Sistemi Operativi

Appunti di Jaime Corsetti, supervisione di Gianluca Sassetti

June 25, 2020

1 Definizione e storia

Un sistema operativo è un insieme di programmi che funge da interfaccia fra hardware e software applicativo. Il suo scopo principale è quello di gestire le risorse hardware, ma fornisce anche un alto livello di astrazione. Di seguito una sintesi della storia dei sistemi operativi, il cui principio ispiratore fu sempre quello di aumentare l'utilizzo della CPU.

Valvole e schede perforate I primi computer (1946-1955) erano privi di sistema operativo. L'utente inseriva le istruzioni sequenzialmente tramite schede perforate. Successivamente, con la diffusione delle periferiche, vennero sviluppati i primi driver, insieme alle prime librerie. I tempi di setup rendevano la programmazione macchinosa e l'utilizzo della CPU scarso.

Transistor e batch Per massimizzare i termini di utilizzo si separò la figura dell'operatore da quella del programmatore. Si adottò una tecnica di batch, nel quale programmi simili (ad esempio dello stesso linguaggio) venivano raggruppati ed eseguiti insieme per minimizzare i tempi di setup. In caso di errore l'operatore non sapeva effettuare il debug e doveva passare da un job all'altro manualmente. Per superare queste problematiche venne introdotto il resident monitor, di fatto il primo sistema operativo. Questo veniva automaticamente invocato all'avvio ed aveva il compito di passare il controllo da un job all'altro quando necessario. La sequenza di esecuzione dei job era decisa da una serie di schede di controllo scritte in un Job Control Language (JCL). Il resident monitor era dunque costituito da un interprete JCL, un sequenzializzatore di job e dai driver per l'accesso ai dispitivi di I/O. Un miglioramento delle prestazioni si ebbe con la sovrapposizione di I/O e CPU, nel quale la CPU, anzichè leggere e scrivere su schede, utilizzava nastri magnetici molto più veloci, che inoltre potevano essere scritti e eletti da più dispositivi contemporaneamente. Con i dischi magnetici nacque lo spooling, tecnica tramite la quale la CPU poteva leggere sul disco contemporaneamente ai dispositivi di I/O. Nasce in questo modo il paradigma moderno di job scheduling.

Multiprogrammazione e sicurezza Con l'avvento dello spooling nasce il problema dell'esecuzione contemporanea di più job. Sorge inoltre, con l'evoluzione dei dispositivi di I/O, la necessità di assicurare un'adeguata protezione della macchina per evitare che l'esecuzione di un job possa influenzare in maniera indesiderata quella di altro (es: un job scrive nell'area di memoria riservata ad un altro job). I tre tipi di protezione sono:

- **I/O**: si separa l'esecuzione in modalità **User**, che non può accedere all'I/O e la modalità **Kernel** (riservata per le routine di sistema), che invece può. In questo modo le operazioni di I/O passano sempre per il sistema operativo che ne controlla la legittimità.
- **Memoria**: sono associati a ciascun job dei **registri limite**, che segnalano un limite superiore per le aree di memoria indirizzabili da un job.
- **CPU**: ad ogni job è associato un timer, alla scadenza del quale la CPU ritorna al monitor.

2 Interfaccia utente

Elemento pricipale di interfaccia tra SO e utente è l'interprete dei comandi, generalmente implementato come CLI o GUI.

Command Line Interface I comandi sono inseriti in forma testuale in un interprete (o **shell**) ed eseguiti. A seconda dell'implementazione l'interprete può essere un processo separato o eseguito dal kernel stesso. Il codice di ogni comando può essere parte dell'interprete stesso o fare riferimento ad un file separato che specifica il codice (**shell script**). Questo permette un'elevata possibilità di customizzazione.

Graphical User Interface Classica implementazione a desktop. Generalmete più user friendly, cala il livello di prestazioni e di specificità delle operazioni.

Fra SO e i comandi dettati da un utente si interpongono **system call** e **API**(App. Program Interface). Le prime sono programmi che controllano le funzionalità di base del SO non direttamente accessibili dall'utente (come la gestione della memoria). Le system call sono specifiche di ogni SO. Per quanto riguarda le API, queste, a seconda del SO, invocano le system call e restituiscono i risultati all'utente, aggiungendo un ulteriore livello di astrazione. Le API non fanno quindi parte del kernel. In tutti i casi, ovvero sia nell'invocazione dell'API che della system call, si verifica un'**eccezione** o **trap**, nella quale si passa da user mode a kernel mode.

3 Architettura

La progettazione di un SO segue i due principi **KISS** (Keep It Simple Stupid) e **POLA** (Principle of Least Privileges), per il quale ogni componente deve avere tutti e solo i privilegi che gli permettono di svolgere la funzione per cui è stato scritto. Si pone inoltre una netta divisione tra *policy* (caratteristiche che definiscono il sistema) e *meccanismi* (come queste caratteristiche vengono implementate). Oggi la maggior parte dei SO è suddiviso in componenti, il modo in cui questi interagiscono definisce i vari tipi di architettura.

- **Struttura semplice**: SO semplice, organizzato con una minima struttura gerarchica, non suddiviso in moduli ben definiti. Un esempio sono MS-DOS o l'originale UNIX (in realtà più strutturato per la presenza del kernel). Per questo tipo di sistemi è difficile sia l'implmentazione che la manutenzione, d'altro canto si minimizza l'overhead grazie al basso numero di interfacce.

- **Struttura a livelli**: i componenti sono raggruppati in livelli, ogni livello utilizza solo le funzionalità del livello inferiore e fornisce servizi al livello superiore, specificando l'interfaccia. Implementazione e debugging risultano semplificati, ma ci sono difficoltà significative. La prima tra tutte è la definizione efficace di livelli, dovuta al fatto che le funzionalità dipendenti dall'architettura sono sparse tra i vari livelli e questo diminuisce la portabilità. Inoltre, si ha un overhead non trascurabile, dovuto al fatto che eventuali chiamate devono passare attraverso l'intera gerarchia del SO. Oggi si preferisce usare altre tecniche, o al limite usare questo approccio, ma minimizzare il numero di livelli.

- **Sistemi a microkernel**: il sistema è suddiviso in servizi `kernel` e `non kernel`, in modo tale che il kernel sia il più piccolo possibile. La comunicazione fra processo utente ed hardware (o altri processi utente) avviene tramite scambio di messaggi, che passano per il kernel. Questi SO sono facili da mantenere e da estendere, ma non esiste una chiara definizione per cosa dovrebbe stare nel kernel e cosa no. Kernel complessi e di grandi dimensioni diventano **monolitici**.

- **Sistemi a virtual machine**: sono sistemi pensati per poter eseguire sulla stessa macchina più SO contemporanemante, in modo indipendente. Componente chiave è l' `hypervisor`, che si pone fra HW e SO, emulando un hardware diverso da quello effettivo. Gli hypervisor di tipo 1 si distinguono poichè girano direttamente sull' HW e sopra di loro i sistemi operativi da eseguire, mentre l' hypervisor di tipo 2 gira come un normale processo su una macchina con un SO (e sopra di esso ancora altri SO).

- **Sistemi client-server**: Simile al microkernel nel modo in cui sposta ai livelli superiori le funzioni non essenziali. Il kernel si occupa della comunicazione tra client e server. Questa architettura è essenzialmente usata nei sistemi distribuiti.

Per quanto riguarda l'implementazione delle system call, attualmente si utilizzano soprattutto linguaggi ad alto livello come C/C++ per la rapidità di implementazione, portabilità e debugging. Tuttavia molte direttive, laddove più efficiente, sono scritte in assembler.

4 Gestione dei processi

Nei sistemi multiprogrammati un processo è definito come un *insieme di informazioni relativo ad un programma in esecuzione*. La gestione dei processi è importante per garantire multiprogrammazione e time sharing (il cui obiettivo è massimizzare l'utilizzo della CPU e dare ai processi l'illusione di avere pieno controllo sul sistema). Un processo è descritto dal suo **Process Control Block**(PCB), una struttura dati con una *tabella in memoria principale* che conserva le seguenti informazioni:

- **Stato**: in esecuzione, in attesa, etc.

- **Program Counter (PC)**: puntatore alla prossima istruzione da eseguire (in alcuni testi è chiamato Istruction Pointer)

- **Utilizzo dei Registri**: vengono conservati i valori dei registri CPU utilizzati dal processo

- **CPU scheduling**: priorità data al processo per l'utilizzo della CPU

- **Utilizzo della memoria**: settori della memoria principale (RAM) occupati dal processo
- **Accounting**: Process ID (PID) e percentuale di utilizzo della CPU
- **Utilizzo dell'I/0**: lista dei dispositivi e dei file allocati al processo

Tenere traccia dello stato di un processo e prevedere risposte del sistema in base a questo è fondamentale per poter massimizzare l'utilizzo della CPU. Gli stati un processo possono essere:

- **New**: il processo è appena stato avviato, deve essere ancora caricato in memoria dal `long-term scheduler`
- **Ready**: il processo è in memoria, sta aspettando che lo `short-term scheduler` gli lasci l'utilizzo della CPU
- **Running**: sta usando la CPU (ci può essere un solo processo in running alla volta)
- **Waiting**: il processo è in attesa di un evento, o di soddisfare una richiesta di utilizzo di un dispositivo I/O non disponibile al momento (la separazione di waiting dagli altri stati è un passo fondamentale per migliorare l'utilizzo della CPU)
- **Terminated**: è terminato

Per la gestione degli stati Waiting e Ready il sistema crea una serie di code, generalmente rappresentate come una `linked list di PCB`. Sono presenti una `ready list` per i processi attualmente in memoria e varie `device lists` (una per ogni dispositivo collegato) con i processi che hanno richiesto un dispositivo di I/O. Il controllo dei processi è compito degli scheduler.

Long-term Scheduler si occupa del caricamento dei processi dal disco alla memoria principale (decide chi ammettere in memoria). Controlla il `grado di multiprogrammazione`, ovvero il massimo numero di processi in memoria. Per rendere il sistema bilanciato, generalmente questo scheduler crea un equilibrio tra processi *CPU-bound* e *I/O-bound*. Viene invocato con intervalli nell'ordine dei secondi.

Short-term Scheduler Più veloce ed invocato con frequenze molto maggiori del precedente, si occupa del *dispatching*, ovvero il passaggio da Ready a Running. La parte più importante e costosa del dispatching è il cambio di contesto, `context switch`, ovvero il passaggio dall'esecuzione di un processo all'altro, momento in cui si salvano i dati del vecchio processo nel suo record PCB e si caricano i dati (dal relativo PCB) del nuovo processo da eseguire. Short-term scheduler che eseguono cambi di contesto troppo spesso rischiano di causare un grande overhead al sistema, degradando notevolmente le prestazioni, mentre effettuare cambi di contesto troppo poco spesso porta a tempi di risposta ed attesa eccessivamente lunghi.

Mid-term scheduler Presente solo in alcuni sistemi (in realtà in tutti i moderni), è deposto alla gestione dello `swapping`. Un processo, alla fine di un burst di CPU può essere infatti posto nel disco, in una coda diversa da quella di ready, per essere poi recuperato in seguito. Questo scheduler aggiunge un ulteriore controllo al grado di multiprogrammazione del sistema.

5 Operazioni sui processi

Per quanto riguarda la creazione dei processi, la maggior parte dei SO permette a processi di crearne di nuovi tramite apposite chiamate di sistema (in UNIX esiste solo la fork()). Si parla in questo caso di processi *padre* e processi *figlio* (in questi sistemi di stabilisce un vera e propria gerarchia dei processi, che in UNIX ha inizio con `init`, processo a radice dell'albero). A seconda delle politiche del SO si hanno diverse opzioni: il padre può aspettare che terminino i figli o continuare la propria esecuzione parallelamente ai figli (`esecuzione concorrente`), le risorse del figlio possono essere allocate dal SO o essere passate dal padre, il figlio può eseguire lo stesso codice del padre o codice diverso. Per quanto riguarda la terminazione di un processo, questa può avvenire, oltre che per il termine del proprio compito, in altri modi. Ad esempio, se il figlio sta consumando troppe risorse o se il suo compito non è più richiesto. Alcuni SO inoltre impediscono al figlio di sopravvivere al padre, pertanto terminano l'esecuzione di tutta la discendenza al termine del padre (`cascading termination`).

Quando un processo invoca la `exit()`, la sua memoria viene rilasciata, ma la tabella del processo rimane in memoria fino alla `wait()` del padre. In questo caso il processo figlio è detto *zombie*, poichè è ancora parzialmente in vita. Un processo è invece detto *orfano* (solo nei sistemi che non usano cascading termination) se il padre muore prima di lui. Nei sistemi UNIX questi processi vengona adottati da `init`, che periodicamente invoca la `wait()`.

6 Inter-Process Communication (IPC)

Nonostante esistano processi *indipendenti*, la maggior parte dei processi risulta *cooperante*, ovvero scambia in qualche modo informazioni con altri processi durante l'esecuzione. Le tecniche per garantire l'IPC sono generalmente due:

- **Memoria condivisa**: viene definita un'area di memoria in cui entrambi i processi posso leggere e scrivere dati. Generamente più veloce, in quanto non richiede l'intervento del kernel, tuttavia è necessario un meccanismo di sincronizzazione per gli accessi per evitare conflitti ed inconsistenze
- **Scambio di messaggi**: il kernel crea una coda di messaggi alla quale i processi accedono, che quindi comunicano senza condividere variabili. Sebbene richieda l'intervento del kernel, oggi sta diventando popolare e conveniente grazie alla tecnologia multicore. Può essere facilmente esteso ai sistemi distribuiti.

Generalemente i SO implementano entrambe queste tecniche, per poter poi usare l'una o l'altra a seconda delle scelte implementative.

Scambio di messaggi Esistono varie alternative per implementare questo meccanismo. In ogni caso è possibile scegliere tra una *lunghezza fissa* o *variabile* dei messaggi. Nel primo caso abbiamo una più semplice implementazione da parte del SO, ma una più complessa implementazione per i programmi applicativi, mentre è vero l'esatto contrario nel caso della lunghezza variabile. In ogni caso si ha la necessità di stabilire un *link logico* tra i processi che può essere:

- **Diretto**: se ogni processo invia direttamente il messaggio all'altro. La comunicazione può essere *simmetrica* se sia il mittente che il destinatario conoscono l'uno il nome dell'altro, o

asimmetrica se lo il mittente conosce in anticipo il nome. In entrambi i casi lo svantaggio principale sta nel fatto che un cambiamento nel nome del processo implica un cambiamento del codice.

- **Indiretto**: i processi non comunicano direttamente, ma utilizzano una `mailbox` condivisa su cui leggono e scrivono messaggi. A seconda dell'implementazione possono esistere più mailbox condivise tra due processi, oppure la stessa mailbox può essere usata da più di due processi. In questo caso è necessario, in caso di `receive()` multiple, decidere a chi vada il messaggio. Inoltre, la mailbox può avere un processo server, nel qual caso solo il server può ricevere e controllare la mailbox, mentre gli altri possono solo scrivere. Un'alternativa è che la mailbox sia gestita direttamente dal SO.

Oltre che per l'utilizzo di mailbox o meno, l'implementazione può differenziarsi nelle politiche di sincronizzazione di ricevitore e mittente. Si possono avere:

- Ricevitore:
  - Bloccante: una volta invocata la `receive()` si blocca fino alla ricezione del messaggio
  - Non Bloccante: continua la sua esecuzione anche dopo la `receive()`
- Mittente:
  - Bloccante: dopo la `send()` aspetta conferma dalla mailbox o dal receiver prima di continuare
  - Non Bloccante: non aspetta alcun ACK

Un'ultima classificazione riguarda la *bufferizzazione* dei messaggi, ovvero la dimensione della queue:

- Automatic Buffering: capacità 0, il mittente si blocca all'invio di ogni messaggio per attendere conferma dalla mailbox
- Bounded Buffering: il buffer ha una capacità limitata. il sender si blocca una volta raggiunta
- Unbounded Buffering: il buffer simula capacità illimitata, il sender non si blocca mai

Memoria Condivisa Generalemente, in un sistema a memoria condivisa, uno dei processi crea un *segmento condiviso*, con un nome, al quale può accedere l'altro processo. La sincronizzazione di lettura e scrittura è comunque gestita dai processi e non dal SO.

7 Threads

Un thread è definito come l'unità minima di utilizzo della CPU. All'interno di un processo, i thread condividono codice, dati e risorse, ma ognuno può avere stati e registri diversi. La suddivisione di un processo in più thread permette che parte di essi sia in esecuzione mentre parte del processo è bloccato per l'attesa di un evento o di I/O, ma questo è solo uno dei vantaggi. Dato che per definizione condividono codice e dati, la comunicazione e la condivisione di risorse tra thread è più semplice rispetto a quella tra processi; anche la creazione di thread è più veloce della creazione di processi. Inoltre, i processi multi-thread sono fatti per lavorare su sistemi multi processore.

Gli stati di un thread sono gli stessi di un normale processo, ma la loro correlazione dipende dall'implementazione. Ad esempio un thread in wait potrebbe bloccare o meno altri thread dello stesso processo o anche l'intero processo. Esistono essenzialmente due modi per implementare i thread in un SO, poi un terzo che è la combinazione dei primi due, di seguito riportati.

User-Level Thread Implementati tramite apposite librerie (`pthread.h`). Tutti i thread sono gestiti da un unico processo nel kernel, il che rende il sistema molto economico. Tuttavia, non si ha un vero e proprio parallelismo fra thread ed una chiamata di sistema bloccante causa il blocco di tutti i thread.

Kernel-Level Thread Ad ogni thread a livello utente corrisponde un singolo thread a livello kernel. Questo è costoso per il kernel, ma permette di avere parallelismo e risolve il problema delle chiamate bloccanti.

Approccio misto Ad N thread a livello utente corrispondono $M \leq N$ thread a kernel level. Si ha un buon parallelismo e risulta meno costoso del full kernel-level.

8 SO come processo

A seconda dell'implementazione, le parti del SO possono o meno essere considerate dei processi. Si hanno le seguenti alternative:

- **Kernel Separato**: il kernel esegue al di fuori dei processi utente, in uno spazio di memoria riservata ed in esecuzione privilegiata. Il concetto di processo è quindi applicabile solo ai processi utente. Questa modalità è tipica dei primi SO.
- **Kernel nei processi utente**: ogni processo ha un `kernel stack` condiviso che contiene funzionalità del SO eseguibili come codice protetto. Ciò velocizza le chimate di sistema, in quanto richiede solo un cambio di modalità (da user mode a kernel mode) anzichè un cambio di contesto (da processo utente a processo di SO).
- **Kernel come processo**: una parte del kernel (solitamente lo scheduler) esegue separatamente, per il resto, ogni parte del SO è un processo separato. Questo è molto vantaggioso nei sistemi multicore, che di solito riservano dei processori ad esclusivo utilizzo del SO.

9 Scheduling della CPU

Ottimizzare lo short-term scheduler è fondamentale per massimizzare l'utilizzo della CPU, pertanto sono state studiate molte tecniche e algoritmi per migliorare le prestazioni.

Dal punto di vista della CPU ogni processo può essere visto come una serie di *CPU burst* e *I/O burst* (dove per burst si intende un intervallo di utilizzo necessario al completamento della task). A partire da questi, si definisce una prima classificazione delle politiche di scheduling:

- **Pre-emptive**: prelazione in italiano; al processo è assegnato un timer, questo può essere rimesso in ready queue dallo scheduler anche prima di aver terminato l'attuale CPU burst, ovvero prima della scadenza del timer, rimuovendo forzosamente la CPU al processo.

- **Non pre-emptive**: un processo può essere rimesso in ready queue solo se richiede una risorsa di I/O, altrimenti deve prima terminare il suo attuale CPU burst.

Per valutare un algoritmo di scheduling si utilizzano apposite metriche di scheduling:

- **Utilizzo della CPU**: percentuale di utilizzo media della CPU
- **Throughput**: numero di processi completati nell'unità di tempo
- **Turnaround time**: tempo totale dall'inizio del processo al termine. Comprende il waiting time. $Turnaround = ExitTime - ArrivalTime$ o anche $Turnaround = WaitingTime + CPUburst$
- **Waiting time**: tempo speso dal processo nella ready queue
- **Response time**: tempo compreso tra l'arrivo del processo nella ready queue e la sua prima esecuzione, il primo dispatch

A seconda del sistema si può decidere di bilanciare tali metriche o focalizzarsi sulla minimizzazione/massimizzazione di una o più. Ad esempio, nei sistemi interattivi si preferisce minimizzare la varianza del tempo di risposta piuttosto che la media. Di seguito sono riportati i principali algoritmi di scheduling:

- **First Come First Served (FCFS)**: algoritmo di base, semplice da implementare e leggero, bassi tempi di risposta, lungo waiting time medio. Si utilizza una FIFO e si eseguono i processi in ordine di arrivo. Soggetto all' "effetto convoglio": l'esecuzione di molti processi corti è ritardata da processi più lunghi. Essendo inoltre non pre-emptive, è un algoritmo problematico per i sistemi a time sharing.
- **Shortest Job First (SJF)**: esegue i processi in ready queue partendo da quello con *prossimo CPU-burst minore*. L'algoritmo è ottimo per la minimizzazione di waiting time medio e quindi di turnaround (vedi algoritmi greedy), tuttavia è difficile stimare il burst di un processo. Per ovviare al problema si stima il prossimo burst di CPU tramite media espnenziale

  $$T_{n+1} = \alpha t_n + (1 - \alpha) T_n$$

  dove T_n è la media dei burst passati e t_n la durata dell'ultimo burst registrato, α è un valore compreso tra 0 e 1. In questo modo si ottiene una buona stima del prossimo burst di CPU. SJF non è pre-emptive, ne esiste anche un versione pre-emptive, chiamata **Shortest Remaining Time First**, la quale lavora allo stesso modo, ma può sostituire il processo in esecuzione con un uno appena arrivato se il burst di questo è più breve del tempo rimasto per la fine dell'esecuzione del processo correntemente in running.

Scheduling a priorità Con lo SJF è stato introdotto lo scheduling a priorità (inversamente proporzionale al burst di CPU). Più in generale, la priorità di un processo può essere stabilita sia *internamente* al SO (per limiti di tempo o memoria) che *esternamente* (come ad esempio dall'amministratore di sistema). L'implementazione base dello scheduling a priorità richiede una serie di code FCFS, ciascuna per ogni priorità specificata, questo comporta dei problemi tipici. Primo tra tutti la `starvation`: alcuni processi non saranno mai eseguiti (o comunque con tempi

di risposta troppo lunghi) poichè altri processi con priorità più alta li precedono costantemente. Per ovviare a tale problema si utilizza la tecnica di `aging`, tramite la quale la priorità viene periodicamente aggiornata e aumenta col passare del tempo in ready queue.

- **Highest Response Ratio First**: Algoritmo a priorità non pre-emptive, sviluppato come variante dello SJF. Per ogni processo la priorità viene calcolata come

$$P = 1 + \frac{WaitingTime}{BurstTime}$$

  ed è aggiornata al termine di ogni processo. Si supera con HRRN il favoritismo per i job corti, dato che la priorità dipende fortemente dal tempo di attesa. La stima del CPU burst è fatta con la media esponenziale come per SJF.

- **Round Robin (RR)**: Algortmo pre-emptive, che cerca di distribuire il tempo della CPU il più equamente possibile fra i processi. Viene definita un'unità (*quanto*), che è il tempo massimo di esecuzione di un processo prima che venga rimesso nella ready queue, gestita come una coda circolare. Circolarmente viene fornito un quanto di tempo a disposizione per i processi, inserendo ad inizio coda di solito i processi appena arrivati, per minimizzare i tempi di risposta. Nel caso un processo termini prima della fine del quanto a disposizione si verifica uno `yield` delle risorse al sistema, quindi lo sceduling del processo subito successivo. Il problema principale si basa sulla scelta del quanto di tempo: se troppo grande RR degenera in FCFS, se troppo piccolo l'overhead del cambio di contesto degrada le prestazioni. In generale con RR, rispetto a SJF, si ha un tempo di turnaround medio maggiore, ma un tempo di risposta minore.

Multilevel Queue Scheduling Anzichè utilizzare un unico algoritmo ed un'unica queue, è possibile adottare un algoritmo più generale che preveda N code, ognuna con una specifica priorità e algoritmo di scheduling. Diventa così necessario un algoritmo di scheduling fra le code. Nell'implementazione più semplice ciascuna coda ha priorità fissa (ovvero un processo viene inserito in una coda a seconda della sua priorità) e sono eseguite con FCFS. Si può ovviamente incorrere in starvation dei processi a bassa priorità. Si può invece decidere di basare lo scheduling tra le code su `time-slice`: ad ogni coda è assegna una *percentuale di utilizzo della CPU*, che è maggiore per le code a priorità alta.

In entrambe queste tecniche, la coda di un processo è fissa, non c'è modo per un processo di cambiare coda e di conseguenza priorità. Si ottiene con uno *scheduler multilivello con fallback* una struttura più flessibile. I processi possono cambiare code, salendo o scendendo di priorità, in base a specifici eventi. Ad esempio, un processo in una coda con RR che non finisce la sua esecuzione nel quanto di tempo assegnatogli, può essere degradato ad una coda con priorità minore e diverso scheduling. Tale strategia può essere inoltre usata per implementare l'`aging`, aumentanto il waiting time, il processo può essere spostato a code con priorità maggiore. Questo modello rimane tuttavia più complesso a causa dell'ulteriore livello di scheduling aggiunto.

Fair Share Scheduling (FFS) Le politiche di scheduling finora sono state orientate ai processi, mentre con FFS si schedulano utenti o gruppi di processi. Ad esempio si possono schedulare tutti i processi di una specifica applicazione. L'obiettivo è quello di fornire a ciascun gruppo di processi/utente la stessa percentuale di utilizzo di CPU.

Valutare un algoritmo di scheduling In quanto differenti sistemi richiedono differenti prestazioni, è opportuno applicare delle tecniche per valutare gli algoritmi di scheduling in differenti contesti:

- **Modello deterministico**: si definisce un `workflow` (insieme di processi con determinate caratteristiche) e lo si testa su carta con gli algoritmi descritti. Semplice e rapido, ma accurato solo per il workflow specifico.
- **Medello a reti di code**: si stimano matematicamente intensità e probabilità dei burst. Il modello è descritto come un rete di server, ognuno con una coda e rappresentante una CPU o un dispositivo di I/O. Relativamente semplice ed adattabile, spesso produce risultati poco realistici.
- **Simulazione**: si simula lo scheduler con un apposito software e dei grandi dataset. I risultati sono molto precisi, ma l'esecuzione è lenta ed è un modello molto costoso.
- **Implementazione su macchina**: il metodo più preciso e l'unico che permette di avere dei risultati sicuri. Si implementa l'algoritmo in un SO e si misurano le prestazioni con un reale carico di lavoro.

10 Sincronizzazione tra processi

Con l'avvento della multiprogramamzione e dei sistemi multicore è nata la necessità di gestire l'accesso concorrente dei processi (in particolare i thread) alle risorse condivise. Solitamente infatti, accedendo i thread ad un'area di memoria comune, si possono verificare errori nella consistenza (`consistency`) dei dati. Si parla in particolare di **race condition** laddove il contenuto di un'area di memoria condivisa o il valore letto da questa dipende dall'ordine di esecuzione dei processi. Ciò è chiaramente un problema in quanto fa sì il codice non sia più univoco nelle operazioni svolte e nei risultati generati a parità di input (matematicamente non sarebbe più una funzione). La sezione di codice contenente le istruzioni di accesso alle risorse che pongono il sistema in race condition è chiamata **sezione critica**. Una qualsiasi soluzione al problema della sezione critica deve avere i seguenti requisiti:

- **Mutua esclusione**: se un processo esegue nella sezione critica nessun altro processo deve eseguire in essa.
- **Progresso**: solo i processi che vogliono entrare nella sezione critica possono partecipare per decidere chi di loro può entrare. Tale decisione deve avvenire in tempo finito.
- **Attesa limitata**: esiste un numero massimo di volte consecutive per il quale un processo può aspettare prima di entrare in sezione critica (ovvero deve poter entrare prima o poi).

Il problema di sincronizzazione può essere risolto tramite software o hardware. Entrambe le soluzioni richiedono codice aggiuntivo, ma la soluzione con HW è considerata generalmente più efficiente, in quanto, nonostante necessiti di apposito HW, usa meno risorse. Inoltre, le architetture moderne non supportano più la sincronizzazione software, in quanto la naturale evoluzione dell'HW ha reso tale soluzione molto più conveniente. La soluzione HW è realizzata definendo particolari istruzione che sono `atomiche`, ovvero sono eseguite in un unico ciclo di istruzioni e non possono essere interrotte. Le uniche due istruzioni atomiche sono la `test and set` e la `swap`, che utilizzzate

correttamente garantiscono una soluzione la problema della sezione critica. La `test and set` prende un valore come argomento e imposta la variabile del valore a TRUE, restituendo poi il vecchio valore; la `swap` effettua solo uno scambio di valori tra due variabili. L'utilizzo di tali funzioni è tuttavia non banale per un programmatore ed inoltre le soluzioni che le implementano direttamente sono tutte basate su attesa attiva (**busy waiting**), la tecnica per cui un processo aspetta in un ciclo infinito il verificarsi di una condizione per poi passare alle istruzioni successive (considerato dannoso), con conseguente spreco di CPU. Per risolvere questi problemi sono stati introdotti i semafori, ovvero variabili intere (o binarie nel caso dei `mutex` o semafori binari) che i processi condividono e che sono modificabili solo attraverso le sopracitate operazioni atomiche. Si fa normalmente uso di due funzioni:

- **P** (o anche wait in pseudocodice) che decrementa il valore attuale del semaforo o, nel caso questo sia a 0, si blocca ed aspetta che diventi maggiore di 0, per poi decrementarlo.
- **V** (o anche signal o increase in pseudocodice) che incrementa il valore del semaforo

I semafori interi sono da utilizzare in particolare per bloccare o avviare un processo/thread in seguito ad un evento, come l'assenza di spazio rimasto in un buffer di scrittura o la presenza di nuovo spazio libero, mentre i semafori binari si specializzano maggiormente. Con un semaforo binario con valore iniziale 0 si possono sincronizzare le operazioni di due processi: basta che uno sia in `wait()` su un semaforo ed aspetti l'`increase()` dell'altro per essere avviato. Mentre con un semaforo binario con valore iniziale 1 si parla di `Mutex`, semafori dedicati a garantire la mutua esclusione. L'utilizzo dei mutex è sufficiente per garantire un soluzione al problema dell'accesso alla sezione critica: basta porre una `wait()` all'inzio dell'area di codice ed una `increase()` alla fine della stessa per garantire che nessun altro processo possa accedere alla risorse concorrentemente, eliminando la possibilità di race condition. **NB**: quella dei semafori non è considerata una soluzione con busy waiting perchè nel caso il processo debba bloccarsi ed aspettare, questo non resta in un ciclo di durata indefinita di test (while(mutex==0);), bensì nel codice della `wait()` è presente uno `yield()`, che restituisce subito tutte le risorse alla sistema.

Sebbene i semafori siano una soluzione funzionante ed efficiente, il loro utilizzo corretto è complesso ed il debug difficile. Allo scopo di semplificare la programmazione concorrente sono stati introdotti i monitor. Un **monitor** è un tipo di dato astratto che permette di specificare al prorpio interno un serie di funzioni, le quali gestiscono autonomamente la mutua esclusione. La sincronizzazione è possibile grazie a determinate variabili dette `condizioni`, sulle quali si possono invocare la signal e la wait. I monitor facilitano la programmazione, ma richiedono memoria condivisa e sono implementati, tuttora, da pochi linguaggi.

11 Deadlock

Si può definire logicamente un sistema come un insieme di *istanze di risorse di diverso tipo* che possono essere richieste, utilizzate e rilasciate da *gruppi di processi*. Si verifica un **deadlock** in un gruppo di processi quando ognuno di essi è in attesa di un evento (come il rilascio di una risorsa) che può essere causato solo da un processo dello stesso gruppo. Le **condizioni necessarie** per il verificarsi di un deadlock sono le seguenti:

1. **Mutua esclusione**: almeno una delle risorse deve essere non condivisibile. Se un processo richiede una di queste risorse ed essa è già utilizzata dovrà aspettare il rilascio ella stessa.

2. **Hold and Wait**: deve esserci almeno un processo nel gruppo che detiene una risorsa, ma è bloccato, in quanto necessita di almeno un'altra risorsa, detenuta da un altro processo.

3. **No pre-emption**: le risorse possono essere rilasciate solo dopo il loro utilizzo, la possessione di una risorsa non può essere rescissa forzatamente.

4. **Attesa circolare**: devono esistere N processi in attesa del rilascio delle risorse tali che P_0 attenda P_1, P_1 attenda P_2,...,P_{n-1} attenda P_n.

Di seguito si esaminano le principali soluzioni per la prevenzione, rilevazione e correzione dei deadlock.

Deadlock Prevention (prevenzione statica) Si evitano deadlock eliminando una delle quattro condizioni necessarie affinchè si verifichi.

1. **Mutua esclusione**: impossbile impedire totalmente la mutua esclusione per alcuni tipi di risorse

2. **Hold and Wait**: per evitare che un processo allochi risorse che non utilizza è possibile imporre che *l'allocazione di tutte le risorse avvenga contemporaneamente*. In alternativa si può imporre che un processo debba rilasciare tutte le risorse prima di poter richiederne altre. Questo metodo aumenta il rischio di starvation per i processi che necessitano di risorse molto richieste e generalemente diminuisce l'utilizzo delle risorse, drawback considerevole.

3. **No pre-emption**: si può imporre che un processo in waiting che richiede una risorsa non disponibile rilasci tutte le risorse in suo possesso. In alternativa, le sue risorse possono essere cedute ad un altro processo, se saranno sufficienti a farlo eseguire. Questo è però solo applicabile per risorse particolari il cui stato può essere salvato e ripristinato facilmente, come CPU o memoria.

4. **Attesa circolare**: si stabilisce un valore di priorità per ogni risorsa e si impone che un processo possa effettuare richieste in serie solo con priorità crescente. Il problema sta nello stabilire la priorità, in quanto questo dovrebbe rispettare l'ordine delle richieste.

In generale queste soluzioni sono molto costose e possono degradare notevolmente le prestazioni.

Deadlock Avoidance (prevenzione dinamica) La prevenzione statica ha lo svantaggio di diminuire l'utilizzo delle risorse quindi il throughput del sistema. Gli algoritmi di prevenzione dinamica tendono a migliorare le prestazioni, ma necessitano delle informazioni sulle richieste che ogni processo potrebbe fare. Per ogni processo è necessario sapere il numero massimo di instanze di ogni risorsa che potrebbe richiedere e le eventuali risorse già allocate. Questi algoritmi si basano sul mantenere uno **stato safe del sistema**. Un sistema si definisce in safe state se data una sequenza di processi (P_1,...,P_n) esiste un ordinamento tale per cui per ogni P_i le sue richieste possono essere soddisfatte utilizzando le risorse disponibili o quelle detenute da tutti i P_j con $j < i$, cioè aspettando che tutti i processi precedenti a P_i terminino e rilasciono le risorse detenute. Se tale sequenza non esiste il sistema è detto in *stato unsafe*, ovvero protebbe occorrere un deadlock. Tra gli algoritmi che si basano sul mantenere uno stato safe del sistema riportiamo i seguenti:

- **Algoritmo Resource Allocation Graph (RAG)** Funziona *solo con un'unica istanza per risorsa*. Si rappresentano processi e risorse come nodi di un grafo orientato. Gli archi assumono significati diversi a seconda del tipo: R→P indica che la risorsa R è allocata a P, P→R indica che il processo P richiede la risorsa R, esiste un terzo tipo di freccia P−− >R detta *claim edge* che indica che P potrebbe richiedere R. Durante l'esecuzione i claim edge si trasformano in allocazioni se la richiesta è soddisfatta. Richieste che causano un **ciclo nel grafo** non vengono accettate, in quanto causano uno stato unsafe.
- **Banker's Algorthm** Come il RAG, ha bisogno di sapere il numero massimo di istanze che ogni processo potrebbe richiedere per ogni risorsa, ma funziona con qualsiasi numero di istanze. Si basa sul mantenere delle strutture dati (matrici) con le informazioni sulle risorse richieste e allocate da ogni processo e sulle risorse di sistema disponibili. Ad ogni richiesta, si stabilisce se questa è possibile da soddisfare restando in uno stato safe, simulando l'assegnazione delle risorse. Il problema principale è la sua inefficienza, il semplice controllo dello stato safe/unsafe ha costo $O(n^2m)$, con n numero di processi e m tipi di risorsa.

Deadlock Detection and Recovery Entrambi i metodi di prevenzione visti tendono a ridurre la disponibilità delle risorse per la prevenzione di un problema che si presenta raramente. Si può optare invece per degli algoritmi di rilevamento del deadlock e di ripristino, che sebbene siano svantaggiati dal costo di ripristino, non richiedono conoscenze a priori sulle risorse e le richieste. Di seguito due algoritmi di rilevamento, ciascuno derivato da uno degli algoritmi di avoidance.

- **Algoritmo Wait-for** Basato sul RAG, funziona solo con un'unica istanza per risorsa. Viene mantenuto in memoria un `grafo wait-for`, struttura dati creata appositamente per questa funzione, ottenuta eliminando i nodi risorsa dal RAG e collassando i vertici. Nel grafo così ottenuto un arco $P_i \to P_j$ indica che P_i è in attesa di una risorsa detenuta da P_j. Questo si ha solo se nel RAG era presenta una risorsa R_x tale che $P_i \to R_x$ e $R_x \to P_j$. Periodicamente viene invocato un algoritmo $O(n^2)$ per stabilire l'esistenza di un ciclo, che indica un deadlock in corso.
- **Banker's algorithm** Utilizza una struttura dati molto simile all'omonimo algoritmo di deadlock avoidance, funziona quindi con qualsiasi numero di istanze. Si basa sul verificare in tempo $O(n^2m)$ che una serie di richieste non causi un deadlock. Rispetto all'algoritmo originale non richiede di imporre un massimo sulle richieste.

La frequenza di invocazione di un algoritmo di rilevamento dipende dalla frequenza con cui si verificano deadlock nel sistema. Generalmente vengono invocati ogni N cicli, o quando l'utilizzo della CPU scende sotto una certa soglia (sintomo di deadlock). Invocarli ad ogni richiesta di risorse è la scelta più sicura, ma anche la più costosa. Per quanto riguarda la recovery, generalmente si opta per la gestione automatica da parte del sistema, che può intervenire in due modi: uccidendo tutti processi coinvolti o riprendendosi le risorse contese. Nel primo caso, l'uccisione di tutti i processi, si ha un elevato costo per il sistema e, nel caso i processi stessero interagendo con periferiche, si rischiano incosistenze. Spesso si sceglie un'uccisione selettiva, ma in tal caso è necessaria una qualche politica per stabilire un ordine di priorità per la sequenza di uccisione. Inoltre, dopo ogni uccisione, è necessario invocare nuovamente l'algoritmo di rilevamento per stabilire se c'è ancora un deadlock. Se si sceglie la prelazione delle risorse, si deve anche in questo caso definire una politica per stabilire a chi vengono tolte le risorse ed in quale ordine. Le operazioni di `rollback`, ovvero di rispristino ad uno stato passato per un processo, sono particolarmente costose ed esiste la possibilità di starvation.

In conclusione, nessuna delle tecniche esposte è efficace in tutte le situazioni. Pertanto, generalmente, si opta per un *partizionamento delle risorse in classi*, ad ognuna delle quali si applicano le tecniche più efficaci. Si deve però riconoscere come qualsiasi tecnica di prevenzione, sia statica che dinamica, e qualsiasi di detection-recovery, sia costosa per il sistema e possa degradare notevolmente le prestazioni, specie se si decide di avere il grado di sicurezza massimo, ovvero controllare lo stato del sistema ad ogni richiesta/allocazione. Per evitare tale onere, poichè i deadlock si verificano con una bassissima frequenza, per la gestione si sceglie spesso di usare l'*algoritmo dello struzzo*, che di fronte ad un deadlock, spaventato, mette la testa sottoterra e non fa nulla.

12 Gestione della memoria

Un compito fondamentale di ogni SO è la gestione della memoria, in particolare in seguito all'introduzione della multiprogrammazione è diventata necessaria la definizione di **spazio di indirizzamento**, ovvero l'insieme degli spazi di memoria indirizzabili da un processo, e la sua protezione, realizzata con limiti superiori non oltrepassabili, poichè se non gestita può portare al sovrapporsi degli spazi di indirizzamento tra due o più processi. Le tecniche di gestione utilizzabili sono generalmente dipendenti dall'hardware, in quanto questo implementa apposite istruzioni e algoritmi di gestione. Dal punto di vista di un processo, la maggior parte degli accessi in memoria avviene tramite *decodifica degli indirizzi*, ovvero per la traduzione di un **indirizzo logico** (come una variabile) in un **indirizzo fisico**, che corrisponde ad una vera e propria locazione in memoria. Tale processo di traduzione è definito **binding** e può avvenire in differenti modi e tempi:

- **a tempo di compilazione**: deve essere già nota la locazione del programma, pertanto gli indirizzi fisici sono scritti direttamente nel codice. Il binding avviene a compile time ed è definito **statico** in quanto gli indirizzi fisici e logici sono uguali e se questi cambiano il codice deve essere ricompilato.

- **a tempo di caricamento**: nel codice si specifica un indirizzo assoluto come primo indirizzo di allocazione e gli altri sono scritti in relazione a quest'ultimo (*offset*). Anche questo tipo di binding è statico, ma il codice va ricompilato solo se cambia l'indirizzo di riferimento.

- **a tempo di esecuzione**: non sono presenti indirizzi assoluti (fisici) nel codice, perchè la locazione in memoria potrebbe cambiare anche durante l'esecuzione. Questo tipo di binding è definito **dinamico** ed è utilizzato oggi nella maggior parte dei SO, ma richiede HW apposito per essere efficiente, ovvero la **Memory Management Unit (MMU)**.

Dynamic linking and loading Le operazioni di linking e di loading possono essere effettuate in modo *statico* o *dinamico*.

- Linking: associazione a librerie esterne, il metodo statico prevede di creare una copia del codice delle librerie usate. Nel metodo dinamico il codice è invece sostituito da **stub**, ovvero riferimenti alla specifica libreria. Se questa viene richiamata, lo stub carica in memoria la libreria. Si ha la possibilità di avere librerie condivise se ci sono più processi che utilizzano le stesse. In questo modo si ha un generale risparmio di codice, d'altro canto, in quanto questo è condiviso, si ha la necessità di supervisione da parte del SO.

- Loading: associazione a librerie di sistema, con il metodo statico tutto il codice delle routine è caricato insieme al programma, creando codice inutile, come per il linking statico. Con il

metodo dinamico, il caricamento è posticipato al momento del primo utilizzo del metodo. Risulta così molto efficiente, specie se si fa uso di codice usato molto raramente, come le routine di errore.

In generale, in un sistema multiprogrammato non è possibile conoscere in anticipo la posizione in memoria di un processo e l'esigenza di avere lo swap impedisce il binding a tempo di caricamento. Pertanto, nei sistemi general-purpose si usa soltanto il binding a tempo di esecuzione.

Protezione della memoria Per assicurare che non ci siano accessi illegali alla memoria, ogni processo è associato ad un **registro base** ed ad un **registro limite**, contenuti nel PCB. Il registro base contiene il più basso valore indirizzabile da un processo, mentre il registro limite contiene il totale di indirizzi specificabili dal programma, in modo tale che (reg. base + reg. limite) diano il più piccolo valore non indirizzabile dal programma, il limite superiore dello spazio degli indirizzi. Pertanto un indirizzo logico è sempre minore del registro di base e gli indirizzi fisici devono essere compresi tra reg. di base e (reg. di base + reg. limite). Nel caso di binding dinamico, o a run time, basta cambiare il valore del registro di base, tutti gli indirizzi fisici saranno calcolati durante le operazioni con uno spiazzamento sugli indirizzi logici, in particolare $ind.Fisico = ind.Logico + reg.Base$.

13 Allocazione della memoria

Esistono differenti politiche per il caricamento dei processi in memoria e per riservare loro lo spazio. Gli obiettivi principali di questo ambito di gestione sono l'aumentare l'utilizzo della memoria, allocando più processi possibili ed aumentando il grado di multiprogrammazione.

Allocazione contigua La memoria di un processo è formata da un range di indirizzi continuo. Tutta la memoria è suddivisa in partizioni, che possono essere di dimensione fissa o variabile.

- **Partizioni Fisse** Il SO crea appositamente delle partizioni di dimensione fissa e predeterminata all'avvio. Si può scegliere se implementare una queue per ciascuna partizione, scelta poco flessibile, oppure un'unica queue, con la quale le partizioni vengono assegnate con differenti algoritmi, alcuni dei quali: *Best-fit, Worst-fit, First-fit.* Questa implementazione limita il grado di multiprogrammazione e produce grande frammentazione interna ed esterna.

- **Partizioni variabili** Il SO crea appositamente delle partizioni per i processi che devono essere caricati in memoria e mantiene una *lista delle partizioni libere e occupate.* Quando una di esse si libera viene aggiunta alla lista della memoria libera ed eventuali blocchi contigui vengono fusi. Questo metodo non produce frammentazione interna (in realtà c'è sempre un po' frammentazione interna, ma è minimizzata, non si può fare meglio), in quanto ogni partizione ha dimensione pari a quella richiesta dal processo. Tuttavia in media aumenta la frammentazione esterna, in quanto si producono molte partizioni di piccola dimensione.

- **Buddy System** Compromesso fra i modelli precedenti, utilizza blocchi di dimensione 2^k per qualche k tc. $lim.inferiore \leq k \leq log_2(dim.memoria)$. Alla richiesta di un blocco, si divide in due il blocco principale fino ad ottenere la Best-fit per il blocco richiesto. Blocchi liberi contigui di uguale dimensione vengono fusi in un unico blocco di dimensione doppia. La frammentazione interna è ampiamente ridotta rispetto all'allocazione a partizioni fisse,

ma persiste, inoltre, riduce il costo compattazione della memoria, in quanto i blocchi liberi sono sempre contingui.

Paginazione Questa tecnica evita del tutto la frammentazione esterna (come le part. variabili) in quanto permette l'allocazione dei processi non contigua. Questo è possibile con la divisione della memoria fisica in **frame** e la memoria logica in **pagine** di uguale dimensione. Quando un processo richiede memoria gli vengono assegnate delle pagine, per cui la frammentazione interna è limitata al più ad una pagina per processo. Le corrispondenze pagine-frame sono registrate in un'apposita **page table**. Quando la CPU legge un indirizzo logico, la MMU effettua la traduzione grazie alla tabella. Tale metodo ha un'efficienza accettabile solo se la tabella è ad accesso rapido, altrimenti il costo di accesso risulta proibitivo. Le alternative per l'implementazione sono:

- **Utilizzo di registri**: le entry del processo attuale sono memorizzate in appositi registri della CPU. La velocità di accesso è massima, ma questa soluzione causa parecchio overhead nel context switch. Il problema principale, tuttavia, è lo scarso numero di registri disponibili e quindi delle entry memorizzabili.

- **Tabella in memoria**: questo metodo prevede di tenere in memoria la tabella di ogni processo in esecuzione. La locazione della stessa è memorizzata in un apposito registro detto **Page Table Base Register (PTBR)**. Questo riduce di molto l'overhead in quanto durante il context switch la CPU dovrà solo cambiare il valore del PTBR. Il problema principale di questo approccio è il tempo di accesso alla memoria, in quanto ogni traduzione richiede un accesso per leggere la tabella ed un accesso per l'indirizzo fisico. Questo ritardo è intollerabile senza ulteriori modifiche.

  Per risolvere questo problema si utilizza una cache apposita detta **Translation Look-aside Buffer (TLB)**, che consiste in una serie di coppie chiave-valore, contenente le associazioni pagina-frame. La realizzazione HW di tale cache permette il confronto parallelo di tutte le entry e risulta pertanto estremamente veloce. Se un'associazione non viene trovata nel TLB, si procede ad un accesso in memoria e si copia l'associazione nella TLB. Sebbene questa possa contenere un numero limitato di entry, la sua hit ratio è molto alta, riducendo così il tempo di accesso sensibilmente. Generalmente, il TLB è smontata ad ogni context switch, per evitare accessi non autorizzati ad indirizzi attivi in un altro processo. In alternativa, è possibile memorizzare in ogni tupla un ID del processo, per evitare di smontare ogni volta la cache.

  Per garantire sicurezza è possibile aggiungere *bit di flag* alle entry della tabella delle pagine:

  - **bit di validità** alle volte (spesso) un processo non usa tutti gli indirizzi logici del proprio spazio e pertanto alcune pagine rimangono vuote. Questo bit segnala tali pagine, i tentativi di accedere ad esse sono bloccati.
  - **bit di accesso** utilizzabili per marcare una pagina come eseguibile, read-only, read/write, etc. In generale per definire i diritti di accesso.

Pagine Condivise Possibilità interessante che si apre con la paginazione è la condivisione di pagine fra più processi. Il requisito è che tali pagine contengano solo codice read-only. Questo meccanismo può essere usato per esempio quando si esegue lo stesso programma più volte con dati diversi.

Paginazione Gerarchica L'evoluzione dell'hardware ha portato ad un aumento esponenziale dello spazio di indirizzamento logico e di conseguenza anche della dimensione della tabella delle pagine (vedi dopo memoria virtuale). Per limitare la dimensione e risparmiare spazio in memoria, è possibile adottare una struttura gerarchica per la tabella, che equivale a crare due o più livelli di tabelle delle pagine. Si deve tenere conto che ogni livello aggiuntivo richiede un ulteriore accesso in memoria, quindi si limita di solito questo tipo di struttura ad un massimo di due o tre livelli (costo troppo elevato in caso di cache miss nel TLB).

Tabella delle pagine invertita Altra tecnica pensata per far fronte ai problemi della gestione di page table di grandi dimensioni. Anzichè avere una tabella per ogni singolo processo, si ha un'unica tabella di sistema, ordinata per numero di frame, dove le associazioni sono invertite: non sono più pagina $\rightarrow$ frame, ma frame $\rightarrow$ pagina. Diventa necessario memorizzare anche il PID del processo che detiene il frame. In generale, con questa tecnica si ha unn risparmio di memoria, bilanciato da un incremento del tempo di look-up. Per risolvere questo problema generalmente si opta per usare una *tabella di hash* indicizzata sulle coppie $< pid, pageIndex >$che rende il tempo di look-up $O(1)$. L'efficacia di questo metodo dipende principalmente dall'implementazione della tabella ed in particolare dalla gestione delle collissioni.

Segmentazione La segmentazione permette la divisione dello spazio di indirizzamento di un processo in segmenti di dimensione variabile. Solitamente questi rispecchiano la visione del processo che ha l'utente, pertanto si avranno, per esempio, un segmento per il codice, uno per i dati, etc. Similmente alla paginazione si ha una tabella dei segmenti costituita da coppie $< base, limite >$, rappresentanti rispettivamente l'indirizzo di partenza del segmento e la dimensione dello stesso. Gli indirizzi logici sono formati dal numero di segmento (posizione della entry nella tabella) e dall'offset interno allo stesso. Questo rende immediato il controllo di legalità dell'indirizzo. Per come è strutturata, la segmentazione supporta naturalmente la protezione e la condivisione di codice. Uno svantaggio sta nel fatto che all'allocazione dei segmenti è deposto il SO e si può incorrere in frammentazione esterna.

Segmentazione paginata Una tecnica mista che consiste nel paginare segmenti di memoria. In questo modo si ha una tabella delle pagine interna ad ogni segmento e la tabella dei segmenti conterrà gli indirizzi base a tali tabelle. Volendo si può anche adottare la paginazione gerarchica. In generale si elimina il problema dell'allocazione e della paginazione esterna.

14 Memoria Virtuale

Le tecniche viste in memoria prevedono che tutti i dati di un processo siano caricati in memoria al momento della sua esecuzione. Con l'evoluzione del software, i suoi picchi di *bloatware* e la naturale espasione delle funzionalità offerte, caricare tutto in memoria, con software di grandi dimensioni, non sempre è possibile, ma soprattutto non è sempre necessario. Con la memoria

virtuale è possibile considerare parte del disco come memoria primaria e caricare in memoria solo le parti del programma necessarie per l'esecuzione attuale. Sebbene generalmente complessa da implementare, la memoria virtuale permette di caricare più processi in memoria, aumentando quindi il throughput, ed inoltre permette facilmente la condivisione di risorse fra processi. Esistono varie tecniche per realizzare la memoria virtuale, ma si vedrà come per la paginazione la memoria virtuale sembri quasi una naturale evoluzione.

Paginazione su richiesta (on demand) Se un sistema utilizza la paginazione, è possibile usare questa tecnica *senza ulteriore hardware di gestione*. All'avvio del processo una parte delle pagine è *caricata in memoria e le restanti su disco*, utilizzando nella page table un *bit di validità* per segnalare le pagine in memoria. Ad ogni istruzione, se la pagina richiesta è presente in memoria si va avanti normalmente, altrimenti si verifica un **page fault**: un interrupt (trap) del SO, il quale, stabilita la legalità dell'istruzione, cerca un frame vuoto o dal quale rimuovere una pagina ed effettua lo *swap dal disco della pagina richiesta*, quindi modifica il relativo flag di validità e ripristina il processo. Ovviamente, nel caso una pagina sia stata rimossa (pagina *vittima*), anche questa dovrà essere swappata sul disco, aumentando il tempo di recupero per il page fault. Si possono migliorare le prestazioni, con impatto moderato, evitando di effettuare lo *swap di pagine non modificate* (**dirty bit** o modified bit), ma è più che mai necessario tenere il tasso di page fault il più basso possibile.

Page Replacement Algorithms Il tasso di page fault, che deve essere tenuto basso per non inficiare le prestazioni, dipende molto da come viene scelta la pagina da rimpiazzare. Per effettuare tale scelta si utilizzano degli appositi algoritmi, i quali vengono valutati in base al numero di page fault derivati da una stringa di riferimenti in memoria. Nella valutazione il numero di frame rimane fisso e il *tasso di page fault è inversamente proporzionale al numero di frame* (più frame → più spazio per tutti).

- **FIFO** La prima pagina caricata è la prima ad essere rimpiazzata, segue la nuova testa della coda. Algoritmo terribile, non tiene conto dell'importanza di una pagina e tende ad aumentare i page fault. Soffre di quella che è definita come **anomalia di Belady**, per la quale il tasso di page fault può aumentare con l'aumentare del numero di frame. Si studia per non ripetere gli stessi errori.

- **Algoritmo Ottimo** Si dimostra che questo algoritmo garantisce il minimo tasso di page fault. Rimpiazza la pagina che non verrà utilizzata per più tempo. In quanto presuppone la conoscenza futura dei riferimenti in memoria è pressochè non implementabile se non con approssimazioni, quindi utilizzato principalmente come metro di confronto.

- **Least Recently Used (LRU)** Si tratta di un'approssimazione dell'algoritmo ottimo. Come spiega il nome, viene rimpiazzata la pagina a cui non si accede da più tempo. Risulta molto migliore della FIFO, tuttavia in ogni sua implementazione necessita di un adeguato supporto hardware, altrimenti il costo dei continui interrupt per l'aggiornamento dei dati sull'ultimo accesso risulterebbe proibitivo. Fra le possibili impllementazioni ci sono:

  - **Contatori** si associa ad ogni pagina un numero (di solito il clock time) che indica l'ultimo accesso, che va aggiornato ad ogni riferimento. Viene rimossa la pagina con clocktime più basso, che però va cercata nella lista.

– **Stack** le pagine sono tenute in uno stack (solitamente una lista concatenata). Quando si fa riferimento ad una pagina, questa viene estratta dalla sua posizione e messa in cima. Eliminare una pagina ha costo $O(1)$, in quanto la pagina designata è sempre in fondo, ma aggiornare il riferimento, rimuovendo un elemento dal centro, richiede una ricerca $O(n)$.

Entrambe queste tecniche implementano deterministicamente LRU, ma sono costose in termini di HW. Esistono implementazioni alternative meno costose, che sono tuttavia a loro volta delle approssimazioni:

– **Bit di Reference** ad ogni pagina è associato un bit a 0, settato ad 1 una volta che la pagina è stata referenziata. Non si sa in che ordine le pagine sono state referenziate, ma sicuramente si sa quali pagine non sono state accedute e sono per questo vittime preferibili. Un'alternativa simile è usare più bit di reference in un registro a scorrimento aggiornato periodicamente. In questo modo, trattando il valore del registro come un intero, sarà sufficiente scegliere la pagina LRU con valore minore.

– **Second chance** Utilizzando il bit di riferimento, si scorre la linked list delle pagine. Tutte le pagine con bit a 1 vengono messe a 0 (seconda chance). Se si incontra una pagina con bit a 0 (dopo al massimo una scansione completa) questa è scelta come vittima. In alternativa, si memorizza anche un bit di modifica, crando di fatto due classi di pagine di cui si sceglierà quella di classe minore come vittima (replacing unmodified page >> replacing modified page).

– **Tecniche di conteggio** Si mantiene un contatore con il numero di riferimenti per ogni pagina. A seconda dell'algoritmo si elimina quella con più (MFU) o meno (LFU) riferimenti. Entrambe queste soluzioni funzionano male e sono costose (c'è comunque una scansione $O(n)$).

Allocazione di frame Per rendere efficiente la memoria virtuale è fondamentale scegliere un buon algoritmo di allocazione, che decida come ed a chi distribuire i frame ai processi in esecuzione. Uno dei vincoli fondamentali è che il minimo numero di frame assegnati ad un processo corrisponde al massimo numero di indirizzi specificabile in un'istruzione. Se minore, si potrebbe verificare una situazione di costante page fault per il processo. Una volta specificato questo vincolo, è possibile classificare gli algoritmi in due famiglie, a seconda della politica di allocazione dei frame: rimpiazzamento **locale** o **globale**. Nel primo caso i frame vittima possono essere scelti solo tra i frame allocati al processo stesso (sceglie la vittima tra i frame che ha); in questo modo il numero di frame per processo rimane fisso. Se si parla di rimpiazzamento globale, la vittima può essere scelta tra tutti i frame in memoria, quindi anche degli altri processi. In questo modo è più difficile tracciare il tasso di page fault, ma in generale questo metodo dinamico aumenta il throughput ed è per questo preferito.

Un problema da evitare è l'occorrenza del **thrashing**, fenomeno che causa un improvviso declino del throughput, dovuto all'overhead nella gestione dei page fault. Quando un processo non ha abbastanza frame, genera più page fault (inversamente proporzionale), decrementando quindi il throughput. Il SO, rilevando una diminuzione del throughput, aumenta il grado di multiprogrammazione, ma così facendo diminuisce ulteriormente il numero di frame disponibili, creando così un circolo vizioso definito appunto thrashing.

Oltre che per tipologia di rimpiazzamento, si possono classificare le tecniche di allocazione in base al numero di frame per processo:

- **Allocazione fissa** Il numero di frame per processo rimane fisso per tutta la durata dell'esecuzione. Si può scegliere una suddivisione dei frame in parti uguali, proporzionale alla dimensione del processo o dipendente dalla priorità. Risulta semplice da gestire, ma poco flessibile in quanto si adatta poco alle esigenze spesso variabili di memoria del processi.

- **Allocazione variabile** Il numero di frame associati ad ogni processo varia nel corso del tempo. Per l'implementazione esistono essenzialmente due approcci:

  - **Stima del working set** Si definisce **working set** la quantità di memoria che un processo richiede per svolgere le proprie funzioni in un dato istante t. Il working set dipende dalla località del programma (variabili/oggetti/strutture dati allocate) ed è per questo definito in funzione del tempo. Per la sua stima, un metodo semplice quanto veloce consiste del considerare un dato intervallo di riferimenti Δ: tutte le pagine che compaiono nell'intervallo fanno parte del working set ed il numero ne costituisce la cardinalità (dimensione). In base alla stima del working set varia il numero di frame allocati. Un'approsimazione di tale metodo consiste nel registrare i reference bit di ogni pagina ogni n riferimenti, anche mantenendo una storia. In questo modo in caso di page fault tutte le pagine con reference bit a 1 saranno considerate parte del working set attuale. Sebbene abbastanza costoso, tale metodo aumenta il throughput, anche grazie al fatto che i processi con working set troppo piccoli vengono rimandati per far posto ad altri, evitando così il thrashing.
  - **Page Fault Frequency** Partendo dal presupposto che sia necessario evitare il thrashing, questo metodo tiene traccia dei page fault di ogni processo, cercando di mantenerne il *ratio* tra un limite superiore ed uno inferiore. Il superamento del threshold superiore è sintomatico della necessità di più frame da parte di un processo, che gli vengono quindi allocati. Al contrario, si possono togliere frame a quei processi che oltrepassano il limite inferiore. Come per il modello con stima del working set, i processi con troppi pochi frame vengono rimandati. Questo metodo, presupponendo una corretta stima dei limiti, risulta essere molto preciso.

Un'ulteriore considerazione riguarda la dimensione delle pagine, generalemente fissata a seconda della tecnologia hardware. Pagine di piccola dimensione offrono maggiore *granularità nei trasferimenti* e *minore frammentazione*, ma implicano una *maggiore dimensione della tabella delle pagine* ed un *costo I/O* non ammortizzato. Le pagine di grandi dimensioni hanno, d'altro canto, caratteristiche speculari.

Infine, molti SO permettono il **page locking**, ovvero la possibiltà di tenere sempre in memoria pagine non rimpiazzabili. Generalmente, tali frame contengono moduli del kernel o buffer di I/O.

15 Memoria Secondaria

Vi sono differenti tecnologie utilizzabili per implementare la memoria secondaria: nastri magnetici, dischi magnetici e memorie a stato solido. Generalmente un dispositivo di I/O si interfaccia col

resto del sistema tramite un controllore che gestisce le richieste del SO ed effettua i trasferimenti necessari.

Struttura dei dischi Ogni disco magnetico è suddiviso in un insieme di parti fisiche.

- track zona circolare di un disco, quelle interne sono più piccole e quelle esterne più grandi.
- cilindri insieme delle tracce di ogni disco, sovrapposte
- settori unità che compone una track, è la più piccola, di solito è di 512 byte.

La lettura/scrittura di un settore necessita anche di un tempo di ricerca dello stesso nel disco. Il *tempo di accesso al disco* è descritto come $t_{accesso} = SeekTime + Latenza + TransferTime$, dove:

- **Seek time** è il tempo necessario per spostare la testina di lettura sulla track giusta.
- **Latenza** è il tempo necessario, una volta trovata la track giusta, necessario per arrivare al settore desiderato.
- **Transfer time** è il tempo di lettura/scrittura del settore

Con la tecnologia attuale per i valori ottenuti il *seek time risulta dominante*. Questo può essere ridotto utilizzando un buon algoritmo di disk scheduling. Il transfer time effettivo dipende invece dai buffer e dai bus messi a disposizione, a livello di software non si può fare nulla per migliorarlo. La vista logica del SO su disco è formata da un *array monodimensionale di gruppi di settori*, detti **cluster**, che costituiscono la minima unità di lettura/scrittura.

Disk Scheduling Per misurare le prestazioni dei vari algoritmi si usa come input una stringa di numeri che rappresenta la lista di richieste d'accesso ad un settore in uno specifico disco. L'algorimo migliore dovrebbe *minimizzare la distanza percorsa dalla testina*, minimizzando il seek time medio e quindi anche i tempi di accesso. Di seguito i maggiori algorimi di accesso al disco.

- **FCFS** il solito first come first served, non offre nessun particolare vantaggio, se non la facilità di implementazione. Da qui si può solo migliorare la riduzione del seek time.
- **Shortest Seek Time First (SSTF)** come lo SJF per lo scheduling di CPU, viene data la priorità alla richiesta in queue con seek time minore. Minimizza il tempo di attesa in queue medio (alg. greedy), ma aumenta il rischio di starvation per alcune richieste.
- **SCAN e C(ircular)-SCAN** nella SCAN la testina di muove sempre in una direzione, da inizio a fine traccia, eseguendo di volta in volta le richieste incontrate sul cammino, qualsiasi sia il loro ordine o posizione nella queue. Arrivata a fine traccia, la testina ripete l'operazione, ma nella direzione opposta, da fine ad inzio. Nella C-SCAN le operazioni sono le stesse, ma la traccia non è vista come un vettore, ma come una coda circolare: arrivata a fine traccia la testina ripartirà dall'inizio della traccia, aggiungendo quindi il costo necessario per riportare la testina dalla n-esima traccia alla 0.
- **LOOK e C(ircular)-LOOK** molto simile alla SCAN, la testina cambia direzione però non quando finisce la traccia, bensì quando non ci sono più richieste di accessi nella direzione di percorrenza corrente. Se la testina sta scandendo la traccia crescentemente dal settore 0,

arrivata al settore n, verificato che non ci sono più accessi da n+1 in poi, inverte il senso di percorrenza che diventa decrescente. Anche la variante C-LOOK vede la traccia come una coda circolare ed invece che cambiare senso di percorrenza, ricomincia dall'inizio della traccia.

In conclusione gli algoritmi più utilizzati sono SSTF e LOOK, ma ci sno molti fattori che possono influenzare l'analisi, ad esempio la dimensione degli accessi può ammortizzare il tempo di seek ed il numero di richieste di I/O dipende molto dall'organizzazione del file system. Molti SO implementano il disk scheduling come un modulo a parte, consentendo diverse scelte di algoritmo.

Formattazione del disco La formattazione è processo che crea su un disco le strutture e le suddivisioni in settori necessarie per l'utilizzo da parte del SO, tale operazione si suddivide in formattazione fisica e logica. La prima effettua l'iniziale suddivisione in settori, ognuno dei quali è provvisto, oltre che del *campo informativo*, di un **header** contenente un **numero di settore** ed un **trailer** con un **Error Connecting Code (ECC)**, che fornisce un sistema limitato di correzione degli errori. Generalmente tale formattazione è effettuata in fabbrica. Per quanto riguarda la formattazione logica, questa prevede la suddivisione del disco in **partizioni** e la definizione del file system montato su ciascuna di esse, con le relative strutture di supporto. Questo processo definisce anche eventuali partizioni speciali, come quella dedicata per i file di swap, ed i **boot blocks** del disco, le zone di memoria del **bootstrap**, contenenti le routine di avvio del SO.

Gestione dei Bad Block Un bad block è un settore del disco in cui il ECC ha rilevato un errore che non può essere corretto, il che rende di fatto inutilizzabile il blocco. Esistono principalmente due tecniche di gestione dei bad block:

- **offline** tipico dei primi controllore IDE, permette la *marcatura* dei bad block solo durante la fase di formattazione. La marcatura impedisce al file system di allocare tali blocchi.
- **online** metodo più sofisticato. Viene creata, durante la formattazione fisica, una lista di bad block da aggiornare durante l'utilizzo del disco. Vengono inoltre mantenuti dei blocchi non allocati (*sector sparing*). In caso di rilevamento di un bad block, il controllore mapperà il bad block su di un blocco libero. Per mantenere efficiente l'algoritmo di disk scheduling, si cerca di mappare il bad block su di un blocco libero nello stesso cilindro.

Gestione dello Swap In quanto la memoria virtuale può essere onerosa, è importante scegliere quale parte del disco sarà dedicata allo swap, in quanto questo influenza la velocità di I/O. Le soluzioni sono principalmente due: usare il file system o una partizione dedicata. Nella prima, parte del file system viene utilizzata come spazio di swap; è una soluzione semplice da implementare, ma anche pesante, in quanto si aggiunge il costo dell'attraversamento delle strutture di file system. Nel caso si scelga di implementare una partizione dedicata, si deve tener conto che questa può essere allocata solo al momento della formattazione logica, dato che deve essere definita come una vera e propria partizione. Questa soluzione è molto più efficiente, nonchè la più utilizzata. In ogni caso, alcuni SO permettono l'utilizzo di entrambe le tecniche in caso di necessità.

16 Reduntant Array of Inexpensive Disks (RAID)

Un sistema RAID è un configurazione di un insieme di dischi fisici che un SO vede come un unico supporto di memoria. Esistono differenti configurazioni di seguito elencate, ciascuna con vantaggi

e svantaggi caratteristici. Le tecniche utilizzate dai vari livelli di RAID sono il data mirroring (copia integrale dei dati su un secondo HD, migliora la sicurrezza) ed il data striping (distribuire "strisce" di dati sui vari dischi per parallelizzare le letture/scritture, aumenta le prestazioni). La fault tolerance è garantita, se non per mirroring, dall'uso di hamming code o più spesso da bit di parità.

Livelli di RAID:

- **RAID 0** Puro data striping per migliorare soltanto le prestazioni
- **RAID 1** Puro mirroring, senza migliorare la velocità
- **RAID 2** Si usa lo striping dei dati a livello di bit (ogni bit in sequenza è scritto in un disco diverso), congiunto all'uso del codice di Hamming per garantire fault tolerance. Lo striping a livello di bit ha dei considerevoli drawback: gli accessi non possono essere paralleli, poichè i dati devono essere ricostruiti mano a mano che le letture proseguono, e la rotazione dei dischi deve essere sincronizzata, proprio perchè si deve leggere un bit da un disco ed un bit dall'altro alternatamente. Sconsigliato.
- **RAID 3** Striping a livello di byte congiunto con i bit di parità, che garantiscono la fault tolerance, salvati su un disco dedicato. Vale quanto detto per il bit-level striping, non ci sono particolari vantaggi nell'usare i livelli 2 e 3.
- **RAID 4** Lo striping avviene a livello di blocco, questo fa sì che le operazioni di accesso ai dischi siano parallelizzabili, fattore decisivo nel miglioramento delle prestazioni. Non c'è più necessità, per ricostruire una singola parola, di effettuare letture alternate tra i dischi, non è quindi più necessaria la sincronizzazione tra i dischi. I parity bit sono ancora salvati in un disco dedicato e separato. Il RAID 4 costituisce un balzo significativo rispetto ai livelli 2 e 3.
- **RAID 5** Ancora block-level striping, i bit i parità sono però distribuiti tra i dischi, dove si alternano ai blocchi di dati da leggere. Sono necessari almeno 3 hard disk per realizzare questo livello di RAID ed è tollerato il malfuzionamento di massimo un disco alla volta.
- **RAID 6** Aumenta l'affidabilità del livello 5. Block-level striping con bit di parità distribuiti tra i dischi. Viene aumentata però la ridondanza dei bit di parità, in modo tale che il sistema tolleri il malfunzionamento di ben 2 dischi contemporaneamente.
- **RAID 1+0** Data striping (come per il livello 1 si intende a livello di blocco, altrimenti non aumenterebbe le prestazioni) combinato con il mirroring.

17 File System

L'introduzione al file system in questo riassunto sarà particolarmente sintetica, poichè sono molti gli aspetti da trattare che risultano spesso anche poco utili al fine della preparazione per l'esame, come ad esempio le modalità di protezione dei file (access control list e user classification) o le modalità di accesso ai file (sequenziale o diretta) o ancora il file system mounting. Gli aspetti fondamentali e le loro scelte implementative, sono trattati normalmente come per gli altri argomenti. Per approfondire vedi libro.

Si definisce file system l'insieme di strutture ospitate su disco che permette la conservazione e gestione di dati non volatili. Il file system è quindi preposto alla gestione dei file, un tipo di dato astratto definito come la minima unità logica di rappresentazione dell'informazione nella memoria secondaria. Le directory sono a loro volta file, da considerarsi speciali poichè conservano solo dati necessari per il sistema relativi ai file a loro collegati (lunghezza, blocco per il primo accesso, diritti di accesso,etc.). Proprio grazie alle directory il file system assume una struttura caratterizzante, definita durante l'implementazione, che può essere:

- **Single Level** Tutti i file sono contenuti in un'unica directory. Con questa struttura insorgono problemi di nomenclatura e di raggruppamento, è impossibile da usare con più utenti.
- **Two level** Ogni utente usa una User File Directory (UFD) con i propri file. Al livello superiore vi è una Mother File DIrectory (MFD) con i dati relativi alle varie UFD. Si possono definire cartelle parallele alle UFD, usate per i file di sistema. Diventa possibile specificare un search path, non più coincidente con lo stesso file name, che identifichi univocamente un file.
- **Tree** Ottenuto generalizzando la struttura a due livelli. Le entry di una qualsiasi directory possono essere file o directory a loro volta. Permette la possibilità di raggruppamento mantenendo l'efficienza della ricerca. Con questa struttura viene introdotto il concetto di working directory di un processo, nonchè la suddivisione in path assoluto o relativo. Dal punto di vista implementativo, vanno introdotte delle system call per il cambio di directory e vanno definite delle policy per l'eliminazione delle directory, oltre che per la protezione degli accessi a determinate directory.
- **Grafo Aciclico** Diretta evoluzione della struttura ad alberi, permette alle directory di contenere link a file o altre directory non nella stessa gerachia per permetere la condivisione di risorse. Le policy sviluppate in merito si basano sulla distinzione dei link in due sottogruppi: symbolic link o hard link. Il link simbolico è generalmente un semplice file contenente un percorso ad un file/directory reale. Nel caso la risorsa venga eliminata si può scegliere se ricercare nel file system i vari link simbolici relativi ed eliminare anche essi oppure lasciarli pendenti. Corrispondono ai collegamenti in UNIX/Windows. Gli hard link, d'altro canto, non permettono l'eliminazione del file linkato se esistono ancora hard link che lo puntano. Viene tenuto per il file un contatore con il numero di riferimenti ad esso, il file può essere eliminato solo se tale contatore è a 0.
- **Grafo Generico** Sono permessi cicli nel file system. Generalmente non è utilizzato per il costo degli algoritmi di garbage collection.

Allocazione del file system Esistono principalmente tre modalità di allocazione del file system su disco. A seconda del tipo, un file system può permettere l'utilizzo di una o più di esse.

- **Allocazione Contigua** Lo spazio di ogni file viene allocato in blocchi di memoria adiacenti. Questo semplifica le entry nella directory, in quanto è necessario tenere solo il blocco di inizio del file e la sua lunghezza. Permette anche anche sia l'accesso sequenziale che quello diretto, inoltre riduce il tempo di seek su disco. Tuttavia è necessario stabilire una politica di allocazione (Best-fit, First-fit, etc.) e, proprio come con l'allocazione contigua nella memoria primaria, si incorre in un'elevata frammentazione del disco. Potrebbe essere necessario implementare costosissimi algoritmi di deframmentazione. Risulta inoltre molto problematico

estendere i file, in quanto deve esserci spazio libero contiguo all'attuale locazione del file. In conclusione l'allocazione contigua porta ad un utilizzo molto inefficiente della memoria.

- **Allocazione a Lista** Un file è costituito da una lista di blocchi concatenati, di cui il primo è memorizzato nella entry della directory. Questa tecnica non crea frammentazione esterna, ma permette efficaciemente solo l'accesso diretto. In ogni caso, richiede una seek per ogni blocco da leggere e parte della memoria è sprecata per il puntatore e altre informazioni che garantiscono l'affidabilità. Una variante più ottimizzata per l'accesso casuale fa uso della **File Access Table (FAT)**. Questa è una tabella che occupa i primi blocchi del volume e contiene una entry per ogni blocco allocato, indicizzate sul numero di blocco logico. Ad ogni entry si registra anche la posizione sulla tabella del prossimo blocco del file da leggere, rendendo così le entry di un file sulla FAT collegate come una linked list, ma molto più veloci da scorrere in caso di accesso diretto. Purtroppo è facile vedere come le dimensioni della FAT (spesso FAT32 perchè con entry di 32 bit) impediscano una scansione efficiente e comincino ad occupare una porzione significativa di memoria con l'aumentare delle dimensioni del disco. Questi svantaggi rendono oggi l'uso della FAT proibitivo.

- **Allocazione indicizzata** Tecnica ispirata alla paginazione della memoria primaria. Ogni file mantiene un blocco indice contenente alla i-esima posizione l'indirizzo dell'i-esimo blocco del file. Permette l'accesso diretto e non si ha frammentazione, tuttavia rispetto all'allocazione a lista (non alla FAT) tende a sprecare più memoria, in quanto per ogni file richiede l'allocazione di un blocco indice che potrebbe essere non del tutto utilizzato. Per lo stesso motivo la dimensione del blocco indice limita la dimensione del file massima. Per risolvere questo problema si può adottare uno dei seguenti schemi multilivello:

  - **Linked scheme** Anzichè un unico blocco indice si crea una linked list di blocchi indice. Pertanto se un blocco indice non è sufficiente, se ne crea un altro riferibile dall'ultimo indirizzo del blocco precedente.
  - **Multilevel scheme** Simile alla paginazione gerarchica. Si ha una tabella esterna i cui puntatori puntano ai blocchi indice. Può anche essere a più di due livelli, permettendo la creazione di file molto grandi.
  - **Combined scheme** Usata dai file system dei sistemi UNIX. Ad ogni file è associato un inode, blocco particolare che fa anche da blocco indice. Nelle prime n posizioni questo contiene i puntatori diretti a blocchi di dati. Nelle ultime tre posizioni contiene puntatori *single inderect, double indirect, triple indirect.*

  C'è inoltre da dire che, per poter navigare un file, deve essere caricato in memoria anche il blocco indice. Sarà rimosso al momento della chiusura del file.

Per quanto riguarda le performance dell'allocazione indicizzata, si hanno molte similitudini con quella a lista, perchè in ogni caso potrebbero essere necessarie molte operazioni di I/O per la lettura dei blocchi.

In generale, la maggior parte dei SO general-purpose implementano strutture per più di un tipo di allocazione, in modo d'avere un utilizzo della memoria più efficiente a seconda del caso.

Implementazione delle directory Ogni directory mantiene una lista con le informazioni di ogni file/sottodirectory discendente. Le implementazioni di questa lista influenzano l'efficienza in memoria e le prestazioni. Le principali scelte implementative sono: lista lineare o tabella di hash. Nel primo caso la directory contiene una lista (lineare o concatenata) di puntatori al primo blocco di ogni file (o al blocco indice). Tende ad essere pesante, poichè ogni operazione richiede un look-up $O(n)$. Si ha la possibilità di migliorare il look-up se si mantiene una lista ordinata con una ricerca binaria ($O(log_2 n)$). Nel caso si decida di usare una tabella di hash si ha una struttura dati più pesante ma molto più performante della lista, con la quale vanno però gestite le collisioni (lista trabocco).

Gestione dello spazio libero Il sistema deve mantere informazioni relative allo spazio libero, in particolare dei puntatori a tali blocchi. Scegliere come tenere traccia di tali informazioni influenza la velocità di ricerca dei blocchi liberi e di conseguenza quella di allocazione.

- **Bit Vector o Bit Map** In un disco di N blocchi si mantengono in un'area di memoria apposita N bit 0/1 che indicano se un blocco è libero oppure occupato. Questo metodo è veloce e soprattutto rende molto efficiente la ricerca di uno spazio contiguo libero (lo spazio contiguo è importante nel caso si usi allocazione contigua o per ridurre i tempi di seek durante gli accessi). Purtroppo però è poco efficiente in termini di spazio, inoltre, per avere buone prestazioni, deve essere *mantenuto sempre in memoria*.
- **Linked List** Ogni blocco libero mantiene un puntatore al successivo blocco libero. In questo modo in memoria si tiene solo il primo puntatore della lista. Tuttavia, sebbene effettuato raramente, è costoso attraversare tutta la lista e non è ottenibile spazio contiguo.
- **Raggruppamento** Simile all'allocazione indicizzata. Si tiene una linked list i cui nodi contengono riferimenti a blocchi liberi (come un blocco indice) e ad altri nodi liberi "indice".
- **Conteggio** Partendo dal presupposto che spesso i blocchi solo liberati in maniera contigua (specialmente se il sistema utilizza clustering o allocazione contigua), si mantiene una lista di entry. Ogni entry è costituita dall'indirizzo del primo blocco di zona libero e dal *numero di blocchi liberi consecutivi*. La lista risulta molto più corta di una classica linked list.

18 Sistema di I/O

Dal punto di vista hardware i dispositivi di I/O sono collegati al sistema principale tramite varie tipologie di bus e porte. Ognuna di queste interfacce possiede un *controllore*, nei cui registri la CPU può scrivere determinate istruzioni da eseguire. Questi registri solitamente sono:

- `Data IN` e `Data Out`
- `Di stato`: viene letto dalla CPU, permette di rilevare errori e di verificare lo stato di completamento delle operazioni.
- `Di controllo`: viene scritto per fornire istruzioni e modificare le modalità di operazione.

L'accesso a tali registri può avvenire attraverso apposite istruzioni di I/O, si parla in questo caso di periferiche **I/O mapped**, oppure mappando i dispositivi in determinate aree di memoria (**Memory Mapped**). Esistono inoltre differenti tecniche con le quali la CPU ed il controllore comunicano:

- **Polling** L'host controlla il valore di due bit nei registri del controllore, **busy bit** e **write bit**, ciclicamente, scrivendo le istruzioni quando è libero. Questa è chiaramente un tecnica di **busy waiting** e causa uno spreco delle risorse e del tempo di CPU, specialmente se si cerca di comunicare con periferiche molto lente.

- **Interrupt** L'host ordina un'operazione al controller e questo manda un segnale di interrupt in una linea dedicata per comunicare il completamento (o un errore) all'host. Alla ricezione dell'interrupt la CPU invoca l'apposito *handler*, identificato con un codice specifico, che si occupa della gestione. Gli interrupt sono una problematica complessa, poichè è necessario che siano numerati in un apposito *vettore di interrupt* (affinchè a dato interrupt corrisponda un dato handler) e gestibili tramite un ordine di priorità.

- **Direct Memory Access (DMA)** Questa tecnica è basata sull'idea di scaricare l'onere della gestione dell'I/O da parte della CPU su un *controllore dedicato*. Questo, una volta ottenuta l'istruzione dalla CPU, interagisce direttamente con il controller della periferica, per poi copiare direttamente in memoria i dati letti. Al termine dell'operazione, notifica la CPU con un interrupt. Sebbene durante il trasferimento il controllore vieti l'accesso alla memoria alla CPU (al fine di evitare inconsistenze), tale tecnica è molto usata in quanto migliora sensibilmente il throughput: si nota infatti una netta diminuzione degli interrupt da gestire da parte della CPU, che si riducono ad un solo interrupt per blocco trasferito.