplici: infatti ottenere il blocco di parità è calcolabile solo con lo XOR di stripe dello stesso livello.

Il vantaggio è che **la lettura aumenta di un fattore al più** n perché **tutti i dischi hanno blocchi di dati**.

Schema livelli RAID


(a)
Strip 0
Strip 4
Strip 8
Strip 1
Strip 5
Strip 9
Strip 2
Strip 6
Strip 10
Strip 3
Strip 7
Strip 11
RAID level 0
(b)
Strip 0
Strip 4
Strip 8
Strip 1
Strip 5
Strip 9
Strip 2
Strip 6
Strip 10
Strip 3
Strip 7
Strip 11
Strip 0
Strip 4
Strip 8
Strip 1
Strip 5
Strip 9
Strip 2
Strip 6
Strip 10
Strip 3
Strip 7
Strip 11
RAID level 1
(c)
Bit 1
Bit 2
Bit 3
Bit 4
Bit 5
Bit 6
Bit 7
RAID level 2
(d)
Bit 1
Bit 2
Bit 3
Bit 4
Parity
RAID level 3
(e)
Strip 0
Strip 4
Strip 8
Strip 1
Strip 5
Strip 9
Strip 2
Strip 6
Strip 10
Strip 3
Strip 7
Strip 11
P0-3
P4-7
P8-11
RAID level 4
(f)
Strip 0
Strip 4
Strip 8
Strip 12
P16-19
Strip 1
Strip 5
Strip 9
P12-15
Strip 16
Strip 2
Strip 6
P8-11
Strip 13
Strip 17
Strip 3
P4-7
Strip 10
Strip 14
Strip 18
P0-3
Strip 7
Strip 11
Strip 15
Strip 19
RAID level 5



Solid State Disk (SSD)

Gli SSD utilizzano tecnologie più simili alle RAM, garantendo però una memoria non volatile. Sono caratterizzati da una forma di usura legata alla cancellazione dei dati. In generale gli SSD hanno molti aspetti positivi:

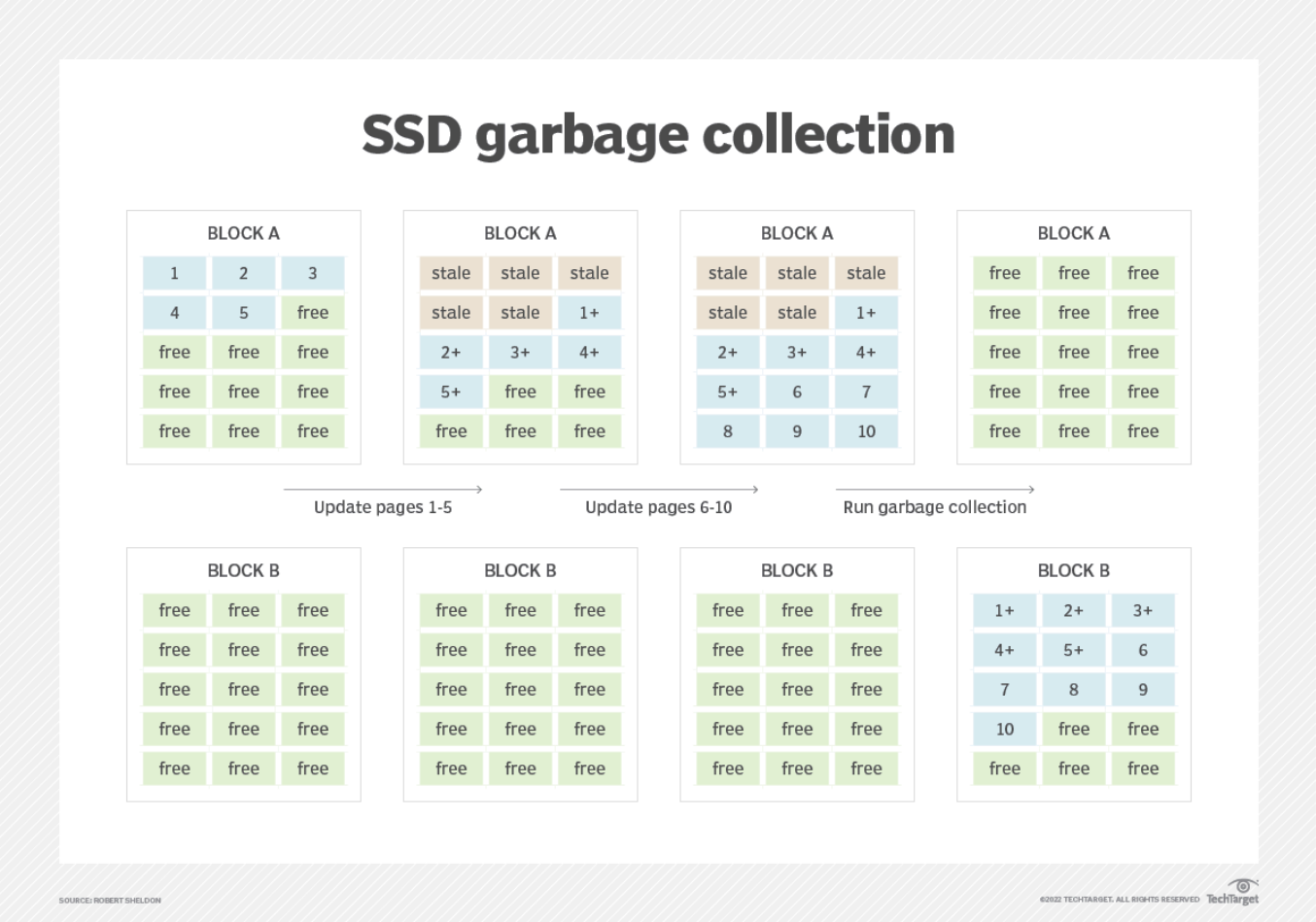
- assenza di parti meccaniche: si garantisce silenziosità e leggerezza nel trasporto;
- spiccata velocità, con transfer rate che surclassano i meccanismi elettromeccanici;
- linearizzazione dei blocchi.

L'unità SSD è in grado di gestire un certo numero di blocchi, ma ogni blocco è suddiviso in pagine. Quando un processo deve leggere e scrivere all'interno di un SSD, legge e scrive pagine. Il blocco è costituito da più pagine ed ha un ruolo nella cancellazione dei dati. La cancellazione, infatti, non viene effettuata per pagine, ma per blocco. Pertanto se bisogna sovrascrivere il contenuto della pagina, l'intero blocco va cancellato, e le altre pagine verranno riscritte. Questo chiaramente funge da limite di performance per file system come FAT: la cancellazione interpella molte volte la tabella presente nella partizione. Inoltre, le cancellazioni usurano notevolmente la SSD.

Il problema della riscrittura viene bypassato considerando che la cella viene considerata come una cella con contenuto vecchio. Verrà usata una nuova pagina dello stesso blocco o di un altro blocco (in base a dove c'è spazio) e facciamo corrispondere la mappatura del blocco nel sistema operativo al blocco nell'SSD. La cancellazione del blocco infatti usurerebbe l'intero blocco. Inoltre, se ci sono pagine libere il blocco poteva essere utilizzato riempendole. Il **controller** si occupa della mappatura tra i blocchi logici identificati dal sistema operativo e le pagine sull'SSD. Quindi la sovrascrittura di un blocco logico corrisponde ad una rilocazione di questo.

Per migliorare le prestazione dell'SSD, il controller si occupa di eseguire una **garbage collection**, recuperando la capienza del blocco al costo di una cancellazione. La cancellazione viene rimandata, ripulendo il blocco e mantenendo le informazioni. Più precisamente, quando l'SSD deve eseguire aggiornamenti sul contenuto delle pagine, il controller marca le pagine originali come pagine **stale** (ossia invalide) e scrive il contenuto su altre pagine. Contenuti aggiuntivi ricevuti vengono inseriti in altre pagine, fino a quando il blocco non viene riempito.

Quando il blocco viene riempito, i contenuti non marcati come invalidi vengono copiati in altre pagine di altri blocchi e il controller esegue la cancellazione.



TRIM

Quando parliamo di dati spazzatura o non utili parliamo comunque di dati creati dal controller stesso: per gestire la riscrittura di un blocco di un file, il controller decide di sua sponte di eseguire la scrittura in una nuova pagina. Esiste però un altro tipo di spazzatura, ossia i **blocchi cancellati nel volume logico**. Ad esempio, in un file system FAT la cancellazione porta ad un aggiornamento delle informazioni relative ai blocchi, segnando come liberi i blocchi precedentemente occupati. I blocchi che il sistema operativo ha cancellato e che sono liberi, quindi che contengono spazzatura per il sistema operativo, per il controller sono informazioni da rappresentare nel volume logico, ossia sono ancora blocchi usati. Nell'atto di ripulire, il controller **non li cancella**. Così facendo c'è un disallineamento dal punto di vista del controller e del sistema operativo, nonché un rallentamento delle prestazioni del disco.

Oggi i sistemi operativo utilizzano l'operazione **TRIM**, un'operazione che informa il controller che determinati blocchi possono essere cancellati. Il sistema operativo userà questo strumento per la cancellazione del file, finché non si scriverà qualcosa di nuovo sopra. Fintanto che il sistema operativo utilizza l'operazione TRIM, allora ogni SSD può montare **qualunque file system**. Sono stati sviluppati anche dei file system appositamente per l'utilizzo degli SSD,

come **F2FS**, ma oggi basta che il sistema operativo si adegui all'operazione TRIM. Ad esempio, Windows permette TRIM per partizioni NTFS, ma su una partizione FAT32 non viene supportata l'operazione.