

# Sistemi Operativi

XXXXXXXXXX

March 2023

# Contents

<b>1 Introduzione e richiami di architettura degli elaboratori</b>	<b>6</b>
1.1 Cos'è un sistema operativo . . . . .	6
1.2 Modalità utente e modalità kernel . . . . .	6
1.3 Richiami di architettura degli elaboratori . . . . .	7
1.3.1 CPU . . . . .	7
1.3.2 Modalità di esecuzione più in dettaglio . . . . .	8
1.3.3 Multithreading e multiprocessore . . . . .	9
1.3.4 GPU . . . . .	10
1.3.5 Memorie . . . . .	10
1.3.6 Dispositivi di I/O . . . . .	12
1.3.7 Metodi di esecuzione delle operazioni di I/O . . . . .	13
1.3.8 Bus . . . . .	14
1.4 Panoramica dei sistemi operativi . . . . .	15
1.5 Struttura dei sistemi operativi . . . . .	16
1.6 Macchine virtuali . . . . .	19
1.6.1 Come funziona una VM . . . . .	20
1.6.2 Hypervisor . . . . .	20
1.6.3 Simulazione e virtualizzazione . . . . .	21
1.6.4 Modello a container . . . . .	22
<b>2 Processi, thread, IPC e scheduling</b>	<b>23</b>
2.1 Processi . . . . .	23
2.1.1 Vita di un processo . . . . .	24
2.1.2 Stati di un processo . . . . .	25
2.1.3 Accodamento processi . . . . .	27
2.2 Thread . . . . .	27
2.2.1 Organizzazione dei thread . . . . .	28
2.2.2 Operazioni sui thread . . . . .	29
2.2.3 Thread e programmazione multicore . . . . .	29
2.3 Implementazione dei thread . . . . .	30
2.3.1 Modello 1 a molti (thread a livello utente) . . . . .	30
2.3.2 Modello 1 a 1 (thread a livello kernel) . . . . .	31
2.3.3 Modello molti a molti (ibrido) . . . . .	32
2.3.4 Classifica tipi di context-switch (da più veloce al più lento) . . . . .	32
2.3.5 Come sono implementati i thread negli OS moderni . . . . .	32
2.4 InterProcess Communication (IPC) . . . . .	33
2.4.1 Pipe . . . . .	33
2.4.2 Problemi . . . . .	33
2.4.3 Race conditions . . . . .	33
2.5 Mutua esclusione . . . . .	34
2.5.1 Condizioni per risolvere Race condition . . . . .	34

2.5.2	Un esempio di mutua esclusione	35
2.6	Implementazione della mutua esclusione	35
2.6.1	Disattivazione degli interrupt	35
2.6.2	Variabili di lock	36
2.6.3	Alternanza stretta	36
2.6.4	Soluzione di Peterson	37
2.6.5	Istruzione TSL (e XCHG)	39
2.6.6	Precisazione su variabile di lock	39
2.6.7	Sleep e wakeup	40
2.6.8	Produttore-consumatore	40
2.7	Semafori	42
2.7.1	Utilizzi dei semafori	43
2.7.2	Mutex con thread utente	44
2.7.3	Futex	45
2.8	Monitor	45
2.8.1	Produttore-consumatore con monitor	47
2.9	InterProcess Communication	47
2.10	Accesso esclusivo da parte di processi concorrenti: il problema dei 5 filosofi	48
2.10.1	Problema dei 5 filosofi con monitor	51
2.11	Accesso ad un database: problema dei lettori e scrittori	51
2.11.1	Soluzione con semaforo mutex	52
2.11.2	Soluzione 1 con monitor	53
2.11.3	Soluzione 2 con monitor	54
2.11.4	Soluzione 3 con monitor	54
2.12	Scheduling	55
2.12.1	Obiettivi di un algoritmo di scheduling	56
2.12.2	Algoritmi di scheduling per sistemi batch	57
2.12.3	Algoritmi di scheduling per sistemi interattivi	59
2.12.4	Round-Robin	60
2.12.5	Scheduling con priorità	60
2.12.6	Scheduling a code multiple	61
2.12.7	Shortest Process Next	62
2.12.8	Altri tipi di scheduling su sistemi interattivi	62
2.12.9	Scheduling dei thread	63
2.12.10	Scheduling su sistemi multiprocessore	63
2.12.11	Cosa usano i nostri OS	65

<b>3</b>	<b>Gestione della memoria</b>	<b>66</b>
3.1	Gestione dei processi in memoria centrale	66
3.1.1	Rilocazione	67
3.1.2	Spazio degli indirizzi	68
3.1.3	Swapping	69
3.2	Gestione dell'allocazione	70

3.2.1	Bitmap . . . . .	70
3.2.2	Liste . . . . .	70
3.3	Memoria virtuale . . . . .	71
3.3.1	Paginazione . . . . .	72
3.3.2	Uso reale di una tabella delle pagine . . . . .	74
3.3.3	Dettaglio su una voce della tabella delle pagine . . . . .	75
3.3.4	Tabella dei frame . . . . .	76
3.3.5	Progettazione di una tabella delle pagine . . . . .	77
3.3.6	TLB (memoria associativa) . . . . .	77
3.3.7	Effective Access Time: TLB vs Standard . . . . .	78
3.3.8	Tabella delle pagine multilivello . . . . .	79
3.3.9	Tabella delle pagine invertita . . . . .	80
3.4	Piazzamento della cache della memoria . . . . .	81
3.5	Algoritmi di sostituzione delle pagine . . . . .	82
3.5.1	Not Recently Used (NRU) . . . . .	82
3.5.2	FIFO e Seconda Chance, Clock . . . . .	83
3.5.3	Least Recently Used (LRU) . . . . .	84
3.5.4	Not Frequently Used (NFU) . . . . .	84
3.5.5	Algoritmo di Aging . . . . .	85
3.5.6	Confronto delle prestazioni . . . . .	86
3.6	Allocazione dei frame . . . . .	88
3.6.1	Allocazione locale, globale, thrashing, località . . . . .	89
3.6.2	Working set . . . . .	89
3.6.3	Page Fault Frequency . . . . .	90
3.6.4	Politica di pulitura dei frame . . . . .	91
3.6.5	Dimensione della pagina . . . . .	91
3.7	Pagine condivise . . . . .	92
3.7.1	Copy-on-write . . . . .	94
3.7.2	Zero-fill-on-demand . . . . .	95
3.7.3	Librerie condivise con linking . . . . .	95
3.7.4	Librerie condivise con file mappati . . . . .	96
3.8	Allocazione della memoria per il kernel . . . . .	97
3.8.1	Slab allocator . . . . .	97
<b>4</b>	<b>File system e dischi</b> . . . . .	<b>98</b>
4.1	Struttura di un file system . . . . .	99
4.1.1	MBR . . . . .	99
4.1.2	Partizioni . . . . .	99
4.2	Implementazione dei file . . . . .	100
4.2.1	Allocazione contigua . . . . .	100
4.2.2	Allocazione con liste linkate . . . . .	100
4.2.3	FAT . . . . .	101
4.2.4	I-node . . . . .	101

4.2.5	Extent	103
4.3	Implementazione delle directory	103
4.4	Condivisione di file su un file system	104
4.4.1	Soft-link/symbolic link	104
4.4.2	Hard-link/link fisico	105
4.5	Gestione dei blocchi liberi	106
4.5.1	Liste	106
4.6	Controlli di consistenza	106
4.6.1	Controllo di consistenza sui blocchi	106
4.6.2	Controllo di consistenza sui riferimenti agli i-node	107
4.6.3	Journaling	107
4.7	Ottimizzazione delle prestazioni del disco	108
4.7.1	Cache del disco	108
4.7.2	Scheduling del disco	109
4.7.3	RAID	112
4.8	SSD (Solid State Drive)	116

# 1 Introduzione e richiami di architettura degli elaboratori

## 1.1 Cos'è un sistema operativo

L'hardware di un moderno calcolatore, in virtù della sua elevata capacità di calcolo, è molto complesso. Sarebbe praticamente impossibile per un programmatore averci a che fare direttamente. E' necessario un software che gestisca certi aspetti dell'hardware al posto del programmatore. Cioè è necessario un software che crei delle opportune astrazioni che rendano più semplice al programmatore eseguire certe azioni.

Questo software (che alla fine non è altro che un insieme di procedure e servizi) è detto **sistema operativo**.

Possiamo vedere il sistema operativo come un "creatore di astrazioni". Infatti la maniera più efficace per far comprendere al programmatore (o meglio, al programma da lui scritto di cui verranno eseguiti i processi) gli aspetti più complessi dell'hardware.

Ad esempio, consideriamo l'astrazione **file**. Un file non è altro che un insieme di dati che devono essere memorizzati sul disco e modificati all'occorrenza. Dato che è molto complesso memorizzare fisicamente dei dati su un disco (si pensi che un manuale di un disco con interfaccia SATA conta circa 450 pagine) se ne occuperà il sistema operativo al posto del programmatore. Esso vedrà invece questi dati sottoforma di un oggetto (il file, appunto). Quindi per esso sarà facilissimo memorizzare un file nel disco (bastano un paio di comandi da terminale ).

Si può intuire che il sistema operativo sia a sua volta un software molto complesso. Il kernel (nucleo) di un moderno sistema operativo conta più di 5 milioni di righe di codice. E' per questo che la maggior parte degli OS in commercio riutilizza (con opportune modifiche) il kernel di un qualche OS precedente.

N.B. Si potrebbe pensare che un sistema operativo sia costituito anche dalla CLI o GUI con cui l'utente ci interagisce, ma non è così.

## 1.2 Modalità utente e modalità kernel

L'hardware può operare in due diverse modalità:

- **Modalità utente:** La modalità con cui opera di default. In questa modalità vengono infatti eseguiti tutti i processi utente, cioè tutti quei processi che fanno parte di programmi e applicazioni esterne al sistema operativo. In modalità utente l'OS può eseguire soltanto un set limitato di istruzioni (anche per motivi di sicurezza).
- **Modalità kernel (o supervisor):** La modalità in cui può essere eseguito tutto il set di istruzioni facente parte dell'OS. In particolare, possono essere eseguite tutte quelle istruzioni che gestiscono ad esempio, le operazioni di I/O, oppure gli interrupt.

A partire da ciò si potrebbe dire che qualunque "cosa" eseguita in modalità kernel sia parte del sistema operativo. Cioè, che il sistema operativo è costituito da tutti quei processi eseguiti in

modalità kernel. Ciò non è più completamente esatto, perché in molti casi i processi utente possono richiedere l'esecuzione di un'istruzione "vietata". Vedremo successivamente cosa succede in questi casi.

## 1.3 Richiami di architettura degli elaboratori

### 1.3.1 CPU

La CPU, in generale, esegue un'istruzione in 3 passi: fetch dell'istruzione, decodifica dell'istruzione ed esecuzione fisica dei calcoli necessari (ALU). Essa è supportata dalla presenza di piccole aree di memoria all'interno dello stesso chip: i registri. I registri sono delle memorie di capienza 1 word con un tempo di accesso molto inferiore rispetto al tempo di accesso della memoria centrale (RAM). Quindi la CPU memorizza in essi tutti gli operandi di cui necessita per eseguire l'istruzione, al posto di memorizzarli in RAM.

La maggior parte dei registri sono dei registri generici, dove è possibile salvare ogni tipo di dato. I registri più importanti sono però quelli cosiddetti **speciali**:

- **Program Counter** (PC): ha il compito di memorizzare l'indirizzo (quindi è un puntatore) della prossima istruzione da eseguire;
- **Program Status Word** (PSW): memorizza tutti i bit di stato (flag) che indicano insieme lo stato della CPU durante l'esecuzione di un'istruzione. Ad es. flag che indica l'esito di un confronto, oppure flag che indica la modalità (kernel o utente) in cui si sta lavorando. Alcuni flag sono modificabili solo in modalità kernel (esempio banale: il secondo flag menzionato).

Una menzione a parte la merita lo **Stack Pointer** (SP).

Esso è un registro speciale che contiene l'indirizzo della cima dello stack "attuale" (cioè dello stack della modalità in cui si è attualmente).

Come già noto, lo stack è in generale un'area di memoria (può essere visto anche come una struttura dati, ma in questo caso non è questa la definizione) organizzata appunto come una pila, con una cima e un fondo. I dati vengono memorizzati e prelevati in modalità LIFO (l'ultimo che viene memorizzato è il primo che viene prelevato). Lo stack viene usato per memorizzare i dati utilizzati dalle varie istanze delle procedure (funzioni/sottoprogrammi, come volete chiamarli) eseguite dalla CPU. Ogni procedura avrà un suo frame (o record) di attivazione, che viene creato al momento della CALL della procedura ed eliminato dopo il RETURN.

Adesso, bisogna considerare come è fisicamente organizzata la memoria. La memoria è divisa in diverse parti (lo stack appunto, l'heap, che sarebbe la parte "libera" della memoria, la parte che contiene i dati e la parte che contiene le istruzioni). Lo stack, per ciò che è stato detto prima, ha bisogno di cambiare continuamente le sue dimensioni fisiche. Quindi fisicamente lo stack è "capovolto" nella memoria (cioè la cima sta sotto e la base sta sopra).

La CPU può eseguire le istruzioni una alla volta, quindi aspetta che l'istruzione  $x$  sia completata prima di eseguire l'istruzione  $x + 1$ , oppure può fare in modo che se un'istruzione  $x$  è giunta ad un certo passo di esecuzione, venga eseguito il passo precedente dell'istruzione  $x + 1$  e così via.

Questo metodo viene detto **pipeline**. Ciò permette di sfruttare i tempi morti, cioè i momenti in cui qualche componente della CPU è "fermo". Esempio: se per una certa istruzione si stanno eseguendo dei calcoli (quindi lavora l'ALU) significa che nessuna istruzione sta venendo decodificata e nessuna viene fetchata. Quindi l'istruzione successiva viene decodificata, e l'istruzione ancora successiva viene fetchata.

Per migliorare ulteriormente le prestazioni, sono state create delle particolari CPU, dette "superscalari", cioè con più pipeline. Possono eseguire quindi ancora più istruzioni contemporaneamente. Il problema di queste CPU è che le istruzioni potrebbero essere eseguite "fuori ordine" (perchè le pipeline sono separate tra loro). In questo caso l'hardware si deve occupare direttamente di far rispettare l'ordine delle istruzioni. Il sistema operativo può agire limitatamente, perchè questi metodi di esecuzione non sono del tutto trasparenti ad esso. Per questo, se e quando dovrà agire, gli sarà imposta una quantità non indifferente di complessità.

### 1.3.2 Modalità di esecuzione più in dettaglio

Come anticipato prima, in alcuni (moltissimi) casi un processo utente avrà necessità di eseguire operazioni "vietate" dal sistema operativo (perchè ovviamente si è in modalità utente), per portare a termine il suo compito. Dato che non può eseguirle direttamente lui, si serve di particolari call, dette **system call** o syscall, cioè delle chiamate ad un'istruzione detta **TRAP** che si occupa di eseguire l'istruzione vietata richiesta. Quindi la TRAP fa da "tramite" tra il processo utente e il sistema operativo. Possiamo pensare che in questi casi il processo utente "chieda" al sistema operativo di fare qualcosa che lui non può direttamente fare, come se una persona andasse in un ufficio comunale a richiedere un documento. La persona non può direttamente accedere all'archivio dell'ufficio a prendere il documento, deve chiederlo all'impiegato lì presente.

Tornando a noi, ricordiamoci che ci si trova in modalità utente, quindi la TRAP deve prima cambiare la modalità da utente a kernel, poi eseguire l'istruzione kernel richiesta. Alla TRAP viene passato un parametro (argomento) che consiste nell'indice dell'istruzione kernel da eseguire in una particolare tabella propria del compilatore; l'istruzione corrispondente viene poi cercata in un'altra tabella, sta volta propria dell'OS, in cui è memorizzato anche l'indirizzo di memoria che contiene l'istruzione. Infine viene recuperata dal kernel ed eseguita. Dopo l'esecuzione, si ritorna alla modalità utente, e il processo continua ad essere eseguito normalmente.

Nota: dopo la chiamata di sistema, tutti i registri sia generici che speciali vengono salvati temporaneamente nello stack del kernel (cioè lo stack attuale quando si è in modalità kernel), in modo che l'istruzione kernel abbia a disposizione tutti i registri liberi, e che il contenuto originale di essi (che ovviamente servirà al processo utente) non venga intaccato dall'istruzione kernel.

Dopo la fine dell'esecuzione di essa, i registri vengono ripristinati.

P.S. Questo ci ricorda ciò che succedeva quando veniva chiamato un sottoprogramma in linguaggio Assembly.

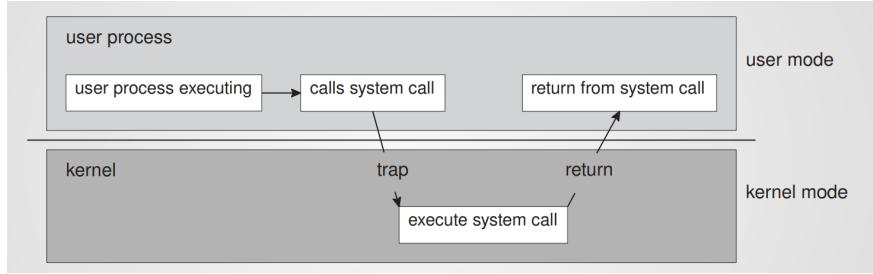


Figure 1: Switch della modalità di esecuzione.

Lo switch di modalità può essere necessario anche in caso di interrupt. Infatti le routine di gestione degli interrupt, per OVV motivi, sono eseguibili solo in modalità kernel. Quindi si eseguono gli stessi passi di prima, con una differenza: la routine di gestione di un interrupt ha la priorità su tutto (sia processi utente che chiamate di sistema), anche sugli altri interrupt; per cui la CPU disattiva temporaneamente il sistema di notifica degli interrupt, per non essere "disturbata". I registri vengono salvati nello stack attuale e poi ripristinati come nella situazione precedente. Si deve notare che per conseguenza della priorità massima delle routine di gestione degli interrupt, esse potrebbero essere eseguite in qualsiasi modalità ci si trovi. Quindi se ci si trovava già in modalità kernel, lo switch non è necessario.

### 1.3.3 Multithreading e multiprocessore

Riprendendo il discorso precedente sullo sfruttamento dei "tempi morti", si può notare, vedendo il tutto da un livello più alto, che durante certe operazioni (ad esempio il prelevamento di un dato dalla memoria RAM) l'esecuzione di un processo si ferma, perché deve aspettare che il dato sia prelevato dalla memoria. Questo è un tempo morto, perché la CPU non sta eseguendo nulla. Allora, analogamente a prima, l'idea è di eseguire un altro processo durante questo tempo morto, in modo da sfruttare sempre al massimo la CPU. Questo metodo prende il nome di **multithreading**. In realtà, non ci si riferisce a due diversi processi, ma a due diversi flussi di esecuzione (appunto, thread) di uno stesso processo. Però l'idea è quella. Quindi la CPU terrà lo stato di due diversi thread contemporaneamente. Effettuando lo switch tra i due thread abbastanza velocemente, sembrerà che essi siano eseguiti in parallelo (non è così), cioè sembrerà così al sistema operativo. Esso li vedrà come due CPU diverse, ma in realtà sono 2 CPU "virtuali" e la CPU fisica è sempre una. In pratica, il multithreading si basa sul concetto di **multiplexing**, nel tempo e nello spazio. Cioè il concetto di dover dividere una risorsa tra più utilizzatori. Il multiplexing nel tempo viene realizzato con il concetto di **time sharing**; ogni tot di tempo viene eseguito un thread diverso (cioè si effettua un **context-switch**). Come detto poco sopra, questo switch verrà effettuato così velocemente da creare un'illusione di parallelismo d'esecuzione (pseudoparallelismo).

Il multiplexing nello spazio viene implementato invece "dividendo" la memoria disponibile in piccole porzioni, ognuna riservata soltanto ad un certo thread.

Il passo successivo è realizzare dei veri e propri multiprocessori, cioè dei processori con più CPU al suo interno, su diversi chip. In questo caso si parla di parallelismo vero e proprio; i thread verranno eseguiti ognuno su una CPU diversa. In questo caso l'OS deve stare attento, perché come ricordiamo, per lui le due situazioni presentate sono esattamente identiche. Quindi

potrebbe pensare di allocare due thread sulla stessa CPU, anche se la soluzione più ovvia sarebbe di alstrarli su due CPU diverse.

I vantaggi del multiprocessore sono che si può disporre di più CPU su un calcolatore solo, con riduzione dei costi (economia di scala), più "potenza" (throughput, cioè il numero di operazioni eseguite in un lasso di tempo) e dato che le risorse (memoria) sono condivise, più affidabilità (nel caso di diversi calcolatori, le varie memorie non potrebbero essere sincronizzate). Per lo stesso motivo, i processi possono comunicare tra loro e migliorare l'efficienza. Ovviamente se si posseggono N CPU non si avrà un tempo di esecuzione esattamente N volte minore, perchè avendo la memoria condivisa inevitabilmente più processi potrebbero dover accedere agli stessi dati.

Il passo ancora successivo è disporre le varie CPU all'interno dello stesso chip, con vantaggi di dissipazione del calore e ancora di velocità. I processori di questo tipo vengono detti **multicore**. I sistemi con processori multicore possono essere di due tipi: asimmetrici o simmetrici. In quelli asimmetrici uno dei core, detto principale, controlla, gestisce il sistema, organizza e assegna il lavoro ai core secondari. Essi potrebbero avere anche un compito predefinito. Nei sistemi simmetrici, (cioè quelli più diffusi), ogni core può eseguire qualunque processo, senza subordinazione gerarchica tra loro.

### 1.3.4 GPU

Dedichiamo un piccolo paragrafo alle GPU. Una GPU è anch'essa un'unità di elaborazione, ma con compiti diversi. Si occuperà infatti di computare in maniera più efficiente possibile l'elaborazione vettoriale, permettendo di visualizzare grafica a schermo (immagini, video, rendering di modelli 3D, ad esempio videogiochi). È composta da centinaia, adesso anche migliaia di piccoli core. L'OS non ci "entra" mai, però cerca di sfruttarla.

### 1.3.5 Memorie

Premessa: la CPU non può accedere direttamente ai dati nelle memorie non volatili. Quindi l'OS deve occuparsi di trasferire i dati necessari alla CPU dalla memoria non volatile alla memoria centrale. Nei sistemi moderni, è possibile attraverso la paginazione (file di paging su Windows per intenderci) che la CPU abbia a disposizione più dati rispetto alla capienza massima della RAM. L'OS trasferirà a poco a poco i dati necessari, tenendo gli altri in attesa in una particolare area del disco. Questo ovviamente comporta una diminuzione della velocità d'accesso a questi dati.

Vediamo intanto in dettaglio com'è divisa la memoria di un calcolatore. Possiamo dire che più aumenta la capienza della memoria, più diminuisce la velocità di trasferimento dei dati.

**Registri.** La memoria più piccola (1 word) ma più veloce, essendo sullo stesso chip della CPU. L'accesso non comporta alcun ritardo.

**Cache.** La memoria cache è una memoria, più capiente dei registri e quindi meno veloce, che è comunque più veloce rispetto alla memoria centrale. Quindi la CPU sfrutta questa memoria

per salvarci dentro le istruzioni e i dati più frequentemente utilizzati. Per la precisione, la CPU ricerca un particolare dato prima nella cache: se non lo trova (cache miss) lo preleva dalla RAM e lo salva nella cache, perchè se l'ha recuperato è probabile che venga utilizzato più di una volta. La volta successiva il dato verrà ritrovato nella cache (cache hit), quindi la CPU ci avrà avuto accesso in molto meno tempo rispetto a prima (prelievo dalla RAM). La cache è suddivisa in più livelli (aumenta la capienza ma diminuisce la velocità d'accesso). Ad esempio la cache L2, che può essere condivisa da tutti i core (Intel) oppure ogni core ha la propria cache (AMD). Nel secondo caso si potrebbero avere problemi di sincronizzazione (perchè a livello logico due processi su due core diversi della stessa CPU condividono la stessa memoria, però a livello fisico accedono a due memorie diverse).

Le CPU attuali hanno anche una cache L3.

I sistemi operativi fanno frequentemente uso della cache, per memorizzare pezzi di file usati con molta frequenza, oppure per salvare indirizzi risultanti dalla conversione di lunghi percorsi di file, in modo da non ripetere la ricerca con conseguente lunga e dispendiosa conversione.

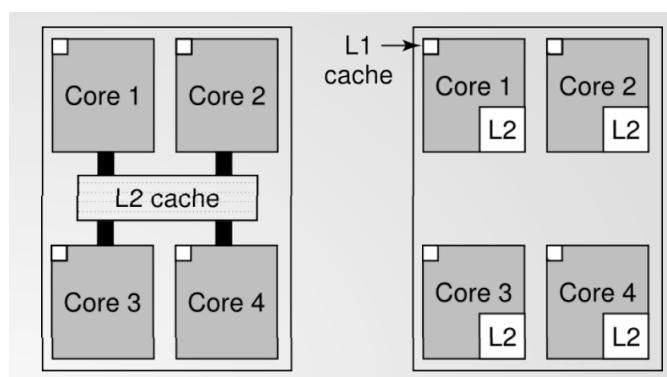


Figure 2: Cache di una moderna CPU multicore.

**Disco meccanico.** Il disco meccanico (hard-disk) è composto da un **braccio** e da dei **piatti metallici** disposti uno sopra l'altro che girano a migliaia di giri al minuto. Ogni piatto viene diviso in regioni anulari dette **tracce**. Tutte le tracce di una certa posizione del braccio (messe una sopra l'altra) compongono un **cilindro**. Ogni testina del braccio legge una certa traccia. A loro volta le tracce sono divise in **spicchi**; ognuno di loro corrisponde a delle coordinate. Il sistema operativo farà in modo che i processi vedano questi spicchi, disposti normalmente in maniera "dimensionale" (su più livelli), in modo linearizzato (altra astrazione offerta dall'OS).

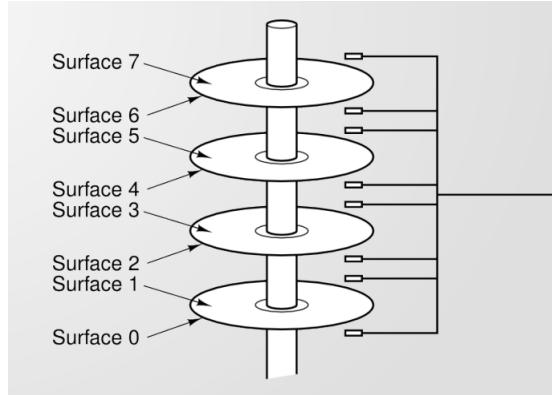


Figure 3: Composizione fisica del disco meccanico.

### 1.3.6 Dispositivi di I/O

In generale, un dispositivo di I/O possiede due componenti fondamentali:

- Un **controller**, che permette all'OS di "pilotare" il dispositivo;
- Il dispositivo vero e proprio.

Questo perchè è molto complesso addirittura per l'OS stesso pilotare il dispositivo; necessita di un'interfaccia intermedia (il controller, che è un pezzo di hardware) che gli faciliti il lavoro. Quindi avremo due interfacce, tra OS e controller, e tra controller e dispositivo (ad esempio SATA).

Dobbiamo considerare una cosa molto importante: il sistema operativo non può conoscere tutte le procedure utilizzate dal dispositivo in questione, perchè non sono metodi standard. Bisogna "mappare" (far corrispondere) queste procedure ai metodi standard dell'OS. Possiamo vederla come se ci fossero due persone che parlano due lingue diverse: necessitano di un interprete che conosca tutte e due le lingue e traduca all'occorrenza. L'interprete in questione è un software chiamato **driver**, che solitamente viene fornito insieme al dispositivo. Si capisce che il driver è l'interfaccia tra OS e controller.

Un esempio di controller potrebbe essere la **GPU**, la cosiddetta "scheda grafica" o "scheda video".

Il controller possiede dei registri (detti porte di I/O) su cui la CPU può scrivere, eseguendo delle istruzioni dette di tipo IN/OUT. Queste istruzioni devono essere eseguite in modalità kernel. Per questo è importantissimo che i driver siano stabili.

Alcuni OS, soprattutto quelli con microkernel, eseguono queste istruzioni in modalità utente. Nei sistemi moderni, invece, si sceglie un approccio ibrido: vengono "sbloccate" dall'OS solo alcune delle porte di I/O, autorizzando il processo utente ad eseguire istruzioni kernel soltanto su queste porte. Le istruzioni in questione quindi non sono più privilegiate.

Un altro metodo è quello di riservare una parte della memoria al dispositivo (cioè un determinato numero di indirizzi, detto **spazio degli indirizzi**), ed eseguire le istruzioni standard di lettura e scrittura dell'OS. Questo metodo è detto **mappatura in memoria**.

### 1.3.7 Metodi di esecuzione delle operazioni di I/O

Le operazioni di lettura e scrittura su disco sono molto lente rispetto alle possibilità della CPU. Per questo si dicono **operazioni bloccanti**. Quando un processo effettua una chiamata di sistema, esso viene messo in standby dall'OS, e fa eseguire alla CPU altri processi nel frattempo. Nel caso di operazioni di I/O ci sono 3 metodi diversi per eseguirle:

- **Busy waiting:** il processo effettua la syscall; l'OS si occuperà utilizzando il driver di pilotare il controller e scrivere o leggere sul disco. Nel frattempo, la CPU leggerà continuamente la porta di I/O coinvolta (polling), finché non legge l'esito dell'operazione. Se l'operazione è andata a buon fine, i dati letti sono prima memorizzati nel buffer interno del controller, poi memorizzati in RAM. Il problema del busy waiting è che la CPU sarà continuamente sotto sforzo per leggere la porta, ciò comporta uno spreco di risorse della CPU, che potrebbero essere utilizzate per altre operazioni.  
Nota: idealmente la CPU dovrebbe concentrare il 100% delle sue risorse sui processi utente. Se deve eseguire delle istruzioni kernel, le risorse impiegate sono classificate come spreco.
- **Utilizzo degli interrupt:** il driver, attraverso uno speciale controller detto interrupt controller, dopo l'esecuzione dell'operazione di I/O, notifica un interrupt alla CPU: essa saprà quindi che l'operazione è terminata senza dover verificare continuamente come nel busy waiting. L'esito dell'operazione sarà gestito dalla routine associata all'interrupt (quindi verifica dell'esito positivo o negativo, invio dei dati in memoria o viceversa, ecc.). In pratica, è la CPU che dovrà occuparsi di trasferire i dati letti dal buffer alla RAM. Dopo la fine dell'interrupt, il processo riprende la sua esecuzione normalmente.

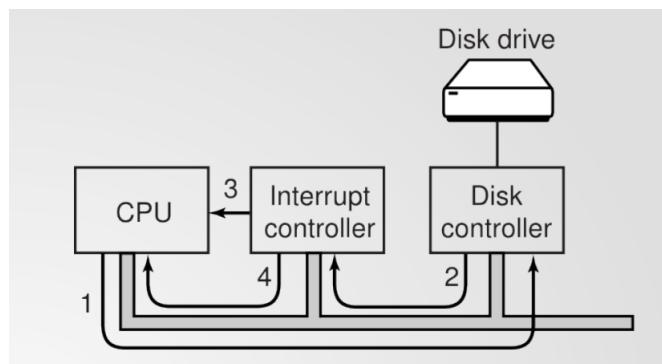


Figure 4: Passi di gestione di operazioni I/O con interrupt.

- **DMA:** Il metodo più moderno è quello che sfrutta un particolare hardware detto chip DMA (Direct Memory Access). Questo chip ha la facoltà di poter accedere direttamente alla RAM, senza passare dalla CPU. Nella fase "istruttoria" della richiesta (syscall) viene specificato in quale locazione memorizzare i dati. Il chip DMA esegue l'operazione e chiama l'interrupt. La CPU sarà comunque chiamata in causa (deve gestire l'interrupt), ma impiegherà molto meno tempo a farlo, perché i dati sono stati già trasferiti in memoria centrale dal DMA.

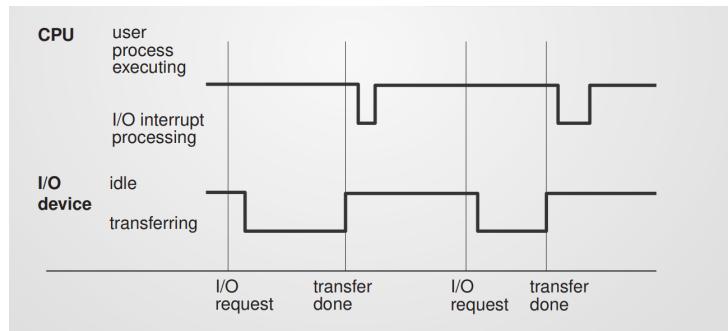


Figure 5: Passi di gestione di operazioni I/O con DMA.

N.B. Tutto ciò qui è stato presentato nel caso di lettura. E' tutto uguale, ma speculare, nel caso della scrittura.

### 1.3.8 Bus

Nei primi calcolatori era sufficiente un solo bus per far comunicare la CPU con tutti gli altri componenti. Ma l'aumento della complessità dell'architettura interna dei calcolatori successivi ha fatto sì che fossero necessari più bus, ognuno con una specifica funzione e specializzati per particolari operazioni, con velocità differenti. La maggior parte dei bus sono standard. Ad esempio, il bus principale è il **PCIe** (Peripheral Component Interconnect Express), che è un moderno bus (arrivato alla versione 5.0) ad alta velocità, utilizzato soprattutto nel caso di periferiche esterne. Un altro bus è il **DMI** (Direct Media Interface) sempre ad alta velocità, ma che connette la CPU al controller delle porte I/O, quindi a tutti i dispositivi di I/O.

I bus possono essere di due tipi: paralleli o seriali. I bus paralleli trasferiscono tutti i bit della word parallelamente, cioè ogni bit su una linea diversa. I bus seriali invece trasferiscono i bit della word "uno dietro l'altro". Su ogni linea ci sarà un bit di una diversa word. E' ovvio dire che alla fine, si evince che il throughput è all'incirca uguale, a parità di tecnologia utilizzata.

Il PCIe è un bus seriale, ma funziona in un modo un po' particolare: effettua un multiplexing nel tempo tra i vari utilizzatori. Cioè alloca un certo numero di linee ad una certa richiesta. Il numero di linee allocate dipende dalla priorità della richiesta (processo). Le linee allocate potranno essere utilizzate solo da quel processo. Per questo motivo il controller del PCIe è abbastanza complesso: deve "miscelare" le varie richieste e spartire in modo conveniente le linee.

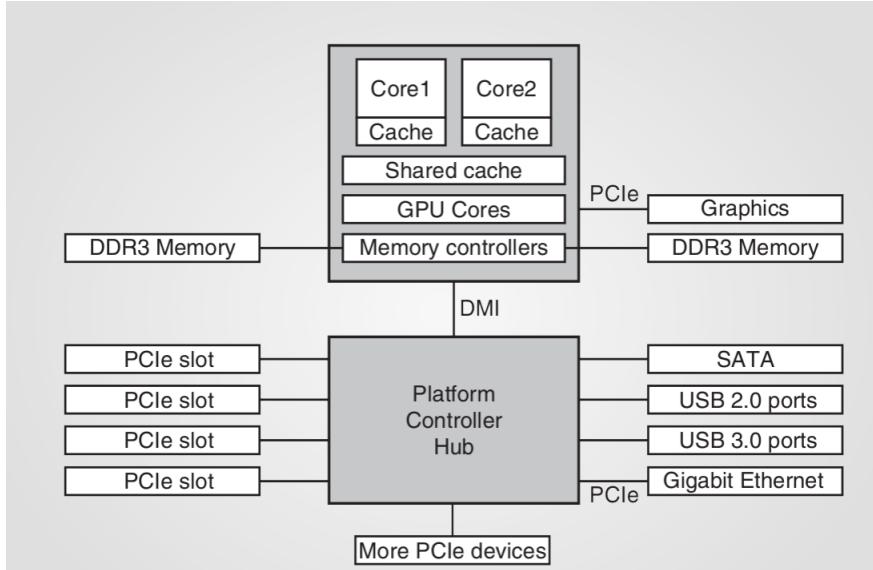


Figure 6: Schema dei bus di un moderno calcolatore.

## 1.4 Panoramica dei sistemi operativi

Esistono diversi sistemi operativi, creati per soddisfare le esigenze di diversi tipi di calcolatori. Alcuni di essi, pensati per un certo "scenario", si potranno adattare ad altri, ma non funzioneranno al massimo della loro possibilità. Possiamo dividere le varie classi di sistemi operativi così:

- Sistemi operativi per mainframe/server: I mainframe e i server, essendo dei grandi computer composti da centinaia di dischi e decine di CPU, necessitano di gestire grandi moli di dati e di processi attivi. Quindi un OS per mainframe e server deve poter gestire più CPU fisiche in parallelo, ed un alto carico di richieste di I/O, provenienti da utenti remoti. Nel caso dei server, essi devono sempre gestire richieste provenienti dall'esterno, fornire loro ciò che chiedono e rimanere sempre attivi. Quindi anche un OS per server deve supportare la multiprogrammazione e deve essere molto stabile, quasi privo di bug.
- Sistemi operativi per PC: simili agli OS per mainframe, devono anch'essi riuscire a gestire multiprogrammazione, ma con carichi molto più bassi. In questo caso è molto importante la multiutenza. E' soprattutto necessario (anche per i server e i mainframe) garantire la protezione (modalità utente/kernel). Dato che i PC saranno utilizzati da utenti comuni molto inesperti, è necessario facilitargliene l'uso. Introdotte le GUI per questo.
- Sistemi operativi mobile: esigenze simili a quelli per PC, ma non necessaria multiutenza. I permessi e le risorse vengono gestiti in modo diverso (es. in Android i permessi vanno concessi ad ogni app singolarmente al momento dell'installazione). Le GUI sono pensate per un tipo diverso di input (touch screen) e i processi sono completamente isolati tra di loro. L'OS (o la distribuzione specifica per quel modello) è "cucita su misura", quindi il passaggio ad un altro OS o ad un'altra distribuzione è molto difficile (ad es. rom custom Android fanno perdere alcune feature a certi modelli di smartphone rispetto alle rom personalizzate dal produttore).

- Sistemi operativi per sistemi embedded (con uno signolo specifico compito): Il kernel è spesso condiviso con OS mobile e desktop (spesso Linux). Si usa ancora il modello a processi, con una fondamentale differenza: i processi che girano sono pochi, e già noti agli sviluppatori dell'OS in questione. L'utente non può eseguirne altri. Quindi non necessitano l'isolamento: non "parleranno" tra di loro.
- Sistemi operativi real-time: Utilizzati spesso in sistemi industriali. Modello a processi, e natura di essi molto simile a quelli che girano sui sistemi embedded. Una differenza importantissima: le azioni devono essere eseguite tempestivamente. Se un processo viene eseguito in ritardo, ci potrebbero essere problemi gravi (pensate alla catena di montaggio di un missile balistico). Per garantire questa tempestività, pochi processi, OS progettato insieme all'hardware e soprattutto ogni processo può utilizzare la CPU al massimo delle possibilità, con un limite di tempo prefissato. Quindi si perde multiprogrammazione. Cioè: non sono degli OS preemptive (rilasciano un processo per eseguirne un altro, operando un context-switch. Tipico dei sistemi desktop e server).

## 1.5 Struttura dei sistemi operativi

Il kernel di un sistema operativo può essere progettato strutturandolo in diversi modi. La maggior parte degli OS moderni sono "ibridi", cioè le loro parti sono strutturate in modo diverso tra loro, sfruttando diverse delle strutture sotto presentate.

- **Struttura monolitica:** usata dai primi OS. Nei sistemi operativi con struttura monolitica, il codice di tutte le procedure e i servizi è raccolto in un unico blocco di codice. Quindi il kernel è contenuto tutto in un solo programma binario eseguibile. Ogni componente può "vedere", richiamare gli altri e ogni processo può accedere a tutto. Scheduling e protezione introdotti successivamente (UNIX). E' molto complesso gestire un unico blocco non organizzato. Negli ultimi sistemi monolitici però si può vedere un'organizzazione di base. Cioè i processi utente possono richiedere procedure di sistema con una chiamata TRAP, e i processi di sistema vengono eseguiti, insieme a dei processi detti di supporto. E' possibile dividere la struttura monolitica in tre strati su questa base.
- **Struttura a livelli:** Il problema dei sistemi monolitici è la poca organizzazione. Si è pensato quindi di organizzare il sempre unico kernel in una gerarchia a livelli, attraverso delle astrazioni. Cioè ogni livello (o strato) avrà il compito di implementare una particolare funzione dell'OS. Esempio: un livello riservato all'implementazione delle CPU virtuali. L'idea che sta alla base dei livelli è quella che ogni livello debba fornire dei servizi e procedure ai livelli superiori attraverso un'interfaccia software (ad esempio dei metodi, che però NON SONO SYSCALL). L'insieme di tutti i livelli compone le chiamate di sistema. Salendo di livello, cresce il livello di astrazione. Possiamo notare che questa struttura ha un concetto di base simile a quello che caratterizza la OOP.  
Solitamente ogni blocco è circoscritto ad un solo specifico compito. Ciò facilita il lavoro di implementazione e di testing da parte del programmatore. Anche il debug viene facilitato, perché sarà limitato solo allo strato affetto dal bug. Le procedure di un livello non

possono interagire con quelle degli altri livelli, diversamente dai sistemi monolitici. I problemi principali della struttura a strati sono la decisione, in sede di progettazione, dell'ordine dei vari livelli. Bisogna ricordare infatti, che in questa struttura il livello superiore dipende direttamente soltanto dal livello inferiore. Un ordine "sbagliato" potrebbe obbligare i programmatore di alcuni livelli ad uno sforzo maggiore, mancando di alcune astrazioni presenti solo nei livelli superiori. Un altro problema è quello delle chiamate nidificate: ad esempio: una procedura di un certo livello chiama due procedure del livello inferiore, che chiamano ognuna a loro volta altre 3 procedure del livello ancora inferiore, e così via. Ci si espone quindi a problemi di overhead, non presenti nei sistemi monolitici (a causa del fatto che una procedura di un livello può sfruttare solo quelle del livello immediatamente inferiore)

- **Struttura a livelli con anelli concentrici:** questa variante è sfruttata dal sistema operativo MULTICS. I livelli sono fisicamente separati tra loro, non solo a livello di vincoli dati al programmatore, grazie a hardware ad hoc. Questo hardware fa sì che ogni livello giri a livelli di protezione diversi: più si sale di livello, meno processi sono disponibili perché bloccati. Se un processo di un livello deve eseguire un'azione di cui si occupa una procedura di livello inferiore, si effettua una chiamata alla TRAP. Si ha più robustezza a livello di codice, ma il problema overhead già presente viene amplificato di molto. Quindi il MULTICS non ha avuto il successo sperato.

- **Microkernel:** l'idea è quella di creare un OS con tutti i componenti necessari, raggrupparne alcuni e fare di loro il kernel. I componenti messi all'interno del kernel saranno quelli "indispensabili" per il funzionamento dell'OS, ad esempio i componenti che gestiscono lo scheduling dei processi, gestione della memoria, inter-process-communication (IPC) e gestione degli interrupt. Questi saranno gli unici componenti che gireranno in modalità kernel. Gli altri gireranno in modalità utente (sottoforma di processi utente, gestiti dal microkernel), anche se effettivamente sono considerate parte dell'OS. Questo fa in modo che il kernel sia "piccolo", di dimensioni ridotte a livello di righe di codice. Di conseguenza sarà scritto meglio, ciò riduce la probabilità che si presentino bug gravi. I componenti fuori dal kernel sono isolati tra di loro, e molto piccoli come dimensione, per lo stesso motivo di prima. I componenti esterni, come ad esempio i driver, vengono isolati anch'essi, collocandoli in un processo utente. Per cui, se un driver è instabile, non potrà creare problemi agli altri processi e soprattutto al microkernel.

In questa struttura i processi comunicano tra loro, e con i processi del microkernel attraverso dei particolari messaggi, cioè dei pacchetti (blocchi) di dati. Da non confondere con i pacchetti di rete. Il microkernel si occuperà di recapitare questi messaggi.

I componenti non kernel sono ad esempio il file system, gestione dispositivi I/O, gestione dischi. La divisione in componenti comporta anche una gestione da parte del microkernel più dinamica. C'è infatti un processo del microkernel (**reincarnation**) che controlla lo stato degli altri processi. Se ci dovessero essere problemi in qualche processo, viene killato e ricreato. In pratica, questo sistema si "autoripara". Tra i vari componenti si può individuare una sorta di gerarchia.

Due esempi di OS a microkernel possono essere Windows NT e Mac OS X. Anche se precisamente Mac OS X ha una struttura ibrida tra microkernel e struttura a moduli, che vedremo tra poco.

Per esaminare i problemi principali dei sistemi con struttura a microkernel pura prendiamo come esempio proprio Windows NT. Era stato concepito per i server, ma trasportato anche su macchine desktop. La struttura a microkernel creava problemi perché la presenza massiccia di processi utente (maggiori in numero rispetto ai sistemi monolitici) portava a creare un overhead intrinseco. Questo anche a causa del sistema a messaggi, ovviamente più lento rispetto alle normali syscall. In un sistema monolitico un processo può comunicare con un processo kernel attraverso una sola syscall, il sistema a messaggi fa sì che per gestire il messaggio ci vogliono più syscall; un messaggio inviato viene notificato in maniera simile ad un interrupt, anche questo contribuisce all'overhead. Esempio: in caso di dispositivi I/O che non sfruttano la mappatura in memoria, i processi utente devono comunicare forzatamente tramite i messaggi.

Le versioni successive di Windows hanno parzialmente risolto questo problema integrando più componenti dentro il kernel. Windows 11 non si può quindi definire un sistema completamente a microkernel, perché il kernel come dimensioni è aumentato di molto. Si può definire più come un ibrido tra microkernel e monolitico.

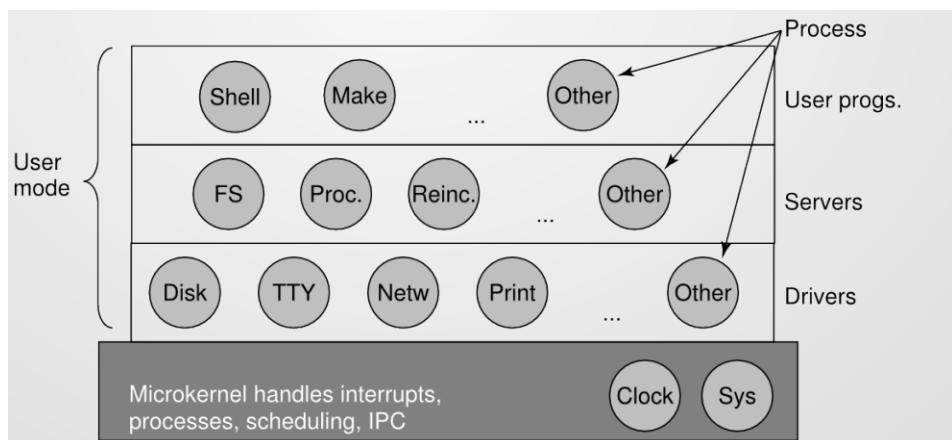


Figure 7: Organizzazione di un OS a microkernel (MINIX 3).

- **Struttura a moduli:** il concetto è sempre quello che sta alla base della struttura a microkernel, cioè kernel con dimensioni ridotte e componenti separate tra loro, dette **moduli** in questo caso. La differenza è che questi moduli vengono eseguiti tutti insieme, in modalità kernel. Nessuna componente dell'OS gira quindi in modalità utente. In pratica, è l'applicazione della OOP al kernel. Ogni modulo implementa un aspetto specifico, ad es. per ogni file system supportato ci sarà un modulo. Tutti i moduli hanno un'interfaccia comune tra loro. Altro esempio: Linux può gestire lo scheduling dei processi (ordine di esecuzione dei processi) applicando diversi algoritmi; per ogni algoritmo di scheduling ci sarà un differente modulo che lo applica.

Ogni modulo gira in modalità kernel come detto prima, per cui si perde parzialmente uno dei vantaggi della struttura a microkernel: cioè la stabilità data dal fatto che i "componenti" fossero in modalità utente. In modalità kernel, si ha più libertà d'azione quindi più

probabilità di causare danni agli altri processi. Da ciò deriva anche il principale vantaggio: si abbandona il sistema di comunicazione a messaggi, tornando alle classiche syscall, aumentando l'efficienza. Ogni modulo può anche essere sostituito all'occorrenza.

Infatti in fase di compilazione del kernel si sceglie cosa includere nel kernel e cosa lasciare sottoforma di moduli. In pratica, alcuni moduli (I/O) non vengono caricati direttamente all'avvio, ma soltanto in caso di necessità di usare dispositivi di I/O, ad avvio già completato. Il modulo sarà caricato nel kernel e diventerà parte di esso,

La differenza principale tra le varie distribuzioni di Linux è la composizione del kernel. Cioè quali moduli sono "dentro" il kernel e quali stanno "fuori". Altri moduli esterni vengono scritti dalla community. Ciò paradossalmente garantisce la loro stabilità, perché essendo open source, vengono tenuti d'occhio da tutti i membri della community, e ogni bug viene corretto tempestivamente appena scoperto.

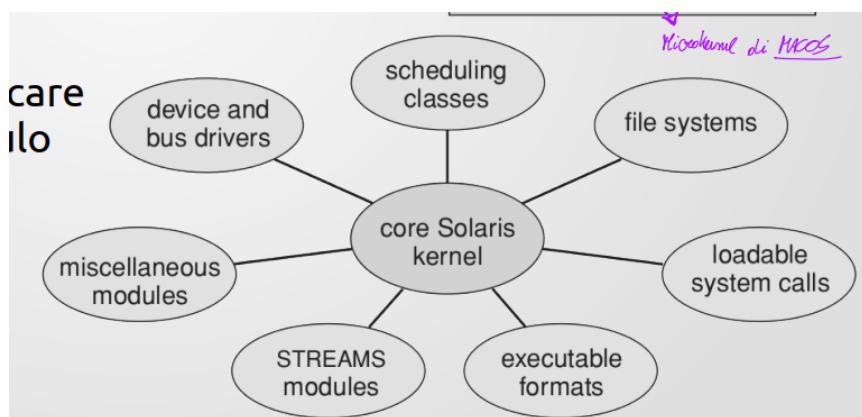


Figure 8: Organizzazione di un OS a moduli (Solaris).

## 1.6 Macchine virtuali

Abbiamo già visto cosa si intende per astrazione, e come le astrazioni facciano "creare" qualcosa di non esistente sfruttando cose già esistenti. Ad esempio, la CPU virtuale. La CPU virtuale viene implementata attraverso il time-sharing, facendo in modo di avere due thread diversi. Per l'OS ci saranno due CPU, ma in realtà ce n'è una sola fisica.

La macchina virtuale estremizza questo concetto, ed astrae un'intera macchina, non solo un particolare componente o aspetto. Si avrà quindi un hardware "virtuale", sfruttando l'hardware fisico già presente, e un software virtuale, cioè un OS virtuale su cui girano processi virtuali. In pratica, i processi virtuali saranno eseguiti in realtà dalla CPU fisica, i dati saranno contenuti nella memoria fisica, ma agli occhi dell'OS virtuale sarà come se ci fossero un'altra CPU, un'altra memoria ecc.

I vantaggi principali sono quelli di poter far girare diversi OS sulla stessa macchina fisica, con un'efficienza paragonabile alle macchine fisiche (tranne I/O, come vedremo poi). Infatti VM molto utili per sviluppo app multipiattaforma, e sviluppo siti web. In questi casi è necessario poter testare il software su ambienti diversi. L'OS virtuale non è consci di esserlo, quindi non è necessario supporto ad hoc. Anche a livello aziendale, l'utilizzo di VM permette di poter continuare l'utilizzo di software obsoleto ma indispensabile per l'attività, potendo comunque

dismettere la vecchia macchina e sostituirla con una più moderna e performante. Il vecchio software girerà su una VM con le stesse caratteristiche della vecchia macchina. Addirittura alcune VM possono essere spostate tra macchine fisiche senza interromperne l'esecuzione. Altri vantaggi da non sottovalutare sono quelli a livello di sicurezza: se un servizio su una VM viene attaccato dall'esterno, il processo malevolo potrà accedere solo ai processi/servizi presenti sulla VM. Non può accedere al resto della macchina fisica. In molte aziende è prassi creare diverse VM ad hoc per farci girare singoli servizi in maniera separata. Se ne viene attaccato uno, gli altri sono "salvi". Questa idea veniva una volta applicata facendo uso di diverse macchine fisiche con reti DMZ (demilitarized zone). Da questo si può notare anche un vantaggio in termini di economia di scala.

### 1.6.1 Come funziona una VM

La VM viene implementata facendo sì che il codice eseguito sulla VM (i processi) venga eseguito sulla CPU fisica. Ciò spiega la buona efficienza raggiunta da esse. Il processo viene eseguito normalmente finché non si rende necessario effettuare una syscall. A quel punto il controllo passa ad un particolare software detto **Hypervisor**, che si occupa della gestione della VM. L'Hypervisor serve a far eseguire la TRAP in modalità kernel virtuale, cioè simula la modalità kernel sull'OS virtuale. Questo perché il processo associato alla VM, sull'OS "reale", è un processo utente. Dato che in realtà qualunque processo virtuale, sia kernel che utente, viene eseguito in modalità utente "reale", c'è un massiccio spreco di risorse, anche a causa del lavoro che svolge l'Hypervisor. Ciò comporta i problemi di efficienza dell'I/O accennati prima, causati anche dal fatto che la VM non può dialogare direttamente con il dispositivo fisico.

### 1.6.2 Hypervisor

Il software Hypervisor può essere di due tipi:

- Tipo 1: il più "vecchio". Gira direttamente sull'hardware fisico, e si occupa di istanziare la VM. In questo caso esso funge da OS per la VM. Per cui non c'è un OS "ospite". Cioè, non gira un OS virtuale, perché le sue funzioni sono svolte dall'hypervisor. Questo tipo di hypervisor viene utilizzato quando si deve virtualizzare un servizio specifico. Usato a livello professionale (aziende ecc.);

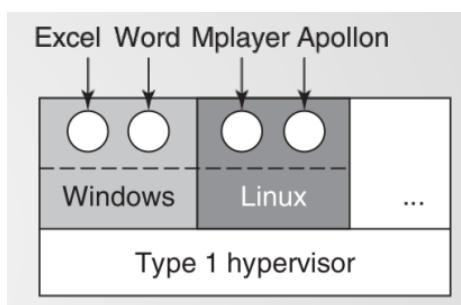


Figure 9: Hypervisor di tipo 1.

- Tipo 2: quello più diffuso al giorno d'oggi, soprattutto tra gli utenti (es. VirtualBox). È un qualunque processo utente che gira sull'OS principale. Cioè, un programma che gira sulla macchina fisica. Permette di istanziare VM su cui girano OS virtuali (non è più lui che fa da OS). Effettua le varie operazioni (es. scrittura su disco) appoggiandosi all'OS principale. Quindi in alcuni casi avrà bisogno di effettuare syscall.

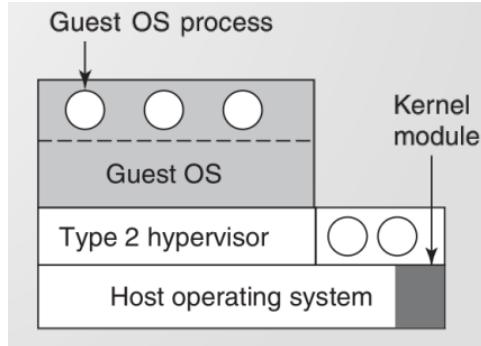


Figure 10: Hypervisor di tipo 2.

Un esempio di Hypervisor può essere KVM su Linux. Viene considerato un Hypervisor di tipo 1 perchè contenuto nel kernel, ma dato che ogni VM gira come processo utente su Linux, viene considerato da altri un hypervisor di tipo 2.

### 1.6.3 Simulazione e virtualizzazione

**Simulazione** (o emulazione) è una tecnica che permette di eseguire una VM avente architettura hardware anche diversa dalla macchina fisica su cui è eseguita. Quindi la CPU e la memoria della VM vengono "simulate". Cioè: le istruzioni vengono "interpretate" in modo da essere comprese da quell'architettura e ne viene simulato il loro effetto su quell'architettura. Ciò comporta un grande dispendio di risorse, per questo l'emulazione richiede una macchina fisica molto più potente rispetto alla macchina che si sta emulando. Esempio: gli emulatori di console per videogiochi. Una Nintendo Switch ha una CPU con architettura ARM e 4 GB di RAM. Ma per far girare in maniera soddisfacente il suo software su un emulatore, necessaria una macchina con CPU da almeno 8 core, e una memoria molto maggiore (16 GB di RAM), oltre ad una GPU con molti più core di quella inclusa nella console. Questo perchè bisogna "tradurre" le istruzioni e simularne l'effetto su una macchina con architettura ARM (emulatore gira su macchina con architettura x86).

**Virtualizzazione** è tecnica che permette di eseguire una VM avente la stessa architettura del sistema fisico, eseguendo i processi utente virtuali direttamente sulla CPU fisica, senza bisogno di interpretazioni varie. La paravirtualizzazione è virtualizzazione, ma con OS virtuale "conscio" di esserlo. Ciò porta ai due OS, ospite e principale, a comunicare tra loro, aumentando l'efficienza. Es. XEN (crea istanze di Linux con kernel modificato).

#### 1.6.4 Modello a container

Negli ultimi tempi il modello a VM sta venendo lentamente sostituito dal modello a **container** (ad esempio Docker). Un problema delle VM è che creano overhead sulla macchina ospitante. Per risolverlo, si decide di inserire i servizi che girano sulla VM, in un altro ambiente isolato, simile ad una VM, ma detto container. Hanno una differenza sostanziale: i servizi nel container si appoggiano direttamente sull'OS principale. Non presente OS virtuale. Ciò è il motivo principale per cui si riduce overhead. La CPU non si deve occupare di eseguire i processi dell'OS virtuale, si deve occupare soltanto di eseguire quei servizi.

## 2 Processi, thread, IPC e scheduling

### 2.1 Processi

Definizione di processo: è un'istanza di esecuzione di un programma. In pratica, un programma può essere eseguito più volte. Ogni istanza è un processo diverso di quel programma (In parole povere, cliccando 3 volte la calcolatrice, si apriranno 3 calcolatrici diverse, cioè 3 finestre. Ognuna di esse è un processo).

Ad un processo è associato uno spazio degli indirizzi, cioè una parte della memoria centrale riservata solo a lui, a cui solo lui può accedere. Questo spazio degli indirizzi è diviso in diverse parti (l'avevamo già visto): il codice del programma, i dati utilizzati dal processo, l'heap e lo stack. Lo spazio di memoria è lineare. Dal punto di vista del processo, si parte da una locazione 0 fino ad arrivare ad una locazione massima, dipende dallo spazio riservatogli.

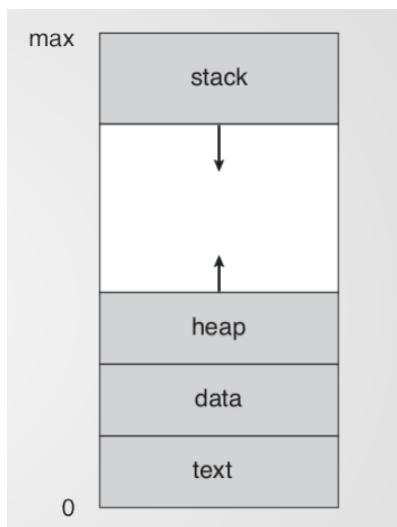


Figure 11: Spazio degli indirizzi di un processo.

Lo stato dei registri della CPU durante l'esecuzione del processo è anche associato ad esso. Infatti, ad ogni context-switch dovrà essere memorizzato, per poterlo poi ripristinare ad un successivo context-switch, quando sarà rieseguito lui.

Un processo probabilmente avrà bisogno di accedere a dei file. I file aperti dal processo saranno associati ad esso.

Ogni processo (tranne quello "genitore", creato dall'OS e che crea tutti gli altri; su Linux è INIT) è imparentato con gli altri. Questo perchè un processo viene creato a seguito di una richiesta effettuata da un altro processo. Il processo creato sarà figlio del processo che ha richiesto la sua creazione. I processi possono quindi essere visualizzati come un albero n-ario che mostra le parentele tra loro.

Per salvare tutti gli elementi associati ad ogni processo (quelli descritti sopra) l'OS sfrutta una tabella, detta **tabella dei processi**. Il singolo record di questa tabella viene chiamato **PCB** (Process Control Block), ed è associato ad un solo processo. Il singolo PCB contiene quindi tutti gli elementi associati al processo, e ne indica anche il "padre". Gli indici della tabella identificano il PCB, e di conseguenza anche il processo. L'indice viene detto **ID del processo**.

Quando un processo muore, il PCB si svuota, e può essere assegnato ad un nuovo processo. La tabella è interna al kernel, e memorizzata in RAM.

N.B. In alcuni casi, tra i vari processi del codice può essere condiviso.

Il modello dei processi si basa sull'esecuzione sequenziale o in pseudoparallelismo di essi. Cioè nel primo caso vengono eseguiti uno dopo l'altro. Il pseudoparallelismo è comunque esecuzione sequenziale in realtà, ma grazie ad astrazione CPU virtuale, context-switch e time-sharing all'OS sembrerà che l'esecuzione sia parallela. In ogni caso, ogni CPU è dedicata ad un singolo processo. Se ci sono più CPU (core), c'è vero parallelismo.

### 2.1.1 Vita di un processo

Il processo INIT viene creato durante l'inizializzazione dell'OS, da lui stesso. In generale, un qualunque processo viene creato dopo una syscall da parte di un altro processo. Ci sono due modi per farlo, uno sfruttato da UNIX, l'altro da Windows:

- **Fork** (UNIX): una normalissima syscall. Un processo P1 chiama la fork, e OS crea un clone di P1, detto P2, con un ID diverso (è un altro processo a tutti gli effetti), ma il suo codice, il suo stack ecc. sono delle copie identiche di quelli di P1. Dopo si effettua un'altra syscall, detta exec(). Dato che il codice dei due processi è uguale, necessario qualcosa che distingua il contesto (padre o figlio) in cui ci si trova. Il codice del programma può quindi essere scritto in modo che distingua i casi in cui è il processo padre, e i casi in cui è il processo figlio. Banalmente, basta un if. Da quel momento, i due processi eseguiranno codice diverso e non avranno alcun legame tra loro. Potranno comunicare soltanto tramite IPC (inter-process-communication). La primitiva exec(), chiamata da P2, gli azzerà lo stato (lo "pulisce") e verrà caricato lo stato passato come parametro all'exec. In pratica, uso il processo clone per farci altro, dopo averlo azzerato. A prima vista questo approccio può sembrare dispendioso, e alcuni azioni "inutili". Ma si vedrà che ciò viene eseguito istantaneamente e che in effetti conviene agire così.
- **CreateProcess** (Windows): Windows effettua chiamata CreateProcess che crea direttamente un nuovo processo da 0, senza clonarlo da altri, specificando quale codice dovrà eseguire e gli altri dati necessari. A prima vista sembra più efficiente e rapido rispetto alla fork, ma successivamente si vedrà che con la fork si avrà più "spazio di manovra".

Un processo può terminare per diverse cause, riassumiamo le principali:

- uscita normale/ordinaria (volontaria): nel codice del programma è presente una istruzione exit(). A quel punto le risorse utilizzate dal processo vengono rilasciate e il suo PCB viene svuotato, rendendolo disponibile ad un futuro processo. Cioè è il processo stesso che decide di terminare;
- uscita su errore (volontario): se si verifica un'anomalia prevista dal codice, il processo chiama sempre la exit() ma stavolta visualizza un messaggio d'errore contenente un codice. I codici che indicano un errore sono solitamente diversi da 0, mentre il codice 0 ci riporta al primo caso, cioè terminazione senza errori. Esempio, return 0 in C/ C++.

Se non si fosse capito, non è un errore esterno al codice, ma un errore rispetto a quella che è la logica del programma, quindi previsto dal programmatore (costrutto try-catch ad esempio);

- Errore critico (involontario): qui si tratta di un errore più grave, esterno al processo, che lo fa terminare contro la sua volontà. Ad esempio, istruzione non esistente nell'ISA dell'architettura in uso, oppure divisione per zero come anche un tentativo di accesso a locazioni senza autorizzazione (locazioni di un altro processo). La CPU non può fare nulla in questi casi, quindi fa terminare il processo. E' tutto fatto dall'hardware, non vengono effettuate syscall. Potrebbe essere ritornato un feedback (messaggio di errore).
- Terminato a causa della richiesta esterna di un altro processo (involontario): un processo P2 termina perchè un altro processo P1 ne ha fatto richiesta. P1 invia un segnale (vedremo poi cosa sono) a P2 per farlo terminare. P2 può accettare o rifiutare (anche a causa della logica del codice, per esempio il classico controllo per vedere se un file è ancora aperto prima della fine del programma) la richiesta. E' permesso ciò solo in alcuni casi, ad esempio quando P1 e P2 sono processi appartenenti allo stesso utente. Se P2 accetta, viene eseguita una routine che permette di terminarlo in modo pulito. La syscall che deve effettuare P1 si chiama **kill()**, in UNIX. La kill() non ha soltanto questo compito. N.B. non conta di solito la parentela tra i due processi.

### 2.1.2 Stati di un processo

Un processo può assumere 3 stati principali: **ready**, **blocked** e **running**. Ci sono altri due stati (new e terminated) che rappresentano rispettivamente creazione e terminazione del processo. Lo stato **ready** (pronto), come dice il nome, indica che il processo è pronto per poter essere eseguito dalla CPU. L'OS quando deve scegliere un nuovo processo a cui dedicare la CPU deve "pescare" tra i processi pronti. Tutti questi processi (o meglio, i loro PCB) sono memorizzati in una coda, denominata: **coda dei processi pronti**. Questa coda (con priorità, perchè vedremo che ogni processo avrà una priorità diversa) è dinamica, perchè le transizioni (cambiamenti di stato) sono molto frequenti. E' implementata tramite una lista doppiamente linkata, per velocizzare le operazioni. Anche la tabella dei processi è dinamica.

Lo scheduler pesca dalla coda il processo che avrà l'onore di usare la CPU. Quindi il processo cambia stato, da ready a **running** (in esecuzione). Il **dispatcher** salva lo stato del processo precedentemente in esecuzione e predisponde il tutto per l'esecuzione dell'attuale processo. Se si ha una sola CPU fisica o virtuale che sia, ci sarà un unico processo in stato running. Se n CPU, n processi in stato running. L'informazione sullo stato di un processo è contenuta nel suo PCB. Un processo in stato **blocked** (bloccato), non può usare la CPU anche se essa è disponibile. Questo si verifica nei casi di syscall lente, tipicamente operazioni di I/O. Il processo deve attendere la risposta del controller, nel frattempo non può fare nulla. Quindi entra in stato bloccato. Viene quindi tolto dalla coda dei processi pronti, e sostituito con un altro. Quando viene ricevuta la risposta, viene rimesso in questa coda. Lo scheduler non darà comunque la priorità a lui. In alcuni casi il processo può richiedere di autobloccarsi, ad esempio quando chiama la fork e aspetta la creazione del figlio.

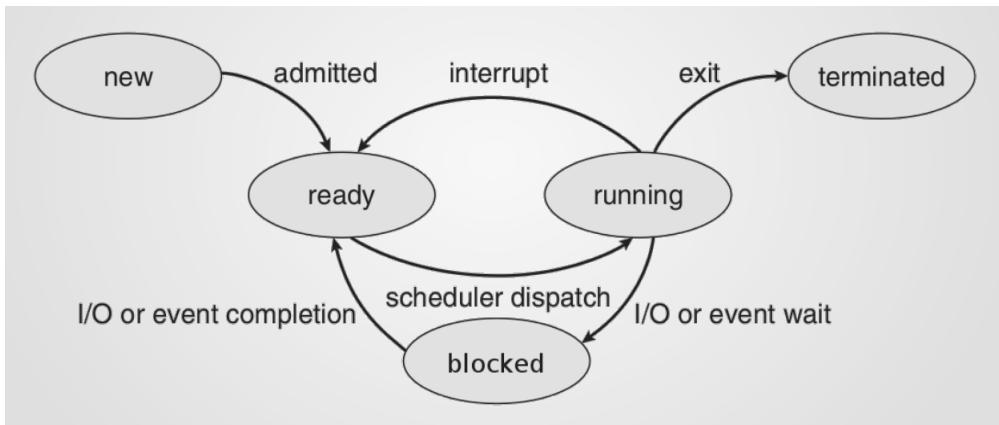


Figure 12: Stati di un processo e transizioni.

Un tipico evento non I/O che può far bloccare un processo è l'attesa della terminazione di un altro processo. Un classico esempio è l'esecuzione di un comando da terminale. La shell (processo P1) chiama la fork e genera P2. Dopo P1 chiama la syscall wait() e si blocca, in attesa che P2 esegua il suo blocco di codice (il comando che abbiamo dato da terminale) e termini. Quando P2 termina, e P1 si risveglia. Infatti la shell riprende la sua esecuzione, e magari mostra il risultato, solo dopo che le azioni eseguite da quel comando sono state completate. Teoricamente sarebbe possibile non far chiamare la wait, mettendo una & dopo il nome del comando, ma questo causa molti problemi.

N.B. Un processo in running può essere tolto oppure no dalla coda dei processi pronti, dipende dall'approccio utilizzato.

N.B.(2) Il processo pronto da far eseguire alla CPU viene scelto dallo scheduler applicando un algoritmo.

Cosa succede in caso di interrupt? Si attiva la routine di gestione, che individua il processo che l'ha causato. Questo processo viene rimesso nella coda dei processi pronti, diventa di nuovo ready. Simile a ciò che succede in caso di interrupt è transizione da running a ready. Questa transizione è possibile solo in caso si tratti di un sistema preemptive (prelazione). In un sistema con prelazione un processo può usare la CPU soltanto per un periodo limitato di tempo. A tempo scaduto, anche se ancora non terminato "di suo", viene terminato forzatamente "a metà" e rimesso nella coda dei processi pronti. Non ha ancora completato la sua esecuzione, e non è neanche bloccato. Ovviamente il suo stato viene salvato. Il meccanismo di prelazione viene implementato con interrupt, però usato come una specie di sveglia. Cioè: controlla periodicamente se il tempo è scaduto. Dato che funziona come un normale interrupt, si eseguono i classici passi che si eseguono nel caso di interrupt:

- salvataggio di PC e PSW del processo nello stack attuale;
- caricamento della routine dal vettore degli interrupt;
- salvataggio degli altri registri del processo, nel suo PCB. Vengono salvati anche i **memory limits** (inizio e fine dello spazio riservato al processo nella RAM);
- esecuzione della routine;

- (se il tempo è effettivamente scaduto) interrogazione dello scheduler;
- ripristino dello stato (dal PCB) del processo scelto;
- esecuzione processo scelto.

Se un processo termina, i suoi eventuali figli vengono solitamente "adottati" dal processo INIT. In alcune distribuzioni Linux moderne questi "orfani" vengono adottati dal processo **systemd**, cioè un demone che rappresenta la radice dei processi utente. Tutti i processi utente sono suoi discendenti.

### 2.1.3 Accodamento processi

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

Una cosa simile accade anche nel caso di eventi software, ad esempio: se un processo viene bloccato da un semaforo, viene messo in una coda contenente tutti i processi messi in coda da quel semaforo. Queste code sono implementate con delle liste doppiamente linkate.

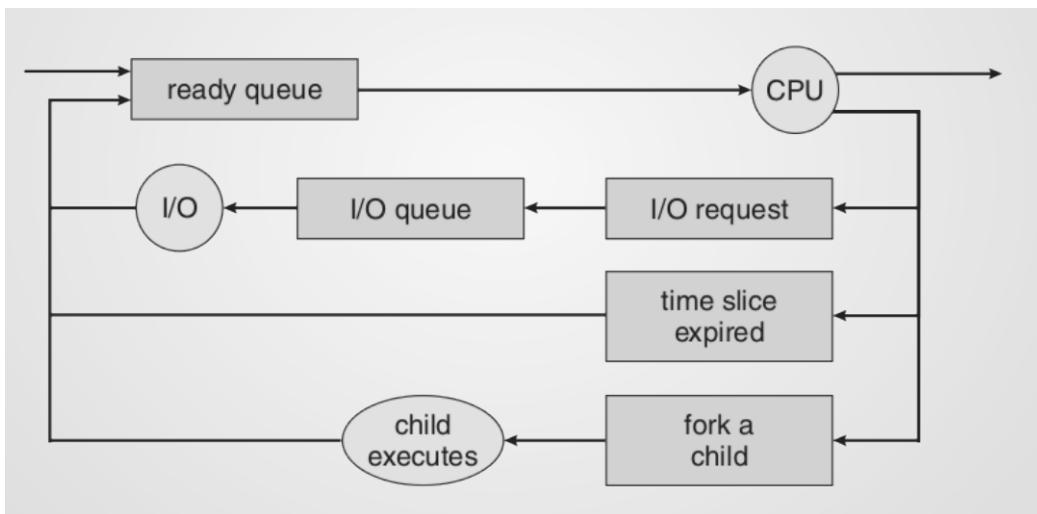


Figure 13: Diagramma di accodamento dei processi.

## 2.2 Thread

Possiamo vedere un processo come una collezione di risorse (spazio di memoria, registri, file aperti, ecc.). Possiamo distinguere anche un flusso di esecuzione, cioè il codice eseguito in quel momento dal processo. Questo flusso viene chiamato **thread**. Fino ad ora abbiamo dato per scontato che ci fosse soltanto un thread per ogni processo, cioè che ogni processo eseguisse solo una sequenza di istruzioni. Ma in realtà, dentro un processo ci possono essere più flussi di esecuzione, più thread. Si parla di programmazione **multithread**. Un'applicazione (processo)

può eseguire così più task senza "bloccarsi".

I thread sono indipendenti tra loro (ognuno ha la sua CPU virtuale o non), ma condividono le risorse su cui vanno ad operare, cioè le risorse del processo. Quindi: usano le stesse risorse, ma fanno "cose" diverse. In un programma multithread, ogni "parte" del programma è un thread diverso. C'è un thread principale, che assegna i compiti agli altri thread.

Vantaggio: i thread collaborano tra di loro, perchè operano sulle stesse risorse (es. un albero in memoria: un thread lo riempie, un altro lo consulta per poi fare delle cose). Ovviamente ci sono anche svantaggi, che saranno spiegati successivamente. Tutte le applicazioni moderne sono multithreading.

Per esempio un browser: nell'atto da parte dell'utente di visitare un sito, partono una serie di download (operazioni simili ad I/O), cioè vengono scaricati i componenti della pagina Web. Bisogna gestire tutti questi download e il rendering delle parti scaricate. Per non far rimanere per troppo tempo una schermata vuota, sarebbe auspicabile renderizzare le parti già scaricate durante il download delle parti mancanti. Ognuna di queste è un'operazione bloccante, quindi un processo "normale" (un solo thread) si bloccherebbe in attesa. Ma se ci sono più thread che si occupano ognuno di una diversa operazione (uno si occupa del download, un altro del rendering) se uno si blocca, l'altro può essere eseguito senza problemi. Un terzo thread potrebbe essere deputato ad eseguire gli script JS della pagina.

N.B. l'OS in realtà, fisicamente non vede dei processi, ma dei thread.

Un altro esempio molto calzante è quello di un web server. Esso ha memorizzate delle risorse in locale che rende disponibili a chi ne fa richiesta. Nel caso di pagine attive (con del backend, es. PHP) il server deve renderizzare il tutto, magari usando risorse recuperate da un database esterno e restituire la risposta. Dopo l'interrogazione del database, il web server deve bloccarsi ed attendere la risposta del database, non potendo gestire altre richieste. Questo se facciamo finta che il web server abbia soltanto un thread. Se ha più thread, avrà anche un dispatcher che assegna le richieste pendenti agli altri thread liberi (può creare nuovi thread o riciclarne di vecchi rimasti liberi). Quindi ogni thread si occuperà di una diversa richiesta. Le richieste saranno gestite in parallelo, vero o pseudo.

Un terzo piccolo esempio può essere quello di un'applicazione desktop dotata di GUI. Essa deve essere gestita da un thread apposito, per evitare che si freezi mentre l'applicazione sta facendo qualcosa.

N.B. Anche tra i thread può esserci concetto di priorità, vedremo dopo.

L'unico rallentamento portato dalla programmazione multithread è dato dal coordinamento tra i thread (ricordiamo che le risorse sono condivise).

## 2.2.1 Organizzazione dei thread

I thread non sono del tutto indipendenti tra loro: lavorano sullo stesso oggetto (es. word processor: si effettuano diverse operazioni sullo stesso documento). Ma i thread in sè sono diversi tra loro, sono delle entità separate, hanno una loro "storia".

Nello specifico: ogni thread ha un suo stack, i suoi registri, il suo PC e il suo stato. Possiamo distinguere, oltre alla tabella dei processi, anche la **tabella dei thread**, che poi in realtà negli OS moderni (che vedono soltanto i thread) è l'unica tabella fisicamente esistente. In questo

modello, il processo non è altro che un contenitore di thread. Anche gli scheduler reali lavorano direttamente sui thread.

Come dicevamo, il resto delle risorse rimane comune tra tutti i thread, ad esempio: strutture dati, codice, file aperti. Cioè se un file è aperto da un thread, è leggibile/modificabile anche dagli altri. Quando si passa da un thread ad un altro, accade qualcosa di molto simile al context-switch spiegato precedentemente, però qui avviene a livello dei thread. L'unica particolarità di questo caso è che dipende dalla parentela tra thread. In pratica, è più facile passare da un thread all'altro se essi sono fratelli. Questo perchè condividono la memoria, per cui non hanno bisogno di cambiare l'area di memoria dove si lavorerà (non bisogna riprogrammare la MMU, Memory Management Unit), né cambiare processo. Lo scheduler, sapendolo, cercherà di selezionare un thread fratello di quello attualmente in esecuzione, se possibile.

## 2.2.2 Operazioni sui thread

Queste operazioni non coinvolgono il kernel.

- **thread\_create**: un thread ne crea un altro;
- **thread\_exit**: il thread chiamante termina. Un processo termina quindi quando tutti i suoi thread terminano;
- **thread\_join**: un'operazione simile alla chiamata wait dei processi. Un thread si blocca in attesa della terminazione di un altro thread;
- **thread\_yield**: collaborazione tra thread. In pratica, un thread può volontariamente rilasciare la CPU ad un altro thread. Se il codice è scritto bene, i thread fratelli si rilasceranno la CPU tra loro quando più opportuno.

Ovviamente anche i thread possono assumere degli stati, completamente identici a quelli che possono assumere i processi.

## 2.2.3 Thread e programmazione multicore

Avere più thread dà vantaggi in termini di scalabilità, ma anche in termini di velocità pura, se pensiamo a CPU multicore. Cioè: si può fare uso del parallelismo, come detto diverse volte. Nel caso di una singola CPU, si parla di pseudoparallelismo, gestito con delle CPU virtuali, idealmente una assegnata ad ogni thread. Fisicamente, i thread si "contendono" l'utilizzo della vera CPU. Nel caso di più CPU, i thread si distribuiscono tra le CPU: alcuni thread useranno un core, altri thread ne useranno un altro, e così via.

Rovescio della medaglia: la programmazione multithread e multicore è molto più complessa, dal punto di vista del programmatore. Possiamo vedere cosa significa ciò facendo l'esempio di una applicazione già esistente, che sfrutta un solo thread. Facciamola diventare multithread. Dovremo tenere conto di diversi aspetti:

- Separazione dei task: bisogna identificare i "task" (singoli compiti) che l'applicazione deve eseguire, e dedicare un thread ad ognuno. Riprendiamo l'esempio del word processor di prima. Riusciamo ad identificare diversi compiti: un task dovrà modificare il documento, un altro dovrà occuparsi della correzione ortografica, ecc.  
Non si può separarli a caso. Bisogna seguire certi criteri, per capire quali operazioni singole è possibile raggruppare dentro un task.
- Bilanciamento dei task: non bisogna creare né dei task troppo grandi, né troppo piccoli. Bisogna tener conto del ruolo del task, del tempo in cui rimarrà in vita e del suo carico computazionale; quindi capire ad esempio se le operazioni che effettua possano essere bloccanti.
- Suddivisione e dipendenza dei dati: parliamo di thread, quindi la memoria (i dati) sono condivisi tra loro. Bisogna decidere come suddividere i task, cioè decidere su quali dati debba operare ogni task. Bisogna anche coordinare bene gli accessi, per evitare accavallamenti tra le operazioni di più task.  
Importantissimo anche valutare le dipendenze dei dati: le attività di un task possono dipendere dalle attività di un altro task? Ovviamente sì, magari un task deve leggere un dato, ma deve leggerlo solo dopo che un altro task lo abbia già modificato. Qui è ovvio pensare a problemi di sincronizzazione. Deve essere eseguito prima il task che scrive o quello che legge? Oppure: se la struttura dati rimane vuota, cosa deve fare il task che ne leggeva gli elementi? Verrà "addormentato", e risvegliato una volta che la struttura conterrà nuovamente qualcosa, ma non può sempre controllare lui stesso che sia effettivamente così. Sarà risvegliato da qualcuno di "esterno". Consideriamo l'esempio opposto, possibile in caso la struttura sia statica: se è piena? Il task che ne inseriva i dati viene addormentato (quindi stessa cosa ma al contrario).
- Sequenzialità delle operazioni: dato che i flussi sono in parallelo, non è possibile garantire che l'ordine di esecuzione sia rispettato. Si tratta di un bug molto comune in questi casi, causato dall'interlacciamento tra i thread. Bisogna farli interlacciare nel modo desiderato.
- La complessità del debugging e dei test.

N.B. Si potrebbero verificare dei deadlock tra thread.

## 2.3 Implementazione dei thread

E' possibile implementare il modello a thread in 3 diversi modi. Non c'è uno più conveniente dell'altro (tranne il terzo, vedremo perchè) ma dipende dall'OS su cui si lavora.

### 2.3.1 Modello 1 a molti (thread a livello utente)

Questo modello si usa nei casi in cui l'OS non supporti i thread a livello kernel. Questo vale soprattutto se si tratta di vecchi OS.

In pratica: i thread vengono implementati "dentro" il processo, a livello utente, dato che OS

vede solo quelli. Per implementare le varie operazioni (viste nel paragrafo 2.2.2) e le relative strutture dati in modalità utente, ci si serve di particolari librerie, dette **runtime environment** o **runtime system**. Queste librerie "faranno le veci" dell'OS nella gestione dei thread. Non possiamo sfruttare le potenzialità delle CPU multicore, perchè il core viene assegnato dall'OS al singolo processo (stessa cosa se parliamo di CPU virtuali). Deve essere divisa manualmente dal programmatore tra i thread.

E' vitale che ogni thread rilasci la CPU quando necessario. Se non la rilascia, nessun altro thread può essere eseguito. Quando invece la rilascia, deve salvare il suo stato (non può modificare PSW ricordiamo, siamo in modalità utente) dentro una particolare tabella del thread contenuta nella memoria del processo. e precisamente dentro il runtime system. Sarà compito del runtime system anche lo scheduling soltanto dei thread nel processo in questione. Uno dei pro è che lo scheduling può essere personalizzato secondo le esigenze di quel processo, tanto è direttamente il programmatore che lo implementa. Per effettuare context-switch non c'è neanche bisogno di effettuare una trap, perchè implementato da runtime system. Ciò si traduce in minor overhead. Veniamo adesso ai problemi: se un thread effettua una chiamata bloccante, OS vede come se la chiamata l'avesse fatta l'intero processo: blocca tutto il processo. Tutti i thread vengono bloccati a loro volta. Questo problema può essere mitigato seguendo una certa strategia, che consiste nell'uso, da parte del thread, della syscall **select** (ad esempio su un dispositivo di I/O), per poter vedere se la chiamata che sta per effettuare sarà bloccante o meno in quel momento. Se vede che la chiamata non è bloccante, la esegue. Se invece in quel momento la chiamata è bloccante, cede la CPU (yield) ad un suo fratello. La CPU sarà "passata di mano" tra vari fratelli prima di ritornare a lui. Quando gli ritorna, rieffettua la select: se la chiamata è ancora bloccante, cede ancora la CPU ad un suo qualsiasi fratello, e questo in loop finchè la chiamata non è più bloccante. Possiamo considerare questo comportamento una specie di *busy-waiting*. Un altro caso in cui si presenta questo problema è quello in cui, usando la paginazione, si incontra un **page-fault**. Cioè la CPU non trova il dato richiesto in memoria, e dà il compito all'OS di recuperarlo dal disco e metterlo in memoria. Ovviamente questo comporta un blocco di quel processo. Il blocco è asincrono, il thread non sa che il blocco è avvenuto a causa sua, quindi non possibile gestire questo problema con chiamata select.

### 2.3.2 Modello 1 a 1 (thread a livello kernel)

Se i thread sono implementati dall'OS a livello kernel, è possibile sfruttare gli strumenti che ci messe egli stesso a disposizione.

Ci sarà una tabella dei thread, contenente tutti i thread creati sulla macchina, memorizzata nel kernel. Il context-switch necessita ora una trap, quindi più lento rispetto al modello 1 a molti. Dato che l'OS vede i thread, molto più facile ed efficiente crearli e distruggerli, rispetto ai processi (dalle 10 alle 100 volte). In pratica si tratta solo di inserire o rimuovere voci dalla tabella dei thread. Ovviamente queste operazioni hanno comunque un costo. Si sceglie di implementare modello a worker: vengono creati un numero fisso di thread all'avvio, detti **worker**, e tenuti in idle. Quando necessari, vengono risvegliati. Quando un thread termina il suo compito, non viene distrutto, ma rimesso in idle. Si risparmia sul costo delle creazioni e distruzioni.

### 2.3.3 Modello molti a molti (ibrido)

Abbiamo visto che entrambi i modelli hanno diversi problemi. Si sceglie semplicemente di prenderne solo i pregi, e costruire un modello ibrido, detto modello **molti a molti**. Si implementa comunque su un OS che supporta i thread kernel, ma si farà distinzione tra thread utente e thread kernel. Cioè: è possibile assegnare più thread utente ad un solo thread kernel, oppure assegnare un solo thread utente ad un thread kernel. Questo a completa discrezione del programmatore (solitamente dipende dall'importanza delle task). Stiamo applicando tutti e due i modelli, ma in piccolo. Ovviamente è più complicato da gestire. Il pregio più grande è quello di scegliere quando applicare una strategia e quando un'altra. Riprendiamo per l'ennesima volta l'esempio del word processor: le operazioni di I/O e quella di correzione ortografica possono coesistere. Se il thread kernel si blocca, non è un problema che si blocchino tutti i thread utente. Se sto effettuando un'operazione di I/O, non mi importa più di tanto che in quel momento non funziona il correttore ortografico.

Discorso diverso per la GUI: non devo permettere assolutamente che si freezi: avrà un thread kernel dedicato.

### 2.3.4 Classifica tipi di context-switch (da più veloce al più lento)

1. Thread fratelli, con thread implementati a livello utente: non c'è trap;
2. Thread fratelli, con thread implementati a livello kernel: proprio per questo c'è necessità di fare la trap;
3. Processo-processo: i thread non sono parenti, bisogna cambiare totalmente area di memoria, riprogrammare MMU.

### 2.3.5 Come sono implementati i thread negli OS moderni

Ormai quasi tutti gli OS moderni hanno implementati i thread a livello kernel. Ad esempio Windows vede soltanto i thread.

Linux invece vede il tutto a livello di **task**, con un significato diverso da quello dato nelle pagine precedenti. Un task si può definire come un ibrido tra processo e thread. Esiste una chiamata detta **clone** che è più modulabile della fork, che può funzionare sia come essa sia come una `thread_create`. Anzi, può funzionare in tanti modi anche ibridi; `fork` e `thread_create` sono soltanto due casi particolari.

La semantica delle operazioni è molto simile tra gli OS: ciò che cambia, è la sintassi. Ci sono però delle altre librerie, dette **wrapper**, che riescono ad unificare le varie operazioni tra i diversi OS.

La libreria standard di accesso ai thread kernel è la libreria **Pthreads** conforme agli standard POSIX, che è usata da Solaris, Linux, Windows e Mac OS. Altre librerie consentono di gestire i thread a livello di utente, e sono diverse per ogni OS. Scendendo ancora, possiamo vedere che i thread sono implementabili anche a livello di singolo linguaggio. Ad esempio, linguaggi come Java, Rust, C# e Go implementano il concetto di thread.

## 2.4 InterProcess Communication (IPC)

I processi hanno la necessità di comunicare tra loro. Ogni processo possiede dei canali di comunicazione standard, uno per l'input, uno per l'output e un altro, sempre di output, per i messaggi di errore. Nei sistemi UNIX, un processo è visto come qualcosa che prende dei dati in input, li elabora e li restituisce in output. Il canale dell'output deve rimanere "pulito", non contaminato da messaggi di errore vari, per questo c'è il terzo canale dedicato agli errori. I canali hanno delle assegnazioni standard. L'input è associato alla tastiera, l'output e gli errori sono associati al terminale.

### 2.4.1 Pipe

Dalla shell si possono lanciare due o più processi in contemporanea, che saranno in comunicazione tra loro attraverso i canali standard. Cioè il canale di output di P1 è connesso al canale di input di P2. Dato che i processi sono eseguiti in parallelo, bisogna coordinarli. Ci sarà un buffer intermedio tra i due, che conserva l'output di P1 in attesa che P2 possa riceverlo in input. Se il buffer si svuota, P2 si blocca (non ha altro input). Se uno dei due processi termina, si chiude il canale di comunicazione. L'altro processo se ne accorge, e solitamente termina anch'esso. Questo modello di comunicazione è detto **a pipe**: molto limitato, ma semplice da capire e utilizzare.

### 2.4.2 Problemi

Al di fuori di questo canale offerto dall'OS, i processi sono isolati tra loro. Per metterli in comunicazione ugualmente, senza sfruttare il canale, si può pensare ad una finestra di memoria condivisa, in cui uno dei due possa memorizzare dati e l'altro possa prelevarli. Questa idea molto simile al funzionamento dei thread, che hanno una memoria condivisa, ma qua si parla di processi.

Possono presentarsi problemi di accavallamento delle operazioni: se ad esempio P1 legge il dato X mentre P2 scrive nella locazione dov'è salvato X, P1 potrebbe acquisire informazioni "outdated". E' necessaria una sincronizzazione tra i due processi.

### 2.4.3 Race conditions

Per far capire il problema rappresentato dalle **race conditions** (corse critiche), facciamo un classico esempio, ripreso da tutti i testi sull'argomento.

Cioè l'esempio dei versamenti su un conto corrente. Ipotizziamo che due processi abbiano una finestra di memoria condivisa, in cui è memorizzata una variabile (che rappresenta il saldo del conto corrente). I due processi effettuano dei versamenti (incrementi sulla variabile). Ricordiamo che l'incremento non è un'operazione elementare, è la concatenazione di più operazioni elementari. Bisogna prelevare il dato dalla memoria, salvarlo su registro, incrementare il valore, riscriverlo su registro, fare la STORE in memoria. Ipotizziamo che questi processi siano eseguiti su una sola CPU. Se non si usa CPU virtuale, quindi P1 e P2 eseguiti in maniera sequenziale, nessun problema. Dopo l'esecuzione dei due, vedremo il valore del saldo incrementato di 2.

Il grosso problema viene quando i due processi sono eseguiti in contemporanea. Facciamo finta che venga eseguito per primo P1. Durante l'esecuzione di P1, il meccanismo di prelazione dell'OS vede che il tempo è scaduto, toglie la CPU a P1 e la dà a P2. P1 potrebbe essersi interrotto proprio nella fase intermedia tra incremento e salvataggio in memoria: avremo che P1 ha già incrementato (da 0 a 1), ma non ha ancora salvato il risultato in memoria. Nel frattempo viene eseguito P2: P2 va a buon fine, incrementa 0 ad 1 e lo salva in memoria. In memoria c'è attualmente 1, quindi.

Ora arriva di nuovo il momento di P1, che riprende da dove si era fermato prima: deve salvare il suo dato incrementato in memoria, e lo fa. Ma il dato in questione è ancora 1, perchè ricordiamo che aveva incrementato 0. Ci saremmo aspettati di trovare il valore 2 in memoria, ma in realtà ci troviamo 1. Questo genere di problema è quello che viene chiamato **race condition**, corsa critica. Può accadere anche a livello kernel, con più processi che aprono un file, od effettuano operazioni su tabella dei processi, o coda dei processi pronti. Qui la gravità del problema aumenta.

La soluzione principale si dice **mutua esclusione**. Cioè: se ci sono "pezzi" di codice dei (per semplicità due) processi che accedono alla memoria condivisa, detti **sezioni critiche**, si pongono delle condizioni alla loro esecuzione. In pratica: solo un processo alla volta può eseguire la sua sezione critica, in modo da poter accedere esclusivamente lui in un dato momento. Entreranno nella memoria condivisa "in fila".

## 2.5 Mutua esclusione

NOTA INIZIALE: SI PUO' PARLARE DI THREAD O PROCESSO IN MODO INTERSCAMBIABILE. LE CONSIDERAZIONI FATTE VALGONO IN ENTRAMBI I CASI.

Abbiamo capito come risolvere il problema delle race conditions: concetto di mutua esclusione. Tra tutti i thread che condividono la struttura dati, solo uno di essi si può trovare in una sezione critica che riguarda quella struttura, in un dato istante. Si evita così l'interlacciamento tra operazioni di accesso alla memoria (e modifica) di più thread, quella che abbiamo chiamato race condition.

### 2.5.1 Condizioni per risolvere Race condition

1. La condizione indispensabile per risolvere problema. Cioè mutua esclusione.
2. La validità della soluzione non deve dipendere da fattori come velocità della CPU o il suo numero di core. Non si faranno ipotesi su timing e cose varie, perchè altrimenti la soluzione funzionerebbe soltanto con alcuni hardware.
3. Un processo/thread che è fuori dalla sua sezione critica non deve bloccarne un altro che sta per entrarci. Da tenere in conto: una delle nostre soluzioni sembrerà "buona" ma non soddisferà questa condizione. Essa è molto importante perchè un processo deve bloccarsi solo il minimo indispensabile, per una buona ragione.

- Nessun processo deve attendere all'infinito l'ingresso in una sezione critica. Dovrà sblocarsi entro un certo limite di tempo (quantificato), garantito dal software.

### 2.5.2 Un esempio di mutua esclusione

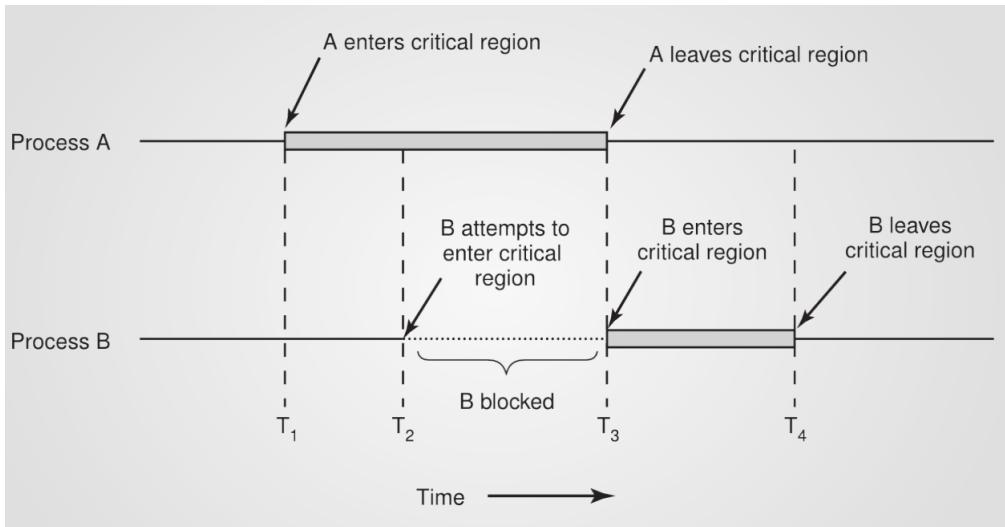


Figure 14: Esempio di mutua esclusione tra due processi.

Abbiamo due processi:  $P_a$  e  $P_b$ .  $P_a$  all'istante  $T_1$  senza problemi, perchè  $P_b$  sta facendo "altro". All'istante  $T_2$ ,  $P_b$  necessita di entrare nella sua sezione critica, ma già  $P_a$  è in una sezione critica.  $P_b$  dovrà aspettare la fine della sezione critica di  $P_a$ , che avverrà all'istante  $T_3$ . Cioè  $P_b$  effettuerà una chiamata bloccante. All'istante  $T_3$   $P_b$  può entrare nella sua sezione critica ed eseguirla. Se  $P_a$  volesse rientrarci, dovrà attendere  $P_b$ .

## 2.6 Implementazione della mutua esclusione

Come implementare questo meccanismo, esattamente come abbiamo visto sopra? Ci sono diverse soluzioni, che via via diventeranno sempre più complesse, ma saranno sempre migliori (cioè funzionanti in più casi e più efficienti). Di default assumiamo si stia parlando di un sistema con CPU singlecore.

### 2.6.1 Disattivazione degli interrupt

Ogni OS moderno sfrutta il meccanismo della prelazione per garantire la multiprogrammazione. Ricordiamo che la prelazione ha il "potere" di bloccare un processo, per assegnare la CPU ad un altro processo. È proprio la prelazione a causare il problema della race condition; e sappiamo bene la prelazione sfrutta gli interrupt per funzionare. Si può pensare: tolta la prelazione (almeno per quell'istante), problema risolto. In alcuni casi potrebbe anche funzionare abbastanza bene.

In dettaglio: all'ingresso di una sezione critica, gli interrupt vengono disabilitati, fino alla fine di essa. Così, nel mentre si trova nella sezione critica, quel processo non può essere bloccato in

alcun modo. Tutto semplice, e funziona anche. Ma nei sistemi multicore, gli interrupt vengono disattivati solo su quella CPU. Se l'altro processo si trova su un'altra CPU, ci può entrare comunque, perché le CPU lavorano in parallelo. Il problema è tale e quale a prima. Si potrebbe parlare di come disabilitare interrupt sia una cosa gravissima e problematica, soprattutto se fatto per un tempo prolungato (pensate al fatto che se disattivazione è prolungata la coda degli interrupt si riempie ed alcuni vengono addirittura scartati. Gravissimo). Ma questa soluzione è (era) usata soltanto con processi a livello kernel, che devono eseguire operazioni delicate per cui è necessario disattivare la prelazione, e soprattutto non fanno nulla di "strano", perché scritti dai programmati del kernel. In questo caso funziona, ed è molto semplice da implementare.

## 2.6.2 Variabili di lock

Questa e le soluzioni successive implemeteranno delle funzioni, le cui chiamate delimitano la sezione critica. Le funzioni in questione si chiamano **enter\_region()** e **leave\_region()**. Si occuperanno di eseguire le operazioni necessarie a realizzare la mutua esclusione, ovviamente l'implementazione dipende dalla soluzione scelta.

Torniamo a noi: si può creare una variabile condivisa tra tutti i thread, detta **variabile di lock** che può assumere due valori. Vale 0 se nessun thread è nella sua sezione critica, 1 se qualcuno lo è. Ogni thread passerà il valore della variabile all'`enter_region`, che effettua continuamente un controllo finché il valore non sarà 0. A quel punto farà entrare il thread nella sua sezione critica. Ovviamente l'altro che la sta occupando, quando finisce, chiama `leave_region` e rimette il valore a 0. Questa soluzione (e quasi tutte le successive) sfruttano il meccanismo di **spin lock**, che altro non è che busy waiting. Ora pensiamo ad una cosa: il problema era quello di avere risorse condivise. Ma la stessa variabile è condivisa. Quindi il problema non si risolve, si sposta sulla variabile. E' essa che causa la race condition, ed `enter_region()` fallirà in certi casi!

## 2.6.3 Alternanza stretta

```

int N=2
int turn

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

function leave_region(int process)
    turn = 1 - process

```

Figure 15: Pseudocodice dell'alternanza stretta.

La variabile è sempre condivisa, ma non sarà più uno switch 0/1. La nuova variabile, detta **turn**, assume come valore l'ID del processo abilitato ad entrare nella sezione critica. Quando un processo chiama la `enter`, le passa il suo ID. Se il valore di `turn` è diverso dall'ID del processo, significa che un altro processo è dentro la sua sezione critica, quindi busy waiting. L'altro

quando chiama la leave, assegnerà a turn l'ID del processo che ha chiamato la enter. A quel punto lo spin lock termina ed esso può entrare nella sua sezione critica.

Qui non si presenta la race condition della variabile turn, perchè intanto le chiamate sono coordinate, e poi il valore di turn è univoco. Cioè, se due chiamate enter in contemporanea, una si blocca, l'altra no, perchè turn contiene per forza di cose l'ID di uno dei due (se parliamo di due processi). Il punto di forza di questa soluzione è anche la possibilità di generalizzarla ad N processi. Esempio: 10 processi devono entrare nella loro sezione critica. 9 aspetteranno, 1 entrerà.

Il problema di questa soluzione è la rigidità delle turnazioni. Cioè viola la condizione 3. Proviamo a capire perchè con un esempio.

Abbiamo processi P0 e P1. P0 entra nella sezione critica, la completa ed esce. Cede il turno a P1, che però sta facendo ancora altro. Ipotizziamo che nel frattempo P0 deve rientrare nella sezione critica, ma P1 non ci è ancora entrato. Il turno è di P1, quindi P0 dovrà aspettare finchè P1 arriva alla sezione critica e la completa, anche se potenzialmente P0 potrebbe entrare anche adesso senza creare nessun problema. Quindi il problema principale è che i processi possono rimanere bloccati per molto, per rispettare le turnazioni.

## 2.6.4 Soluzione di Peterson

```
int N=2
int turn
int interested[N]

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

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

Figure 16: Pseudocodice della soluzione di Peterson.

Come le precedenti, neanche la soluzione di Peterson necessita di hardware aggiuntivo. È implementata totalmente a livello di software, può essere quindi implemetata anche da utente, non solo da kernel. Lo pseudocodice qui sopra si applica al caso in cui in gioco ci siano soltanto due processi. È possibile generalizzare la soluzione, spiegheremo più avanti come.

Si basa sempre sull'idea di turnazione, ma la principale differenza è che sfrutta anche un array di supporto booleano chiamato interested. Ogni indice è associato ad un processo; se il valore corrispondente a quell'indice è true, significa che il processo è "interessato", cioè deve entrare nella sua sezione critica. Di default ovviamente tutti i valori sono false.

La enter\_region prende come parametro sempre l'ID del processo chiamante, e si "procura" id dell'altro processo. Il suo slot in interested diviene true, significa che vuole entrare nella sezione critica. Ogni processo può idealmente modificare soltanto il suo slot. Anche se il turno non è ancora suo, turn = process. Vedremo successivamente il perchè. Il while ci fa notare che la

soluzione di Peterson fa ancora uso del busy waiting, ma questa volta è più "elaborato". Cioè: dando per scontato che `turn == process` sempre vero (modificata nella linea appena sopra!), bisogna soltanto fare in modo che `interested[other] == false`. Perchè, se `interested[other] == true`, significa che la `enter_region` è stata già chiamata dall'altro processo, quindi si trova sicuramente ancora dentro la sezione critica. Se ne fosse già uscito, il suo slot sarebbe falso, perchè è esattamente quello che fa quando chiama la `leave_region`. Quindi appena `interested[other] == false`, significa che l'altro processo ha abbandonato la sua sezione critica.

Questo fa in modo che non ci sia una turnazione così rigida, che portava al fallimento della soluzione con alternanza stretta. Ma cosa succede se `enter_region` viene chiamata contemporaneamente dai due processi? Significa che entrambi sono fuori, e ognuno aggiorna il suo slot con il valore `true`. La variabile `turn` è ancora condivisa, però. Cosa succede? Uno dei due la aggiornerà per primo, e verrà inevitabilmente sovrascritto da quello che arriva per secondo.

**Qua entra in gioco la seconda condizione.** Il processo che avrà aggiornato `turn` per ultimo, si ritroverà con la condizione del `while` soddisfatta (anche la seconda), mentre l'altro no. Quindi entra per primo l'altro, e il secondo aspetta. Riusciamo comunque a rispettare la mutua esclusione anche in caso di chiamata contemporanea.

Per generalizzare questa soluzione anche per  $N$  processi, pensiamo al fatto che i due processi, all'aggiornamento di `turn`, è come se si sfidassero a duello. Solo uno dei due vincerà. Quindi l'idea per gestire  $N$  processi è farli competere in una sorta di "torneo", in cui il vincitore di uno scontro sfida un altro processo, e così via finché non rimane un solo vincitore, che sarà il fortunato ad entrare nella sezione critica.

Questa soluzione sembra funzionare in ogni caso. Ma: non funziona in caso di sistemi multicore. Per far capire il motivo, facciamo un esempio.

Abbiamo due thread,  $T_1$  e  $T_2$ , che effettuano due istruzioni del tipo `FETCH` e `STORE` su due variabili,  $x$  e  $y$  poste inizialmente a 0.  $T_1$  memorizza il valore 1 nella variabile  $x$ , poi preleva il valore di  $y$  e lo mette nel suo registro  $R_1$ .  $T_2$  invece aggiorna il valore di  $y$  sempre ad 1, e preleva il valore di  $x$  mettendolo nel suo registro  $R_2$ . Quanto valgono i rispettivi registri? Dipende dal timing con cui vengono eseguiti i thread, che ricordiamo sono eseguiti su due CPU diverse. Ci sono tre casi abbastanza prevedibili, ma un quarto caso che risulta imprevisto, cioè quello in cui sia  $R_1$  che  $R_2$  valgono 0. Cosa è successo? I sistemi multicore gestiscono le varie CPU riordinando spesso le istruzioni che eseguono, in modo da migliorare efficienza, ma facendo in modo che le istruzioni portino allo stesso risultato, anche se il loro ordine di esecuzione viene rimescolato. In questo caso il meccanismo di riordino viene tratto in inganno dal fatto che si parli di due variabili distinte, anche se condivise (lui non lo sa!).

Questa è proprio la situazione in cui ci si ritrova quando si aggiornano i due slot di `interested`, nella soluzione di Peterson. Cioè può capitare che tutti e due i slot siano messi a `false`, causando il blocco di entrambi i processi (un deadlock). Il problema viene parzialmente risolto da particolari istruzioni previste per i sistemi multicore/a pipeline che permettono la sincronizzazione tra più thread, dette barriere. In parole povere queste istruzioni mettono una "barriera" ad una certa istruzione, in modo che tutti i processi si fermino lì prima di continuare.

## 2.6.5 Istruzione TSL (e XCHG)

Questa soluzione sfrutta l'hardware, per permettere la realizzazione della mutua esclusione anche su CPU multicore. Viene implementata una nuova istruzione ad hoc: **TSL** (Test and Set Lock), che su architetture Intel prende il nome di XCHG (eXCHanGe). La sintassi di TSL è molto semplice: TSL Registro, LOCK. Dove LOCK è una locazione di memoria. Cosa fa effettivamente TSL?

TSL effettua il fetch del valore contenuto in LOCK sul registro. Successivamente scrive un predeterminato valore diverso da 0 (es. 1) nella locazione L. Cioè doppia azione FETCH e STORE. Possiamo quindi vederla come l'unione di due istruzioni MOV. Cosa cambia? TSL garantisce atomicità: fa le stesse cose, ma viene vista dall'OS come una singola istruzione. Tecnicamente non è possibile interlacciare le due MOV con un'altra istruzione, oppure riordinarle. Quindi questa proprietà di atomicità risolve il problema del riordino che invalidava la soluzione di Peterson, ma questo solo nel caso dei single core. Ma Peterson funzionava di suo nei sistemi single core! Quello che fa la differenza nei sistemi multicore è l'implementazione di TSL offerta da essi: conserva atomicità bloccando il bus della memoria agli altri processi fino alla fine di TSL. Con questa implementazione, TSL fa funzionare le soluzioni basate sui lock (come Peterson) anche sui sistemi multicore.

Ma come implementare, ad esempio, la soluzione con variabili di lock con TSL?

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

Figure 17: Pseudocodice assembly della variabile di lock con TSL.

Quando viene chiamata enter\_region, viene eseguita TSL. Cioè viene letto il vecchio valore di LOCK (0 o 1) e viene scritto il valore 1 su essa. Per vedere se il valore fosse già 1 in precedenza, oppure se è stato aggiornato, basta comparare il valore del registro (il "vecchio" valore di LOCK) e 0. Se il confronto è vero, è stata cambiata da questo processo, quindi esso è libero di entrare nella sua sezione critica. Se il confronto è falso, significa che LOCK era già stata cambiata da un altro processo, per cui c'è busy waiting. All'uscita dalla sezione critica, la leave\_region rimette 0 dentro LOCK. Questa soluzione fin ora è la migliore di tutte, ma si porta dietro il problema del busy waiting, che vedremo poi che non è solo un problema di efficienza. La XCHG di Intel fa la stessa cosa di TSL, ma implementata diversamente: effettua uno scambio tra LOCK e il registro, niente di più, niente di meno. La soluzione con TSL viene impiegata sia a livello utente che a livello kernel. A livello kernel potrebbe andare bene la soluzione con disabilitazione degli interrupt, ma non più nel caso di sistemi multicore.

## 2.6.6 Precisazione su variabile di lock

Una variabile di lock viene associata ad una sola struttura (o ad un solo gruppo di strutture dati). Questo perchè ovviamente le varie sezioni critiche sono associate anch'esse ad una singola

struttura dati, o ad un singolo gruppo. Cioè, se due thread entrano in delle loro sezioni critiche in contemporanea, ma esse si riferiscono a due strutture dati (gruppi) differenti, non c'è alcun problema di race condition. Non necessaria mutua esclusione. Ci potrebbero anche essere delle sezioni critiche miste, cioè che operano su più strutture dati o gruppi. In questi casi, si effettua un blocco multiplo. A seconda dell'ordine in cui vengono effettuati i lock, si potrebbe verificare deadlock. Meglio stabilire un ordine canonico per effettuare i lock. Si potrebbe pensare di dividere una sezione critica mista in più sezioni critiche singole, ma questo porterebbe a sezioni critiche troppo piccole, quindi ad un eccessivo numero di chiamate `enter_region` e una certa quantità di overhead.

### 2.6.7 Sleep e wakeup

Rimane ancora un problema da risolvere: il busy waiting. Esso comporta problematiche più gravi rispetto alla semplice perdita di efficienza, su cui si potrebbe "passare sopra". Capiamolo con un esempio:

Consideriamo due processi, PH e PL. Qui entra in gioco la nozione di priorità. PH ha alta priorità, PL bassa priorità. Ovviamente PH scelto per primo. Quando PH si blocca, PL può avere chance di essere scelto dallo scheduler. Ma quando PH si sblocca, PL viene subito bloccato per far "rientrare" PH. Questo è un problema, in soluzioni come quella TSL. Cioè il famigerato busy waiting causa il problema dell'**inversione di priorità**.

Assumiamo che PH si blocchi quando è fuori dalla sezione critica. Viene eseguito PL, che può ed entra nella sezione critica. Se PH si risveglia quando PL non è ancora uscito, PL viene comunque bloccato e PH prende il suo posto nuovamente. Cosa succede adesso? PH arriva alla sua sezione critica, vuole entrare, ma variabile di lock settata in modo da far entrare PL. Ma PL non aveva ancora terminato, non può chiamare `leave_region`. Per cui PH rimarrà in attesa che PL sblocchi il lock, ma PL non verrà mai rieseguito finché PH non termina. Attesa infinita. Si capisce che questo problema è molto grave, ed è causato dal busy waiting. Bisogna liberarsene.

Quindi: pausa attiva deve diventare una pausa passiva. Bisogna far bloccare e risvegliare il processo dall'OS, con una syscall. Così non dovrà essere il processo a controllare da solo se può risvegliarsi, ma sarà qualcuno "dall'alto" a farlo; sarà necessario anche manipolare particolari strutture dati a livello kernel, ad esempio la coda dei processi pronti. Queste syscall si dicono **sleep** e **wakeup**. Per capire come applicare al nostro problema (race conditions) e risolverne anche un altro (sincronizzazione tra processi) prendiamo in esempio il problema del produttore-consumatore: un problema astratto, che però può aiutare a trovare soluzioni a problemi simili, ma concreti.

### 2.6.8 Produttore-consumatore

Il "produttore" e il "consumatore" sono due processi, con una struttura dati buffer condivisa. Daremo per scontato che la mutua esclusione su questo buffer sia già garantita (vedremo poi come fare). Il problema principale è sincronizzare le azioni dei due processi. Il produttore è un processo che genericamente "produce" una risorsa, e la inserisce nel buffer. Il consumatore

invece estrae una risorsa dal buffer. Bisogna fare in modo che il produttore si blocchi nel momento in cui il buffer è pieno (non può inserire altro); il consumatore deve invece bloccarsi nella situazione opposta, cioè buffer vuoto (non può estrarre nulla). Come si sbloccano? Devono sbloccarsi "tra di loro": produttore sblocca consumatore quando il buffer non è più vuoto, viceversa quando il buffer non è più pieno. Soffermiamoci sul funzionamento dei due processi.

**Produttore** Cosa fa il produttore? Il suo codice è caratterizzato da un ciclo infinito con alternanza di fasi. Intanto produce un item (qualunque cosa sia) e deve procedere ad inserirlo nel buffer. Ma prima deve controllare che il buffer non sia pieno (controlla il contatore degli elementi presenti nel buffer). Se è pieno, chiama la sleep() e si addormenta. Altrimenti, inserisce l'item e aggiorna il contatore (nota: assumiamo mutua esclusione anche su contatore, che è condiviso). Ultima cosa: deve occuparsi di risvegliare il consumatore. Controlla count: se è ad 1 (quindi precedentemente era vuoto e ora non più) chiama wakeup(consumer), cioè risveglia il consumatore.

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

Figure 18: Pseudocodice del produttore.

**Consumatore** Cosa fa invece il consumatore? Esattamente il contrario del produttore: se buffer è vuoto (count = 0) si addormenta, altrimenti estrae l'item, aggiorna il contatore, e si occupa di risvegliare il produttore qualora il buffer non sia più pieno (cioè quando il contatore diventa pari ad N - 1). Infine, "consuma" l'item.

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

Figure 19: Pseudocodice del consumatore.

Si evita busy waiting: ognuno sarà risvegliato dall'altro. Non necessario controllare "da sè". Ora: qual è il problema? E' legato alle race conditions, ma non dipende dal buffer (assunta mutua esclusione). Dipende dalle condizioni di risveglio e addormentamento, magari in caso di prelazione o esecuzione parallela.

Facciamo un esempio di prelazione "sfortunata": viene eseguito il consumatore, con buffer vuoto. Al momento del controllo della condizione bloccante, si accorge di ciò. Fa per addormentarsi, ma prima che riesca completamente ad eseguire la chiamata sleep() interviene la

prelazione. Il nostro consumatore non si è addormentato. Prima o poi toccherà anche al produttore. Esso, ignaro di tutto, produce un item, lo inserisce, aggiorna count ad 1. Questa è proprio la condizione di risveglio del consumatore: il produttore prova a risvegliare il consumatore, ma è già sveglio! Non è riuscito ad addormentarsi. Quindi possiamo dire che il produttore ha chiamato una "sveglia" a vuoto; ha "perso" la sveglia. Quando è di nuovo il turno del consumatore, succede che il consumatore riparte dalla sleep e finalmente si addormenta. Il problema grosso è che il count non tornerà mai più ad 1, almeno al momento in cui il produttore effettuerà ogni controllo, questo perchè la "soglia" è stata già superata da parte del produttore. La condizione non sarà mai più vera e il consumatore non si risveglierà mai più. Ad un certo punto inevitabilmente si riempie il buffer, e si addormenta anche il produttore. Cosa è successo? Tutti e due si sono addormentati e di conseguenza non possono più risvegliarsi tra di loro. Era necessario che almeno uno dei due fosse sveglio, per "controllare" l'altro.

Come risolvere il problema? Si può provare a metterci una pezza: la wakeup sarà chiamata ad ogni iterazione, indipendentemente dallo stato del buffer. Questo funziona, ma è inelegante e soprattutto inefficiente. Non bisogna chiamare syscall "a caso". Bisogna trovare quindi un modo per non far perdere la sveglia. Si usa un bit di attesa, che di default è a 0, quando una sveglia viene persa viene aggiornato ad 1. Quando uno qualunque dei due cerca di addormentarsi, controlla il valore del bit. Se è 1, c'è una sveglia disponibile, da "consumare": il processo non chiamerà la sleep, e aggiornerà il valore a 0. Anche in caso di prelazione, nessun problema. Bisogna però generalizzare al caso in cui ci sono più produttori e consumatori. In questo caso, un bit solo non basta, perchè potrebbero essere perse più sveglie.

## 2.7 Semafori

Un semaforo non è altro che la generalizzazione del bit di wakeup. Al posto di esso, c'è un contatore (variabile intera): l'obiettivo è quello di contare le sveglie perse, che poi verranno "consumate" dai vari processi. Il semaforo è implementato direttamente dall'OS, ed è una soluzione software. Il contatore ha una proprietà: non è mai negativo.

Per incrementare e decrementare il contatore, si usano due operazioni: **down** (o wait) e **up** (o signal). La wait possiamo associarla ad una sleep, e una signal possiamo associarla ad una wakeup. L'implementazione della down deve impedire che il contatore diventi negativo: per cui, se il contatore è già a 0, la down diventa una chiamata bloccante. Sarà sbloccata dalla up, che incrementa un attimo ad 1, ma poi down sbloccata e decremente di nuovo (era rimasta a quel punto). Per cui non verranno contate sveglie inesistenti.

Si ripresenta una race condition sul semaforo stesso: la variabile è condivisa. Se le operazioni down e up non sono atomiche, la famigerata prelazione potrebbe causare i soliti problemi. La proprietà più importante del semaforo (oltre al contatore mai negativo) è l'atomicità delle operazioni down e up. Cioè una mutua esclusione su di esse; verranno trattate come se fossero delle sezioni critiche. Si ritorna al punto di prima: come applicare mutua esclusione? Siamo nel kernel, potremmo utilizzare la soluzione di disabilitazione degli interrupt. Solito problema: e su sistemi multicore? Si prova ad adattare la soluzione con istruzione TSL/XCHG, rendendola atomica. Ad ogni semaforo sarà associata una variabile di lock, con delle enter e leave\_region. Questa soluzione aveva il problema dello spin lock come ricordiamo: possiamo tollerarlo? Sì,

perchè:

1. Wait e signal hanno codice piccolo, lo spin lock non durerà molto.
2. Non si verifica inversione di priorità, perchè semplicemente in questo contesto non esiste concetto di priorità.

In realtà, negli OS moderni, comunque viene implementata una soluzione senza spin lock. Dato che ci si trova a livello kernel, è possibile sfruttare le strutture dati kernel. Cioè, ad ogni semaforo sarà associata una coda dei processi bloccati da esso. Quando ci sono sveglie disponibili, viene estratto un processo da questa coda, e risvegliato (se ce ne sono).

### 2.7.1 Utilizzi dei semafori

- Per mutua esclusione: si parla di semaforo **mutex**. Associamo un semaforo ad una struttura dati, come se fosse una variabile di lock. Viene inizializzato ad 1. Perchè proprio 1? Lo vedremo tra poco. Intanto, attorno ad ogni sezione critica è necessario del codice preliminare (e finale); in pratica, una wait prima e una signal dopo.  
Inizialmente, la prima wait decrementa semaforo a 0. Se un altro processo arriva alla sua sezione critica, la wait diventerà bloccante per lui. Dopo la signal da parte del processo 1, il semaforo ritorna ad 1 e si sblocca il processo 2. Di conseguenza il semaforo ritorna a 0. In pratica: il semaforo vale 0 quando c'è qualcuno (le wait diventeranno bloccanti), altrimenti 1. Abbiamo compreso il motivo per cui è necessario inizializzarlo ad 1. Possiamo vedere ciò in maniera più generalizzata: il valore di inizializzazione indica quanti processi possono accedere in contemporanea alla struttura dati. Ovviamente per applicare mutua esclusione  $S = 1$ .
- Per sincronizzazione tra processi: torniamo al problema del produttore-consumatore. Abbiamo concepito il semaforo proprio come soluzione a questo problema. Prendiamo due piccioni con una fava: possiamo risolvere sia problema di sincronizzazione, che garantire mutua esclusione su buffer e variabili condivise.

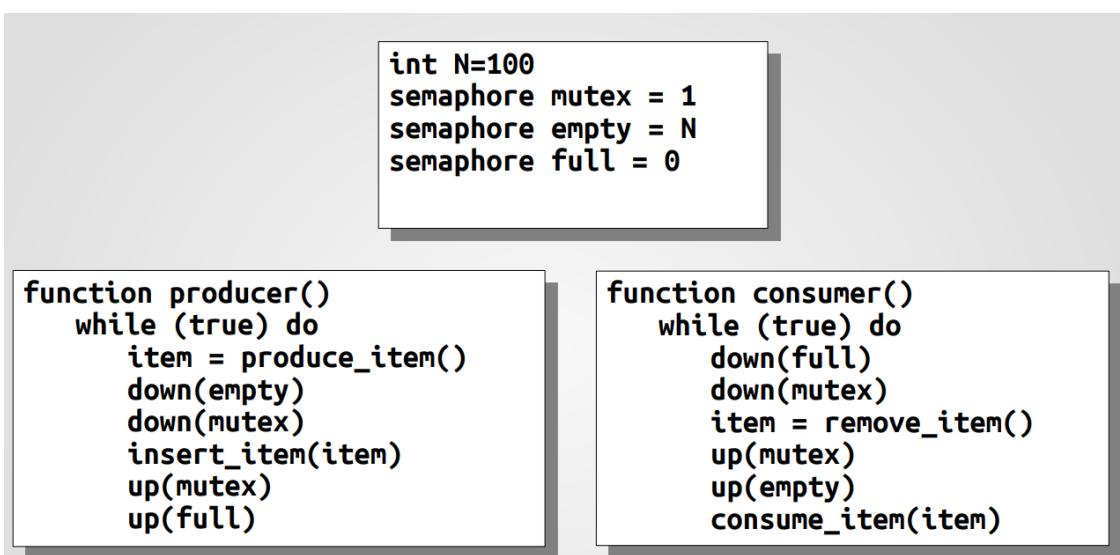


Figure 20: Produttore-consumatore con semafori.

Inizializziamo 3 semafori condivisi: `full` = 0, `empty` = N e `mutex` = 1. Come si può capire dai nomi, uno dei 3 serve per la mutua esclusione sul buffer. La sezione critica è compresa tra `down(mutex)` e `up(mutex)`. Gli altri due semafori fungono da contatori. In particolare, `empty` indica il numero di slot liberi nel buffer, mentre `full` indica il numero di slot occupati. Inizialmente il buffer è vuoto.

Il producer produce l'item, ed effettua `down(empty)`, prima anche dell'inserimento. E' come se "prenotasse" lo slot in cui effettuare l'inserimento. L'istruzione di inserimento è nella sezione critica, quindi chiama `down(mutex)`. Se il `mutex` è 0, il consumatore è già nella sezione critica, quindi producer si blocca. Se invece entra, effettua inserimento ed esce dalla sezione critica con `up(mutex)`. Adesso deve aggiornare il semaforo `full`, con chiamata `up(full)`, perchè adesso nel buffer c'è effettivamente un item in più.

Il consumer funziona in maniera molto simile, ovviamente chiamerà `down(full)` e `up(empty)`, perchè fa esattamente il contrario di ciò che fa il producer: rimuove un item. Anche qui, la sezione critica è composta soltanto dall'operazione `remove_item`.

I semafori riescono a risolvere i problemi di race conditions e di sincronizzazione tra processi grazie al fatto che riescono a rendere atomiche le operazioni di sleep e wakeup. Tutti e due i problemi erano infatti causati dalla originaria non atomicità di queste operazioni, per cui la prelazione poteva scattare in momenti non opportuni.

I semafori risolvono il problema del produttore-consumatore anche nel caso generale, con N produttori e M consumatori. Bisogna però notare una cosa: l'ordine delle operazioni è fondamentale. Perchè? Capiamolo con un esempio.

Prendiamo il produttore, ed invertiamo `down(empty)` con `down(mutex)`. Praticamente abbiamo incluso l'aggiornamento dei semafori contatori di risorse nella sezione critica. Nel caso di buffer pieno, il produttore entra nella sezione critica ancora prima di chiamare `down(empty)`. `Empty` = 0 perchè il buffer è pieno, e si blocca. Quando la "palla" passa al consumatore, esso proverà ad estrarre l'elemento, ma non potrà farlo, perchè impossibilitato ad accedere alla sezione critica (c'è già il produttore). Si blocca anche il consumatore, e c'è stallo. Lo stallo è causato dall'inclusione di `down(empty)` del produttore nella sua sezione critica. Se fosse stato al di fuori (come nella soluzione corretta), il consumatore avrebbe potuto rimuovere l'item in tutta comodità.

Da questo esempio possiamo capire anche cosa è giusto includere dentro una sezione critica: solo chiamate NON bloccanti. Non per forza l'inversione dell'ordine causerà problemi, ad esempio invertendo le due `up` non succede nulla, ma meglio evitare di includere nella sezione critica operazioni per cui non è utile garantire la mutua esclusione. Una sezione critica deve essere più piccola possibile.

## 2.7.2 Mutex con thread utente

Riprendiamo il discorso riguardo i thread implementati a livello utente (paragrafo 2.3.1). Ricordiamo come essi avessero problemi con chiamate bloccanti (bloccato intero processo, di conseguenza bloccati tutti i thread). Ma se avessimo necessità di utilizzare i semafori con thread utente, dovremo affrontare per forza il problema.

E' possibile applicare la soluzione con istruzioni TSL, ma senza spin lock. Trattandosi di thread fratelli, è possibile per un thread bloccarsi e cedere la CPU ad un suo fratello, in modo collaborativo, con una `thread_yield`.

```
mutex_lock:  
    TSL REGISTER,MUTEX  
    CMP REGISTER,#0  
    JZE ok  
    CALL thread_yield  
    JMP mutex_lock  
ok:RET
```

Figure 21: Mutex con thread utente

Dal codice di `mutex_lock` (simile ad `enter_region`) vediamo che in caso di confronto non andato a buon fine (false), la CPU viene ceduta e si salta all'inizio della procedura. Visto così può sembrare a prima vista spin lock, ma ricordiamo che la CPU è in mano ad un thread fratello. Non ci può essere attesa attiva del nostro thread, dato che la CPU non è più in mano sua. La `mutex_unlock` provvederà semplicemente ad aggiornare `MUTEX` a 0, come una qualunque `leave_region`.

### 2.7.3 Futex

L'uso dei mutex con thread utente può convenire per il fatto che non richiedono chiamate di sistema. Il problema è che non è possibile sfruttare gli strumenti messi a disposizione dall'OS. Linux risolve questo problema implementando dei mutex "ibridi": i **futex** (fast user space mutex). In pratica, quando bisogna usare i semafori su Linux, vengono in realtà usati i futex. Il futex ha due componenti, uno in user space, uno nel kernel. Il componente in user space è una libreria che usa una variabile di lock, gestita attraverso l'istruzione TSL, come un qualunque mutex in spazio utente. La particolarità del mutex è che in caso di alta contesa (molti thread coinvolti) viene invece effettuata la syscall al kernel, come un mutex in spazio kernel. Il thread viene bloccato passivamente, non c'è spin lock. Quindi evita uno dei problemi dei mutex in spazio utente: l'eccessiva durata dello spin lock, preservandone i punti forti (in caso di bassa contesa no syscall, più efficiente). I thread bloccati vengono messi in una particolare coda.

## 2.8 Monitor

Un semaforo è uno strumento abbastanza complesso da usare nella forma in cui è stato presentato. Per facilitare la vita al programmatore, è stato progettato uno strumento più ad alto livello (che svolge esattamente gli stessi compiti), che nasconde molti dettagli implementativi, il **monitor** (utilizzabile soltanto dai thread). Adesso osserveremo attentamente questi dettagli. Quindi, cos'è precisamente un monitor? Un monitor è un tipo di dato astratto implementato in alcuni linguaggi (quelli più moderni), è definito a livello utente. Viene implementato dal compilatore sfruttando strumenti kernel già esistenti, come semafori e mutex. Cioè è un oggetto caratterizzato da strutture dati (come quella da "proteggere") e metodi. Il monitor garantisce

la mutua esclusione nel codice dei metodi che contiene. Questi metodi servono ad accedere e manipolare le strutture dati interne citate prima; così si evitano sicuramente race conditions. Linguaggi di programmazione più vecchi e a basso livello come C, C++ e Pascal non implementano i monitor; uno strumento abbastanza simile, ma non identico, può essere usato in questi linguaggi attraverso la libreria Pthread. I linguaggi moderni (Java, C#, Go) implementano nativamente i monitor. In Java, la parola chiave "synchronized" nella signature di un metodo permette di applicare un monitor a quel metodo.

Ma come viene risolto dai monitor il problema della sincronizzazione dei thread (produttore-consumatore)? Dentro essi sono definite delle **variabili di condizione**, cioè delle "etichette" (una per ogni "evento", qualsiasi cosa si intenda) su cui si chiamano operazioni wait e signal. In pratica, se un thread si rende conto che deve bloccarsi, chiama la wait e aggiorna la variabile di condizione, bloccandosi. Ogni variabile di condizione ha ad essa associata una coda di thread bloccati a causa di quell'evento: quando un altro thread chiama la signal, un thread viene prelevato da questa coda e risvegliato. La coda in questione è implementata sfruttando un semaforo, "appoggiandosi" sulla sua coda. Quindi la coda dei thread bloccati di una variabile di condizione non è altro che la coda di un semaforo.

N.B. La variabile di condizione non ha un valore ben specifico, come era per la variabile "count" dei semafori. Questo perchè non è necessario "contare" le sveglie, dato che un monitor può essere usato da un thread alla volta.

I monitor così presentati hanno un problema. Esempio: T1 si blocca durante l'esecuzione del metodo monitor A. T3 esegue il metodo B, ad un certo punto chiama la signal, che potrebbe risvegliare T1. Se effettivamente si risveglia lui, riprende l'esecuzione di A da dove si era fermato. Sta qui il problema: due thread stanno eseguendo in contemporanea dei metodi interni del monitor, non rispettando la mutua esclusione, e si interlacciano istruzioni di metodi diversi. Questo si risolve definendo in maniera opportuna la semantica della signal. Sono state ideate nel tempo 3 diverse semantiche:

- Monitor Hoare (teorico): si può definire come un monitor di tipo signal & wait. Cioè quando T3 lancia la signal, prima si blocca lui, e solo dopo si risveglia T1. Quindi sul monitor ci sarà sempre e comunque un solo thread. Però: si continuano ad interlacciare istruzioni di metodi diversi. Ecco perchè è solo teorico.
- Monitor Mesa: implementazione reale del monitor in Java. Si può definire un monitor di tipo signal & continue. Quando T3 chiama la signal, T1 non si risveglia subito: si risveglia soltanto dopo la fine dell'esecuzione del metodo B da parte di T3. Ecco perchè si dice "continue". L'esecuzione di B da parte di T3 continua comunque fino al suo return.
- Signal & return: usato in concurrent Pascal, e anche negli pseudocodici di questo corso. Con questo approccio, semplicemente la signal sarà l'ultima istruzione ad essere eseguita in B prima del return. In pratica, quando si risveglia T1 il metodo B è già ritornato.

## 2.8.1 Produttore-consumatore con monitor

```
monitor pc_monitor
  condition full, empty;
  integer count = 0;

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

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

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

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

Figure 22: Pseudocodice

I codici di produttore e consumatore sono molto semplici. Il produttore produce, ed inserisce l'item nel buffer, chiamando uno dei metodi del monitor. Il consumer prima rimuove l'item dal buffer, sempre con un metodo monitor, e poi consuma.

Dentro il monitor è definita la solita variabile count, e due variabili di condizione, full ed empty. Queste variabili saranno aggiornate sulla base del valore di count.

Come possiamo vedere, infatti, in insert se count = N (buffer pieno) il producer si deve bloccare, quindi chiama la wait sulla variabile di condizione full. Se count = 1, chiama la signal su empty facendo sbloccare il consumer. In pratica la stessa cosa che succedeva con la soluzione con i semafori, ma qui il codice è molto più semplice e comprensibile. Il consumer fa la stessa identica cosa, ma al contrario. Se count = 0, il buffer è vuoto, quindi si blocca con wait su empty. Se count = N-1, cioè il buffer non è più pieno, risveglia il producer con la signal su full. Al momento del risveglio di uno dei due, non è necessario ricontrrollare la condizione, è sicuramente falsa a questo punto. In soluzioni reali, però, si tende a ricontrrollarla. La soluzione qui presente si può applicare con tutte le semantiche viste prima, potrebbe dare problemi soltanto con la semantica del concurrent Pascal (signal & return).

## 2.9 InterProcess Communication

Adesso possiamo finalmente vedere come comunicano tra loro due processi. Solitamente due processi sulla stessa macchina comunicano con un modello a messaggi, lo stesso applicato nel caso di comunicazioni tra macchine diverse (Reti). Si capisce già che questo modello si può generalizzare a tutti e due i casi (nel secondo caso si sfrutta libreria MPI, magari nel caso di un cluster di server).

Si sfruttano (ovviamente) delle code, dette con molta fantasia code di messaggi, che permet-

tono la sincronizzazione tra sender e receiver. Possiamo vederli come se fossero produttore e consumatore.

```
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)
    consum_item(item)
```

Figure 23: Pseudocodice IPC

Come in quell'esempio, il sender "produce" un item, e lo deve inviare al receiver, che lo consumerà. Il sender invia il messaggio al receiver con la chiamata send. Finché il messaggio non arriva, il receiver deve rimanere bloccato, cioè la chiamata receive diventa bloccante. Quando la coda dei messaggi si satura, la chiamata send diventa anch'essa bloccante: il sender non può inviare altro. Il modello a messaggi è utile perché permette la comunicazione in assenza di memoria condivisa, con il kernel che fa da "postino". La coda dei messaggi serve per stipare i messaggi quando il receiver non è dentro la chiamata receive (non può ricevere in quel momento).

Per generalizzare questa soluzione ad N sender e M receiver si usa il metodo di indirizzamento a mailbox. Il destinatario della send, e a sua volta il mittente della receive non saranno più receiver/sender, ma una o più mailbox intermedie. Il problema di questa soluzione è la poca efficienza. Per mettere/prelevare il messaggio dalla mailbox bisogna effettuare delle syscall, che nel caso del prelievo deve anche cercare il messaggio dentro la mailbox.

## 2.10 Accesso esclusivo da parte di processi concorrenti: il problema dei 5 filosofi

Il problema dei 5 filosofi è un altro di quei problemi che prima abbiamo chiamato "giocattolo": è una situazione astratta ma che ci può essere di aiuto per capire come risolvere problemi reali. In questo caso, si tratta di capire come modellare l'accesso esclusivo (temporaneo, la risorsa viene acquisita e poi rilasciata) a delle risorse da parte di processi che lavorano in concorrenza, in questo caso 5, in generale N.

In che situazione ci troviamo? Ci sono 5 filosofi, seduti a tavola, con davanti il loro pranzo. Essendo filosofi, stanno seduti a pensare, ma ad un certo punto qualcuno di loro, sopraffatto dalla fame, può decidere di smettere di pensare ed iniziare a mangiare. Qual è il problema? Ogni filosofo si ritroverà una forchetta alla sua sinistra e una alla sua destra. Stranamente, per mangiare, avrà bisogno di usare tutte e due le forchette; ma la forchetta sinistra del filosofo i-esimo è allo stesso tempo la forchetta destra del filosofo (i-1)-esimo, e la forchetta destra del filosofo i-esimo è la forchetta sinistra del filosofo (i+1)-esimo.

Come possono fare a sincronizzarsi tra loro? E' importante precisare che più filosofi possono mangiare in contemporanea, basta che non siano adiacenti (quindi non condividono forchette).

```

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)

```

Figure 24: Pseudocodice della prima soluzione

- Questa soluzione usa delle variabili di lock. Ogni filosofo ha una sua funzione, che è un ciclo infinito di "pensare" e "mangiare". Prima pensa, ad un certo punto decide di prendere le due forchette e mangiare, poi le posa, ipotizzando l'esistenza di apposite funzioni che eseguono queste azioni. Ad ogni forchetta è associata una variabile di lock. Se il lock è attivo, significa che la forchetta è stata già presa da qualcuno, la take\_fork diventa bloccante. La put\_fork su quella forchetta disattiverà il lock. Questa soluzione sembra funzionare nella sua semplicità, ma ha un problema: se tutti e 5 decidessero in contemporanea di prendere la forchetta alla loro destra, ognuno di loro non avrebbe la possibilità di prendere anche quella alla loro sinistra (il filosofo alla sua sinistra gliela ha già presa), finendo per bloccarsi tutti e 5.
- Il problema si risolve facendo in modo che se l'i-esimo filosofo riesce ad afferrare una sola forchetta, la rilascerà. Quindi: o le prende tutte e due, o non ne prende nessuna. Dopo un tot di tempo fisso, ogni filosofo che ha fallito riproverà a prendere tutte e due le forchette. Ma qui si verifica un altro problema: se i filosofi decidono di mangiare in contemporanea (come prima), potrebbe capitare che rimangano tutti e 5 senza forchette. Si bloccano comunque, perché riprovano sempre dopo la stessa quantità di tempo.
- A questo punto si può decidere di rendere questo tempo random, quindi diverso per ogni filosofo. Questo effettivamente funziona. Ma pensandoci, non è necessario arrivare a tal punto, perdendo molto in termini di efficienza. C'è una soluzione migliore.
- Prima di vedere la soluzione migliore, proviamo a fare un'ipotesi. Se il filosofo i-esimo che vuole mangiare metta il lock su tutto il tavolo? Ogni problema è risolto, c'è un però gigantesco: mangerebbe un solo filosofo alla volta!
- Ecco la soluzione migliore (generale): usiamo dei semafori mutex.

```

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

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

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

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

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

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

```

Figure 25: Pseudocodice con mutex

Abbiamo 3 stati: THINKING, HUNGRY, EATING. Lo stato HUNGRY è uno stato intermedio, che indica la volontà del filosofo di mangiare, però non potendolo ancora fare. Per monitorare lo stato di ogni filosofo usiamo un vettore "state" che memorizza lo stato del filosofo  $i$ -esimo nella posizione  $i$ . Poi un semaforo mutex, e un vettore di semafori, sempre uno per ogni filosofo. Questo vettore serve per far bloccare l' $i$ -esimo filosofo quando necessario.

Le funzioni sono `take_forks` e `put_forks`, che a differenza delle precedenti fanno afferrare direttamente tutte e due le forchette, per i motivi spiegati sopra. Cioè: se ritornano, il filosofo ha preso/rilasciato tutte e due le forchette. Il semaforo mutex serve per proteggere la sezione critica in cui il filosofo  $i$ -esimo cambia stato (dentro `take_forks` e `put_forks`). Analizziamo la `take_forks`. Come abbiamo detto, il mutex protegge la fase di cambiamento di stato (da THINKING ad HUNGRY). Dopo, il filosofo chiama la funzione `test` (mai bloccante), che verifica se effettivamente esso possa prendere tutte e due le forchette. Se può, cambia lo stato in EATING (non dimenticandosi di aggiornare il semaforo), e può effettivamente mangiare. La `test` deve verificare delle condizioni per poter capire se il filosofo può afferrare le forchette. Cioè: se il filosofo alla sua sinistra NON sta mangiando, e neanche quello alla sua destra, le forchette sono libere, e può prenderle. A cosa serve la prima condizione, all'apparenza sempre vera? Lo vedremo poi. Ora: perchè la `up` del semaforo è dentro la `test` (e dentro la sezione critica), ma la `down` è fuori? Semplicemente perchè la `down` può diventare bloccante (quando il filosofo non riesce a prendere le forchette, e serve proprio a questo) e non bisogna mai mettere una chiamata bloccante dentro una sezione critica, abbiamo già visto cosa succede.

Adesso analizziamo la `put_forks`. Il filosofo finisce di mangiare, e vuole tornare a pensare. Chiama la `put_forks` (protetta dal mutex), che gli cambia lo stato in THINKING. Adesso, una cosa molto importante. Deve chiamare la `test` sui filosofi accanto a lui, per farli risvegliare se necessario. Come fa? Qui entra in gioco la prima condizione della

test: se il filosofo in questione è HUNGRY, quindi vuole mangiare, bisogna risvegliarlo e permettergli di mangiare (ho rilasciato le forchette, adesso sono disponibili). Altrimenti, significa che ancora sta pensando, e non è necessario il risveglio.

### 2.10.1 Problema dei 5 filosofi con monitor

```

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

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

    function take_forks(int i)
        state[i] = HUNGRY
        test(i)
        if state[i] != EATING
            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)
    
```

Figure 26: Pseudocodice con monitor

La struttura dati da "proteggere" è il vettore degli stati, ricordate? Lo facciamo "proteggere" dal monitor. Cioè il vettore state è dentro il monitor. Di conseguenza, i metodi take\_forks, put\_forks e test sono metodi del monitor. Al posto di avere 1 semaforo per ogni filosofo, qui abbiamo una variabile di condizione (che non hanno valore) per ogni filosofo. Il semaforo mutex di prima non è più necessario, perchè il monitor garantisce mutua esclusione, e gli altri N semafori sostituiti da un vettore di N variabili di condizione. Per il resto, il codice è praticamente uguale, si comporta alla stessa maniera (il filosofo, quando deve bloccarsi o sbloccarsi/sbloccare altri chiama signal e wait su sua variabile di condizione/quella dell'altro per cui ha fatto la chiamata, al posto di fare down e up su semafori. Ma è la stessa cosa in pratica). Dato che però le variabili di condizione non hanno "memoria" (non contengono valori al loro interno) prima di afferrare le forchette bisogna controllare che effettivamente lo stato del filosofo i-esimo sia "EATING", altrimenti si blocca.

## 2.11 Accesso ad un database: problema dei lettori e scrittori

Il terzo problema "giocattolo" modella l'accesso in lettura e scrittura ad un database (o in generale una struttura dati) differenziando le due operazioni. Questo perchè N operazioni di lettura contemporanee sono tollerate (non si modifica nulla), mentre operazioni di lettura e scrittura oppure operazioni soltanto di scrittura devono essere fatte in modo esclusivo, per evitare di "sporcare" i dati.

N.B. Nella libreria Pthread questa cosa è già implementata, con un lock particolare detto proprio lock lettore/scrittore.

Utilizzando un mutex nel modo che già sappiamo costringeremmo anche i lettori ad accesso esclusivo, non necessaria qualcosa così restrittiva. Le operazioni di lettura contemporanee possono tranquillamente coesistere tra loro. Cosa fare allora?

### 2.11.1 Soluzione con semaforo mutex

```
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()

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

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

Figure 27: Pseudocodice soluzione con mutex

Il fulcro di questa soluzione consiste nel vedere i lettori come un gruppo unico. Cioè: si garantisce l'accesso esclusivo o al GRUPPO di lettori, o ad un solo scrittore. Per cui usiamo una variabile "rc" che indica il numero di componenti del gruppo di lettori. Il semaforo mutex protegge questa variabile, l'altro protegge il database, ossia fa accedere ad esso soltanto una delle due entità (gruppo di lettori, scrittore).

Cosa fa il lettore? Prima di leggere, incrementa il valore di rc, per indicare che sta entrando anche lui a far parte del gruppo di lettori. Se  $rc = 1$  (e quindi prima era 0) significa che è il primo lettore ad entrare. Deve "chiudere la porta" agli eventuali scrittori che accorreranno, con una `down(db)`. Questo è ovviamente un compito soltanto del primo lettore, ecco spiegata la condizione. Dopo la lettura del database, decrementa rc (abbandona il gruppo dei lettori) e se è l'ultimo del gruppo ( $rc$  diventa 0), significa che il database sarà libero dopo la sua uscita. Quindi "riapre la porta" agli scrittori con una `up(db)`. Verrà risvegliato soltanto uno scrittore in pratica, tanto può accedere soltanto lui, non ci possono essere più scrittori nello stesso momento.

Cosa fa lo scrittore? Entra, blocca l'accesso a chiunque ci sia fuori, che siano altri scrittori o il gruppo di lettori, scrive, e poi sblocca l'accesso (sveglia o un lettore o uno scrittore a caso). Se sveglia un lettore, quel lettore, con `up(mutex)`, sveglierà gli altri in massa (sono bloccati dal `down` su `mutex`, perchè solo uno alla volta può aggiornare la variabile). Con questo metodo i lettori vengono effettivamente visti come un tutt'uno. Per questo `down(db)` nel lettore è volutamente dentro una sezione critica. Se un lettore si blocca a causa della presenza di uno scrittore, devono bloccarsi anche tutti gli altri, e se si sblocca un lettore, devono sbloccarsi tutti gli altri.

La soluzione funziona, ma ha un problema. Implementata così, possiamo dire che dia priorità al gruppo dei lettori, perchè uno scrittore non può entrare finchè il gruppo non si svuota tutto.

E quando uno scrittore esce e c'è un gruppo di lettori in attesa, sbloccherà loro, non un altro scrittore. Quindi ogni scrittore potrebbe rimanere bloccato per un tempo indefinito, non gli si garantisce nulla. La soluzione più ovvia sarebbe un timeout per ogni gruppo di lettori, dopo il quale il gruppo deve abbandonare necessariamente il database.

## 2.11.2 Soluzione 1 con monitor

```

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()

```

Figure 28: Pseudocodice soluzione 1 con monitor

Il database non è inglobato dentro il monitor, perchè altrimenti sarebbe impedito l'accesso contemporaneo ai lettori. Il monitor protegge soltanto le sezioni critiche protette prima dal mutex, invece la mutua esclusione dal punto di vista degli scrittori viene garantita da variabile `busy_on_write`, che sarà `true` soltanto nel caso in cui uno scrittore sia dentro il database.

I metodi del monitor spezzettano le fasi di lettura e scrittura in due parti (le due sezioni critiche di prima, qua si capisce che il monitor fa le veci del mutex). Quando un lettore vuole entrare (`start_read`) deve controllare la variabile `busy_on_write`, per vedere se già c'è uno scrittore. Se non c'è, incrementa la variabile `rc` ed entra nel gruppo. Dopo la lettura (`end_read`), decrementa `rc` e, se è l'ultimo del gruppo, risveglia uno degli scrittori, come prima. Ovviamente qui si tratta di un monitor, quindi si usano operazioni `signal` sulle variabili di condizione.

A cosa serve la `signal(read)` in `start_read`? Se il lettore in questione è il primo del gruppo, risvegliato dallo scrittore appena fuoruscito, deve risvegliare in massa gli altri.

Quando entra uno scrittore (`start_write`) sia se c'è già il gruppo dei lettori, sia se c'è uno scrittore (`busy_on_write`), si blocca. Altrimenti entra e mette a `busy_on_write` a `true`, per bloccare chiunque altro. Dopo la scrittura (`end_write`) `busy_on_write` viene rimesso a `false`, e adesso, se c'è un gruppo di lettori in attesa, ne risveglia uno (che risveglia gli altri nel modo che abbiamo visto sopra) altrimenti risveglia un suo "collega". Possiamo vedere che questa soluzione privilegia molto i lettori; addirittura ogni scrittore cerca prima di risvegliare i lettori rispetto agli altri scrittori. A differenza della soluzione con semafori, questa è una scelta voluta; prima veniva scelto "a caso".

### 2.11.3 Soluzione 2 con monitor

```

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

function start_read()
    if (busy_on_write OR in_queue(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()
    
```

Figure 29: Pseudocodice soluzione 2 con monitor

Vediamo che è identica alla prima, solo per un piccolo ma importante dettaglio: un qualsiasi lettore, prima di entrare, controlla ANCHE se ci siano scrittori in attesa (condizione in rosso). Se ci sono, si blocca e "lascia spazio" ad uno di essi. Questo permette agli scrittori di essere meno "discriminati". Soluzione più equilibrata rispetto alla prima.

Vediamo un esempio di funzionamento: si verificherà una situazione in cui c'è un gruppo di lettori R1 dentro il DB, degli scrittori in attesa e un altro gruppo di lettori R2 in attesa. Quando R1 si esaurisce, il primo lettore di R2 lascia spazio ad uno degli scrittori, che dopo sbloccherà uno dei lettori di R2, che sbloccherà tutti gli altri.

### 2.11.4 Soluzione 3 con monitor

```

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

function start_read()
    if (busy_on_write OR in_queue(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(write))
        signal(write)
    else
        signal(read)
    
```

```

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()
    
```

Figure 30: Pseudocodice soluzione 3 con monitor

La soluzione 3 ricalca la 2, però in questo caso gli scrittori fanno un po' i furbi. Decidono di favorire i loro simili, cioè: nella `end_write`, quando lo scrittore deve scegliere chi sbloccare, decide di dare la priorità agli altri scrittori. Controlla prima se ci sono scrittori bloccati, e soltanto se non ce ne sono sblocca il gruppo dei lettori.

Notare che ognuna di queste soluzioni soffre della "starvation", cioè quel problema spiegato prima per cui lo scrittore non sa quanto aspetterà.

## 2.12 Scheduling

Molto tempo fa avevamo dato per scontata la fase di scheduling di un processo. Ma in cosa consiste? Consiste nell'atto di scegliere quale processo deve usare la CPU, quando questa si libera. Dobbiamo capire in base a quali criteri viene fatta la scelta. Specifichiamo che ovviamente il processo viene scelto tra quelli pronti. Se abbiamo una normale coda FIFO, il processo viene già scelto in maniera implicita. In realtà viene scelto eseguendo un algoritmo di scheduling. Dopo che l'algoritmo effettua la scelta, interviene il **dispatcher**, che si occupa fisicamente di ripristinare lo stato del processo scelto, in modo che possa utilizzare la CPU e ripartire da dove si era fermato. Ovviamente tutto questo ha un costo non da sottovalutare (latenza di dispatch), che varia in base al contesto (thread fratelli o processi completamente diversi). Lo scheduler deve scegliere in modo che la CPU sia sempre occupata.

Possiamo distinguere i processi in due tipi:

- CPU bounded, i processi che useranno soprattutto la CPU;
- I/O bounded, i processi che useranno poco la CPU, e faranno soprattutto richieste di I/O.

Quindi come li possiamo distinguere? I processi CPU bounded sono quelli che hanno lunghi tempi di **CPU burst** (utilizzo CPU), quelli I/O bounded hanno CPU burst di durata inferiore.

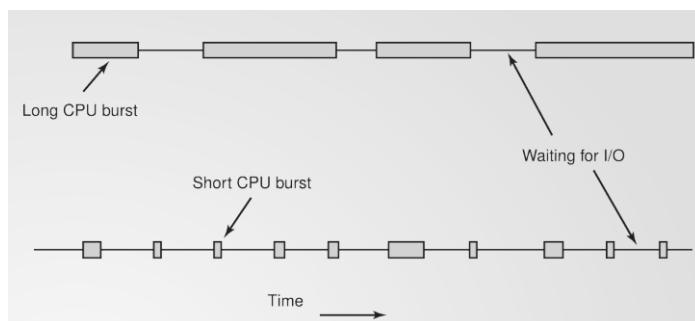


Figure 31: Sopra un processo CPU bounded, sotto un processo I/O bounded.

Un processo può cambiare comportamento nel corso dell'esecuzione. Un compilatore, ad esempio, in alcune fasi sarà CPU bounded, in altre sarà I/O bounded. Col passare del tempo, i processi diventano mediamente sempre più I/O bounded. Perchè? Semplicemente perchè più le CPU diventano potenti, meno dureranno le fasi di CPU burst, a parità di task. Questa distinzione diventa importante perchè dà allo scheduler un criterio su cui fare la scelta.

Allo scheduler conviene mischiare processi di tutti e due i tipi, e scegliere per primi processi I/O bounded. Questi processi tendono a cedere dopo poco tempo la CPU, quindi il comparto I/O avrà sempre qualcosa da fare, e non incide sulle capacità computazionali della CPU. L'unico problema è effettivamente distinguere.

Lo scheduler si attiva in generale quando la CPU si libera (processo terminato, prelazione nei sistemi in cui è presente, chiamata bloccante), o in caso di creazione di un nuovo processo. E' importante che lo scheduler faccia velocemente la sua scelta, il tempo di scelta è overhead aggiuntivo.

### 2.12.1 Obiettivi di un algoritmo di scheduling

La definizione degli obiettivi di un algoritmo di scheduling dipende dall'ambiente in cui ci si trova. Se ci si trova in sistemi batch (come in questo corso), dobbiamo tenere conto del fatto che i processi batch non sono interattivi, sono delle routine, per cui fanno poco I/O, con poche chiamate bloccanti.

In generale, un algoritmo di scheduling deve essere **equo**. Cioè non deve privilegiare processi rispetto ad altri, almeno nel caso in cui non ci siano priorità o cose simili. Un'altra cosa importante è il bilanciamento nell'assegnazione delle risorse, ad esempio, con sistemi multicore, l'algoritmo deve spartire le risorse equamente anche tra i vari core; non ci devono essere core molto sotto sforzo, e altri praticamente a riposo.

Nei sistemi batch, solitamente bisogna tenere conto delle metriche. Metriche principali:

- throughput: misura il numero di task completati per singola unità di tempo. Un algoritmo di scheduling deve massimizzarlo. Usarlo come unica metrica non conviene; in questo caso lo scheduler darebbe priorità ai task piccoli, lasciando per ultimi quelli più grandi, causando rallentamenti.
- Tempo di turnaround: bisogna introdurre anche questa metrica. Il tempo di turnaround è il tempo che passa tra l'inserimento del processo tra i processi pronti, e il suo completamento. Bisogna minimizzarlo.
- Tempo di attesa: è una parte del tempo di turnaround: è il tempo in cui il processo rimane pronto senza essere eseguito. Questa metrica è la più importante, perché dipende direttamente dall'algoritmo usato, mentre il resto (tempo di esecuzione) dipende dalla CPU. Bisogna minimizzare soprattutto questo tempo, che di conseguenza minimizza il tempo di turnaround. Nell'esempio fatto nella spiegazione del throughput, il problema è che ci si trovava in una situazione con throughput alto, ma con tempo di attesa alto. Per l'algoritmo è importante misurare tempo di attesa medio, per poter fare scelte ponderate.

In un sistema interattivo, gli obiettivi sono per forza di cose diversi, proprio per la natura del sistema. Alcune delle considerazioni precedenti non contano. Un sistema interattivo si basa molto sull'input esterno, quindi la **reattività** è molto importante. Cioè, deve ridurre al minimo il tempo di risposta su una richiesta, ad esempio quando si clicca su un collegamento, l'applicazione deve aprirsi velocemente. Lo scheduler di un sistema interattivo privilegerà i processi più I/O bounded. Questi processi rilasceranno presto la CPU, quindi il processo successivo

sarà schedulato più velocemente.

I sistemi real-time per loro natura hanno delle scadenze specifiche da rispettare ad ogni costo, e le scelte dello scheduler devono essere prevedibili, per eliminare il più possibile ritardi. Si usano degli scheduler fatti ad-hoc.

### 2.12.2 Algoritmi di scheduling per sistemi batch

Nei sistemi batch, le operazioni di I/O da parte dei processi sono ridotte al minimo. Daremo per scontato che un processo, una volta schedulato, utilizzerà la CPU fino al suo completamento. Ci sono diversi algoritmi di scheduling per sistemi batch:

- **FCFS**: sta per First-Come First-Served, in pratica non aggiunge nulla al funzionamento tipico di una coda FIFO. Lo scheduler semplicemente effettua un'estrazione in testa quindi pesca dalla coda il processo che è stato inserito per primo tra quelli presenti, senza considerare altro.

Esempio:

*Esempio*

<u>Processo</u>	<u>Durata</u>
P <sub>1</sub>	24
P <sub>2</sub>	3
P <sub>3</sub>	3

$$\begin{aligned} t.m.a. &: (0+24+27)/3 = 17 \\ t.m.c. &: (24+27+30)/3 = 27 \end{aligned}$$

Figure 32: Esempio 1, FCFS

Abbiamo 3 processi, schedulati esattamente in questo ordine. Prima P1, poi P2 e infine P3, perchè così sono stati inseriti nella coda; all'istante 0 sono tutti e 3 già nella coda. Sapendo l'ordine, e sapendo la loro durata, possiamo calcolare le metriche:

- Per P1: tempo di attesa 0 (viene schedulato per primo, all'inizio), tempo di completamento 24 (la sua durata);
- Per P2: tempo di attesa 24 (schedulato dopo la fine di P1), tempo di completamento 27 (dura 3);
- Per P3: tempo di attesa 27, tempo di completamento 30.

Abbiamo capito che per calcolarle è molto importante tenere conto dell'istante in cui il processo viene inserito nella coda. Ribadiamo che in FCFS non c'è prelazione.

Le metriche in questione dipendono molto dall'ordine in cui vengono schedulati i processi. Lo vedremo adesso.

- **SJF**: sta per Shortest Job First. Non si implementa una logica FIFO, o meglio, sì, però la coda viene riordinata (con confronto) per durata crescente. Cioè, i processi più "corti"

vengono scelti per primi. Quindi, se i processi hanno tutti la stessa durata, SJF diventa praticamente un algoritmo FIFO. Possiamo vedere che applicando SJF all'esempio di prima le metriche migliorano.

Adesso è naturale chiedersi, come si fa a conoscere la durata dei processi? Un sistema batch è un sistema chiuso, non interattivo. I processi che si susseguono sono tutti delle stesse tipologie, e si ripresentano di continuo, quindi si può prevedere con ragionevole precisione la loro durata. Se così non fosse, naturalmente non si potrebbe usare SJF.

Ma perchè SJF funziona (abbastanza) bene? Abbiamo detto che la coda viene riordinata, dal processo più corto al più lungo. La cosa importante da tenere in considerazione è, che quando un processo lungo si scambia di posto con uno corto, le metriche cambiano. Il tempo di attesa del processo corto migliora, e peggiora quello del processo lungo. Ma: Il tempo di attesa del processo corto migliora PIU' di quanto peggiora il tempo di attesa del processo lungo. Quindi nell'insieme il tempo di attesa migliora.

C'è un problemino: tutto questo vale solo nel caso in cui i processi siano inseriti nella coda tutti all'istante 0. Se i vari processi vengono inseriti in istanti diversi, SJF non è più ottimale.

Esempio:

*Esempio  
SJF non è ottimale*

<u>Processo</u>	<u>Arrivo</u>	<u>Durata</u>
P <sub>1</sub>	0	2
P <sub>2</sub>	0	4
P <sub>3</sub>	3	1
P <sub>4</sub>	3	1
P <sub>5</sub>	3	1
		t.m.a.
		SJF $(0+2+3+4+5)/5 = 2.8$
		altern. $(7+0+1+2+3)/5 = 2.6$

Figure 33: Esempio 2, SJF

Notiamo che i processi P<sub>3</sub>, P<sub>4</sub> e P<sub>5</sub> vengono inseriti nella coda all'istante 3. Cosa comporta? All'inizio, lo scheduler deve scegliere tra P<sub>1</sub> e P<sub>2</sub>. Sceglie P<sub>1</sub>, perchè dura di meno. All'istante 2 finisce P<sub>1</sub>, lo scheduler sceglie P<sub>2</sub>, l'unico rimanente. Appena finisce P<sub>2</sub>, all'istante 6, nella coda ci saranno gli altri 3 nuovi processi, che hanno la stessa durata quindi presi in ordine. Il tempo di attesa di ognuno dovrà tenere conto di quando esso è entrato nella coda.

Cioè: P<sub>1</sub> ha tempo di attesa 0, P<sub>2</sub> ha tempo di attesa 2, P<sub>3</sub> ha tempo di attesa 3 (entra ad istante 3, schedulato ad istante 6), P<sub>4</sub> ha tempo di attesa 4 e P<sub>5</sub> ha tempo di attesa 5.

Calcolando tempo di attesa medio, vediamo che risulta 2,8. Ma possiamo accorgerci che ordinandoli inversamente (schedulando prima quello più lungo) il tempo di attesa medio diminuisce, precisamente a 2,6. Vediamo che SJF **non è ottimale**.

- **SRTN**: sta per Shortest Remaining Time Next. In pratica, SJF ma con prelazione. Come funziona? Effettua delle valutazioni "in itinere". Cioè viene eseguito ogni volta

che un nuovo processo viene inserito in coda. Se c'è un processo in coda che dura meno del tempo residuo del processo che attualmente sta usando la CPU, entra in gioco la prelazione, il processo attuale viene "buttato fuori", e viene schedulato quello nuovo. Nell'esempio precedente, all'inizio tutto come prima. Ma all'istante 3, quando entrano in scena P3, P4, P5, lo scheduler viene richiamato. Qual è il processo che dura di meno? P2 ha durata residua 2, i nuovi processi hanno durata 1. Interviene prelazione su P2, e gli passano davanti P3, P4, e P5. Dopo, rimane soltanto P2, che viene rischedulato e completato. Ovviamamente, non è che P2 viene lasciato SEMPRE per ultimo: il processo "cacciato" dalla prelazione viene rimesso nella coda, con durata pari al suo tempo residuo, e trattato come chiunque altro. Ricalcolando il tempo di attesa medio vediamo che si abbassa considerevolmente: 1,6. Dobbiamo tener conto che i tempi di attesa dei processi che subiscono la prelazione devono essere sommati.

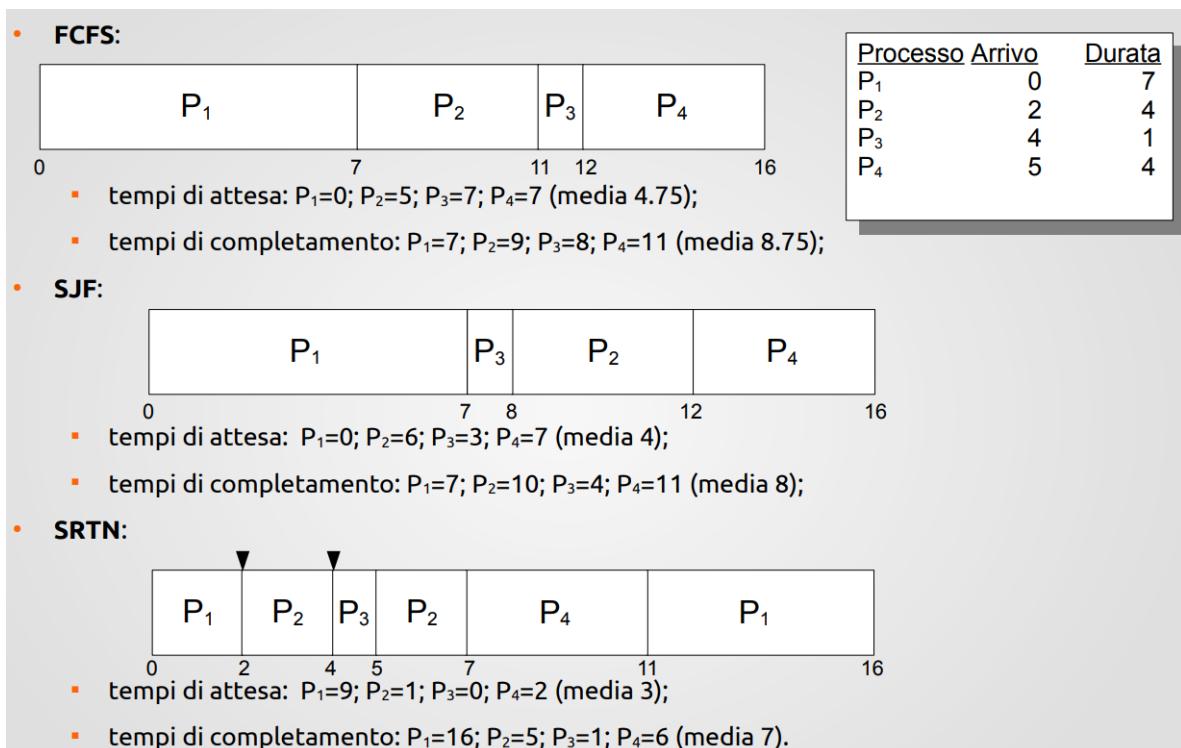


Figure 34: Esercizi su scheduling batch

### 2.12.3 Algoritmi di scheduling per sistemi interattivi

Come dicevamo, nei sistemi interattivi conta soprattutto la reattività del sistema, per cui le metriche di prima perdono di senso, e non saranno più utilizzate. In questi casi dobbiamo tenere in considerazione anche la presenza della prelazione, quindi l'interlacciamento tra i processi; non possiamo prevedere la durata di ogni processo. Gli algoritmi precedenti sono perfettamente inutili se applicati da soli, ma come vedremo avranno comunque un ruolo importante.

I processi I/O bounded si bloccano rapidamente e spontaneamente, mentre quelli CPU bounded potrebbero monopolizzarla, avendo burst molto lunghi. Per cui si viene a creare un effetto "convoglio", in cui i processi I/O bounded si accodano dietro quelli CPU bounded. Senza la prelazione, ci sarebbe un bloccaggio perenne a causa di questi ultimi.

## 2.12.4 Round-Robin

L'algoritmo di scheduling Round-Robin funziona esattamente come FCFS per sistemi batch, ma con la presenza della prelazione. Si sceglie sempre il processo in testa alla coda, ma gli si assegna un certo quantitativo di tempo ("quanto") predefinito; allo scadere del quanto, la prelazione toglie la CPU, e la dà al processo appena successivo, che ora è in testa. Dove va a finire il processo "cacciato"? Torna nella coda dei processi pronti, all'ultimo posto, e sarà rischedulato solo quando sarà di nuovo il suo turno, come un qualunque altro processo.

Se il processo effettua una chiamata bloccante, usa il suo quanto solo parzialmente. Viene intanto messo nella coda dei processi bloccati, dopo che il blocco termina viene rimesso nella coda dei processi pronti. Quando viene rischedulato, gli viene riassegnato un nuovo quanto " pieno", non avrebbe senso fargli consumare il quanto rimanente da prima. Altre implementazioni del Round-Robin potrebbero posizionare l'ex-bloccato in testa, in modo da fargli "finire il lavoro" il prima possibile.

Possiamo notare una bella cosa: i processi CPU bounded sono prelazionati più di frequente, perché hanno CPU burst lunghi, solitamente più lunghi del quanto. I processi I/O bounded sono prelazionati di meno. A cosa ci serve questa informazione? Abbiamo un modo per riconoscere e distinguere processi I/O bounded da processi CPU bounded.

Il vantaggio della prelazione è l'assenza certa di starvation: nessuno rimarrà ad aspettare un tempo indefinito, e si possono dare garanzie ad ogni processo. In pratica, ogni processo sa che entro un tot di tempo avrà la CPU per sè. A quanto corrisponde questo "tot di tempo"? Assumendo di avere n processi e che il quanto duri tempo q, arriviamo alla conclusione che un processo qualunque avrà la garanzia di avere la CPU entro  $(n - 1)q$  ms, nel peggior dei casi (appena prelazionato, ci sono n-1 processi davanti a lui nella coda).

Si può capire che quanti più piccoli permettono di avere un sistema più reattivo, perché il tempo di risposta è basso (un nuovo processo viene schedulato in meno tempo, la CPU viene passata di mano più velocemente). L'unico problema è che aumenta il numero di context-switch, ciò comporta un aumento dell'overhead (ogni context-switch ha un overhead intrinseco). In pratica, il singolo CPU burst viene "spezzettato" di più.

Bisogna tenere in mente una cosa, che può sembrare abbastanza ovvia: se il quanto diventa più grande di ogni CPU burst, è come se scomparisse la prelazione.

## 2.12.5 Scheduling con priorità

La realtà è più brutta di quanto si pensi, e nella realtà non si può applicare Round-Robin così come l'abbiamo presentato noi, che dà per scontato che i processi siano tutti "uguali". La realtà ci dice che i processi hanno varie **priorità**. Cosa si intende per priorità? E' semplicemente un'etichettatura numerica. Nel corso solitamente parleremo genericamente di alta e bassa priorità. Le priorità possono essere assegnate **staticamente**, appena il processo viene inserito nella coda, eseguendo indicazioni esterne (es. processi di utente X hanno priorità alta), oppure **dinamicamente**, cambiandolo durante la sua permanenza nella coda, questo accade solitamente quando ci sono indicazioni interne. Come si può facilmente intuire, viene schedulato per primo il processo con priorità più alta.

La prelazione nello scheduling con priorità viene applicata in un modo particolare: quando un processo viene inserito nella coda, viene confrontata la sua priorità con quella del processo in esecuzione: se è più alta, viene effettuata prelazione. Nel caso di priorità assegnata dinamicamente, si può cambiare la priorità del processo in base alla sua durata, con la relazione  $1/durata$ . Si dà così priorità ai processi più corti. Se non ci fosse la prelazione, avremmo praticamente SJF; così come presentato, è praticamente SRTN ma con priorità.

Un problema intrinseco dello scheduling con priorità è la **starvation**, cioè l'attesa indefinita dei processi a bassa priorità, nel caso in cui i processi ad alta priorità non finiscano mai. Questo problema viene risolto applicando la tecnica di **aging**, almeno in caso di priorità dinamiche: più un processo "invecchia" (è dentro la coda), più la sua priorità viene aumentata.

## 2.12.6 Scheduling a code multiple

Si può applicare algoritmo SJF anche in caso di priorità: un modo è quello di creare code multiple, una per ogni priorità della scala. Si applicano solitamente due tipi di algoritmi, che sfruttano internamente algoritmi già noti:

- **Verticale**: Si sceglie la coda con priorità più alta, non vuota ovviamente. Per schedulare all'interno della coda si usa di solito Round-Robin. Se un processo si blocca prima della fine del quanto, tornerà comunque nella coda originaria. Sarebbe possibile anche applicare un algoritmo diverso per ogni coda. Si preferisce un approccio ibrido: si applica Round-Robin per tutte le code, ma per ogni coda (ogni priorità) si definiscono quanti diversi. In dettaglio: più alta è la priorità della coda, più piccolo è il quanto. Questo permette di aumentare la reattività del sistema all'aumentare della priorità (molto desiderabile). Alla coda con priorità minore si può benissimo applicare FCFS, nonostante siano processi CPU bound (infatti stanno nella coda più "bassa"). Perchè sono gli ultimi della lista, non devono rendere conto a nessuno, perchè in caso interviene la prelazione. Il problema per loro è rappresentato ancora dalla starvation, per ovvi motivi. In questo caso però si può applicare un metodo più "elegante" dell'aging: si assegna un quanto di tempo non al singolo processo, ma a TUTTE le code. In che senso? Questo tempo viene "spartito" tra le code, a seconda delle priorità che hanno, e ogni coda spartisce il suo spicchio di tempo tra i suoi processi. Esempio: ci sono 4 priorità, quindi 4 code. Di 5 secondi, il 60% viene assegnato alla coda con priorità 4, il 20% alla coda 3, il 16% alla coda 2, e 4% alla coda 1. Questo risolve starvation, perchè, cascasse il mondo, ogni coda avrà del tempo di CPU a disposizione, assegnato all'inizio, a bocce ferme.
- **Orizzontale**.

Qual è il problema delle code multiple? Che le priorità sono fisse. Si risolve il problema con la **retroazione**, cioè i processi vengono spostati dinamicamente da una coda all'altra (ovviamente questo quando viene tolto dalla sua coda, poi quando torna pronto viene cambiato di coda), in base a come usa il quanto a sua disposizione. Se ne usa un'alta percentuale, significa che ha CPU burst grandi (è CPU bounded), e viene "scalato" di coda, finchè non userà una percentuale abbastanza bassa (aumenta il quanto, i suoi CPU burst rimangono uguali). Gli

upgrade vengono decisi sulla base di una media pesata. Windows utilizza questo metodo di scheduling (code multiple con retroazione).

### 2.12.7 Shortest Process Next

Lo **Shortest Process Next** non è altro che l'SJF adattato ai sistemi interattivi. Non avendo qui un dato già pronto sulla durata del processo, si considera la durata del prossimo CPU Burst di quel processo; verrà schedulato il processo con il prossimo CPU Burst stimato più basso, similmente al funzionamento dell'SJF, in modo da favorire i processi I/O bounded (obiettivo prefissato). Come si stima la durata del prossimo CPU Burst? Ci si basa sulla stima precedente, e sulla durata (effettivamente misurata) dell'ultimo CPU Burst. Se la stima è calcolata correttamente, essa varierà nel tempo.

Consideriamo  $T_n$ , cioè l' $n$ -esimo CPU Burst (nel nostro caso l'ultimo, osservabile a posteriori quindi), e  $S_n$ , che sarebbe l'ultima stima calcolata, al CPU Burst precedente. Con questi due valori calcoleremo la nuova stima, detta  $S_{n+1}$ . Precisamente, con una formula che terrà conto sia di questi valori, sia di un certo parametro  $a$ , compreso tra 0 ed 1. Il valore di  $a$  farà "pesare" più la stima precedente (avvicinandosi a 0), oppure la misurazione reale (avvicinandosi ad 1). Con valori agli estremi,  $a = 0$  sarà considerata soltanto la stima precedente, con  $a = 1$  soltanto il valore del CPU Burst precedente.

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

In pratica, si cerca di intuire dove "andrà"  $T_{n+1}$  e "inseguirlo" con  $S_{n+1}$  in modo da andargli il più vicino possibile. Il valore migliore di  $a$  sarebbe  $\frac{1}{2}$ , questo perchè è giusto far pesare un po' di più la misurazione reale, senza però dimenticarci di ciò che abbiamo stimato in precedenza (è come se  $S_n$  ci facesse ricordare del "passato"). Un adattamento troppo repentino renderebbe instabile la stima.

Questo fa in modo che Shortest Process Next non abbia bisogno della prelazione, perchè la stima è sufficientemente precisa. Con prelazione, si rendono necessari degli adattamenti.

### 2.12.8 Altri tipi di scheduling su sistemi interattivi

- **Garantito:** usato realmente su Linux, poi spiegheremo. Lo scheduling garantito ha il compito di risolvere il problema della starvation una volta per tutte; con i sistemi precedenti, ci si mettevano delle pezze, cioè delle euristiche, che però non funzionano sempre.

Partiamo dal presupposto che non possiamo correlare completamente l'aumento della priorità con l'aumento della quantità (in pratica, se ad esempio la priorità aumenta da 1 a 2, non possiamo dire effettivamente che quel processo usi il doppio delle risorse rispetto a prima). Per cui, lo scheduling garantito fa delle "promesse" ai processi, promesse sulla percentuale di tempo per cui il processo userà la CPU, riferito ad un certo lasso di tempo. Lo scheduler dovrà poi calcolare quanto effettivamente quel processo userà la CPU, per vedere quanto vicino ci è andato a mantenere la promessa. Se è rimasto indietro rispetto alla promessa, cioè quel processo ha usato meno la CPU di quanto aveva

detto lo scheduler, avrà una priorità più alta, diciamo. In pratica: verrà schedulato ogni volta il processo rimasto più indietro rispetto alle promesse.

- **A lotteria:** Si divide il lasso di tempo in tanti piccoli quanti. Ad ogni quanto corrisponderà un cosiddetto **ticket**. I ticket verranno assegnati randomicamente ai processi, e poi ci saranno delle estrazioni, come una lotteria appunto. Se fosse distribuito lo stesso numero di ticket ad ogni processo, si avrebbe equità totale. Alla fine, più ticket ha un processo, più utilizza la CPU, probabilmente. Possiamo vedere ciò come una promessa, del tipo: se hai un ticket, prima o poi verrai estratto, ed userai la CPU. I processi possono collaborare tra loro, scambiandosi i ticket. Tipico il caso di processi padre che cedono alcuni ticket a processi figli, in modo da velocizzare dei task associati a quel programma. Anche questo metodo di scheduling privilegia i processi I/O bounded, perché rimanendo per meno tempo dentro la coda dei processi pronti, consumeranno meno ticket, e ad ogni estrazione ne avranno di più.
- **Fair-share:** l'ideale sarebbe ripartire la CPU equamente tra i processi. L'approccio fair-share spiegato qui applica quest'idea, ma non a livello di singolo processo, bensì a livello di utenza (ideale in un sistema multiutente). Cioè l'utilizzo della CPU viene ripartito tra gli utenti sulla macchina, indipendentemente da quanti processi hanno. Esempio: utente 1 con 1 processo, utente 2 con 9 processi. Un fair-share a livello di processo, assegnerebbe il 10% della CPU all'utente 1, e il 90% all'utente 2. Invece fair-share a livello di utenza gliene assegna 50 e 50. Su container tipo Docker si applica una cosa del genere: la CPU viene ripartita tra i vari container.  
Si può applicare una logica ibrida tra fair-share e lotteria: i ticket vengono ripartiti equamente tra gli utenti, che poi li ripartiscono internamente tra i processi.

### 2.12.9 Scheduling dei thread

Per parlare dello scheduling dei thread dobbiamo distinguere due situazioni:

- **Thread utente:** lo scheduler, con annesso algoritmo, è implementato all'interno del runtime environment; per ovvi motivi, non si baserà sulla prelazione. Facilitato dal fatto che i thread utente sono collaborativi;
- **Thread kernel:** si può schedulare in base alla pesantezza del context-switch. In pratica si tende a scegliere, nei limiti dell'equità (prima o poi si dovrà schedulare qualche thread di qualche altro processo), thread imparentati tra loro.

### 2.12.10 Scheduling su sistemi multiprocessore

In presenza di più CPU (core), bisogna porsi anche il problema di spartire i processi tra i core a disposizione. Ad esempio: ho N processi e 4 core. Dovrò fare in modo che tutti e 4 lavorino: non voglio situazioni del tipo 3 core sfruttati al massimo, e il quarto praticamente in idle. Poi preferirei avere overhead minimo, e ovviamente poter riapplicare gli algoritmi precedenti, senza

spremermi le meningi per concepirne di nuovi.

Chiediamoci: di cosa si dovranno occupare i core? Ci sono due approcci:

- **Multielaborazione asimmetrica:** un core principale, detto **master**, coordinerà le azioni degli altri core **slave** (ancora si può dire), ed ogni core slave si occuperà di un compito ben preciso. Precisamente, il core master esegue le routine kernel, e gestisce la coda dei processi pronti. Il problema di questo approccio, che l'ha portato a non essere più diffuso, è la sua bassa scalabilità. Con troppi core, il core master, da solo, dovrebbe gestirne troppi, ritrovandosi sotto sforzo. Su sistemi piccoli, il master sarebbe praticamente sottosfruttato.
- **Multielaborazione simmetrica:** l'approccio più utilizzato. Ogni core si occupa delle stesse cose, e si gestiscono da soli, ciò favorisce la scalabilità. Ci chiedevamo prima: come si spartiscono il lavoro? L'ideale, con  $N$  processi ed  $M$  core, sarebbe fare  $\frac{N}{M}$ , ma non è così facile. Dipende dal numero di code. Potrei avere una sola grande coda **condivisa**, accessibile in concorrenza. Se un core non ha nulla da fare, chiama lo scheduler, che pesca dalla coda condivisa e gli assegna un processo. Ovviamente, si ripresenta il problema delle race condition; è necessario un meccanismo di lock. A causa del lock, in presenza di tanti core, si possono verificare bottlenecks, perché aumenta grado di concorrenza (numero di core che vogliono accedere alla coda, chiamando lo scheduler). Con il lock, una CPU trova lavoro, le altre dovranno aspettare ad oltranza, finché non arriva il loro turno. Possiamo vederlo come un moderno centro di collocamento.

Il problema si risolve con  $M$  code, una per ogni core. Idealmente non ci sarà più concorrenza, e ogni CPU avrà il suo processo senza aspettare (tranne nel caso di coda vuota ovviamente). Ma: il lock ci servirà comunque, in maniera molto più limitata, vedremo dopo. Risolto un problema, se ne presenta un altro, come al solito: il bilanciamento. Che si intende per bilanciamento? Se arriva un processo nuovo, lo metto nella coda più vuota, e ok. Ma comunque si tratta di un sistema dinamico, che varia continuamente, tra processi che si bloccano, processi terminati e processi inseriti. Può capitare comunque che una coda si svuoti completamente: la CPU proprietaria della coda rimane in idle. Con la coda unica, bastava pescare da essa. Per risolvere quest'altro problema, si prevedono dei sistemi di **migrazione**, cioè lo spostamento di processi da una coda all'altra. La migrazione **guidata** (push) prevede un approccio alla Robin Hood: viene chiamata periodicamente, da qualcuno dei core, una routine a livello kernel, che controlla qual è la coda più vuota, e quella più piena, e sposta alcuni processi dalla più piena alla più vuota. Ecco perchè serve il lock: alcune volte una CPU (quella addetta in quel momento alla chiamata della routine) dovrà accedere alle code di altre CPU. Si capisce però che questo lock sarà sporadico.

La migrazione **spontanea** entra in gioco ogni volta che una delle code rimane completamente vuota. La CPU con la coda vuota prende un paio di processi dalla coda più ricca. Linux preferisce un approccio ibrido, a seconda delle situazioni.

Si possono distinguere due tipi di **predilezione** nello scheduling: **forte** e **debole**. La predilezione debole è quella usata fino ad ora, senza vincoli. Ogni processo può essere inserito in qualunque

coda. La predilezione forte prevede un vincolo, cioè che un certo processo venga eseguito soltanto su un certo core, quindi inserito soltanto nella sua coda. In casi eccezionali può essere violato, ma l'OS "se lo segna". Comunque, di default, la predilezione è disattivata.

## 2.12.11 Cosa usano i nostri OS

- **Windows:** si usano code di priorità multiple, con priorità di base influenzata da admin, ma si possono gestire dinamicamente, con downgrade e upgrade (retroazione praticamente). Windows ha euristiche legate a processi interattivi, con GUI. Cioè boost di priorità a processi che tornano da bloccaggio, specialmente se dovuto ad I/O, così l'interattività del sistema è migliorata. Lui cerca di prioritizzare il processo associato alla finestra in primo piano, in modo da rendere il tutto più fluido per l'utente. Windows ha problemi di starvation, più subdoli, cioè inversione di priorità. Se ho uno a priorità alta, e uno a bassa, che collaborano, è come se fossero produttore e consumatore. Succede quello che abbiamo già spiegato ai tempi (stallano tra loro). E' colpa dello scheduling verticale di Windows. Per evitarlo si usa euristica di **autoboost**, che cerca di tracciare l'uso di risorse da parte dei processi. Se un processo con bassa priorità ne blocca uno con alta, gli sarà dato un boost temporaneo in modo da far sbloccare tutto, svuotando il buffer.
- **Linux:** usa come algoritmo il CFS, **Completely Fair Scheduler**, cioè lo scheduler garantito. Ogni processo ha un **Virtual Run-Time**, VRT, il tempo per cui ha usato la CPU in precedenza. La coda dei processi pronti di Linux è basata su un red-black tree, ciò permette di estrarre massimo e minimo, in base al VRT. In pratica schedula quello con VRT più piccolo. Appena il VRT del processo schedulato cresce, e non è più il più piccolo (è correntemente in esecuzione), si rifà context switch. C'è prelazione, e non ci sono quanti di tempo. Quando arriva un processo nuovo, gli si dà VRT fittizio. Quindi VRT di tutti cresce in modo equo. Cioè il VRT di ognuno cresce di poco, nel caso di I/O bounded, così hanno un boost naturale garantito. Priorità qui sono viste come fattori di decadenza, cioè un delta moltiplicato al VRT, in modo che processi ad alta priorità abbiano VRT più basso, e quindi preferiti. Non c'è neanche la starvation, perché VRT non viene incrementato se sta fermo, quindi prima o poi diventa il minimo. Prima o poi VRT va in overflow, ma basta normalizzarlo, sottraendogli il minimo.
- **Mac OS X:** molto simile a quello di Windows.

# 3 Gestione della memoria

Come avevamo anticipato nell'introduzione, la memoria di un calcolatore è organizzata secondo una gerarchia, ordinata in base al tempo di accesso (da più veloce a più lenta; allo stesso tempo, dalla meno alla più capiente). Le memorie cache, e i registri, sono gestite direttamente dalla CPU, mentre la RAM è gestita in simbiosi dalla CPU con l'OS. Queste sono le memorie volatili, le più veloci. Le memorie lente sono le memorie secondarie, non volatili, come dischi magnetici ed SSD. Queste memorie sono dette "di massa", perché possono immagazzinare grandi quantità di dati, e non sono accessibili direttamente dalla CPU, l'OS deve fare da intermediario.

## 3.1 Gestione dei processi in memoria centrale

Come possiamo gestire la memorizzazione dei processi (codice e dati) in RAM? Inizialmente non si parlava di sistemi multiprocesso, per cui si adoperava un modello **senza astrazioni**, precisamente **senza multiplexing nello spazio**. I processi usano direttamente gli indirizzi fisici quando eseguono operazioni di fetch e store. In sistemi molto elementari, tipo embedded, questo approccio funziona ancora, dato il numero esiguo di processi, evitando l'overhead dato dalle astrazioni. Il problema principale di questo modello è la sua applicazione in sistemi multiprocesso: come farli coesistere? Dobbiamo intanto capire come "dividere" la memoria:

- Un primo modo potrebbe essere quello di riservare una porzione al codice e ai dati dell'OS: da qualche parte dovranno pur stare. Il resto della memoria è potenzialmente sfruttabile in toto dal processo in esecuzione, dato per scontato che ce ne sia solo uno;
- Si può caricare l'OS in una ROM, cioè in una memoria non volatile: possibile se l'OS in questione è molto minimale, come nel caso di sistemi embedded. L'OS è già precaricato, e tutta la RAM è disponibile per il processo;
- Usiamo un software simil-BIOS/UEFI caricato in ROM, che all'avvio caricherà l'OS in RAM, il resto della memoria sempre riservata al processo.

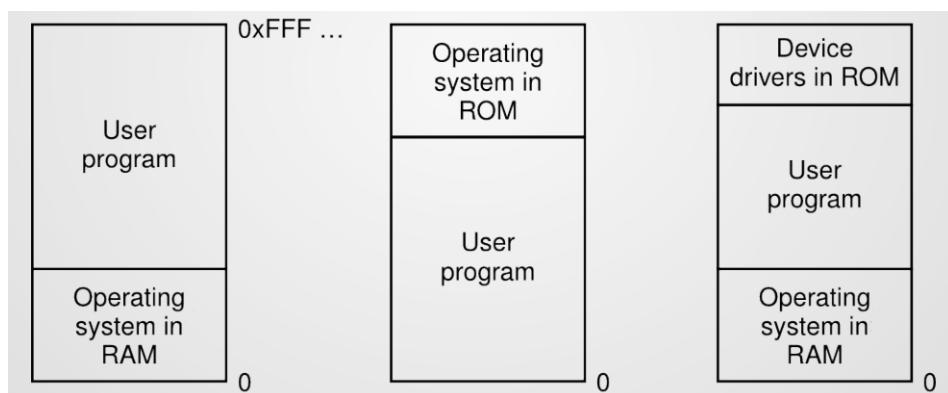


Figure 35: Modelli di organizzazione della memoria

Il fatto è, che non avendo alcuna astrazione, dobbiamo effettuare una divisione "a priori" dello spazio da spartire tra i processi, nel caso di un sistema multiprocesso. Cioè dobbiamo decidere i

range di memoria (il "pezzo" di memoria) riservati ad ognuno dei processi, a compile-time. Ciò è ovviamente un limite, perché o eseguo i processi in modo esclusivo (due processi potrebbero avere lo stesso range), o ricompilo tutto ogni volta che devo eseguire quel processo su una macchina diversa, con memoria divisa in maniera di versa, cambiando il range. In generale, devo fare in modo che indipendentemente da come è scritto il codice del processo e dove viene eseguito, la memoria sia spartita correttamente. Potrei anche fare multiplexing nel tempo, ma dovrei spostare i dati del processo precedente sul disco rigido... troppo dispendioso.

### 3.1.1 Rilocazione

Proviamo a spiegare con un esempio qual è il problema. Facciamo finta di avere una memoria ripartita tra tre processi P1, P2, P3. P1 ha riservato il range che va da 0 a un certo limite LP1, nel suo codice, e P2 ha un altro range standard, che va sempre da 0 a LP2 (programmatore ovviamente non sa nulla riguardo la divisione della memoria su cui sarà eseguito il suo programma, assegnerà dei range canonici). Vabbè, prendiamo P2. Il suo codice può generare riferimenti (puntatori) ad altre zone del codice (call, salti) oppure ai suoi dati, con altri puntatori. Se P2 ad un certo punto deve saltare alla locazione 1000, e  $LP1 > 1000$ , P2 "invade" lo spazio di P1. Serve la **rilocazione** dei processi, cioè spostare i range dei processi in modo da poterli dividere bene, senza che si invadano tra loro. Senza disporre di astrazioni, non possiamo rilocare a run-time, ma solo in due modi:

- **A compile-time**: si decidono le zone a priori, a tempo di compilazione appunto. Però per spostarli di slot, diciamo così, devo ricompilare tutto.
- **Rilocazione statica**: una rilocazione a **loading-time**, cioè durante il "caricamento" del processo (appena gli viene assegnata la CPU, prima che esegua la prima istruzione). Abbiamo già detto che il codice dei processi fornisce un range "canonico", non precisato, o per meglio dire, non adattato alla particolare situazione, generico. Parleremo di **spazio di indirizzamento logico** (cioè si parte da 0). Si manipola quindi il codice appena prima dell'esecuzione, quando già lo scenario è delineato. L'OS assegna dei range ai processi, con una locazione di inizio e una di fine. Questo tipo di rilocazione scansiona il codice, in cerca di riferimenti, prende quegli indirizzi e li mappa, sommando il generico indirizzo i alla locazione di inizio, in modo da farlo rientrare nel range effettivo assegnatogli. Esempio molto semplice: range del processo P2 va da locazione 20000 a 30000. Nel suo codice è contenuto un riferimento alla locazione logica 3000. Si somma 3000 a 20000, e si ottiene la locazione effettiva, fisica: 23000.

Funziona anche bene, ma ovviamente ha un alto costo, che fa rallentare il caricamento del processo.

Altro problema è quello della **protezione** della memoria: nessun processo deve poter accedere a locazioni di proprietà di altri processi, e soprattutto non deve poter accedere a locazioni di proprietà dell'OS. Ogni zona deve essere un compartimento stagno, diversamente da come succedeva in MS-DOS. Si potrebbe pensare di realizzare protezione mettendo l'OS in ROM, ma così proteggi solo il suo codice, i dati sono tenuti comunque in RAM. Per ottenere protezione

totale, ho bisogno del supporto dell'hardware.

Il sistema operativo OS/360, di IBM, usava un metodo particolare quanto semplice: **lock & key**. La RAM era divisa in "porzioni" da 2 KB, e ad ognuna era associata una **chiave**, cioè un valore in binario. E ogni processo nella tabella dei processi ha a sua volta una chiave diversa. Come funziona? Quando un processo, ad esempio il processo P1, viene schedulato, la sua chiave viene copiata nella PSW. Le porzioni di memoria di sua proprietà avranno anche loro la sua chiave. Quando P1 viene eseguito, e deve effettuare una fetch o una store, e genera un indirizzo, confronta la chiave di quell'indirizzo (precisamente, della porzione di cui fa parte) con la chiave dentro la PSW. Se coincide, si sta scrivendo/leggendo una locazione che è dentro il range, altrimenti no e l'operazione viene negata. Ovviamente, ad ogni context-switch, la chiave nella PSW viene sovrascritta, con la chiave del nuovo processo. All'aumentare del numero di processi questo metodo diventa poco scalabile.

### 3.1.2 Spazio degli indirizzi

A questo punto, possiamo introdurre una prima astrazione: lo **spazio degli indirizzi**. Sono presenti due registri, un **registro base** (RB) e un **registro limite** (RL). Nel primo registro è contenuto l'indirizzo della prima locazione, quella da dove parte il range, e nel secondo registro è contenuta la lunghezza del range. E' quindi possibile identificare il range di proprietà del processo. Queste informazioni sono contenute nel PCB del processo.

Useremo un altro tipo di rilocazione, la **rilocazione dinamica**. Partiamo dal fatto che il nostro codice con indirizzi canonici. In questo caso però interviene in aiuto un subcomponente della CPU, probabilmente già citato ed accennato prima: **MMU** (Memory Management Unit), che si occupa di tradurre l'indirizzo logico (canonico) in un indirizzo fisico, appoggiandosi sui due registri RB ed RL. Fino ad ora sembra la stessa cosa della rilocazione statica, con l'aiuto di un componente esterno, ma in realtà c'è una distinzione importante: la traduzione viene fatta **a run-time**. Quindi, al momento in cui si arriva ad un'istruzione di fetch o store, viene preso l'indirizzo e tradotto in quel momento dalla MMU, allo stesso modo di prima: viene sommato indirizzo logico a contenuto di RB. Ma prima effettua un controllo: vede se l'indirizzo logico è maggiore del valore di RL: se sì, l'indirizzo sicuramente sfiorerà il range. Questo controllo garantisce la protezione della memoria citata sopra. Se questo controllo va male, viene effettuata una trap. Una routine dell'OS terminerà il processo. La terminazione viene fatta in modalità kernel perchè altrimenti il processo potrebbe eludere il controllo.

La rilocazione dinamica è stata introdotta con la CPU Intel 8088, che possiamo vedere come una "capostipite" delle architetture CPU moderne (x86). La rilocazione dinamica è migliore anche perchè nella rilocazione statica la somma veniva effettuata ad ogni fetch.

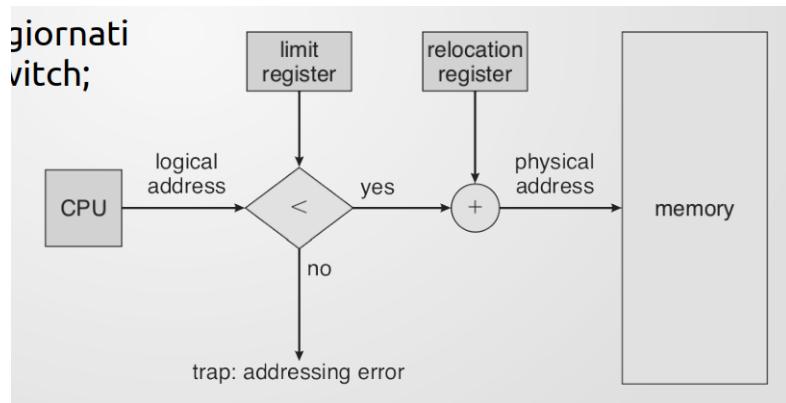


Figure 36: Come funziona la rilocazione dinamica

### 3.1.3 Swapping

In un sistema multiprogrammato si presenta un altro problema, abbastanza scontato: tanti processi riempiono velocemente la RAM. Una mente brillante potrebbe pensare di aumentare semplicemente la quantità di RAM, ma non siamo tutti ricchi. Dobbiamo trovare il modo di liberarla se necessario.

La prima (e storica) soluzione è lo **swapping**: in pratica l'OS prende un processo tra quelli che sono stati allocati in memoria e lo "parcheggia" nel disco, cioè prende tutta la zona di memoria di quel processo (i dati) e li memorizza temporaneamente nel disco. Questo processo non potrà essere più schedulato finché i suoi dati non ritornano in RAM. Quest'ultima si libera, e l'operatore è libero di poterne lanciare un altro, anche se inizialmente la RAM era piena. Ovviamente sarà scelto un processo che se "parcheggiato" non intacca l'esperienza dell'utente. Qui entra in gioco lo scheduler **di medio termine**, che sceglie quali processi parcheggiare nel disco, ed usa algoritmi diversi rispetto agli scheduler che abbiamo visto (quelli si dicono di breve termine), ha criteri diversi: sceglierà un processo abbastanza grande da poter liberare la RAM necessaria, o magari uno che sta in background. Il processo, dopo il parcheggio, non termina: il suo PCB rimane, e quando viene riportato in RAM riparte da dove si era fermato, è come se si bloccasse semplicemente. Si spostano soltanto i dati. Ogni volta che un processo termina, si controlla se il processo parcheggiato può essere rimesso in RAM. Con rilocazione statica, devo ricaricare il codice, per rimappare gli indirizzi. Con rilocazione dinamica, devo cambiare il valore del registro base, perché quei dati vengono rimessi in una partizione diversa da quella di prima, mica è sicuro che si riliberi quella di prima. Il flusso di esecuzione non viene alterato.

Si possono verificare problemi nel caso di I/O pendenti: se per esempio un processo P2 sta scrivendo su un file, o in generale qualsiasi operazione di I/O, magari usando DMA, e nel frattempo viene spostato in disco, che succede? Al posto dei dati di P2 vengono messi i dati del nuovo processo, il controller non se ne accorge, e magari scrive su una locazione che prima conteneva una struttura dati di P2, sporcando i dati del nuovo processo.

Con lo swapping devo tenere conto anche del vincolo di contiguità: tutti i dati di un processo devono essere contigui in memoria. Se ad esempio si liberano dei "buchi" non contigui (**frammentazione esterna**), e la somma di questi è maggiore della dimensione richiesta dal nuovo processo, non si può allocare, non si può "spezzettare". Si potrebbero spostare le par-

tizioni degli altri processi, per far diventare il "buco" contiguo (**memory compaction**), ma è un'operazione troppo onerosa, e crea inceppamenti.

Bisogna stare attenti al fatto che in realtà un processo non ha dimensione fissa, ma varia a run-time. Si conosce la sua taglia base (codice + dati al momento della prima istruzione); l'unica soluzione, non buona, è sempre lo spostamento, per garantire contiguità. La **frammentazione interna** è il problema principale che si verifica a causa della dinamicità della dimensione di un processo: potrebbero rimanere delle porzioni non usate dentro la partizione, se il processo si restringe, oppure una porzione lasciata vuota di proposito, per supportare la crescita del processo. Questo è spazio sprecato, che potrebbe non essere mai usato.

## 3.2 Gestione dell'allocazione

Come faccio a tenere d'occhio la situazione delle allocazioni nella RAM? Cioè, come faccio a tenere traccia di quali locazioni sono assegnate a qualche processo e quali no? Ho bisogno di particolari **strutture dati**, che "fanno ricordare" alla CPU cosa è successo in precedenza. Ci sono due metodologie. Tutte e due si basano sulla definizione di una **unità minima allocabile**, cioè si divide logicamente la RAM in piccoli blocchetti, tutti di dimensione pari a questa unità minima allocabile.

### 3.2.1 Bitmap

Si crea una "tabella" di bit, con un numero di bit pari al numero di blocchi presenti in RAM. Il singolo bit indica lo stato del blocco, 1 se allocato, 0 se è libero. Bisogna stare attenti però a non definire un'unità minima allocabile troppo piccola: ci sarebbero più blocchetti, ciò implica più bit, e la bitmap occuperebbe troppo spazio in RAM (ovviamente viene salvata lì). Se invece viene definita troppo grande, si creano fenomeni di frammentazione interna, a causa dell'elevato arrotondamento. Un problema della bitmap è la sua dimensione fissa: occupa lo stesso spazio, anche se ci sono pochi (o nessuno) processi in memoria.

### 3.2.2 Liste

La soluzione all'ultimo problema è semplicemente una struttura dati dinamica, magari una **lista doppiamente concatenata**, generalmente ordinata per indirizzo. Vedremo perchè proprio doppiamente concatenata. Comunque: il generico nodo di questa lista rappresenta o una partizione associata ad un processo, oppure un "buco". Ogni nodo contiene nome del processo, blocco di partenza e numero di blocchi occupati. Con un solo nodo tengo traccia di più blocchi. La cosa bella di questa metodologia è l'applicazione della **coalescenza**, che viene applicata ogni volta che una partizione si libera: controlla se ci sono 2 nodi "liberi" contigui, e li fonde in uno, con lunghezza dei blocchi pari alla somma dei nodi precedenti. Così da 2 nodi si passa ad un solo nodo, e la lista si riduce di dimensione. Ciò riduce anche la durata delle ricerche (che faremo) in questa lista.

La lista viene memorizzata nella parte libera di RAM. Un paradosso: meno spazio libero rimane (più processi allocati), più la lista si ingrandisce, erodendo ancora lo spazio libero. Ma comunque

teoricamente ci entra in ogni caso, le differenze in dimensioni tra lista e spazio libero sono di diversi ordini di grandezza. La lista è doppiamente concatenata per favorire la coalescenza: nel PCB di ogni processo c'è il puntatore al suo nodo, posso tornare indietro o andare avanti a piacimento, senza problemi. Tornerà utile anche nelle ricerche che spiegheremo adesso. Si ha bisogno di trovare il miglior "buco" possibile per poter inserire una nuova partizione: si effettua ricerca nella lista, potendo applicare 4 diversi algoritmi (cioè 4 diversi criteri di scelta, applicabili anche con bitmap):

- **First fit:** la ricerca parte dalla testa, ci si ferma al primo "buco" abbastanza grande da contenere nuova partizione. Se il "buco" è più grande della partizione, rimarrà un piccolo "buchino", che come conseguenza ha 1 nodo in più nella lista, e probabile spazio inutilizzabile, cioè frammentazione esterna. Dato che si sceglie il primo buco disponibile partendo dall'inizio, i processi si addensano nelle prime locazioni;
- **Next fit:** permette di rendere più uniforme la distribuzione delle partizioni. Non si parte più dall'inizio, si parte dall'ultimo nodo che ha allocato, per il resto effettua la scelta esattamente come prima.
- **Best fit:** l'obiettivo è quello di evitare il più possibile la frammentazione esterna. Cerco quindi di trovare un buco della taglia esatta della partizione. Se non c'è, si prende buco più piccolo, tra quelli abbastanza capienti ovviamente. Paradossalmente il problema della frammentazione esterna peggiora, perchè così verranno a crearsi dei microbuchi non contigui che sicuramente non saranno mai riempiti, perchè troppo piccoli.
- **Worst fit:** Si usa la psicologia inversa: si sceglie il buco più grande possibile, in modo che il nuovo buco che si viene a creare sia abbastanza grande da contenere un'ulteriore partizione.

First fit e next fit sono più efficienti come complessità, visitano meno blocchi. Gli altri due devono effettuare una scansione totale della lista, per come funzionano.

Si potrebbero usare due liste distinte, una contenente solo le partizioni allocate, l'altra solo i buchi. First e next migliorano, perchè avranno una complessità minore (posso guardare solo i buchi, senza perdere tempo scansionando anche partizioni allocate). E' più complesso convertire un buco a partizione "piena", e viceversa. Devo spostare il nodo da una lista ad un'altra. Però se la lista dei buchi viene ordinata per dimensione, migliorano best e worst: avranno tempo costante, perchè best sceglierà sempre la testa, e worst sempre la coda.

### 3.3 Memoria virtuale

Gli approcci che abbiamo visto in precedenza si basavano sull'allocazione contigua dei dati del processo. Ciò causava molti problemi, ad esempio la frammentazione esterna. L'unica soluzione è cercare di applicare allocazione non contigua. Come fare? Possiamo vedere i dati del processo da un punto di vista logico, detto **spazio di indirizzamento virtuale**; da questo punto di vista le varie locazioni sono contigue, vanno da 0 ad un certo MAX. Questo fa in modo che

virtualmente il processo abbia a disposizione infinita memoria. Fisicamente, i vari dati saranno "sparpagliati" nella memoria, e bisognerà tenere traccia di dove siano stati messi.

Questa cosa si può implementare dividendo lo spazio di indirizzamento virtuale in **pagine**, ognuna di dimensione (fissa) nell'ordine dei KB. Come abbiamo detto, la memoria disponibile è virtualmente infinita, ma in realtà ovviamente ciò è impossibile. Per risolvere questo problema, si sceglie di non tenere tutte le pagine allo stesso tempo nella RAM: le restanti saranno memorizzate in un file su disco (detto area di swap). La RAM sarà a sua volta divisa in **frame**, di dimensione uguale a quella delle pagine. Ogni pagina è numerata, partendo da 0: questo numero si dice **numero di pagina**. L'idea alla base di questo metodo è quello di "inserire" la pagina virtuale in un frame fisico, in un qualunque punto della memoria fisica, come se fosse uno slot. Quindi, ad esempio, la pagina con numero 2 può essere inserita in un qualunque frame. Si capisce subito che si risolve il problema della frammentazione esterna: ogni "buchino" (frame) sarà riempito con qualche pagina, appartenente ad un qualsiasi processo. Ci fornisce una protezione intrinseca: le pagine sono separate tra loro, nessun processo può invadere le locazioni di un altro. Facendo un altro esempio: se ho due processi P1 e P2, e viene richiesta un'operazione di fetch o store sulla locazione logica 1000 di P1, e magari dopo sulla locazione logica 1000 di P2, saranno due locazioni fisiche diverse, appartenenti a due pagine diverse. Il tutto è gestito dalla MMU, perchè si necessita di hardware ad hoc per la gestione delle pagine e dei frame. Dato che soltanto una parte delle pagine di un processo saranno in disco, il processo non si bloccerà, sarà ancora eseguibile, perchè le altre pagine sono in RAM: diversamente dallo swapping, che passava TUTTO nel disco, facendo bloccare il processo.

### 3.3.1 Paginazione

Adesso la cosa più importante: come si associa (si "mappa") la pagina al frame? Attraverso una struttura dati gestita dall'OS, detta **tabella delle pagine**, di cui è presente un'istanza per ogni processo. Il numero di voci di ogni tabella sarà pari al numero delle pagine di quel processo; dato che la dimensione delle pagine è fissa per tutti, il numero di voci dipende dalla dimensione dello spazio di indirizzamento virtuale del singolo processo.

Capiamo come funziona con un semplice esempio. Supponiamo di avere uno spazio di indirizzamento virtuale pari a 64 KB, e la dimensione della pagina/frame pari a 4 KB. Abbiamo quindi 16 pagine, ognuna numerata, e che copre un certo range di indirizzi. Ad esempio, la prima pagina, numerata con 0, copre le locazioni logiche che vanno da 0 a 4095. La RAM è divisa sempre in frame da 4 KB, anch'essi numerati, con un **numero di frame**. Ogni voce della tabella corrisponde ad una certa pagina virtuale, e al suo interno è contenuto il numero di frame in cui è stata inserita la pagina, se è stata inserita ovviamente.

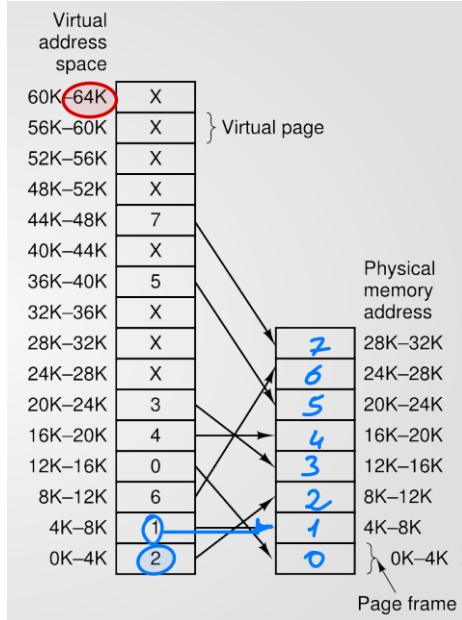


Figure 37: Tabella delle pagine

Nella voce dedicata alla pagina 0, c'è il numero 2. Ciò significa che la pagina 0 è contenuta nel frame 2 nella memoria fisica. Invece le pagine che non sono in RAM sono barrate con una X. Se viene richiesta una pagina che non si trova in RAM, si verifica **page fault**, che non è un errore, semplicemente indica all'OS che deve recuperare la pagina dal disco, mettendola in un frame vuoto o magari al posto di un'altra pagina che verrà messa su disco a sua volta. La tabella viene aggiornata, e si ritenta la richiesta. Questa volta la richiesta andrà a buon fine. Per vedere se una pagina è in RAM o meno, esiste un **bit di presenza**: se questo bit è ad 1, la pagina è in RAM, e nella voce è scritto il numero di frame corrispondente, altrimenti no, ed è presente un valore fittizio. La tabella viene gestita e modificata dalla MMU. Essa si occupa anche di un altro compito, ancora più importante: traduce l'indirizzo logico in un indirizzo fisico, sfruttando le informazioni contenute nella tabella (pagina a cui appartiene indirizzo virtuale, frame a cui appartiene indirizzo fisico). Come fa?

- Divide l'indirizzo virtuale per la dimensione di una singola pagina, ad esempio, con un indirizzo virtuale 8196, si esegue la divisione INTERA  $\frac{8196}{4096}$ . Il quoziente della divisione (2) è il numero di pagina, mentre il resto (4) è l'**offset** (la "distanza" tra la prima locazione della pagina, cioè l'inizio della pagina, e la locazione effettiva. Possiamo dire che sia la posizione della locazione internamente alla pagina).
- Si consulta la tabella: se il bit di presenza è 1, posso continuare con il prossimo passo, altrimenti page fault.
- Si ottiene il numero di frame corrispondente (6) consultando ancora la tabella; si moltiplica il numero di frame per la dimensione del frame, ovviamente uguale alla dimensione della pagina. Nel nostro esempio:  $6 \cdot 4096 = 32768$ . La locazione 32768 è la prima locazione del frame 6.
- Rimane soltanto da sommare l'offset (4). Quindi  $32768 + 4 = 32772$ . Abbiamo trovato la locazione fisica corrispondente alla locazione fisica 8196, cioè 32772. In pratica: abbiamo

trovato il frame giusto, adesso cerchiamo la locazione corretta all'interno del frame, cioè la quarta locazione del frame.

Da precisare: CPU utilizza soltanto indirizzi virtuali. La traduzione è effettuata interamente dalla MMU. Questo lavoro può potenzialmente creare bottleneck, perché MMU necessita di informazioni a loro volta memorizzate in RAM (esempio la tabella delle pagine).

### 3.3.2 Uso reale di una tabella delle pagine

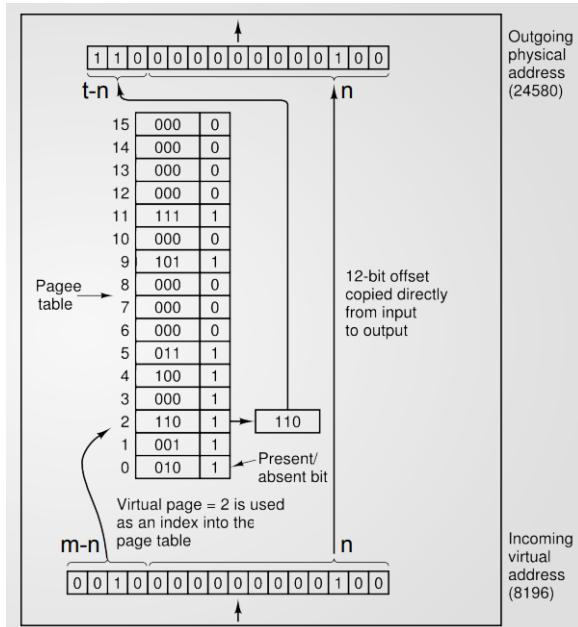


Figure 38: Tabella delle pagine

Gli indirizzi virtuali vengono visti naturalmente da MMU come maschere in bit. Per cui, se la dimensione della pagina, la dimensione dello spazio di indirizzamento virtuale, e la dimensione della RAM sono delle potenze di 2, le operazioni di traduzione spiegate prima sono molto più semplici ed efficienti.

Nell'esempio di prima, lo spazio di indirizzamento virtuale è 64 KB, quindi gli indirizzi virtuali sono a 16 bit. La dimensione della pagina è 4 KB, cioè  $2^{12}$ . Possiamo capire che per dividere, basta prendere il valore in bit della locazione virtuale e shiftare a destra di 12 posizioni. Otterremo che i 12 bit meno significativi saranno l'offset, e gli altri 4 saranno il numero di pagina (16 pagine, 4 bit, tutto torna). Consultando la tabella, ottengo il numero di frame corrispondente. Nel nostro esempio, il numero di frame è a 3 bit (abbiamo 32 KB di RAM, quindi 8 frame). La moltiplicazione è altrettanto semplice: si shifta il numero di frame a sinistra di 12 posizioni: si otterrà un numero a 15 bit (famicoce caso, RAM è 32 KB, cioè  $2^{15}$ ) in cui i 12 bit meno significativi sono tutti nulli, e gli altri 3 sono il numero di frame: praticamente, è l'indirizzo della prima word del frame. Sommare l'offset è un gioco da ragazzi: l'offset è grande 12 bit, esattamente quanto i bit nulli di questo nuovo numero. Con un bitwise OR si "copia" l'offset al posto dei bit nulli: abbiamo ottenuto l'indirizzo fisico, composto dai primi 3 bit più significativi che indicano il frame, e gli altri indicano l'offset.

Generalizzando, si può dire che se la dimensione dello spazio di indirizzamento virtuale è  $2^m$  ( $2^{16}$ ), e la dimensione della pagina è  $2^n$  ( $2^{12}$ ), allora il numero di pagina è dato dagli m-n bit più significativi dell'indirizzo virtuale ( $16 - 12 = 4$ , infatti avevamo 16 pagine,  $2^4$ ). L'offset è pari ad n. Gli indirizzi fisici invece sono pari a  $2^t$  ( $2^{15}$ ), per cui il numero di frame è dato dai t-n bit più significativi (3, infatti avevamo 8 frame).

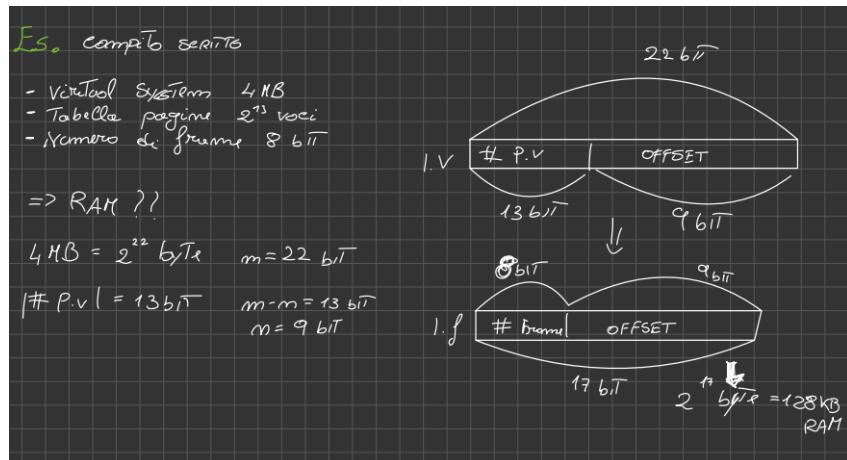


Figure 39: Esercizio 1 da compito su paginazione

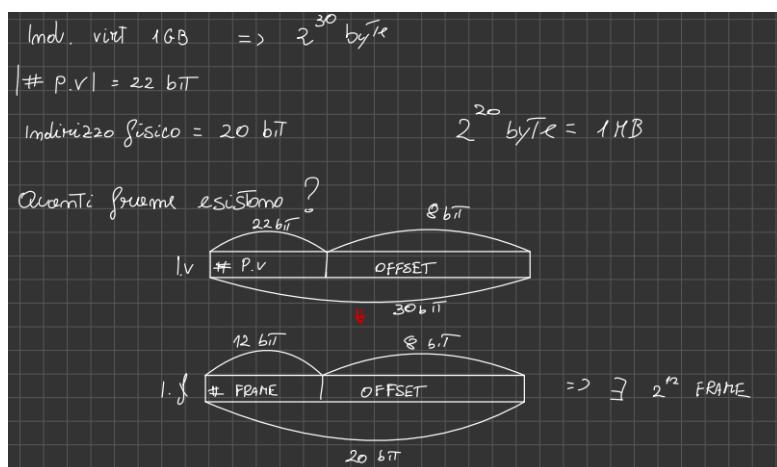


Figure 40: Esercizio 2 da compito su paginazione

### 3.3.3 Dettaglio su una voce della tabella delle pagine

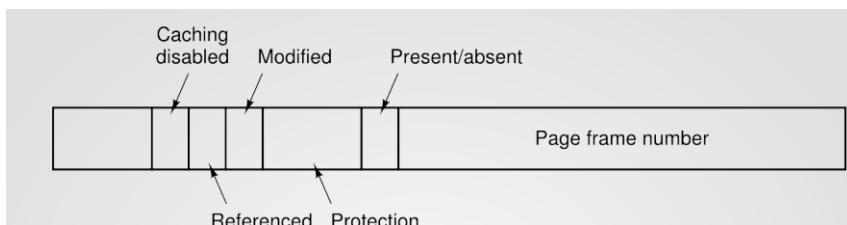


Figure 41: Una generica voce della tabella delle pagine

Capiamo che in un sistema reale, una voce della tabella delle pagine non può contenere soltanto il numero di frame e il bit di presenza: servono altre informazioni.

- **Protezione:** 2 bit che indicano permesso in lettura e in scrittura. A cosa potrebbe servire vietare, ad esempio, la scrittura? In molti casi le pagine devono essere accessibili soltanto in lettura, come nel caso di due istanze dello stesso programma, quindi due processi diversi ma con lo stesso codice. Per risparmiare memoria, il codice viene memorizzato una volta sola, e tutti e due i processi punteranno a quel codice. Ma se uno dei due modifica il codice, le modifiche si ripercuotono sull'altro. Per cui, le pagine del codice devono essere di sola lettura.  
Ci potrebbe essere un terzo bit, detto esecuzione. Nel senso: indica se quella pagina contiene codice eseguibile dalla CPU, e MMU deve controllare questo bit per evitare esecuzioni anomale.
- **Dirty bit:** in alcuni casi, ci potrebbe essere una copia della stessa pagina sia in RAM che in disco, magari nel caso in cui una pagina è stata prima spostata in disco, e poi rimessa in RAM. La copia nel disco può non essere cancellata, ad una condizione: le copie devono essere identiche. Il dirty bit serve proprio a questo: per vedere se quella pagina è stata modificata, quindi è diversa dalla copia in disco. In questo caso, la copia su disco viene sovrascritta con quella più aggiornata. Il bit viene aggiornato da MMU.
- **Referenziamento:** anche questo gestito da MMU. Possiamo dire che serve da "statistica" periodica. Viene posto ad 1 se la pagina è stata referenziata, cioè se è stato fatto almeno un accesso su di essa in un determinato lasso di tempo. Periodicamente (cioè dopo il lasso di tempo di sopra) il bit viene azzerato. Serve quindi per prendere determinate decisioni, magari durante l'esecuzione di un algoritmo di sostituzione delle pagine (quale tengo in RAM? quale invece metto nel disco?).
- **Disabilità cache:** indica che quella pagina non deve essere messa nella cache. In caso in cui la pagina contenga istruzioni di I/O, dopo il caching non si possono più controllare le porte di input e output (cioè la copia in cache non viene più aggiornata, e la CPU userà quella come riferimento, "purtroppo"), per cui la pagina non deve essere messa in cache.
- **Bit di validità:** indica se la pagina è allocata oppure no. Nel senso: quella pagina è parte dell'heap del processo, oppure è una zona "libera"? Se si prova ad accedervi, ed è effettivamente non allocata, si genera un'eccezione.

### 3.3.4 Tabella dei frame

A differenza della tabella delle pagine, una per ogni processo, per ovvi motivi, la tabella dei frame è una sola, con una voce per ogni singolo frame della RAM. La sua dimensione è quindi proporzionale alla quantità di RAM disponibile, a parità di "taglio" dei frame. La generica voce della tabella dei frame contiene informazioni sullo stato, cioè un flag che indica se il frame è occupato o libero, e, solo se occupato, indica l'ID del processo che lo occupa. NON indica quale pagina è contenuta dentro il frame. Posso occupare qualsiasi frame libero, indipendentemente dal processo che me lo chiede (niente frammentazione esterna, avevamo già chiarito questa cosa). La tabella dei frame viene consultata quando bisogna "piazzare" una pagina in memoria,

ad esempio, dopo un page fault, dobbiamo riuscire a portare quella pagina in RAM il prima possibile, e con minor overhead possibile. Ma questi sono dettagli, l'importante è che bisogna consultare questa tabella per vedere quali frame sono liberi, e scegliere in base a certi criteri che saranno spiegati successivamente.

### 3.3.5 Progettazione di una tabella delle pagine

Torniamo alle tabelle delle pagine. Come possiamo implementarla fisicamente? Due parametri sono importanti: **velocità** nella consultazione, e la **dimensione** (questo sarà un grande problema in caso di indirizzi a 64 bit, come vedremo). Per soddisfare il requisito della velocità ci sono due strategie principali:

- Tenere la tabella nei registri della MMU. Ai tempi, era presente un numero esiguo di pagine, quindi era possibile mettere ogni voce nei registri della MMU, direttamente nel suo package. Questo si traduceva in tempi d'accesso bassissimi, di conseguenza in una traduzione d'indirizzo molto veloce. Il problema si presentava in caso di context-switch: ogni volta la MMU doveva azzerare i propri registri, per poi scriverci sopra la tabella delle pagine del nuovo processo. Intendevamo questo quando parlavamo di "riprogrammare la MMU". Questo non causava particolari rallentamenti una volta, perché si trattava di tabelle piccole, e pochi processi. Adesso, una strategia del genere sarebbe inefficientissima; capiamo che non è un approccio scalabile.
- L'altra soluzione è quella di tenere la tabella (le tabelle) interamente in RAM, precisamente nello spazio dedicato all'OS. Dobbiamo tener conto però che le voci della tabella devono essere memorizzate contiguamente. In MMU risiede soltanto un particolare registro, detto **PTBR** (Page-Table Base Register), contenente il puntatore alla prima voce della tabella del processo attualmente in uso. In pratica, PTBR punta alla tabella. Il context-switch adesso è molto più efficiente: basta aggiornare PTBR. Ovviamente, perdiamo dal punto di vista dei tempi di accesso; prima dobbiamo accedere alla RAM per consultare la tabella, poi, dopo la traduzione, bisogna accedere ancora alla RAM per leggere/scrivere la locazione fisica interessata. Quindi si hanno due accessi. Il secondo accesso, con questo metodo, ha costo 1, perché abbiamo anche l'offset, ricordiamo. Ma non si tratta di una memoria cosiddetta "associativa", che vediamo tra poco.

### 3.3.6 TLB (memoria associativa)

Per evitare di avere un accesso aggiuntivo per ogni fetch/store, si può pensare di usare un qualche sistema di caching. Si potrebbe usare la cache della CPU, ma dipende la sua posizione. Oppure, la **TLB** (Translation Lookaside Buffer), anche detta **memoria associativa**. È una specie di cache contenuta nel package della MMU, specializzata soltanto nella memorizzazione di voci della tabella delle pagine, che è particolarmente necessaria nel caso in cui lo spazio di indirizzamento virtuale sia a 64 bit. Contiene circa 1024 registri, che permettono di memorizzare 1024 voci. Per essere efficace, bisogna sfruttare la ricerca parallelizzata in hardware: in pratica, l'hardware permette di cercare più elementi in contemporanea, in parallelo.

TLB è gestita dall'hardware, e si inserisce "in mezzo" tra la tabella e la RAM, diciamo, all'interno del processo di traduzione. Quindi, cosa contiene TLB? TLB contiene semplicemente alcune delle voci della tabella delle pagine, precisamente, quelle utilizzate più di recente, proprio come farebbe una cache. La generica voce nella TLB è quasi un sottoinsieme della generica voce della tabella. Il bit di referenziamento è ridondante, perché se è dentro la TLB, è ovvio che sia stato già referenziato. Viene aggiunta un'informazione: il numero della pagina. Questo perchè nella tabella le voci sono memorizzate a partire da 0, in maniera sequenziale, qui no, per cui è necessario identificare la pagina a cui la voce si riferisce. Il bit di validità cambia di significato, ci dirà se la voce è piena o vuota. Protezione e dirty bit rimangono, perchè devo avere i permessi "a portata di mano", e nel caso del dirty bit, devo sapere se alla cancellazione della voce, devo riaggiornarla in RAM, perchè gli aggiornamenti li faccio direttamente nella TLB.

TLB funziona esattamente come la cache: se la voce della pagina interessata viene trovata, si parla di **TLB hit**, altrimenti di **TLB miss**. In caso di TLB hit, trovo subito il numero di frame, senza passare dalla RAM. In caso di miss, devo accedere alla RAM, e scrivere la voce in TLB. Se è piena, MMU sceglie la voce da scartare, con l'algoritmo LRU (Least Recently Used). Ma perchè TLB funziona, cioè ottimizza? Esattamente per lo stesso motivo per cui funziona la cache: un processo tenderà ad utilizzare spesso soltanto una piccola porzione del suo codice e dei suoi dati, per cui è altamente probabile che siano già in TLB. Ci saranno più hit che miss. Anche l'OS ha una sua tabella delle pagine, TLB gli farebbe comodo. Allora riserva delle voci di TLB per sè.

Il context-switch deve tenere conto di come funziona TLB. Si deve azzerare, tranne le voci appartenenti all'OS, perchè avendo un identificativo univoco soltanto nell'ambito del singolo processo (numero di pagina), si creerebbero problemi ad avere voci di più processi, nel caso in cui abbiano stesso identificativo. Questo si chiama **flush**. Il flush non è molto efficiente, perchè all'inizio TLB sarà vuoto, con conseguente alta percentuale di miss. Ma soprattutto, le vecchie voci potrebbero tornare utili in caso di nuovo scheduling di quel processo, che si ritroverebbe delle sue voci già in TLB. Ovviamente, se si sceglie questa strada, bisogna trovare il modo di identificare univocamente la voce; per cui, si inserisce un altro identificativo, l'**ASID** (Address-Space Identifier), che altro non è che l'identificativo del processo a cui appartiene quella pagina. Adesso è possibile usare questa strategia. Nota: se si esegue context-switch tra thread fratelli, non bisogna modificare nulla in TLB.

### 3.3.7 Effective Access Time: TLB vs Standard

Adesso proviamo a calcolare il tempo di accesso medio effettivo ad una voce della tabella, con e senza TLB. Dipende dal tempo di accesso a TLB, ma anche dal tempo d'accesso alla RAM, perchè in caso di TLB hit faremo soltanto un accesso alla memoria, invece in caso di TLB miss dovremo farne due. Cioè: in caso di TLB hit, avremo un tempo d'accesso pari a  $t_{TLB} + t_{RAM}$ , altrimenti  $t_{TLB} + 2t_{RAM}$ . Per trovare il tempo di accesso medio effettivo, dobbiamo conoscere il **TLB ratio**, cioè la percentuale di TLB hit sul totale delle richieste. Tenendo conto che in un sistema reale il TLB ratio è pari ad almeno l'80%, il tempo d'accesso effettivo EAT sarà minore con TLB rispetto che senza, come dimostra l'esempio sotto:

- Facciamo un **esempio**:
  - tempo di accesso alla memoria = 100 nsec;
  - tempo di accesso alla TLB = 20 nsec;
- **tempo effettivo di accesso** sarà in questo caso:
  - 120 nsec per TLB hit;
  - 220 nsec per TLB miss;
- ipotizziamo un TLB ratio (percentuale di successi) dell'80%;
  - tempo (medio) effettivo di accesso:  $0.8 \times 120 + 0.2 \times 220 = 140$  nsec
- in generale:
  - **tempo di accesso alla memoria**:  $\alpha$
  - **tempo di accesso alla TLB**:  $\beta$
  - **TLB ratio**:  $\epsilon$
  - **EAT** =  $\epsilon (\alpha + \beta) + (1 - \epsilon) (2\alpha + \beta)$

Figure 42: Esempio di calcolo EAT

### 3.3.8 Tabella delle pagine multilivello

Ci rimane ancora da risolvere il problema della dimensione della tabella. E' un problema duale: se risolviamo questo, andremo inevitabilmente a peggiorare le prestazioni (il tempo d'accesso): bisogna trovare un equilibrio.

Questo non era tanto un problema quando si aveva uno spazio di indirizzamento a 32 bit: ipotizzando pagine da 4 KB ognuna, avremmo circa 1 milione di voci, la tabella occuperebbe 4 MB (4 byte per 1 milione di voci). Ma il mondo è andato avanti, e adesso la totalità delle CPU desktop/server in commercio è a 64 bit. Ci sono appunto  $2^{64}$  indirizzi disponibili: immaginate soltanto a quante pagine da 4 KB corrispondono (per la precisione:  $2^{52}$ ). Si capisce che la tabella avrebbe così tante voci da occupare più spazio di quanto ce ne sia in tutta la RAM! Che si fa? Una cosa molto semplice: si cerca di non memorizzare tutta la tabella, cioè di non rappresentare tutte le pagine, ma ricordiamo che le voci devono essere memorizzate contiguamente. Si adotta un approccio simile a quello che ci ha portato alla memoria virtuale. Cioè si divide la tabella in **gruppi di pagine**, ripartite su più livelli, con una struttura ad albero. Una semplice struttura a 2 livelli sarebbe così ripartita: la tabella di primo livello sarebbe la **tabella dei gruppi**: ogni sua voce è un puntatore ad una tabella di secondo livello (al suo primo elemento). Adesso la struttura dell'indirizzo virtuale in bit cambia: i bit più significativi del numero di pagina indicano il gruppo, gli altri la voce singola (la pagina) dentro il gruppo. I bit rimanenti sono ovviamente quelli dell'offset. Il PTBR punterà adesso alla voce della tabella di primo livello, che a sua volta punta alla tabella di secondo livello corrispondente, e da lì si può trovare la pagina. Per ogni livello in più, ci sarà una fetch in più: ad esempio, con 2 livelli, ci sono 3 fetch. Qui il vantaggio dato da TLB si fa sentire maggiormente, da 3 fetch o più si passa ad uno solo. Più i gruppi diventano piccoli, più livelli ci sono, più fetch dobbiamo fare. Ma in caso di indirizzi a 64 bit è l'unica scelta possibile, che ci salva la vita: non abbiamo più un "bloccone" contiguo: abbiamo tanti "blocchetti", che sono salvabili in memoria non contiguamente, in maniera molto simile alle pagine stesse. Questo significa che è possibile direttamente non salvare in memoria alcuni di questi blocchetti. Se in un gruppo tutte le pagine sono vuote, ad esempio i gruppi

che rappresentano la parte di memoria (virtuale) heap del processo, non ho bisogno di quella particolare tabella. La tabella di primo livello avrà un puntatore NULL in quella specifica voce. Ciò significa che in un moderno sistema a 64 bit, la maggior parte delle tabelle non è direttamente in memoria, non viene neanche creata: la memoria RAM disponibile (di solito da 8 a 32 GB in sistemi desktop) è molto minore rispetto alla memoria virtuale. Rimane da capire una cosa: quanti livelli ci sono? In Pentium l'OS decideva il numero dei livelli, ma in realtà nei sistemi moderni gli indirizzi generati dai compilatori non sono realmente a 64 bit, ma viene "tagliata" una parte, che è nulla. Per cui abbiamo indirizzi a 48 bit effettivamente, in modo da avere massimo 4/5 livelli.

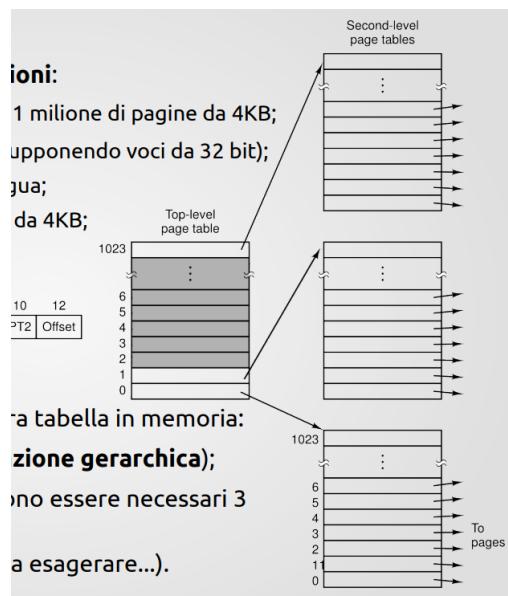


Figure 43: Tabelle a due livelli

### 3.3.9 Tabella delle pagine invertita

Esiste anche un altro approccio, ora non più usato, ma usato ai tempi dalle CPU Intel Itanium, le prime totalmente a 64 bit. Questo approccio dipende fortemente dall'hardware, perché MMU deve conoscere la struttura della tabella.

La tabella usata qui è detta **tabella delle pagine invertita**, e ne è presente solo una, senza gerarchie né altro. Si tratta in pratica di una generalizzazione della già citata tabella dei frame: una voce per ogni frame, che indica stato di allocazione del frame e anche informazioni sulla pagina associata, se il frame è stato allocato (le informazioni sono precisamente PID, numero di pagina e tutte le altre informazioni dentro una voce della tabella delle pagine standard). Se non trovo la pagina in questa tabella, si può già dire che si tratti di page fault, questo perchè quel numero di pagina non esiste nella tabella. Se ho un riscontro, ottengo subito il numero di frame. Uno dei tanti problemi di questo approccio è il fatto che la ricerca sia lineare: immaginate, con un quantitativo di RAM pari a quello dei sistemi attuali, quanto durerebbe una ricerca dentro questa tabella. Si potrebbe risolvere parzialmente implementandola con una tabella hash, o utilizzando ancora TLB (è compatibile). Un altro problema è che le voci contengono soltanto le informazioni sulle pagine allocate in memoria (è un 1:1 con la tabella dei frame abbiamo detto),

ma se voglio informazioni sulle altre pagine (disponibili ma non ancora associate ad alcun frame) mi serve comunque una tabella delle pagine; questo in caso di page fault per esempio, è proprio il caso in cui mi servono informazioni sulle altre pagine. Mettiamo la tabella delle pagine in disco, ma ci sarebbero dei rallentamenti. Comunque, questo approccio è abbastanza problematico: ad esempio, la maggior parte degli OS in commercio è tarato sull'approccio multatabella, non facile adattarli a questo; per questi motivi, è stato abbandonato.

### 3.4 Piazzamento della cache della memoria

Quando si accede alla cache "originale" (quindi no TLB, la cache della RAM), prima o dopo l'accesso alla MMU? In realtà si possono scegliere entrambe le strade, con certi vantaggi e svantaggi. Se piazzata prima della MMU, per forza di cose dovrà usare indirizzi virtuali, perché ancora non è stata fatta la traduzione, mentre se viene piazzata dopo, può usare gli indirizzi fisici, perché al momento dell'ingresso nella cache si ha già a disposizione l'indirizzo fisico tradotto. Cosa cambia precisamente?

- **Dopo MMU (indirizzi fisici):** la CPU genera l'indirizzo virtuale, viene passato alla MMU che traduce. Viene interrogata la cache con l'indirizzo fisico. Se c'è hit, viene recuperato il dato e fine. Dato che abbiamo indirizzi fisici, per definizione univoci, nella cache può esserci "roba" di tutti i processi. Il loro identificativo sarà appunto l'indirizzo fisico del dato in RAM. Il problema è che il caching è lento, perchè l'interrogazione viene fatta dopo la traduzione da parte dell'MMU, che fa da bottleneck, è abbastanza lenta;
- **Prima di MMU (indirizzi virtuali):** Appena dopo la generazione dell'indirizzo virtuale, si interroga subito la cache. Se c'è hit, il dato viene recuperato e l'interrogazione finisce, senza passare neanche dall'MMU. Si capisce che così il caching è molto più veloce, non c'è nessun collo di bottiglia, nessuno in mezzo. Ma qual è il problema? Lo stesso che avevamo con TLB: l'univocità. Non possiamo usare l'indirizzo virtuale come identificativo univoco avendo più processi in cache, più processi possono usare lo stesso indirizzo virtuale. Allora si possono applicare le stesse soluzioni viste con TLB: il flush, oppure l'ASID: la scelta più conveniente è ovviamente l'ASID, già abbiamo spiegato il motivo.

Nella pratica, in un sistema moderno, useremo un mix di tutti e due i metodi. Ricordo che abbiamo diversi livelli di cache, di solito 3: L1, L2, L3. La cache L1 deve essere quella più veloce, quasi immediata: la mettiamo prima di MMU. La capienza sarà limitata (c'è pure ASID che occupa spazio). Le cache di livello più alto, che dovranno essere capienti (quindi mettere ASID non va bene, prende spazio), ma se sono più lente fa nulla, saranno posizionate dopo MMU.

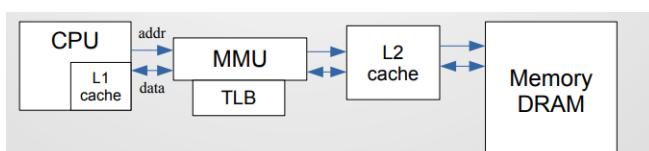


Figure 44: Cache in un moderno PC

## 3.5 Algoritmi di sostituzione delle pagine

Quando si verifica un page fault, è l'hardware (MMU) a rilevarlo, leggendo il bit di presenza di quella pagina. Genera un'eccezione, "passando la palla" all'OS, che chiamerà una routine di gestione del page fault, la quale altro non dovrà fare che spostare la pagina (che ha causato page fault) in RAM. Problema: se i frame sono tutti pieni? Semplice (come in caso di cache piena): bisogna scartare (rimettere in disco) una pagina, in modo da liberare un frame. Il criterio in base a cui scegliere la pagina da scartare è in generale uno, cioè si sceglie la pagina più "sacrificabile", cioè la pagina che, se tolta, causerà meno page faults. Ci sono diversi **algoritmi di sostituzione delle pagine** che faranno questo lavoro, implementati dall'hardware; implementarli via software sarebbe troppo inefficiente.

Che scelte dovranno effettuare questi algoritmi, per essere efficaci? La scelta più sbagliata sarebbe quella di scartare una pagina che verrà usata appena successivamente dalla procedura contenuta nella pagina a cui abbiamo fatto spazio. Mentre la scelta migliore sarebbe ipoteticamente quella di scartare una pagina che non verrà referenziata mai più, o più in generale, la pagina che sarà referenziata il più avanti possibile. L'algoritmo (puramente teorico) che effettua questa scelta è detto **OPT** (ottimale), che però può essere usato come termine di paragone, per verificare l'efficienza di quelli reali: rappresenta un *lower bound*, non si può fare meglio. Perchè è puramente teorico? Potenzialmente potremmo implementarlo, associando ad ogni pagina un'etichetta contenente il numero di istruzioni che saranno eseguite PRIMA di referenziarla. Sarebbe scartata quella col numero più alto; e avremmo anche tutte le informazioni necessarie per riempire queste etichette (l'intero codice del processo, e il loro stato), ma per simulare tutti gli scenari ci vorrebbe una quantità di tempo enorme.

### 3.5.1 Not Recently Used (NRU)

Questo algoritmo scarta una pagina che non è stata usata di recente. Una domanda: come fa a sapere che non è stata usata di recente? Divide le pagine in 4 classi, in base a due bit contenuti nelle voci della tabella delle pagine: **referenziamento (R)** e **modifica (M)**. Abbiamo già spiegato cosa sono. Il bit più significativo è il bit R, perché l'algoritmo si basa su un'idea molto semplice: dato che il bit R si azzerà periodicamente, se è ad 1 la pagina è stata referenziata (consultata, "usata") di recente, non avrebbe senso toglierla, probabilmente verrà referenziata di nuovo nel prossimo futuro (criterio alla base di molti algoritmi di sostituzione pagina). Mentre a parità di referenziamento, preferisco non scartare una pagina che è stata modificata, dovrei aggiornare anche la copia in disco, mentre in caso contrario la tolgo dal frame e basta. L'algoritmo intanto sceglie da quale classe prendere la pagina: la classe più bassa, non vuota.

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

Figure 45: le 4 classi di NRU

Dentro la classe, solitamente la pagina singola si sceglie seguendo un normalissimo algoritmo di tipo FIFO. Questo perchè così facendo, viene scartata la pagina più "vecchia", cioè da più tempo in RAM.

Questo algoritmo non è molto efficace, perchè non considera QUANTO una pagina è stata usata, ma solo SE è stata usata. Per cui, immaginiamo uno scenario in cui ci sono due pagine referenziate di recente, ma una è stata referenziata in totale 1000 volte, l'altra soltanto 2 volte, e magari la prima è la più vecchia. Verrebbe scartata quella, non ha molto senso, ci saranno molti page faults.

### 3.5.2 FIFO e Seconda Chance, Clock

Posso prendere in considerazione soltanto l'età delle pagine, ed applicare soltanto l'algoritmo FIFO, senza suddivisione in classi: sarà scartata la più vecchia (in generale). Già prima abbiamo fornito una motivazione per cui questo approccio non è efficace: una pagina vecchia potrebbe essere comunque molto usata. Ma possiamo capire all'incirca se è molto usata: se lo è, sicuramente è stata referenziata di recente, ha il bit R ad 1. Allora, quando viene applicato il criterio FIFO, cioè scartata la testa, prima si guarda il bit R di essa; se è 1, viene "risparmiata", messa in coda, e il suo bit R azzerato, come se fosse la pagina più "giovane". Verrà fatto lo stesso controllo per ogni nuova testa: appena viene trovata una testa con bit R a 0 (pagina relativamente "vecchia", e anche poco usata, perfetto) e viene scartata quella. Potenzialmente potrebbe essere la vecchia testa, già risparmiata una volta. Per questo motivo, viene detto algoritmo della **seconda chance**.

Una variante più efficiente della seconda chance è l'algoritmo **Clock**. Useremo una coda circolare, con ogni nodo avente un puntatore al successivo, e un puntatore alla testa: possiamo vederla come un orologio (circolare) con una lancetta (il puntatore alla testa). Questo renderà più efficiente lo spostamento della testa in coda, semplicemente spostando la lancetta al nodo successivo. Così la nuova testa sarà il successivo, e la vecchia testa sarà la coda, esattamente come prima; ma qui basta aggiornare il valore di un puntatore, lì bisognava effettuare un'estrazione e un inserimento.

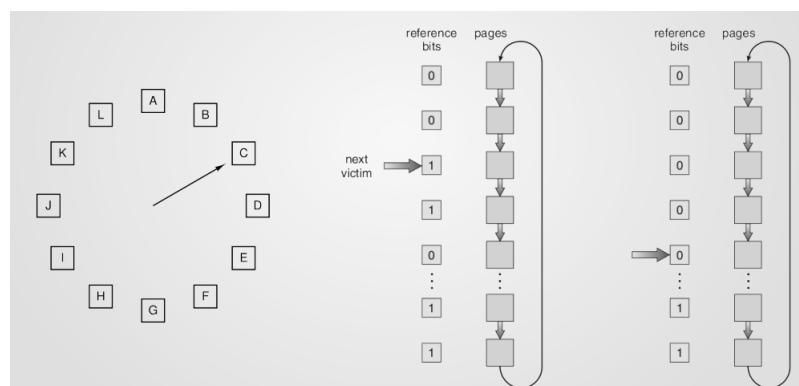


Figure 46: La coda nell'algoritmo Clock

### 3.5.3 Least Recently Used (LRU)

Questo algoritmo viene usato anche nelle comuni cache di memoria. Si scarterà la pagina usata meno di recente, cioè la pagina che non viene toccata da più tempo. A differenza degli altri, non si basa sul tempo che la pagina ha trascorso in RAM, perché non è un indicatore abbastanza affidabile, ma va più nello specifico: la regola è, che se una pagina è stata referenziata molto recentemente, probabilmente verrà referenziata di nuovo in un futuro prossimo. Se invece una pagina non viene referenziata da molto, probabilmente non sarà referenziata per tanto tempo ancora. Questo algoritmo (molto dispendioso da implementare e da eseguire, richiede un contatore a 64 bit in hardware) richiede di tenere un timestamp (contatore) che parte (si azzerà cioè) dall'istante in cui viene referenziata la pagina la prima volta. Ad ogni istruzione verrà incrementato. Questo contatore verrà inserito nella voce della tabella dei frame, perché ovviamente ha senso usarlo solo per le pagine già in memoria. Il timestamp viene aggiornato dall'MMU, perché se dovesse farlo l'OS ci sarebbe una store per ogni aggiornamento (overhead elevato). Ogni volta che la pagina viene referenziata, il contatore viene scritto nel record corrispondente della tabella delle pagine. Ciò significa che una pagina referenziata molto in là nel passato avrà un contatore non aggiornato, molto basso. Viene quindi scartata la pagina con il contatore più basso.

Il contatore, come detto prima, può essere un campo della voce della tabella dei frame, oppure una **matrice di bit**. Come funziona questa matrice? E' una semplice matrice quadrata, in cui ogni riga (e colonna) rappresenta un frame: precisamente, il vero e proprio contatore sarà la riga; adesso spiegheremo il motivo per cui serve pure la colonna. Appena viene referenziata la pagina contenuta nell'i-esimo frame, la riga di quel frame (i-esima riga) viene messa tutta ad 1, e azzerata la i-esima colonna. Perchè questo? Perchè questo fa in modo che ad ogni referenziamento di una qualche pagina, le altre pagine "perdano" un bit 1. Ciò significa che più non viene referenziata una pagina, più la sua riga sarà piena di zeri, mentre una che viene referenziata si ritroverà tutti 1 tranne un bit che sarà a 0. Per cui, la pagina con più zeri sarà quella ad essere scartata al momento del page fault. Il numero di bit 1 determina il "peso" della pagina, cioè quanto di recente è stata utilizzata.

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	1	1	1	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	1	1	0	0	1	1	0	0	0	1	0	0	0
2	0	0	0	0	0	0	0	0	1	1	0	1	1	1	0	0	1	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0
				→frame 0				→frame 1				→frame 2			→frame 3				
					0	0	0	0	0	1	1	1	0	0	1	0	0	0	0
					1	0	1	1	0	0	1	1	0	0	0	1	1	0	0
					1	0	0	1	0	0	0	0	0	0	0	1	1	0	0
					1	0	0	0	0	0	0	0	0	0	0	1	1	1	0
					→frame 1				→frame 0				→frame 3			→frame 2			→frame 3

Figure 47: Matrici di bit all'opera

### 3.5.4 Not Frequently Used (NFU)

Abbiamo detto che LRU è piuttosto dispendioso da implementare, anche via hardware. Necesitiamo di approssimarla per poterla implementare via software: una prima approssimazione,

abbastanza rozza alla fine, è l'algoritmo **Not Frequently Used** (NFU), che come dice il nome, si basa sull'idea di scartare la pagina usata meno di frequente. Cioè ad ogni pagina in memoria (frame) viene associato sempre un contatore, che viene aggiornato periodicamente, però in modo diverso; precisamente, ad ogni riferimento, viene chiamata una procedura che somma il contatore al bit di referenziamento R. A cosa porta ciò? Ricordiamo che R viene azzerato anch'esso periodicamente, quindi fornisce un'indicazione sul fatto che quella pagina sia stata usata o meno recentemente. Per cui: prima di ogni azzeramento, viene fatta questa somma. Se R è spesso ad 1, si capisce che il contatore arriverà a valori alti, viceversa se è spesso a 0. Una pagina con contatore basso è stata usata poco, verrà scartata la pagina con il contatore più basso, cioè la pagina che è stata usata "meno di frequente". Ovviamente il contatore non conta il numero di accessi alla pagina, perché R rimane sempre ad 1, anche se ci sono più accessi in quel lasso di tempo.

Il problemino principale di questo algoritmo, che lo rende un'approssimazione "rozza" come dicevamo, è che non "pesa" gli eventi nel tempo: vorremmo che una referenziazione recente pesi di più di una fatta nel passato remoto. In presenza di una pagina usata molte volte nel passato, ma non più usata di recente, avrà un contatore molto alto, e non sarà comunque scartata. Questa è una scelta infelice, perché molto probabilmente questa pagina non sarà usata neanche nel prossimo futuro. Diciamo che la pagina "vive di rendita" ("si è fatta la nomina").

### 3.5.5 Algoritmo di Aging

Modifichiamo l'NFU, rendendolo più simile all'LRU, quindi aggiungendo un "peso" in base agli istanti di tempo in cui è referenziata la pagina. Cosa intendo? Il contatore viene aggiornato in maniera diversa: si shifta di una posizione verso destra, facendo diventare il bit più significativo nullo, e si copia il bit R in questa posizione. In pratica, si "tiene traccia" soltanto degli ultimi N cicli, istanti (N sarebbero i bit del contatore), e se la pagina è stata usata ultimamente, il contatore aumenta il suo valore di molto. Ad esempio, una pagina con contatore 10000000 sarà risparmiata ad una pagina con contatore 01000011, anche se quest'ultima è stata usata più volte. Conta anche quando è stata utilizzata, abbiamo aggiunto il concetto di "peso". Questo shift non deve essere fatto troppo di frequente, per non causare elevato overhead. Come prima, non sappiamo QUANTE volte la pagina è stata usata, ma semplicemente sappiamo che in quel dato istante la pagina è stata usata almeno una volta. Però a noi interessa soltanto scartare pagine che sono state usate molto tempo fa, che è il principio dell'LRU. Infatti una pagina con contatore basso non viene usata da tanto tempo, ha i bit più significativi a 0.

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

Figure 48: Esecuzione dell'aging

### 3.5.6 Confronto delle prestazioni

Dopo aver spiegato il funzionamento dei principali algoritmi di sostituzione delle pagine, confrontiamo le loro prestazioni. Che metrica useremo? Ci basiamo sul **numero di page faults**. Ogni algoritmo ha l'obiettivo di fare scelte che riducano al minimo questo numero. Precisiamo che a parità di numero di processi e dimensione dei frame, il numero di faults diminuirà all'aumentare della memoria RAM (ovvio, dai).

Adesso, facciamo una simulazione. Abbiamo a disposizione 3 frame in memoria fisica, e viene data in input una sequenza di numeri di pagina, che rappresenta la sequenza degli accessi alle pagine. Ai termini della simulazione non ci interessa l'offset, cioè la singola locazione, ragioniamo soltanto in termini di pagina, d'altronde è quello che vedono gli algoritmi. La sequenza non avrà doppioni consecutivi: se la pagina non c'è la prima volta, c'è fault e viene messa in memoria, le altre volte ci sarà; se la pagina c'è già la prima volta, nessun fault. Quindi avendo N doppioni, ci interessa soltanto cosa succede la prima volta, li consideriamo come un accesso solo.

Vabbè, partiamo con la simulazione. Abbiamo questa sequenza:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Come termine di paragone usiamo l'algoritmo ottimale, OPT, sapendo che meglio di questo non possiamo fare.

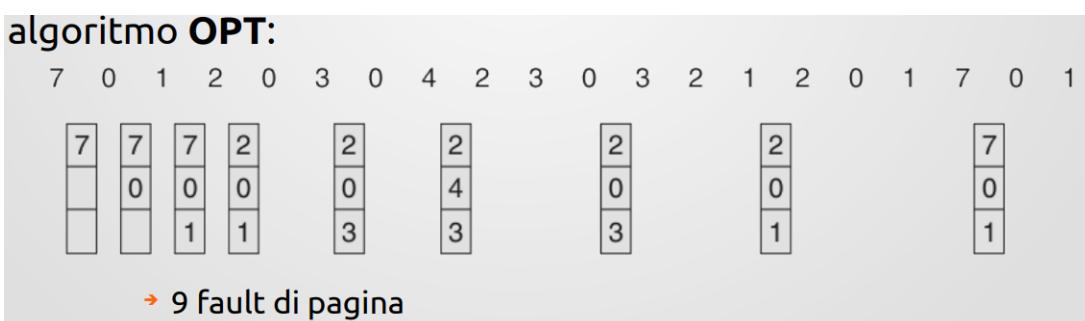


Figure 49: Simulazione con OPT

9 page faults per OPT. Già sappiamo che meglio di 9 non possiamo fare: OPT è un algoritmo teorico, fa delle scelte "prevedendo il futuro". Nel senso: OPT sceglie la pagina che verrà referenziata più in là nel futuro. Qui lo possiamo fare, abbiamo la sequenza davanti agli occhi, nella realtà non funziona così, non si può viaggiare nel tempo.

Adesso applichiamo l'algoritmo FIFO:

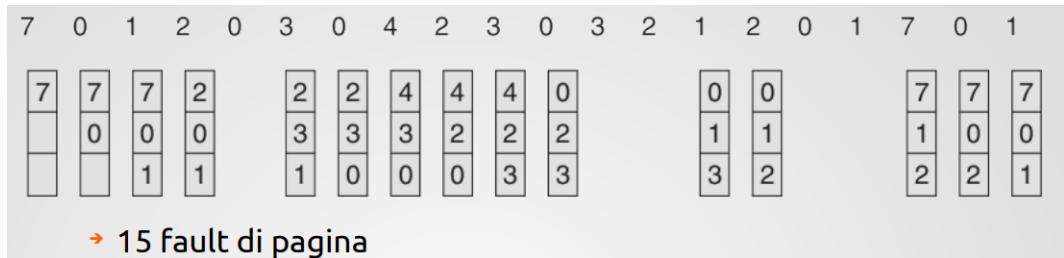


Figure 50: Simulazione con FIFO

FIFO è decisamente peggiore dell'OPT: ben 15 page faults. Ma non è questo il problema: prendendo una particolare sequenza, precisamente

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

notiamo una cosa molto strana. Vediamo che, aumentando la RAM disponibile, paradossalmente il numero di faults aumenta.



Figure 51: Anomalia di Belady

Questo fenomeno si chiama **anomalia di Belady**, capita soltanto con alcune sequenze. Da notare che non si presenta con tutti gli algoritmi. Ad esempio, come vedremo, non si presenta applicando LRU.

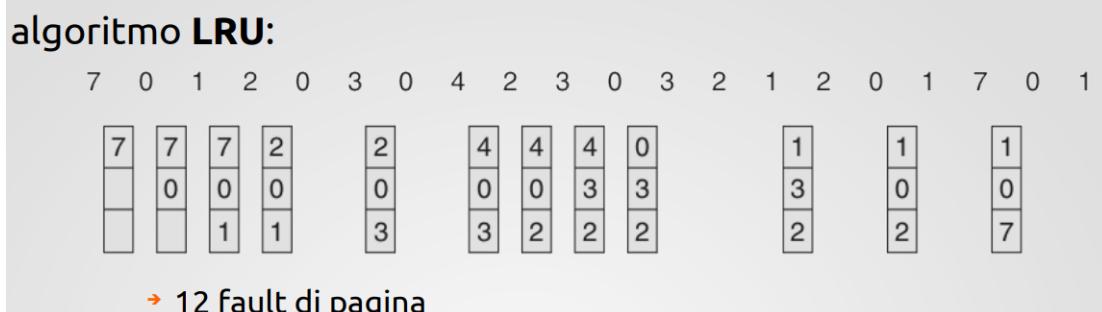


Figure 52: Simulazione con LRU

LRU si comporta decisamente meglio di FIFO: 12 faults, e non soffre dell'anomalia di Belady, questo perchè negli algoritmi basati su LRU (quindi anche NFU e Aging) vale la **proprietà di inclusione**: l'insieme delle pagine in memoria in un dato istante, avendo N frames, è un sottoinsieme dell'insieme delle pagine in memoria in quello stesso istante, però avendo N+1 frames. Questo è permesso dalla natura delle scelte fatte da LRU: in memoria rimangono soltanto le pagine usate più di recente, per cui, avendo un frame in più, in quel frame ci sarà sicuramente una pagina più vecchia: le altre ci sarebbero comunque anche avendo un frame in meno, sono un sottoinsieme. FIFO invece si basa sull'età della pagina: potrebbe toccare anche pagine usate di recente, se sono state inserite in memoria tanto tempo fa.

Capiamo una cosa: gli algoritmi LRU-based non soffrono dell'anomalia, quelli FIFO-based (FIFO, seconda chance, clock) invece sì. C'è da dire che in casi di utilizzo reale non ci importa dell'anomalia: il numero di frame è fisso.

## 3.6 Allocazione dei frame

Pensiamo ad un altro problema: come allocare i frame? Cioè, quanti frame devo assegnare ad ogni processo? E, dopo avere i frame a disposizione, quali pagine ci vado ad inserire all'inizio, non potendole inserire tutte? Rispondiamo alla seconda domanda: non si scelgono pagine da inserire a priori nei frame: si lasciano vuoti. Si applica concetto di **demand paging**: quando non trovo una pagina, c'è fault e la inserisco. Ovviamente all'inizio ci saranno un sacco di faults. Torniamo alla prima domanda: quanti frame assegno ad un processo?

- Potrei assegnargliene il minimo permesso dall'architettura in uso: non è la migliore scelta. Potrebbe non bastare, causando diversi problemi che vedremo fra un po'. Il minimo è deciso in base al tipo di istruzioni che quell'architettura esegue: capiamolo con un esempio.

Prendiamo un'architettura che permette di eseguire l'istruzione Assembly  
MOV [1000], #1

Bisogna prelevare l'istruzione, e potrebbe esserci già un page fault, magari la pagina che la contiene è in disco. Dopo, bisogna effettivamente accedere alla locazione 1000 per scriverci 1, ma magari la locazione 1000 è in una pagina che è in disco. Già due page faults solo per questa istruzione: se avessimo un frame soltanto sarebbe impossibile eseguirla. Per cui ci vuole un numero minimo di frame obbligatorio.

- Gli assegno tutta la memoria libera in quel momento, diciamo che sia un po' utopistico.

Trasferiamoci in uno scenario reale, in cui abbiamo M frame da spartire tra N processi.

- Allocazione equa: assegno  $M/N$  frame ad ogni processo. Ma devo tenere conto della **taglia** del processo, cioè quanto spazio occupa. Ciò mi obbliga ad assegnare più spazio a processi più grandi e viceversa.
- Si arriva all'allocazione proporzionale: assegno i frame in base alla taglia del processo. Ad un processo piccolo assegno pochi frame; ad un processo grande assegno tanti frame,

semplice.

Assumendo che  $S_i$  è la taglia dell'i-esimo processo, ne dovrò assegnare

$$A_i = \frac{S_i}{S \cdot M}$$

dove  $S = \sum S_i$ .

- Allocazione per priorità: anche se un processo è piccolo, potrebbe avere alta priorità, per cui gliene assegno comunque tanti: deve essere il più efficiente possibile, riducendo al minimo i page faults.

### 3.6.1 Allocazione locale, globale, thrashing, località

Gli algoritmi di sostituzione delle pagine si possono applicare sia a livello **locale**, cioè considerando soltanto le pagine del processo che ha causato il fault; oppure a livello **globale**, cioè considerando tutte le pagine in memoria. Nel caso globale, può mutare il numero di frame posseduti da ogni processo: ad esempio, se un processo P1 causa un page fault, potrebbe essere scartata una pagina di P2: P2 perde un frame, P1 ne acquista uno. Quindi possiamo dire che il tutto si autobilancia, cioè se un processo necessita tanti frame, se li "conquisterà", a discapito di altri processi che però ne hanno bisogno pochi.

Cosa succede se un processo ha assegnati pochi frame?

- Ne ha poco più del minimo: si parla di **thrashing**, cioè si verificano così tanti page faults da rallentare molto l'esecuzione del processo, e anche degli altri. Precisamente, si parla di thrashing quando si usa più tempo per gestire i faults che per l'effettiva esecuzione delle istruzioni.
- Ne ha meno del minimo: il processo deve essere inevitabilmente sospeso, e swappato tutto in disco, o direttamente killato in casi particolari (tipo allocazioni infinite).

Come possiamo evitare il thrashing? Bisogna commisurare il numero di frame alle esigenze del processo. Non intendiamo la taglia: intendiamo la **località** del processo. Cosa si intende per località? Con località ci si riferisce a tutti i dati e le istruzioni che in un dato istante il processo ha bisogno di utilizzare; si capisce che il concetto di località è dinamico, variabile nel tempo. Possiamo dire quindi che un processo, in un dato istante, ha bisogno di avere la sua località in memoria, e dobbiamo garantirgliela. Quindi non va bene dare un quantitativo fisso di frame ad un processo: questo quantitativo deve variare in base alla località.

### 3.6.2 Working set

Per poter fare quello di cui parlavamo prima, per forza di cose dobbiamo sapere quanto sia "grande" effettivamente questa località, e lo dobbiamo sapere in ogni istante. Usiamo il metodo del **working set**, cioè si controlla la sequenza degli ultimi  $\Delta$  accessi alla memoria (sempre in termini di pagine ovviamente). Guardando la sequenza passata, dobbiamo vedere QUALI pagine sono state referenziate in questo lasso di tempo, e metterle in un insieme, detto appunto

working set. Bisogna scegliere un  $\Delta$  abbastanza grande (non troppo) da avere una "visione" totale della località del processo. Con cardinalità del working set possiamo vedere se il processo è in sofferenza: se lo è, gli si assegnano altri frame, e viceversa. Possiamo prevenire il thrashing: ce ne accorgiamo prima. Se la sofferenza è generale, possiamo farci poco: c'è poca RAM. Ma comunque possiamo capire se c'è: si sommano tutte le cardinalità dei working set, e si confrontano con il numero di frame disponibili. Se questo numero è maggiore, bisogna inevitabilmente swappare qualche processo, anche nella sua interezza.

Come faccio a calcolare, approssimandolo, il working set? Ci serve il solito bit R. Terremo un **log** dei valori che ha assunto R, negli ultimi  $\Delta$  cicli, in modo da vedere come varia nel tempo. Avremo un log per ogni pagina: mettendoli insieme, possiamo ottenere un effettivo working set, per cui possiamo vedere quali pagine sono state utilizzate, se nel log c'è almeno un 1. Se sono tutti 0 ovviamente, quella pagina non è utilizzata. Viene chiamato un interrupt, periodicamente, e viene preparato questo log.

### 3.6.3 Page Fault Frequency

Possiamo monitorare il "fabbisogno" del processo anche in un modo più diretto: misuriamo la frequenza dei page faults causati da esso. Un processo in sofferenza avrà più faults della media, se "sta bene" ne avrà meno. Monitorando le variazioni di questa frequenza possiamo prendere di conseguenza delle decisioni. In dettaglio, si stabiliscono due soglie: se la Page Fault Frequency supera la soglia superiore, il processo è considerato in sofferenza, e gli verranno assegnati nuovi frame; questo perché probabilmente si è verificato un cambio di località, e la nuova località è più grande di quella precedente. Se invece scende sotto la soglia inferiore, gli verranno tolti dei frame: la nuova località è più piccola della precedente, il processo non ha bisogno di nuovi frame. Il meccanismo si autobilancia, un processo in sofferenza otterrà dei nuovi frame, probabilmente tolti ad un processo con la PFF sotto soglia. Con una frequenza intermedia (in mezzo alle due soglie) il processo non subirà variazioni di frame, è ancora nella stessa località.

Il monitoraggio della PFF e il calcolo del working set sono praticamente equivalenti. Entrambi i metodi ci permettono di "delimitare" le località e rilevare la sofferenza del processo (thrashing), semplicemente lo fanno in due modi diversi; sono anche collegati. Perchè?

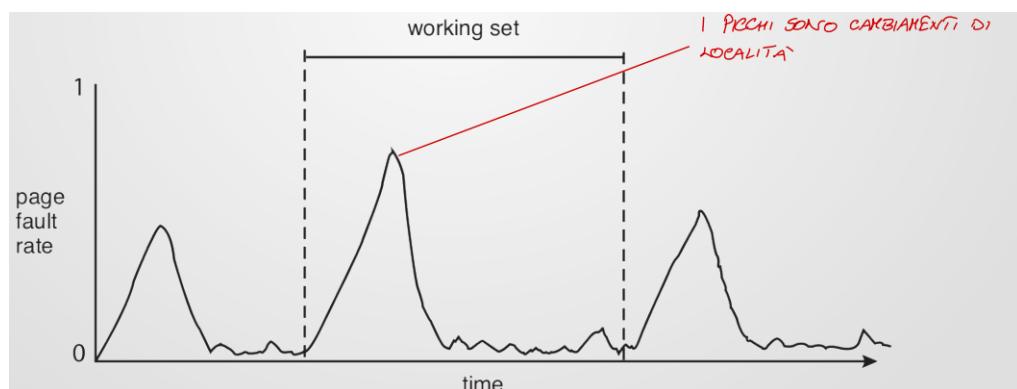


Figure 53: Variazione della PFF nel tempo

Guardiamo questo grafico: possiamo distinguere delle "fasi", delimitate dai picchi. Questi

picchi indicano il cambiamento di località; cambiano codice e strutture dati, per cui devo andarli a recuperare dal disco (quasi sicuramente non saranno già in memoria, prima non servivano) causando tanti faults in poco tempo. Con le pagine disponibili in memoria, la frequenza si abbassa e si assesta, fino al prossimo cambiamento di località. Qual è il collegamento con working set? La fase picco + assestamento non è altro che un working set, rappresenta un'intera località.

### 3.6.4 Politica di pulitura dei frame

Negli OS moderni vengono applicate tante strategie per ottimizzare la gestione dei page fault. Distinguiamo due casi:

- Ci sono dei frame liberi;
- Tutti i frame sono occupati.

Il secondo caso è complesso da gestire. Dobbiamo applicare l'algoritmo di sostituzione delle pagine, e altre cose. Nel primo caso, basta leggere la pagina in disco, scriverla in un frame e aggiornare le tabelle delle pagine/frame. Idea: facciamo in modo che la maggior parte delle volte ci si trovi nel primo caso. Quindi si cerca di costruire una **scorta** di frame sempre e comunque liberi, ma come mantenerla? Se ne occupa il **paging daemon**. Questo demone viene chiamato periodicamente, precisamente quando il numero dei frame facenti parte della scorta cala sotto una certa soglia, e libererà alcuni frame. Sceglierà i frame da liberare applicando globalmente un algoritmo di sostituzione delle pagine, in modo da scegliere le pagine più "inutili" e liberare i rispettivi frame; i processi proprietari non se ne accorgeranno, quelle pagine erano ormai abbastanza inutili per loro. I frame liberati saranno parte della scorta. Il demone viene chiamato solo in periodi in cui il sistema è in idle, cioè periodi in cui la CPU ha bassa percentuale di utilizzo.

Ma conviene così tanto ridurre il tempo di gestione del page fault? Sì, perché quando si verifica un page fault, il processo rimane bloccato finché la pagina non è in RAM. Se riduco durata della gestione del page fault, si riduce tempo in cui il processo è bloccato.

Possiamo limitare molto i danni anche in caso l'algoritmo faccia una scelta sbagliata (pagina scartata che viene richiesta subito dopo): un frame viene contrassegnato come libero, al momento in cui la pagina viene scartata; ma in realtà il contenuto del frame (la pagina) non viene mica tolta dal frame: sarà sovrascritta solo nel momento in cui il frame viene preso da un nuovo processo. Quindi se si verifica un fault del genere, e il frame è ancora libero, c'è la "roba vecchia" ancora dentro. Quindi basta "ripescare" il frame, cioè aggiornare le tabelle delle pagine/frame. In pratica noi applichiamo l'algoritmo anche prima di quando effettivamente necessario, per scegliere il frame da pulire, tanto in caso di una scelta sbagliata possiamo rimediare facilmente e velocemente.

### 3.6.5 Dimensione della pagina

Come si stabilisce la dimensione della singola pagina (quindi del frame)? Si può scegliere tra un range ben preciso, delimitato dall'hardware. L'OS sceglierà una potenza del 2, molto più facile

ed efficiente da gestire (basti pensare agli shift).

Cosa succede se scelgo di avere delle pagine "grandi"?

- **Le tabelle saranno più piccole:** con pagine più grandi, a parità di memoria e spazio di indirizzamento virtuale, si avranno meno frame e meno pagine; una pagina/frame "copre" più spazio. Per cui avremo meno voci nelle tabelle.
- **Operazioni I/O più efficienti:** in particolare con dischi meccanici (HDD). Per capire il motivo, bisogna sapere, almeno in maniera superficiale, il funzionamento di un HDD. Il costo principale di una lettura dipende dal tempo di posizionamento della testina sul blocco. Se la testina deve leggere soltanto dei blocchi adiacenti, si posizionerà una volta sola, dal primo blocco, e li legge tutti in sequenza. Se deve leggere dei blocchi sparsi, deve ogni volta posizionare la testina in un posto diverso, e ciò costa molto. Come ben sappiamo, i dati di una pagina sono contigui, adiacenti. Quindi con una pagina più grande si hanno più blocchi contigui in disco, la testina dovrà riposizionarsi meno volte, quindi l'efficienza della lettura da disco aumenta. Stesse considerazioni ovviamente valgono per la scrittura.
- **Minor numero di page fault:** con una pagina più grande, si possono portare più dati in memoria, a parità di pagine portate. Per questo aumenta la probabilità di trovare ciò che si cerca, logicamente.

Se però scelgo una pagina più piccola, c'è il vantaggio di poter ridurre la frammentazione interna alla pagina. Per forza di cose non verranno usate tutte le locazioni della pagina: se una pagina è piccola, probabilmente ne rimarranno poche vuote. In media, rimane vuota circa metà della pagina.

Da questo deriva un altro vantaggio: possiamo definire meglio la risoluzione del working set, cioè possiamo identificare meglio la località, dato che effettivamente la maggior parte della pagina è fatta da "roba" che effettivamente servirà al processo, e non locazioni vuote.

Possibilmente la dimensione della pagina deve essere un multiplo della dimensione del blocco in disco, in modo da non dover leggere parzialmente un blocco.

## 3.7 Pagine condivise

Fino ad ora abbiamo dato per scontato che ogni processo abbia le sue pagine, e che esse siano diverse tra loro. Ma se due o più processi avessero delle pagine che sono identiche nel loro contenuto? E' un caso molto frequente, basta pensare a più istanze in esecuzione dello stesso programma (lanciato più volte). Quindi di solito si tratta delle pagine che contengono il codice. Non avrebbe senso caricare più copie della stessa identica pagina in memoria: si carica una sola copia, e le tabelle delle pagine di tutti questi processi punteranno allo stesso frame, e fino ad ora va benissimo, perchè il codice è in sola lettura. La stessa cosa può funzionare anche quando si tratta di strutture dati, magari usate come strumento di IPC. Gli indirizzi virtuali delle pagine uguali saranno diversi, per forza di cose, ma punteranno comunque allo stesso frame. Un altro caso: la fork(). I processi figli saranno identici al padre: ha senso creare delle copie identiche

di OGNI pagina del padre? I figli punteranno ai frame già del padre. Qui il problema sta nel fatto che, dopo una exec(), o altro, il contenuto delle pagine dei figli cambierà, non vogliamo che questi cambiamenti li veda anche il padre (sarebbero modificati i suoi frame). Poi vedremo come risolvere questo problemino.

Concentriamoci sul problema grosso: gli indirizzi virtuali sono diversi. Si parla di **aliasing**, i due processi vedono la stessa cosa, ma la chiamano con due nomi diversi. Prima ne abbiamo parlato in maniera tranquilla, ma i problemi si presentano con le cache "pre-MMU" (passatemi il termine), quelle cache che usano indirizzi virtuali. Capiamo con un esempio perché è un problema. P1 cerca una pagina nella cache. La cerca usando la chiave della pagina, che è ASID + indirizzo virtuale, cioè (P1, 1000000). Nella cache c'è una voce con questa chiave, P1 ha trovato la pagina. Tutto bene fino ad ora. P2 cerca la stessa pagina, ma userà una chiave diversa, perché nel suo spazio di indirizzamento virtuale ha un indirizzo diverso. Userà la chiave (P2, 500000). Che succede? Che non c'è nessuna voce con questa chiave, c'è cache miss. Ma quella pagina c'è, è quella con chiave (P1, 1000000). Dopo il miss la pagina con la chiave (P2, 500000) sarà messa in cache: ci saranno due pagine identiche, ma con due chiavi diverse. La preoccupazione principale non è tanto per lo spazio, quanto per la sincronizzazione; la risorsa è condivisa, deve essere sincronizzata tra i due processi, devono avere a disposizione la STESSA versione della pagina, e ognuno deve vedere le modifiche fatte dall'altro.

La soluzione più stupida è quella di disattivare la cache per le pagine condivise, ma capite che non è la cosa più conveniente da fare, non posso privarmi della cache per così tante pagine. Quella più intelligente è l'uso di cache con **tag fisici**. Adesso cerchiamo di capire cosa sia effettivamente questo tag fisico. Noi sappiamo che un indirizzo fisico e un indirizzo virtuale condividono un "pezzo", un gruppo di bit: l'offset. Cambiano soltanto i bit più significativi, quelli che identificano il numero di pagina/numero di frame. Per cui, definiamo l'offset come tag fisico in questo caso, e usiamolo per effettuare una ricerca "preliminare" nella cache IN PARALLELO alla classica traduzione virtuale->fisico operata da MMU. Questa ricerca preliminare permette di velocizzare le operazioni rispetto ad una cache con indirizzi fisici (non perdiamo quindi tutti i vantaggi di una cache con indirizzi virtuali), perché se non viene trovata nessuna voce con lo stesso tag fisico (offset), significa che c'è sicuramente un miss, e non perde tempo. Altrimenti, vengono scremate alcune voci, e al momento dell'arrivo dell'indirizzo fisico tradotto (abbastanza presto con TLB), è più veloce ricercarlo dentro la cache dato che non devo guardare tutte le voci, ma solo quelle che mi sono rimaste dalla ricerca preliminare.

Possiamo capire che questo tipo di cache è più lenta e costosa rispetto ad una normale cache con indirizzi virtuali, ma è comunque più veloce di una cache con indirizzi fisici, e il vantaggio di avere pagine condivise è troppo grande per rinunciarci.

Le pagine condivise danno problemi anche in caso di utilizzo di una tabella delle pagine invertita (probabilmente uno dei motivi per cui è stata abbandonata). Essendo praticamente una tabella dei frame, abbiamo lo stesso problema della cache: una stessa risorsa è vista da due processi con due chiavi diverse. Quindi si verificano "falsi" page fault, la pagina in realtà c'è ma ha un altro nome. Qui il problema è che dobbiamo andare necessariamente a consultare la tabella delle pagine classica per vedere che le due pagine puntano allo stesso frame (uno dei difetti di questa tabella, lo spiegavamo già prima). Ogni volta, devo andare a modificare il record nella tabella invertita con la nuova chiave. E' troppo lento, e in caso di CPU multicore, addirittura

inutile, perchè se i processi sono su core diversi, serve molto più lavoro di correzione.

### 3.7.1 Copy-on-write

Avevamo accennato a possibili problemi che potevano verificarsi nel caso in cui si accede ad una pagina condivisa in scrittura, dobbiamo risolverlo senza perdere i vantaggi offerti dalle pagine condivise. Allora, prendiamo il caso di due processi che condividono una pagina, ma di cui devono avere una copia senza modifiche da parte dell'altro; cioè, ognuno deve avere la sua copia "personale". Che si fa? Finchè la copia non viene scritta, tutto a posto. Ma se viene scritta? Semplice: a questo punto, viene creata una copia IDENTICA della pagina condivisa, che viene scritta dal processo che ha richiesto la scrittura. D'ora in poi, ognuno avrà realmente la sua copia fisica, perchè a questo punto esse sono diverse. Non perdiamo comunque i vantaggi delle pagine condivise, perchè, come abbiamo appena detto, delle pagine non modificate rimane comunque una sola copia, condivisa. Questo metodo viene detto **copy-on-write**.

Vi ricordate di quando si parlava della fork()? Alle perplessità sulla effettiva miglior efficienza della fork() rispetto alla controparte Windows, possiamo rispondere adesso. In pratica, un processo può fare la fork() a costo zero, almeno in termini di frame, non vengono assegnati nuovi frame: è ovvio, i suoi figli sono identici a lui, le loro tabelle punteranno ai frame del padre. Poi, per forza di cose, i figli dovranno modificare le loro pagine, ma con copy-on-write verranno create nuove copie soltanto di quelle pagine, le altre restano condivise e si risparmia tanta memoria.

Ma nello specifico, come fa l'OS ad accorgersi del tentativo di scrittura, e capire che deve applicare copy-on-write? Le pagine condivise sono read-only di default, l'OS manipola le tabelle e mette il bit di scrittura a 0. Appena un processo vuole scrivere su una di esse, MMU controlla i permessi nel corrispettivo record della tabella delle pagine, ovviamente vedrà il bit di scrittura a 0, ma al contrario del caso tipico, non è un errore fatale. MMU non chiama un'eccezione, chiama un'altra procedura dell'OS, che crea la copia, effettua le modifiche richieste dal processo, cambia il puntatore nella tabella del processo che aveva richiesto la scrittura (per l'altro non c'è mica bisogno, gli rimane la copia "originale"), poi ripristina i bit di scrittura ad 1 di TUTTE E DUE le pagine, in tutte e due le tabelle. Da adesso ognuno può scrivere liberamente sulla sua copia, senza rendere conto a nessuno.

Ovviamente questo non è l'unico caso. Un altro esempio di copy-on-write può essere un padre con due figli: se il padre vuole scrivere su una pagina condivisa, viene creata un'altra copia, che condividono i due figli (quindi non crea due nuove copie immediatamente, le crea soltanto se uno dei figli scriverà a sua volta), che per forza di cose deve rimanere read-only.

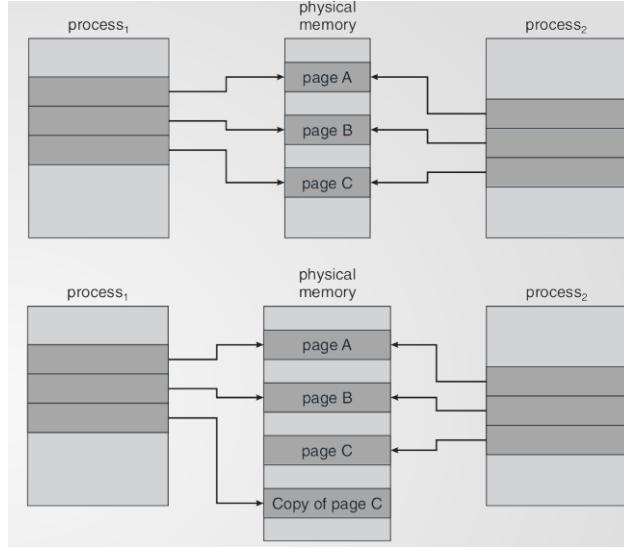


Figure 54: Copy-on-write

### 3.7.2 Zero-fill-on-demand

Quando un processo ha bisogno di nuove pagine, per ampliare il suo heap, effettua la chiamata **sbrk()**. OS dovrà destinare quindi un certo numero di frame all'heap del processo, per permettergli di alloggiare le sue nuove pagine. Come abbiamo scoperto quando parlavamo di ripescaggio delle pagine, un frame non viene subito svuotato, anche se è contrassegnato come "libero", rimane sporco, con al suo interno la precedente pagina; per cui, OS deve pulire tutti questi frame, per darli al processo che ha chiamato sbrk().

E' possibile assegnare nuovi frame al processo in maniera più efficiente di questa: **zero-fill-on-demand**. Viene previsto un frame speciale, detto **read-only static zero page**, contenente una pagina tutta piena di zeri. In pratica una pagina che non contiene alcuna informazione. Quando il processo chiama la sbrk(), OS fa puntare le pagine coinvolte a read-only static zero page. OS ha mentito al processo, gli fa credere che ha dei frame subito disponibili, ma non è vero. Cosa succede quando il processo deve effettivamente usare alcuni di questi frame? Semplicissimo: copy-on-write. OS pulisce il numero di frame necessario (cioè crea le "nuove" copie, come fa copy-on-write), e cambia i puntatori nella tabella delle pagine. In parole povere: OS non dà subito frame al processo, ma non glielo dice. Gli dà soltanto quelle necessarie, quando processo le deve usare, non prima.

### 3.7.3 Librerie condivise con linking

Molto spesso, soprattutto oggi, nel codice di un processo sono incluse delle call a funzioni facenti parte di librerie esterne. Le stesse funzioni probabilmente saranno usate anche da altri processi. Intanto parliamo di come includere questo codice all'interno del nostro:

- **Linking/compilazione statica:** la libreria è già precompilata. Compilo il mio sorgente, inserendo il codice binario delle funzioni, nei posti in cui vengono chiamate. Quindi tutto il codice delle funzioni è già nell'eseguibile. Vantaggio: ho tutto il codice subito a disposizione (sia funzioni che sorgente). Svantaggio: l'eseguibile occupa molto spazio.

- **Linking/compilazione dinamica:** nell'eseguibile c'è soltanto il sorgente. A runtime, appena si esegue la prima call, la libreria viene spostata TUTTA in RAM, e messa a disposizione del processo: è come se la libreria fosse parte dello spazio di indirizzamento del processo, della sua memoria in pratica. Quando verrà chiamata ancora la stessa funzione, o un'altra della stessa libreria, essa è già nello spazio di indirizzamento del processo. Ovviamente, se un altro processo necessita della stessa libreria, verrà inclusa nel suo spazio di indirizzamento, ma non ne viene creata una nuova copia. Tutti i processi punteranno alla stessa copia della libreria in RAM, ma ognuno di essi la vedrà come se fosse nel loro spazio di indirizzamento.

### 3.7.4 Librerie condivise con file mappati

C'è un altro metodo, migliore (eh ovvio): i **file mappati**. Cerchiamo di capire cosa sono. Intanto noi sappiamo che quando un processo deve leggere o scrivere un file su disco, effettua queste operazioni direttamente sul disco. Però in realtà potrebbe *mapparlo*, cioè portarlo in RAM, e includerlo direttamente nel suo heap. In realtà, non vengono allocati frame in RAM: queste pagine (il file) punteranno al disco. Il file verrà visto come se fosse in RAM, ma in realtà resta nel disco. Quindi da questo momento, il processo potrà leggere/scrivere il file autonomamente nel disco. Non è che il processo accede direttamente al disco: il bit di validità di queste pagine è 0, quando esso le vuole usare si verifica page fault, si prendono le pagine dal disco e portate in RAM, dove il processo può farci quello che vuole. Nel momento in cui quei frame devono essere liberati, le pagine (dirty a quel punto, se l'ho scritta) saranno sovrascritte nel disco, e il contenuto del file viene aggiornato. E' questo che si intende per *scrivere su un file in disco*. E' immediatamente comprensibile come questo metodo possa aiutarci con le librerie condivise: non portiamo TUTTA la libreria in RAM come prima, portiamo soltanto le pagine contenenti le funzioni che servono in quel momento. In pratica: codice fa call alla funzione, fa salto al frame di memoria in cui c'è la pagina interessata, ed è solo in quel momento che la pagina viene effettivamente trasferita in RAM, in quel frame.

I file mappati si possono sfruttare anche in fase di inizializzazione di un processo. Cioè il codice binario (l'eseguibile, che sta su disco) viene mappato nello spazio di indirizzamento del processo. All'inizio ovviamente ci saranno diversi page fault, necessari per portare fisicamente il codice in memoria, ma solo quello richiesto dal processo in quel momento.

Nello spazio di indirizzamento di un processo, tra la zona del codice e l'heap, sono presenti altre due zone di memoria:

- La zona superiore contiene i dati NON inizializzati (variabili fino a quel momento soltanto dichiarate) del processo, cioè una zona piena di zeri, detta **BSS**. Questa zona viene mappata in memoria utilizzando la Static Zero Page.
- La zona inferiore contiene i dati già inizializzati, viene mappata dal file in disco.

L'heap è dinamico, contiene tutte le strutture dati e gli oggetti allocati dal programmatore: viene gestito dalle librerie del linguaggio di programmazione usato. La memoria utilizzata per l'heap viene allocata contiguamente: si devono usare gli algoritmi già visti per gestire l'allocazione

contigua (next fit, best fit, ecc.), ma trattandosi di memoria virtuale la parte di heap non occupata viene direttamente deallocated. La frammentazione esterna in ambito virtuale è quindi un problema marginale.

## 3.8 Allocazione della memoria per il kernel

Anche il kernel ha bisogno di allocare della memoria, per contenere le strutture dati di supporto, ad esempio la tabella dei processi (deve allocare spazio per nuovi PCB in caso di creazione di nuovi processi). Il kernel ha a disposizione una zona della RAM, tutta per sè. Dobbiamo capire che in certi casi, dato che alcuni dispositivi interagiscono direttamente con la memoria fisica, non si può usare la paginazione; negli altri casi sì. Quindi si useranno tutti e due i metodi (indirizzi fisici e paginazione).

### 3.8.1 Slab allocator

Linux usa lo **slab allocator**. Uno **slab** è una collezione/gruppo di pagine, fisicamente contigue tra loro, in pratica un pezzo fisico di memoria contiguo; una **cache** è una collezione di slab. Ogni cache è "specializzata" in un determinato ambito: nel senso, una cache è specializzata nel contenere oggetti da 1 KB, un'altra contiene oggetti da 2 KB, ecc. Cioè ogni cache contiene slab di dimensioni diverse. Ad esempio, una cache specializzata nel contenere oggetti da 3 KB, ogni suo slab sarà capiente esattamente 3 KB, per cui la cache avrà capienza pari ad un multiplo di 3, e anche ad un multiplo della dimensione della pagina (uno slab contiene più pagine): in pratica la dimensione della cache sarà m.c.m. tra dimensione della pagina e dimensione dello slab. Questa tecnica di allocazione risolve il problema della frammentazione esterna (pagine dentro lo slab sono contigue, ma gli slab tra loro non lo sono), ma anche interna (lo slab è grande esattamente quanto l'oggetto che deve contenere, ecco perchè sono specializzati). Questo funziona ovviamente soltanto dentro il kernel, la dimensione degli oggetti è definita a priori.

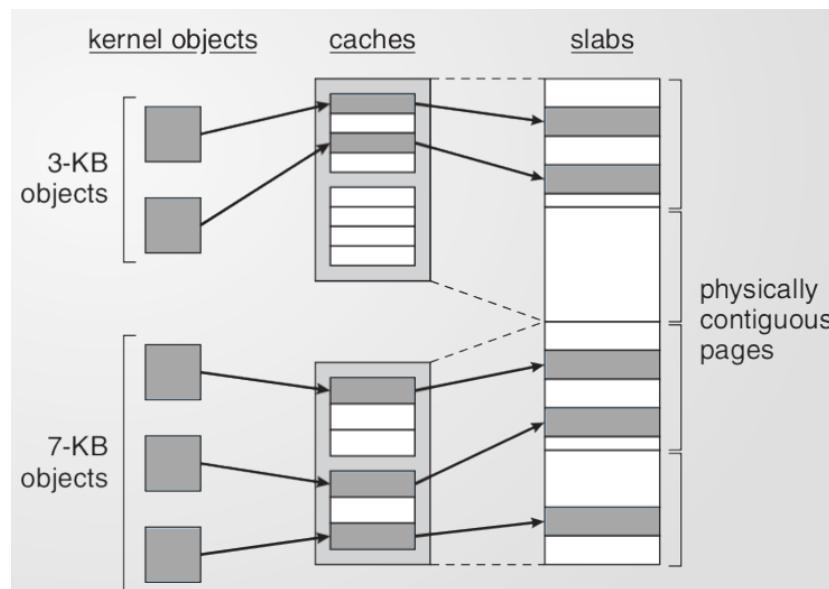


Figure 55: Slab allocation

## 4 File system e dischi

Si ha la necessità di gestire grandi moli di dati, in maniera persistente e accessibili a più processi. Per questo si usano delle memorie non volatili (dischi), ad esempio dischi elettromeccanici e a stato solido. L'astrazione che ci permette di organizzare questi dati persistenti è detta **file**. Il file è una lunga sequenza di byte, alla fine, ma che sono raggruppati in una sola entità. I file vengono implementati da una componente software detta **file system**, che si occuperà dei dettagli di gestione ed implementazione (con la parola file system si può intendere anche l'istanza di tutti i dati sul disco/partizione), ad esempio:

- Nome: il file gestisce le regole sull'identificativo del file. Ad esempio, il file system FAT (che vedremo) permetteva soltanto di usare numeri per nominare un file, o comunque una sequenza limitata di caratteri. Un altro esempio: il file system di Windows è case insensitive: il file "Casa" e il file "casa" saranno lo stesso file, anche se in realtà si accorge che sono file diversi perché la sequenza di caratteri è diversa.
- Tipo di file: ci sono file normali, e file diciamo "speciali". Il file **directory** è uno di essi. Il file directory contiene altri file al suo interno (o almeno i suoi metadati, come vedremo tra poco), anche altre directory. Sono quelle che noi comunemente chiamiamo "cartelle". Per cui, dalla directory principale si snoda un **albero** di directory con diverse diramazioni. Appunto la directory principale è la radice, perché contiene tutti i file dentro quel disco/partizione.

Un altro tipo speciale di file detto file **dispositivo**, che sono raggiungibili col percorso /dev/sda. Questi file rappresentano virtualmente dei dispositivi, ad esempio stampanti, mouse, ecc. La lettera A di "sda" non è fissa: rappresenta il nome del disco su cui siamo, in questo caso il disco A. Se il disco è diviso in più partizioni, ci saranno più directory, tipo "sda1", "sda2"....

- Tipo di accesso: l'accesso ai file avviene in maniera sequenziale o diretta? Ad oggi avviene in maniera diretta, con nuove memorie flash.
- Metadati: un file non è fatto soltanto dal suo contenuto, ma anche dai vari dati informativi su di esso, che permettono di identificarlo e conoscere le sue caratteristiche: nome, dimensione, estensione, maschera dei permessi. Queste informazioni sono molto importanti, anche più del contenuto stesso, e devono essere persistenti.
- Le politiche di accesso (operazioni supportate sui file): un file può essere scritto/letto da un solo utente, ma può essere anche condiviso tra più utenti. Oppure: più processi possono leggere/scrivere sullo stesso file? Sì, usando un meccanismo di lock, simili ai lock visti in RAM per garantire mutua esclusione. Il lock qui può essere di due tipi: **mandatory** e **advisory**. Il lock di tipo mandatory è appunto obbligatorio: quel processo non potrà leggere/scrivere il file in alcun modo, il lock è messo direttamente dall'OS. Il lock di tipo advisory può essere bypassato senza problemi: l'OS avvisa soltanto che ci potrebbero essere problemi in caso di accesso al file. Windows usa mandatory, UNIX advisory. I lock si possono anche distinguere tra **shared** ed **exclusive**. Questi lock

rappresentano praticamente il problema dei lettori e scrittori (2.11) il gruppo dei lettori avrà un lock di tipo shared, mentre il singolo scrittore avrà un lock di tipo exclusive.

Sono importanti anche le strutture dati di supporto al file system: ad esempio, deve sapere in quali directory stanno i file, o anche la loro posizione fisica. Alcune di queste strutture dati risiedono in RAM, e sono locali, cioè ogni processo ne ha una. Il singolo processo nel suo PCB contiene la lista dei file aperti da esso, e il file offset, cioè il puntatore alla locazione che sta leggendo/scrivendo. Altre devono essere globali, tipo i metadati.

## 4.1 Struttura di un file system

Intanto: lo spazio su disco viene diviso generalmente in diversi piccoli "blocchi", gruppi di locazioni contigue tra loro. C'è però un'astrazione intermedia: la **partizione**. Possiamo dividere il disco in "zone", indipendenti tra loro. Ad esempio, ogni partizione può contenere un OS diverso, tutti sullo stesso disco.

### 4.1.1 MBR

Storicamente una parte del disco conteneva un **MBR** (Master Boot Record). Una precisazione: si avevano dei limiti nel numero di partizioni per disco. MBR conteneva anche partition table, con un numero di voci predefinito. Un modo di bypassare questo limite erano le partizioni estese.

MBR conteneva un blocco dinamico caricato dal BIOS in memoria per avviare l'OS. Il codice contenuto in questo blocco (bootloader) doveva occuparsi di individuare la partizione che conteneva l'OS selezionato all'avvio, caricare in RAM il **boot block** (uno per ogni partizione), un codice che si occupava a sua volta di caricare il codice kernel dell'OS contenuto in quella partizione in RAM. Questo perchè il codice di MBR è generico: non può gestire procedure di avviamento del singolo OS. Il bootloader di Linux è GRUB: in quel caso è il codice di GRUB ad essere dentro MBR.

Adesso si usa **GPT** (GUID Partition Table), che è definito dallo standard EFI. Esiste una partizione EFI contenente una serie di eseguibili, che si occupano eventualmente di avviare altri OS.

### 4.1.2 Partizioni

La partizione in sè contiene strutture dati, oggetti e qualsiasi tipo di informazione. Contiene anche gli i-node, delle tabelle particolari (tranquilli, vedremo anche queste) che rappresentano il file dal punto di vista del file system (spoiler: ci dicono dov'è fisicamente memorizzato il file). Il contenuto effettivo del file è memorizzato in un'altra area della partizione. Poi c'è il file root directory: la directory che contiene tutti i file. Infine, ci sono delle strutture dati che gestiscono lo spazio libero.

## 4.2 Implementazione dei file

Abbiamo già detto che la partizione è divisa in tanti piccoli blocchi di dimensione fissa. Una domanda: dove memorizzo un file? Devo scegliere una quantità sufficiente di blocchi per far "entrare" tutto il file, ed allocarli

### 4.2.1 Allocazione contigua

Riservo una sequenza sufficiente grande da poter contenere il file. Non ho detto "sequenza" a caso: devono essere blocchi contigui. Si arriva facilmente alla conclusione che l'ultimo blocco potrebbe essere usato soltanto parzialmente (già un problema: frammentazione interna). Ovviamente, in tutti i casi di allocazioni contigue, quindi anche qui, è presente frammentazione esterna: alcuni buchi rimarranno vuoti "in mezzo". Dei file system particolari potrebbero anche vedere i blocchi come **cluster**: un cluster è considerata da essi l'unità minima allocabile, ed è composta da più blocchi, quindi un cluster ha dimensione pari ad un multiplo della dimensione del blocco.

Se voglio accedere al file F, come faccio a sapere dove risiede nella partizione? Dentro la directory, nella voce riguardante F, è contenuto il puntatore al primo blocco, e la dimensione del file in byte. Posso effettuare così l'accesso diretto, molto efficiente: avendo indirizzo del primo blocco, divido offset per la dimensione del blocco, sommo il risultato all'indirizzo del primo blocco, e trovo il blocco adatto (la prima word del blocco cioè). Questo è possibile perché sono allocati contiguamente. Il problema dell'allocazione contigua si presenta nel caso in cui il file cambia dimensioni, precisamente quando si espande. Se il prossimo blocco è occupato, che si fa? Si dovrebbero lasciare dei blocchi liberi ad hoc, così però se non sono usati si crea frammentazione esterna. Ma non posso fare swapping o cose simili come in RAM, siamo nel disco, dove si possono "parcheggiare"? Questo è il motivo per cui l'allocazione contigua viene usata solo nelle memorie di sola lettura, ad esempio CD e DVD. In questo caso, scrivo in memoria un "bloccone" di dimensione fissa (l'ISO), che da quel momento è immutabile. Posso implementare allocazione contigua senza problemi.

In generale, l'allocazione contigua ha prestazioni ottimali quando si parla di dischi elettromecanici. I blocchi saranno contigui anche fisicamente nel disco, la testina si posiziona una sola volta, sul primo blocco, e legge anche tutti gli altri in sequenza.

### 4.2.2 Allocazione con liste linkate

Ogni nodo della lista è un blocco fisico del file. Non importa dove siano i blocchi, sono collegati tra loro attraverso i puntatori della lista linkata, questo permette di risolvere la frammentazione esterna: ogni blocco può essere riempito. Il file system cercherà il più possibile di allocarli contiguamente, per sfruttarne i vantaggi, ma se non trova più blocchi contigui liberi, salta al prossimo blocco libero, anche se non contiguo.

Essendo una lista linkata, ogni blocco avrà un puntatore al successivo, tranne l'ultimo, che ha un puntatore fittizio. Qua si presenta il primo problema: 4 byte sono "rubati" dal puntatore, per cui lo spazio effettivo nel blocco non sarà una potenza del due; ad esempio, se blocchi da 1 KB, effettivamente dentro il blocco ci saranno soltanto 1020 byte. Se devo leggere 1 KB di

dati, mi aspetterei di leggere soltanto quel blocco, ma in realtà dovrò leggere anche i primi 4 del blocco successivo, non molto conveniente. Un altro problema: l'accesso deve essere per forza di cose sequenziale, devo attraversare tutta la lista, quindi tempo di accesso lineare  $O(n)$ .

#### 4.2.3 FAT

Per risolvere questi problemi, creo una tabella in cui raggruppo tutti i puntatori a NEXT, questa tabella è detta **FAT** (File Allocation Table). Tolti i puntatori dai blocchi, ho tutto lo spazio nuovamente a disposizione e ho risolto il primo problema. Il secondo problema lo risolvo mettendo la FAT in RAM.

Ogni voce della FAT è un puntatore a NEXT: la  $i$ -esima voce punta al blocco  $i+1$ -esimo. Per cercare un file, devo prima rivolgermi alla directory, per vedere dove sta il primo blocco e l'offset, dopo consulto la FAT e, a seconda di quanto è l'offset diviso, faccio un certo numero di salti nella tabella, poi troverò il blocco che cercavo. Ad esempio, vogliamo accedere al file F. Cerchiamo nella directory il suo blocco di inizio: è il blocco 4. Dividendo l'offset, otteniamo che stiamo cercando il quarto blocco a partire dal primo, cioè il blocco di indice 3. Andiamo nella FAT, e cerchiamo il blocco 4. Dopo, saltiamo al suo successore (magari è il blocco 2), saltiamo ancora (blocco 100) e ci fermiamo al terzo salto (blocco 1). Il blocco 1 è quello che cerchiamo.

Come abbiamo visto, dobbiamo comunque fare un numero  $O(n)$  di salti, ma qui siamo in RAM, molto più rapido fare i salti rispetto alla lista semplice precedente, situata in disco. In realtà la FAT viene tenuta anche in disco (ovvio, deve essere persistente, ce ne sono addirittura due copie per prevenire danneggiamenti) ma viene portata in RAM e sincronizzata periodicamente.

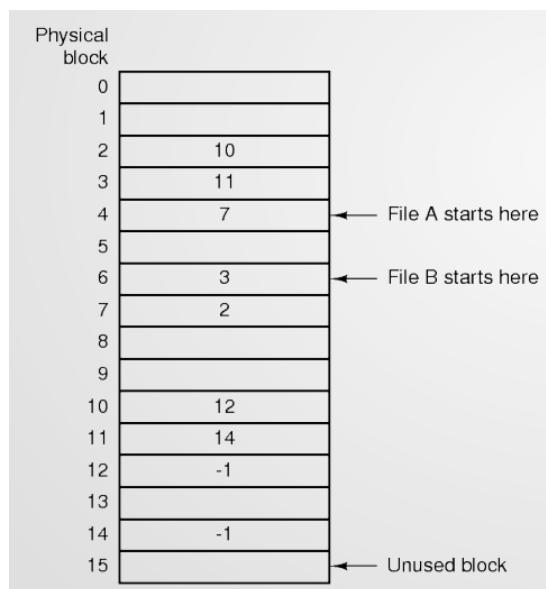


Figure 56: Un esempio di FAT

#### 4.2.4 I-node

**I-node** è una struttura dati di medio-piccole dimensioni dedicata ad un file (ogni file ne ha una). Il singolo i-node è identificato da un intero detto **i-number**. Nella directory adesso avremo soltanto nome del file e i-number. Tutti gli altri metadati sono dentro l'i-node. Per

trovare l'ubicazione di un file, lo cerco in directory attraverso il nome, trovo il record con l'i-number: posso adesso trovare l'i-node contenente tutti i metadati e la sequenza dei blocchi. Precisamente: la prima "parte" dell'i-node raggruppa tutti i metadati, la seconda parte è divisa in 13 record che descrivono la sequenza dei blocchi del file. Aspettate: soltanto 13? Sì, i primi 10 contengono effettivamente il numero di blocco, gli altri 3 sono dei record particolari. Il primo di essi (l'undicesimo) punta in realtà ad un'altra tabella, detta **blocco indiretto singolo**, che contiene altri 1024 puntatori diretti a blocchi (1024 perchè abbiamo assunto che un blocco contenga 4 KB, e ogni puntatore occupa 4 byte. In pratica, la "tabella" viene memorizzata nel blocco stesso). Il secondo (il dodicesimo) punta ad una tabella detta **blocco indiretto doppio**, i cui record contengono dei puntatori ad altre tabelle ancora, queste effettivamente continuano la sequenza dei blocchi. Infine il terzo (tredicesimo) punta al **blocco indiretto triplo**: punta ad altre tabelle che a loro volta puntano alle tabelle finali, che contengono la sequenza. Si capisce che possiamo espandere molto l'i-node potendo tracciare anche file molto grossi. Possiamo ricostruire l'ordine dei blocchi così:

- I primi 10 blocchi dell'i-node;
- Blocco indiretto singolo;
- Tutti i blocchi indiretti doppi;
- Tutti i blocchi indiretti tripli.

Si possono tracciare file di dimensioni fino ad:

$$(10 + 1024 + 1024^2 + 1024^3) \cdot \text{dimblocco}$$

Se abbiamo blocchi da 4 KB, possiamo tracciare file di dimensioni fino a  $4,40 \cdot 10^{12}$  byte, cioè circa 4 TB.

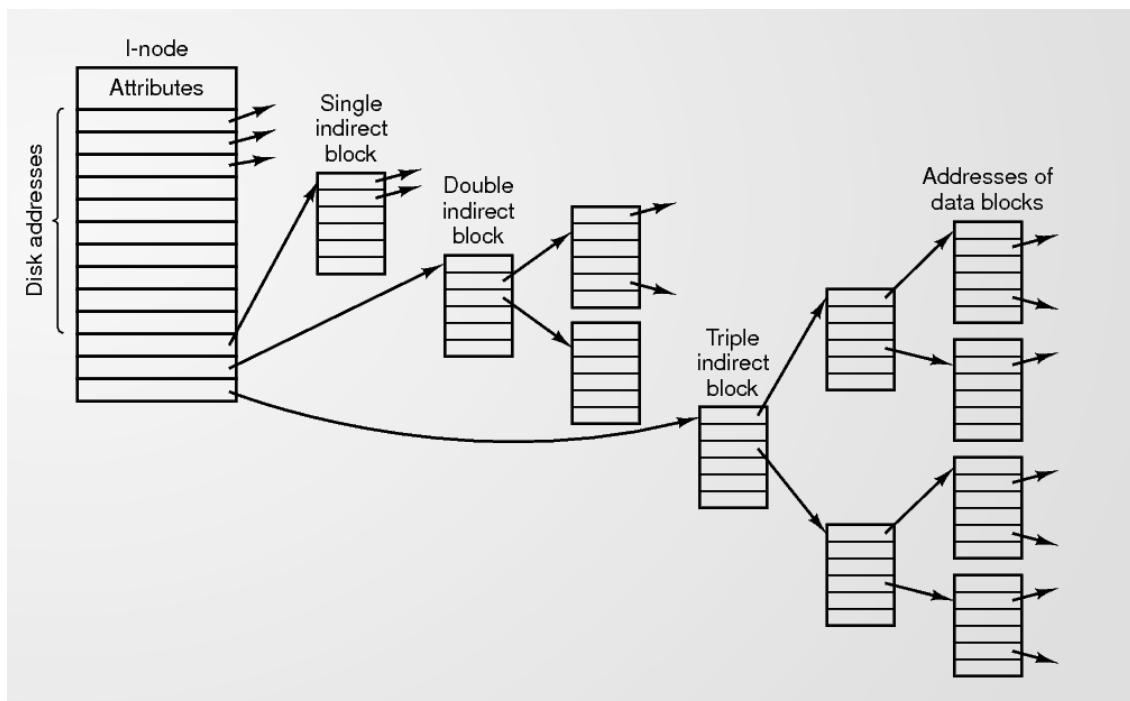


Figure 57: Un esempio di i-node

## 4.2.5 Extent

**Extent** è un gruppo di blocchi contigui. Usando extent, nella lista/FAT/i-node non si tiene più traccia del singolo blocco, ma del singolo extent. Cioè tengo traccia dei blocchi raggruppandoli. Extent ci permette, combinato appunto con FAT o i-node, di rimpicciolire molto queste strutture, a parità di dimensione del file.

## 4.3 Implementazione delle directory

E' già noto che le directory sono dei file speciali. Come ogni file, hanno un nome e dei metadati, e un contenuto. L'importante è implementare il contenuto (l'elenco dei file dentro essa, con nome, i-number ecc.). Possiamo vedere una directory come un vettore di record dinamico, in cui ogni record rappresenta un file (anche una sottodirectory), e dentro ogni record sono contenuti tutti i metadati del file, o i-number (se i-node, senza metadati), o magari il puntatore al primo blocco, se uso FAT. In generale, alcune informazioni, tipo quelle sopracitate, hanno dimensione fissa. L'unica informazione che non ha dimensione fissa è il **nome**. I nomi potrebbero anche essere molto lunghi, avere molti caratteri. come si gestiscono? Soluzione semplice: fissare un limite di caratteri che può avere un nome, quindi allocare direttamente quella quantità di memoria, ma se si sceglie un nome piccolo è uno spreco. Vogliamo qualcosa di dinamico.

Prima strategia: nella generica voce, si raggruppano i campi di dimensione fissa in una parte del record. La seconda parte è un vettore di caratteri, di dimensione variabile, che contiene il nome. Per tenere traccia della lunghezza del record, si prevede un campo iniziale, in cima, detto **File entry length**. Data la natura dinamica del vettore di caratteri, è necessario un carattere di terminazione per indicare la fine del nome. Lo spazio rimanente (libero, ma comunque di proprietà del record) serve sia come "riserva" in caso di rinomina, ma soprattutto per l'**allineamento**, cioè per fare in modo che la dimensione della entry sia un multiplo della dimensione della word generica, per migliorare efficienza. Se lo rinomino quindi ci sarà problema di frammentazione, ma in questo caso si tratta di una struttura piccola, porto il record in RAM, deframmento, e sincronizzo sul disco.

Seconda strategia: divido in gruppi. Che gruppi? La prima parte di vettore sarà composta interamente da tutti i campi di dimensione fissa di tutti i file. C'è un puntatore alla seconda parte di vettore (un heap) in cui sono memorizzati TUTTI i nomi. L'heap è dinamico. Risolviamo problema dell'allineamento, perchè viene portato tutto in RAM all'occorrenza. Servono soltanto caratteri di terminazione.

## izziare i metadati/attributi?

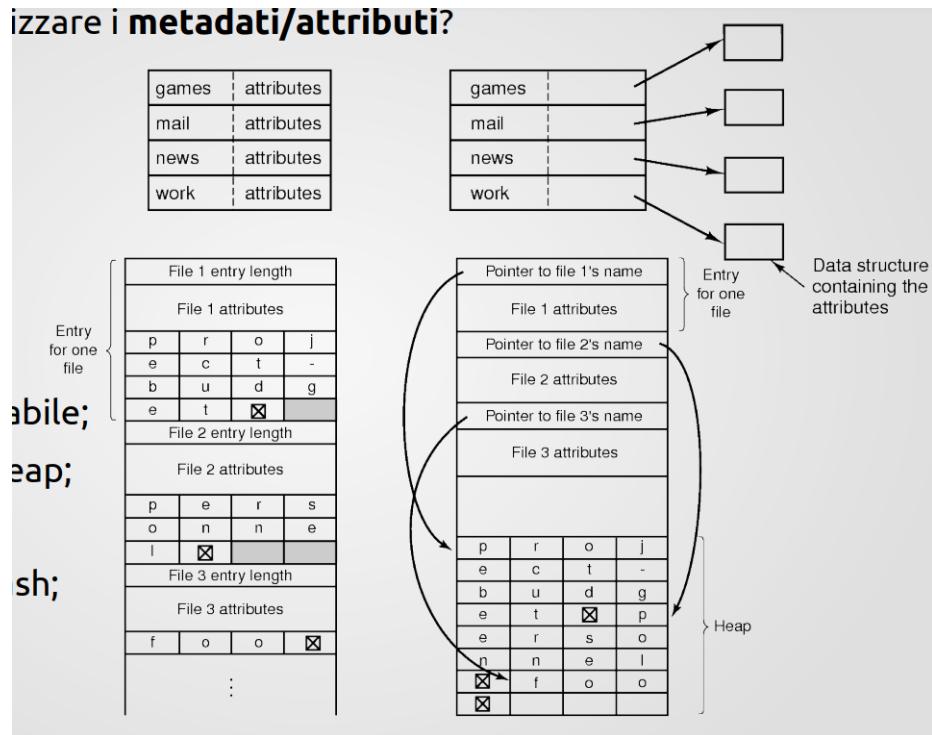


Figure 58: Le due strategie spiegate sopra

E' più facile deframmentare l'heap in caso di rinomina del file: i nomi memorizzati sono contigui, si deve spostare tutto e deframmentare, ma l'heap è piccolo. Se si tratta di una grande directory, con migliaia/milioni di file? C'è il problema della ricerca, che per forza di cose è lineare in ogni caso. Si può risolvere usando hash tables, oppure sfruttare RAM come "cache" del disco (vedremo anche questo). Perchè se si opera molto sui file di questa directory, le entry di quelli più usati saranno sicuramente in RAM, la ricerca è sempre lineare ma dura molto meno.

## 4.4 Condivisione di file su un file system

Un file system moderno deve prevedere la possibilità che un file venga letto/scritto da più utenti, deve adattarsi ad un OS multiutente. Scenario più semplice: due utenti vogliono condividere un singolo file. L'idea è quella di creare un file in una qualsiasi directory del secondo utente (a sua discrezione, chiamiamola genericamente DIR2) che contenga un riferimento al file originale, posseduto dal primo utente, nella DIR1. In pratica, se si usano i-node, deve puntare allo stesso i-node a cui punta il file originale (devono avere lo stesso i-number). In realtà, esistono due metodi diversi (per i-node).

### 4.4.1 Soft-link/symbolic link

**Soft-link** è un file speciale, contenuto dentro la DIR2, gestito dall'OS. Può avere qualunque nome, anche lo stesso nome del file originale, e ha un suo i-node. Il suo contenuto è particolare: è il percorso del file a cui punta. Aprendo il soft-link, OS segue il percorso, raggiunge il file originale in DIR1, e lo apre. Le modifiche dell'utente 2 si rifletteranno quindi sul file originale,

perchè effettivamente sta scrivendo su quello. Quando il file originale viene eliminato, o spostato, o rinominato, cambia il percorso: il soft-link non viene aggiornato dall'OS, perchè aggiornare tutti i soft-link ad ogni cambiamento sarebbe troppo dispendioso. A questo punto il soft-link si rompe, ritorna un errore, perchè quel percorso non porta più da nessuna parte. I permessi vengono gestiti coerentemente alla maschera dei permessi sul file originale.

N.B. un soft-link può puntare ad un altro soft-link, e così via.

#### 4.4.2 Hard-link/link fisico

**Hard-link** è sempre un file, contenuto in DIR2: ciò che cambia è la natura dello stesso. Non è più un file speciale come il soft-link, è un semplicissimo file che però punta allo stesso i-node a cui punta il file originale in DIR1. Quindi non contiene un percorso, ha praticamente lo stesso contenuto del file originale, avendo lo stesso i-number. Questo permette di gestire facilmente la cancellazione del file in DIR1: l'i-node non viene cancellato, viene soltanto cancellato il riferimento. Il secondo riferimento in DIR2 continuerà a puntare a quell'i-node senza alcun problema. Come fa OS a capire che non deve cancellare l'i-node? Prevede un contatore di riferimenti contenuto tra i metadati. Contiamo quanti riferimenti sono fatti al singolo i-node. Inizialmente, quando l'unico riferimento è il file originale in DIR1, il contatore è 1. Il contatore incrementa mano a mano vengono creati nuovi riferimenti, nella stessa directory o in altre. Se uno qualsiasi dei riferimenti (anche quello iniziale, quello che chiamiamo "file originale") viene cancellato, il contatore decrementa. Quando il contatore arriva a 0, non ci sono più riferimenti a quel file, e l'i-node viene deallocated. Alla luce di tutto questo, per l'OS è impossibile capire chi è il riferimento originale.

Un primo problema: se creo una reference ad una directory "antenata" si potrebbe creare un loop nell'albero di directory. Se l'algoritmo di attraversamento non riesce a capire che c'è un loop, potrebbe ciclare all'infinito nel tentativo di aprire il file. E abbiamo detto che OS non distingue i riferimenti. La soluzione è molto semplice: è impossibile creare riferimenti a directory antenate, almeno a livello utente. Con soft-link è invece possibile: posso distinguere i due riferimenti (originale e soft-link), sono due file di natura diversa.

Un altro difetto degli hard-link è l'impossibilità di creare i cosiddetti **cross-file system link**, cioè riferimenti a file di file system diversi (possibile invece con soft-link), su UNIX. Questo perchè su UNIX i file system sono "a cascata": nel senso, i vari file system sono contenuti dentro la root directory, devo raggiungere usando percorso assoluto (per cui con soft-link ci riesco). Dato che hard-link punta all'i-node, il problema è che l'i-number è univoco soltanto all'interno dello stesso file system. Quindi, se volessi puntare all'i-node 100 contenuto in un altro file system, potrei puntare all'i-node 100 contenuto nel "mio" file system, che è un altro file.

Terzo problema: anomalia dell'**accounting**. Se viene cancellato il riferimento originale, il file rimane comunque di proprietà dell'utente 1. Se altri utenti accedono a quel file, avendone il riferimento, le operazioni peseranno sulla quota di utilizzo del disco associata all'utente 1, anche se lui effettivamente non sta facendo nulla.

**Con FAT** Nella FAT non si possono creare riferimenti: alcune informazioni non sono contenute nella tabella, perché sono nella DIR1, l'utente 2 non possederà tutte le informazioni sul file, il riferimento punterebbe solo alla FAT. Per cui, si deve duplicare la lista con i riferimenti ai blocchi.

## 4.5 Gestione dei blocchi liberi

Bisogna tenere traccia anche di quali blocchi sono liberi, dobbiamo sapere dove poter memorizzare nuovi file. Come si faceva in memoria centrale (3.2), si possono usare delle **bitmap**, grande in proporzione alla grandezza dei blocchi e della partizione. La bitmap ha dimensione fissa, ed è memorizzata in disco. Sarebbe comodo averla in RAM, e infatti si pagina.

### 4.5.1 Liste

Oppure, come prima, uso delle liste concatenate, per loro natura dinamiche, e memorizzate su disco. Particolarità: ogni nodo si riferisce ad un blocco libero... perchè non memorizzare il nodo che si riferisce al blocco X, nello stesso blocco X? Così facendo, non andiamo proprio a toccare alcuna porzione di memoria, non si "ruba" spazio a nessuno. Quindi, quando quel blocco viene allocato, riempito, viene cancellato il nodo, tanto non c'è più bisogno. Più il disco si riempie, più la lista si rimpicciolisce, diversamente dalla bitmap che ha dimensione fissa. Volendo, si potrebbe includere più blocchi liberi contigui in un solo nodo, con numero del primo blocco e lunghezza della sequenza, ma si dovrebbe implementare il meccanismo di coalescenza.

## 4.6 Controlli di consistenza

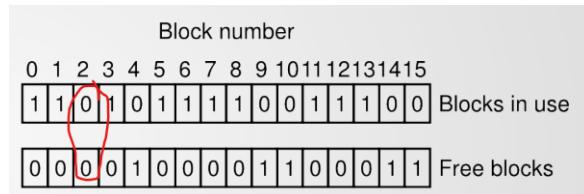
Se l'OS crasha durante un'operazione sul file system, esso può diventare inconsistente. Ci sarebbe un problema di coerenza: quali informazioni sono corrette e quali no? Ci poniamo questa domanda nel caso di interruzione forzata di operazioni di manipolazione su strutture dati del file system. Ad esempio: sto cancellando un file, e nel frattempo va via la luce. La cancellazione di un file è una macro-operazione, è composta da più operazioni atomiche. Potrebbe capitare magari che i blocchi vengono ripuliti e liberati, ma che la voce in directory di quel file punti ancora all'i-node, che a sua volta punta a quei blocchi, pensando che siano pieni. L'OS, se rileva dei problemi, esegue dei **controlli di consistenza**, ancora prima di poter usare il file system.

### 4.6.1 Controllo di consistenza sui blocchi

Le strutture dati deputate al mantenimento dello stato dei blocchi sono coerenti tra loro? Confronto struttura contenente i blocchi liberi con quella contenente i blocchi in uso (i-node/FAT). Si usano due vettori, inizialmente pieni di zeri. Durante la scansione dell'i-node, se vedo l'i-esimo blocco, incremento il contatore associato ad esso nel primo vettore. Se il contatore è maggiore di 1, c'è un problema. Ho allocato due volte lo stesso blocco? Strano. Poi guardo i blocchi liberi, e aggiorno il secondo vettore, alla stessa maniera. Stesso discorso vale qui: se

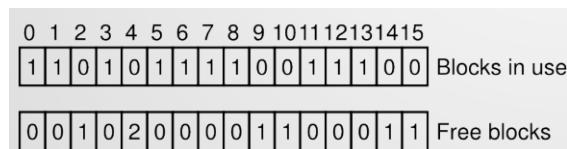
qualche contatore è maggiore di 1, ho liberato due volte quel blocco? Non è possibile. Quindi mi aspetto di trovare soltanto 0 o 1, e soprattutto che i due vettori siano complementari: se l'i-esimo blocco nel primo vettore è pari ad 1, nel secondo deve essere a 0, e viceversa. Adesso pensiamo a come correggere i problemi.

- **Primo caso:** non sono complementari.



Come vediamo, il blocco 2 è sia "non in uso", sia "non libero". Che si fa? Lo contrassegnamo come libero, mettendolo ad 1 nel vettore dei liberi.

- **Secondo caso:** contatore maggiore di 1.



Il blocco 4 è occupato due volte... è quindi coinvolto nella sequenza di due file diversi, non è possibile, un blocco non può avere due "proprietari" diversi. Si deve ricreare la sequenza dei due file, togliendo da essa quel blocco: inevitabilmente uno dei due file si danneggerà, perché effettivamente gli mancano i dati contenuti in quel blocco.

#### 4.6.2 Controllo di consistenza sui riferimenti agli i-node

Usiamo sempre due vettori, ma ora si scansionano tutte le directory, partendo dalla radice, e si contano quante volte ogni i-node è riferito, in questo caso il vettore può avere contatori maggiori di 1, perchè ci possono essere più riferimenti. Questi contatori sono dentro il primo vettore. Nel secondo vettore si contano i riferimenti andando però a controllare gli i-node stessi, si mettono nel secondo vettore. I vettori dovranno essere uguali.

#### 4.6.3 Journaling

Controlli di consistenza come quelli precedenti sono dispendiosissimi con file system di grosse dimensioni, come quelli moderni. Si adotta un'altra tecnica: il **journaling**: si costruisce un log in cui, per la prossima macro-operazione da eseguire, ci si appunta le sotto-operazioni atomiche da fare, ancora prima di farle. Solo dopo averle appuntate in questo log, si inizia ad eseguirle. Adesso, nel caso in cui si verifichi un crash durante lo svolgimento della macro-operazione, OS ovviamente non sa a quale sotto-operazione era arrivato. Si potrebbe implementare una sorta di to-do list, si mette una "spunta" alle sotto-operazioni già effettuate, ma magari il crash

si verifica tra la fine dell'operazione e l'inserimento della spunta. Allora si fa una cosa molto semplice: si rieseguono tutte le sotto-operazioni, dalla prima, facendo in modo che rieseguendole (alcune le avrò eseguite per forza) non si crei alcun danno. Nel journal non vengono appuntate le macro-operazioni sui dati, ma quelle sui **metadati**, enormemente più importanti rispetto all'effettivo contenuto del file, per cui sono sicuro che rieseguendole non vado a danneggiare nulla.

Qual è il grande vantaggio del journaling? In pratica, faccio sempre un controllo di consistenza, ma lo faccio soltanto sul file che stavo modificando in quel momento, non devo controllare tutto il file system. Controllo soltanto gli i-node coinvolti.

## 4.7 Ottimizzazione delle prestazioni del disco

### 4.7.1 Cache del disco

Incredibile, anche il disco ha una cache! Ma non è una memoria dedicata, è una porzione della RAM (in pratica tutti i frame liberi in un certo istante) che OS riserva per sé, apposta per questo compito. Questa cache è implementata via software. Funziona come la cache normale: controllo se il blocco che cerco è già memorizzato in questa porzione di RAM, se sì è hit, altrimenti miss e devo recuperarlo dal disco. Essendo in RAM, si deve ragionare in termini di frame; li vedremo come un "blocco di blocchi". Ma quale algoritmo dovrò usare per scartare blocchi nel caso di cache piena? L'ideale sarebbe usare LRU, ma questa è una cache software e abbiamo già detto che LRU è dispendioso da implementare in software.

Si ordinano i blocchi con una lista concatenata, in cui la testa è il blocco LRU, e la coda è il blocco MRU (Most Recently Used). Quindi ogni volta che un blocco viene usato, lo si mette in coda, e gli altri (quelli che erano successivi ad esso) "scalano" in avanti. Questa struttura deve essere aggiornata in RAM ad ogni operazione su disco, ed è anche un costo trascurabile, perché di molto inferiore al costo di un'operazione su disco.

Cosa succede in caso di miss (e cache piena)? Viene estratta la testa, preso il blocco richiesto dal disco e inserito in coda. Per vedere se c'è cache hit o cache miss, è necessario effettuare una ricerca (per forza lineare) dentro la lista: meglio usare delle tabelle hash, in una maniera abbastanza particolare. Ogni nodo avrà i due puntatori PREV e NEXT riferiti alla lista, ma anche due puntatori riferiti alla entry nella tabella hash, usiamo una tabella hash con concatenamento: ad esempio, nella seconda entry della tabella hash abbiamo nodo 1 e nodo 7, il secondo NEXT del nodo 1 punterà al nodo 7, mentre l'altro NEXT punterà ovviamente al nodo 2.

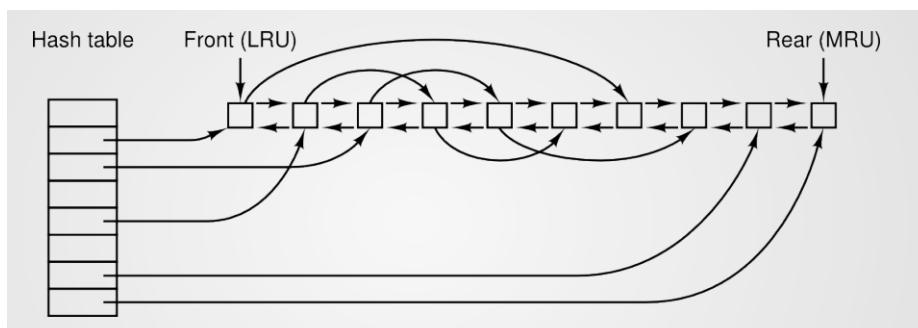


Figure 59: L'esempio descritto sopra.

Questa cache funziona ovviamente anche in scrittura: quando scrivo su un blocco, scrivo sulla copia in RAM. Per la sincronizzazione ho due scelte: scrittura **asincrona**, cioè si sincronizza soltanto periodicamente, o al momento in cui il blocco viene scartato. Il problema è che in caso di crash si possono perdere alcune modifiche. Se sincronizzo soltanto al momento dello scarto, per alcuni blocchi (quelli nella zona finale della lista, quelli usati più di recente) si ritarderà sempre di più la sincronizzazione, ed è un grosso problema, perchè questi blocchi solitamente contengono metadati. Quindi sui metadati si usa scrittura **sincrona**: il blocco viene sincronizzato subito dopo la scrittura.

La cache del disco si può ottimizzare in due modi:

- **Free-behind**: quando scrivo su un blocco, diventa MRU, ma qua adottiamo un ragionamento differente: se l'ho scritto adesso, è poco probabile che debba riscriverlo in un futuro prossimo, quindi lo scarto.
- **Read-ahead**: si usa nei dischi elettromeccanici. Quando si porta un blocco in cache, si portano in cache anche i blocchi adiacenti: se ho bisogno di un blocco, probabilmente avrò bisogno anche dei blocchi appena successivi, quasi sicuramente faranno parte dello stesso file.

#### 4.7.2 Scheduling del disco

Se consideriamo i dischi elettromeccanici, l'efficienza di lettura e scrittura è legata alla velocità del suo hardware. In che senso? Molto semplicemente dipende dal **seek-time**, cioè il tempo di posizionamento della testina, i costi di lettura e scrittura in sè sono trascurabili. Quanto ci mette la testina a raggiungere la traccia su cui deve leggere/scrivere?

Dobbiamo considerare anche che ci troviamo nell'ambito di un sistema multiprogrammato, con molteplici richieste di I/O. Dato che il tempo di posizionamento della testina è proporzionale alla distanza tra la testina e il cilindro da raggiungere, bisogna schedulare tutte le richieste nell'ordine che ci permette di dover percorrere la minor distanza possibile, misurata in numero di tracce percorse, in modo da avere il minor seek-time possibile. Useremo degli algoritmi di scheduling che come principio si basano su quelli visti nello scheduling dei processi.

Proveremo questi algoritmi su uno specifico esempio. Considereremo una lista di richieste di I/O, guardando soltanto il numero del cilindro coinvolto, alla fine è quello che ci interessa ai fini del seek-time.

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

Assumiamo che la testina sia posizionata inizialmente al cilindro 53, quindi dobbiamo includere nella somma anche la distanza tra 53 e la prima richiesta (che dipende dall'algoritmo). Il nostro piatto contiene 200 tracce, quindi ci saranno 200 numeri di cilindro, da 0 a 199.

**FCFS** Il classico algoritmo di scheduling First Come First Served, secondo la logica FIFO, li scheduliamo in ordine di arrivo. Quante tracce abbiamo percorso? 640.

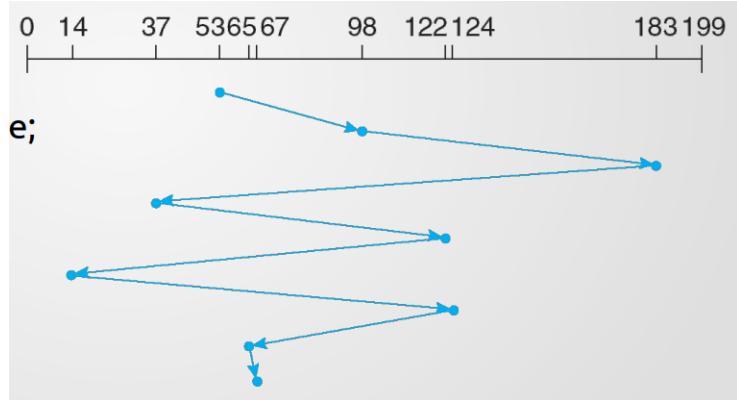


Figure 60: La distanza percorsa dalla testina usando FCFS

L'algoritmo di gran lunga più inefficiente: la testina si trova a passare continuamente dal "settore di sinistra" al "settore di destra", percorrendo anche grandi distanze solo per raggiungere la richiesta successiva. L'unico pregio di questo algoritmo è la sua equità: non favorisce nessuna richiesta particolare, le soddisfa in ordine di arrivo.

**SSTF** Shortest Seek Time First. Anche questo l'abbiamo visto in salsa un po' diversa: è lo scheduling per brevità, cambia soltanto il concetto di brevità. Qui verrà scelta la richiesta con la distanza più breve dalla posizione attuale della testina. Nel nostro esempio, sceglieremo per prima la richiesta al cilindro 65, perché la meno distante dalla testina, al cilindro 53. Dopo, la richiesta 67, perché la più breve dalla testina, al cilindro 65, e così via. L'ordine delle richieste soddisfatte:

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

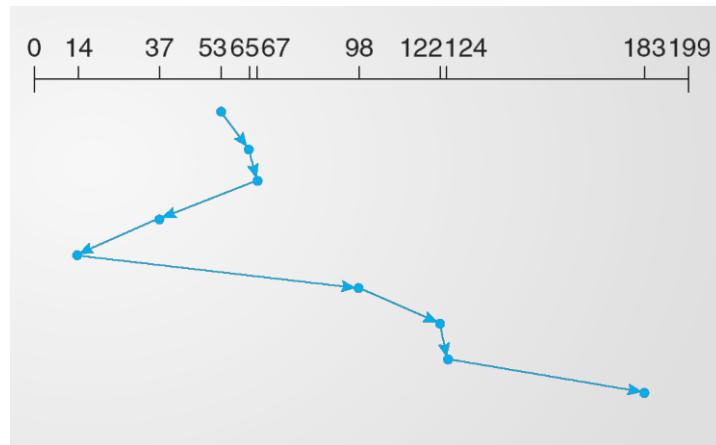


Figure 61: La distanza percorsa dalla testina usando SSTF

Abbiamo percorso in totale soltanto 236 tracce, molto meno delle 640 precedenti; dal punto di vista del seek-time è l'algoritmo ottimale. Sembra tutto buono, tranne una cosa: non è equo. Consideriamo uno scenario reale, in cui entrano continuamente nuove richieste: se continuano ad entrare richieste vicine alla posizione della testina, saranno soddisfatte solo quelle, quelle più "vecchie" ma più "lontane" rimarranno in una situazione di **starvation**; non hanno garanzia

di essere eseguite. Il tempo di esecuzione dei processi che fanno queste richieste aumenta a dismisura.

**Scheduling per scansione** L'algoritmo detto "dell'ascensore", ora vediamo il perchè del nome.

La testina adesso avrà un **verso**. Quindi essa andrà soltanto verso su (in ordine crescente di cilindro), o verso giù. Quando arriva all'ultimo cilindro, cambia verso. Questo garantisce un'attesa massima per ogni richiesta: dato un certo numero di richieste, esse saranno tutte esaudite in un ciclo di "su" e in uno di "giù". Nel nostro esempio, assumendo che la testina sia nel verso "su":

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

299 tracce percorse, più di SSTF, ma qui ci sono garanzie, non c'è più starvation. **N.B. Nelle implementazioni reali (e anche nel compito di OS) la testina cambia verso dopo che ha soddisfatto l'ultima richiesta in quel verso, non arriva alla fine/inizio prima di cambiare verso.**

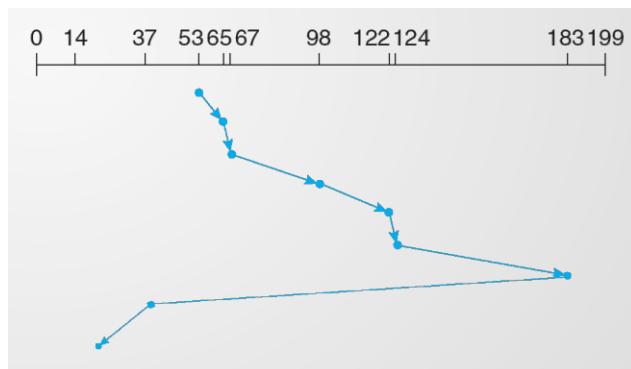


Figure 62: La distanza percorsa dalla testina usando l'ascensore

Come possiamo vedere, dopo la richiesta 183, la testina arriva a 199 e torna indietro, seguendo il verso opposto, ed esaudisce 37 e 14. In una situazione reale, avrebbe esaudito anche nuove eventuali richieste, ma soprattutto avrebbe esaudito le richieste 37 e 14 già in coda da prima. Viene da qui il nome: quando chiamiamo l'ascensore, esso si trova già in un verso, sta salendo o scendendo. Se noi siamo al secondo piano, e lo chiamiamo, magari lui si trova al terzo e sta salendo ancora. Prima o poi, quando arriva in cima all'edificio, dovrà scendere per forza, e arriverà al secondo piano (ovviamente molto semplificato, un vero ascensore si basa anche sull'ordine delle richieste). Ecco perchè non si verifica la starvation. Il caso peggiore di questo algoritmo è quello in cui entra la richiesta  $x$ , e la testina in quel momento si trova al cilindro  $x + 1$  (o  $x - 1$ , dipende dal verso). La testina percorrerà massimo un certo numero di tracce prima di ripassare dal cilindro  $x$ , perchè cambierà verso.

**Scansione circolare** La variante circolare dell'algoritmo dell'ascensore. Questa volta la testina mantiene sempre lo stesso verso. Quando arriva all'ultimo/primo cilindro, riparte dall'inizio/fine. Questo migliora il tempo d'attesa medio per ogni richiesta, almeno in caso di alto carico. Considerando che in media le richieste più vecchie sono quelle più lontane da

dove si trova la testina, ripartendo dall'altro lato si ha la possibilità di soddisfarla più velocemente, invece che aspettare che la testina rifaccia lo stesso percorso all'inverso. Ragionamento: cerco di soddisfare anche le vecchie richieste, ignorando quelle più nuove che mi farebbero perdere tempo. In quello precedente, quando la testina ritornava indietro, doveva soddisfare le richieste che incontrava sulla sua strada, qui le ignora direttamente, in modo da arrivare prima a quelle più vecchie e lontane. Le richieste vecchie aspettano in media di meno, quelle nuove aspetteranno in media un po' di più, e il tutto si bilancia.

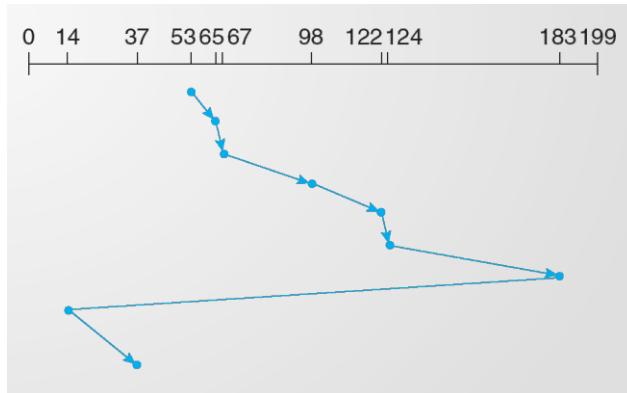


Figure 63: La distanza percorsa dalla testina usando scansione circolare

Quindi per alto carico conviene scegliere questo, per basso carico meglio la variante classica, o SSTF, dato che sono poche richieste, è basso il rischio di starvation.

#### 4.7.3 RAID

Un altro metodo per aumentare le prestazioni delle operazioni su disco è detto **RAID** (Redundant Array of Inexpensive Disks, o anche Redundant Array of Independent Disks). Con questo metodo si può ottenere parallelismo anche nelle operazioni I/O su disco. Si installano più dischi fisici, e si crea un'astrazione per cui essi vengono visti dall'OS come un unico storage, un unico disco virtuale. Questi dischi fisici devono essere pilotabili in modo indipendente tra loro. L'astrazione in questione può essere implementata in software a livello di OS, oppure in hardware, dentro il controller. E' preferibile, per ottenere prestazioni massime, che i dischi abbiano stessa capienza, e stesse prestazioni. Da qua fino alla fine, assumeremo che sia così. Intanto possiamo capire che lo spazio disponibile aumenta proporzionalmente al numero di dischi che si hanno: in teoria, con 4 dischi da 1 TB, avremo 4 TB di spazio disponibile, o almeno in teoria (poi vedremo che le cose stanno un po' diversamente). Ma come si fa a garantire prestazioni moltiplicate in lettura/scrittura? P.S. da questo momento assumeremo di avere un RAID di 4 dischi fisici.

**Striping** Si usa la tecnica dello **striping**: un certo file F viene diviso in parti, dette **stripe**, che a loro volta sono composte da un certo numero di blocchi. Successivamente questi stripe vengono distribuiti in maniera più omogenea possibile tra i vari dischi, seguendo una logica Round Robin. Quindi ad esempio, lo stripe 0 sul disco 1, lo stripe 1 sul disco 2, lo stripe 2 sul disco 3, lo stripe 3 sul disco 4, lo stripe 4 di nuovo sul disco 1, ecc. Come migliora le

prestazioni? Se dobbiamo leggere il file F, dobbiamo leggere tutti gli stripe. Se ad esempio il file F fosse composto da 4 stripe, per leggerlo basta semplicemente che ogni disco legga lo stripe che contiene, alla sua velocità. Ogni disco leggerà in contemporanea il suo stripe. Cosa significa? Che la velocità di lettura di tutto il file sarà pari alla velocità di lettura del singolo stripe da parte del singolo disco, lavorano in parallelo. Cioè prestazioni quadruplicate. Stessa cosa ovviamente per la scrittura. Ovviamente lo striping non viene fatto su ogni file (troppo complesso tracciare tutto), ma viene fatto a livello di file system. Il file system viene diviso in stripe, stesso principio. Il caso peggiore è quello in cui tutti gli stripe vengono messi su un solo disco: a quel punto sarà come avere un solo disco fisico, quindi nel peggior dei casi stesse prestazioni di prima, nel caso migliore prestazioni quadruplicate; ma usando Round Robin, è molto probabile che gli stripe siano distribuiti in modo omogeneo, quindi prestazioni di solito vicine al x4.

**RAID 0** Lo striping applicato semplicemente come spiegato sopra è detto **RAID 0**. C'è un grande problema: se uno solo dei dischi si rompe, si perde tutto. Si guasta tutto il volume logico (mi mancherà un "pezzo" di file system). Ma avendo n dischi, la probabilità che si rompa il volume logico aumenta di n: ne basta uno solo.

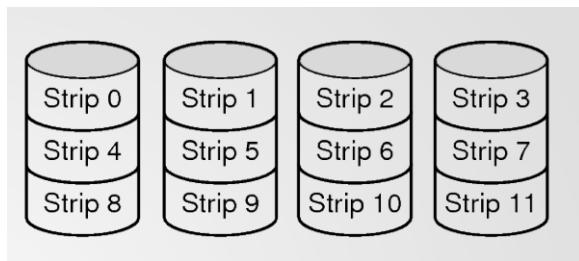


Figure 64: RAID 0

**RAID 1** Anche detto **mirroring**. Si sistema il problema della fragilità introducendo una **ridondanza**, in questo caso la ridondanza consiste nel duplicare tutti gli stripe. Quindi il contenuto di ogni disco viene copiato in un altro disco, o meglio: ogni modifica su un disco viene copiata nell'altro. Purtroppo per garantire lo stesso storage e le stesse prestazioni di prima avremo bisogno del doppio dei dischi. Quindi avremo 8 dischi, ma sempre 4 TB di spazio, e in casi particolari anche x8 in lettura (le scritture avranno prestazioni dimezzate rispetto al RAID 0, uno stripe sarà prima scritto nel disco originale, successivamente nel disco copia). Gli altri 4 dischi servono soltanto da backup. Cosa succede se si rompe un disco? Nulla: c'è l'altro disco copia, ma si entra in uno stato degradato, se si rompe anche quest'altro disco si perde tutto. Quando il disco rotto viene cambiato fisicamente, il contenuto viene ricostruito, copiando il contenuto del disco di backup. Nei server si usano addirittura dei dischi **spare** che si attivano al momento della rottura di qualche disco principale.

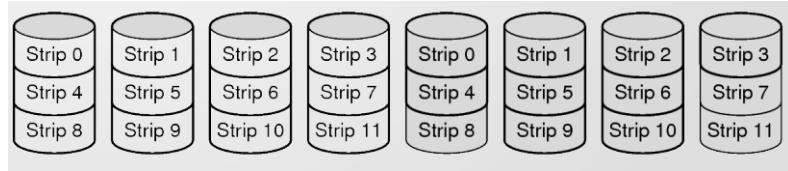


Figure 65: RAID 1

**RAID 2** Il doppio dei dischi è un prezzo troppo alto da pagare per avere le stesse prestazioni e storage di RAID 0: cerchiamo di ridurre questo "sovraprezzo", cambiando tipo di ridondanza usata. Useremo il **codice di Hamming**, quello che abbiamo visto in Reti, in modo da poter correggere un bit errato. Quindi qui lo striping lo facciamo a livello di bit. Avendo 4 dischi dati, avremo codeword da 4 bit di dati, e 3 di ridondanza. In totale una codeword da 7 bit, divisa in 7 dischi (già meno degli 8 di prima). Con il codice di Hamming non possiamo fare di meglio, come già abbiamo visto nel grafico mostrato in Reti. Cosa succede adesso se si rompe l'i-esimo disco? Si romperà soltanto l'i-esimo bit di TUTTE le codeword, che possiamo correggere tranquillamente con il codice di Hamming (il come lo spieghiamo sempre a Reti). Correggendoli, recuperiamo l'intero contenuto di quel disco. Unico problema: necessaria sincronia tra i dischi. Se qualcuno perde troppo tempo, leggo codeword incomplete, con bit mancanti. Questo ovviamente perchè abbiamo fatto striping proprio a livello di bit.

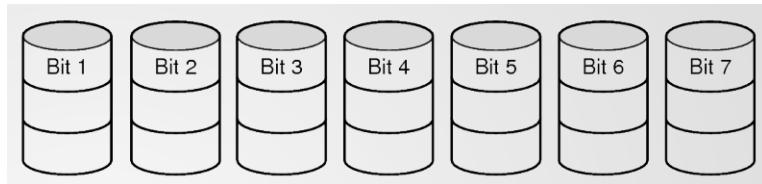


Figure 66: RAID 2

**RAID 3** Usiamo un'altra ridondanza, che ci permette di ridurre i dischi totali a 5: il **controllo di parità**. Il controllo di parità rileva soltanto gli errori, non li corregge (di suo). Ma ora vedremo che possiamo comunque correggere l'errore. Dando per scontato il funzionamento del controllo di parità (Reti), abbiamo sempre codeword da 4 bit di dati, ma la ridondanza è un solo bit, che memorizzeremo nel quinto disco. Cosa succede quando si rompe l'i-esimo disco? Come dicevamo, il controllo di parità rileva soltanto che c'è un errore, ma non ci dice qual è il bit da correggere, diversamente da Hamming. Una cosa: noi sappiamo quale disco è rotto, quindi sappiamo qual è il bit rotto. Possiamo comunque correggerlo. Se si rompe il disco con le parità, basta ricalcolare i bit. Essendo ancora striping a livello di bit, è ancora necessario il sincronismo.

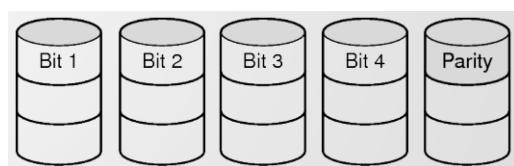


Figure 67: RAID 3

**RAID 4** Siamo ancora a 5 dischi come in RAID 3, ma torniamo allo striping a livello di stripe, molto meno pesante per l'hardware. L'ultimo disco contiene sempre le ridondanze, semplicemente stiamo cambiando per l'ennesima volta il tipo di ridondanza. La ridondanza sarà calcolata con uno XOR tra tutti gli stripe della stessa "riga" (nelle immagini), e il risultato messo nel quinto disco, alla "riga" corrispondente. Se ad esempio si rompe il disco 3, posso ricostruire? Sì, basta fare uno XOR tra i blocchi di parità che contengono gli stripe coinvolti, e tutti gli altri stripe dello storage. Non ho neanche bisogno del sincronismo, ci sono gli stripe, non più bit.

Che succede quando scrivo? Devo aggiornare la ridondanza associata a quello stripe. Cioè devo scrivere anche sul quinto disco. Per ricalcolare il nuovo blocco di parità, dovrei rifare lo XOR tra tutti gli stripe di quella "riga", quindi devo effettuare delle letture in tutti i dischi. Ma alla fine, tra vecchio blocco di parità, e il nuovo, cambia soltanto lo stripe che ho modificato. Quindi, faccio XOR tra blocco vecchio, nuovo stripe e stripe vecchio. Lo stripe vecchio è probabilmente ancora in cache, mi risparmio anche sta lettura in disco. Qua subentra il problema: ad ogni modifica di ogni stripe di ogni disco, sarà sempre coinvolto il disco 5. Il disco 5 lavorerà molto più degli altri, con il risultato che probabilmente sarà quello che si rompe per primo. Dobbiamo ribilanciare questo carico.

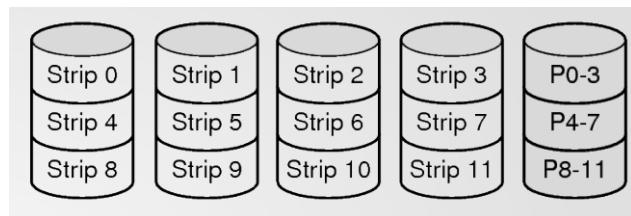


Figure 68: RAID 4

**RAID 5** Facciamo una bella cosa: perchè mettere le ridondanze tutte in un solo disco? Le si distribuisce in modo omogeneo tra tutti e 5 i dischi. Otteniamo anche un altro effetto positivo: adesso anche i dati sono sparsi in modo omogeneo tra i 5 dischi, anche il quinto disco memorizza dati. Quindi avremo prestazioni x5 in lettura/scrittura, e stesso spazio a disposizione di RAID 0/1/2/3/4.

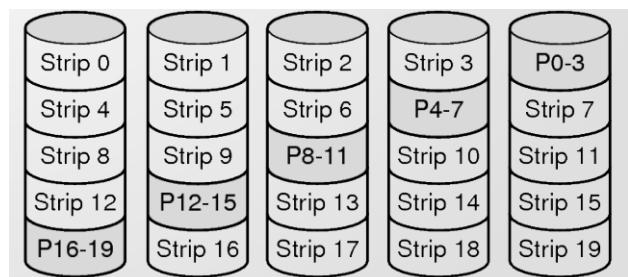


Figure 69: RAID 5

## 4.8 SSD (Solid State Drive)

Gli SSD sono delle memorie di massa non volatili che sono basati su memorie **NAND**, dette anche **memorie flash**. Come funzionano le memorie NAND? Intanto, bisogna sapere che le scritture sono molto più lente rispetto alle letture: questo perché la singola cella deve essere prima controllata, e poi scritta, dato che una cella non può essere sovrascritta se non viene prima cancellata. Il numero di scritture è limitato, dopo aver superato un certo limite quella cella smette di funzionare.

La memoria è divisa in **blocchi flash**, a loro volta composte da diverse unità minime di allocazione, dette **pagine** (non QUELLE pagine). Un blocco può essere vuoto, pieno, o parzialmente pieno.

Cosa succede quando devo scrivere su una pagina? Come abbiamo detto prima, prima di sovrascrivere, dobbiamo cancellare tutto. Una metodologia è il **F2FS** (Flash-Friendly File System), che contrassegna quella pagina come "spazzatura" e memorizza il nuovo contenuto in un'altra pagina dello stesso blocco, se disponibile. Qual è il problema? Le singole pagine non possono essere cancellate: dobbiamo cancellare tutto il blocco, per cancellare anche le pagine "spazzatura". Il **garbage collector** si occupa di riscrivere tutte le pagine valide in un nuovo blocco libero. Questo è possibile grazie all'istruzione TRIM, che permette di "ignorare" queste pagine spazzatura, e non riscriverle nel nuovo blocco; riducendo anche il numero di scritture e di conseguenza il degrado dell'SSD. Aumentano pure le prestazioni, dato che non si deve riscrivere ogni volta tutto.

In parole povere:

1. Delle pagine libere vengono scritte;
2. Se devo sovrascrivere queste pagine, le "nuove versioni" vengono scritte su nuove pagine, e le vecchie vengono contrassegnate come "non valide";
3. Quando ho bisogno di allocare un blocco, devo prima cancellarlo tutto, compresi ovviamente anche le pagine valide. Sposto SOLTANTO le pagine valide in un altro blocco libero, grazie a TRIM che mi permette di ignorare le pagine non valide, e infine cancello tutto il blocco (cancellando soltanto le pagine non più valide in pratica).

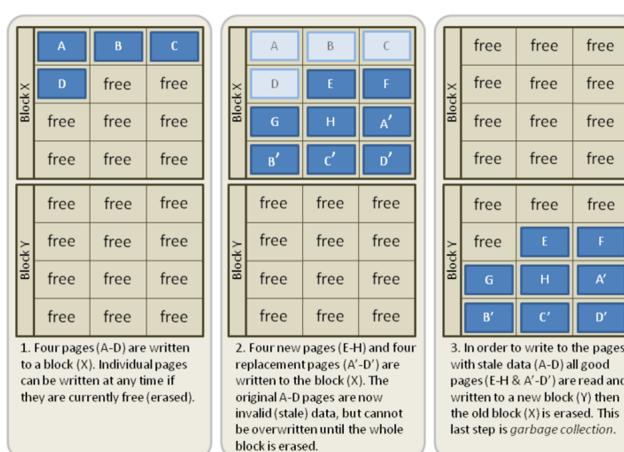


Figure 70: Processo di Garbage Collection

Il più grande vantaggio degli SSD è l'assenza della testina, che permette di ridurre a 0 il seek-time (non ha senso proprio parlare di seek-time).