SISTEMI OPERATIVI

SISTEMI OPERATIVI

SISTEMI OPERATIVI

CAPITOLO 1

Che cos'è un sistema operativo?

- Un OS è un "*creatore di astrazioni*", fornisce ai programmatori delle risorse astratte e hardware. Si può vedere come macchina estesa (fornisce un'interfaccia astratta più semplice per gestire l'hardware -> estende le funzionalità) e gestore delle risorse.
    - *Esempio:* astrazione **FILE**, insieme di dati salvati sul disco e modificabili, l'OS si occuperà di salvare i dati sul disco al posto del programmatore, il quale vedrà i dati sottoforma di oggetto.

Che cos'è un'astrazione?

- E' la chiave per gestire la complessità, suddividendo le attività complesse in sotto-attività più semplici. L'OS deve implementare una buona astrazione per gestire in modo semplice le varie funzionalità.

Quali sono le modalità di operazione hardware?

- **User mode:** *default*, esegue i processi utente (facenti parte di programmi e app esterne all'OS), in questa modalità l'OS può gestire un set limitato di istruzioni;
- **Kernel mode:** dove possono essere eseguite le istruzioni dell'OS, tra cui operazioni I/O e gli interrupt.

Come si passa da una modalità all'altra?

- Per cambiare modalità si usano:
    - Da user a kernel la syscall **TRAP** in caso di istruzioni vietate, in questo modo il controllo passa dal processo utente al kernel che eseguirà tali istruzioni. Alla fine, il kernel eseguirà un'istruzione di ritorno **RETURN** per far continuare l'esecuzione al processo;
    - Quando si invoca la TRAP, si passa un *parametro* corrispondende all'*id* della tabella propria del compilatore che permette di riconoscere l'istruzione kernel da eseguire; lato kernel, l'OS userà tale *parametro* per recuperare l'istruzione da una tabella propria dell'OS;
    - In caso di interrupt, al ritorno, il kernel può dare il controllo al processo utente corrente o ad un altro, scelto dallo scheduling.

In cosa consiste il multiplexing?

- Consiste nella **condivisione delle risorse** in 2 modalità, nel *tempo* e nello *spazio*.
  Nel caso del tempo si parla di **time sharing**, ogni tot di tempo viene eseguito un thread diverso, usando un context-switch.
  Nel caso dello spazio, si divide la memoria disponibile contenente la risorsa in piccole porzioni, ognuna riservata ad un thread. (Es: la memoria principale viene suddivisa tra più programmi di esecuzione).

Quali sono le fasi eseguite dalla CPU durante l'esecuzione di un'istruzione?

- Fetch dell'istruzione, decodifica ed esecuzione fisica dei calcoli necessari (**ALU**).

Quali sono i registri speciali della CPU?

- **Program Counter (PC):** puntatore che ha il compito di memorizzare l'indirizzo della prossima istruzione da eseguire;
- **Program Status Word (PSW)**: memorizza tutti i bit di stato (flag) della CPU per l'esecuzione di un'istruzione.

Esempio: flag che indica l'esito di un confronto, flag che indica la modalità (kernel o utente) in cui si sta lavorando. Alcuni flag sono modificabili solo in modalità kernel (*cambio modalità*).

- **Stack Pointer (SP):** registro speciale che contiene l'indirizzo della cima dello stack "attuale".

Cosa sono il multithreading e il multiprocessore?

- Il **multithreading** consiste nell'esecuzione di due flussi dello stesso processo sfruttando i "*tempi morti*" (intervalli in cui l'esecuzione di un processo si ferma in attesa di qualcosa) della CPU attraverso dei context-switch (multiplexing). Si parla di *pseudo-parallelismo* in quanto l'esecuzione è sempre sequenziale e gira un solo thread per volta.
  ***PROBLEMA*:** Ogni thread appare all'OS come un processo diverso, rischiando di schedulare 2 thread diversi appartenenti allo stesso processo nella stessa CPU.
- Il **multiprocessore** consiste in un *reale parallelismo* dovuto alla presenza di più CPU; se tali CPU risiedono nello stesso chip si parla di multicore, che si distinguono in 2 tipi:
  - ***Asimmetrici***, dove vi è un core che gestisce gli altri;
  - ***Simmetrici***, dove ogni core può eseguire ogni processo senza gerarchie.

Come si compone un dispositivo di I/O?

- Sono costituiti da 2 parti:
  - **Controller**, interfaccia che permette all'OS di pilotare il dispositivo;
  - **Dispositivo** in sé.
- Per conoscere i comandi da poter eseguire e le modalità di esecuzione, l'OS si avvale di un interprete, il **driver**.
- Il controller possiede registri, le *porte di I/O*, su cui la CPU opera con operazioni IN/OUT.
  - Alcuni OS eseguono le istruzioni in kernel mode -> i driver devono essere stabili;
  - Altri le eseguono in user mode, come quelli microkernel;
  - Quelli moderni usano un approccio ibrido -> l'OS sblocca al processo utente alcune porte I/O dove questo potrà eseguire istruzioni kernel;
  - Un altro approccio è quello della mappatura in memoria, si riserva una parte della memoria al dispositivo eseguendo istruzioni R/W standard dell'OS.

Quali sono i metodi di gestioni di I/O?

- Bisogna introdurre dei metodi di gestione in quanto le operazioni di I/O sono bloccanti.
  Ci sono 3 metodi:
  - **Busy Waiting:** il processo esegue una syscall e il kernel lo tradurrà in una chiamata di procedura, da qui l'OS gestirà il controller usando i driver e alla fine restituisce il controllo al chiamante. L'OS legge continuamente la porta di I/O (*polling*) in attesa di un esito; se l'operazione va a buon fine, si salvano i dati in un buffer e dopo in RAM.
    PROBLEMA: la CPU è sempre sotto sforzo per leggere la porta.
  - **Uso di interrupt:** dopo l'esecuzione dell'operazione di I/O, il driver notifica alla CPU un interrupt grazie ad un controller speciale -> interrupt controller. Ci sono 4 passi di esecuzione di questo metodo:
    - ❖ Il driver dice al controller cosa fare scrivendo sui suoi registri;
    - ❖ Il driver segnala l'interrupt al chip del controller tramite dei bus;
    - ❖ Il controller, se è pronto a ricevere interrupt, passa i dati alla CPU che li salverà in un buffer per poi trasferirli alla RAM;
    - ❖ Il controller mette il numero del device sul bus così da notificare alla CPU quali dispositivi hanno finito.

- **DMA:** si usa un chip speciale, DMA (*Direct Memory Access*), che può accedere alla RAM senza passare dalla CPU. Toglie il lavoro alla CPU di trasferimento dei dati eseguendo l'operazione e scatenando l'interrupt. La CPU dovrà quindi occuparsi solo di gestire l'interrupt impiegando meno tempo.

Che tipi di bus esistono?

- Sono di 2 tipi: *paralleli* e *seriali*. Quelli **paralleli** trasferiscono ogni bit della word su una linea diversa (parallelamente); i **seriali** trasferiscono i bit della word in sequenza e su ogni linea ci sarò un bit di una word diversa.
- Alcuni pc hanno la capacità di riconoscere e configurare automaticamente nuovi dispositivi hardware senza l'intervento dell'utente attraverso la tecnologia del **Plug&Play**, che permette al sistema di raccogliere info dei dispositivi I/O, assegna centralmente i livelli di interrupt e indica ad ogni scheda i numeri dei livelli.
- Esempi: **DMI**, seriale, che connette la CPU al controller delle porte di I/O; **PCIe** (*principale*) usato per periferiche esterne, è seriale e trasferisce i bit della word sulle linee come i pacchetti di rete.

Che tipi di OS esistono?

- Gli OS si distinguono per tipologia e per struttura.
- Le tipologie sono:
  - **Mainframe/server**: composti da molti dischi e CPU, gestiscono una grande mole di dati e sono orientati all'esecuzione di numerosi lavori alla volta.
    Esistono 3 tipi di servizi:
    - ***Batch:*** esegue lavori di routine automaticamente;
    - ***Elaborazione di transazioni:*** trattano numerose richieste;
    - ***Time-sharing:*** consente a più utenti remote di eseguire lavori sul server in parallelo.
  - **PC**: simili ai mainframe, gestiscono la multiprogrammazione, ma con carichi più bassi, e la multiutenza. Garantiscono protezione (user/kernel mode) e introducono la *GUI* per semplificare l'utilizzo ad utenti inesperti;
  - **Mobile**: simili agli OS per PC, senza la necessità di multiutenza e le GUI sono pensate per un diverso tipo di input (*touch*);
  - **Embedded**: sistema chiuso con determinati processi disponibili e in cui non è necessaria alcuna protezione tra i processi perché tutto il software è sul ROM.
  - **Real-time**: per sistemi industriali, le azioni devono essere *tempestive*. Per fare ciò si hanno pochi processi, l'OS è progettato con l'hardware e ogni processo può usare la CPU al massimo delle possibilità a tempo limitato.
    Ci sono 2 tipi di sistemi real-time, ***stretto*** (l'azione deve avvenire entro un dato margine di tempo) e ***leggero*** (ci possono essere ritardi ma meglio evitare).

- Le strutture sono:
  - **Monolitica**: primi OS, tutto il codice del kernel è contenuto in un unico programma binario eseguibile, a *blocco*; ogni componente può vedere/richiamare gli altri e ogni processo può accedere a tutto, scheduling e protezione si aggiungono dopo, tutto è eseguito in **kernel mode**. Per costruire il programma si compilano le procedure individuali, fuse poi insieme in un file unico, usando il **linker**.
    - Poca organizzazione
  - **A livelli**: suddivide il kernel in una gerarchia a livelli, tramite astrazioni. Ogni livello implementa una funzione dell'OS cosicché ogni livello fornisca un servizio a quelli superiori tramite un'interfaccia software (simile alla programmazione ad oggetto); la specifica di livello per **task** facilita implementazione e testing utili ad individuare bug, in quanto esso sarà contenuto

all'interno di un determinato livello.
Le procedure di un livello possono interagire con quelle degli altri;
- ❖ Difficoltà nella scelta dell'ordine dei livelli con rischio di sforzo maggiore per alcuni livelli, chiamate nidificate con conseguente overhead (Es: un livello richiama x procedure di un livello sottostante che a sua volta chiamerà altre y procedure, ...)

- **A livelli con cerchi concentrici**: separazione fisica dei livelli tramite un hardware ad hoc che permette di avere un diverso livello di protezione per livello (più si sale, meno processi saranno disponibili perché bloccati). Per richiamare un servizio di un livello inferiore si esegue una TRAP;
  - ❖ Overhead amplificato
- **Microkernel:** il kernel sarà minimizzato ai componenti *"indispensabili"* per il funzionamento dell'OS, gli altri gireranno in **user mode**. Gli altri componenti sono piccoli e isolati tra loro, così come i componenti esterni (driver), isolati in un processo utente. I processi comunicano tra loro e con il microkernel tramite ***pacchetti di messaggi*** recapitati da quest'ultimo.
  Il sistema si *"autoripara"* grazie alla presenza di un processo del microkernel, **reincarnation**, che controlla lo stato dei processi e li killa/ricrea in caso di problemi.
  Esempio: OS **MINIX 3**, dove la maggior parte dell'OS è suddiviso in numerosi processi utenti indipendenti, il sistema è strutturato in 3 fasce di processi in user mode quali **Drivers**, **Servers** e **Programmi Utente**, il *microkernel* gestisce interrupt, processi, scheduling e IPC.
  - ❖ Codice del microkernel scritto meglio (meno bug gravi), processi instabili non creano problemi ad altri, gestione da parte del microkernel più dinamica.
  - ❖ Overhead intrinseco per via del sistema a messaggi (lento), per comunicare servono più syscall di un sistema monolitico.
- **A moduli**: simile a microkernel per dimensioni e separazione dei componenti, detti *moduli*, tutti in **kernel mode** (nessuno sarà eseguito in user mode).
  In fase di compilazione del kernel si decide cosa includere nel kernel e cosa lasciare come moduli; alcuni moduli non sono caricati all'avvio, ma quando necessario (moduli di I/O);
  - ❖ Si usano le syscall al posto dei messaggi (più efficienza), ogni modulo può essere sostituito in caso di necessità.
  - ❖ Tutto in kernel mode -> più rischio di causare danni ai processi.
- **Macchine virtuali**: astrazione di una macchina, avremo un software e un hardware *virtuale* (basato su quello fisico).
  I processi virtuali girano sulla CPU fisica fino a quando non si necessita di effettuare una syscall; a questo punto il controllo passa all'**Hypervisor**, software che gestisce la VM. Esso permette di eseguire la TRAP in kernel mode virtuale (simula il kernel mode sull'OS virtuale) in quanto ogni processo della VM è in realtà in user mode;
  - ❖ Si possono avere più OS nella stessa macchina, sviluppo di app multipiattaforma, utilizzo di software obsoleto e livello di sicurezza (un servizio della VM attaccato dall'esterno non compromette la macchina fisica)
  - ❖ Spreco di risorse per il lavoro svolto dall'Hypervisor e inefficienza dovuta alla mancanza di un'interazione diretta con il dispositivo fisico, overhead.
- **Container:** sostituisce il modello a VM inserendo i servizi in un ambiente isolato, *container*, che si appoggiano all'OS principale (Es: Docker).

Quanti tipi di Hypervisor ci sono?

- Ci sono 2 tipi di Hypervisor:
  - **Tipo 1**: gira sull'hardware fisico e istanzia la VM, usato per virtualizzare un determinato servizio;
  - **Tipo 2**: processo utente che gira sull'OS principale, istanzia VM su cui girano OS virtuali ed effettua operazioni appoggiandosi all'OS principale. Per aumentare le prestazioni è stato aggiunto un **modulo kernel** sull'OS principale che esegue il grosso del lavoro.
    (Es: VirtualBox)

SISTEMI OPERATIVI

Spiega la differenza tra Simulazione e Virtualizzazione.

- La **Simulazione** è una tecnica che permette di eseguire una VM con architettura hardware diversa dalla macchina fisica, le istruzioni vengono interpretate (Es: emulatori di console)
- La **Virtualizzazione** è una tecnica per eseguire una VM con la stessa architettura della macchina fisica, senza interpretazioni.
- Vi è anche la **Paravirtualizzazione**, dove l'OS virtuale è conscio di esserlo, permettendogli di comunicare con l'OS principale (più efficienza).

SISTEMI OPERATIVI

CAPITOLO 2

Cos'è un processo?

- E' un'istanza di esecuzione di un programma, il quale è eseguibile più volte: ogni esecuzione corrisponde ad un processo diverso. Ciò permette di avere operazioni pseudo-concorrenti anche se vi è una sola CPU disponibile.
- Ogni processo ha uno spazio degli indirizzi (parte della memoria centrale) riservata a lui e con accesso privato. Tale spazio è diviso in parti: codice del programma, dati, heap e stack. Si parte da una locazione 0 ad una massima dipendente dallo spazio riservatogli.
- Ad ogni processo corrisponde lo stato dei registri della CPU, memorizzato ad ogni context-switch, e avrà associati tutti i file che ha aperto. Per salvare gli elementi associati al processo, l'OS sfrutta la tabella dei processi (dinamica, interna al kernel e memorizzata in RAM), dove ogni record, detto PCB (Process Control Block) è associato ad un solo processo e ne indica il padre. Gli indici della tabella identificano il PCB ed è detto ID del processo (PID).
  Quando un processo muore, il record PCB si svuota.

Definisci il modello dei processi.

- Il modello dei processi si basa sull'esecuzione sequenziale o in pseudoparallelismo. Nel caso dello pseudoparallelismo si tratta comunque di un'esecuzione sequenziale, ottimizzata dall'astrazione della CPU virtuale, dall'uso di context-switch e dal time-sharing che la fanno semprare parallela.
- Quando si parla di CPU virtuale, in realtà si avrà sempre una sola CPU fisica che però andrà da un processo all'altro (**multiprogrammazione**).
- Ad ogni context-switch si rende necessario fare in modo che le operazioni del processo scelto vengano svolte entro un determinato intervallo di tempo.
- Ogni CPU è dedicata ad un singolo processo, se si hanno più CPU si ha vero parallelismo.

Descrivi la gerarchia dei processi.

- Abbiamo un processo "genitore" creato dall'OS (INIT su Linux). In generale, tutti i processi sono imparentati in quanto ognuno (processo figlio) nasce dalla richiesta di un altro (processo padre).
  I processi si possono vedere come un albero n-ario che mostra le parentele.
- Al genitore viene dato un token chiamato **handle** usato per gestire il figlio.
- *Windows non ha la gerarchia dei processi, Unix sì.*

Come si crea e termina un processo?

- Ci sono 4 eventi che scatenano la creazione di un processo:
  - Inizializzazione dell'OS;
  - Syscall di creazione effettuata da un altro processo;
  - Richiesta dell'utente di creare un processo;
  - Inizio di un job in batch mode.
- Ci sono 2 tipi di processi: **attivi** (con interazione utente) e **in background**; tra questi ultimi, se i processi gestiscono attività sono chiamati **demoni** (daemon).
- Ci sono due modi per creare processi, uno usato su Linux e l'altro su Windows:
  - **Fork (Linux):** syscall chiamata da un processo che clona il processo con ID diverso, mantenendo tutte le altre info del PCB (codice, stack, ...); a seguito, il processo figlio chiama la syscall exec(), che cambia lo stato del processo con quello ricevuto come parametro.
    Dato che il codice è uguale, bisogna differenziare il contesto dal processo padre, basta un if.

Da quel momento, i 2 processi eseguiranno codice diverso senza legami e comunicheranno tra loro tramite l'IPC (inter-process-communication);
- **CreateProcess (Windows):** crea un nuovo processo da 0, specificando il codice da eseguire e i dati necessari. Win32 ha ulteriori funzioni per gestione e sincronizzazione di processi.

- La Fork è più dispendiosa ma ha più spazio di manovra (conviene).
- Processo padre e figlio avranno spazio degli indirizzi distinti.
- Un processo termina per varie cause:
  - **Uscita normale (volontaria):** syscall exit() (ExitProcess in Windows), le risorse vengono rilasciate il PCB svuotato;
  - **Uscita su errore (volontaria):** eccezione gestita da codice, richiama la syscall exit() che visualizza un messaggio di errore in forma di codice (diversi da 0);
  - **Errore critico (involontario):** errore fatale (divisione per 0, istruzione inesistente, accesso a locazioni non autorizzate, …) dove la CPU non può agire, potrebbe tornare un feedback;
  - **Terminato su richiesta di un altro processo (involontario):** un processo manda un segnale ad un altro per farlo terminare tramite la syscall **kill()**, quest'ultimo può accettare o rifiutare. E' permesso solo se i processi appartengono allo stesso utente.

Quali sono gli stati di un processo?

- L'info dello stato del processo è contenuta nel PCB, ci sono 5 stati:
  - **New:** creazione;
  - **Ready:** indica che il processo è pronto per essere eseguito dall CPU. I PCB dei processi sono memorizzati nella coda dei processi pronti (dinamica per via delle frequenti transizioni);
  - **Blocked:** non può usare la CPU (anche se libera) finché non avviene un qualche evento esterno. Si verifica nel caso di syscall lente, e di conseguenza viene tolto dalla coda dei processi pronti. Se il processo si blocca in attesa di un evento di I/O, sarà inserito in una coda dedicata. Quando verrà reinserito al completamento della syscall, lo scheduler non gli darà priorità.
    In alcuni casi il processo si può autobloccare (Es: chiamata fork, in attesa della creazione del figlio);
  - **Running:** lo scheduling pesca dalla coda dei processi pronti e tale processo cambia stato. Il dispatcher (componente che implementa le decisioni dello scheduler) salva lo stato del processo precedentemente in esecuzione e predispone tutto per il nuovo processo;
  - **Terminated:** terminazione. Quando un processo termina, i suoi figli generalmente vengono adottati da INIT, in alcune distribuzioni Linux vi è il demone **systemd** che si occupa di gestire i processi orfani.

- Le *transizioni* di un processo sono:
  1. Un processo **NEW** può passare a **READY** (admitted).
  2. Da **READY**, il processo può essere eseguito e passare a **RUNNING** (scheduler dispatch).
  3. Da **RUNNING** ci possono essere 3 possibili sviluppi:
     - tornando a **READY** (interrupt), gestito solo nel caso in cui il sistema è con prelazione, quindi un processo può usare la CPU per un limitato periodo di tempo;
     - passa a **BLOCKED** (I/O o attesa di un evento);
     - termina a **EXIT** (exit).
  4. Da **BLOCKED** può tornare nella coda dei processi pronti, **READY** (completamento dell'evento o dell'I/O).

SISTEMI OPERATIVI

Cosa sono i Thread?

- Il thread è un flusso di esecuzione di un processo, cioè il codice eseguito in quel momento dal processo e semplificano il suo lavoro. Il processo è, in pratica, un contenitore di thread.
  Sono indipendenti tra loro ma condividono le risorse del processo padre, eseguendo operazioni diverse. Anche tra i thread vi può essere priorità e può trovarsi in uno degli stati descritti per i processi.
- Sono più leggeri e più veloci dei processi e si possono creare facilmente ma non aumentano le prestazioni se sono thread CPU bound (consumano più risorse).
- Ci possono essere più flussi di esecuzione (multithread) dove un main thread assegna i task agli altri.
- Permettono la sovrapposizione di attività e sono in grado di comunicare tra loro grazie al fatto che operano sulle stesse risorse, l'unico rallentamento è dato dalla sincronizzazione tra loro.
- Sono molto utili in sistemi multicore, con il vero parallelismo.
- L'OS in realtà non vede i processi, ma dei thread.
- Pur lavorando sulle stesse risorse sono separati, ognuno ha un proprio stack, dei registri, il Program Counter e uno stato. Sono memorizzati nella tabella dei thread (gli OS moderni vedono solo questa tabella e non quella dei processi).
- Anche gli scheduler lavorano sui thread (danno priorità ai fratelli). Per passare da un thread ad un altro si applicano dei context-switch, più facile se sono fratelli.

Che operazioni si possono fare sui thread?

- Le operazione sui thread non coinvolgono il kernel e sono 4:
  - Thread_create: un thread ne crea un altro;
  - Thread_exit: il thread chiamante termina (se terminano tutti quelli del processo, questo termina);
  - Thread_join: un thread si blocca in attesa della terminazione di un altro;
  - Thread_yield: rilascio della CPU ad un altro thread, collaborando.

Come si crea un programma multithread/multicore?

- Ci sono alcuni aspetti da considerare:
  - **Separazione dei task:** identificare i singoli compiti e dividerli in thread;
  - **Bilanciamento:** le dimensioni dei task devono essere il più simile possibile;
  - Suddivisione e dipendenza dei dati: definire i dati su cui operano i thread considerando i problemi di sincronizzazione. Se la struttura dati di un thread si svuota, questo dovrà "addormentarsi" e sarà svegliato da qualcuno di esterno;
  - **Sequenzialità delle operazioni:** cercare di garantire che l'ordine di esecuzione sia rispettato;
  - **Complessità del debugging e test dei task**;
  - **Verifica di deadlock tra thread**.

Descrivi cos'è PThread.

- **POSIX Thread** è un package (libreria) di thread che permette la scrittura di dati portatili. Ogni thread PThread ha un id, dei registri e degli attributi.
- I metodi che contiene sono: *PThread_create*, *PThread_exit*, *PThread_join*, *PThread_yield*, *PThread_attr_init* (crea e inizializza la struttura attributi) e *PThread_attr_destroy* (rimuove la struttura).

Come si implementano i Thread?

- Ci sono 3 tecniche di implementazione:

- **Modello 1 a molti (livello utente):** implementati dentro il processo, si usano librerie che gestiranno i thread al posto dell'OS, dette runtime environment o runtime system, che si occuperanno anche dello scheduling. I core della CPU devono essere divisi manualmente.
  I thread devono assolutamente rilasciare la CPU quando necessario, altrimenti blocca gli altri; al rilascio salva il suo stato dentro la tabella dei thread salvata nella memoria del processo (nel runtime system).
  - ❖ Per fare un context-switch non è necessario richiamare TRAP -> minor overhead.
  - ❖ Se un thread fa una chiamata bloccante, l'OS blocca tutto il processo.

  Per mitigare il problema si usa la syscall select, richiamata dal thread, che controlla se la chiamata sarà bloccante e in caso cedere la CPU (yield) al fratello. Ma in caso di page-fault non è possibile, per la select, capire se il blocco sia dovuto dal thread.

- **Modello 1 a 1 (livello kernel):** ci sarà una tabella dei thread nel kernel e l'OS inserisce/rimuove i thread dalla tabella (il che ha un costo)
  - ❖ Il context-switch ha bisogno di una TRAP, lento. I segnali sono inviati tra processi e non tra thread, quindi non è possibile definire quale thread deve occuparsene.

  Si implementa il modello a worker, numero fisso di thread creati all'avvio tenuti in idle, svegliati se necessario e rimesso in idle alla terminazione del task (si risparmia su creazioni/distruzioni)

- **Modello molti a molti (ibrido):** si implementa su un sistema che supporta i thread kernel, ma ci sarà una differenza tra i thread kernel e utente. C'è una suddivisione tra spazio kernel (con i thread kernel) e spazio utente (con i thread utente).
  Un thread kernel è in grado di gestire più thread utente, i quali seguiranno la logica del multiplexing nello spazio, condividendo le risorse del thread kernel.
  Se il thread kernel si blocca, si bloccheranno anche i thread utente che gestisce.
  - ❖ E' possibile scegliere quando applicare una strategia e quando un'altra.
  - ❖ E' più complicato da gestire.

Come comunicano i processi?

- Ogni processo ha 3 canali di comunicazione standard: 1 per l'input e 2 per l'output, di cui uno dedicato agli errori. Il canale degli errori è separato per mantenere l'output principale pulito. L'input è associato alla tastiera e l'output al terminale. Si devono evitare gli interrupt.
- Una tecnica di comunicazione, detta a pipe, consiste nel relazionare l'output di un processo con l'input di un altro tramite un buffer su cui verrà memorizzato l'output. Se il buffer si svuota, il processo che legge in input si blocca, l'altro processo se ne accorge e termina anch'esso. [sfrutta l'OS]
- Per mettere in comunicazione i processi senza sfruttare l'OS si può pensare ad una finestra di memoria condivisa, dove uno scrive e l'altro legge (simile ai thread).
  - ❖ Accavallamento delle operazioni: se P1 legge X mentre P2 scrive dov'è salvato X, P1 potrebbe leggere dati errati -> manca la sincronizzazione. Bisogna verificare la corretta sequenzialità in caso di dipendenze (*problema simile al multithread*).

Cos'è e a che serve la mutua esclusione?

- Ipotizzando di avere una finestra di memoria condivisa tra 2 processi, in cui è memorizzata una variabile, e che i due processi effettuino operazioni di fetch-edit-store su questa variabile, se il meccanismo di prelazione dell'OS interrompe a metà uno dei due, la variabile in comune peccherà di un aggiornamento.
- *Esempio Race Condition*: P1 recupera la variabile, la incrementa di 1 e la prelazione lo interrompe prima del salvataggio passando il controllo a P2, il quale recupera la variabile con il valore vecchio, la incrementa di 1 e lo salva. Dopodiché, il controllo passa a P1 che salverà il valore incrementato, il quale però sarà identico a quello già salvato. Risultato atteso -> 2; Risultato ottenuto -> 1.

- La soluzione a questo problema è la **mutua esclusione**, dove i pezzi di codice dei processi che accedono alla memoria condivisa, detti **sezioni critiche**, pongono condizioni alla loro esecuzione, facendo in modo che i processi possano agire sui dati condivisi uno alla volta.
- Le condizioni per risolvere la **Race Condition** sono:
    - Mutua esclusione;
    - La soluzione non deve dipendere dalle prestazioni dell'hardware (come CPU speed e core);
    - Un processo fuori dalla sezione critiche non deve bloccare l'accesso ad un altro;
    - Nessun processo deve attendere all'infinito, ci sarà un limite di tempo garantito dal software.
- *Esempio Mutua Esclusione*: abbiamo 2 processi P1 e P2.
    - All'istante T1, P1 esegue senza problemi, perché P2 sta facendo altro.
    - All'istante T2, P2 vuole accedere alla sezione critica, ma dovrà aspettare che P1 finisca.
    - All'istante T3, P1 esce dalla sezione critica e P2 accede eseguendo le proprie operazioni

    I 3 punti si ripetono ad ogni richiesta di accesso alla sezione critica.

Come si implementa la mutua esclusione?

- Ci sono varie soluzioni che si sono andate a sviluppare nel corso del tempo, e ognuna andava a risolvere problematiche riscontrate con le soluzioni precedenti (dalla 2° vengono implementate delle funzioni, le cui chiamate delimitano le sezioni critiche):
    - **Disattivazione degli interrupt:** il problema della race condition è causata dalla prelazione, quindi si può pensare di rimuoverla quando un processo accede ad una sezione critica, in questo modo non sarà mai interrotto a metà.
        - ❖ Questo metodo funziona in CPU singlecore e con processi a livello kernel, per operazioni delicate.
        - ❖ In sistemi multicore, non è una soluzione perché se l'altro processo che vuole accedere alla sezione critica è eseguito in un altro core della CPU, non è possibile bloccare l'accesso perché i core lavorano in parallelo.
    - **Variabili di lock:** implementa le funzioni **enter_region()** e **leave_region()**. Viene creata una variabile di **lock** *condivisa* che può essere 0 se la sezione critica è libera, 1 altrimenti. L'enter_region() verifica che la variabile di lock sia 0, condizione per cui il thread può accedere. All'accesso, la variabile di lock diventa 1, all'uscita (leave_region()) viene riportata a 0. Questa soluzione sfrutta il meccanismo dello **spin lock**, busy waiting.
        - ❖ Si richiedeva di gestire l'accesso alle variabili condivise, ma è stata introdotta un'altra variabile condivisa (lock) con lo stesso problema.
    - **Alternanza stretta:** la variabile condivisa introdotta, detta **turn**, assume come valore l'ID del thread all'interno della sezione critica. Se il valore è diverso dall'ID del thread che richiede l'accesso, allora la sezione è già occupata -> busy waiting.
      Il thread che richiama la leave, assegnerà a turn l'ID del processo che ha chiamato la enter, permettendogli l'accesso.

```
int N=2
int turn

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

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

        - ❖ Nessuna race condition, in quanto il valore di turn è univoco. Tale soluzione è generalizzabile ad N processi.

- ❖ Rigidità delle turnazioni, violando il punto 3 -> caso in cui un processo P1 lascia la sezione critica dando l'accesso a P2, il quale però sta facendo altro; ad un certo punto P1 dovrà accedere di nuovo alla sezione critica, che è libera in quanto P2 non è ancora entrata, ma non può perché il turno è di P2 (attesa indefinita).

- **Soluzione di Peterson:** come il precedente, si basa sulla turnazione, ma si affida ad un array booleano chiamato interested, dove ogni indice è associato ad un processo; se per un indice il valore è true, allora quel processo vuole accedere alla sezione critica. Inizialmente tutti gli slot sono a false.
  La enter_region() ha come parametro l'ID del processo chiamante e si procura quello dell'altro processo, trasforma lo slot del processo a true, anche se il turno non è ancora suo ***turn =***

```
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
```

process (agisce solo sul suo slot).
La leave_region() cambia il valore dello slot a false.

Il while indica la presenza di busy waiting, ma più elaborato perché, dando per scontato che ***turn = process*** bisogna verificare che ***interested[other] = false***, perché se è true, significherà che la enter_region è stata già chiamata da un altro processo.

- ❖ La turnazione non è rigida. Tale soluzione è generalizzabile ad N processi.
  Se enter_region è chiamata da 2 processi in contemporanea, gli slot dell'array saranno aggiornati a true e, dato che turn è condiviso, uno lo aggiornerà per primo ma sarà sovrascritto dal secondo (**PROBLEMA**); questo problema è risolto dalla seconda condizione del while, che verifica l'attuale valore di turn: in questo modo, il secondo processo che ha richiamato la enter_region non avrà accesso alla sezione critica (***do nothing***), mentre il primo sì.
- ❖ Non funziona in sistemi multicore -> avendo due thread T1 e T2, che effettuano FETCH e STORE su 2 variabili x e y: T1 aggiorna x e memorizza y su R1, T2 il contrario su R2, i possibili valori dei registri sono:

R1	R2
1	1
0	1
1	0
0	0

e tutti sono risultati prevedibili, tranne l'ultimo, causato dal riordino delle istruzioni effettuato dai sistemi multicore per una maggiore efficienza. In questo caso, il meccanismo di riordino è stato tratto in inganno dal fatto che i thread operano su 2 variabili distinte, anche se condivise.
Questa è anche la situazione in cui si aggiornano 2 slot di interested nel codice, può capitare che entrambi vengano messi a false -> **deadlock**. Tale problema è parzialmente risolto dall'implementazione di istruzioni nei sistemi multicore che permettono la sincronizzazione tra più thread, dette **BARRIERE** (mettono una barriera ad un'istruzione, così da fermare tutti i processi prima di continuare).

- **Istruzione TSL (e XCHG):** risolve il problema della gestione sul multicore sfruttando l'hardware. Si introduce una nuova istruzione **TSL** (*Test and Set Lock*) che in Intel prende il nome di **XCHG**. La sintassi è: ***TSL Registro, LOCK*** con **LOCK** locazione di memoria.
  TSL effettua il fetch del valore di LOCK sul registro e poi scrive un valore predeterminato diverso da 0 nella locazione LOCK (doppia FETCH + STORE): è quindi l'unione di 2 istruzioni MOV. In questo modo garantisce *atomicità* (indivisibilità), facendo tali operazioni come singola istruzione, ed evita le interruzioni di mezzo.

```
enter_region:
TSL REGISTER,LOCK       | copia il lock nel registro e lo imposta a 1
CMP REGISTER,#0         | il lock era zero?
JNE enter_region        | se non era zero, il lock era stato impostato, per cui esegui il ciclo
RET       | torna al chiamante; si è entrati nella regione critica
leave_region:
MOVE LOCK,#0            | memorizza 0 in lock
RET             | torna al chiamante
```

Figura 2.25 Entrata e uscita da una regione critica usando l'istruzione TSL.

  Quando si invoca *enter_region()* si chiama TSL, che legge il vecchio valore di LOCK e sovrascritto con 1. Per sapere quale fosse il vecchio valore di LOCK, o se è stato aggiornato, basta comparare il valore del registro e 0 (**True =** scambiata dal processo, il quale è libero di entrare nella sezione critica; **False =** LOCK era già stata cambiata da un altro processo, busy waiting).
  La *leave_region()* rimette 0 in LOCK.
  La **XCHG** fa la stessa cosa ma scambia LOCK e registro.
  Tale soluzione è impiegata a livello utente e a livello kernel.
  - ❖ Risolve il problema del riordino Peterson, nei sistemi multicore conserva atomicità bloccando il bus di memoria agli altri processi (diverso dall'interrupt, ma così facendo si evita l'accesso alla sezione critica) fino alla fine di TSL.
  - ❖ Mantiene il problema del busy waiting.
- **Sleep e Wakeup:** resta il problema del busy waiting, in situazioni in cui dobbiamo gestire la priorità di accesso, si potrebbe verificare ciò che viene chiamato **problema dell'inversione di priorità**: avendo 2 processi **PH** (*alta priorità*) e **PL** (*bassa*), inizialmente la sezione sarà occupata da PH e quando questo invoca la *leave_region*, PL accede; se nel mentre PH si sblocca e richiede accesso alla sezione critica, avendo priorità alta, deve poterlo fare, ma ciò al momento non è possibile a causa del busy waiting.
  Per risolvere il problema, vengono implementate 2 funzioni a livello kernel che gestiscono la situazione dall'alto: **sleep** (syscall che blocca il chiamante, sospeso finchè non viene risvegliato dall'esterno) e **wakeup** (syscall che risveglia il processo con l'indice specificato nel parametro). *Possibilmente, sia sleep che wakeup, potrebbero avere entrambi un parametro che è l'indirizzo di memoria usato per far combaciare ogni sleep con ogni wakeup.*

- *Esempio di uso di sleep e wakeup-* ***Produttore-Consumatore*****:** abbiamo 2 processi **"Produttore"** e **"Consumatore"** che condividono una struttura dati *buffer*, bisogna sincronizzarli. Produttore produce item da inserire nel buffer, consumatore lo estrae.
  Produttore si deve bloccare se il buffer è pieno, Consumatore se è vuoto.
  Come si sbloccano? Devono farlo tra loro: Produttore sblocca Consumatore quando il buffer non è più vuoto, Consumatore sblocca Produttore quando non è più pieno.

SISTEMI OPERATIVI

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

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

- **Produttore:** ha un ciclo infinito, produce un item da inserire nel buffer, controllando che non sia pieno; se lo è, invoca la sleep e si addormenta, altrimenti lo inserisce e aggiorna il contatore. Deve anche risvegliare il consumatore se questo è addormentato, verificando di aver inserito un item nel buffer che precedentemente era vuoto.
- **Consumatore:** se il buffer è vuoto si addormenta, altrimenti estrae l'item e decrementa il contatore, risveglia il produttore se il buffer era precedentemente pieno, e consuma l'item.

➢ Si evita busy waiting, ognuno sveglia l'altro.

➢ Race condition derivata dalle condizioni di addormentamento e risveglio in caso di prelazione o esecuzione parallela (Es: si esegue consumatore con buffer vuoto, il quale prova ad addormentarsi ma, prima di invocare la *sleep()*, interviene la prelazione, quindi resta sveglio; nel mentre, produttore inserisce 1 item, risveglia il consumatore che però è già sveglio e riprenderà da dove si era fermato, invocando la sleep(), **riaddormentandosi**. Il count non tornerà mai più ad 1 e il consumatore resterà addormentato. Ad un certo punto anche il produttore si addormenterà e si verificherà un **deadlock**).

- Si può provare a mettere una "**pezza"** al problema aggiungendo un *bit di attesa*, 0 di default e aggiornato ad 1 se si perde una sveglia. Quando uno dei 2 processi prova ad addormentarsi, controlla il bit: se è 1, c'è una sveglia da consumare e il processo non chiamerà la sleep, ma piuttosto aggiorna il bit a 0.
  ➢ Questa soluzione non funziona nel caso in cui ci siano più produttori-consumatori, perché potrebbero andare perse più sveglie.

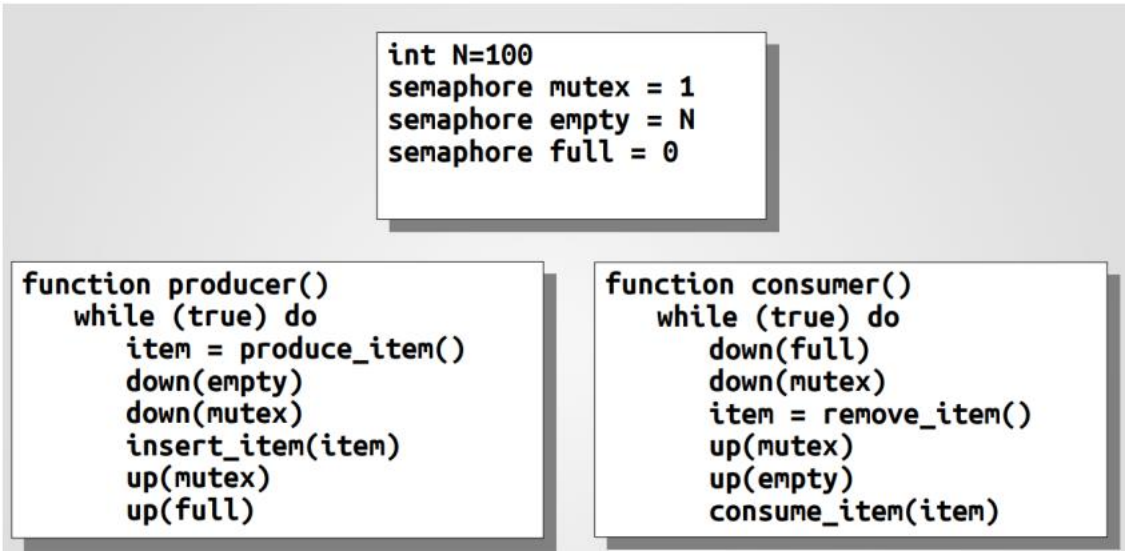
Cosa sono i semafori e che problema risolvono?

- Generalizzazione del bit di wakeup, ma al suo posto avremo un **contatore** che memorizzerà quante sveglie sono state perse. Il semaforo è implementato dall'OS ed è software.
- Il contatore non è mai negativo e per aggiornarlo si usano 2 operazioni **atomiche** perchè la variabile contatore è condivisa, quindi si tratteranno tali operazioni come sezioni critiche e, dato che siamo nel kernel, possiamo adattare la soluzione con TSL, associando ad ogni semaforo una **variabile di lock**, con delle *enter* e *leave region*. Le operazioni sono:
  - **down (wait)**, come la *sleep*, decrementa il contatore e deve impedire che diventi negativo;
  - **up (signal)**, come una *wakeup*, incrementa il contatore.
- Se la down è chiamata con contatore a 0, diventa bloccante; sarà sbloccata dalla up che incrementa il contatore ad 1, ma poi down (sbloccata) lo farà tornare a 0 -> non vengono contate sveglie inesistenti.
- L'associazione della variabile di lock al semaforo non causa il **problema dello spin lock** perché:
  - Wait e signal hanno codice piccolo, lo spin lock durerà poco;
  - Non vi è inversione di priorità perché non abbiamo concetto di priorità in questo contesto.

- **Utilizzo dei semafori:**
  - **Per mutua esclusione (mutex):** associamo il semaforo ad una struttura dati, inizializzato ad 1 (valore che indica quanti processi possono accedere in contemporanea alla struttura). La sezione critica è delimitata da una wait prima, che decrementa il semaforo a 0 (blocca l'accesso alla risorsa) e una signal dopo, che incrementa il valore (sblocca l'accesso);

SISTEMI OPERATIVI

- **Per sincronizzare processi:** problema del produttore-consumatore. Implementiamo 3 semafori condivisi:



1. **Mutex = 1:** per la mutua esclusione, la sezione critica è compresa tra down(mutex) e up(mutex);
2. **Empty = N (contatore):** indica il numero di slot liberi nel buffer;
3. **Full = 0 (contatore):** indica il numero di slot occupati.

❖ Il produttore produce l'item e invoca down(empty) per "prenotare" lo slot in cui inserirlo; invoca down(mutex) per richiedere l'accesso alla sezione critica: se mutex è 0 la sezione critica è già occupata dal consumatore, perciò il produttore si blocca, se entra effettua l'inserimento ed esce dalla sezione critica con up(mutex).
Infine, aggiorna il semaforo full, invocando up(full).

❖ Il consumatore è simile, chiama down(full) per prenotare l'estrazione, richiede l'accesso alla sezione critica delimitata da down(mutex) e up(mutex) e, se riesce ad accedere, rimuove un item.
Uscito dalla sezione, aggiorna il semaforo empty con up(empty) e consuma l'item.

- **Tale soluzione funziona anche nel caso in cui si abbiano più produttori-consumatori.**

- L'ordine delle operazioni è fondamentale: invertendo ad esempio per il produttore down(empty) e down(mutex) e ipotizzando il buffer pieno, il produttore entrerà nella sezione critica occupandola prima di chiamare down(empty), ma Empty = 0, ciò comporta il blocco del produttore.
Quando la palla passa al consumatore, proverà ad estrarre l'item ma non potrà accedere alla sezione critica (già occupata dal produttore) bloccandosi -> stallo.

Come si possono usare i semafori con thread utente?

- I thread a livello utente hanno problemi con chiamate bloccanti (se si blocca il processo, si bloccano tutti i thread figli). Volendo usare i semafori, dobbiamo affrontare il problema.
  - **Mutex con thread utente:** si può applicare la soluzione con TSL, senza spin lock. Operando su thread fratelli, questi possono cedersi la CPU con l'istruzione thread_yield.

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

```
mutex_unlock:
    MOVE MUTEX,#0
    RET
```

mutex_lock è simile ad enter_region e, se il confronto restituisce false, la CPU viene ceduta e si salta all'inizio della procedura. Non ci può essere attesa attiva del nostro thread, in quanto la CPU non è più in mano sua.
mutex_unlock aggiorna MUTEX a 0, come una leave_region.

- **Futex:** il problema dei mutex con thread utente è che non possono usare gli strumenti forniti dall'OS. Linux risolve il problema implementando mutex "ibridi", chiamati **futex** (*fast user space mutex*), che hanno 2 componenti:
    1. **in user space:** libreria che usa una variabile di lock gestita con TSL;
    2. **nel kernel**: coda di attesa che permette a più processi che non sono in esecuzione (sbloccabili dal kernel) di attendere un lock.

  La particolarità è che in caso di alta contesa (molti thread coinvolti) il futex lavora in spazio utente e poi effettua una syscall al kernel (come un mutex in spazio kernel). Il thread viene bloccato passivamente, senza spin lock. Senza contese, il kernel non viene chiamato.
  I thread bloccati sono inseriti in una coda apposita.

Cosa sono i monitor e perché sono nati?

- I **monitor** sono, concettualmente, semafori progettati in maniera più semplice e ad alto livello, definiti a livello utente. E' implementato dai compilatori sfruttando strumenti kernel quali i mutex.
  E' una raccolta di strutture dati e metodi utili a manipolare le prime evitando race condition.
- I processi possono chiamare i metodi quando vogliono, ma non possono accedere direttamente alle strutture dati da procedure esterne al monitor.
- In vecchi linguaggi, sono implementati grazie alla libreria *PThread*.
- Risolvono il **problema della sincronizzazione dei thread** implementando delle variabili di condizioni (***"etichette"***) su cui i thread chiamano *wait* e *signal*, aggiornandoli.
  Ogni variabile rappresenta un evento ha associata una coda di thread bloccati a causa di quell'evento; tale coda è implementata con i semafori e permette di gestire il risveglio dei thread nel caso in cui qualcuno invochi la signal.
  La variabile di condizione ha un valore specifico perché ogni monitor è usato da un singolo thread e non è più necessario contare le sveglie.
  - ➢ Così descritti, il problema consiste nella **mancata mutua esclusione assicurata** per via dell'utilizzo di 2 metodi diversi appartenenti allo stesso monitor da parte di 2 thread diversi, cioè: se T1 si blocca mentre esegue il metodo A, poi T2 esegue il metodo B e invoca la signal che, *casualmente*, risveglia T1, questo continuerebbe ad eseguire il metodo A -> entrambi i thread lavorano sullo stesso monitor, si interlacciano istruzioni di due metodi diversi.
- Per risolvere il problema, sono state ideate 3 semantiche:
  - **Monitor Hoare (signal&wait - teorico):** quando T2 invoca la signal, si blocca e attiva T1, così si accederà al monitor uno per volta. Resta il problema dell'interlacciamento di metodi diversi;
  - **Monitor Mesa (signal&continue):** implementazione in Java, quando T2 invoca la signal, prima completa il metodo B fino al return e poi risveglia T1;
  - **Signal&Return:** in concurrent Pascal, la signal sarà l'ultima istruzione, prima del return, ad essere eseguita nel metodo B -> T1 si risveglia quando il metodo è già ritornato.
  - ➢ I monitor non riescono a gestire la mutua esclusione su più CPU.
- Per risolvere il problema, si è adottato il sistema di scambio dei messaggi (***IPC – InterProcess Communication***). Si sfruttano code di messaggi che permettono la sincronizzazione tra sender e receiver, 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)
```

SISTEMI OPERATIVI

- Il *sender* produce un item e lo invia al *receiver* con la chiamata **send()**; finchè il messaggio non arriva, il **receive()** del *consumatore* è bloccante; quando la coda si satura, il send() è bloccante per il *produttore*.
- Si può generalizzare questa soluzione con N sender e M receiver, usando il metodo di indirizzamento a **mailbox** (*buffer di messaggi*). Non ci saranno più sender e receiver, ma *1+ mailbox* intermedie.
  - ➢ Scarsa efficienza, per mettere/prelevare (e cercare) messaggi dalla mailbox si usano syscall.
- Al posto dei buffer, si può fare in modo che il *sender* si blocchi alla send() finchè non riceve conferma di ricezione, e passare il messaggio direttamente al receiver senza buffer intermedi. Questo approccio è detto **rendezvous.**
  - ➢ Meno flessibile, mittente e destinatario devono essere eseguiti appaiati.

Spiega la soluzione di produttore-consumatore con i 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)
```

- Il monitor ha 2 variabili di condizione, *full* ed *empty*, una variabile *count* e 2 metodi, *insert* e *remove*.
- Come nei semafori, il produttore crea l'item e invoca il metodo **insert**, dove controlla se il buffer è pieno (wait sulla variabile full, si blocca) o se è stato appena inserito un item nel buffer vuoto (signal su empty, sblocca il consumatore).
- Il consumatore rimuove l'item con il metodo **remove**, dove controlla se il buffer è vuoto (wait su empty, si blocca) o se ha appena liberato uno slot dal buffer pieno (signal su full, sblocca il produttore) e poi consuma l'item.

Descrivi il problema dei 5 filosofi e spiega come gestirlo con mutex e monitor.

- **Panoramica situazione:** ci sono 5 filosofi seduti a tavola a pensare, con davanti il loro pranzo, ad un certo punto, uno di loro decide di mangiare; ogni filosofo ha una forchetta a destra e una a sinistra e, per mangiare, deve usarle entrambe. La forchetta alla sinistra dell'i-esimo è la destra dell'(i-1)-esimo, quella a destra è la sinistra dell'(i+1)-esimo. Più filosofi *non adiacenti* possono mangiare in contemporanea.
- **Come sincronizzarli?** Si possono usare variabili di lock. Ogni filosofo è un loop di pensare e mangiare:

```
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)
```

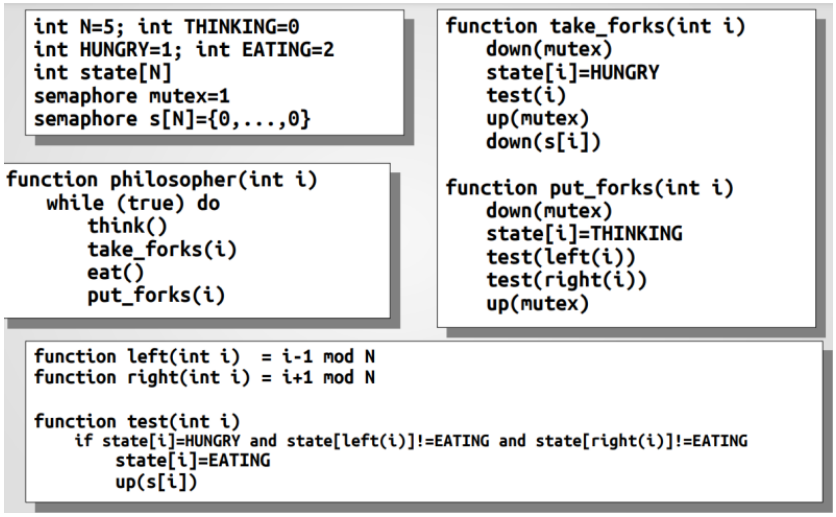
pensa, decide di mangiare, prende entrambe le forchette e mangia e poi le posa.
Ad ogni forchetta è associata una variabile di lock, se attivo la forchetta è occupata e la *take_fork* diventa bloccante. La *put_fork* disattiva il lock.

➢ Se tutti, in contemporanea, decidessero di prendere la forchetta a destra, nessuno potrebbe prendere quella a sinistra (**deadlock** e **starvation**).

- Tale problema si risolve facendo in modo che, se un filosofo riesce a prendere una sola forchetta, la rilascerà (o tutte e due o nessuna); una volta rilasciata, si potrebbe pensare di far attendere un tot di tempo random per filosofo e far riprovare, ma per evitare ulteriori conflitti si potrebbe usare dei **semafori mutex**.
- L'esempio dei 5 filosofi aiuta a modellare processi in competizione per l'accesso esclusivo ad un numero limitato di risorse (dispositivi I/O).

SISTEMI OPERATIVI

- **Soluzione con semaforo 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 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 philosopher(int i)
    while (true) do
        think()
        take_forks(i)
        eat()
        put_forks(i)
```

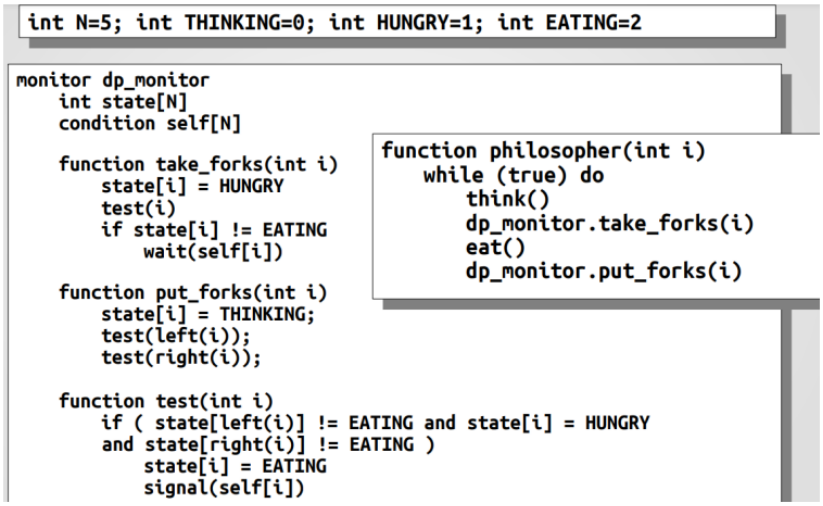
```
function left(int i)  = i-1 mod N
function right(int i) = i+1 mod N

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

- Abbiamo 3 stati: **THINKING** (sta pensando), **HUNGRY** (vuole mangiare ma non può ancora) e **EATING** (prende le forchette e mangia).
Per monitorare lo stato di un filosofo usiamo il vettore state, dove ogni indice corrisponde ad un filosofo, e serve a far bloccare l'i-esimo filosofo se necessario.
- Le funzioni sono *take_forks* e *put_forks* che fanno afferrare/rilasciare entrambe le forchette.
- Il mutex protegge la sezione critica in cui l'i-esimo filosofo cambia stato.

a) **Take_forks:** il mutex protegge il cambio stato da **THINKING** a **HUNGRY**, viene chiamato **test**, metodo che verifica che entrambe le forchette siano libere, cambio stato in **EATING**.
Vi sono poi la up e la down che operano sul semaforo ***s*** per l'i-esimo filosofo, dove è indicato se il filosofo è riuscito a prendere entrambe le forchette; up e down sono rispettivamente su ***test*** e su ***take_forks***, separati perché la down, se il filosofo non riesce a prendere entrambe le forchette, può diventare bloccante.

b) **Put_forks:** finisce di mangiare e cambia stato in **THINKING**, chiama la test sui filosofi accanto per farli svegliare se necessario (1° condizione in test -> state[i] = **HUNGRY**)

- **Soluzione 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)
```

- Si protegge il vettore degli stati con il monitor; *take_forks*, *put_forks* e *test* sono metodi del monitor. Piuttosto che avere un semaforo si ha una variabile di condizione per filosofo (**self**).
- Dato che le variabili di condizione non hanno memoria, bisogna controllare che lo stato dell'i-esimo filosofo sia **EATING** prima di prendere le forchette, altrimenti si blocca.

SISTEMI OPERATIVI

Descrivi il problema dei lettori e scrittori e spiega come gestirlo con mutex e monitor.

- **Problema:** bisogna modellare l'accesso in lettura e scrittura ad un database, differenziando le operazioni, ci possono essere più letture contemporanee, ma la scrittura deve essere esclusiva per non sporcare i dati.
- **Soluzione con MUTEX:** i lettori sono visti come un gruppo unico, può accedere il gruppo di lettori o un solo scrittore. La variabile **rc** indica il numero di lettori nel gruppo. Il semaforo mutex protegge il gruppo, l'altro il db.

```
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)
```

  Il **lettore**, prima di leggere, incrementa rc e, se è il primo ad entrare nel gruppo (rc = 1), locka il db. Infine decrementa rc e, se è l'ultimo (rc = 0), unlocka il db.
  Lo scrittore entra e blocca l'accesso a tutti, opera e poi libera il db.
  La down(db) è dentro la sezione critica per i lettori cosicchè si considerino questi come gruppo, se si blocca un lettore deve farlo tutto il gruppo, stessa cosa viceversa.
  - ➢ Si dà priorità ai lettori, sia perché lo scrittore dovrà attendere che finisca tutto il gruppo, sia perché, quando lo scrittore esce e c'è il gruppo di lettori in attesa, sbloccherà loro.
    Un modo per risolvere sarebbe un timeout che farà usare il db al gruppo a tempo limitato.

```
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)
```

Soluzione 2

```
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()
```

Soluzione 3

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

- **Soluzione 1 con monitor:** Il db è esterno al monitor per permettere l'accesso contemporaneo ai lettori. I metodi si dividono in 2 parti per lettore e scrittore:
  - **Lettore:**
    a) **Start_read:** controlla la variabile *busy_on_write* per controllare se vi siano già scrittori nel db, se non c'è incrementa ***rc*** ed entra nel gruppo. La *signal(read)* permette al lettore risvegliato dallo scrittore di risvegliare il gruppo;
    b) **End_read:** dopo la lettura decrementa rc e, se è l'ultimo, risveglia uno scrittore.
  - **Scrittore:**
    a) **Start_write:** se vi sono lettori (*rc > 0*) o un solo scrittore (*busy_on_write = true*) si blocca, altrimenti entra e mette *busy_on_write* a true per bloccare gli altri;
    b) **End_write:** mette *busy_on_write* a false e, se ci sono lettori in attesa li sblocca, altrimenti sblocca un altro scrittore.
  - ➢ Resta il problema del privilegio ai lettori, ma, a differenza dei mutex, è una scelta voluta.
- **Soluzione 2 con monitor:** viene aggiunta una condizione nello ***start_read*** per far controllare al lettore se vi siano altri scrittori in coda e, in caso, dare loro priorità, in modo da mitigare la discriminazione.
- **Soluzione 3 con monitor:** si invertono le condizioni nell'***end_write*** cosicché gli scrittori diano priorità ad altri scrittori piuttosto che al gruppo di lettori.

Tutte le soluzioni soffrono del problema della "starvation" -> lo scrittore attende per un tempo indefinito.

SISTEMI OPERATIVI

In cosa consiste e come funziona lo Scheduling?

- Lo scheduling consiste nella scelta del processo, tra quelli **READY**, che deve usare la CPU quando questa si libera. Si usa un *algoritmo di scheduling*.
- Dopo che l'algoritmo sceglie, interviene il **dispatcher**, che salva lo stato del ***processo precedente*** e dei suoi registri e ripristina lo stato del ***processo scelto*** cosicché possa usare la CPU. Ciò ha un costo che varia per contesto (thread fratelli o processi diversi).
- Quando lo scheduler interviene sui thread, non ha riguardo sul fatto che siano fratelli o appartenenti a processi diversi.
- Si possono distinguere 2 tipi di processi:
  - **CPU bounded:** useranno soprattutto la CPU (lunghi tempi di **CPU burst** – uso della CPU);
  - **I/O bounded:** faranno più richieste di I/O.
- Con il passare del tempo, i processi diventano sempre più I/O bounded, perché le CPU saranno più potenti (CPU burst inferiori). Questo dà allo scheduler un criterio di scelta, mischiando processi di tutti i tipi e dando priorità agli I/O bounded, che cedono la CPU dopo poco. Il problema è distinguerli.
- Lo scheduler deve essere veloce nella scelta, tale tempo è ***overhead aggiuntivo***, e deve essere **equo**, senza privilegiare alcuni processi, e deve bilanciare **l'assegnamento delle risorse**.
- Gli algoritmi di scheduling si dividono in **preemptive** (tempo limitato di esecuzione, **sistemi interattivi**) e **non preemptive** (tempo illimitato, finchè non si blocca o non rilascia la CPU, **sistemi batch**).
- Per lo scheduling dei thread, ci sono 2 situazioni: **Thread utente**, lo scheduler è implementato nel runtime environment e non userà prelazione in quanto tali thread sono collaborativi, e **Thread kernel**, dove si schedula sulla base della pesantezza del context-switch -> si scelgono thread imparentati.

- Nei **sistemi batch**, solitamente, bisogna considerare le metriche:
  - **Throughput:** numero di task completati per unità di tempo;
  - **Tempo di turnaround:** tempo tra l'inserimento del processo scelto e il suo completamento;
  - **Tempo di attesa:** tempo in cui il processo rimane pronto senza essere eseguito.

  Il 3° dipende dall'algoritmo, gli altri 2 dalla CPU.

❖ **Algoritmi di scheduling per sistemi batch:**

- **FCFS (Fist-Come First-Served):** funzionamento tipico di una coda FIFO, estrazione dalla testa
  Abbiamo 3 processi, schedulati in ordine P1, P2 e P3. Al tempo 0 sono tutti e 3 in coda. Si possono calcolare le metriche sapendo l'ordine e la durata:
  - **P1:** tempo di attesa 0, completamento 24 (durata);
  - **P2:** tempo di attesa 24 (dopo P1), completamento 27;
  - **P3:** tempo di attesa 27, completamento 30.

  In FCFS non c'è prelazione (**non-preemptive**)

Processo	Durata
P_1	24
P_2	3
P_3	3

t.m.a.: (0+24+27)/3 = 17
t.m.c.: (24+27+30)/3 = 27

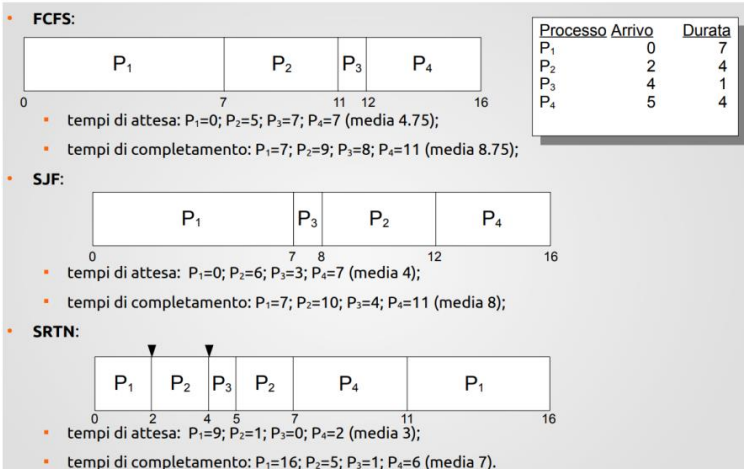
- **SJF (Shortest Job First):** la coda si riordina per durata crescente; i processi più corti sono scelti prima. Se i processi hanno tutti la stessa durata, si tratta di una FIFO (**non-preemptive**).
  Per conoscere la durata, si fa una stima per tipo di processo, dato che sono tutti della stessa tipologia; avremo ragionevole precisione di stima.
  Il tempo di attesa del processo corto migliora, il tempo di attesa del processo lungo peggiora.
  Tale ragionamento vale se i processi sono inseriti tutti all'istante 0, altrimenti SJF **non è più ottimale**.
  I processi che entrano nella coda dei processi pronti durante l'esecuzione di un altro, verranno ordinati per durata crescente.
  Ipotizziamo 5 processi: all'inizio, lo scheduler sceglie tra P1 e P2 e prende P1 (dura meno), all'istante 2 sceglie P2 (unico rimasto) che finisce all'istante 6. I processi P3, P4, P5 sono inseriti in coda all'istante 3 e avranno la stessa durata, quindi saranno estratti con logica FIFO.

Processo	Arrivo	Durata
P_1	0	2
P_2	0	4
P_3	3	1
P_4	3	1
P_5	3	1

	t.m.a.
SJF	(0+2+3+4+5)/5 = 2.8
altern.	(7+0+1+2+3)/5 = 2.6

P1 ha tempo di attesa 1, P2 ha 2, P3 ha 3 (entra all'istante 3 e viene schedulato all'istante 6), P4 ha attesa 4 e P5 ha attesa 5 -> tempo medio: 2.8
Ordinandoli inversamente, il tempo medio di attesa sarebbe 2.6, quindi **non è ottimale**.

- **SRTN (Shortest Remaining Time Next):** come SJF ma con prelazione (**preemptive**). Si esegue ogni volta che un nuovo processo è inserito in coda -> se entra uno con durata minore del processo corrente, ci sarà la prelazione e verrà schedulato quello nuovo.
  Nell'esempio di prima, all'ingresso di P3,P4 e P5 (tutti con durata 1), verranno schedulati prima questi, in ordine FIFO, di P2 (con durata 2), che sarà rimesso in coda.
  Il tempo di attesa medio scende a 1.6



- In **sistemi interattivi** conta la reattività, quindi bisogna velocizzare il tempo di risposta -> lo scheduler privilegia processi I/O bounded.

❖ **Algoritmi di scheduling per sistemi interattivi:** non si può prevedere la durata dei processi, gli I/O bounded si bloccano rapidamente, quelli CPU bounded possono monopolizzarla, creando un effetto *"convoglio"* (gli I/O bounded si accodano dietro i CPU bounded, bloccandoli perennemente).

- **Round-Robin:** come FCFS con prelazione (**preemptive**), si assegna un tempo di esecuzione limitato a processo, che, terminato, cede la CPU al successivo. Quello terminato sarà messo in fondo alla coda.
  Se il processo schedulato fa una chiamata bloccante, usa una parte del **quanto** di tempo a disposizione, per cui viene messo nella coda dei processi bloccati e, quando si sblocca, torna nei processi pronti. Alla schedulazione viene ridato il **quanto** di tempo *"pieno"*. In alcuni casi, allo sblocca, il processo torna in cima alla coda per fargli terminare il lavoro subito.
  Dato c, *tempo di context-switch*, è possibile calcolare la **percentuale di overhead**: **[c/(c+q)]%**
  - E' possibile distinguere i processi CPU bounded, prelazionati più spesso per via della durata, e quelli I/O bounded, prelazionati di meno. Avendo n processi e q come quanto di tempo, avremo la certezza che un qualsiasi processo verrà eseguito entro **(n-1)q** millisecondi.
  - Aumenta il numero di context-switch, overhead. Se il quanto è maggiore di ogni CPU burst, non vi sarà prelazione.
- **Con Priorità:** i processi hanno varie priorità, assegnata **staticamente** (all'inserimento nella coda) o **dinamicamente** (cambiato durante la permanenza nella coda). Si dovrà schedulare per primo il processo ad alta priorità.
  In caso di priorità statica, applica la prelazione al momento in cui viene inserito un processo in coda, confrontando la sua priorità con quella del processo in esecuzione; in caso di priorità dinamica, si cambia la priorità del processo in base alla durata, con la relazione **1/durata,** dando priorità ai processi più brevi. Così presentato, abbiamo un SRTN con priorità.
  - **Starvation,** nel caso in cui i processi ad alta priorità non finiscano mai.
  - Problema risolto con la tecnica dell'**aging**, si aumenta la priorità in base al tempo di permanenza in coda.

- **A Code Multiple:** si può applicare l'algoritmo SJF con priorità, con code multiple, una per priorità di scala. Ci sono 2 tipi di algoritmi:
    1. **Verticale:** si hanno più code, una per priorità, e la schedulazione interna è fatta con Round-Robin (si può anche applicare un algoritmo diverso per coda).
       C'è un *approccio ibrido*: si applica Round-Robin a tutte le code, le quali avranno a disposizione quanti di tempo diversi, direttamente proporzionale alla priorità.
       Alla priorità inferiore si applica il FCFS, in quanto non rendono conto a nessuno, in caso interviene la *prelazione*.
        - **Starvation**
        - Si risolve assegnando un quanto di tempo per tutte le code piuttosto che per processo. *Esempio:* ci sono 4 priorità (code) e un quanto di 5 secondi, il quale sarà diviso tra le code in modo che il 60% vada alla coda con priorità 4 (alta), 20% alla 3, 16% alla 2 e 4% alla 1.
    2. **Orizzontale.**
    - Si hanno priorità fisse.
    - Per risolvere, si usa la **retroazione**, i processi vengono spostati **dinamicamente** da una coda all'altra in base all'utilizzo del suo quanto (se ne usa una grande percentuale ha CPU burst grandi e viene scalato di coda finchè non ne userà una piccola percentuale)
- **Shortest Process Next:** SJF per sistemi interattivi, si considera come durata del processo, la durata del CPU burst successivo, schedulando il processo con CPU burst più breve, così da favorire gli I/O bounded.
  Considerando Tn l'n-esimo CPU burst e Sn l'ultima stima calcolata al CPU burst precedente, con queste info calcoliamo:

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

  con a compreso tra 0 e 1, che farà pesare di più la stima precedente (a = 0 -> si considera solo la stima precedente; a = 1 -> si considera il CPU burst precedente). Il valore migliore sarebbe a = 0.5.
  Il calcolo della stima è detto **aging**.
  Questo algoritmo non necessita di prelazione, la stima è abbastanza precisa.
- **Garantito:** risolve il problema della **starvation**. Lo scheduler fa delle ***"promesse"*** stimando il tempo di utilizzo della CPU da parte di un processo e, alla fine, verifica se tale stima è stata rispettata: se è rimasto indietro, ossia il processo ha usato la CPU per un tempo minore di quello stimato, avrà priorità più alta.
  Verrà schedulato ogni volta il processo rimasto più indietro.
- **A lotteria:** si divide il lasso di tempo in quanti, detti **ticket**, assegnati *randomicamente* ai processi. Poi ci saranno le **estrazioni**, permettendo l'uso della CPU. Se tutti i processi hanno lo stesso numero di ticket, si avrà equità; solitamente si danno tanti ticket quanta CPU verrà usata probabilmente dal processo. *I processi possono scambiarsi i ticket*.
  Questo metodo favorisce gli I/O bounded, che, usando meno CPU, accumuleranno più ticket.
  Ai processi più *importanti*, si assegnano più ticket per fargli avere più probabilità di ***"vincere"***.
- **Fair-share:** ripartisce la CPU equamente tra le **utenze** (sistema multiutente), indipendentemente da quanti processi hanno.
  Esempio: user 1 con 1 processo, user 2 con 9 processi -> un fair-share tra *processi* assegnerebbe a user 1 il **10%** della CPU e a user 2 il **90%**; un fair-share tra *utenze* garantisce un ***50-50***.

Si può applicare una logica ibrida tra **fair-share** e **lotteria**, dove i ticket si ripartono equamente tra utenze e poi saranno ripartite internamente tra i processi.

- I **sistemi real-time** hanno scadenze da rispettare, lo scheduler deve eliminare ritardi. Si dividono i **hard real-time**, con scadenze più rigide, e **soft real-time**, con più tollerabilità.

❖ **Algoritmi di scheduling real-time:** si dividono in **statici**, se la decisione di scheduling avviene prima dell'esecuzione, e **dinamici**, se la decisione avviene durante l'esecuzione.

SISTEMI OPERATIVI

- Scheduling su **sistemi multiprocessore**, ci sono 2 approcci:
  - **Multielaborazione asimmetrica:** vi sono un core **master**, che gestisce il kernel e gli altri core, e gli **slave**, che svolgono piccoli compiti.
    - ❖ Bassa scalabilità, con troppi core il master sarebbe sotto sforzo e su sistemi piccoli sarebbe sottosfruttato.
  - **Multielaborazione simmetrica:** più utilizzato, i core si gestiscono da soli (scalabilità) e si spartiscono il lavoro.
    Per dividere il lavoro, inizialmente si era pensato ad una **coda dei processi condivisa** tra i core da cui lo scheduler preleva i processi da eseguire, ma ciò causa *race condition*, il che implica l'introduzione del **lock**, che però causa un effetto ***"bottleneck"*** -> un core locka la coda, gli altri resteranno in attesa.
    Il problema si risolve con **M code**, una per core; il lock servirà comunque, ma in maniera più *limitata*. Resta da risolvere un altro problema: il **bilanciamento**; bisogna fare in modo che i processi si collochino in tutti i core, evitando che alcuni vadano in stato di *idle*.
    Se arriva un nuovo processo, va alla coda più vuota; ma, essendo in un sistema dinamico, dove la dimensione delle code varia, bisogna fare in modo che i processi passino da una coda ad un'altra. Si introducono dei **sistemi di migrazione**, che sono di 2 tipi:
    1) **Migrazione guidata:** qualcuno dei core chiama una routine a livello kernel che si occupa di spostare alcuni processi alle code più vuote;
    2) **Migrazione spontanea:** quando una coda rimane completamente vuota, il core recupera dei processi dalla coda più piena.

    Ci sono 2 tipi di **predilizione** nello scheduling: **forte** (*vincolo*), un certo processo viene eseguito solo su un certo core, tranne che in casi eccezionali (rendicontati dall'OS), e **debole** (*senza vincoli*), ogni processo può essere messo in ogni coda.

SISTEMI OPERATIVI

CAPITOLO 3

Come si gestiscono i processi in memoria centrale?

- In sistemi semplici, i processi usano direttamente gli **indirizzi fisici** per operazioni di fetch e store. Bisogna capire come dividere la memoria in sistemi **multiprocesso**, ci sono *3 possibilità*:
  - riservare una porzione della RAM al codice e ai dati dell'OS, il resto è potenzialmente sfruttabile dai processi;
  - Se l'OS è minimale, si carica in ROM, in modo da avere l'OS precaricato e la RAM disponibile per il processo.
  - Si può usare un software **simil-BIOS/UEFI** caricato in ROM, che caricherà l'OS in RAM, il resto è per il processo.

  ➢ La prima e l'ultima opzione, se un programma utente ha qualche difetto, potrebbero generare danni cancellando porzioni dell'OS (*Es.:* disco inutilizzabile).

  Per avere parallelismo si potrebbe usare il multithreading, che però non risolve il caso in cui l'utente voglia usare diversi programmi non relazionati.

- Bisogna effettuare una partizione dello spazio tra i processi, riservando i range di memoria ad ogni processo, a **compile-time**.

  ➢ *Limite compile-time:* o eseguo i processi in modo esclusivo, o si ricompila tutto quando il processo viene eseguito in una macchina diversa.

- Si deve fare in modo che, indipendentemente dal codice del processo, la memoria sia spartita correttamente. Si potrebbe fare multiplexing nel tempo, ma si dovrebbero spostare i dati del processo precedente sul disco rigido (**dispendioso**).
- Esporre la *memoria fisica* direttamente ai processi può portare a dei problemi: **operazioni di modifica/cancellazione di porzioni dell'OS** (come descritto in precedenza), **difficoltà di eseguire molteplici programmi** (saranno uno per volta).
- Per la gestione, quindi, si introducono 3 concetti:
  - **Rilocazione:** può succedere che, avendo 2 processi P1 e P2 salvati in range di RAM diversi, P2 effettui una chiamata ad una locazione appartenente al range in cui è memorizzato P1, invadendo il suo spazio. Si deve evitare, rilocando i processi, ci sono 2 modi:
    1) **A compile-time:** si decidono le zone a **compile-time**, per spostarli di slot serve ricompilare;
    2) **Rilocazione statica:** a **loading-time** del processo. Il codice dei processi fornisce un range *"canonico"*, detto **spazio di indirizzamento logico**; si manipola il codice prima dell'esecuzione, scansionando il codice in cerca di ***riferimenti*** (*call, fetch, store, …*), prendendo gli indirizzi e mappandoli, sommando l'indirizzo della locazione di inizio del processo, in modo da farlo rientrare nel range assegnato: se P1 ha un range 20.000-30.000 e il codice ha un riferimento alla locazione logica 3.000, si somma 20.000 (loc. di inizio del processo) a 3.000 ottenendo 23.000 -> **locazione fisica**.

    ❖ Funziona, ma ha un alto costo.

    Un altro problema è quello di **protezione** della memoria, bisogna fare in modo che nessun processo acceda a locazioni di altri processi o dell'OS; è possibile farlo con *supporto hardware*. Un metodo semplice è il **lock&key**, dove la RAM è divisa in porzioni da xKB, ognuna associata ad una chiave (*valore in binario*): quando un processo viene schedulato, la sua chiave viene copiata nella **PSW** e le porzioni di memoria avranno anche la sua chiave; quando il processo viene eseguito e genera un indirizzo, *confronta la chiave dell'indirizzo con quella del PSW* -> se **coincide**, è dentro il range, altrimenti l'operazione viene **negata**.

    ❖ Ad ogni context-switch, la chiave nella PSW è sovrascritta e, all'aumentare del numero di processi, diventa **poco scalabile**.

SISTEMI OPERATIVI

- **Spazio degli indirizzi:** insieme di indirizzi che un processo può usare per indirizzare la memoria. Abbiamo 2 nuovi registri *hardware*, un **registro base** (**RB**), che contiene l'*indirizzo della prima locazione del range*, e un **registro limite** (**RL**), con la *lunghezza del range*. Queste info sono salvate nel **PCB**.
  Si usa la **rilocazione dinamica**, dove interviene il **subcomponente** della CPU, l'**MMU** (*Memory Management Unit*) che traduce l'indirizzo logico in fisico sulla base delle informazioni dei registri (***RB+RL***).
  A differenza della **rilocazione statica**, la traduzione è fatta a **run-time** (ossia, al momento in cui si arriva ad un fetch o uno store, e non prima); prima però effettua un controllo, verifica se l'indirizzo logico sia maggiore di RL, in tal caso sforerà il range ed eseguirà una **TRAP**, cosicché una routine dell'OS termini il processo in kernel mode. In questo modo si protegge la memoria.
  I registri RB e RL sono protetti in modo tale che siano modificabili solo dall'OS.
  - ❖ Per ogni riferimento bisogna effettuare una somma e un confronto; le somme possono essere lente, a meno che non si abbia un circuito speciale per svolgerle.

- **Swapping:** in un **sistema multiprogrammato**, tanti processi riempiono velocemente la RAM, che deve essere liberata se necessario; l'OS prende la zona di RAM di un processo tra quelli allocati e la memorizza nel disco. Il processo non potrà più essere schedulato finchè i suoi dati non tornano in RAM.
  Per lo scambio, il processo rimosso deve essere tale che non intacchi l'**UEX** (*User Experience*); per fare ciò, interviene lo *scheduler di* ***medio termine***, che si occupa di questa scelta, con diversi criteri:
  1) **Il processo deve essere abbastanza grande da liberare la RAM necessaria o...**
  2) **Prende un processo che sta in background.**

  Il processo, dopo che viene spostato, non termina; il suo PCB si mantiene così da poter riprendere da dove si era fermato quando torna in RAM -> si spostano solo i dati.
  Si fa una verifica per riportare il processo in RAM ogni volta che ne termina qualcun'altro.
  Dopodiché, si applica la **rilocazione dinamica** per aggiornare i registri, in quanto gli indirizzi dei dati sono cambiati al momento dello spostamento sul disco.

  - ❖ Si possono verificare problemi in caso di **I/O pendenti**: se P1 esegue una chiamata di I/O, ma viene spostato e sostituito con P2, la risposta del controller, non sapendo dello scambio, sarà inviata a P2, sporcando la sua struttura dati.

  Frammentazione esterna: bisogna considerare il **vincolo di contiguità**: tutti i dati di un processo devono essere contigui in memoria (*non spezzettati*); se si liberano dei buchi non contigui si parla di **frammentazione esterna**, e se la somma di questi è superiore alla dimensione richiesta dal nuovo processo, non si può allocare, non deve essere frammentata.
  Si potrebbero spostare le partizioni degli altri processi per risolvere (**memory compaction**), ma sarebbe un'operazione *troppo onerosa*.
  Frammentazione interna: la *dinamicità* della dimensione di un processo potrebbe portare ad avere porzioni della partizione inutilizzati, se il processo si restringe, o a porzioni lasciate vuote di proposito, per la crescita del processo. Questo è **spazio sprecato**.

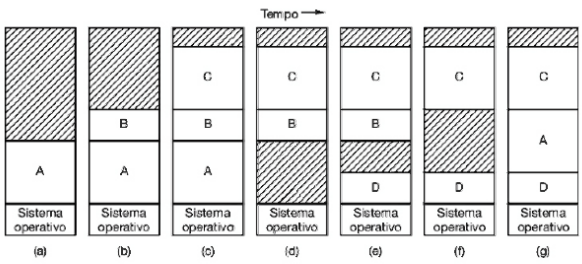


Figura 3.4 Cambiamenti nell'allocazione della memoria quando i processi arrivano in memoria e la lasciano. Le zone retinate in grigio sono memoria inutilizzata.

SISTEMI OPERATIVI

Com'è possibile gestire l'allocazione?

- Per sapere quali locazioni sono assegnate ad un processo, si introducono **strutture dati** che rappresentano uno storico per la CPU. Ci sono 2 metodologie basate sulla divisione della RAM in *blocchetti* della dimensione di una **unità minima allocabile**, esse sono:
    1) **Bitmap:** tabella di bit, con un numero di bit pari al numero di blocchetti della RAM, che indica lo stato del blocco (1 = allocato, 0 = libero).
       Se l'unità è ***troppo piccola***, avremo molti bit e la bitmap occuperà molto spazio.
       Se l'unità è ***troppo grande***, avremo **frammentazione interna** (arrotondamento).
        - Dimensione fissa: occupa lo stesso spazio anche con pochi processi.
    2) **Liste:** struttura *dinamica*, **lista doppiamente concatenata**, ordinata per indirizzo.
       Ogni nodo rappresenta una ***partizione*** associata ad un processo o un ***buco***, e contiene nome del processo, numero di blocchi occupati e blocco di partenza.
       E' memorizzata nella parte libera della RAM, il che teorizza un paradosso dovuto al fatto che, più la RAM è occupata, più il numero di nodi nella lista aumenta, diminuendo lo spazio disponibile ulteriormente; questo problema non si presenta nella pratica, perché le dimensioni tra *lista* e *spazio libero* sono di diversi ordini di grandezza.
        - La lista doppiamente concatenata favorisce la **coalescenza**: quando si libera una partizione, si controlla se ci siano più buchi contigui; se è così, si fondono i 2 nodi rappresentanti i buchi in uno solo (meno nodi nella lista).
- Serve trovare il **miglior buco** per poter inserire una nuova partizione, ricercando nella lista. Ci sono 4 possibili algoritmi:
    - **First fit:** parte dalla *testa* e si ferma al primo buco abbastanza grande per la partizione. Se è troppo grande, resta un buchetto -> **spazio inutilizzabile**, frammentazione esterna. Inoltre, i processi si addensano nelle prime locazioni;
    - **Next fit:** distribuisce uniformemente le partizioni, parte dall'*ultimo nodo allocato*;
    - **Best fit:** cerca un buco della taglia **esatta** e se non c'è prendo il più piccolo tra quelli abbastanza capienti. Cerca di evitare la frammentazione esterna, ma la peggiora in quanto porta alla generazione di ***microbuchi***, mai riempiti;
    - **Worst fit:** psicologia inversa al Best -> sceglie il buco più grande, in modo tale che il buco generato sia abbastanza grande per una successiva partizione.
- **First** e **Next** sono più *efficienti*, mentre gli altri 2 svolgono sempre una *scansione completa*.
  Se si usassero 2 liste separate, una per i **blocchi occupati** e l'altra per quelli **liberi**, First e Next migliorerebbero, avendo complessità minore; ma, al riempimento di un buco, bisognerebbe spostare un nodo da una lista all'altra.
  Se la lista è ordinata per dimensione, migliorano **Best** e **Worst**, avendo *tempo costante*, in quanto Best sceglierà sempre la testa e Worst sempre la coda.

Cos'è e a cosa serve la memoria virtuale?

- Il problema della *frammentazione esterna* è risolvibile applicando un'allocazione **non contigua**. Per farlo senza avere problemi nel recupero dei dati dalla memoria, è possibile vedere i processi da un punto di vista logico, detto **spazio di indirizzamento virtuale**: così è possibile avere virtualmente una memoria con locazioni contigue a disposizione del processo; in realtà, i dati saranno sparsi per la memoria fisica (serve tracciare tali allocazioni).

Come si può implementare la memoria virtuale?

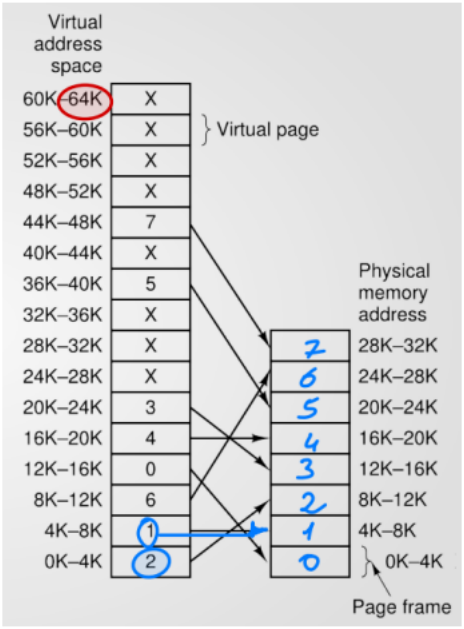
- E' possibile implementarla dividendo lo spazio di indirizzamento virtuale in **pagine**, ognuna di xKB fissi. Tutto è gestito dalla **MMU**.
- La memoria disponibile è *virtualmente infinita*, ma realmente ciò è impossibile: dato questo problema, non si tengono tutte le pagine in RAM, alcune saranno salvate in un *file su disco*, **area di swap**.

SISTEMI OPERATIVI

- La RAM è divisa in **frame**, con la stessa dimensione delle pagine. Ogni pagina ha un **numero di pagina**.
- L'idea è, quindi, di inserire la pagina virtuale in un qualsiasi frame fisico, come se fosse uno slot; *ogni buchino sarà riempito dalle pagine, risolvendo la frammentazione esterna*.
- Le pagine sono separate, non è possibile per un processo invadere le locazioni di un altro.
  Esempio: avendo due processi P1 e P2, entrambi con riferimento ad una locazione x, tale locazione sarà distinta per pagina di appartenenza.
  - ➢ A differenza dello swapping, non carica tutto su RAM, ma solo alcune pagine, così da evitare che il processo si blocchi.

Come si gestisce la paginazione?

- Per associare la *pagina al frame*, ogni processo possiede una struttura dati detta **tabella delle pagine**, con tanti elementi quante sono le pagine del processo. Il numero di elementi dipende dalla dimensione dello spazio di indirizzamento virtuale del processo.
- *Esempio:* avendo uno spazio di indirizzamento virtuale di **64KB** e la dimensione della pagina/frame a **4KB**, abbiamo **16** pagine numerate, e ognuna copre un certo range di indirizzi.



La pagina 0 copre le locazioni da 0 a 4095. La RAM è divisa in frame da 4KB numerati con un numero di frame.
Ogni **voce** della tabella rappresenta una pagina virtuale con all'interno il frame in cui è inserita la pagina.
La pagina 0 (voce numero 0) è memorizzata nel frame 2 della RAM fisica (valore 2 della voce). Le pagine che non sono in RAM sono barrate con X.

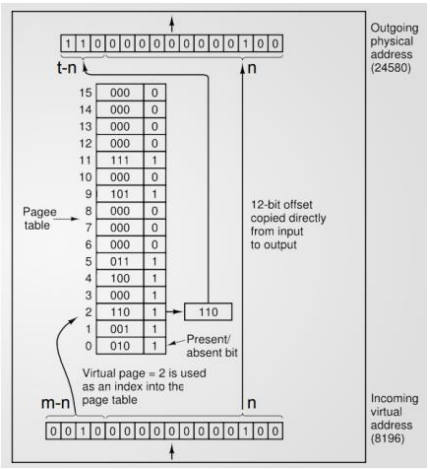
- Se la pagina richiesta non è in RAM, si verifica un **page fault**, TRAP che porta l'OS a recuperarla dal disco, assegnandola ad un frame vuoto o al posto di un'altra pagina, la quale verrà passata, invece, su disco.
Aggiornata la tabella, si ritenta la richiesta, che andrà a buon fine.

- Esiste un bit che indica se la pagina si trova in RAM, detto **bit di presenza** (**1 =** *presente*, **other =** *assente*).

- La tabella è gestita e modificata dall'**MMU**, che traduce l'indirizzo logico in fisico sulla base delle informazioni della tabella, per farlo:
  1) **Divide l'indirizzo virtuale con la dimensione di una pagina:** se l'indirizzo virtuale fosse 8196 e la dimensione di una pagina 4KB si farebbe 8196/4096, che torna come quoziente 2 (numero della pagina) e come resto 4 (offset -> distanza tra la prima locazione della pagina e quella effettiva);
  2) **Consulta il bit di presenza** e continua se è 1, altrimenti page fault;
  3) **Si ottiene il numero di frame corrispondente consultando la tabella**, che si moltiplica per la dimensione del frame: se il numero del frame fosse 6, allora 6*4096 = 32768 -> prima locazione del frame;
  4) **Si somma l'offset** (4): 32768 + 4 = 32772 -> locazione fisica.
  - ➢ Dato che della traduzione si occupa solo l'MMU, si potrebbe avere bottleneck, dato che deve recuperare le info dalla RAM.

- Gli indirizzi virtuali sono visti da MMU come ***maschere di bit***; quindi, se le dimensioni di frame, pagine e spazio di indirizzamento virtuale sono **potenze di 2**, le operazioni sono più *efficienti*.

SISTEMI OPERATIVI



- *Esempio:* lo spazio di indirizzamento virtuale è 64KB, quindi gli indirizzi virtuali sono a 16bit, la dimensione della pagina è 4KB (2^12) e per dividere basta uno shift a destra di 12 posizioni. Per la moltiplicazione uno shift a sinistra.
  Dividendo, i 12 bit meno significativi saranno l'offset, gli altri 4 il numero di pagina.
  La RAM è a 32KB, quindi avremo 8 frame da 3 bit.
  Per sommare l'offset (12 bit) basta un **bitwise OR**, che copia l'offset al posto dei bit nulli.
  In questo modo si ottiene l'indirizzo fisico.

- Quindi, avendo la dimensione dello spazio di indirizzamento virtuale a **2^m** (2^16) e la dimensione della pagina a **2^n** (2^12), il numero di pagine è dato dagli **m-n bit più significativi** dell'*indirizzo virtuale* (16-12 = 4 -> 2^4 = 16 pagine).
  Gli *indirizzi fisici* sono pari a **2^t**, e il numero di frame è dato dai **t-n bit più significativi** (15-12 = 3 -> 2^3 = 8 frame).

Che dettagli sono presenti in una voce della tabella delle pagine?

- Oltre al *numero di frame* e al *bit di presenza*, ci sono altre info:
  - **Protezione:** 2 bit che indicano l'accessibilità alle pagine in lettura e scrittura. Potrebbe esserci un terzo bit di esecuzione, se la pagina contiene codice eseguibile dalla CPU;
  - **Dirty bit:** potrebbe esserci una copia della pagina in RAM e su disco, se questa è stata prima spostata su disco e poi in RAM. La copia del disco si può cancellare se le pagine sono uguali e questo bit verifica se ci sono state modifiche -> *sovrascrittura*;
  - **Referenziamento:** utile per la sostituzione delle pagine e gestito da MMU, verifica se la pagina ha avuto **accessi** in un lasso di tempo. Inizialmente è messo ad 1 e viene *azzerato periodicamente*;
  - **Disabilita cache:** indica che la pagina **non deve essere messa in cache** (*Es:* se la pagina contiene istruzioni di I/O e viene messa in cache, la copia non ha più accesso alle porte di I/O e quindi non viene più aggiornata, ma la CPU userà quella);
  - **Bit di validità:** indica se la pagina è **allocata** (è parte dell'heap del processo?).

Cos'è la tabella dei frame?

- E' una sola per tutti i processi, con una voce per ogni frame della RAM, quindi la dimensione dipende da quest'ultima.
- Ogni voce contiene info sullo **stato** (frame *occupato* o *libero*) e, se occupato, il **PID** (*Process ID*). Non indica quale pagina è contenuta nel frame e viene consultata quando si deve inserire una pagina in memoria per vedere quali sono i frame liberi.

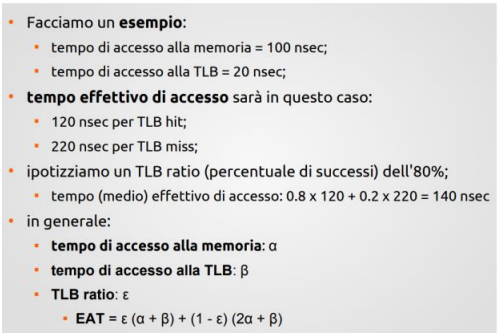
Come si progetta una tabella delle pagine?

- Bisogna considerare la **dimensione** e la **velocità di consultazione**, quest'ultima è gestita in 2 modi:
  1) **Memorizzare la tabella nella MMU**, valido quando vi era un numero esiguo di pagine, a causa dei **context-switch** che portano a riprogrammare la MMU azzerando i registri e caricando la tabella delle pagine del nuovo processo. Ad oggi *non è sufficiente*;
  2) **Memorizzare la tabella in RAM contiguamente**, nello spazio dell'OS. Nell'MMU c'è il registro **PTBR** (*Page-Table Base Register*) che contiene il puntatore alla prima voce della tabella del processo in uso.

  - ➢ Per il context-switch adesso basta aggiornare il PTBR, ma abbiamo un **maggior tempo d'accesso**: prima si accede alla RAM, poi avviene la traduzione dell'MMU e poi si riaccede alla RAM per operare.
- La **dimensione della tabella** è data dalla moltiplicazione tra il *numero di pagine* e il *peso dei puntatori*.
  Il **numero di pagine** è dato dal rapporto tra *spazio di indirizzamento virtuale* e *dimensione di una pagina*.
  Il **peso dei puntatori** è dato dall'*architettura* (32bit: 4byte; 64bit: 8)

Cos'è la TLB e a cosa serve?

- Per evitare un accesso aggiuntivo al fetch/store, entra in gioco la **TLB** (*Translation Lookaside Buffer*), detta anche **memoria associativa**, salvata all'interno dell'**MMU**. Essa, memorizza *alcune delle voci della tabella delle pagine*, quelle usate più recentemente, sfruttando la ricerca parallelizzata in hardware per una maggior efficienza.
- Nel processo di traduzione, è in mezzo tra la tabella e la RAM.
- Non si usa più del bit di **referenziamento** (ridondante) e si aggiunge l'info del **numero di pagina**, così da identificare la pagina di riferimento. Il bit di **validità** dice se la voce è piena; i bit di **protezione** e **dirty** dicono se bisogna riaggiornare la RAM alla cancellazione della voce.
- Come nella cache CPU, se la pagina viene trovata si ha **TLB hit**, ottenendo il numero di frame senza passare dalla RAM, altrimenti **TLB miss**, che implica l'accesso in RAM.
- Quando la cache è piena, MMU usa l'algoritmo **LRU** (*Least Recently Used*) per liberare spazio.
  - ➢ Al context-switch, per non avere voci di più processi, si effettuerà l'operazione di **flush** (non molto efficiente), che svuoterà la cafache con conseguente incremento di TLB miss.
- Per risolvere il problema, si introduce un ulteriore identificativo univoco per voce, l'**ASID** (*Address-Space IDentifier*), così da non scartarle al context-switch, in quanto, se il processo dovesse tornare ad operare, potrebbe aver bisogno ancora di quei dati. In pratica, l'ASID identifica *a quale processo appartiene quella pagina*.
- Tra thread fratelli, la TLB resta immutata.
- Il tempo medio effettivo di accesso ad una voce della tabella dipende dai tempi di accesso a TLB e RAM:
  - 1 solo accesso in caso di TLB hit **-> tTLB + tRAM**;
  - 2 in caso di TLB miss **-> tTLB + 2tRAM**.
- Per conoscere il tempo di accesso medio, serve conoscere il **TLB ratio**, ossia la percentuale di hit sul totale di richieste (80% in un sistema reale).
- Quindi, il **tempo di accesso effettivo** (***s*** – *Effective Access Time*) sarà minore con TLB.



- Se una pagina non è in TLB ma è in memoria, si ha un **soft miss**; se invece non è in memoria si ha un **hard miss**. In tal caso si ha **segmentation fault**.
- Il flusso da percorrere per ottenere una pagina dalla tabella delle pagine si chiama **table walk**.

Cosa sono la tabella delle pagine multilivello e quella delle pagine invertite?

- Resta da gestire il problema della ***dimensione*** della tabella. Ciò non era un problema con i sistemi a 32bit, dove, avendo pagine da 4KB ciascuna, per salvare 1milione di voci, bastavano 4MB; oggi, invece, avendo sistemi a 64bit (2^64 indirizzi disponibili), si dovrebbero memorizzare 2^52 pagine da 4KB, necessitando di più spazio di quello presente in RAM.

SISTEMI OPERATIVI

- **Tabella delle pagine multilivello:** si adotta un approccio simile alla *memoria virtuale*, per non mettere tutta la tabella in RAM e allocare le voci contiguamente, si divide la tabella in **gruppi di pagine** su più *livelli*, con una <u>struttura ad albero</u>.
  Esempio: struttura a 2 livelli -> al 1° livello c'è la tabella dei gruppi con ogni voce puntatore alla tabella di 2° livello.
  - I *bit più significativi* del numero di pagina indicano il **gruppo**, gli *altri* la **singola pagina**, i *rimanenti* indicano l'**offset**.
  - Il **PTBR** punta alla voce della tabella di *1° livello*, che punterà a quella di *2°*, dove trovare la pagina.
  - Normalmente, avendo **n livelli**, si dovrebbero effettuare **n+1 fetch**, a causa del passaggio al livello successivo. *Grazie a TLB, da n+1 si passa ad avere 1 solo fetch*.
  - Nei moderni OS, gli indirizzi generati dai compilatori non sono realmente a 64 bit, ma si taglia una parte, avendoli poi a 48 bit, per cui si necessita di massimo 4/5 livelli.

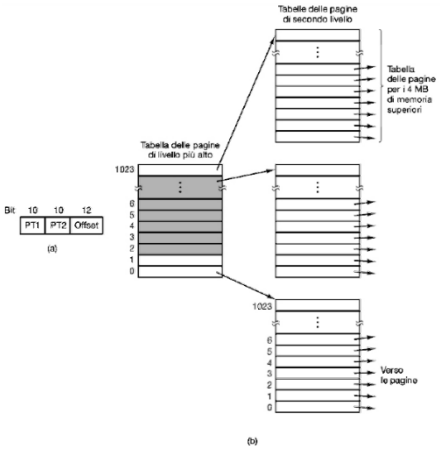


Figura 3.13 (a) Un indirizzo a 32 bit con due campi per le tabelle delle pagine. (b) Tabelle delle pagine a due livelli.

- **Tabella delle pagine invertita:** approccio alternativo, ma **<u>obsoleto</u>**; si ha solo tale tabella, senza gerarchie, che generalizza la *tabella dei frame*, si ha una voce per frame dove viene indicato se esso sia stato allocato o meno.

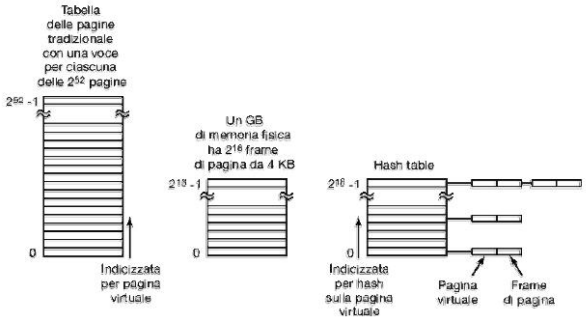


Figura 3.14 Confronto fra una tabella delle pagine tradizionale e una tabella delle pagine invertite.

- Se non si trova la pagina in tabella, si ha un **page fault**, altrimenti ottengo il frame.
  - ➢ **Ricerca lineare:** ad oggi, le dimensioni delle RAM sono maggiori, e una ricerca lineare richiederebbe un elevato tempo computazionale.
- Si risolve parzialmente il problema, implementando la tabella con una tabella hash, o utilizzando TLB.
  - ➢ Un altro problema è dovuto al fatto che si hanno le informazioni solo delle pagine allocate in memoria; in caso di **page fault**, servirebbe avere comunque una tabella delle pagine, allocata sul disco, con relativi *rallentamenti*.

Come si piazza la cache della memoria?

- L'accesso alla cache della RAM (no TLB) può avvenire prima o dopo la traduzione dell'MMU:
  - **Dopo di MMU:** la CPU genera l'indirizzo virtuale, MMU lo traduce e la cache ha l'accesso all'indirizzo fisico; se c'è *hit*, si recupera il dato e fine.
    - ❖ Usando gli indirizzi fisici (*univoci*), la cache potrebbe conservare dati di più processi, i cui identificativi saranno gli indirizzi fisici del dato in RAM.
    - ❖ Il caching è lento per la traduzione dell'MMU, portando ad un **bottleneck**.
  - **Prima di MMU:** la CPU genera l'indirizzo virtuale e si interroga subito la cache; se c'è hit si recupera il dato senza passare dall'MMU.
    - ❖ Caching più veloce.
    - ❖ **Univocità:** non si può usare l'indirizzo virtuale come ID del processo, avendo più processi in cache, perché più processi possono fare riferimento allo stesso indirizzo virtuale.
      *<u>Per risolvere, come per TLB, si può applicare il **flush** o l'**ASID**.</u>*
- In un sistema reale, si userà un **mix**: ci saranno *più livelli di cache* e, le più veloci (L1) saranno **meno capienti** e piazzate **prima di MMU**, le più lente (L3) saranno **capienti** e **dopo MMU**.

SISTEMI OPERATIVI

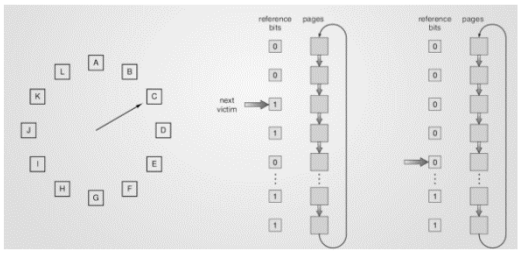
A cosa servono e quali sono gli algoritmi di sostituzione delle pagine?

- Quando si verifica un **page fault**, è necessario andare a recuperare la pagina dal disco, ma se i frame della RAM sono **pieni**, bisogna anche scegliere *quale di questi frame liberare*, e quindi definire quale sia la pagina più *"sacrificabile"* tra tutte, ossia quella che verrà referenziata il meno recentemente possibile, evitando di avere troppi page faults.
- Si introducono quindi degli algoritmi con il suddetto scopo, che avranno come metro di comparazione l'algoritmo teorico **OPT** (*ottimale*), che definisce quale sarà la pagina che non verrà referenziata mai più, o quella che lo sarà il più tardi possibile. E' teorico perché richiede molto tempo: per implementarlo, dovrebbe associare un'**etichetta** (numero di istruzioni da eseguire) ad ogni pagina, così da scartare quella con l'etichetta superiore.
- Gli algoritmi esistenti sono:
  - **Not Recently Used (NRU):** scarta la pagina non usata di recente, per conoscere il tempo di referenziazione, introduce 2 nuovi bit per ogni voce della tabella delle pagine:
    - **referenziazione (R):** più significativo. E' azzerato ad intervalli regolari; così, se al controllo è 1, vuol dire che è stato referenziato di recente.
    - **modifica (M):** a parità di referenziazione, si preferisce non scartare una pagina modificata.

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

    Dentro la classe, la selezione della pagina segue l'algoritmo FIFO.

    ➢ Non efficace, non valuta **QUANTO** è stata usata una pagina, ma solo **SE** lo è stata.
  - **FIFO e Seconda Chance, Clock:** non suddivide in classi ed effettua un controllo sull'età delle pagine e sulla frequenza di utilizzo; per farlo, controlla inizialmente il bit R: se è 1, si **risparmia** la pagina, si azzera il bit (come se fosse la più giovane) e sarà messa in coda; se è 0, si **scarta**, potrebbe essere la pagina risparmiata in precedenza.
    Per la logica che utilizza è anche detto algoritmo della **seconda chance**.
    Una variante è il **Clock**, che implementa una ***coda circolare***, con ogni nodo che punta al successivo. E' con **basso overhead**.

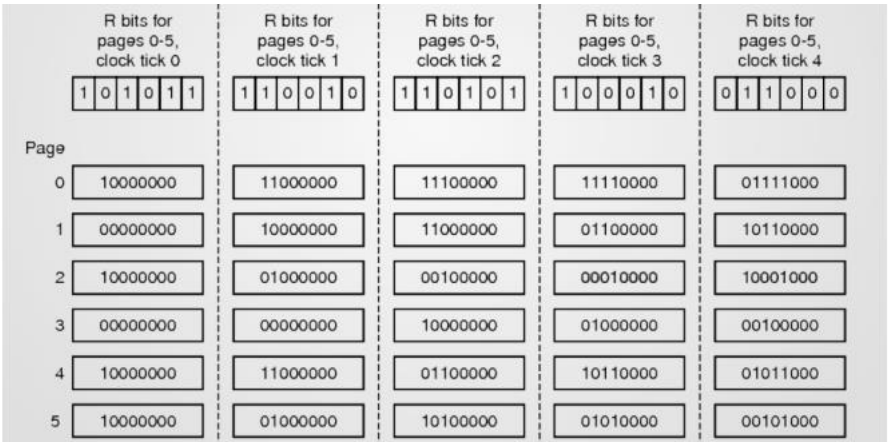
    

    ➢ Si avrà un puntatore alla testa, così, piuttosto che effettuare un'estrazione e inserimento per posizionare la testa in coda, basta spostare il puntatore, molto più efficiente.
  - **Least Recently Used (LRU):** usato nelle comuni cache di memoria, scarta la pagina *usata meno di recente* (non toccata da più tempo). Segue la logica per cui, se una pagina è stata referenziata di recente, ha più probabilità di esserlo nel prossimo futuro.
    Richiede un contatore per ogni pagina che parte dalla sua **prima referenziazione**, e verrà incrementato dall'**MMU** ad ogni istruzione (fatto dall'OS si avrebbe **overhead**).
    Ogni volta che la pagina viene referenziata, il contatore viene scritto nel rispettivo record della tabella delle pagine. *Verrà scartata quella con il contatore più basso*.
    Piuttosto che un campo del record, il contatore può anche essere una **matrice di bit**, nxn, dove ogni riga/colonna rappresenta un **frame**: alla referenziazione di una pagina nell'i-esimo frame, la riga corrispondente sarà messa ad **1**, e la colonna a **0**; così facendo, si arriverà al punto in cui *una pagina avrà la riga composta da 0*, da **scartare**.
    ➢ Dispendioso da implementare, richiede un contatore a 64 bit in hardware.
  - **Not Frequently Used (NFU)**: versione **software** di **LRU** (meno dispendioso), ad ogni pagina del frame si associa un contatore, aggiornato periodicamente sommando, ad ogni riferimento, il bit **R** al contatore. Dato che R è azzerato periodicamente, questa somma ci fa capire che, *se il*

contatore raggiunge valori alti, è per via di numerosi referenziamenti. Si scarta la pagina con il contatore più basso.

- Non pesa gli eventi nel tempo: una referenziazione recente deve pesare di più, così da scartare pagine con contatori elevati ma non referenziati da molto tempo.
  La pagina vive *"di rendita"*.

- **Aging: NFU** con l'aggiunta della logica del **peso** in base a quando è stata referenziata la pagina e con una minor quantità di bit necessari per il contatore. Per aggiornare il contatore, *si shifta verso destra*, rendendo il bit più significativo nullo, copiando **R** in questa posizione -> si tiene traccia degli ultimi **N cicli**; così, se la pagina è stata usata ultimamente, aumenterà di molto il contatore.

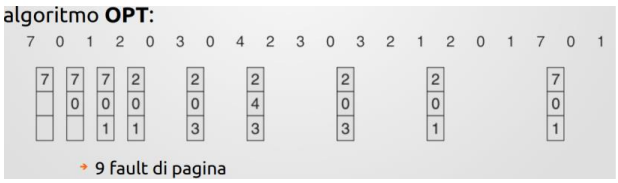


AGING

Esempio: una pagina con contatore 10000000 sarà risparmiata rispetto ad un'altra con contatore 01000011, anche se quest'ultima è stata usata più volte.
Lo shift non deve essere eseguito spesso (overhead elevato).
Non sappiamo **QUANTE** volte la pagina è stata usata, ma sappiamo **SE** e **QUANTO RECENTEMENTE** lo è stata.
Una pagina con contatore avente i bit più significativi a 0, non viene usata da tanto.

- **Confronto delle prestazioni:** ci si basa sul **numero di page faults**, che deve essere minimo.

Ovviamente, maggiore è la dimensione della RAM, minore sarà il numero di page faults.



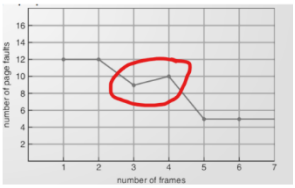
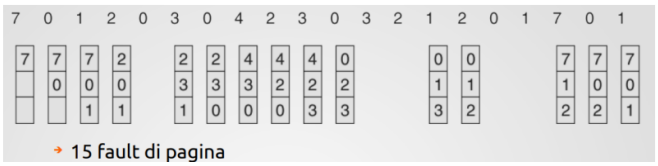


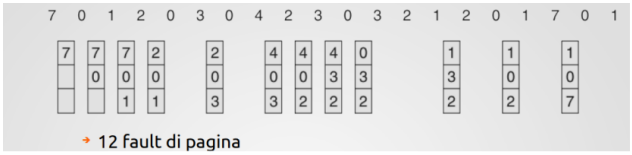
Figure 51: Anomalia di Belady

Per il confronto, prendiamo un esempio in cui abbiamo a disposizione *3 frame in RAM* e, in input, abbiamo una *sequenza di numeri di pagina* che rappresenta la **sequenza di accessi ad esse** (non ci interessa l'offset).
Se non si trova la pagina, avremo un **fault**, ma le volte successive si troverà in memoria.
Come riferimento si prende l'algoritmo **OPT** (non si può fare di meglio):
Con **FIFO** avremo:



decisamente peggiore di OPT. Inoltre, con alcune sequenze, aumentando il numero di frame della RAM, aumenta anche il numero di faults -> soffre dell'**anomalia di Belady**.

Tale anomalia non si presenta con **LRU**, dove avremo:



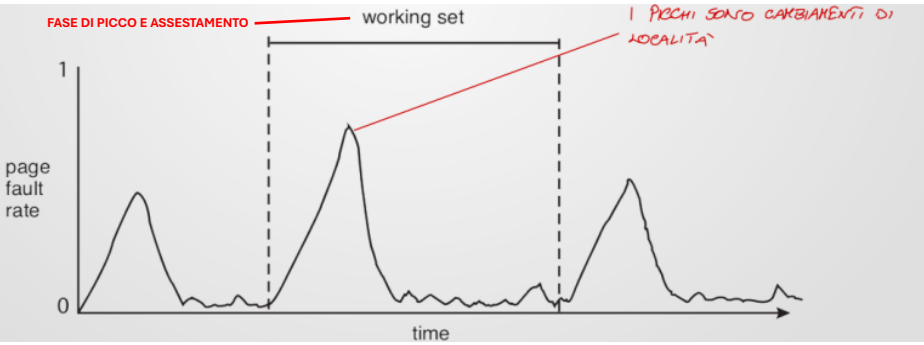
molto meglio di **FIFO**. Inoltre, come detto prima, non soffre dell'anomalia perché, negli algoritmi basati su LRU, vale la **proprietà di inclusione**: avendo 2 insieme S1 e S2, dove il primo rappresenta l'insieme delle pagine in un istante t con N frames e S2 l'insieme delle pagine nello stesso istante t con N+1 frames, allora *S1 è sottoinsieme di S2*.
FIFO si basa sull'età della pagina, quindi potrebbe scartare pagine usate di recente, LRU no.

Come si allocano i frame?

- Bisogna definire **quanti frame assegnare ad un processo**. Precisiamo che il concetto che si applica a priori, quando i frame sono ancora tutti vuoti, è quello di **demand paging**: se non trovo una pagina, c'è fault e la inserisco (*inizialmente ci saranno molti page faults*).
- Ipoteticamente, per assegnare un processo si potrebbero seguire 2 strade:
  - **Assegnare il minimo permesso dall'architettura in uso:** non ottima.
    Esempio: avendo un'architettura che permette di eseguire un'istruzione Assembly MOV[1000], #1 servirebbe innanzitutto *prelevare l'istruzione* e, se la pagina è in disco avremmo page fault, e poi *accedere alla locazione* 1000, che potrebbe puntare ad una pagina in disco, altro page fault. Servirebbe un numero minimo di frame obbligatorio.
  - **Assegnare tutta la memoria libera in quel momento** (utopistico).
- In uno scenario reale, dove bisogna spartire M frame tra N processi, le soluzioni sono:
  - **Allocazione equa:** assegno **M/N** frame a processo;
    - Non considera la taglia del processo.
  - **Allocazione proporzionale:** assegna i frame in base alla taglia dell'i-esimo processo, definito come S_i, con il seguente calcolo -> $\mathbf{A_i = S_i / S*M}$ con $\mathbf{S = \sum S_i}$
  - **Allocazione per priorità:** considera anche la priorità del processo, dando più importanza a questa che alla taglia.
- Gli algoritmi di sostituzione delle pagine si possono applicare a livello **locale**, considerando le singole pagine che hanno causato il fault, o **globale**, considerando tutte le pagine in memoria.
  Esempio caso globale (può mutare il numero di frame del processo): se P1 causa page fault, si scarta una pagina di P2 cosicché P1 possa ottenerlo -> **autobilanciamento**.
- Se un processo non ha assegnata la dovuta quantità di frame si verificano 2 casi:
  1. **Ne ha poco più del minimo:** ci sono tanti page fault che rallentano l'esecuzione di tutti i processi -> **trashing** (si usa più tempo per gestire i fault che per altro);
  2. **Ne ha meno del minimo:** il processo viene sospeso e spostato in disco, o *killato*.
- Per evitare il **trashing** bisogna adattare il numero di frame alle esigenze del processo (non la taglia), ossia la sua **località**, che è l'insieme di dati e istruzioni di cui il processo ha bisogno in un dato istante. Il concetto di località è ***dinamico***, varia nel tempo così come le esigenze.
- Per sapere quanto sia **grande** la località e monitorare il **fabbisogno** del processo, ci sono 2 possibilità:
  - **Working Set:** insieme composto dalle pagine referenziate in un lasso di tempo, recuperate dalla sequenza degli ultimi **Δ** accessi alla memoria, con Δ abbastanza grande, ma non troppo. Così, *si può capire se un processo necessiti di altri frame o se ne abbia troppi*.
    E' possibile anche capire se vi sia ***troppa poca RAM***, sommando le cardinalità dei working set dei vari processi e compararlo al numero di frame disponibili -> se è maggiore, bisogna **swappare** qualche processo (*anche interamente*).
    Per calcolare il working set, si usano i valori del **bit R** negli ultimi Δ cicli, salvandoli in dei **log**, uno per ogni pagina: si può capire se una pagina è stata utilizzata se nel log c'è almeno un 1. Il log è compilato grazie ad un **interrupt periodico**.

- **Page Fault Frequency:** un processo in sofferenza avrà *più page faults della media*. Si definisco 2 soglie, una **superiore**, che se superata indica che il processo *necessita di più frame* (probabile cambio di località), e una **inferiore**, che indica che il processo *ha troppi frame*. Ci sarà **autobilanciamento**, un processo in sofferenza otterrà frame da processi che ne hanno troppi.

- Entrambi i metodi fanno la stessa cosa in modo diverso; è possibile vederlo dal **grafico**:



nei cambi di località, il processo cambia strutture dati e codice, da recuperare dal disco.

Cos'è la politica di pulitura dei frame?

- Utile ad ottimizzare la gestione dei page fault; si controlla se:
  - **Vi siano frame liberi:** legge la pagina in disco, la scrive in un frame e aggiorna le tabelle delle pagine e dei frame.
  - **Tutti i frame sono tutti occupati:** complesso, si applica l'algoritmo di sostituzione delle pagine.
- Una possibile idea è quella di cercare di ritrovarci il più delle volte nel **caso 1**, avendo una *scorta di frame sempre liberi*. Per mantenerli liberi, si introduce il **paging daemon**, chiamato periodicamente, quando la CPU è a bassa percentuale di utilizzo e il numero di frame della scorta scende al di sotto di una soglia, liberandone alcuni tramite l'**algoritmo di sostituzione**.
- Per gestire i casi in cui l'algoritmo faccia una scelta sbagliata, si opta per dichiarare il frame come **libero**, mantenendo ancora occupata la locazione, sino alla **sovrascrittura**: così, se un processo richiama la pagina destinata alla sostituzione, ha ancora la possibilità di trovarla.
- Si applica l'algoritmo prima del necessario per scegliere il frame da pulire, ma rimediare velocemente in caso di scelte sbagliate.

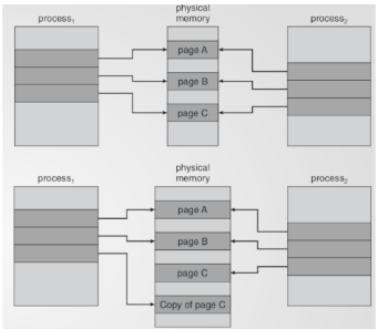
Come si può definire la dimensione della pagina?

- Per stabilire la dimensione della pagina, si sceglie tra un range delimitato dall'hardware. L'OS sceglie una *potenza del 2* (gestione efficiente: basta uno shift).
- La scelta della dimensione delle pagine implica delle conseguenze:
  - **Grande: meno frame e meno pagine** -> meno voci nella tabelle; **lettura/scrittura su disco più efficiente** -> avendo un grande blocco contiguo, la testina del disco meccanico non deve fare troppi riposizionamenti per ottenere i dati; **minor numero di page fault** -> si possono portare più dati in memoria.
  - **Piccola: riduzione della frammentazione interna alla pagina** -> la pagina è piccola, sarà anche più piena; **efficiente definizione del working set (e località)** -> la maggior parte della pagina sarà composta da info utili al processo e non locazioni vuote.
- La dimensione della pagina deve essere multiplo della dimensione del blocco in disco, così da non leggerlo *parzialmente*.

SISTEMI OPERATIVI

Come si possono condividere le pagine tra processi?

- Se 2+ processi dovessero lavorare su una stessa pagina, sarebbe inefficiente caricare 2 volte una copia identica, sia per lo spazio, che per la sincronizzazione di esse; si carica, quindi, una sola copia e i processi punteranno a indirizzi virtuali diversi (per forza di cose) che a loro volta punteranno allo stesso frame.
- Quindi, i processi puntano alla stessa cosa, ma la chiamano in modo diverso; ciò causa un problema alle cache che usano indirizzi virtuali, detto **aliasing**. In pratica, 2 processi, per ricercare una pagina, si basano sulla sua chiave (ASID) + indirizzo virtuale; dato che le chiavi usate dai due processi per la stessa pagina sono diverse, otterranno un riferimento della cache diverso, che potrebbe essere inesistente, causando page fault con successivo caricamento di un clone della pagine già presente in cache.
- L'uso di chiavi diverse, mostra un problema anche per l'uso della tabella delle pagine invertita (tabella dei frame), che causano falsi page fault; quindi, servirebbe consultare anche la tabella delle pagine e modificare, successivamente, il record della tabella invertita con la nuova chiave. Tale processo è LENTO e INUTILE.
- La soluzione è l'uso di tag fisici, che dipendono interamente dal valore di offset, pezzo in comune tra indirizzo virtuale e fisico. Si usa il tag fisico per fare una ricerca preliminare nella cache in parallelo alla traduzione fatta dall'MMU.
  Se non si trova alcuna voce, è sicuramente un miss, e non perde tempo; altrimenti, la ricerca sarà più veloce, dato che si basa sul risultato della ricerca (ridotto rispetto al totale).
  - Questo tipo di cache è più lenta e costosa di una a indirizzi virtuale e più veloce di una a indirizzi fisici.
- Altri problemi della condivisione sono gestiti in altri modi:
  - **Copy-on-write:** per evitare che n processi che condividono un file modifichino le copie degli altri, al tentativo di scrittura vengono create delle copie fisiche per gli altri processi, che non vedranno, così, le modifiche nel file. Di default, i file sono *read-only*, l'OS setta il bit di scrittura a 0; ma al tentativo di scrittura da parte di un processo, l'MMU controlla i permessi e invoca una procedura dell'OS per creare la copia, dopo setta il bit a 1 per entrambi i file, in modo che ognuno abbia un proprio file per scrivere. Così si può individuare il tentativo di scrittura.

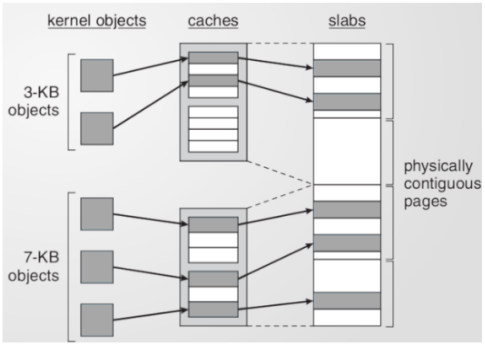


  - **Zero-fill-on-demand:** quando un processo necessita di nuove pagine, invoca **sbrk()** per ampliare l'heap, che farà in modo che l'OS destini altri frame all'heap.
    Come detto in precedenza, i frame non vengono ripuliti subito dopo averli segnati come liberi, per cui l'OS dovrebbe ripulirli prima di destinarli all'heap del processo.
    Per evitare di ripulire frame prima che il processo abbia effettivamente bisogno di quelli nuovi, si introduce la tecnica **Zero-fill-on-demand**, che prevede un frame speciale detto **read-only static zero page**, che contiene pagine piene di zeri, senza informazioni. L'OS farà puntare il processo a questo frame, facendogli credere di avere frame normali liberi da cedergli, anche se non è vero.
    Quando il processo dovrà finalmente usufruire materialmente dei nuovi frame, viene keseguito il **copy-on-write**: l'OS pulisce i frame ***necessari*** e cambia i puntatori nella tabella delle pagine
  - **Librerie condivise con linking:** spesso, i codici di un processo usano metodi di alcune *librerie esterne*, usate anche da altri processi. Ci sono 2 modi per includerle:
    1. **Linking statico**: la libreria è caricata a **compile-time** e il codice delle funzioni sono già nell'eseguibile.
       - Tutto il codice è già a disposizione (funzioni e sorgente).
       - L'eseguibile occupa molto spazio.
    2. **Linking dinamico:** la libreria è caricata in RAM a **run-time** e messa a disposizione del processo, facendo parte del suo spazio di indirizzamento.
       Tutti i processi che necessitano della libreria non avranno una copia l'uno, ma useranno l'originale in RAM, vedendola come se fosse nel loro spazio di indirizzamento.

- **Librerie condivise con file mappati:** consiste nel rendere alcuni frame dei puntatori alle allocazioni del disco in cui risiede il file, che sarà visto come se fosse in RAM. Quindi, quando il processo vorrà lavorare su un file, ci saranno dei page fault, si recupereranno le pagine dal disco e portate in RAM, e da quel momento potrà leggere/scrivere sul file, grazie ai frame puntatori. Quando i frame dovranno essere liberati, le pagine saranno sovrascritte nel disco.
  La stessa logica si può seguire per le librerie condivise, portando in RAM solo le parti utili della libreria, e non tutta.
  Nello spazio di indirizzamento di un processo, tra la zona del codice e l'heap, ci sono alre 2 zone:
  - ❖ **Superiore:** con i dati NON inizializzati (variabili dichiarate), piena di zeri, detta ***BSS***. Si mappa in memoria usando lo ***Static Zero Page***.
  - ❖ **Inferiore:** con i dati inizializzati, mappata dal file nel disco.

Come si alloca la memoria per il kernel?

- ➢ Il kernel ha bisogno di allocare la memoria per contenere le strutture dati di supporto (tabella dei processi). Ha a disposizione una parte della RAM e usa entrambi i metodi di interazione con la memoria fisica e della paginazione, in quanto alcuni dispositivi interagiscono in uno dei due.
- ➢ Linux usa lo slab allocator, dove lo slab è una collezione di pagine allocate fisicamente in modo contiguo; una cache è una collezione di slab.
  Ogni cache è specializzata in un ambito (chi con oggetti da 1KB, chi con 2KB, ...).
  Esempio: una cache da 3KB sarà composta da slab di 3KB e avrà capienza pari al m.c.m tra le dimensione della pagina e dello slab.
  Si risolve la frammentazione esterna (pagine contigue nello slab, ma slab separati) e quella interna (dimensioni degli slab fissi); ovviamente, è possibile solo per il kernel, dove la dimensione degli oggetti è definita a priori.



SISTEMI OPERATIVI

CAPITOLO 4

Che cos'è il File System?

- Si ha la necessità di gestire grandi moli di dati, e l'astrazione che ci permette di organizzare questi dati persistenti è detta **FILE**.
- Il file è una *lunga sequenza di byte* raggruppati in una sola entità, e vengono implementati da una componente software detta **file system**, che si occuperà dei dettagli di gestione ed implementazione come:
    - **Nome**: il file gestisce le regole sull'identificativo del file;
    - **Tipo di file**: normali o *"speciali"*. Il file directory è speciale, e contiene altri file al suo interno (o metadati), anche altre directory comunemente chiamate "cartelle". La directory principale è la radice. Un altro tipo speciale di file è detto **file dispositivo**, che rappresentano virtualmente i dispositivi;
    - **Tipo di accesso**: sequenziale o diretta, ad oggi la seconda con nuove memorie flash;
    - **Metadati**: informazioni del file che devono essere persistenti (nome, dimensione, estensione, maschera dei permessi, ...)
    - **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.
- Più processi possono leggere/scrivere sullo stesso file, usando un meccanismo di lock (simili ai lock visti in RAM per garantire mutua esclusione) che può essere di due tipi:
    - **Mandatory**: obbligatorio, quel processo non potrà leggere/scrivere il file in alcun modo, il lock è messo direttamente dall'OS; [Windows]
    - **Advisory**: può essere bypassato senza problemi, l'OS avvisa soltanto che ci potrebbero essere problemi in caso di accesso al file. [UNIX]
- I lock si possono anche distinguere tra **shared** ed **exclusive**, come nel problema dei lettori e scrittori: il gruppo dei lettori avrà un lock di tipo shared, mentre lo scrittore di tipo exclusive.

Com'è strutturato un file system?

- Lo spazio del disco è suddiviso banalmente in piccoli blocchi contigui. Un'ulteriore *astrazione* è quella delle **partizioni**, zone intere di memoria separate, ognuna delle quali può avere un OS.
- In passato, una parte del disco aveva un **MBR** (*Master Boot Record*), contenente una tabella delle partizioni e un blocco dinamico caricato dal BIOS per avviare l'OS; in tale blocco vi era un codice, il **bootloader**, che doveva individuare la partizione contenente l'OS selezionato e caricare in RAM il codice responsabile del caricamento del kernel di tale OS, il **boot block**.
- Il **bootloader** di Linux è **GRUB**. Adesso si usa **GPT** (*GUID Partition Table*), standard **EFI**.
- La partizione in sé contiene strutture dati, oggetti, altre info e gli **i-node**, che sono tabelle che rappresentano il file dal punto di vista del file system (il contenuto del file è in un'altra partizione); inoltre, si hanno il **file root directory**, che contiene tutti i file, e strutture dati per la gestione dello **spazio libero**.

SISTEMI OPERATIVI

Come si implementano i file?

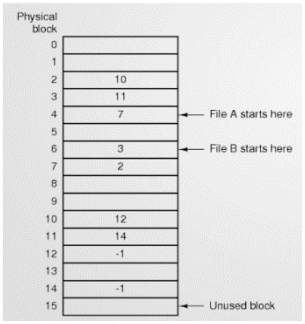
- Bisogna scegliere una sufficiente quantità di blocchi per contenere il file e allocarlo, ci sono 5 tipi di allocazione:
  - **Allocazione contigua:** si deve scegliere una sequenza di blocchi contigui abbastanza grande da contenere il file; per l'accesso al file, dentro la directory che lo contiene, è presente il primo **puntatore** ad esso e la **dimensione**, potendo leggere partendo da lì, sino all'offset derivato dalla dimensione.
    - ❖ **Frammentazione interna:** l'ultimo blocco potrebbe essere usato parzialmente, si deve prevedere la possibilità di *crescita del file*, concedendogli dello spazio utile, che, se non utilizzato, causerebbe uno spreco.
    - ❖ **Frammentazione esterna:** alcuni buchi rimarranno vuoti in mezzo.

    Per via di tali problemi, è ottimo per le memorie di sola lettura, non modificabili e di dimensione fissa, come **CD** e **DVD**.
  - **Allocazione con liste linkate:** ogni nodo è un blocco fisico del file e sono collegati attraverso puntatori, risolvendo la frammentazione esterna (ogni blocco può essere riempito). Il file system cercherà il più possibile di allocare contiguamente, ma se non trova blocchi liberi, salta al prossimo spazio libero.

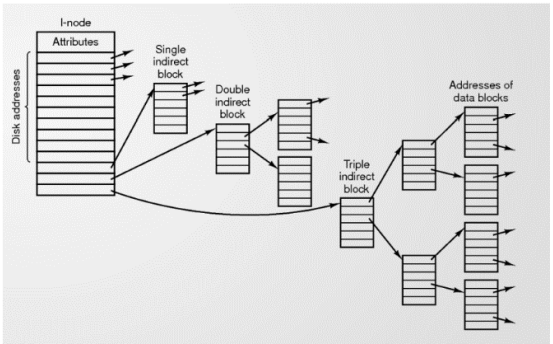
    Ogni blocco ha un puntatore al successivo, tranne l'ultimo, con **puntatore fittizio**.
    - ❖ **Spazio rubato dai puntatori:** avendo blocchi da 1KB, 4 byte saranno rubati dal puntatore, avendo a disposizione 1020 byte, che non è potenza di 2. Quindi, nella lettura di 1KB, bisognerà leggere i 1020 byte di un blocco e altri 4 byte del successivo.
    - ❖ **L'accesso è sequenziale**: attraversando la lista -> accesso lineare O(n).
  - **FAT (File Allocation Table):** risolve i problemi dei puntatori, implementando una tabella che li contiene tutti. La FAT è tenuta in disco e portata in RAM per **sincronizzarla**; quando si mette in RAM si ripresenta il problema dell'accesso lineare, ma è molto più veloce.

    Quindi, per la lettura di un file: recupero le info dalla directory, puntatore e offset, consulto FAT e faccio alcuni salti nella tabella, trovando il blocco ricercato.

    Esempio: file A, con blocco d'inizio 5, ricavato dalla directory, indice 4 della FAT; poi si salta al 7, poi al 2, poi al 10 e infine al 12.

  - **I-node:** struttura dati di medio-piccole dimensioni dedicata ad un file. Ogni nodo è identifica da un numero, detto **i-number**, e contiene tutti i metadati.

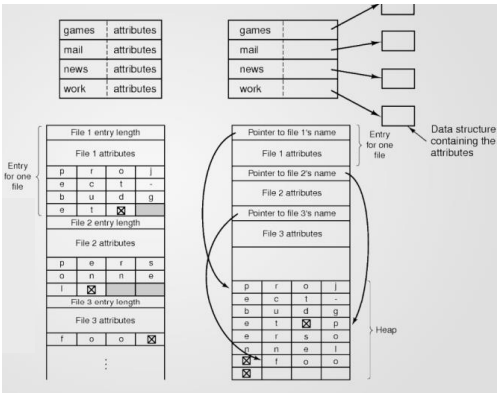
    

    Il file si ricerca in directory attraverso il **nome**, si trova il record con l'**i-number** e da lì si può trovare l'**i-node**, dove la prima parte contiene i *metadati* e la seconda è divisa in 13 record che descrivono la *sequenza di blocchi del file*.
    I primi 10 contengono *puntatori a blocchi*; l'undicesimo è un puntatore ad una tabella detta **blocco indiretto singolo**, contenente altri n *puntatori diretti a blocchi* (n = 1024 se ogni blocco contiene 4KB e ogni puntatore occupa 4 byte -> 4KB/4byte = 1KB = 1024); il dodicesimo punta ad una tabella detta **blocco indiretto doppio**, con altri record che puntano ad altre *tabelle contenenti i puntatori ai blocchi*; il tredicesimo punta ad una tabella detta **blocco indiretto triplo**, che punta a *tabelle che, a loro volta, puntano a quelle finali*.
    Si possono tracciare fino a: *(10 + n + n^2 + n^3) * dimblocco*.
    Esempio: se n = 1024 e dimblocco = 4KB -> 40 * 10^12 byte = **4TB**.
  - **Extent:** *gruppo di blocchi contigui*; usandoli, nelle liste/FAT/i-node si farà riferimento ad essi, gruppi di blocchi, piuttosto che ai singoli blocchi.
    - ❖ Permette di rimpicciolire le strutture FAT/i-node, a parità di dimensione dei file.

SISTEMI OPERATIVI

Come si implementano le directory?

- Le directory sono **file speciali**, con nome, metadati e un contenuto (file). Si può anche vedere come *vettore di record dinamico*, dove ogni record è un file, contenente metadati o i-number (i-node) o puntatore al primo elemento (FAT); tutte le info hanno dimensione fissa, tranne il **nome** -> si potrebbe rendere fissa la dimensione massima del nome, ma se si sceglie un nome piccolo è uno **spreco**. Ci sono 2 possibili strategie:
    1. Nella voce si raggruppano i *campi fissi* in una parte del record, la seconda parte sarà un **vettore di caratteri variabile** (nome). Per tracciare la lunghezza, si considera un campo iniziale, detto **File entry length**, e, data la dinamicità del vettore, si prevede un ***campo di terminazione*** del nome. Lo spazio rimanente, sarà utile in caso di **rinomina**, con *rischio frammentazione*, lieve grazie al fatto che si tratta di una struttura piccola sincronizzabile in RAM, e/o **allineamento**, che fa in modo che la dimensione della entry sia multipla della dimensione della word generica.
    2. Si divide in **gruppi**, la prima parte con i campi fissi dei file e un puntatore alla seconda parte, che consiste in un heap che memorizza i nomi. Si risolve il problema dell'allineamento, dato che l'heap è portato in RAM all'occorrenza.
    E' più facile deframmentare l'heap in caso di rinomina, grazie alla sua piccola dimensione. L'unico problema è la ricerca, che è **lineare** -> risolvibile usando hash tables o sfruttando la RAM come cache del disco.



Come si condividono file sul file system?

- In un sistema *multiutente*, si deve prevedere la possibilità di condividere un file, facendo in modo che l'utente destinatario della condivisione abbia un riferimento al file originale:
    - **I-node:** deve puntare allo stesso i-node dell'originale, e ci sono 2 modi diversi:
        1. **Soft link / Symbolic link:** file speciale dentro la directory destinazione, gestito dall'OS. Può avere un nome diverso dal file originale e i-node proprio, contiene il ***percorso al file originale***. OS segue il path e apre il file.
        Le modifiche dell'utente 2 avverranno direttamente nel file originale.
        Quando il file originale viene **eliminato** o **spostato**, si perde il riferimento e il soft link torna un *errore*. I permessi sono gestiti allo stesso coerentemente al file originale e un soft-link può puntare ad un altro soft-link.
            - Si possono creare reference a directory antenate.
            - Si può fare riferimento a file contenuti in file system diversi.
        2. **Hard link / Link fisico:** file (non speciale), contenuto nella directory dell'utente 2, che punta allo *stesso i-node* del principale, avendo lo stesso **i-number**.
        Quando viene cancellato il file in una delle 2 directory, si perderà solo il *riferimento*, e non verrà cancellato l'i-node; per fare ciò, si usa un **contatore** di riferimenti tra i metadati, che permetterà la cancellazione dell'i-node solo quando scenderà a 0.
            - Creando una reference ad una directory antenata, si genererà un **loop** nel tentativo di aprire il file. Si deve impedire la creazione di riferimenti a tali directory, almeno a *livello utente*.
            - Non si possono creare i **cross-file system link**, ossia riferimenti a file di file system diversi su UNIX, dove i file system sono *"a cascata"*, contenuti nella **root directory**.

Questo perché l'hard-link punta all'i-node, e cambiandolo si cercherà tale i-node nello stesso file-system.

- Anomalia dell'**accounting**: se si elimina il riferimento originale, il file resta di proprietà dell'utente 1 e, se altri usano il riferimento per accedere, le operazioni peseranno sull'uso del disco da parte dell'utente 1, anche se non sta facendo nulla.

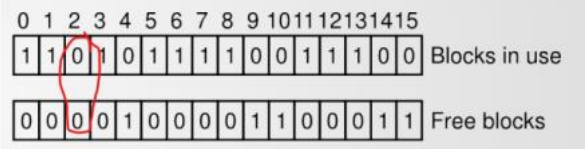
- **FAT:** non si possono creare riferimenti, in quanto le info del file, che risiede nella directory dell'utente 1, non saranno di proprietà dell'utente 2, che potrà leggere solo il puntatore alla FAT. Si deve duplicare la lista con i riferimenti.

Come si gestiscono i blocchi liberi?

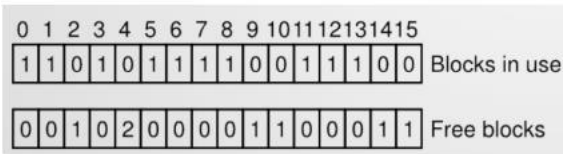
- Per tenere traccia di quali siano i **blocchi liberi**, ci sono 2 modi:
  - **Bitmap:** come in RAM, grandi in proporzione alla grandezza dei blocchi e della partizione; ha dimensione fissa, memorizzata su disco e si pagina per portarla in RAM.
  - **Liste concatenate (dinamiche):** memorizzate su disco, ogni nodo indica un *blocco libero*. Tale nodo, se memorizzato nello stesso blocco, ruberebbe spazio ad altre info.
    Quando il blocco viene *allocato*, il nodo viene **cancellato**. La dimensione della lista è inversamente proporzionale al riempimento del disco.

Come si controlla la consistenza?

- Se l'OS crasha durante operazioni sul file system, si genera **inconsistenza**; ossia si verifica un problema di *correttezza delle informazioni* presenti nelle strutture dati manipolate prima del crash. Al riavvio, l'OS deve eseguire dei **controlli** prima di poter usare il file system, quali:
  - **Controllo di consistenza sui blocchi:** si confrontano la struttura contenente i blocchi *liberi* con quella contenente i blocchi *in uso* (i-node / FAT) tramite 2 vettori inizialmente pieni di zeri. Scansionando i blocchi in uso, se vedo l'i-esimo blocco, incremento il valore nell'indice del vettore corrispondente. Se tale valore è **>1** c'è un problema.
    Stesso controllo si fa scansionando i blocchi liberi verificando la stessa condizione.
    Inoltre, i 2 vettori devono essere ***complementari***, se all'indice i un vettore ha 1, l'altro deve avere 0.
    Per correggere i problemi, ci sono 2 casi da considerare:
    - **Non sono complementari:** si contrassegna come libero, mettendo 1 nel vettore dei liberi;



    - **Contatore >1:** si ricrea la sequenza degli n file, togliendo da essa quel blocco. Inevitabilmente, uno dei 2 file si danneggerà, non avendo i dati del blocco.



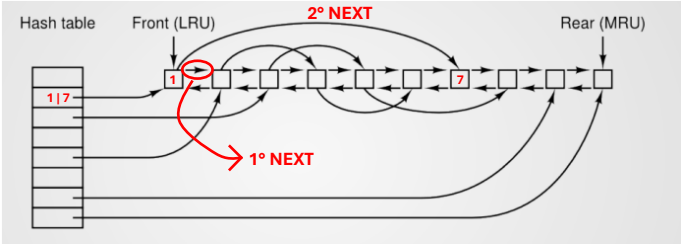
  - **Controllo di consistenza sui riferimenti agli i-node:** si usano 2 vettori, scansionando tutte le directory, *partendo dalla radice*. Il primo vettore conta quante volte ogni i-node è riferito (il

contatore può essere >1, più riferimenti); il secondo conta i riferimenti controllando gli i-node stessi. **I vettori devono essere uguali**.

- **Journaling:** controlli come quelli precedenti sono *dispendiosi* in grossi file system. Si preferisce costruire un **log** dove, prima di eseguire ogni macro-operazione sui **metadati** (non sui dati: *meno importanti*), ci si appunta tutte le piccole istruzioni che la compongono.
  In caso di crash, l'OS, non potendo sapere a quale istruzione della macro-operazione si sia arrivati, le **riesegue tutte**.
  - ➢ Il controllo è localizzato per il file che si stava modificando al momento, non su tutto il file system.

Come si possono ottimizzare le prestazioni del disco?

- Ci sono alcune soluzioni da poter adottare per **aumentare le prestazioni**, come:
  - **Cache del disco:** porzione della RAM che l'OS si riserva per questo compito, è implementata via software. Si controlla se il blocco ricercato sia già presente [2], in tal caso si ha cache **hit**, altrimenti **miss**.
    Essendo in RAM, si ragiona in termini di frame, che si vedranno come blocco di blocchi.
    I blocchi si ordinano in una **lista concatenata** con testa **LRU** e coda **MRU** (*Most Recently Used*) e, ogni volta che un blocco viene usato, viene portato in coda e gli altri vanno avanti. In caso di miss e cache piena, si estrae la testa, si prende il blocco richiesto e si inserisce in coda.
    Per valutare se ci sia hit o miss, si effettua una ricerca lineare nella lista. Si usano le ***tabelle hash concatenate***, dove ogni cella corrisponderà a tale lista.
    Ogni nodo avrà 2 puntatori **PREV** e **NEXT** riferiti alla lista e 2 alla tabella hash.
    Esempio: nella seconda entry della tabella hash si hanno i nodi 1 e 7, il secondo NEXT del nodo 1 punterà al 7 e l'altro punterà al 2.

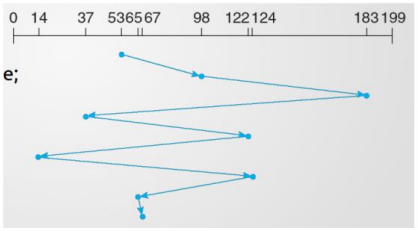


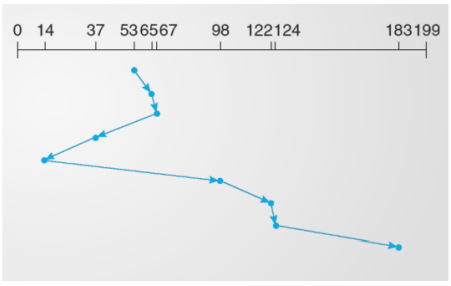
    Tale cache funziona anche in scrittura, alla scrittura del blocco, scrivo sulla copia della RAM. Ci sono 2 tipi di **sincronizzazione**: **asincrona**, periodicamente o al momento dello scarto, con possibile perdita di dati in caso di crash, e **sincrona**, ad ogni scrittura, utile per i metadati. La cache si può ottimizzare in 2 modi:
    - ❖ **Free-behind:** scrivendo su un blocco, diventa **MRU**, ma avendolo scritto adesso, è *più probabile che non sia utile nel prossimo futuro*, si **scarta**.
    - ❖ **Read-ahead:** nei dischi elettromeccanici, quando si porta un blocco in cache, si portano anche quelli adiacenti. Segue la logica per cui, *se si necessita di un blocco, potrebbero essere utili anche quelli appena successivi*, che apparterranno probabilmente allo stesso file.

  - **Scheduling del disco:** nei dischi elettromeccanici, l'efficienza di lettura/scrittura è legata al tempo di posizionamento della testina, detto seek time; bisogna anche considerare che, essendo in un sistema multiprogrammato, vi sono molteplici richieste di I/O. Il seek time dipende dalla distanza che la testina deve percorrere per raggiungere i dati, per cui bisogna schedulare le richieste con un ordine tale da rendere tale lavoro efficiente.
    Esempio: avendo lista di richieste di I/O che prevede come numeri di cilindro [98, 183, 37, 122, 14, 124, 65, 67] con testina posizionata inizialmente in posizione 53 e piatto contenente 200 tracce da 0 a 199, si deve considerare anche la distanza tra 53 e la prima testina. Alcuni algoritmi di scheduling sono:

SISTEMI OPERATIVI

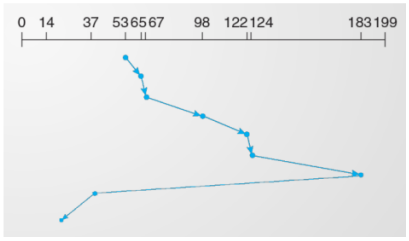
- **FCFS (First Come First Served):** logica FIFO, in ordine di arrivo. La testina si trova a passare da sinistra a destra e si avrà [98, 183, 37, 122, 14, 124, 65, 67] per una distanza totale percorsa di 640.
  Inefficiente.



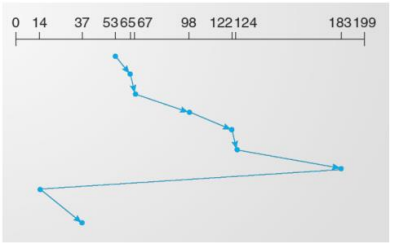


- **SSTF (Shortest Seek Time First):** per distanza più breve dalla posizione attuale. Si avrà un ordine del tipo [65, 67, 37, 14, 98, 122, 124, 183], percorrendo una distanza totale di 236.
  Non equo: se continuano ad entrare richieste riguardanti tracce vicine tra loro, il tempo per schedulare le richieste più vecchie è indefinito, **starvation**.

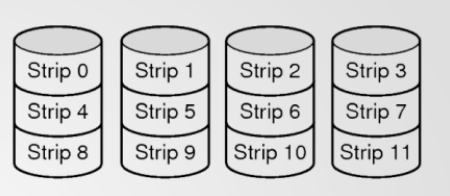
- **Scheduling per scansione:** detto algoritmo dell'*ascensore*, la testina ha un verso, su o giù; arrivato alla fine cambia verso. Si avrà quindi [65, 67, 122, 124, 183, 37, 14], per una distanza totale di 299, ma senza starvation.
  In implementazioni reali, non arriva alla fine delle tracce, ma a quella dell'**ultima richiesta**.



- **Scansione circolare:** variante ***circolare***, dove non cambia verso e, arrivato alla fine del verso prestabilito, salta all'inizio del lato opposto. Si dà così *priorità alle richieste vecchie*, dato che nel cambio verso potrebbero essere aggiunte altre richieste che farebbero perdere tempo. Si avrà [65, 67, 122, 124, 183, 14, 37], per una distanza totale di 322.
  In caso di **alto carico** conviene usare questo, che migliora il tempo d'attesa medio per richiesta; in caso di **basso carico** conviene la variante classica o SSTF.

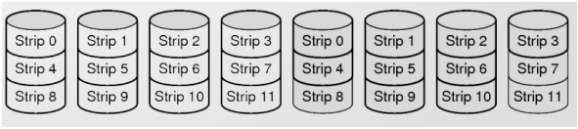


- **RAID (Redundant Array of Inexpensive/Indipendent Disks):** permette di avere parallelismo nelle operazioni di I/O, prevedendo **più dischi fisici**, pilotabili individualmente, e un'**astrazione**, implementabile in hardware nel controller o software nell'OS, per cui essi sono gestiti dall'OS come un unico disco.
  Per prestazioni massime, conviene che i dischi siano uguali di capienza e prestazioni. Così, lo spazio aumenta con il numero di dischi -> 4 dischi da 1TB = capienza di 4TB.
  Esempio - RAID di 4 dischi fisici:
  - **RAID 0 (striping):** divide il file system in parti, dette **stripe**, composte da *blocchi*. Gli stripe sono divisi equamente tra i vari dischi seguendo la logica *Round Robin* (stripe 0 su disco 1, 1 - 2, 2 - 3, 3 - 4, 4 - 1, ...).
    Quindi, per leggere un file, bisogna leggere *tutti gli stripe del file system che lo contengono dai vari dischi*, ognuno dei quali leggerà con una certa velocità.
    La velocità di lettura del file dipenderà dalla velocità di lettura del singolo stripe nel disco; lo stesso vale per la scrittura. Tutto è in **parallelo** tra i dischi.
    Il caso peggiore è che gli stripe vengano inseriti tutti nello stesso disco, che sarà come avere solo quello piuttosto che 4.
    Poco efficiente: e un disco si rompe, si perde tutto.

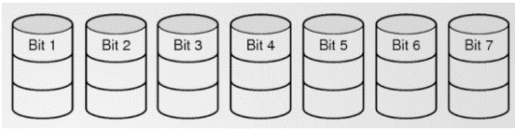


SISTEMI OPERATIVI

- **RAID 1 (mirroring):** introduce la **ridondanza**, duplicando gli stripe. Il contenuto e ogni modifica di ogni disco sono copiati in un altro; si necessita del *doppio dei dischi* (8 dischi, di cui 4 di backup) mantenendo però la **stessa capacità**.
  Quando un disco si rompe, si recupera il contenuto da quello di **backup**, ricostruendolo. Nei server si usano dei dischi **spare**, che si attivano ai guasti.
  Troppo **costoso** per il numero di dischi da implementare alla pari di prestazioni.

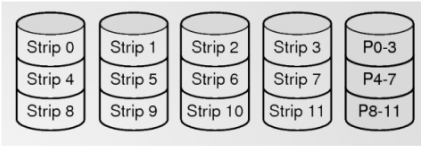


- **RAID 2:** cambia il tipo di ridondanza, usando i **codici di Hamming**, così da poter correggere il bit errato. Lo striping avviene a *livello di bit*: con 4 dischi base, si avranno codeword da 4 bit di dati e 3 di ridondanza, per un totale di 7 bit -> serviranno quindi 7 dischi per ridondare i dati (meglio degli 8 di RAID 1).

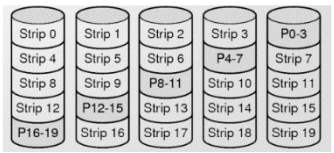


  Alla rottura di un disco, si corromperà solo quel bit per tutte le codeword, correggibili.
  Serve sincronia tra i dischi: se si perde tempo, si leggono codeword incomplete, con bit mancanti. Questo a causa dello striping a livello bit.
- **RAID 3:** cambia il tipo di ridondanza, usando il **controllo di parità**, che rileva gli errori, senza correggerli di suo. Si avranno codeword da 4 bit di dati e 1 di ridondanza, in questo modo il numero di dischi scende a 5.

  Al guasto del disco, il controllo rileva il problema senza indicare *qual è il bit rotto*; per correggere, basta ricalcolare i bit.
  Lo striping è ancora a livello di bit, serve **sincronia**.
- **RAID 4:** stessi dischi di RAID 3, ma lo striping torna a *livello stripe*. Cambia il tipo di ridondanza, calcolata con uno **XOR** tra gli stripe della stessa riga, da salvare nella relativa riga dell'ultimo disco (che contiene tutte le ridondanze).
  Alla scrittura, bisogna aggiornare la ridondanza associata allo stripe; ma, piuttosto che leggere dai primi 4 dischi e scrivere sul 5° disco, basta fare uno *XOR tra blocco vecchio, vecchio stripe* (probabilmente in cache – si risparmiano letture) *e nuovo stripe*.
  Ad ogni modifica agli stripe, lavora anche il disco 5, molto più degli altri, **aumentando la probabilità di guasto di questo**.



- **RAID 5:** stessa logica del RAID 4, ma **distribuisce** le ridondanze in modo omogeneo tra i dischi. Anche il 5° disco memorizzerà dati.
  Si avranno prestazioni x5 in lettura/scrittura e stesso spazio degli altri RAID.



SISTEMI OPERATIVI

Cos'è l'SSD?

- Memorie di massa volatili basati su memorie **NAND**, dette **memorie flash**. Il numero di sovrascritture di una cella è *limitato*.
- La memoria è divisa in blocchi flash, composte da unità minime di allocazioni, dette **pagine** (non le stesse della paginazione). Un blocco può essere vuoto, pieno o parzialmente pieno.
- Le scritture sono molto più lente delle letture, perché controlla la cella per ripulirla prima di sovrascriverla. Un modo per pulire la cella è il **F2FS** (*Flash-Friendly File System*), che la contrassegna come spazzatura e memorizza il nuovo contenuta in un'altra pagina dello stesso blocco; però, le singole pagine non sono cancellabili, *si possono cancellare i blocchi interi*.
- Il **garbage collector** riscrive tutte le pagine valide in un blocco libero, grazie a **TRIM**, istruzione che permette di ignorare le pagine "*spazzatura"* per non riscriverle nel nuovo blocco, riducendo il numero di scritture. ***Aumentano le prestazioni, non dovendo riscrivere tutto*.**
- In questo modo, è poi possibile eliminare l'intero blocco, che avrà solo le pagine "*spazzatura"*.
  - **Assenza della testina:** riduce a 0 il seek-time.

SISTEMI OPERATIVI

FORMULE

PAGINAZIONE

Dimensione pagina = spazio di ind. Virtuale/numero di pagine virtuali

Dimensione frame fisico = Dimensione pagina

Numero di frame fisici = Dimensione ind fisici / dimensione frame fisico

Dimensione tabella delle pagine = numero di pagine * peso dei puntatori

Numero di pagine = spazio di ind. Virtuale / dimensione pagina

Peso puntatori = dato dall'architettura (32bit: 4byte; 64bit: 8)

TLB

α = tempo di accesso alla memoria

β = Tempo di accesso alla TLB

ε = TLB ratio

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

Shortest Process Next

S_{n+1} = $S_n (1 – a) + T_n * a$

OVERHEAD

%Overhead = [c/(c+q)] **c =** tempo context-switch **q =** quanto **c+q =** tempo totale

I-NODE

n = numero di puntatori = dimBlocco / pesoPuntatore **pesoPuntatore =** (32bit: 4byte; 64bit: 8)

Blocchi memorizzabili: (10 + n + n^2 + n^3) * dimBlocco

FAT

Blocchi occupati dall'offset = offset / dimBlocco **x =** numero di blocchi utilizzati dal file

Dim minima del file = (x-1 * dimBlocco) + 1 byte **1 byte ->** ultimo blocco vuoto