

SISTEMI OPERATIVI

Mac™ OS



Libera Mente



Indice

Parte prima Generalità

Capitolo 1 Introduzione

- 1.1 Che cos'è un sistema operativo
- 1.2 Avvio del sistema (Bootstrap)
- 1.3 Interruzioni
- 1.4 Struttura della memoria
- 1.5 Struttura di I/O
- 1.6 Sistema monoprocesso
- 1.6.1 Sistema multiprocesso
- 1.7 Cluster
- 1.8 Multiprogrammazione – job pool
- 1.9 Dual mode
- 1.10 Timer
- 1.11 Gestione dei processi
- 1.12 Cache
- 1.13 Computazione client-server e peer-to-peer (P2P)

Capitolo 2 Strutture dei sistemi operativi

- 2.1 Servizi di un sistema operativo
 - 2.1.2 Interprete dei comandi e GUI
- 2.2 Chiamate di sistema
 - 2.2.1 Categorie di Syscall
- 2.3 Struttura (semplice) del sistema operativo
 - 2.3.1 Struttura (stratificata) del sistema operativo
 - 2.3.2 Struttura (modulare) del sistema operativo
 - 2.3.3 Microkernel
- 2.4 Macchine virtuali
- 2.5 Tipi di Sistema Operativo

Parte seconda Gestione dei processi

Capitolo 3 Processi

- 3.1 Concetto di processo
 - 3.1.2 Stato del processo
 - 3.1.3 Blocco di controllo dei processi (PCB)
 - 3.1.4 Thread
- 3.2 Code di scheduling

- 3.2.1 Scheduler
- 3.2.2 Cambio di contesto
- 3.3 Creazione di un processo
 - 3.3.1 Terminazione di un processo
- 3.4 Comunicazione tra processi (IPC)
 - 3.4.1 Sistemi a memoria condivisa
 - 3.4.2 Sistemi a scambio di messaggi
- 3.5 Comunicazione nei sistemi client – server (Socket)
- 3.6 Pipe

Capitolo 4 Scheduling della CPU

- 4.1 Ciclicità delle fasi d'elaborazione e di I/O
 - 4.1.2 Scheduler della CPU
 - 4.1.3 Scheduling con diritto di prelazione
 - 4.1.4 Dispatcher
- 4.2 Criteri di scheduling
- 4.3 Algoritmi di scheduling
 - 4.3.1 Scheduling in ordine d'arrivo (FCFS)
 - 4.3.2 Scheduling per brevità (SJF)
 - 4.3.3 Scheduling per priorità
 - 4.3.4 Scheduling circolare
 - 4.3.5 Scheduling a code multiple
 - 4.3.6 Scheduling a code multiple con retroazione
- 4.4 Scheduling multiprocessor
 - 4.4.1 Soluzioni di scheduling per multiprocessori
 - 4.4.2 Predilezione per il processore
 - 4.4.3 Bilanciamento del carico
 - 4.4.4 Processori multicore
- 4.5 Hyper-Threading
- 4.6 Virtualizzazione e scheduling
- 4.7 Esempio: scheduling di Linux

Capitolo 5 Thread

- 5.1 Definizione di thread
- 5.2 Multithreading
 - 5.2.1 Vantaggi
 - 5.2.2 Threads nelle architetture multicore
- 5.3 Modelli di programmazione multithread
 - 5.3.1 Modello da molti a uno

- 5.3.2 Modello da uno a uno
- 5.3.2 Modello da molti a molti
- 5.4 Librerie dei thread
 - 5.4.1 Pthreads
- 5.5 Cancellazione dei thread
 - 5.5.1 Gestione dei segnali
- 5.6 Thread pool
- 5.7 Strategie di threading
- 5.8 System call fork() ed exec()
- 5.9 Thread nel sistema Windows

Capitolo 6 Sincronizzazione dei processi

- 6.1 Introduzione
 - 6.1.2 Race condition
- 6.2 Problema della sezione critica
- 6.3 Soluzione di Peterson (applicata a 2 Processi)
- 6.4 Soluzione per più processi: algoritmo del fornaio
- 6.5 Hardware per la sincronizzazione (lock)
- 6.6 Semafori
 - 6.6.1 Uso dei semafori
 - 6.6.2 Realizzazione
 - 6.6.3 Stallo e attesa indefinita (deadlock)

Parte terza Gestione della memoria

Capitolo 7 Memoria centrale

- 7.1 Introduzione
- 7.2 Dispositivi essenziali
 - 7.2.1 Associazione degli indirizzi
 - 7.2.2 Differenze tra spazi di indirizzi logici e fisici
 - 7.2.3 Loader dinamico (DDL)
 - 7.2.4 Collegamento dinamico e librerie condivise (DLL)
- 7.3 Swapping
- 7.4 Allocazione contigua della memoria
 - 7.4.1 Allocazione della memoria
 - 7.4.2 Frammentazione
- 7.5 Paginazione
 - 7.5.1 Implementazione della paginazione

- 7.5.2 Architettura di paginazione e TLB
 - 7.5.3 Protezione
 - 7.5.4 Pagine condivise
 - 7.6 Struttura della tabella delle pagine
 - 7.6.1 Paginazione Gerarchica
 - 7.6.2 Tabelle delle pagine hash
 - 7.6.3 Tabelle delle pagine invertita
 - 7.7 Segmentazione
 - 7.7.1 Architettura di segmentazione
 - 7.7.2 Segmentazione vs Paginazione
 - 7.8 Pentium Intel
 - 7.8.1 Segmentazione in Pentium
 - 7.8.2 Paginazione in Pentium
 - 7.8.3 Intel 64 bit
 - 7.8.4 ARM-1
 - 7.8.5 ARM-2
- 
- ## Capitolo 8 Memoria virtuale
- 8.1 Introduzione
 - 8.2 Paginazione su richiesta
 - 8.2.1 Come funziona la paginazione su richiesta
 - 8.2.2 Lista dei frame liberi
 - 8.2.3 Prestazioni della paginazione su richiesta
 - 8.3 Copiatura su scrittura
 - 8.4 Sostituzione delle pagine
 - 8.4.1 Sostituzione delle pagine secondo l'ordine d'arrivo (FIFO)
 - 8.4.2 Sostituzione ottimale delle pagine
 - 8.4.3 Sostituzione delle pagine usate meno recentemente (LRU)
 - 8.4.4 Sostituzione delle pagine basata su conteggio
 - 8.4.5 Algoritmi con memorizzazione transitoria delle pagine
 - 8.4.6 Applicazioni e sostituzione della pagina
 - 8.5 Allocazione dei frame
 - 8.5.1 Numero minimo di frame
 - 8.5.2 Algoritmi di allocazione
 - 8.5.3 Allocazione globale e allocazione locale
 - 8.5.4 Accesso non uniforme alla memoria (NUMA)

8.6 Thrashing

8.6.1 Cause del thrashing

8.6.2 Working set

8.6.3 Frequenza di page fault

8.7 Prepaganazione

8.8 Linux

8.9 Windows

8.10 Criteri di dimensionamento delle pagine

Parte quarta Gestione della memoria secondaria

Capitolo 9 Memoria secondaria e terziaria

9.1 Introduzione

9.2 Dischi rigidi

9.3 Dispositivi NVM (SSD)

9.3.1 Memorie flash

9.3.2 Algoritmi del controllore delle NAND Flash

9.4 Nastri magnetici

9.5 Formattazione del disco, partizioni e volumi

9.5.1 Blocco d'avviamento (bootstrap)

9.5.2 Blocchi difettosi (bad blocks)

9.6 Gestione dell'area di swapping

9.6.1 Uso dell'area di swapping

9.6.2 Collocazione dell'area di swapping

9.7 Connessione dei dispositivi di memorizzazione

9.8 Strutture RAID

9.8.1 Miglioramento dell'affidabilità tramite la ridondanza

9.8.2 Miglioramento delle prestazioni tramite il parallelismo

9.8.3 Livelli RAID

9.8.4 Scelta di un livello RAID

9.8.5 Problemi connessi a RAID

Capitolo 10 Sistemi di I/O

10.1 Introduzione

10.2 Hardware di I/O

10.2.1 I/O memory mapped

10.2.2 Modalità di scambio di dati

10.3 Interfaccia di I/O delle applicazioni

10.3.1 Orologi e timer

10.4 Sottosistema di I/O del kernel

10.4.1 Scheduling dell'I/O

10.4.2 Gestione del buffer

Capitolo 11 Interfaccia del file system

11.1 Introduzione

11.2 Concetto di file

11.2.1 Attributi dei file

11.2.2 Operazioni sui file

11.2.3 Tipi di file

11.2.4 Struttura dei file

11.2.5 Struttura interna dei file

11.3 Metodi d'accesso

11.4 Struttura delle directory

11.4.1 Directory a un livello

11.4.2 Directory a due livelli

11.4.3 Directory con struttura ad albero

11.4.4 Directory con struttura a grafo aciclico

11.4.5 Directory con struttura a grafo generale

11.5 Protezione

11.5.1 Tipi d'accesso

11.5.2 Controllo degli accessi

11.5.3 Altri metodi di protezione

Capitolo 12 Realizzazione del file system

12.1 Introduzione

12.2 Realizzazione delle directory

12.2.1 Lista lineare

12.2.2 Tabella hash

12.3 Metodi di allocazione

12.3.1 Allocazione contigua

12.3.2 Allocazione concatenata

Sistemi Operativi

Introduzione (Capitolo 1)

1.1 Che cos'è un sistema operativo

Un **sistema operativo** è un programma formato da un insieme di programmi (*software*) che gestisce gli elementi fisici di un calcolatore (*hardware*); agisce da intermediario fra l'utente e la struttura fisica del calcolatore.

Dal punto di vista del calcolatore, il sistema operativo deve decidere come assegnare ai diversi programmi, le risorse necessarie affinché il calcolatore risulti efficiente. Quindi un sistema operativo è un **programma di controllo**: gestisce l'esecuzione dei programmi utenti in modo da impedire che si verifichino errori.

Tra i vari programmi, il sistema operativo è composto da un **kernel** (*nucleo*), che è fondamentale ma non sufficiente a rendere un sistema operativo usabile. Il *kernel* è il software che fa da ponte tra l'hardware (ad esempio la RAM) e i programmi in esecuzione, distribuendo le risorse. Al momento esistono **4 tipi diversi** di kernel: *monolitico*, *microkernel*, *kernel ibrido* e *nanokernel* e si differenziano in base al tipo di accesso alle risorse hardware, i più utilizzati sono il kernel monolitico e il kernel ibrido.

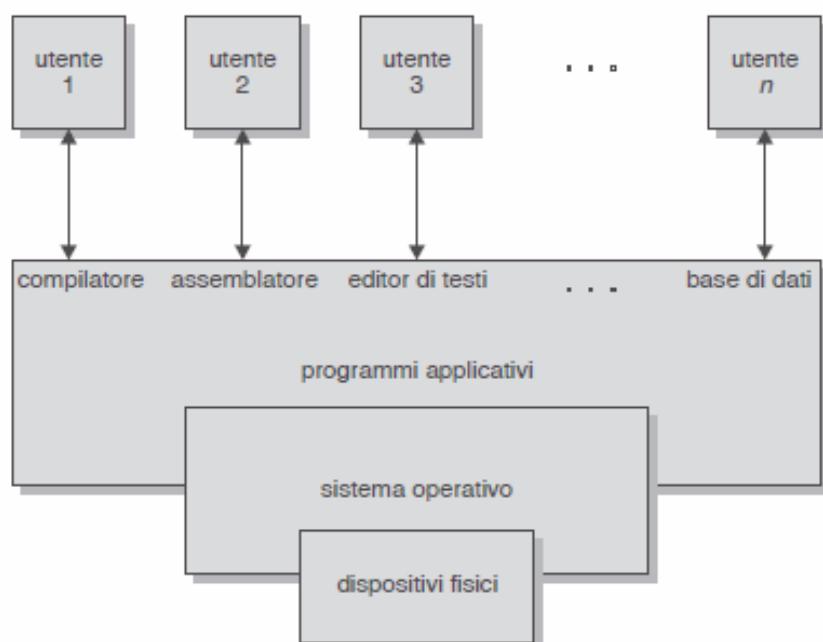
Oltre al kernel vi sono due tipi di programmi: i **programmi di sistema**, associati al sistema operativo e che non fanno parte del kernel, e i **programmi applicativi**, che includono tutti i programmi non necessari al funzionamento del sistema.

Nel 1998, la Microsoft venne accusata di includere troppe funzioni nel sistema operativo promuovendo una concorrenza sleale nei confronti dei produttori e rivenditori di applicazioni.

Un sistema di calcolo si può suddividere in 4 componenti: *dispositivi fisici*, *sistema operativo*, *programmi applicativi* e *utenti*.

I **dispositivi fisici** sono composti dall'unità centrale, CPU (*Central Processing Unit*), dalla memoria e dai dispositivi I/O.

I **programmi applicativi** (editor di testo, compilatori, ...) usano queste risorse per la **risoluzione** dei problemi computazionali degli utenti. Il sistema operativo controlla e coordina l'uso dei dispositivi da parte dei programmi applicativi per gli utenti.



1.2 Avvio del sistema (Bootstrap)

Un moderno calcolatore è composto da una CPU e da un certo numero di *controllori* connessi attraverso un canale di comunicazione comune, chiamato *bus*, che permette l'accesso alla memoria condivisa del sistema. Ciascuno di questi controllori si occupa di un particolare tipo di dispositivo fisico.

L'avviamento del sistema (*booting*) richiede un programma iniziale, detto **programma d'avviamento** (*bootstrap program*), contenuto in tipi di memoria chiamati *firmware*. Il firmware è un programma integrato direttamente in un componente elettronico come le memorie a sola lettura, le *ROM*, e le memorie programmabili cancellabili elettricamente, le *EEPROM*. Questo tipo di memoria presenta il vantaggio di non dover essere inizializzata ed essere immune ai virus. Un problema, generato dal *firmware*, risiede nel fatto che l'esecuzione del codice è più lenta di quanto avvenga con la RAM. Alcuni sistemi operativi, infatti, vengono memorizzati nel *firmware* e copiati nella RAM per essere eseguiti rapidamente. Lo svantaggio è che il *firmware* è molto costoso.

Per sistemi operativi di grandi dimensioni il *bootstrap* è memorizzato nel *firmware* mentre il sistema operativo si trova sul disco. Un disco che contiene una partizione di avvio è chiamato **boot disk** o **disco di sistema**.

Alla fine, il programma di avviamento individua e carica nella memoria il kernel del sistema operativo; il sistema operativo avvia l'esecuzione del primo processo di elaborazione (**init**) e attende che si verifichi qualche evento.

1.3 Interruzioni

Un evento è di solito segnalato da un'interruzione della sequenza di esecuzione della CPU, che può essere causata da un dispositivo fisico (*hardware*) o da un programma (*software*).

Nel primo caso si parla di **segnali di interruzione** o **interruzione** (*interrupt*): si tratta di segnali che i controllori dei dispositivi possono inviare alla CPU attraverso il bus di sistema.

Nel secondo caso si parla di **segnaletica d'eccezione** o **eccezione** (*exception* o *trap*), per esempio una divisione per zero o un accesso alla memoria non valido, oppure a causa di una richiesta effettuata da un programma utente per ottenere l'esecuzione di un servizio del sistema operativo, attraverso un'istruzione detta **chiamata di sistema** (*system call*) o **chiamata supervisore** (*supervisor call*, *SVC*).

Quando la CPU riceve un interrupt, sospende ciò che sta facendo e comincia ad eseguire codice a partire da una locazione fissa, che contiene l'indirizzo di partenza della routine di interrupt. Una volta completata la procedura richiesta, la CPU riprende l'elaborazione.

Il modo più semplice per gestire le interruzioni è quello di impiegare una **procedura generale** che esamina le informazioni presenti nel segnale di interruzione, e invoca la procedura di gestione dello specifico segnale di interruzione. Poiché la gestione di un'interruzione deve essere rapida, si può usare una tabella di puntatori alle specifiche procedure. L'accesso a questa sequenza di indirizzi, detta **vettore delle interruzioni**, avviene per mezzo di un indice.

La gestione delle interruzioni deve anche salvare l'indirizzo dell'istruzione interrotta memorizzando l'indirizzo di ritorno nello *stack* di sistema. Terminato il servizio dell'interruzione, l'indirizzo di ritorno viene caricato nel **contatore di programma** (*program counter*), che contiene l'indirizzo della prossima istruzione da seguire.

1.4 Struttura della memoria

La CPU può caricare le istruzioni solo dalla memoria. I computer *general-purpose* (PC) eseguono la maggior parte dei programmi dalla memoria principale, chiamata **memoria ad accesso diretto: RAM**. La memoria principale è realizzata con una tecnologia basata su semiconduttori chiamata **memoria dinamica ad accesso diretto, DRAM**.

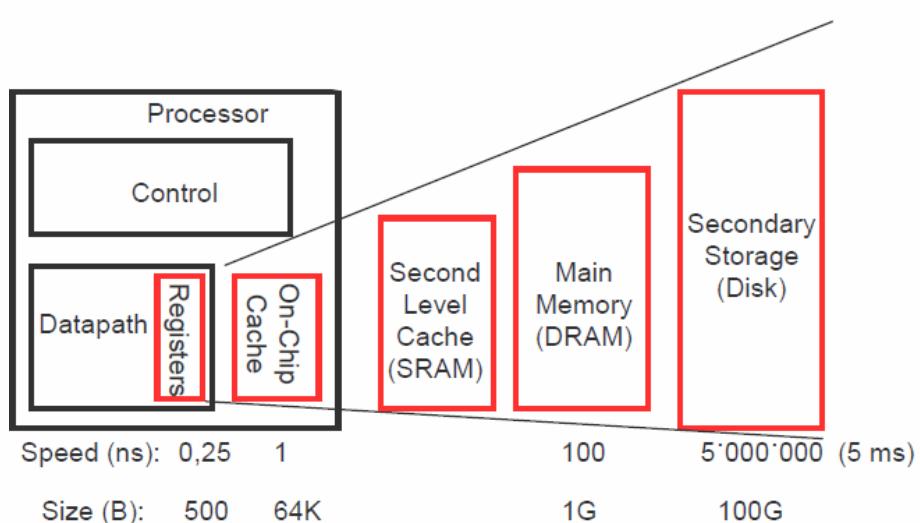
I computer utilizzano anche altri tipi di memoria come la **memoria di sola lettura (ROM)** che non può essere modificata. Le *EEPROM* contengono per lo più programmi statici. Un disco RAM si può progettare per essere volatile o non volatile. Il disco memorizza i dati in un vettore DRAM, che è volatile. Molti dischi RAM contengono un disco rigido magnetico e un alimentatore di riserva per la batteria perché nel caso di un'interruzione della corrente elettrica, il controllore del disco RAM copia i dati dalla RAM sul disco magnetico, per poi eseguire l'operazione inversa al ripristino della corrente elettrica.

La **memoria flash** è più lenta della DRAM ma non necessita di alimentazione esterna. Un'altra memorizzazione non volatile è la **NVRAM**, ovvero una DRAM senza batteria di scorta. Questo tipo di memoria egualgia la DRAM in velocità, ma la sua non volatilità ha durata limitata.

Tutte le tipologie di memoria forniscono un vettore di parole. L'istruzione *load* trasferisce il contenuto di una parola dalla memoria centrale in uno dei registri interni della CPU, mentre *store* copia il contenuto di uno di questi registri nella locazione di memoria specificata.

L'esecuzione di un'istruzione comincia con il **prelievo (fetch)** di un'istruzione dalla memoria centrale e il suo trasferimento nel registro. Successivamente si decodifica l'istruzione e una volta terminata l'esecuzione dell'istruzione sugli operandi, il risultato si può scrivere nella memoria.

Sia i programmi che i dati non possono risiedere nella memoria centrale perché non è sufficiente a contenere in modo permanente tutti i programmi e inoltre è un dispositivo di memorizzazione volatile, ovvero perde il suo contenuto quando si spegne il sistema o si ha un'interruzione dell'alimentazione elettrica. Per queste ragioni la maggior parte dei sistemi di calcolo comprende una **memoria secondaria**, come il *disco magnetico* per la memorizzazione sia di programmi sia dei dati. Esistono altri tipi di memoria, per esempio le memorie *cache*, i *CD-ROM* e i *nastri magnetici*. Le caratteristiche che differenziano i diversi sistemi di memorizzazione sono velocità, costo, dimensioni e volatilità. Infatti, le varie memorie possono essere ordinate attraverso una scala gerarchica: i gradini più alti ospitano i dispositivi più veloci ma anche più costosi; andando verso il basso il costo per bit diminuisce, mentre il tempo di accesso aumenta.



1.5 Struttura di I/O

Una buona parte del codice di un sistema operativo è dedicata alla gestione dell'I/O. Un calcolatore è composto da una CPU e da un insieme di controllori di dispositivi connessi mediante un bus comune. Ciascun controllore deve occuparsi di un particolare tipo di dispositivo. Un controllore **SCSI** (*small computer-system interface*), è capace di controllare 7 o più dispositivi. Un controllore di dispositivo dispone di una propria memoria interna, detta **memoria di transito (buffer)**, e di un insieme di registri. Il controllore trasferisce i dati tra i vari dispositivi periferici ad esso connessi. I sistemi operativi possiedono per ogni controllore del dispositivo un **driver** che funge da interfaccia con il resto del sistema.

Per avviare un'operazione di I/O, il driver del dispositivo carica i registri all'interno del controllore mentre questo comincia a trasferire i dati dal dispositivo al proprio buffer locale.

Al termine del trasferimento, il controllore informa il driver, tramite un'interruzione, di aver terminato l'operazione. Il driver passa, quindi, il controllo al sistema operativo restituendo i dati.

Per trasferimenti massicci si può generare un sovraccarico per questo si utilizza l'**accesso diretto alla memoria (DMA)**. Una volta impostati i buffer, il controllore trasferisce un intero blocco di dati direttamente nella memoria centrale senza alcun intervento da parte della CPU.

Nei sistemi moderni si utilizzano gli *switch*, in cui più dispositivi fisici possono interagire con varie parti del sistema piuttosto che contendersi un unico bus.

1.6 Sistemi monoprocessoress

Un **sistema monoprocessoress** è dotato di una sola CPU principale in grado di eseguire un insieme di istruzioni. Quasi tutti i sistemi possiedono altri processori dotati di un insieme ristretto di istruzioni che non eseguono processi utenti. Di solito sono guidati dal sistema operativo, mentre i processori integrati nell'hardware non sono collegati con il sistema operativo, poiché svolgono il proprio lavoro in autonomia.

1.6.1 Sistemi multiprocessoress

Un **sistema multiprocessoress**, o *sistema parallelo*, dispone più **unità di elaborazione (CPU)** che comunicano tra loro. Tale sistema ha 3 vantaggi principali:

1. **Maggiore produttività (throughput)**: aumentando il numero di unità di elaborazione è possibile svolgere un lavoro maggiore in meno tempo. Con n unità d'elaborazione la velocità non aumenta di n volte. Infatti, se più unità d'elaborazione collaborano nell'esecuzione di un compito, il sistema operativo deve gestire le operazioni affinché tutti i componenti funzionino correttamente producendo un sovraccarico.
2. **Economia di scala**. I sistemi multiprocessoress possono consentire risparmi rispetto a quelli dotati di una sola CPU, poiché possono condividere dispositivi periferici, mezzi di registrazione dei dati e alimentatori elettrici.
3. **Incremento dell'affidabilità**. Se le funzioni si possono distribuire tra più unità di elaborazione, un guasto di alcune di loro non blocca il sistema bensì lo rallenta.

I sistemi multiprocessoress attualmente in uso sono di due tipi. Alcuni impiegano la **multielaborazione asimmetrica (asymmetric multiprocessing AMP)**, in cui ad ogni unità di elaborazione si assegna un compito specifico.

Nei sistemi più comuni abbiamo la **multielaborazione simmetrica** (*symmetric multiprocessing SMP*), in cui ogni processore svolge tutte le operazioni del sistema. Un esempio di tale tecnica è dato da *Solaris*, il quale esegue più processi contemporaneamente (n processi se si hanno n CPU).

Per aumentare la potenza di calcolo nella multielaborazione si aggiungono nuove CPU. Se la CPU ha un controllo di memoria integrato, allora l'aggiunta di CPU può aumentare la quantità di memoria indirizzabile dal sistema. Però questo può causare il cambiamento di accesso alla memoria del sistema trasformando un **accesso uniforme alla memoria** (*UMA*), ovvero l'accesso alla memoria RAM da una qualsiasi CPU in tempo costante, in accesso **non uniforme alla memoria** (*NUMA*), ossia l'accesso ad alcune parti della memoria richiede più tempo.

Nelle recenti CPU sono raggruppate diverse **unità di calcolo** (*core*) in un singolo circuito. Questi circuiti sono più efficienti rispetto ad avere più circuiti dotati di una sola CPU perché la comunicazione all'interno di un singolo circuito è più veloce rispetto a quella tra un circuito e un altro. Inoltre, un circuito dotato di più *core* usa meno potenza rispetto a circuiti con un solo *core*.

Una delle ultime innovazioni sono i **server blade**, che contengono le schede del processore, dell'I/O e della rete. Ogni scheda madre (una scheda che ospita una CPU) di un *server blade*, avvia ed esegue in maniera indipendente il proprio sistema operativo.

1.7 Cluster

I **cluster** (*clustered systems*) sono abbassati sull'uso di più unità di elaborazione riunite per lo svolgimento di attività comuni. Differiscono dai sistemi multiprocessore per il fatto che sono composti da due o più calcolatori completi collegati tra loro. Un cluster si può definire formato da calcolatori che condividono la memoria di massa, connessi da una rete locale (*LAN*). Ciascun calcolatore esegue una serie di programmi e se si presenta un malfunzionamento, il calcolatore può appropriarsi del calcolatore malfunzionante e riavviare le applicazioni che erano in esecuzione.

Nei **cluster asimmetrici** un calcolatore rimane nello stato di **attesa attiva** (*hot standby mode*) mentre l'altro esegue le applicazioni.

Nei **cluster simmetrici** due o più calcolatori eseguono le applicazioni nello stesso tempo e si controllano reciprocamente in modo tale da ottenere una maggiore efficienza.

I programmi per essere efficienti sfruttano una tecnica chiamata **parallelizzazione** che consiste nel suddividere il programma in varie parti, eseguibili in parallelo sui vari computer del cluster.

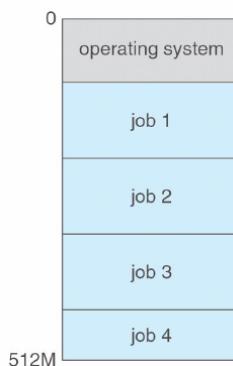
Altre forme sono i cluster di *sistemi paralleli* e quelli dei *sistemi connessi* attraverso reti geografiche (*WAN*). I primi permettono a più calcolatori di accedere agli stessi dati nella memoria di massa, poiché la maggior parte dei sistemi operativi non consente ciò e ricorrere a programmi specifici.

Per ottenere questo accesso condiviso ai dati, il sistema deve evitare i conflitti tra le operazioni attraverso la **gestione distribuita degli accessi** (*DLM*), attiva in alcuni cluster.

1.8 Multiprogrammazione – job pool

Tra le più importanti caratteristiche dei sistemi operativi vi è la **multiprogrammazione** che consente di eseguire più programmi o processi su un singolo processore ed è compito del sistema operativo gestire tutti i processi in modo efficace ed efficiente. In questo modo la percentuale di utilizzo della CPU aumenta e rimane sempre attiva.

Il sistema operativo tiene contemporaneamente in memoria centrale diversi lavori, ma questa è troppo piccola per contenere tutti i programmi da seguire, per questo vengono collocati inizialmente sul disco in un'area apposita, detta **job pool**, contenente tutti i processi in attesa di essere allocati nella memoria centrale.



Il sistema operativo sceglie uno dei processi contenuti nella memoria e inizia l'esecuzione ma a un certo punto potrebbe trovarsi nell'attesa di qualche evento. In questi casi, in un sistema non multiprogrammato, la CPU rimarrebbe inattiva, ma in un sistema con multiprogrammazione il sistema operativo passa un altro lavoro e lo esegue. Quando il primo lavoro ha terminato l'attesa, la CPU ne riprende l'esecuzione finché c'è almeno un lavoro da eseguire e la CPU non rimane mai inattiva.

La **partizione del tempo d'elaborazione** (*time sharing* o *multitasking*) è un'estensione logica della multiprogrammazione; ottimizza il tempo di utilizzo della CPU (**gli intervalli di tempo (time-slice) sono talmente piccoli da non poter essere percepiti dall'utente**), eseguendo più lavori con una frequenza tale da permettere a ciascun utente l'interazione col proprio programma senza rallentamenti. Quindi abbiamo l'esecuzione contemporanea di più processi da parte di un utente sullo stesso computer.

Un **sistema di calcolo interattivo** permette la comunicazione tra utente e sistema: l'utente impartisce le istruzioni direttamente al sistema operativo oppure a un programma e attende una risposta immediata. Il **tempo di risposta** dovrebbe essere quindi breve (< 1 secondo).

Un **sistema operativo multitasking** si avvale dello **scheduling** della CPU e della multiprogrammazione. Lo **scheduler dei processi** è quel componente del sistema operativo che si occupa di decidere quale processo va mandato in esecuzione. E' pensato per mantenere la CPU occupata il più possibile, avviando un processo mentre un altro è in attesa.

Un programma caricato in memoria e pronto per l'esecuzione è noto come **processo**. Un processo durante la sua esecuzione, impegna la CPU per un breve periodo di tempo prima di richiedere operazioni di I/O.

Il **multitasking** e la multiprogrammazione richiedono la presenza di diversi processi in memoria, se alcuni processi sono pronti per il trasferimento in memoria centrale, ma lo spazio disponibile non è sufficiente, il sistema deve fare una selezione. Questa scelta si chiama **job scheduling, pianificazione dei lavori**.

Il sistema operativo garantisce tempi di risposta accettabili grazie alla tecnica dello **swapping**, che consente di scambiare i processi presenti in memoria con quelli che risiedono su disco e viceversa. Un metodo per ottenere lo stesso risultato è la **memoria virtuale**, una tecnica che consente l'esecuzione dei lavori d'elaborazione anche non interamente caricati nella memoria. Il vantaggio della memoria virtuale è che i programmi possono avere dimensioni maggiori della memoria fisica, inoltre essa estrae la memoria centrale in un grande vettore separando la **memoria logica**, vista dell'utente, della **memoria**

fisica. I sistemi a partizione del tempo devono inoltre fornire un *file system*, i quali a loro volta necessitano di una gestione.

Un **file system** indica un meccanismo con il quale i file sono posizionati e organizzati su dispositivi di memoria di massa (come unità a nastro magnetico, dischi rigidi, dischi ottici, unità di memoria a stato solido) o su dispositivi remoti tramite protocolli di rete.

1.9 Dual mode

Per garantire il corretto funzionamento del sistema è necessario distinguere tra l'esecuzione del codice del sistema operativo e il codice scritto dall'utente. Esistono due modalità: *l'user mode* e il *kernel mode*. Per indicare quale sia la modalità attiva, la CPU è dotata di un *bit*, chiamato **bit di modalità**: di *sistema* (0) *utente* (1).

All'avviamento del sistema, il bit è posto in *kernel mode*. Ogni volta che si verifica un'interruzione o una *trap* si passa dall'*user mode* a quella di *kernel mode*, cioè si pone a 0 il bit di modo. Quando il sistema operativo riprende il controllo, prima di passare il controllo al programma utente, viene impostato a 1 il bit di modo passando all'*user mode*.

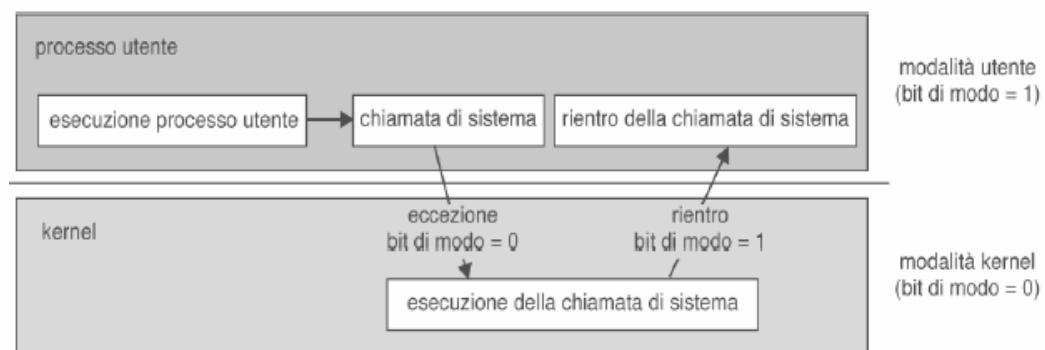
La **dual mode** consente la protezione del sistema operativo proteggendo le **istruzioni privilegiate**, ovvero le istruzioni di macchina che possono causare danni allo stato del sistema. Poiché la CPU consente l'esecuzione di queste istruzioni soltanto nel *kernel mode*, se si tenta di fare eseguire in *user mode* un'istruzione privilegiata, la CPU non la esegue.

Le **chiamate di sistema** (*system call*) sono gli strumenti con cui un programma utente richiede al sistema operativo di compiere operazioni ad esso riservate. Vi sono vari modi per generare una chiamata di sistema. Una chiamata di sistema è realizzata come una *trap*.

Quando un programma utente esegue una *system call*, questa è gestita dalla CPU come un'interruzione. Il controllo passa, tramite il vettore delle interruzioni e si pone il bit di modo in *kernel mode*.

Il sistema MS-DOS per esempio è stato sviluppato per l'architettura *8088 Intel* priva del bit di modo, quindi un programma utente potrebbe cancellare il sistema operativo riscrivendolo.

Gli errori di violazione della modalità sono rilevati dall'hardware e sono gestiti dal sistema operativo. Quando si imbatte nell'errore di un programma, il sistema operativo deve terminare il programma in maniera anomala. Si compone un messaggio di errore e si rilascia la memoria del programma incriminato. Il contenuto della memoria rilasciata è trascritto su un file, perché l'utente o il programmatore possono esaminarlo ed eventualmente correggerlo.



1.10 Timer

Il sistema operativo impedisce che un programma utente entri in un loop infinito, per questo si può utilizzare un **timer** affinché invii un segnale di interruzione alla CPU a intervalli di tempo specifici, che possono essere *fisici* o *variabili*. Un **timer variabile** si realizzi mediante un generatore di impulsi a

frequenza fissa e un contatore. Il sistema operativo assegna un valore al contatore, che si decrementa ad ogni impulso e quando raggiunge il valore 0 si genera un segnale di interruzione.

Prima di restituire all'utente il controllo, il sistema assegna un valore al timer. Le istruzioni usate dal sistema per modificare il funzionamento del timer si possono eseguire soltanto in *kernel mode*.

La presenza di un timer garantisce che nessun programma utente possa essere eseguito troppo a lungo.

1.11 Gestione dei processi

Un **processo** si può considerare come un programma in esecuzione. Un processo necessita di alcune risorse, tra cui tempo di CPU, memoria, file e dispositivi di I/O. Queste risorse si possono attribuire al processo al momento della sua creazione, oppure si possono assegnare durante l'esecuzione.

Un programma di per sé *non* è un processo è un'*entità passiva*, mentre un processo è un'*entità attiva* con un **contatore** che indica la successiva istruzione da eseguire. L'esecuzione di un processo deve essere sequenziale: la CPU esegue le istruzioni del processo una dopo l'altra finché il processo termina. Un processo *multithread* possiede più contatori di programma, ognuno dei quali punta all'istruzione successiva. I sistemi operativi sono responsabile della gestione dei processi.

1.12 Cache

Le informazioni sono mantenute in unità di memoria come la memoria centrale; al momento del loro uso si copiano temporaneamente in unità più veloce: la **cache**. Quando si accede ad un'informazione, si controlla se è già presente all'interno della *cache*; se così fosse si utilizza direttamente la copia contenuta nella *cache*, altrimenti la si preleva dalla memoria centrale e la si copia nella *cache*, perché si suppone che queste informazioni presto servirà ancora.

I registri presenti nella CPU rappresentano per la memoria centrale una *cache* ad alta velocità. La maggior parte dei sistemi è dotata di una o più *cache* di dati nella **gerarchia delle memorie**.

La memoria centrale si può considerare una *cache* per la memoria secondaria, poiché i devono prima essere riportati nella memoria centrale e poi trasferiti nella memoria secondaria.

I dati del **file system** possono apparire a diversi livelli della gerarchia di memoria.

In una struttura gerarchica può accadere che gli stessi dati sono mantenuti contemporaneamente in diversi livelli del sistema di memorizzazione.

Livelli gerarchia delle memorie

- 1) **Registri** → sono la memoria più veloce, sono gestiti dal compilatore che alloca le variabili ai registri e gestisce i trasferimenti allo spazio di memoria.
- 2) **Cache di 1°livello** → si trovano sullo stesso chip del processore (*L1 cache*) e sono una tecnologia SRAM. I trasferimenti dalle memorie di livello inferiore sono gestiti dall'hardware.
- 3) **Cache di 2° e 3°livello** → quando esiste, questa tipologia di cache può essere sia sullo stesso chip del processore (*L2 cache*), sia su un chip separato e sono una tecnologia SRAM.
Il numero dei livelli di *cache* e delle loro dimensioni dipendono da vincoli di prestazioni e costo.
Come per la *cache di 1°livello*, i trasferimenti dalla memoria di livello inferiore sono gestiti dall'hardware.
- 4) **Memoria RAM** → di solito sono in tecnologia DRAM (SDRAM). I trasferimenti dalle memorie di livello inferiore sono gestiti dal sistema operativo (*memoria virtuale*) e dal programmatore.
- 5) **Dischi, memoria secondaria e terziaria.**

Le informazioni vengono di volta in volta copiate solo tra livelli adiacenti.

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

1.13 Computazione client-server e peer-to-peer (P2P)

La rete **client/server** è caratterizzata da una architettura in cui il **client** richiede dei servizi ad un **server** che glieli fornisce. Per poter comunicare tra di loro è necessario che le macchine parlino lo stesso linguaggio, cioè che usino un protocollo applicativo.

Gli utenti della rete quindi, se hanno il permesso, possono accedere alle risorse che il *server* mette a disposizione.

Il server di una rete deve essere gestito da un **amministratore di rete** che implementa anche le misure di sicurezza, il backup dei dati e l'accesso degli utenti alle risorse di rete.

L'amministratore si occupa dunque di fare copie di sicurezza dei dati in modo da garantire il ripristino se i dati dovessero andare persi.

Un sistema *client/server* funziona in questo modo:

1. Il *client* invia una richiesta ad un *server*.
2. Il *server* riceve la richiesta e quindi va in stato di ascolto (*listening*).
3. Interpreta la richiesta e la esegue.
4. Manda la risposta al *client*.
5. Il *client* riceve la risposta.

La modalità di comunicazione di un sistema *client-server* può essere di due tipi:

- **Unicast** dove il *server* comunica con un solo *client* per volta.
- **Multicast** dove il *server* comunica contemporaneamente con più *client*.

Se un server multicast riceve troppe richieste contemporaneamente può entrare in uno stato di congestione. In genere il *client* ha un indirizzo IP dinamico, mentre il *server* deve avere un indirizzo IP statico.

Nelle reti **peer to peer** i dispositivi sono collegati direttamente tra di loro senza l'utilizzo di altri dispositivi di rete. Le reti peer to peer presentano diversi svantaggi:

- ❖ In questo tipo di rete non c'è nessun controllo centrale di gestione della rete e non c'è neanche bisogno di assumere un amministratore di rete dedicato.
- ❖ Ogni utente è dunque responsabile di ciò che vuole condividere con gli altri e questo rende difficile controllare ciò che viene condiviso.
- ❖ Inoltre, non c'è nessun controllo sulla sicurezza, ogni computer utilizza le proprie misure per la protezione dei dati.
- ❖ La rete diventa sempre più complessa da gestire, soprattutto se il numero di computer è elevato. Le reti peer to peer infatti lavorano meglio in ambienti con meno di dieci computer.

Strutture dei sistemi operativi (Capitolo 2)

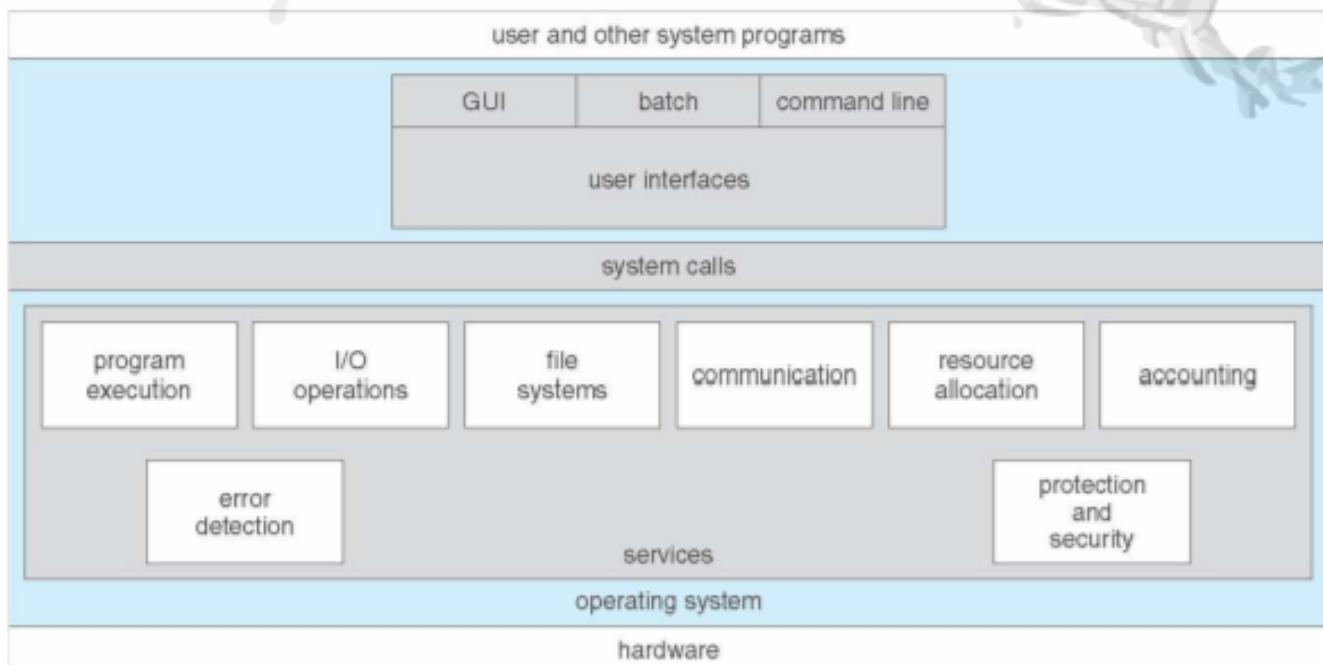
2.1 Servizi di un sistema operativo

Un sistema operativo offre un ambiente in cui eseguire programmi e fornire **servizi**. Ogni insieme di servizi offre funzionalità utili all'utente:

- ❖ **Interfaccia con l'utente (UI).** Può essere di diverse forme. Un'**interfaccia riga di comando (CLI)**, basata su stringhe che codificano i comandi.
Un'**interfaccia batch** prevede che comandi e direttive siano codificati nei file, eseguiti successivamente a lotti.
La forma più diffusa è l'**interfaccia grafica con l'utente (GUI)**, ossia un sistema grafico a finestre dotato di un dispositivo puntatore, il mouse, per comandare operazioni di I/O.
- ❖ **Esecuzione di un programma.**
- ❖ **Operazioni di I/O.** Di solito un utente non può controllare direttamente i dispositivi di I/O, quindi il sistema operativo deve offrire mezzi adeguati.
- ❖ **Gestione del file system.**
- ❖ **Comunicazioni.** Si può realizzare tramite una memoria condivisa o attraverso lo scambio di messaggi, in questo caso il sistema operativo trasferisce pacchetti di informazione tra i vari processi.
- ❖ **Rilevamento di errori.**

I sistemi con più utenti possono guadagnare in efficienza condividendo le risorse del calcolatore. In primo luogo, assegnando delle risorse necessarie ciascun utente.

Contabilizzando l'uso delle risorse registrando quali utenti usino il calcolatore, segnalando quali e quante risorse impieghino. E infine, assicurando protezione e sicurezza assicurando che l'accesso alle risorse del sistema sia controllato. La sicurezza di un sistema comincia con l'identificazione da parte di ciascun utente attraverso parole d'ordine.



2.1.2 Interprete dei comandi e GUI

Analizzando il primo aspetto dei servizi che ci offre il sistema operativo, incontriamo *l'interfaccia con l'utente*. Ci sono due modi per gli utenti di comunicare con il sistema operativo: uno si basa su un'*interfaccia a riga di comando* o **interprete dei comandi**, l'altro sfrutta un'**interfaccia grafica (GUI)**.

- 1) La **shell** (in italiano **interprete dei comandi**), è la componente fondamentale di un sistema operativo che permette all'utente il più alto livello di interazione con quest'ultimo. Tramite la *shell* è possibile impartire comandi e richiedere l'avvio di altri programmi.
- 2) La **GUI** è uno strumento più intuitivo, o *user-friendly*, della *shell*. È costituita da una o più finestre e dai relativi menu, inoltre è composta da un **desktop** che contiene le immagini o **icone** che rappresentano programmi, file, directory e funzioni del sistema.

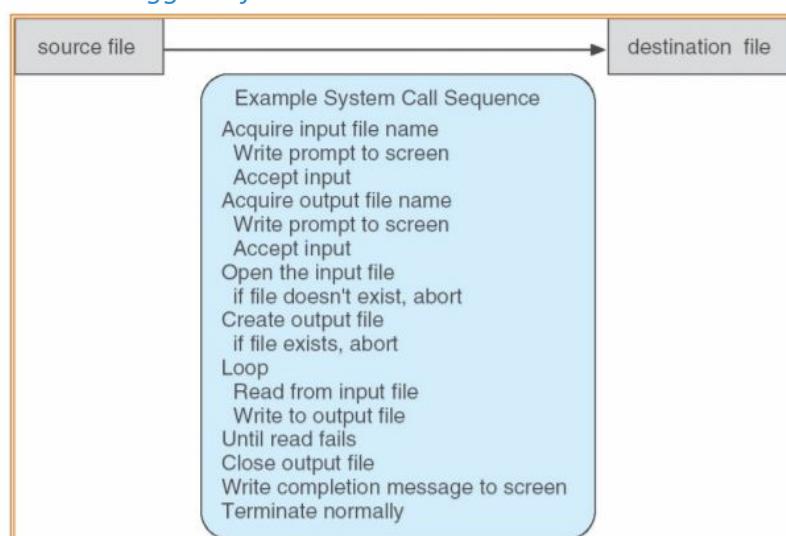
Le interfacce grafiche con l'utente si affacciarono sulla scena nei primi anni '70 dai laboratori di ricerca *Xerox PARC*, per poi avere una piena diffusione con l'avvento dei computer *Apple Macintosh* negli anni '80.

2.2 Chiamate di sistema

La **chiamate di sistema**, *System Call*, abbreviata in *Syscall*, è un metodo utilizzato dai programmi applicativi **per comunicare con il core del sistema**. Le chiamate di sistema fungono da interfaccia tra i programmi in esecuzione ed il sistema operativo: sono solitamente disponibili come speciali istruzioni *assembler* o come delle funzioni nei linguaggi C/C++. Questo metodo viene utilizzato quando un'applicazione o un processo utente deve trasmettere informazioni all'hardware o al kernel stesso. Questo tipo di chiamata è pertanto **l'anello di congiunzione tra la modalità utente (user mode) e la modalità kernel (kernel mode)**. Quindi la chiamata di sistema è necessaria ogni volta che un processo in esecuzione in modalità utente desidera eseguire una funzione che può essere eseguita **SOLO** in modalità kernel.

Fino a quando una *System Call* non è stata elaborata e i dati corrispondenti non sono stati trasmessi o ricevuti, **il core del sistema prende il controllo** del programma o del processo. In questo lasso di tempo l'esecuzione è interrotta. Non appena l'azione richiesta dalla chiamata di sistema è stata eseguita, il kernel abbandona il controllo e il codice del programma continua dal punto in cui è stata avviata la *Syscall*.

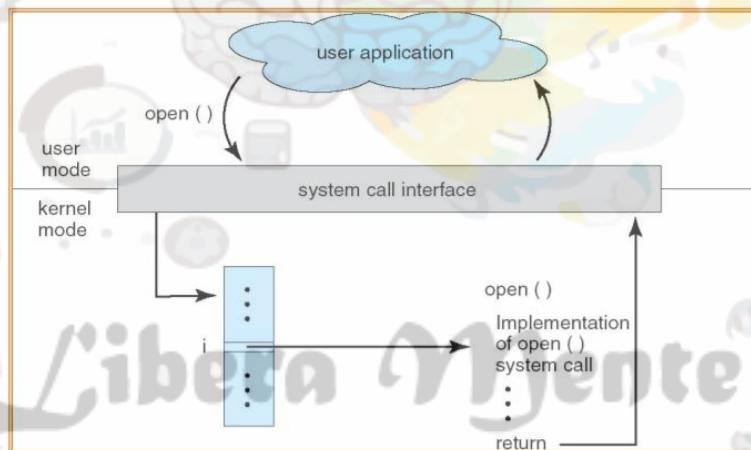
Esempio di un programma che legge un file e lo trascrive in un altro.



La maggior parte dei moderni sistemi operativi mette a disposizione alcune System Calls sotto forma di **funzioni di libreria**, che possono essere eseguite tramite un'**interfaccia di programmazione delle applicazioni (API)**. Le API consentono alle applicazioni o ai servizi di comunicare con altre applicazioni o servizi, semplificando notevolmente il lavoro degli sviluppatori di software poiché non è necessaria una conoscenza della loro implementazione. E' preferibile sfruttare l'API piuttosto che invocare direttamente le chiamate di sistema. Un vantaggio è legato alla portabilità delle applicazioni: un programma sviluppato sulla base di una certa API gira su qualunque sistema che la mette a disposizione, anche se le architetture sono diverse.

Ogni chiamata di sistema è codificata da un numero; il compilatore possiede una tabella delle chiamate di sistema, e per accedere si usano questi numeri come indici. L'interfaccia alle chiamate di sistema invoca di volta in volta la chiamata richiesta, che risiede nel kernel, e passa al chiamante i valori restituiti dalla chiamata di sistema.

Per passare i parametri al sistema operativo si usano tre metodi generali. Il più semplice consiste nel passare i parametri in registri; si possono però presentare casi in cui vi sono più parametri che registri in questo caso si memorizzano i parametri in un blocco di memoria e si passa l'indirizzo del blocco in un registro. Questo è il metodo usato dal sistema operativo Linux. Il programma può anche collocare (*push*) i parametri in una pila da cui sono prelevati (*pop*) dal sistema operativo.



2.2.1 Categorie di Syscall

Le *System Calls* sono classificabili in 5 categorie:

1. **Controllo dei processi.** Tutti i processi di un sistema informatico devono essere controllati, affinché possano essere interrotti in qualsiasi momento o essere pilotati da altri processi. A tal fine le *System Calls* di questa categoria controllano ad esempio l'avvio o l'esecuzione oppure lo stop o l'interruzione dei processi.

Quindi un programma in esecuzione deve potersi fermare in modo sia normale (*end*) sia anormale (*abort*). Se si ricorre ad una *System Call* per terminare in modo normale un programma in esecuzione, si ha la registrazione in un file di un'immagine del contenuto della memoria (*dump*) e l'emissione di un messaggio di errore. Un processo che segue un programma può richiedere di caricare (*load*) ed eseguire (*execute*) un altro programma.

Se al termine del nuovo programma il controllo rientra nel programma esistente, si deve salvare l'immagine della memoria del programma attuale. A questo scopo spesso si fornisce una *syscall* specifica, *create process* oppure *submit job*.

Può essere necessario terminare un processo creato se si riscontra che non è corretto o non serve (*terminate process*).

Per avviare un nuovo processo, la shell esegue la System Call *fork()*; si carica il programma in memoria tramite la System call *exec()* e infine si esegue il programma.

Completato il proprio compito, il processo esegue una chiamata di sistema *exit()* per terminare la propria esecuzione, riportando al processo chiamante un codice di stato 0 oppure un codice di errore diverso da 0.

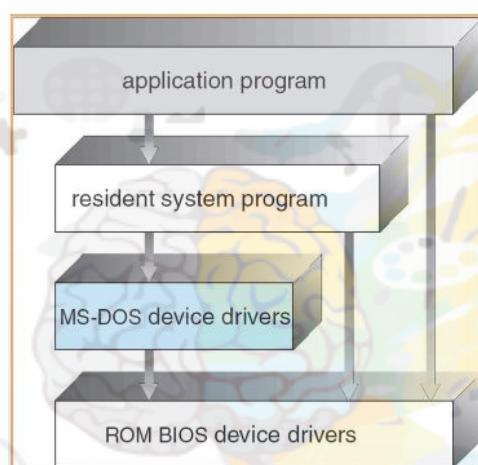
2. **Gestione dei file.** I programmi applicativi richiedono *System Calls* di questo tipo per ottenere l'accesso alle tipiche operazioni sui file.
3. **Gestione dei dispositivi.** La categoria “Gestione dei dispositivi” comprende tutte le *System Calls* che servono per richiedere o gestire le risorse hardware necessarie, come potenza di calcolo o spazio di archiviazione. Una volta richiesto e assegnato il dispositivo, è possibile leggervi (*read*), scrivervi (*write*) e procedere ad un riposizionamento (*reposition*).
4. **Gestione delle informazioni.** I processi sono collegati a molte informazioni, in cui la tempestività e l'integrità svolgono un ruolo importante. Per scambiarle o richiederle, i programmi applicativi utilizzano le *System Calls* di gestione e conservazione delle informazioni.
5. **Comunicazione tra processi.** I processi comunicano tra di loro tramite le *System Calls* attraverso due modi: il **modello scambio di messaggi** e quella **memoria condivisa**.
Nel **modello scambio di messaggi** i processi si scambiano i messaggi per il trasferimento delle informazioni. Prima di effettuare una comunicazione occorre aprire un collegamento. Tutti i calcolatori di una rete hanno un **nome di macchina** (*host name*), analogamente ogni processo ha un **nome di processo**, che si converte in un identificatore. La conversione si compie con le chiamate di sistema *get hostid* e *get processid*. Questi identificatori sono passati alle chiamate di sistema *open* e *close* messe a disposizione dal file system, oppure alle chiamate di sistema *open connection* e *close connection*. Il processo ricevente deve acconsentire alla comunicazione con una chiamata di sistema *accept connection*, mentre la chiamata di sistema *close connection* pone fine alla comunicazione.
Nel **modello a memoria condivisa**, i processi usano chiamate di sistema *shared memory* e *shared memory attach* per creare e accedere alle aree di memoria possedute da altri processi. Il sistema operativo tenta di impedire a un processo l'accesso alla memoria di un altro processo, ma il modello a memoria condivisa richiede che più processi coordino nel superare questo limite.
Lo scambio di messaggi è utile quando è necessario trasferire una piccola quantità di dati; la condivisione della memoria, invece, permette la massima velocità nelle comunicazioni e in ogni caso sussistono sempre i problemi riguardanti la protezione e la sincronizzazione tra processi che condividono la memoria.

Processi	Dispositivi I/O	Comunicazioni
<ul style="list-style-type: none">➢ end, abort➢ load, execute➢ create process, terminate process➢ get process attribute, set process attributes➢ wait for time➢ wait event, signal event➢ allocate memory, free memory	<ul style="list-style-type: none">➢ request device, release device➢ read, write, reposition➢ get device attribute, set device attributes➢ attach device, detach device	<p>Scambio di messaggi</p> <ul style="list-style-type: none">➢ gethostid – IP address➢ getprocessid – PID➢ open connection➢ accept connection➢ send/receive➢ close connection
File	Informazioni	Memoria condivisa
<ul style="list-style-type: none">➢ create file, delete file➢ open, close➢ read, write, reposition➢ get file attribute, set file attributes	<ul style="list-style-type: none">➢ get time, get date, set time, set date➢ get system data, set system data➢ get process, file, or device attributes➢ set process, file, or device attributes	<p>Memoria condivisa</p> <ul style="list-style-type: none">➢ shared memory create➢ shared memory attach

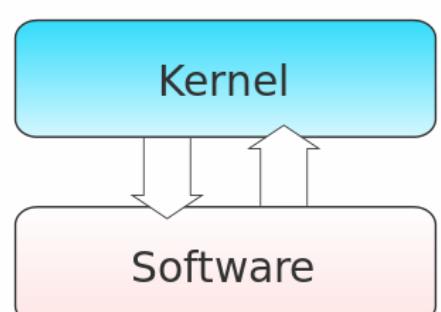
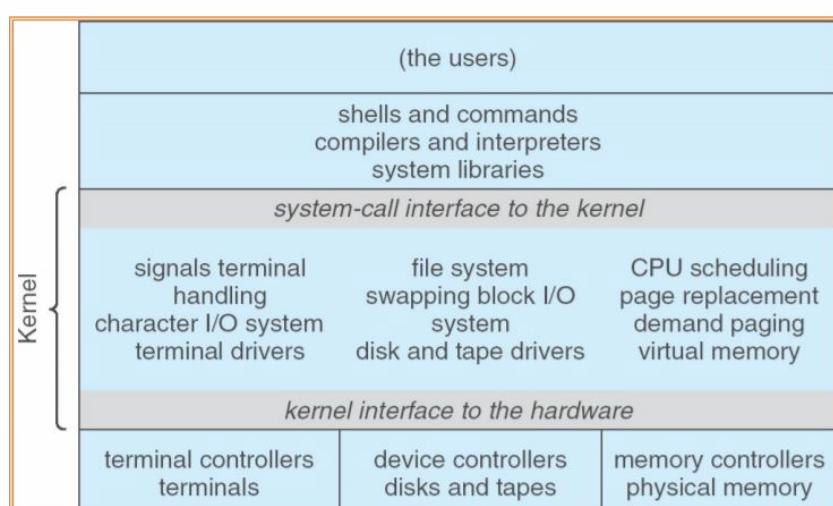
2.3 Struttura (semplice) del sistema operativo

Un tipico sistema operativo **semplice** con una struttura non ben definita fu **MS-DOS**. Non fu suddiviso in moduli poiché, a causa dei limiti dell'architettura, lo scopo prioritario era di fornire la massima funzionalità nel minimo spazio.

In MS-DOS non vi è una separazione tra le interfacce e i livelli di funzionalità, infatti le applicazioni accedevano direttamente alle routine (*gestione di interrupt*) di sistema per l'I/O. Il suo processore, Intel 8088 a 16 bit e 1 MB di memoria e una capacità di indirizzamento a 20 bit, non distingueva *user mode* da *kernel mode* e non offriva la giusta protezione hardware. Ci sono state quindi tre generazioni di processori non ottimali, ma solo con l'avvento del processore a 32 bit (come in Macintosh) le cose cambiarono in meglio e in particolare con Windows XP.



Anche il sistema **UNIX** originale è poco strutturato, a causa delle limitazioni hardware. Il sistema consiste in due parti separate: il kernel e i programmi di sistema. A sua volta, il kernel è diviso in una serie di interfacce e driver dei dispositivi. Tutto ciò che sta al di sotto dell'interfaccia alle chiamate di sistema e al di sopra dell'hardware costituisce il kernel. Esso comprende il file system, lo scheduling della CPU, la gestione della memoria, e altre funzionalità rese disponibili tramite System Calls. Questa è una struttura **monolitica** che rendeva difficile implementazione la manutenzione. Lo svantaggio dei **kernel monolitici** è di rendere impossibile aggiunta di un nuovo dispositivo hardware senza aggiungere il relativo modulo al kernel, operazione che richiede la ricompilazione del kernel.



Kernel Monolitico

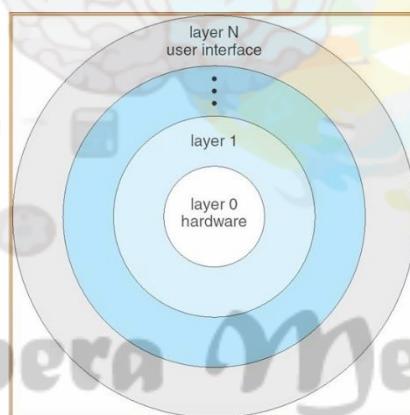
2.3.1 Struttura (stratificata) del sistema operativo

In presenza di hardware appropriato, i sistemi operativi possono essere suddivisi in **moduli** più piccoli e gestibili. Vi sono molti modi per rendere un sistema operativo modulare. Uno di loro è il **metodo stratificato**, secondo il quale il sistema è suddiviso in un certo numero di livelli o strati: il più basso corrisponde all'hardware (*strato 0*), il più alto all'interfaccia con l'utente (*strato N*).

Il vantaggio principale offerto da questo metodo è dato dalla semplicità di progettazione e dalle funzionalità di debug. Gli strati sono composti in modo che ciascuno di essi usi solo funzioni e servizi che appartengono a strati di livello inferiore. Se si riscontra un errore, questo deve trovarsi in quello strato, perché gli strati inferiori sono già stati corretti; quindi la suddivisione in strati semplifica la progettazione e la realizzazione di un sistema. Ogni strato si realizza impiegando le operazioni messe a disposizione dagli strati inferiori. I livelli aumentano con l'esecuzione di vari **player** (*programmi utente*) lanciati da una **macchina virtuale**.

Un problema che risiede nella struttura stratificata è che essa tende a essere meno efficiente delle altre; per esempio, per eseguire un'operazione di I/O un programma utente invoca una chiamata di sistema che è intercettata dallo strato di I/O che, a sua volta, esegue una chiamata allo strato di gestione della memoria, che a sua volta richiama lo strato di scheduling della CPU e che quindi è passata all'opportuno dispositivo di I/O (*strato 0, hardware*). Ciascuno strato aggiunge un carico alla chiamata di sistema.

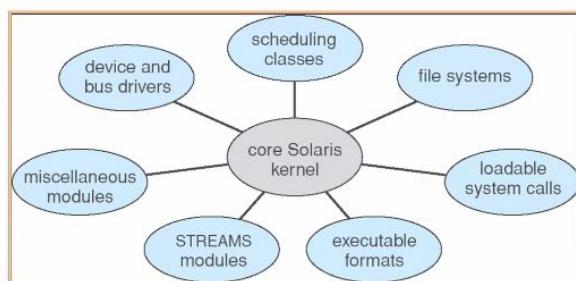
Attualmente si progettano sistemi basati sul numero inferiore di strati però con più funzioni.



2.3.2 Struttura (modulare) del sistema operativo

Il miglior approccio per la progettazione dei sistemi operativi si fonda su tecniche della programmazione orientata agli oggetti per implementare un **kernel modulare**. L'utilizzo di moduli caricati dinamicamente viene sfruttato da UNIX (come Solaris), Linux e Mac OS X.

Questa organizzazione lascia la possibilità al kernel di fornire i servizi essenziali, ma permette anche di implementare dinamicamente certe caratteristiche. E' più flessibile dei sistemi a strati perché ciascun modulo può invocare funzionalità di un qualunque altro modulo. Inoltre, come nei **microkernel**, il modulo principale gestisce solo i servizi essenziali, oltre a poter caricare altri moduli e comunicare con loro. Inoltre, il modello modulare è più efficiente perché i moduli sono in grado di comunicare senza invocare la funzionalità di trasmissione dei messaggi.

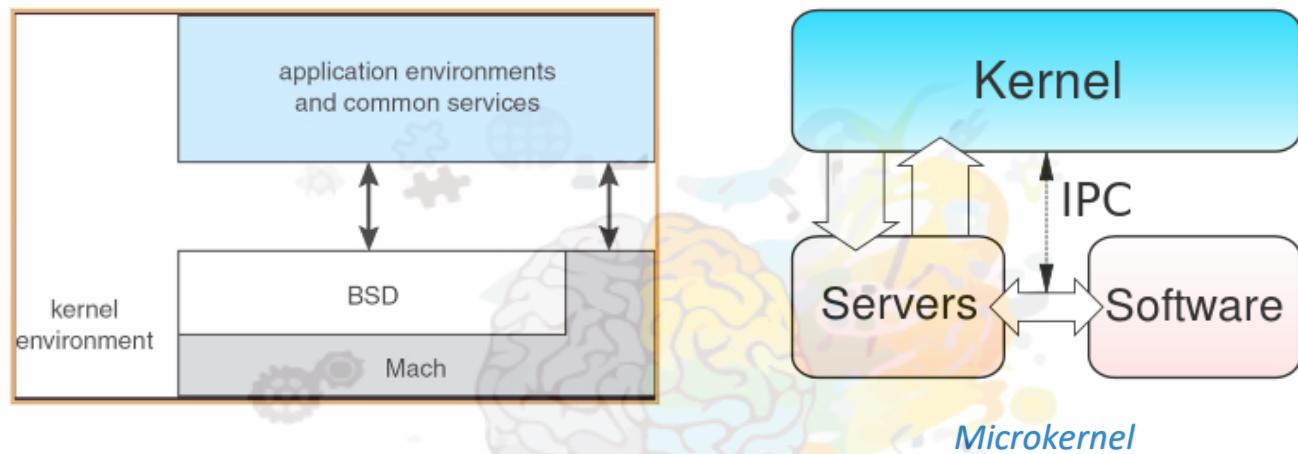


Il sistema operativo Mac OS X adotta una **struttura ibrida**: è organizzato in strati, uno dei quali contiene il *microkernel Mach*. Gli strati superiori comprendono un insieme di servizi e un'interfaccia grafica per le applicazioni. Il kernel si trova in uno strato sottostante, ed è costituito dal *microkernel Mach* e dal *kernel BSD*.

Il primo cura la gestione della memoria, le chiamate di procedure remote, la comunicazione tra processi e lo scheduling dei thread.

Il secondo mette a disposizione un'interfaccia BSD a riga di comando, i servizi legati al file system e alla rete, e le API POSIX.

Le applicazioni e i servizi comuni possono accedere direttamente sia ai servizi di Mach sia quelli di BSD.



2.3.3 Microkernel

Verso la metà degli anni '80 un gruppo di ricercatori progettò e realizzò un sistema operativo, **Mach** (*multiprogrammato* e *multithread*), col kernel strutturato in moduli secondo il cosiddetto orientamento a **microkernel**. Rimossero dal kernel tutti i componenti non essenziali, realizzandoli come programmi di livello utente e di sistema.

Un *microkernel* (è un kernel ridotto all'osso) offre i servizi minimi di gestione dei processi, della memoria e di comunicazioni.

Lo scopo principale del *microkernel* è fornire funzioni di comunicazione tra i programmi client, secondo il modello scambio di messaggi, e vari servizi.

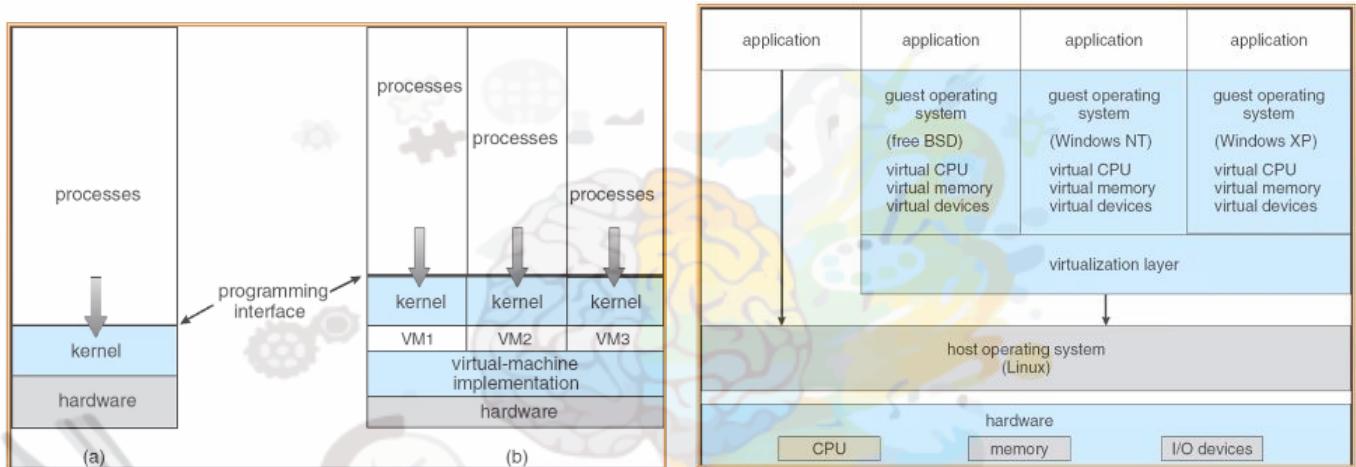
Uno dei vantaggi del *microkernel* è il fatto che i nuovi servizi si aggiungono allo spazio utente e non comportano modifiche al kernel: se il kernel deve essere modificato i cambiamenti da fare sono circoscritti offrendo sicurezza e affidabilità, poiché i servizi si eseguono in gran parte come processi utenti. Un problema dei *microkernel* è che possono incorrere in cali di prestazioni dovuti al sovraccarico indotto dall'esecuzione dei processi utente con funzionalità di sistema.

2.4 Macchine virtuali

La **macchina virtuale (VM)** è un software che, attraverso un processo di virtualizzazione, crea un ambiente virtuale che emula il comportamento di una macchina fisica (PC, client o server) grazie all'assegnazione di risorse hardware.

Tramite lo scheduling della CPU e la memoria virtuale, il sistema operativo può dare l'impressione che ogni processo sia dotato del proprio processore e del proprio spazio di memoria (virtuale). Ogni **processo che viene ospitato** può usufruire di una copia virtuale del calcolatore reale: solitamente il processo ospite è un sistema operativo. In questo modo una singola macchina fisica può far girare più

sistemi operativi, ciascuno sulla sua macchina virtuale. Il vantaggio è che se dovesse andare fuori uso il sistema operativo che gira sulla macchina virtuale, il sistema di base non ne risentirebbe affatto. Il componente centrale e più importante di un sistema basato sulle machine virtuali è l'**hypervisor**, conosciuto anche come *virtual machine monitor (VMM)*. Il suo ruolo è quello di gestire le risorse hardware mettendole a disposizione alle varie macchine virtuali (concetto di **paravirtualizzazione**). Una macchina virtuale è difficile da realizzare poiché bisogna produrre un esatto duplicato della macchina sottostante che ha due modalità, utente e di sistema. I programmi che realizzano il sistema di macchine virtuali si possono eseguire nel kernel mode, mentre ciascuna macchina virtuale può funzionare solo nella *user mode*. Quindi come la macchina fisica, anche quella virtuale possiede due modalità: **virtual user mode** e **virtual kernel mode**.



Architettura VMware

2.5 Tipi di sistema operativo

I **mainframe** sono elaboratori dotati di elevata capacità di elaborazione.

Sui primi mainframe erano installati **sistemi operativi batch** (*a lotti*), ovvero sistemi in cui per accelerare l'elaborazione, task simili venivano raggruppati in lotti ed eseguiti. Nel caso dei sistemi batch il compito del sistema operativo era quello di trasferire il controllo da un lavoro al successivo ed era sempre residente in memoria.

L'evoluzione dei mainframe ha portato ai **sistemi operativi time sharing** (*a partizione di tempo*), che avevano un approccio interattivo coi computer, in cui lo stesso computer veniva usato da più utenti, ciascuno con il proprio terminale. Il controllo della CPU veniva passato rapidamente da un programma di un utente all'altro, dando l'impressione a tutti gli utenti di avere ognuno la propria CPU.

Si iniziò poi a parlare anche di **sistemi operativi real time**, ovvero sistemi dove il tempo di risposta è cruciale e specificato, da un punto di vista teorico in un *sistema RT* non è importante la velocità ma piuttosto il tempo di risposta.

Processi (Capitolo 3)

3.1 Concetto di processo

Un programma caricato in memoria e pronto per l'esecuzione è noto come **processo**. Un processo in esecuzione richiede determinate risorse, come tempo di elaborazione della CPU, memoria, file e dispositivi di I/O. Un processo ha una capacità di indirizzamento di **48 bit** su un *processore di 64 bit*. Un sistema è formato da processi del sistema operativo, che esegue il codice di sistema, e da processi utenti che eseguono il codice utente.

Il sistema operativo è responsabile della **gestione dei processi** e dei **thread** di sistema e utenti.

Un **sistema batch** esegue i **lavori (job)**, mentre un sistema multitasking esegue programmi utenti o task.

Un processo comprende:

- l'attività corrente, rappresentata dal valore del **PC (Program Counter)** e dal contenuto dei registri della CPU;
- una pila (**stack**) contenente i dati temporanei;
- l'**heap** contenente le variabili globali.

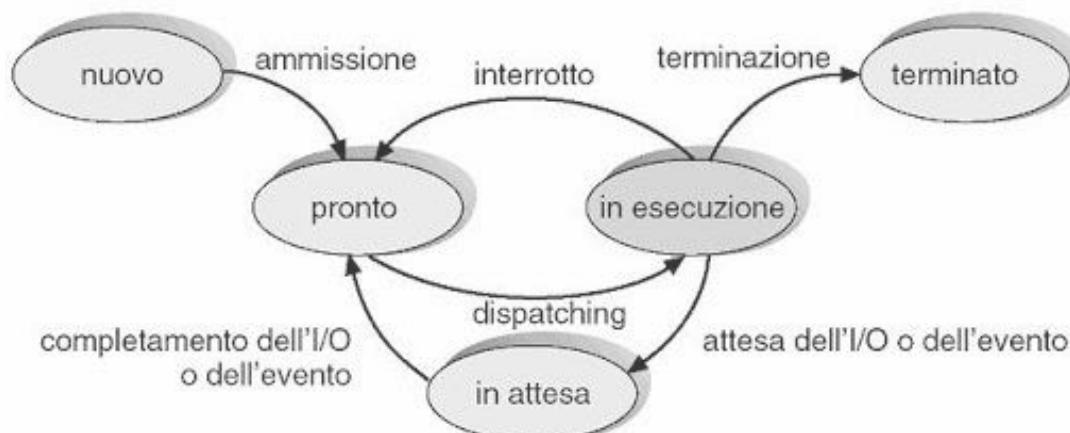
Un **programma** di per sé non è un processo ma è un'**entità passiva**, come il contenuto di un file memorizzato in un disco, mentre un processo è un'**entità attiva**, con un **PC** che specifica qual è l'istruzione successiva da eseguire. Un programma diventa un processo quando è caricato in memoria.

3.1.2 Stato del processo

Un processo durante l'esecuzione è soggetto a **cambiamenti di stato**:

- ❖ **Nuovo**. Si crea il processo.
- ❖ **Esecuzione**. La CPU esegue le istruzioni del relativo programma.
- ❖ **Attesa**. Il processo attende che si verifichi qualche evento, come la ricezione di un segnale o il completamento di un'operazione di I/O.
- ❖ **Pronto**. Il processo attende di essere assegnato ad un'unità di elaborazione.
- ❖ **Terminato**. Il processo ha terminato l'esecuzione.

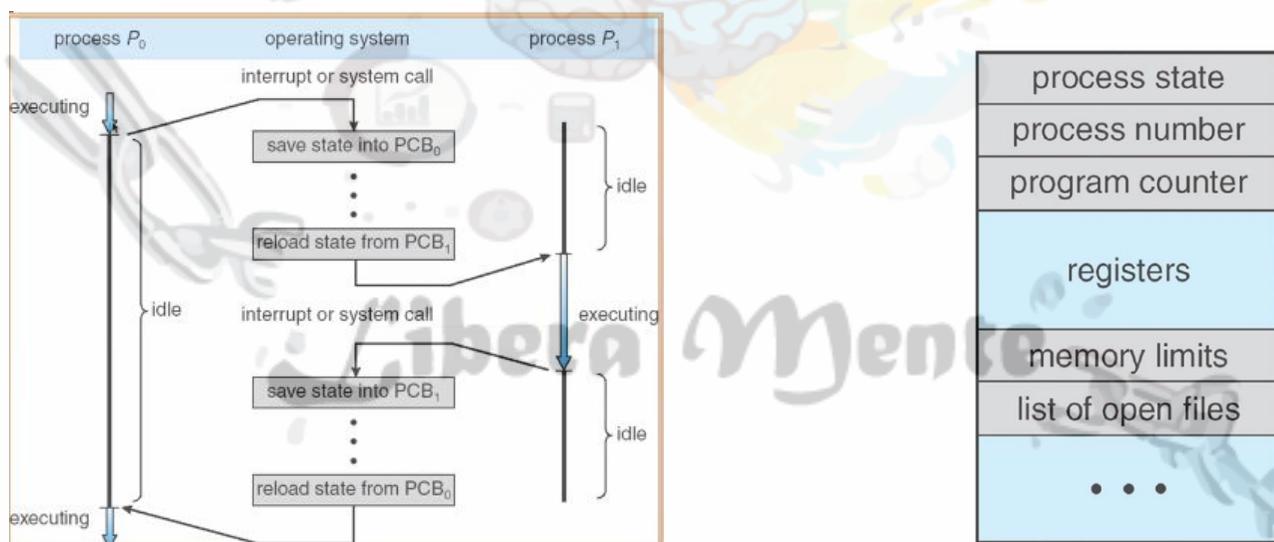
In ciascuna unità di elaborazione può essere in esecuzione un solo processo per volta, gli altri possono essere *pronti* o nello stato di *attesa*. Un **automa a stati finiti** è una macchina dove è possibile individuare uno stato iniziale e uno finale.



3.1.3 Blocco di controllo dei processi (PCB)

Ogni processo è raggruppato nel sistema operativo da un **blocco di controllo di un processo** (*process control block, PCB, o task control block, TCB*). Il **PCB** si usa come deposito per tutte le informazioni relative ai vari processi. Un *blocco di controllo* contiene:

- ❖ **Stato del processo.** Lo stato può essere: nuovo, pronto, esecuzione, attesa, arresto, terminato.
- ❖ **Program counter.** Il PC contiene l'indirizzo della successiva istruzione dai seguire per tale processo.
- ❖ **Registri di CPU.** I registri variano in numero e tipo secondo l'architettura del calcolatore. Essi comprendono accumulatori, vari registri, puntatori alla cima dello *stack*. Quando si verifica un'interruzione della CPU, si devono salvare tutte le informazioni insieme al PC, in modo da permettere la corretta esecuzione del processo in un secondo momento.
- ❖ **Informazioni sullo scheduling di CPU.** Queste informazioni comprendono la priorità del processo, i puntatori alle code di scheduling e tutti gli altri parametri di scheduling.
- ❖ **Informazioni sulla gestione della memoria.**
- ❖ **Informazioni di contabilizzazione delle risorse.** Queste informazioni comprendono il tempo d'uso della CPU e il tempo reale d'utilizzo della stessa, i numeri dei processi, e così via.
- ❖ **Informazioni sullo stato dell'I/O.** Queste informazioni comprendono la lista dei dispositivi I/O assegnati a un determinato processo.



(PCB)

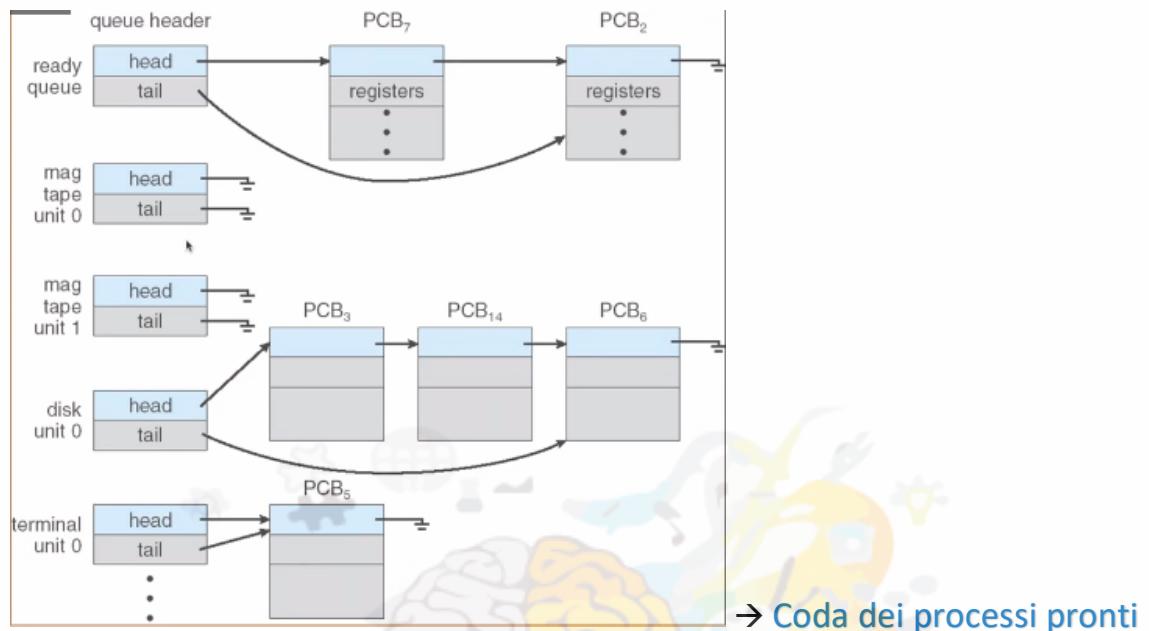
3.1.4 Thread

L'unico **percorso di esecuzione di un processo** che si sta eseguendo, è detto **thread**. In molti sistemi operativi moderni si è esteso il concetto di processo introducendo la possibilità di avere più percorsi di esecuzioni (**multithread**), in modo da permettere che un processo possa svolgere più di un compito alla volta.

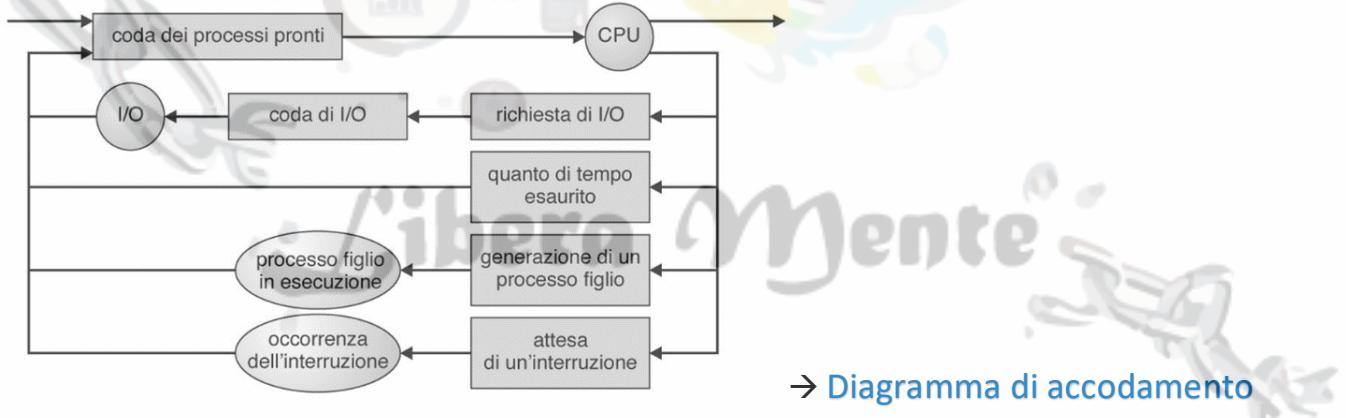
3.2 Code di scheduling

Lo **scheduler dei processi** seleziona un processo da eseguire da un insieme di processi disponibili. Ogni processo è inserito in una **coda di processi**, composta da tutti i processi del sistema. I processi presenti in memoria centrale si trovano in una lista detta **coda dei processi pronti** (*ready queue*), tale coda è memorizzata come una lista concatenata. Il sistema operativo ha anche altre code.

L'elenco dei processi che attendono la disponibilità di un particolare dispositivo di I/O, si chiama **coda del dispositivo**; ogni dispositivo ha la propria coda.



Il **diagramma di accodamento** rappresenta lo scheduling dei processi: ogni riquadro rappresenta una coda. Sono presenti due tipi di coda: la *coda dei processi pronti* e un insieme di *code di dispositivi*. I cerchi rappresentano le risorse che servono le code, le frecce indicano il flusso dei processi.



Un nuovo processo si colloca prima nella *coda dei processi pronti*, finché non è selezionato per poi essere eseguito (**dispatched**). Una volta che il percorso è assegnato alla CPU ed è nella fase d'esecuzione, si può verificare uno dei seguenti eventi:

- Il processo può emettere una richiesta di I/O e quindi essere messo in una coda di I/O;
- Il processo può creare un nuovo processo e attendere la terminazione;
- Il processo può essere rimosso forzatamente dalla CPU a causa di un'interruzione, ed essere reinserito nella coda dei processi pronti.

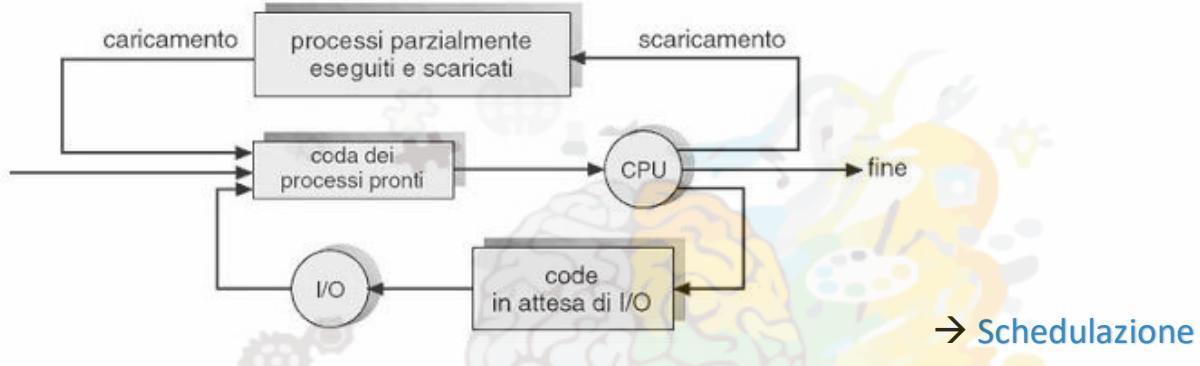
Nei prime due casi, il processo passa dallo stato di *attesa* allo stato *pronto* ed è nuovamente inserito nella *coda dei processi pronti*. Un processo continua questo ciclo fino al termine della sua esecuzione: a questo punto viene allontanato da tutte le code, rimosso il suo PCB e tolte le varie risorse.

3.2.1 Scheduler

Nel corso della sua esistenza, un processo si trova in varie code di scheduling gestite dal sistema operativo che compie una selezione per mezzo di uno **scheduler**. In un **sistema batch**, accade che ci sono troppi processi da eseguire, per questo i vari *jobs* si trasferiscono in dispositivi di memoria secondaria, generalmente dischi, e restano lì fino al momento dell'esecuzione.

Lo **scheduler a lungo termine (job scheduler)**, sceglie i lavori da questo insieme e li carica in memoria affinché siano eseguiti.

Lo **scheduler a breve termine** sceglie i lavori già pronti per l'esecuzione e assegna la CPU ad uno di loro. Questi due *scheduler* si differenziano principalmente per la frequenza con la quale sono eseguiti.



Lo *scheduler a lungo termine* controlla il numero di processi presenti in memoria, inoltre lo si può richiamare solo quando un processo abbandona il sistema. È importante che lo *scheduler a lungo termine* faccia un'accurata selezione dei processi poiché le prestazioni migliori sono date da una combinazione equilibrata di processi con prevalenza di I/O e processi con prevalenza di elaborazione. I sistemi multitasking, come UNIX e mi Windows, sono spesso privi di *scheduler a lungo termine*, infatti caricano in memoria tutti i nuovi processi che sono gestiti dallo *scheduler a breve termine*.

In alcuni sistemi operativi multitasking, si può introdurre un livello di *scheduling intermedio*, detto **scheduler a medio termine**. Questo tipo di scheduler può essere vantaggioso quando si eliminano processi dalla memoria, riducendo il grado di multiprogrammazione del sistema.

Con lo **swapping**, il processo viene rimosso e successivamente caricato in memoria dallo scheduler a medio termine.

3.2.2 Cambio di contesto

Il **cambio di contesto (context switch)** è un'operazione del sistema operativo che conserva lo stato del processo, in modo da poter essere ripreso in un altro momento. Questa attività permette a più processi di condividere la CPU, ed è anche una caratteristica essenziale per i sistemi operativi multitasking. Il **contesto** è rappresentato all'interno del *PCB* del processo, e comprende i valori dei registri della CPU, lo stato del processo e informazioni relative alla gestione della memoria.

Nel *cambio di contesto*, il sistema salva nel suo *PCB* il contesto del processo uscente e carica il contesto del processo successivo, salvato in precedenza. Il cambio di contesto comporta un calo delle prestazioni, perché il sistema esegue solo operazioni volte alla corretta gestione dei processi e non alla computazione. E' meglio evitare un *context switch* poiché è puro **overhead**.

In presenza di un'interruzione, il sistema deve salvare il *contesto del processo* corrente. Il procedimento consiste in un **salvataggio dello stato** corrente della CPU e in un **ripristino dello stato** per poter riprendere l'elaborazione dal punto in cui era stata interrotta.

3.3 Creazione di un processo

Durante la propria esecuzione, un processo può creare numerosi nuovi processi tramite un'apposita system call (`create_process` in Windows). Il processo creante si chiama **processo genitore**, mentre il nuovo processo si chiama **processo figlio**. Ciascuno di questi nuovi processi può creare a sua volta altri processi, formando un **albero di processi**. La maggior parte dei sistemi operativi identifica un processo per mezzo di un numero intero univoco, detto **identificatore** del processo o **pid** (*process identifier*).

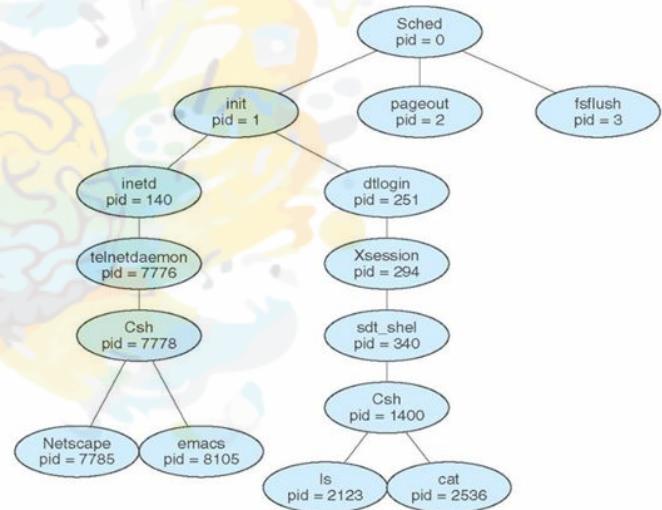
È facile costruire un albero identificando ricorsivamente i processi genitori fino a giungere a **init**. Per eseguire il proprio compito, un processo ha bisogno di alcune risorse. Quando un processo crea un sotto-processo, quest'ultimo può essere in grado di ottenere le proprie risorse direttamente dal sistema operativo, oppure può essere vincolato dal processo genitore. Il processo genitore può dividere le proprie risorse tra i suoi processi figli in modo da evitare che un processo sovraccarichi il sistema creando troppi sotto processi.

Quando un processo ne crea uno nuovo:

- il **processo genitore** continua l'esecuzione insieme ai propri processi figli;
- il **processo genitore** attende che alcuni o tutti i suoi processi figli terminano.

Ci sono due possibilità anche per quanto riguarda lo spazio di indirizzi del nuovo processo:

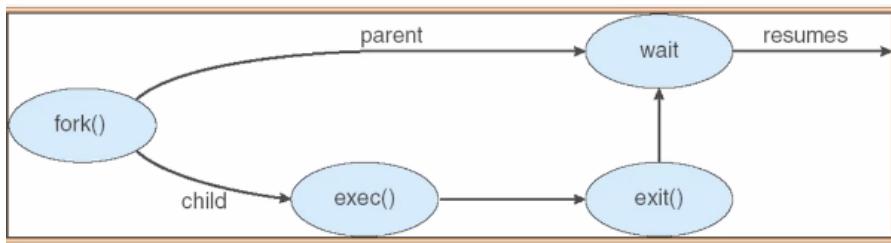
- il **processo figlio** è un duplicato del processo genitore;
- nel **processo figlio** si carica un nuovo programma.



Nel sistema operativo **UNIX**, ogni processo ha un proprio **pid** e la creazione di un nuovo processo avviene tramite la *system call fork()*: la modalità attraverso cui un processo crea in memoria una copia di sé stesso; la copia prenderà il nome di **processo figlio**, mentre il processo originale verrà chiamato **processo padre**. Il nuovo processo è composto di una copia dello spazio degli indirizzi del processo genitore. Questo meccanismo permette al processo genitore di comunicare con il proprio processo figlio.

Il processo padre ed il processo figlio possono scegliere le istruzioni da eseguire tramite il valore di ritorno della *system call fork()*. Il valore di ritorno della chiamata `fork()` vale **0** nel **processo figlio**, un numero **> 0** nel **processo padre** (il valore restituito è proprio il PID del figlio) o un valore **< 0** nel caso in cui non sia stato possibile creare un nuovo processo. Tramite il valore riportato del *pid*, si conosce qual è il processo padre e qual è il processo figlio.

Generalmente, dopo una *system call fork()*, uno dei due processi impiega una *system call exec()* sostituisce l'immagine in memoria del processo con un nuovo programma. Il processo genitore può anche generare più processi figli, oppure, se durante l'esecuzione del processo figlio non ha nient'altro da fare, può invocare la *system call wait()* per rimuovere sé stesso dalla coda dei processi pronti fino alla terminazione del figlio. Usando la *system call execvp()*, il processo figlio sovrappone il proprio spazio di indirizzi con il comando `/bin/ls` di UNIX (che si usa per ottenere l'elenco del contenuto di una directory). Utilizzando la *system call wait()*, il processo genitore attende che il processo figlio termini. Quando ciò accade, il processo genitore chiude la propria fase di attesa e termina usando la *system call exit()* e le risorse impiegate dal processo sono recuperate dal SO.



Esempio:

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid;
    pid = fork(); /* genera un nuovo processo */

    if (pid < 0) { /* errore */
        fprintf (stderr, "generazione del nuovo processo fallita");
        return 1;
    }
    else if (pid == 0) { /* processo figlio */
        execlp ("/bin/ls", "ls", NULL);
    }
    else { /* processo genitore attende il completamento del figlio */
        wait (NULL);
        printf ("il processo figlio ha terminato");
    }
    return 0;
}

```

3.3.1 Terminazione di un processo

Un **processo termina** quando finisce l'esecuzione della sua ultima istruzione e inoltra la richiesta al sistema operativo di essere cancellato usando la *system call exit ()*, a questo punto, il processo figlio può riportare alcuni dati al processo genitore, che li riceve attraverso la *system call wait ()*. Tutte le risorse del processo sono liberate dal sistema operativo.

Solo un processo genitore può causare la terminazione di un altro altrimenti gli utenti potrebbero causare la terminazione forzata dei processi di chiunque. Il processo genitore deve conoscere l'identità dei propri processi figli, perciò quando un processo ne crea uno nuovo, l'identità del nuovo processo viene passata al processo genitore.

Un processo genitore può porre termine l'esecuzione di uno dei suoi processi figli per diversi motivi:

- il processo figlio ha usato troppe risorse che gli sono state assegnate;
- il compito assegnato al processo figlio non è più richiesto;
- il processo genitore termina il sistema operativo non consente un processo figlio di continuare l'esecuzione.

Se un processo genitore termina, tutti i suoi processi figli sono affidati al processo **init ()**, che assume il ruolo di nuovo genitore.

3.4 Comunicazione tra processi (IPC)

I processi possono essere **indipendenti** o **cooperanti**. Un *processo è indipendente* se non influisce su altri processi del sistema o subirne l'influsso. Un *processo è cooperante* se influenza o può essere influenzato da altri processi in esecuzione nel sistema.

La cooperazione tra processi può essere utile per diverse ragioni:

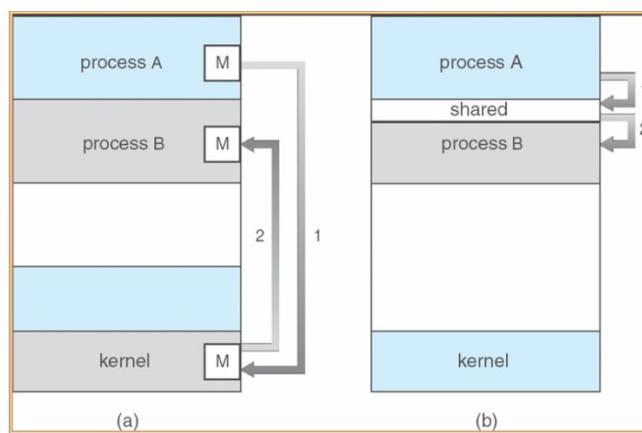
- **Condivisione di informazioni.** Poiché più utenti possono essere interessati alle stesse informazioni.
- **Accelerazione del calcolo.** Alcune attività di elaborazione sono realizzabili più rapidamente se si suddividono in sotto attività eseguibili in parallelo.
- **Modularità.**
- **Convenienza.** Un solo utente può avere la necessità di compiere più attività contemporaneamente.

Per lo scambio di dati e informazioni, i **processi cooperanti** necessitano di un meccanismo di comunicazione tra processi (**IPC**, *interprocess communication*). I modelli della comunicazione tra processi sono due: a **memoria condivisa** e **scambio di messaggi**.

Nel modello **memoria condivisa**, si stabilisce una zona di memoria condivisa dai processi che possono comunicare scrivendo e leggendo da questa zona. Questo metodo è più veloce dello scambio di messaggi, e inoltre ha bisogno dell'intervento del kernel solo per allocare le regioni di memoria condivisa.

Il secondo tipo di modello la comunicazione luogo tramite **scambio di messaggi** tra i processi e questo metodo è utile per trasmettere piccole quantità di dati; non c'è bisogno di evitare conflitti ed è anche più facile da implementare in un sistema distribuito. Lo svantaggio è l'implementazione tramite system call che impegnano il kernel.

Sui sistemi con più core di elaborazione lo scambio di messaggi fornisce prestazioni migliori rispetto alla memoria condivisa. **UNIX** prevede la *IPC* sia tramite memoria condivisa che scambio di messaggi.



3.4.1 Sistemi a memoria condivisa

La comunicazione basata sulla **condivisione della memoria** richiede che i processi comunicanti allochino una zona di memoria condivisa, di solito si trova nello spazio degli indirizzi del processo che la alloca: gli altri processi che desiderano usarla per comunicare dovranno acquistarla nel loro spazio degli indirizzi. Di solito, il sistema operativo tenta di impedire a un processo l'accesso alla memoria di altri processi, per questo due o più processi devono raggiungere un accordo per superare questo limite.

L'esecuzione dei due processi richiede la presenza di un **buffer** che possa essere riempito dal produttore e svuotato dal consumatore. Il buffer si troverà in una zona di memoria condivisa dai due processi. I due processi devono essere sincronizzati in modo tale che il consumatore non tenti di consumare un'unità ancora non prodotta.

Sono utilizzabili due tipi di buffer: quello **illimitato** e quello **limitato**.

In quello **limitato** il consumatore deve attendere che il buffer sia vuoto, viceversa, il produttore deve attendere che il buffer sia pieno.

Il buffer condiviso è realizzato come un array circolare con due puntatori logici: *inserisci* e *preleva*. La variabile *inserisci* indica la successiva posizione libera nel buffer; *preleva* indica la prima posizione piena nel buffer.

Il buffer è **vuoto** se *inserisci == preleva*; è **pieno** se $((inserisci + 1) \% \text{DIM_BUFFER}) == \text{preleva}$.

In **UNIX** la memoria condivisa è organizzata utilizzando i file mappati in memoria, che associano la regione di memoria condivisa ad un file

3.4.2 Sistemi a scambio di messaggi

Lo **scambio di messaggi** è un meccanismo che permette a due o più processi di comunicare e di sincronizzarsi senza condividere lo stesso spazio degli indirizzi. È una tecnica utile perché i processi possono trovarsi su due macchine diverse connesse da una rete.

Un meccanismo per lo scambio di messaggi deve prevedere almeno due operazioni: **send** e **receive**.

Inoltre, i messaggi possono avere lunghezza **fissa** o **variabile**.

Per inviare e ricevere messaggi, esiste un **canale di comunicazione** (*communication link*), realizzabile in molti modi:

- Comunicazione **diretta** o **indiretta**;
- Comunicazione **sincrona** o **asincrona**;
- Gestione automatica o esplicita del buffer.

● Con la **comunicazione diretta**, ogni processo che vuole comunicare deve nominare il **ricevente** o il **trasmittente** della comunicazione:

- *send (P, messaggio)*, invia il messaggio al processo P;
- *receive (Q, messaggio)*, riceve un messaggio dal processo Q.

Ci sono due tipi di modalità di comunicazione diretta: **modalità simmetrica** e **modalità asimmetrica**.

Nello schema **simmetrico** il trasmittente e il ricevente devono nominarsi a vicenda per poter comunicare. Nello schema **asimmetrico** il trasmittente nomina il ricevente, mentre il ricevente non deve nominare nessuno. In questo caso:

- *send (P, messaggio)*, invia il messaggio al processo P;
- *receive (id, messaggio)*, riceve un messaggio da qualsiasi processo; nella variabile id si riporta il nome del processo con cui è avvenuta la comunicazione.

Gli schemi *simmetrico* e *asimmetrico* hanno lo **svantaggio** di una limitata modularità.

● Con la **comunicazione indiretta**, i messaggi si inviano a delle **porte** (dette **mail-box**), identificate da un id univoco, che li ricevono. Due processi possono comunicare solo se condividono una porta; in quel caso si stabilisce un canale di comunicazione. Un canale può essere associato a più di due processi; tra ogni coppia di processi comunicanti possono esserci più canali diversi, ognuno corrisponde ad una porta. Una porta può appartenere al **processo** o al **sistema**. Se appartiene al **processo** occorre distinguere tra il

proprietario, che può soltanto ricevere i messaggi, e l'utente che può solo inviare messaggi. Se, invece, una porta è posseduta dal **sistema operativo** questa non è legata a nessun processo particolare.

● Comunicazione sincrona

In caso di **message passing bloccante** la comunicazione è **sincrona**: il processo mittente si blocca nell'attesa che il processo ricevente riceva il Messaggio, mentre il processo destinatario si blocca nell'attesa di un messaggio.

● Comunicazione asincrona

In caso di **message passing non bloccante** la comunicazione è **asincrona**: il processo mittente invia il messaggio e riprende la propria esecuzione, mentre il destinatario riceve un messaggio valido o nullo.

3.5 Comunicazione nei sistemi client – server (Socket)

Per la comunicazione tra sistemi **client – server**, consideriamo tre strategie: **socket**, **chiamate di procedura remota (RPC)** e **invocazione di metodi remoti (RMI)**.

Una *socket* è un particolare oggetto sul quale leggere e scrivere i dati da trasmettere o ricevere. Una coppia di processi che comunica attraverso una rete usa una coppia di *socket*, una per ogni processo, e ogni socket è identificata da un **indirizzo IP** concatenato ad un numero di porta.

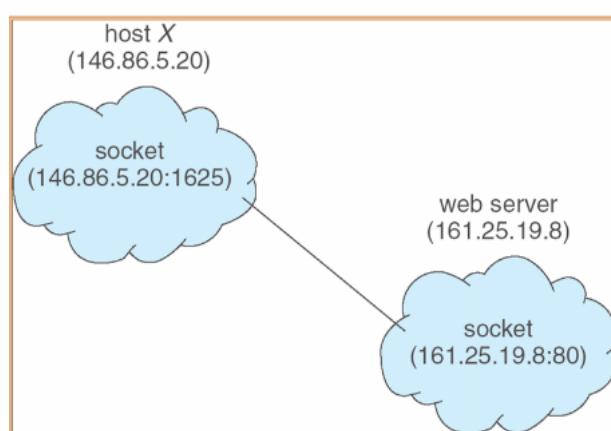
Quando il server riceve una richiesta, se accetta la connessione del client, si stabilisce la comunicazione. Quando un processo client richiede una connessione, il calcolatore che lo esegue assegna una porta specifica che consiste in un numero arbitrario > **1024**.

Se un altro processo, vuole stabilire un'altra connessione con lo stesso server web, riceve un numero di porta maggiore di 1024 è un numero diverso dalla porta del processo precedente. Ciò assicura che ciascuna connessione si è identificata da una distinta coppia di *socket*.

Le *socket* sono illustrate usando il **linguaggio Java**, poiché offre un'interfaccia più semplice dispone di una ricca libreria di strumenti di rete. Il linguaggio Java prevede tre tipi differenti di *socket*:

- 1) quelle orientate alla connessione (**TCP**);
- 2) quelle prive di connessione (**UDP**);
- 3) una socket che permette l'invio simultaneo dei dati a diversi destinatari (**multicast**).

Un client comunica con il server creando una *socket* collegandosi tramite la porta su quel server che è in ascolto. La comunicazione tramite *socket* è diffusa ed efficiente ma è considerata una comunicazione a basso livello, infatti, è responsabilità del client e del server interpretare e organizzare i dati in forme più complesse.



→ **Comunicazione tramite Socket**

3.6 Pipe

Una **pipe** agisce come canale di comunicazione tra processi. Queste sono state uno dei primi meccanismi di comunicazione tra processi (*IPC*) nei sistemi *UNIX*.

Le **pipe convenzionali** permettono a due processi di comunicare secondo una modalità standard chiamata del **produttore- consumatore**: il produttore scrive a un'estremità del canale (**write-end**) mentre il consumatore legge dall'altra estremità (**read-end**). Le *pipe convenzionali* sono **unidirezionali**.

Nel sistema *UNIX* le pipe convenzionali sono costruite utilizzando la funzione: **pipe(int fd[])**.

Essa crea una pipe alla quale si può accedere tramite i descrittori del file *int fd []*:

- *fd[0]* è l'estremità dedicata alla lettura;
- *fd[1]* è l'estremità dedicata alla scrittura.

Si può così accedere alle pipe tramite le system call *read ()* e *write ()*.

Il processo figlio eredita la pipe dal proprio processo padre. E' importante che sia il processo padre che il processo figlio chiudano entrambi le estremità inutilizzate del canale. Poiché la pipe ha bisogno della relazione padre – figlio tra processi, si possono utilizzare solo tra processi residenti sulla stessa macchina.

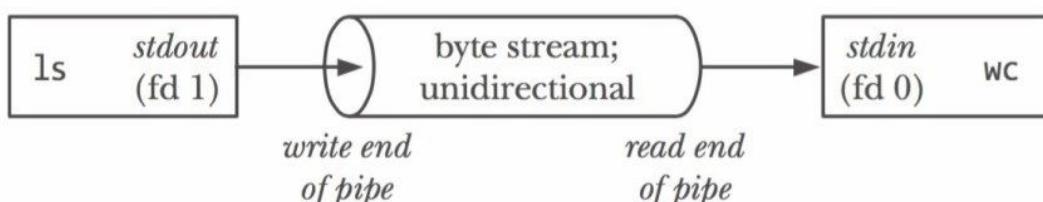
Sia in *UNIX* che in *Windows*, una volta che i processi hanno terminato di comunicare, le pipe convenzionali cessano di esistere.

Le **named pipe** costituiscono uno strumento di comunicazione avanzato, perché la **comunicazione** può essere **bidirezionale** e non è necessaria la relazione padrone – figlio tra processi. Inoltre, le named pipe continuano ad esistere anche dopo che i processi comunicanti terminano. Cesserà di esistere solo quando sarà eliminata dal file system.

Nei sistemi *UNIX* le named pipe sono dette **FIFO**, create attraverso la syscall **mkfifo ()** e vengono manipolate con la *open ()*, *read ()*, *write ()* e *close ()*.

L'unica tipologia di trasmissione consentita è **half duplex** (i dati viaggiano in un'unica direzione alla volta). Nel caso in cui i dati devono viaggiare in entrambe le direzioni, vengono solitamente usate due FIFO. Se si utilizza la FIFO, i processi devono risiedere sulla stessa macchina altrimenti si usa la **socket**.

In Windows, le pipe offrono una comunicazione più ricca perché è permessa la comunicazione **full duplex** (i dati possono viaggiare contemporaneamente in entrambe le direzioni) e i processi comunicanti possono stare sia sulla stessa macchina che su macchine diverse. Attraverso una FIFO di UNIX possono essere trasmessi solo dati **byte-oriented**, mentre i sistemi Windows permettono la trasmissione dei dati sia *byte-oriented* che **message-oriented**.



Scheduling della CPU (Capitolo 4)

4.1 Ciclicità delle fasi d'elaborazione e di I/O

Lo **scheduling della CPU** dipende dalle proprietà dei processi: l'esecuzione del processo consiste in un ciclo di elaborazione ed attesa del completamento delle operazioni di I/O.

L'esecuzione di un processo comincia con fasi in cui viene impiegata soltanto la CPU al 100% senza I/O (**CPU burst**), seguita da una sequenza di operazioni di I/O (**I/O burst**), quindi un'altra sequenza di operazioni della CPU, di nuovo una sequenza di operazioni di I/O, e così via, alternando le due fasi continuamente. L'ultima sequenza di operazioni della CPU si conclude con una richiesta al sistema di terminare l'esecuzione. Le durate delle sequenze di operazioni della CPU vengono rappresentate graficamente da una curva di tipo esponenziale, con brevi sequenze di operazioni della CPU, e poche sequenze di operazioni della CPU molto lunghe.

I **CPU bound** sono i processi che sfruttano al massimo le risorse computazionali del processore, ma non richiedono servizi di I/O, quindi programmi con prevalenza di **CPU burst**. È in contrapposizione a **I/O bound**, programmi con prevalenza di **I/O burst**.

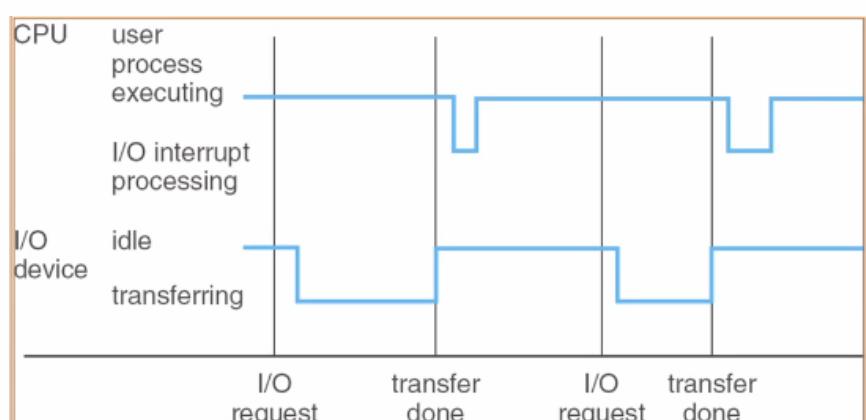
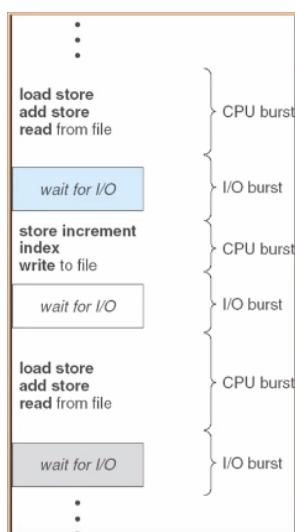
Un classico esempio di tali processi sono i programmi di calcolo matematico, i quali necessitano spesso di un'enorme potenza di calcolo, ma sfruttano l'I/O solo all'inizio della loro vita, per caricare gli input e per produrre gli output.

Gli *scheduler* che interrompono troppo frequentemente il processo in esecuzione portano a privilegiare i task **I/O bound**, i quali sarebbero inattivi per la maggior parte del ciclo di vita, in attesa del completamento delle operazioni di I/O delle periferiche.

L'**I/O bound** si riferisce al tempo necessario per compiere un'elaborazione, determinato dall'attesa delle operazioni di input/output. Questa circostanza si verifica quando si spende più tempo a richiedere i dati che ad elaborarli.

L'[architettura di Von Neumann](#), prevede l'esistenza di un processore centrale che richiede i dati dalla memoria principale, li elabora e infine scrive i risultati in memoria. Siccome i dati devono essere spostati tra la CPU e la memoria attraverso un bus che ha una velocità limitata, esiste una situazione nota come il **collo di bottiglia di Von Neumann**. In parole semplici, la larghezza di banda tra la CPU e la memoria tende a limitare la velocità complessiva dell'elaborazione. Il collo di bottiglia di Von Neumann prevede che sia più semplice far lavorare la CPU più velocemente piuttosto che rifornirla di dati alla velocità sufficiente.

La condizione di **I/O bound** è svantaggiosa poiché costringe la CPU a sospendere la propria elaborazione mentre aspetta il caricamento o lo scaricamento dei dati dalla memoria principale o dalle periferiche di memorizzazione.



4.1.2 Scheduler della CPU

Ogni volta che la CPU passa nello stato di inattività, il sistema operativo sceglie per l'esecuzione uno dei processi presenti nella *coda dei processi pronti*. In particolare, è lo **scheduler e breve termine** a scegliere il processo a cui assegnare la CPU.

La **ready queue** è implementata con diversi algoritmi di scheduling: coda *FIFO*, coda con priorità, un albero o una lista concatenata non ordinata. I processi pronti sono posti nella lista d'attesa per accedere alla CPU. Gli elementi delle code sono i *PCB* dei processi.

4.1.3 Scheduling con diritto di prelazione

Lo scheduling della CPU si attiva quando:

1. un processo passa dallo stato di esecuzione allo stato di attesa (per esempio, richiesta di I/O o richiesta di attesa (*wait*) della terminazione di uno dei processi figli);
2. un processo passa dallo stato di esecuzione allo stato pronto (per esempio, quando si verifica un segnale di interruzione);
3. un processo passa dallo stato di attesa allo stato pronto (per esempio, completamento di un'operazione di I/O);
4. un processo termina.

Quando lo scheduling interviene solo nelle condizioni 1 e 4, si dice che lo schema di scheduling è **senza diritto di prelazione o cooperativo**; altrimenti, lo scheduling è **con diritto di prelazione**.

Nel primo caso, lo *scheduler* deve attendere che il processo termini o che cambi il suo stato da quello di esecuzione a quello di attesa.

Nel secondo caso lo *scheduler* può sottrarre la CPU al processo anche quando questo potrebbe proseguire nella propria esecuzione.

La **prelazione** si ripercuote sulla progettazione del *kernel* del sistema operativo. Durante l'elaborazione di una *syscall*, il kernel può essere impegnato in attività in favore di un processo. Tali attività possono comportare la necessità di modifiche a dati importanti del kernel, come le code di I/O. Se si ha la prelazione del processo durante tali modifiche il kernel deve leggere o modificare gli stessi dati e si può avere il caos. Alcuni sistemi operativi affrontano questo problema attendendo il completamento della chiamata di sistema, quindi il kernel non può esercitare la prelazione.

Questo modello di esecuzione del kernel non è adeguato alla multielaborazione. Il kernel non può sempre ignorare le interruzioni, infatti ci sono alcune sezioni di codice che disattivano le interruzioni al loro inizio e le riattivano alla fine.

4.1.4 Dispatcher

Il **dispatcher** è un modulo che passa il controllo della CPU ai processi scelti dallo scheduler a breve termine. Questa funzione effettua:

- il cambio di contesto;
- il passaggio alla modalità utente;
- il salto alla giusta posizione del programma utente per riavviare l'esecuzione.

Poiché si attiva ad ogni *context switch*, il dispatcher dovrebbe essere quanto più rapido è possibile. Il tempo richiesto dal dispatcher per fermare un processo e avviare l'esecuzione di un altro, è nota come **latenza di dispatcher**.

4.2 Criteri di scheduling

Diversi algoritmi di scheduling la CPU hanno proprietà differenti che soddisfano molti *criteri*. Questi *criteri* sono:

- **Utilizzo della CPU.** La CPU deve essere più attiva possibile, quindi il suo utilizzo deve essere massimizzato.
- **Produttività (throughput).** Il numero di processo completati nell'unità di tempo è detta **produttività (throughput)**. Per processi di lunga durata questo rapporto può essere di un processo all'ora, mentre per brevi transazioni è possibile avere una produttività di 10 processi al secondo.
- **Tempo di completamento.** Il tempo di completamento di un processo è detto **turnaround time**, ed è la somma dei tempi passati nell'attesa dell'ingresso in memoria, nella coda dei processi pronti, durante l'esecuzione nella CPU e nelle operazioni di I/O.
- **Tempo di attesa.** Il tempo di un processo che passa nella fase ready.
- **Tempo di risposta.** In un sistema interattivo, il tempo di completamento può non essere il miglior criterio di valutazione, quindi un'altra misura è data dal tempo che intercorre tra la sottomissione di un processo e la prima risposta. Questa misura è chiamata **tempo di risposta**.

Normalmente si cerca di ottimizzare i valori medi, ma talvolta è opportuno ottimizzare i valori minimi/massimi come nei sistemi interattivi per garantire che tutti gli utenti ottengano un buon servizio: ridurre al minimo il tempo massimo di risposta.

4.3 Algoritmi di scheduling

Gli algoritmi di scheduling determinano l'ordine di esecuzione dei processi in fase ready.

4.3.1 Scheduling in ordine d'arrivo (FCFS)

L'algoritmo più semplice di scheduling della CPU è l'algoritmo di **scheduling in ordine d'arrivo (scheduling first – come, first – served, FCFS)**. La realizzazione del criterio *FCFS* si fonda su una coda *FIFO*, dove i processi che arrivano per primi sono i primi ad essere eseguiti.

Quando un processo entra nella *ready queue*, si collega il suo PCB all'ultimo elemento della coda.

Quando è libera, si assegna la CPU al processo che si trova alla testa della *ready queue*, rimuovendolo da essa.

L'algoritmo di scheduling *FCFS* è **senza prelazione** (non va bene per i sistemi *time-sharing*): una volta che la CPU è assegnata a un processo, questo la trattiene fino al momento del rilascio della sua esecuzione o se va in fase di *wait*.

L'algoritmo *FCFS* è soggetto al cosiddetto **“effetto convoglio”**, cioè casi in cui i processi di breve durata si trovano ad attendere che un processo lungo rilasci la CPU, spesso accade quando ci sono pochi processi *CPU-bounded* e molti processi *I/O-bounded*.

4.3.2 Scheduling per brevità (SJF)

L'algoritmo di **scheduling per brevità (shortest – job – first, SJF)**, si basa sull'idea che i processi che hanno tempi di esecuzione minori debbano essere favoriti rispetto a processi che hanno tempi di esecuzione maggiori; lo scheduling si esegue esaminando la lunghezza della successiva sequenza di operazioni della CPU. Se due processi hanno le successive sequenze di operazioni della CPU della stessa lunghezza, si applica lo *scheduling FCFS*.

L'algoritmo di scheduling SJF è ottimale rendendo minimo il tempo d'attesa medio per un insieme di processi: spostando un processo breve prima di un processo lungo, il tempo d'attesa per il processo breve diminuisce.

Ovviamente non si può conoscere a priori il tempo di completamento del successivo processo. Per prevedere i tempi di esecuzione si calcola la **media esponenziale** delle lunghezze delle precedenti sequenze di operazioni della CPU. Il funzionamento di tale sistema è il seguente:

Sia $t(n)$ la durata dell' n -esima CPU burst e $T(n+1)$ il valore previsto per la successiva sequenza, ne segue:

$$T(n + 1) = \alpha \cdot t(n) + (1 - \alpha) \cdot T(n) \text{ precedente} \quad \rightarrow \quad T(n) \text{ rappresenta la previsione}$$

- Ponendo $\alpha = 0$ si suppone che la situazione attuale sia transitoria e quindi non debba influenzare la previsione.
- Ponendo $\alpha = 1$ si suppone che la storia precedente non influenzi l'attuale sequenza di esecuzione.
- Ponendo $\alpha = 2$ si suppone che la storia passata e lo stato attuale abbiano lo stesso peso e si effettua una media.

L'algoritmo SJF può essere sia con **prelazione** sia **senza prelazione**. La scelta si presenta quando alla *ready queue* arriva un nuovo processo mentre un altro processo è ancora in esecuzione. Un algoritmo SJF con **prelazione** sostituisce il processo attualmente in esecuzione; mentre un algoritmo SJF **senza prelazione** permette al processo in esecuzione di portare a termine la propria sequenza di operazioni della CPU.

4.3.3 Scheduling per priorità

L'algoritmo di **scheduling per priorità** funziona in questo modo. Ad ogni processo che entra nella ready queue viene assegnata una determinata priorità e il processo con la priorità più alta ottiene la CPU, i processi con priorità uguali sono ordinati secondo uno schema *FCFS*.

Le priorità possono essere definite sia **internamente** che **esternamente**. Quelle definite *internamente* usano uno o più quantità misurabili per calcolare la priorità del processo (es. limiti di tempo, requisiti di memoria etc..). Le priorità *esterne* si definiscono secondo criteri esterni al SO. In *UNIX* la priorità è un valore che va da -20 priorità massima a +20 priorità minima.

Lo scheduling per priorità può essere sia con **prelazione** sia **senza prelazione**: nel primo caso si sottrae la CPU al processo in esecuzione se la priorità dell'ultimo processo è arrivato è superiore; nel secondo caso si limita a porre l'ultimo processo arrivato alla testa della ready queue.

Un problema importante relativo agli algoritmi di scheduling per priorità è **l'attesa indefinita (starvation)**. Un processo pronto per l'esecuzione, ma che non dispone della CPU, si può considerare bloccato. Una soluzione al problema dell'attesa indefinita dei processi con bassa priorità è costituita dall'**invecchiamento (aging)**: si tratta di una tecnica che aumenta gradualmente la priorità dei processi che attendono nel sistema da parecchio tempo.

4.3.4 Scheduling circolare (RR)

L'algoritmo di *scheduling circolare (round – robin, RR)* è stato progettato per i sistemi a tempo ripartito; è simile allo scheduling *FCFS* con la differenza che l'*RR* è un algoritmo **preemptive**: esegue i processi nell'ordine d'arrivo ed esegue la prelazione del processo in esecuzione, ponendolo alla fine della coda

dei processi in attesa, qualora l'esecuzione duri più della "quantità di tempo" stabilita, e facendo proseguire l'esecuzione al successivo processo in attesa. Per realizzare lo *scheduling RR* si gestisce la *ready queue* come una coda FIFO.

Ciascun processo riceve una piccola quantità del tempo della CPU, chiamata **quanto di tempo** o **porzione di tempo (time slice)**. Questo meccanismo è garantito dalla presenza di un **timer**, settato dalla *time slice*, che si occupa di invocare un *context switch* allo scadere di ogni *time slice*.

Le prestazioni di quest'algoritmo sono dunque influenzate dal tempo medio d'attesa. La scelta del *time slice* è influenzata dal tempo medio di completamento, se la *time slice* fosse molto bassa si verificherebbe troppi *context switching*, in caso la *time slice* fosse troppo alta si avrebbe lo stesso comportamento del FCFS. Una pratica che si adotta è che l'80% delle CPU burst devono essere eseguite in una *time slice*.

4.3.5 Scheduling a code multiple

Spesso ci si trova in situazioni dove ci sono diversi processi classificabili in diversi gruppi, dove ogni gruppo necessita di un algoritmo di scheduling differente. Una distinzione è quella che si fa tra i processi che si eseguono in **primo piano (foreground)**, o **interattivi**, e i processi che si eseguono in **sottofondo (background)**, o a **lotti (batch)**.

L'algoritmo di **scheduling a code multiple (multilevel queue scheduling algorithm)** suddivide la ready queue in code distinte, ogni coda ha il proprio algoritmo di scheduling. La coda dei processi in **foreground** si può gestire con un *algoritmo RR*, mentre quella dei processi in **background** si può gestire con un *algoritmo FCFS*.

Per gestire le code è necessario a sua volta un algoritmo di scheduling tra le code, che è solitamente, un algoritmo di *scheduling a priorità*, oppure un'altra possibilità è di suddividere le risorse della CPU tra le varie code:

- Nel primo caso, un processo presente in una coda inferiore non potrà essere eseguito fino a quando le code con priorità superiore non saranno vuote, inoltre, se un processo si trova in fase di esecuzione e un nuovo processo di una coda superiore viene aggiunto in coda e se l'algoritmo è *preemptive*, allora viene effettuata una prelazione in favore di quest'ultimo.
- Nel secondo caso viene settata una *timeslice* variabile per ogni coda in base alla priorità delle stesse.

Nell'algoritmo di *scheduling a code multiple*, abbiamo in ordine di priorità:

1. Processi di sistema;
2. Processi interattivi;
3. Processi interattivi di editing;
4. Processi eseguiti in sottofondo;
5. Processi degli studenti.

4.3.6 Scheduling a code multiple con retroazione

Di solito i processi si assegnano in modo permanente a una coda e non si possono spostare fra queste. Lo **scheduling a code multiple con retroazione (multilevel feedback queue scheduling)**, permette ai processi di spostarsi fra le code: se un processo usa troppo tempo di elaborazione della CPU, viene spostato in una coda con priorità più bassa.

Questo schema mantiene i processi con prevalenza di I/O e i processi interattivi nelle code con priorità più elevata. In modo analogo, si può spostare in una coda con priorità più elevata un processo che attende troppo a lungo, effettuando l'**aging**.

Uno *scheduler a code multiple con retroazione* è caratterizzato:

- ❖ numero di code;
- ❖ algoritmo di scheduling per ciascuna coda;
- ❖ metodo usato per determinare quando spostare un processo in una coda con priorità maggiore;
- ❖ metodo usato per determinare quando spostare un processo in una coda con priorità minore;
- ❖ metodo usato per determinare in quale coda si deve mettere un processo quando richiede un servizio.

4.4 Scheduling multiprocessor

Nei sistemi in cui le unità di elaborazione sono **identiche (sistemi omogenei)**, si può usare qualunque unità di elaborazione disponibile per eseguire qualsiasi processo presente nella coda.

4.4.1 Soluzioni di scheduling per multiprocessori

In base al tipo di multielaborazione ci sono comportamenti diversi dello scheduler:

- **Asymmetric multiprocessing:** lo scheduling, l'elaborazione dell'I/O e le altre attività del SO sono affidate ad un unico processore, detto **master**, questo riduce la necessità di condivisione dati grazie al fatto che un unico processore ha l'accesso alle risorse del sistema;
- **Symmetric multiprocessing (SMP):** i processi pronti vanno a formare un'unica coda comune oppure vi è una coda per ogni processore. Ogni processore ha un proprio scheduler che esamina la coda e preleva il prossimo processo; l'accesso concorrente di più processori alla stessa coda rende delicata la programmazione degli scheduler.

Linux e gli attuali sistemi operativi utilizzano **Symmetric multiprocessing**.

4.4.2 Predilezione per il processore

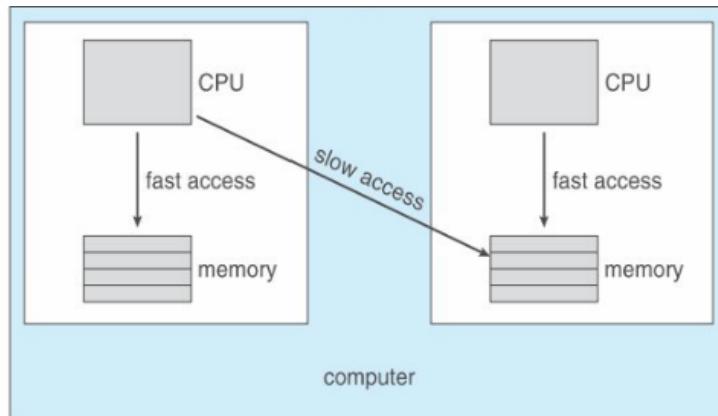
La **predilezione** per il processore è il tentativo di mantenere un processo in esecuzione sullo stesso processore al fine di riutilizzare il contenuto delle cache, infatti, se un processo si sposta su un altro processore, i contenuti della memoria cache devono essere invalidati sul processore di partenza, mentre la cache del processore di arrivo deve essere nuovamente riempita.

- **Predilezione debole (soft affinity):** il SO tenta di mantenere il processo sullo stesso processore, ma non vieta la migrazione;
- **Predilezione forte (hard affinity):** il processo è vincolato all'esecuzione su uno o più processori.

Linux implementa entrambi i tipi di predilezione.

NUMA

L'architettura della memoria influenza la predilezione, nel caso di **NUMA (non uniform memory access)** sistemi costituiti da diverse schede con una o più CPU e memoria, le CPU di una scheda accedono più velocemente alla memoria locale, ogni processo dovrebbe risiedere nella memoria del processore su cui viene eseguito.



4.4.3 Bilanciamento del carico

Sui sistemi SMP è importante che il carico di lavoro sia distribuito equamente tra tutti i processori per sfruttare al meglio la multielaborazione. Se ciò non avvenisse, alcuni processori potrebbero restare inattivi mentre altri vengono intensamente sfruttati. Questo problema viene risolto con il **bilanciamento del carico** che tenta di ripartire il carico di lavoro in modo uniforme tra tutti i processori di un sistema SMP. Questa tecnica è necessaria solo nei sistemi in cui ciascun processore possiede una coda privata di processi passibili di esecuzione. Nei sistemi che mantengono una coda comune, il bilanciamento del carico è inutile, poiché un processore inattivo passerà eseguirà subito un processo dalla coda comune.

Il bilanciamento del carico può seguire due approcci:

- **migrazione guidata (*push migration*)**: prevede che un processo controlli periodicamente il carico di ogni processore. Nel momento in cui sia una sproporzione, la *push migration* riporterà l'equilibrio, spostando i processi dal processore saturo ad altri più liberi o inattivi.
- **migrazione spontanea (*pull migration*)**: Si ha quando un processore inattivo sottrae, ad un processore sovraccarico, un processo in attesa.

Linux implementa entrambe i meccanismi di migrazione. Il bilanciamento del carico è in antitesi rispetto alla filosofia di predilezione.

4.4.4 Processori multicore

I sistemi SMP hanno reso possibile la concorrenza tra **thread** con l'utilizzo di diversi processori fisici. Lo scopo è di inserire più **unità di calcolo (core)** in un unico chip fisico, dando origine ad un processore **multi-core**. Ogni core ha i registri che gli servono per conservare informazioni sul suo stato e appare al sistema operativo come un processore fisico separato. I sistemi SMP che usano processori *multi-core* sono più veloci e consumano meno energia dei sistemi in cui ciascun processore è costituito da un singolo chip.

I processori multicore possono complicare lo scheduling: quando un processore accede alla memoria, una quantità importante di tempo trascorre in attesa della disponibilità dei dati. Questa situazione, nota come **stallo della memoria**, può verificarsi per varie ragioni, come la mancanza dei dati richiesti nella cache. Per rimediare a questa situazione, molti hardware recenti implementano dei core **multi-thread** in cui due o più *thread hardware* sono assegnati ad un singolo core. In questo modo, se un thread è in situazione di stallo in attesa della memoria, il core può passare il controllo ad un altro thread.

Dal punto di vista del sistema operativo, ogni *thread hardware* appare come un *processore logico* in grado di eseguire un trade software.

Ci sono due modi per rendere un processore ***multithread***:

- ❖ **multithreading grezzo (coarse-grained)**: un thread resta in esecuzione su un processore fino al verificarsi di un evento a lunga latenza, come ad esempio uno stallo di memoria. A causa dell'attesa, il processore deve passare a un altro thread e iniziare ad eseguirlo. Il costo per cambiare il thread in esecuzione è alto perché occorre ripulire la pipeline dalle istruzioni.
- ❖ **multithreading fine (fine-grained)**: si passa da un thread a un altro con un livello molto più fine di granularità.

Un processore ***multi-thread*** e ***multi-core*** richiede due livelli diversi di scheduling:

1. Un primo livello ci sono le decisioni di scheduling che devono essere intraprese dal sistema operativo per stabilire quale *thread software* mandare in esecuzione su ciascun *thread hardware* (processore logico). Per realizzare questo tipo di scheduling il processore può scegliere qualunque algoritmo.
2. Nel secondo livello, ogni unità di calcolo decide quale *thread hardware* eseguire: si può usare un algoritmo circolare oppure altre strategie.

4.5 Hyper-Threading

Quasi tutte le CPU oggi sono ***multi-core***, ovvero contengono diverse unità di elaborazione in grado di gestire diverse attività contemporaneamente.

Che cos'è il multi-threading?

Il ***multi-threading*** è una forma di parallelizzazione o divisione del lavoro per consentire l'elaborazione simultanea. Invece di assegnare un carico di lavoro elevato a un singolo core, i programmi threadizzati dividono il lavoro in più *thread software*. Questi *thread* vengono quindi elaborati in parallelo da diversi core della CPU per risparmiare tempo.

Che cos'è l'Hyper-Threading?

La tecnologia **Hyper-Threading** è un'innovazione hardware che consente l'esecuzione di più *thread* su ciascun core. Più *thread* attivi significano più lavoro svolto in parallelo.

Quando la tecnologia **Hyper-Threading** è attiva, la CPU genera due contesti di esecuzione per ogni core fisico. **Questo significa che un singolo core fisico funziona come due "core logici", in grado di gestire diversi *thread software*.**

Due ***core logici*** possono svolgere le attività in modo più efficiente rispetto a un core tradizionale ***single-threaded***. Sfruttando anche i tempi di inattività in cui il core deve attendere il completamento di altre attività, la tecnologia **Hyper-Threading** permette di migliorare il throughput della CPU (fino al 30% nelle applicazioni server).



4.6 Virtualizzazione e scheduling

Un sistema dotato di virtualizzazione, anche se a singola CPU, agisce come un sistema multiprocessore. La virtualizzazione software offre uno o più CPU virtuali a ogni macchina virtuale in esecuzione sul sistema, e quindi pianifica l'utilizzo della CPU fisica condividendola con le varie macchine virtuali. Ogni algoritmo di scheduling del sistema operativo ospite viene influenzato negativamente dalla virtualizzazione. I singoli sistemi operativi visualizzati sfruttano solo una parte dei cicli di CPU disponibili, invece, effettuano lo scheduling come se sfruttassero tutti i cicli fisicamente disponibili. L'orologio all'interno di una macchina virtuale fornisce valori sbagliati perché i contatori impiegano più tempo a scattare di quanto farebbero su una CPU dedicata. La virtualizzazione può quindi agire negativamente sugli algoritmi di scheduling del sistema operativo ospitato sulla macchina virtuale.

4.7 Esempio: scheduling di Linux

In Linux lo scheduling si basa sul concetto di **timesharing**, per cui ad ogni processo è assegnato un quanto di tempo massimo per l'esecuzione. Lo **scheduler di Linux** usa un algoritmo di scheduling con prelazione, basato sulle **priorità**, con due gamme di priorità separate: un intervallo **real-time** che va da 0 a 99 e un intervallo detto **nice** compreso tra 100 e 140: i valori più bassi sono le priorità più alte. Linux assegna ai task con priorità più alta, porzioni di tempo più ampie e a quelle dalla priorità più bassa quanti di tempo più brevi.

Una **priorità** può essere:

- **Dinamica**: introdotta per evitare il fenomeno della *starvation*;
- **Statica**: introdotta per consentire la gestione di processi **real-time**. Ai processi real-time è assegnata una priorità maggiore di quella assegnata ai processi ordinari.

Linux usa uno **scheduling preemptive** quindi ad un processo viene tolta la CPU se:

- 1) Esaurisce il quanto di tempo a sua disposizione.
- 2) Un processo a priorità più alta è pronto per l'esecuzione (*task running*).

Il kernel elenca i task pronti per l'esecuzione in una struttura dati detta **runqueue** (*coda di esecuzione*).

Ogni coda di esecuzione contiene 2 array di priorità:

1. **Attivo**: contiene tutti i task che hanno ancora tempo da sfruttare;
2. **Scaduto**: elenca i task scaduti.

Entrambi gli array contengono una lista di task, ordinata progressivamente secondo la priorità.

Problemi

Le prestazioni dello scheduler di Linux degradano al crescere del numero di processi; infatti le priorità devono essere ricalcolate per tutti i processi. Il valore predefinito per il quanto può essere eccessivo nei sistemi con carico elevato. Il supporto fornito ai processi real-time è limitato.

4.8 Valutazione degli algoritmi

Ci si può chiedere quale algoritmo di scheduling della CPU bisogna applicare. Bisogna rendere massimo l'utilizzo della CPU con il vincolo che il massimo **tempo di risposta** sia di 1 secondo e rendere massima la **produttività** in modo che il **tempo di completamento** sia linearmente proporzionale al **tempo d'esecuzione totale**.

La **valutazione analitica**, secondo l'algoritmo dato e il carico di lavoro del sistema, fornisce una formula o un numero che valuta le prestazioni dell'algoritmo per quel carico di lavoro. Consiste nel valutare gli algoritmi su dati concreti, i risultati ottenuti sono numeri esatti che consentono il confronto tra gli algoritmi.

Reti di code

Il sistema di calcolo si descrive come una rete di server, ciascuno con una coda d'attesa. La CPU è un server con la propria coda dei processi pronti e il sistema di I/O ha le sue code dei dispositivi. Se sono noti l'andamento degli arrivi e dei servizi, si possono calcolare la lunghezza media delle code e il tempo medio d'attesa e così via.

Simulazioni

Per riuscire ad avere una valutazione più precisa degli algoritmi di scheduling, ci si può servire di **simulazioni**. Le *simulazioni* implicano la programmazione di un modello del sistema di calcolo; le strutture dati rappresentano gli elementi principali del sistema.

Durante l'esecuzione della *simulazione* si raccolgono i dati che vengono stampati e confrontati. Poiché per effettuare una buona simulazione richiedono diverse ore di tempo di elaborazione, le *simulazioni* possono essere tuttavia molto onerose. Una *simulazione* dettagliata dà risultati molto precisi ma richiede anche una grande quantità di tempo.

Realizzazione

Persino una simulazione ha dei limiti per quel che riguarda la precisione. L'unico modo assolutamente sicuro per valutare un algoritmo di scheduling consiste nel codificarlo, inserirlo nel sistema operativo e osservarne il comportamento nelle reali condizioni di funzionamento. Le spese sono dovute alla codifica dell'algoritmo e alle modifiche da fare al sistema operativo e alle reazioni degli utenti sulle modifiche del sistema operativo.

Thread (Capitolo 5)

5.1 Definizione di thread

Un **thread** è un **percorso di controllo** all'interno di un processo, in particolare è l'unità base d'uso della CPU. Un processo è un programma in esecuzione con un unico percorso di controllo. Molti sistemi operativi moderni permettono che un processo possa avere più percorsi di controllo, ovvero possedere dei **multi-thread**.

Un *thread* comprende:

- **TID:** identificare univoco del thread;
- **Contatore di programma;**
- **Insieme di registri;**
- **Stack.**

Inoltre, un *thread* condivide con gli altri *thread* che appartengono allo stesso processo:

- la **sezione del codice;**
- la **sezione dei dati;**
- **altre risorse allocate dal processo originale** (file aperti e segnali).

Un processo tradizionale, chiamato **processo pesante (heavyweight process)**, è composto da un solo *thread* (processo pesante in riferimento allo spazio di indirizzamento). I *thread* sono anche detti **processi leggeri** perché hanno un contesto più semplice. Un processo *multi-thread* è in grado di lavorare a più compiti in modo concorrente. Il *thread* è shared memory.

5.2 Multithreading

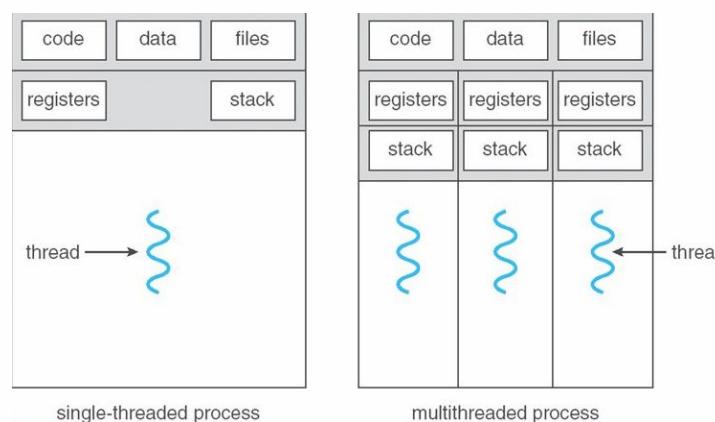
Al giorno d'oggi la maggior parte degli applicativi sono **multi-thread** poiché necessitano di eseguire un numero elevato di operazioni in parallelo e la gestione dei processi può diventare molto onerosa sia dal punto di vista computazionale che per l'utilizzo di risorse:

- **Creazione:** allocazione dello spazio degli indirizzi e successiva popolazione;
- **Context switch:** salvataggio e ripristino degli spazi di indirizzamento (codice, dati, stack) di due processi.

Uno dei vantaggi principali della programmazione *multithread* è dato dal fatto che i *thread* appartenenti allo stesso processo ne condividono i dati. In particolari circostanze, ogni *thread* può necessitare di una copia privata di certi dati, chiamata **dati specifici dei thread**. Per associare ciascun *thread* al relativo identificatore si possono usare i dati specifici dei thread.

Se un'applicazione generasse molti processi ci sarebbe il rischio che il SO passi più tempo a eseguire operazioni di gestione piuttosto che codice applicativo.

I *thread* sono importanti anche nei sistemi che impiegano le **RPC** (*remote procedure call*); ovvero un sistema che permette la comunicazione tra processi. I *server RPC* sono *multi-thread* poiché il server delega la gestione a un *thread* separato e in questo modo può gestire diverse richieste in modo concorrente.



5.2.1 Vantaggi

I vantaggi della programmazione **multi-thread** aumentano nelle architetture multiprocessore, dove i *thread* possono essere eseguiti in parallelo, inoltre l'impiego della programmazione *multithread* in un sistema con più CPU fa aumentare il grado di parallelismo. I vantaggi del *multithread* si possono classificare in quattro fattori principali:

1. **Tempo di risposta:** durante l'esecuzione di un'applicazione *multi-thread* se uno dei *thread* è bloccato nell'esecuzione di un'operazione particolarmente lunga, questo non implica il rallentamento degli altri *thread* che possono continuare l'esecuzione riducendo così il tempo medio di risposta;
2. **Condivisione di risorse:** i *thread* di uno stesso processo risiedono nello stesso spazio di indirizzamento, pertanto condividono la memoria e le risorse e questo comporta comunicazioni molto più rapide con memoria condivisa e *message passing*;
3. **Economia:** la creazione di un nuovo processo è molto costosa, a differenza dei *thread* che è molto più leggera ed è più facile gestire i *context switch*;
4. **Scalabilità:** la programmazione *multi-thread* comporta ancora più vantaggi nelle architetture multiprocessore dove i *thread* possono essere eseguiti in parallelo;

5.2.2 Thread nelle architetture multicore

Un **sistema parallelo** può eseguire simultaneamente più *task*, questi sistemi necessitano di un'architettura multiprocessore. Nei **sistemi multicore** i *thread* possono essere eseguiti in parallelo, poiché ogni *thread* può essere assegnato ad un core specifico. Un **sistema concorrente**, invece, supporta più *task* consentendo a tutti di progredire nell'esecuzione.

La programmazione *multithread* offre un utilizzo più efficiente dei processori e aiuta a sfruttare al meglio la **concorrenza**.

In un sistema con una **singola unità di calcolo**, “**esecuzione concorrente**” significa che l'esecuzione dei *thread* è stratificata nel tempo o **interfogliata** (*interleaved*) perché la CPU in grado di eseguire un solo *thread* alla volta.

Su un **sistema multicore**, invece, “**esecuzione concorrente**” significa che i *thread* possono funzionare in parallelo, dal momento che il sistema può assegnare *thread* diversi a ciascuna unità di calcolo.

I progettisti di sistemi operativi devono scrivere algoritmi di scheduling che utilizzano diverse unità di calcolo per permettere un'esecuzione parallela. I programmati di applicazioni, invece, devono modificare i programmi esistenti e progettare nuovi programmi *multithread* per trarre vantaggio dai sistemi multicore. I principali obiettivi della programmazione dei sistemi multicore sono:

1. **Identificazione dei task:** in questa fase vengono analizzati i task, vengono identificati alcuni task che potrebbero essere eseguiti in parallelo su core distinti;
2. **Bilanciamento:** durante l'individuazione dei task viene analizzata la loro mole di lavoro cercando di equilibrarla;
3. **Suddivisione dei dati:** come per i task, anche i dati su cui loro agiscono devono essere separati per essere utilizzati su core distinti;
4. **Dipendenze dei dati:** alcuni task potrebbero utilizzare dati prodotti da altri task questa dipendenza va gestita con meccanismi di sincronizzazione;
5. **Test e debugging:** quando un programma funziona in parallelo su unità multiple, ci sono diversi possibili flussi di esecuzione. Effettuare i test e il debugging di applicazioni multithread su architetture multicore è molto più complesso.

5.3 Modelli di programmazione multithread

I **thread** possono essere divisi in:

- **thread a livello utente**: gestito senza l'aiuto del kernel, realizzato da una libreria (*POSIX* per *UNIX*). Il **vantaggio** è che può essere implementato un pacchetto di thread anche su SO che non supportano i thread. Lo **svantaggio** è che una chiamata di sistema bloccante da parte di un *thread* bloccherebbe tutti gli altri therad (il kernel blocca il processo).
- **thread a livello kernel**: gestito direttamente dal SO ed il kernel si occupa della creazione, scheduling, sincronizzazione e cancellazione.

Esiste una relazione tra i thread a livello utente e i thread a livello kernel.

5.3.1 Modello da molti a uno

Il **modello da molti uno** fa corrispondere molti thread a livello utente a un singolo thread a livello kernel, quindi i thread sono implementati a livello applicativo e il loro scheduler non fa parte del SO che ha solo visibilità del processo.

I **vantaggi** sono:

- Gestione efficiente dei thread nello spazio utente con uno scheduling poco oneroso;
- Non richiede un kernel multithread.

Gli **svantaggi** sono:

- Se un thread effettua una system call bloccante, l'intero processo non può continuare;
- I thread sono legati allo stesso processo a livello kernel e pertanto non possono essere eseguiti su core distinti.

5.3.2 Modello da uno a uno

Il **modello da uno a uno** mette in corrispondenza ciascun thread a livello utente corrisponde a un thread a livello kernel, quindi il kernel vede un flusso diverso per ogni *thread*. I *thread*, in questo modello, vengono gestiti direttamente dallo scheduler del kernel come se fossero processi, questi tipi di *thread* vengono detti **thread nativi**.

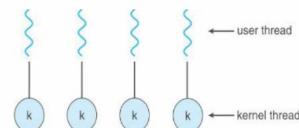
I **vantaggi** sono:

- Scheduling molto efficiente;
- Possibilità di eseguire i vari thread su core distinti;
- Se un thread effettua una system call bloccante gli altri thread possono proseguire la loro esecuzione. Questo modello permette anche l'esecuzione dei thread in parallelo nei sistemi multiprocessore.

Gli **svantaggi** sono:

- Possibile inefficienza dovuta alla creazione di molti thread a livello kernel, abbiamo una limitazione dei thread gestibili dal sistema;

- Richiede un kernel multithread.



5.3.3 Modello da molti a molti

Il **modello da molti a molti** mette in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel, quindi il sistema operativo ha a disposizione un **pool di thread** (detto **worker**) ognuno dei quali, ogni qual volta c'è necessità, viene assegnato ad un *thread* utente.

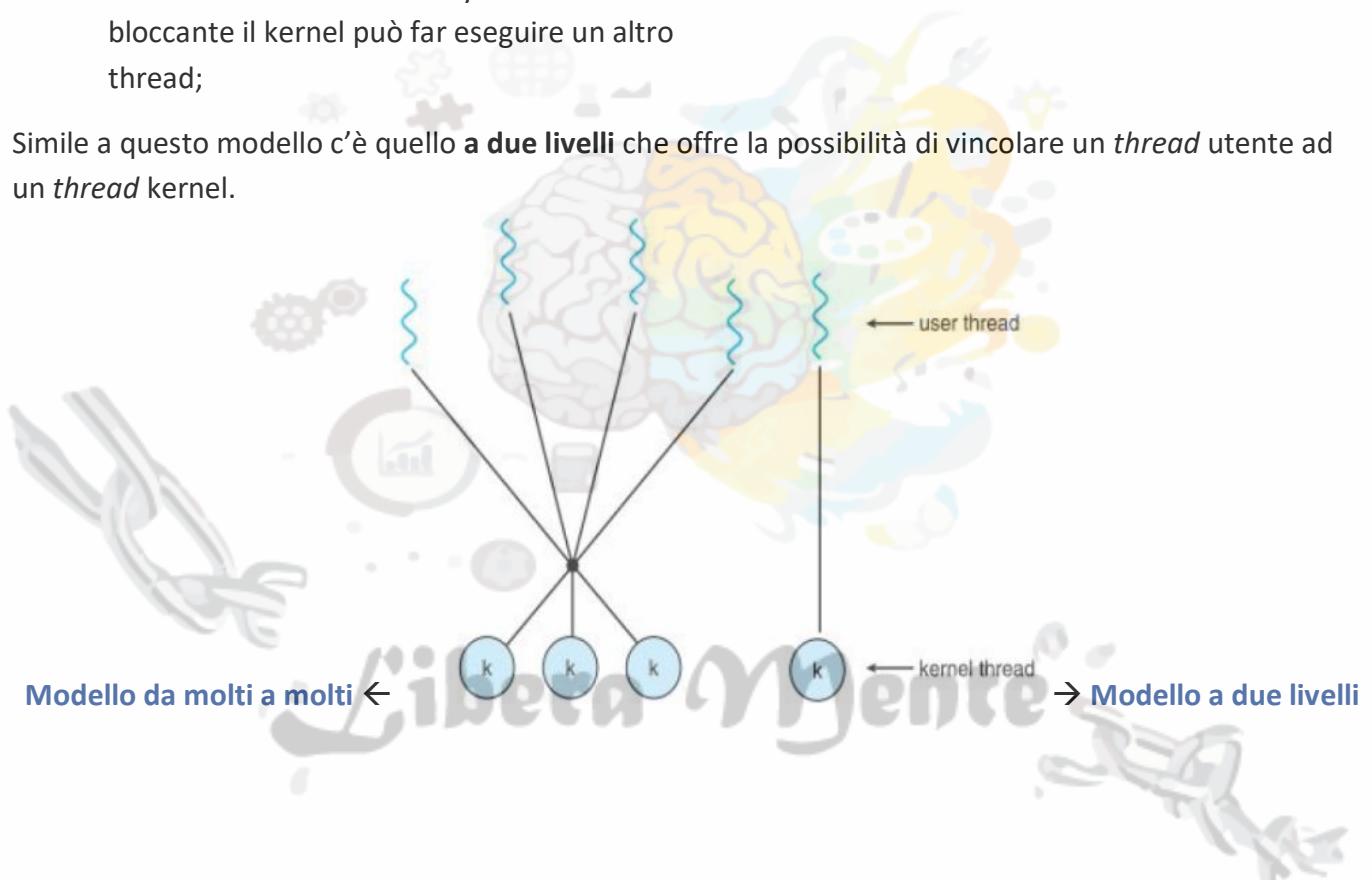
I **vantaggi** sono:

- Possibilità di creare tanti *thread* a livello utente quanti sono necessari e far eseguire i corrispondenti *thread* a livello kernel in parallelo;
- Se un *thread* effettua una system call bloccante il kernel può far eseguire un altro *thread*;

Gli **svantaggi** sono:

- Difficoltà di definire la dimensione del **pool di thread** a livello kernel.

Simile a questo modello c'è quello a **due livelli** che offre la possibilità di vincolare un *thread* utente ad un *thread* kernel.



5.4 Librerie dei *thread*

La **libreria dei *thread*** fornisce al programmatore una API per la creazione e la gestione dei *thread*. I metodi per implementare una libreria dei *thread* sono due:

1. la libreria è collocata interamente a livello utente, senza far ricorso al kernel. Il codice e le strutture dati per la libreria si trovano tutti nello spazio degli utenti, quindi invocare una funzione della libreria vuol dire effettuare una chiamata locale a una funzione e non in una system call.
2. Il secondo metodo consiste nell'implementare una libreria a livello kernel, con l'aiuto del sistema operativo. In questo caso, il codice e le strutture dati per la libreria si trovano nello spazio del kernel. Invocare una funzione della API della libreria provoca un system call al kernel.

Le librerie dei *thread* più utilizzate sono 3:

1. **Pthreads** di *POSIX* (si presenta come libreria a livello utente sia a livello kernel);
2. **Win32** (è una libreria a livello kernel per i sistemi Windows);
3. **Java** (la API per la creazione dei *thread* è gestita direttamente dai programmi Java).

5.4.1 Pthreads

Col termine **Pthreads** ci si riferisce allo standard *POSIX* che definisce la API per la creazione e la sincronizzazione dei *thread*. Si tratta di una **definizione** del comportamento dei thread. Nei programmi Pthreads, i nuovi *thread* sono eseguiti da una funzione specifica.

Esempio

All'inizio dell'esecuzione del programma c'è un unico *thread* di controllo che parte dal main(). Il main() crea un secondo *thread* che inizia l'esecuzione dalla funzione runner(). Entrambi i *thread* condividono i valori globali di sum.

La dichiarazione della variabile pthread_t tid specifica l'identificatore per il *thread* da creare.

La dichiarazione di pthread_attr_t attr riguarda la struttura dati per gli attributi del *thread*, i cui valori si assegnano con la system call pthread_attr_init (&attr).

La system call pthread_create crea un nuovo *thread* che gli passa l'identificatore, i suoi attributi, il nome della funzione da cui il nuovo *thread* inizierà l'esecuzione (in questo caso runner()) e il numero intero fornito da argv[1].

A questo punto il programma ha due *thread*: il **thread iniziale** (o genitore), in main() e il **thread che esegue la somma** (o figlio), in runner().

Dopo aver creato il secondo, il primo *thread* attende la fine del secondo chiamando la funzione pthread_join().

Il secondo thread termina quando si invoca la funzione pthread_exit() e il thread iniziale produce in uscita il valore sum della sommatoria.

5.5 Cancellazione dei thread

La **cancellazione dei thread** è l'operazione che permette di terminare un *thread* prima che completi il suo compito. Per esempio, quando un utente chiude un programma Web per interrompere il caricamento di una pagina.

Un *thread* da cancellare è spesso chiamato **thread bersaglio** (*target thread*) e la sua cancellazione avviene in due modi diversi:

1. **cancellazione asincrona**. Un *thread* fa subito terminare il *thread bersaglio*;
2. **cancellazione differita**. Il *thread bersaglio* può controllare periodicamente se deve terminare in modo da riuscirci in modo adeguato.

La difficoltà con la **cancellazione asincrona** si presenta nei casi in cui ci siano risorse assegnate ad un *thread* cancellato, o se si cancella un *thread* mentre sta aggiornando dei dati che condivide con altri *thread*. Il sistema operativo di solito si riprende le risorse di sistema usate da un *thread* cancellato, ma spesso non le riprende tutte, quindi la cancellazione di un *thread* in modo asincrono potrebbe non liberare una risorsa necessaria per tutto il sistema.

La **cancellazione differita**, invece, funziona tramite un *thread* che segnala la necessità di cancellare un certo *thread bersaglio*; la cancellazione avviene soltanto quando il *thread bersaglio* verifica se deve essere cancellata oppure meno. In **Linux** la cancellazione è gestita tramite i **segnali**.

5.5.1 Gestione dei segnali

Nei sistemi *UNIX* si usano i **segnali** per comunicare ai processi il verificarsi di alcuni eventi. Un segnale si può ricevere in modo **sincrono** o **asincrono**, dipende dalla sorgente e dalla regione della segnalazione dell'evento. Tutti i segnali seguono lo stesso schema: all'arrivo di un particolare evento si genera un segnale; si invia il segnale è un processo; e una volta ricevuto, il segnale deve essere gestito.

Segnali sincroni

Un accesso illegale alla memoria o una divisione per zero generano **segnali sincroni**. I segnali sincroni si inviano lo stesso processo.

Segnali asincroni

Quando un segnale è causato da un evento esterno al processo in esecuzione, tale processo riceve il segnale in modo asincrono, ad esempio la terminazione di un processo con il comando `<control> <C>`. Un segnale asincrono si invia ad un altro processo.

Ogni segnale si può gestire in due modi: tramite un **gestore predefinito di segnali** (eseguito dal kernel quando deve gestire un segnale) o tramite **un gestore di segnali definito dall'utente**.

Sia i segnali sincroni che asincroni sono gestibili in modo diverso: alcuni si possono ignorare altri si possono gestire terminando l'esecuzione del programma.

Per i *processi a singolo thread* la gestione dei segnali è semplice mentre per i *processi multithread* si pone il problema del *thread* a cui si deve inviare il segnale.

I segnali sincronici si devono inviare al *thread* che ha generato l'evento. Se si tratta di segnali asincroni, il segnale che termina un processo si deve inviare a tutti i *thread*.

Windows non prevede la gestione esplicita dei segnali, questi si possono emulare con le **chiamate di procedure asincrone (APC)**. Le funzioni APC permettono a un *thread* a livello utente di specificare la funzione da richiamare quando il *thread* riceve la comunicazione di un particolare evento. In un ambiente *multithread* UNIX necessita di un criterio di gestione dei segnali mentre l'APC è rivolta a un particolare *thread* e non a un processo.

5.6 Thread pool

Avere un numero illimitato di *thread* presenti nel sistema potrebbe esaurire le risorse del sistema stesso, per questo si preferisce creare un numero ben definito di *thread* (**worker**) al momento della creazione del processo e organizzarli in un gruppo detto **thread pool** in cui ogni *thread* attende che gli venga chiesto di completare un lavoro. I vantaggi sono:

- ❖ Eliminare l'attesa della creazione di un uovo *thread*;
- ❖ Limitare il numero di *thread* a ciascun processo alla dimensione prestabilita;
- ❖ Separare il task da eseguire dal meccanismo di creazione dello stesso.

5.7 Strategie di threading

Esistono due tipi threading:

1. **Threading asincrono**: ogni *thread* viene eseguito in modo indipendente rispetto agli altri e il genitore non ha necessità di sapere quando terminano;
2. **Threading sincrono**: i *thread* figli si eseguono in concorrenza e quando un *thread* ha terminato il proprio compito si unisce al genitore, il genitore può combinare quindi i risultati calcolati dai figli e riprende la sua esecuzione nel momento in cui terminano tutti i *thread*.

Data la complessità di scrittura delle applicazioni *multithread*, esiste una tecnica detta **threading implicito** che trasferisce la responsabilità di gestione del *threading* alle **librerie runtime**.

5.8 System call fork() ed exec()

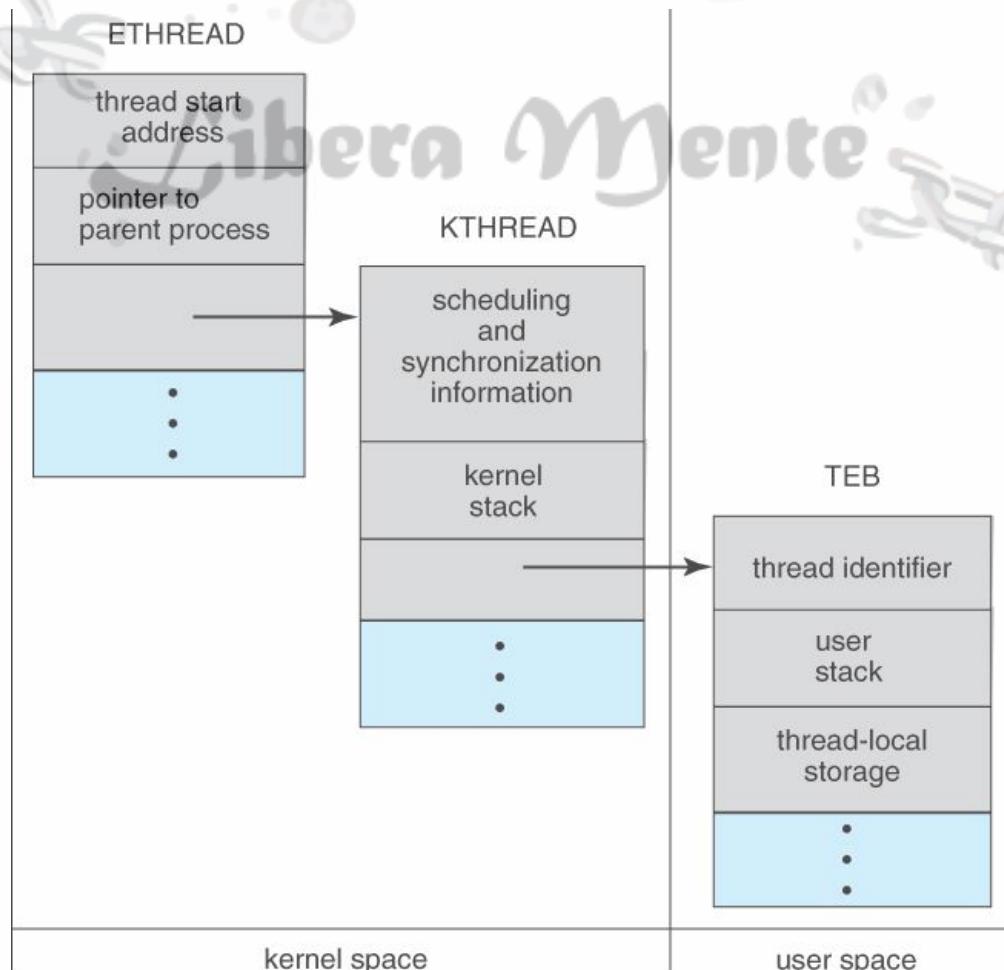
Nel caso delle **applicazioni multithreading**, se un *thread* invoca una `fork()`, il nuovo processo potrebbe contenere un duplicato di tutti i *thread* oppure del solo *thread invocante*: se la system call `exec()` avviene immediatamente dopo la `fork()`, la duplicazione dei *thread* non è necessaria e quindi viene duplicato solo il *thread chiamante*, in caso contrario tutti i *thread* devono essere duplicati.

5.9 Thread nel sistema Windows

L'insieme dei registri, le pile e la memoria privata è detto **contesto del thread**. Le strutture dati principali di un *thread* includono:

- ❖ **ETHREAD (executive thread block)**: formato da un puntatore al processo in cui il *thread* appartiene e l'indirizzo della funzione in cui il *thread* assume il controllo. La struttura *ETHREAD* contiene anche un puntatore alla struttura *KTHREAD*.
- ❖ **KTHREAD (kernel thread block)**: include informazioni per il *thread* relativa allo scheduling e alla sincronizzazione, inoltre contiene la pila del kernel (usata quando il *thread* viene eseguito in modalità kernel) e un puntatore alla struttura *TEB*.
- ❖ **TEB (thread environment block)**: appartiene allo spazio utente e vi si accede quando il *thread* eseguito in modalità utente, inoltre, *TEB* contiene una pila per la modalità utente e un vettore per **dati specifici del thread** che Windows XP chiama **memoria locale del thread** (*thread local storage*).

Le strutture *ETHREAD* e *KTHREAD* risiedono nello spazio del kernel; ciò implica che solo il kernel vi può accedere.



Programma multithread che impiega le API Pthreads

```
#include <pthread.h>
#include <stdio.h>

int sum; /* questo dato è condiviso dai thread */
void *runner(void *param); /* il thread */

int main (int argc, char *argv[]){
    pthread_t tid; /* identificatore del thread */
    pthread_attr_t attr;

    if (argc != 2) {
        fprintf (stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi (argv[1]) < 0) {
        fprintf (stderr, "%d must be >= 0\n", atoi (argv[1]));
        return -1;
    }

    /* reperisce gli attributi predefiniti */
    pthread_attr_init (&attr);

    /* crea il thread */
    pthread_create (&tid, &attr, runner, argv[1]);

    /* attende la terminazione del thread */
    pthread_join (tid, NULL);

    printf ("sum = %d\n", sum);
}

/* il thread assume il controllo da questa funzione */
void *runner (void *param) {
    int i, upper = atoi (param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit (0);
}
```

Sincronizzazione dei processi (Capitolo 6)

Ricorda:

- ❖ Un processo è **indipendente** se la sua esecuzione non influisce sull'esecuzione di altri processi.
- ❖ Un processo è **cooperante** se può influenzarne un altro in esecuzione o subirne l'influenza. Un processo cooperante può condividere uno spazio logico di indirizzi (cioè codice e dati) attraverso l'uso dei *thread*, oppure condividere i dati solo attraverso i figli.

La condivisione dei dati determina la natura cooperante di un processo, viceversa la mancanza di condivisione dei dati determina la natura indipendente del processo.

6.1 Introduzione

Nei **processi cooperanti** l'accesso concorrente ai dati condivisi può generare inefficienza, per garantire la consistenza dei dati si utilizzano meccanismi di **sincronizzazione**.

La soluzione con *memoria condivisa* e *buffer limitato* al problema del produttore consumatore prevede la presenza nel buffer di **BUFFER_SIZE - 1** elementi.

Una soluzione in cui vengano utilizzati tutti gli elementi del buffer necessita di un **contatore** (inizializzato a 0) che aiuti a identificare la posizione libera nel buffer, in particolare questo contatore va incrementato ogni volta che viene inserito un elemento e decrementato ogni volta che si preleva un elemento dal buffer. In particolare, le istruzioni di incremento e decremento del contatore devono essere **atomiche**.

L'aggiornamento del contatore in linguaggio macchina non è un'operazione atomica ma composta da diverse istruzioni: se il produttore e il consumatore tentano di accedere al buffer contemporaneamente le istruzioni in linguaggio macchina possono risultare **interfogliate**, e quindi la sequenza corretta dipende da come i processi del produttore e del consumatore vengono schedulati.

Per evitare le situazioni in cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall'ordine degli accessi (ovvero **le race condition**), occorre assicurare che un solo processo alla volta possa modificare la variabile **contatore**. Questa condizione richiede una forma di sincronizzazione dei processi. Tali situazioni si verificano spesso nei sistemi operativi con componenti diversi che compiono operazioni su risorse diverse.

6.1.2 Race condition

La **race condition** si verifica quando processi cooperanti accedono ai dati condivisi modificandoli, e il risultato dipende dall'ordine nel quale questi processi vengono schedulati. Per evitare *race condition* i processi vanno *sincronizzati*:

- un numero arbitrario di processi compete per l'accesso ai dati condivisi;
- ogni processo ha una sezione di codice, chiamata **sezione critica**, dove accede ai dati condivisi;
- ogni processo può entrare nella propria sezione critica solo se nessun altro processo sta eseguendo una sezione critica analoga;
- l'esecuzione di sezioni critiche da parte di processi cooperanti è mutuamente esclusiva.

La soluzione è progettare un protocollo di cooperazione per l'accesso alla sezione critica.

6.2 Problema della sezione critica

Si consideri un sistema composto di n processi $\{P_0, P_1, \dots, P_{n-1}\}$ ciascuno avente un segmento di codice, chiamato **sezione critica** (detto anche *regione critica*), in cui il processo può modificare variabili comuni, aggiornare una tabella, scrivere in un file e così via.

Quando un processo è in esecuzione nella propria *sezione critica*, gli altri processi non devono essere in esecuzione nella propria *sezione critica*. Quindi, l'esecuzione delle *sezioni critiche* da parte dei processi è **mutuamente esclusiva** nel tempo.

Il problema della *sezione critica* si affronta progettando un protocollo che i processi possono usare per cooperare.

La **sezione d'ingresso** è una parte del codice che fa in modo che ogni processo deve chiedere il permesso per entrare nella propria *sezione critica*.

La *sezione critica* può essere seguita da una **sezione d'uscita**, e la restante parte del codice è detta **sezione non critica**.

Una soluzione del problema della selezione critica deve soddisfare tre requisiti:

- **Mutua esclusione**: se un processo è in esecuzione nella sua *sezione critica*, nessun altro processo può eseguire la propria *sezione critica*;
- **Progresso**: se nessun processo è in esecuzione nella propria *regione critica* e qualche processo desidera accedervi: la scelta del processo che entrerà prossimamente non può essere rimandata indefinitamente;
- **Attesa limitata**: se un processo ha richiesto l'ingresso nella sua *sezione critica*, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive *sezioni critiche* prima che si accordi la richiesta del primo processo.

```
do {  
    Sezione d' ingresso  
    Sezione critica  
    Sezione di uscita  
    Sezione non critica  
} while (true);
```

Si suppone che ogni processo sia seguito a una velocità diversa da zero. Tuttavia, non si può fare alcuna ipotesi sulla **velocità relativa** degli n processi.

Soluzione critica e kernel

In un dato momento, numerosi processi in *user mode* possono essere attivi nel SO e se ciò si verifica, il codice del kernel si trova a dover regolare gli accessi ai dati condivisi. Le due strategie principali per la gestione delle *sezioni critiche* nei SO prevedono l'impiego di:

1. **Kernel con diritto di prelazione**: consente che un processo funzionante in kernel mode sia sottoposto a prelazione, rinviandone l'esecuzione. Questi kernel sono critici nei sistemi multiprocessore SMP, poiché in tali ambienti due processi nel *kernel mode* possono essere eseguiti in contemporanea su processori differenti. I kernel con diritto di prelazione sono fondamentali nella programmazione *real-time* per far valere il diritto di precedenza dei processi, ma sono anche utili nei *sistemi time-sharing* per diminuire il tempo medio di risposta.

2. **Kernel senza diritto di prelazione:** non presentano i problemi legati all'ordine di accesso alle strutture dati del kernel, dato che un solo processo per volta impegnà il kernel. Tale kernel non consente di applicare la prelazione ad un processo attivo in *kernel mode*: l'esecuzione di questo processo durerà finché uscirà da tale modalità, si blocchi o ceda il controllo alla CPU.

6.3 Soluzione di Peterson (applicata a 2 Processi)

Una soluzione software al problema della *sezione critica* è la **soluzione di Peterson**, applicata a due processi P_0 e P_1 , ognuno dei quali esegue alternativamente la propria *sezione critica* e *non critica*.

Se P_i denota uno dei processi, P_j denota l'altro; ossia che $j = 1 - i$.

I due processi condividono due variabili:

```
int turn;
boolean flag[2];
```

La variabile **turn** indica "il turno" del processo per accedere alla propria sezione critica: quindi per dimostrare che la mutua esclusione è preservata si osservi che P_i è autorizzato ad eseguire la propria sezione critica solo se

flag[j] = false oppure turn == i.
L'array **flag** viene utilizzato per vedere se il processo è pronto per entrare nella *sezione critica*. Poiché P_i non modifica il valore della variabile **turn** durante l'esecuzione dell'istruzione **while**, P_i entrerà nella sezione critica (**progresso**) dopo che P_j abbia effettuato non più di un ingresso (**attesa limitata**).

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        sezione critica
    flag[i] = false;
    sezione non critica
} while (true);
```

6.4 Soluzione per più processi: algoritmo del fornaio

L'**algoritmo del fornaio** risolve il problema della *sezione critica* per n processi. È basato su uno schema di servizio usato nella panetteria dove si deve evitare la confusione dei turni. Al suo ingresso nel negozio ogni cliente riceve un numero. Si serve progressivamente il cliente con il più basso. A parità di numero si serve il cliente con il nome minore.

6.5 Hardware per la sincronizzazione (lock)

Qualunque soluzione al problema della *sezione critica* richiede l'uso di un semplice strumento detto **lock** (*lucchetto*). Il corretto ordine degli accessi alle strutture dati del kernel è garantito dal fatto che le sezioni critiche sono protette da lock. In altri termini un processo per accedere alla propria sezione critica deve ottenere il permesso di un lock.

In un sistema dotato di una singola CPU il problema della *sezione critica* si potrebbe risolvere se si potessero proibire le interruzioni mentre si modificano le variabili condivise. In questo modo si assicurerebbe un'esecuzione ordinata e senza prelazione. Questo è l'approccio seguito dai kernel senza diritto di prelazione.

Questa soluzione non è sempre praticabile infatti può comportare sprechi di tempo. Molte moderne architetture offrono istruzioni che permettono di controllare e modificare il contenuto di una parola di memoria, o scambiare il contenuto di due parole di memoria, in modo **atomico** (cioè come un'unità non soggetta ad interruzioni). Queste istruzioni sono utilizzabili per risolvere il problema della *sezione critica*.

Tra le varie architetture, distinguiamo:

- **sistemi monoprocesso**: abbiamo solo la necessità di evitare interruzioni durante la modifica di variabili condivise, ovvero il codice che modifica le variabili condivise viene eseguito senza diritto di prelazione.
- **sistemi multiprocessori**: nei sistemi multiprocessori una soluzione più efficiente è implementare istruzioni atomiche implementate a livello hardware, in particolare, un'istruzione **TestAndSet** che permette di controllare e modificare il contenuto di una parola in memoria, oppure **CompareAndSwap** che permette di scambiare il contenuto di due parole in memoria.

6.6 Semafori

Le varie soluzioni hardware relative al problema della *sezione critica* basate su istruzioni quali *TestAndSet* e *Swap* complicano l'attività del programmatore. Per risolvere questo problema si fa uso dei **semafori**. I semafori sono utilizzati per risolvere il problema della sezione critica con n processi. Gli n processi condividono un semaforo comune, *mutex*, inizializzato a 1.

Un **semaforo S** è una variabile intera che si può accedere solo tramite due operazioni atomiche: **wait** e **signal**. Tutte le modifiche del semaforo sono contenute nelle operazioni **wait** e **signal**: mentre un processo cambia il valore del semaforo nessun altro processo può modificare quello stesso valore.

6.6.1 Uso dei semafori

Ci sono due tipi di semafori:

1. **Semaforo contatore**: intero che può assumere valori in un dominio illimitato in particolare il semaforo è inizialmente inizializzato al numero di risorse disponibili, i processi che invocano `wait()` desiderano utilizzare un'istanza della risorsa e decrementano il semaforo, mentre i processi che rilasciano la risorsa invocano `signal()` e incrementano il semaforo. Quando un semaforo vale 0 e si invoca una `wait()` bisogna attendere che qualche processo rilasci una risorsa.
2. **Semaforo binario**: assume soltanto valori 0 o 1. I semafori binari sono anche detti **lock mutex** perché fungono da “*lock*” che garantiscono la **mutua esclusione**. Il **mutex** indica un procedimento di sincronizzazione fra processi o *thread* concorrenti con cui si impedisce che più task paralleli accedano contemporaneamente ai dati in memoria o ad altre risorse soggette a *race condition*.

6.6.2 Realizzazione

Il principale svantaggio dell'uso dei semafori è che richiede una condizione di **attesa attiva (busy waiting)**. Mentre un processo si trova nella propria *sezione critica*, qualsiasi altro processo che tenti di entrarvi si trova sempre nel ciclo del codice della *sezione di ingresso*. Questo costituisce un problema per un sistema con multiprogrammazione poiché l'**attesa attiva spreca cicli alla CPU**. Questo tipo di semaforo è anche detto **spinlock**, perché i processi “girano” (*spin*) mentre attendono il semaforo. Questi semafori hanno il vantaggio di non richiedere il cambio di contesto nel caso in cui un processo sia fermo in attesa. I semafori **spinlock** sono utili quando i lock sono applicati per brevi intervalli di tempo e vengono implementati nei sistemi multiprocessore, dove un processo gira su un processore mentre un altro *thread* esegue la propria sezione critica su un altro processore.

Per evitare l'**attesa attiva** si può implementare una coda di processi in attesa. Il semaforo, quindi, contiene una struttura contenente: un valore intero (numero di processi in attesa) e un puntatore alla

testa della lista dei processi. Per l'implementazione di tale lista è necessario il supporto del SO, in particolare sono necessarie due system call:

- **block**, posiziona il processo che richiede di essere bloccato in coda e lo sospende;
- **wakeup**, rimuove un processo dalla coda d'attesa e lo sposta in ready.

Quando un processo deve attendere, anziché entrare nell'*attesa attiva*, si blocca e quindi la CPU sceglie un altro processo. Un **processo bloccato** che attende a un semaforo si riavvia attraverso l'operazione `wakeup`. L'operazione `signal` preleva un processo da tale lista e lo attiva. I semafori devono essere eseguiti in modo **atomico**.

Nei sistemi multiprocessore è necessario disabilitare le interruzioni di tutti i processori perché altrimenti le istruzioni dei diversi processi in esecuzione su processori distinti potrebbero interferire fra loro. Le operazioni `wait` e `signal` non eliminano completamente l'*attesa attiva* ma si limitano a rimuoverla dalla sezione di ingresso.

6.6.3 Stallo e attesa indefinita (deadlock)

La realizzazione di un semaforo con coda di attesa potrebbe condurre a situazioni in cui un processo attende l'esecuzione di una `signal()` che può essere innescata solo da uno degli altri processi in coda. Più in generale quando uno o più processi attendono indefinitamente un evento che può essere causato da altri processi in attesa si verifica una situazione di **deadlock o stallo**. Un insieme di processi è in stallo se ciascun processo dell'insieme attende un evento che può essere causato solo da un altro processo dell'insieme.

Un'altra questione connessa alle situazioni di stallo è quella dell'**attesa indefinita**, o **starvation**. Con questo termine indichiamo una situazione che si può presentare se i processi si aggiungono e si rimuovono dalla lista associata a un semaforo secondo un criterio *LIFO*.

Memoria centrale (Capitolo 7)

7.1 Introduzione

Uno dei risultati dello scheduling della CPU consiste nella possibilità di migliorare sia l'utilizzo della CPU sia la rapidità con cui il calcolatore risponde ai propri utenti. Per ottenere questo aumento delle prestazioni occorre tenere in memoria parecchi processi: la memoria deve essere condivisa.

La memoria è formata da un vettore di parole o byte, ciascuno con il proprio indirizzo. La CPU preleva le istruzioni dalla memoria attraverso il contenuto del **Program counter**. Un tipico ciclo di esecuzione prevede che l'istruzione sia prelevata dalla memoria, decodificata ed eseguita, i risultati vengono poi memorizzati in memoria. La memoria prevede solo un flusso di indirizzi, non sa come sono generati oppure a cosa servono.

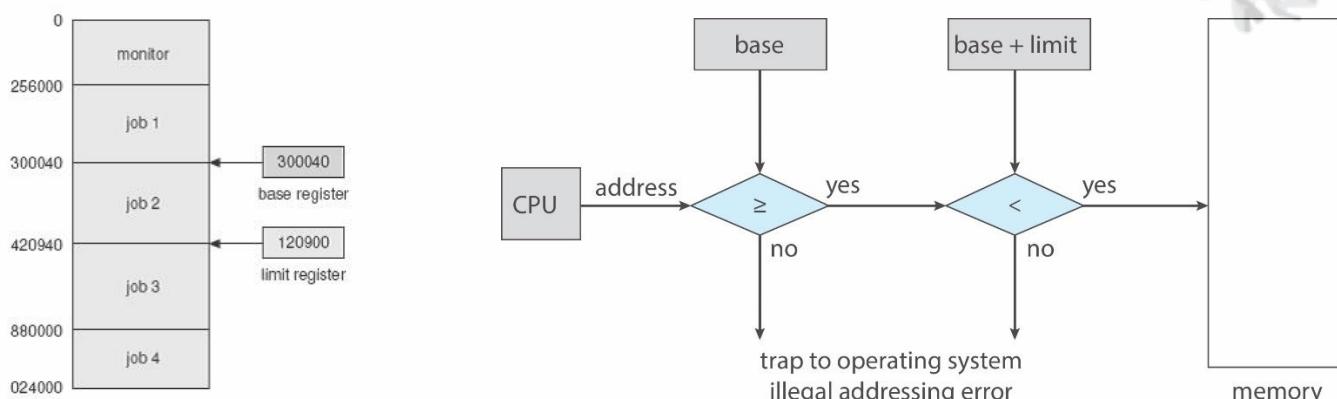
7.2 Dispositivi essenziali

La CPU ha accesso **SOLO** ai registri e alla memoria centrale. La CPU accede ai registri in un **ciclo di clock**; nei casi in cui l'accesso alla memoria centrale richiede diversi *cicli di clock*, il processore entra in una **fase di stallo**: per prevenirla si introduce un altro livello di gerarchia di memoria chiamato **cache** situata tra la memoria principale ed i registri della CPU.

È importante, anche, proteggere il sistema dall'accesso dei processi utenti. La protezione della memoria è fondamentale per l'utilizzo del sistema ed è garantita a livello hardware e non del SO. Innanzitutto, bisogna assicurarsi che ciascun processo abbia uno spazio di memoria separato. Si può implementare il meccanismo di protezione tramite due registri:

- **registri base**: contiene il più piccolo indirizzo legale della memoria fisica;
- **registri limite**: determina la dimensione dell'intervallo ammesso.

Per mettere in atto il meccanismo della protezione, la CPU confronta ciascun indirizzo generato in *user mode* con i valori dei due registri. Qualsiasi accesso di un processo utente ad un'area di memoria riservata al sistema operativo, genera una **trap** e il SO la gestisce come un **errore fatale**. Solo il SO può caricare i *registri base* e *limite* grazie a una **istruzione privilegiata**, tutto ciò è fatto in *kernel mode*.



Registri base e limite

Protezione hardware tramite registri base e limite

7.2.1 Associazione degli indirizzi

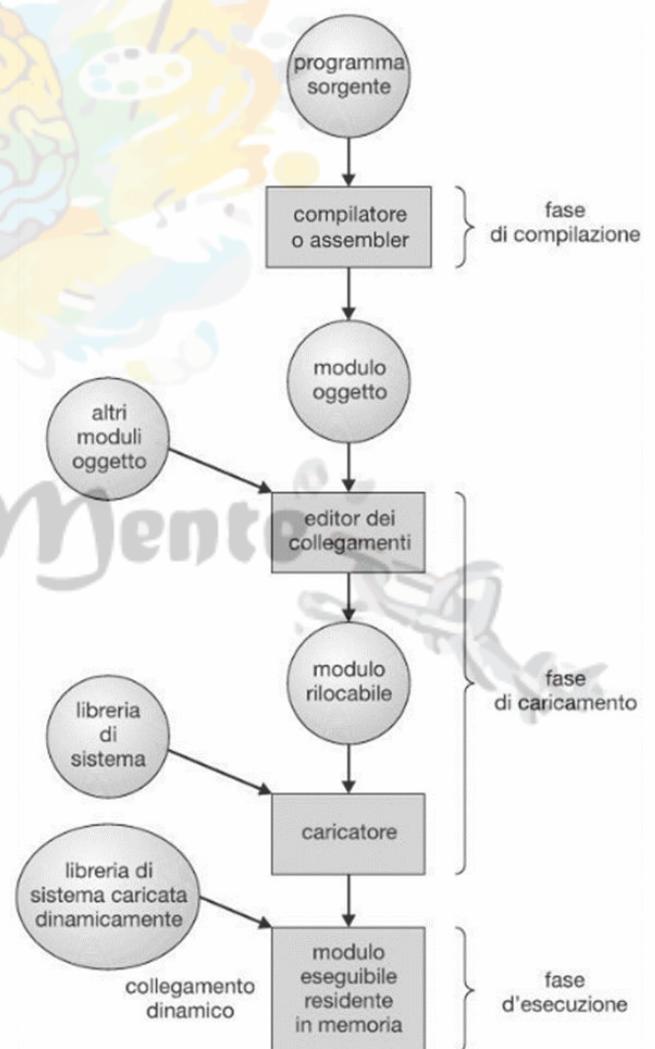
In generale un programma risiede in un disco sotto forma di file binario. Per essere eseguito, il programma va caricato in memoria e inserito all'interno di un processo. Durante la sua esecuzione, il processo si può trasferire dalla memoria al disco e viceversa. L'insieme dei processi presenti nei dischi e che attendono d'essere trasferiti in memoria per essere eseguiti, forma la **coda di ingresso** (*input queue*). La procedura normale consiste nello scegliere uno dei processi appartenenti alla *coda di ingresso* e caricalo in memoria.

Gli indirizzi di un programma si possono classificare in:

- **indirizzi simbolici**, contenuti nel programma sorgente (es. nome della variabile);
- **indirizzi rilocabili**, che il compilatore associa agli indirizzi simbolici;
- **indirizzi assoluti**, che il *linker* o il *loader* fa corrispondere agli indirizzi rilocabili.

Generalmente l'**associazione** (*binding*) di istruzione e dati a indirizzi di memoria si può compiere in qualsiasi fase del seguente percorso:

- ❖ **Compilazione**: può essere applicato se già si sa a priori la posizione del processo in memoria, e al momento della compilazione viene generato **codice assoluto**;
- ❖ **Caricamento**: se la posizione dove sarà caricato il programma non è nota a priori ed è necessario che il compilatore generi **codice rilocabile**; il *loader* poi convertirà gli indirizzi simbolici in indirizzi assoluti.
- ❖ **Esecuzione**: se il processo può essere spostato *run-time* da una parte all'altra della memoria, il *binding* viene rimandato fino all'esecuzione: si avrà **codice dinamicamente rilocabile**. Questo tipo di *binding* necessita di un supporto hardware opportuno per il **mapping degli indirizzi**.



7.2.2 Differenze tra spazi di indirizzi logici e fisici

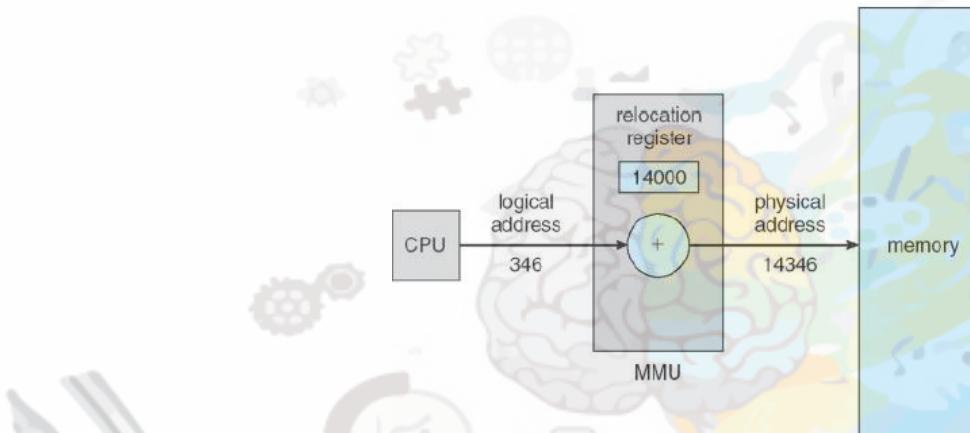
Un indirizzo generato dalla CPU si indica come **indirizzo logico**, mentre un indirizzo visto dalla memoria, cioè caricato nel **registro dell'indirizzo di memoria** (*memory address register, MAR*) di solito si indica come **indirizzo fisico**. Nelle fasi di compilazione e di caricamento vengono prodotti indirizzi logici e fisici identici.

Nella fase d'esecuzione vengono prodotti *indirizzi logici* che non coincidono con gli *indirizzi fisici*. In questo caso gli indirizzi logici vengono chiamati **indirizzi virtuali**.

L'insieme di tutti gli *indirizzi logici* generati da un programma è lo **spazio degli indirizzi logici**, quelli degli *indirizzi fisici* è detto **spazio degli indirizzi fisici**.

L'associazione nella fase d'esecuzione è svolta da un dispositivo hardware detto **unità di gestione della memoria (MMU)**, il quale mappa gli *indirizzi virtuali* in *indirizzi fisici* run-time. La soluzione più semplice implementabile dalla *MMU* è la seguente: il valore del registro di base, chiamato **registro di rilocazione**, sommato all'indirizzo logico genera **l'indirizzo fisico**.

Il programma utente tratta gli indirizzi logici (da 0 a max) mentre l'architettura del sistema converte gli indirizzi logici in indirizzi *fisici reali* (intervallo da **r + 0** a **r + max** per un valore di base **r**).



7.2.3 Loader dinamico (DDL)

Per migliorare l'utilizzo della memoria si può utilizzare il **loader dinamico** mediante il quale si carica una procedura solo quando viene richiamata; tutte le procedure si tengono in memoria secondaria in un formato di *caricamento rilocabile*. Il vantaggio dato dal *loader dinamico* consiste nel fatto che una procedura che non si utilizza non viene caricata. Il *loader dinamico* viene implementato mediante software modulare, e mediante librerie fornite dal SO.

7.2.4 Collegamento dinamico e librerie condivise (DLL)

Esistono due tipi di linking:

- **Linking statico**: le librerie di sistema e il codice del programma sono entrambe presenti nel file binario del programma.
- **Linking dinamico**: viene caricata dinamicamente una libreria in fase di esecuzione. Il codice eseguibile viene separato in librerie in modo da caricare solo la libreria necessaria. Inoltre, una singola libreria, caricata in memoria, può essere utilizzata da più programmi, senza la necessità di essere nuovamente caricata, il che permette di risparmiare le risorse del sistema. Un altro vantaggio è la possibilità di aggiornare un programma modificando solo le **DLL**.

Il **principale svantaggio** è legato al fatto che una nuova versione di una *DLL* potrebbe effettuare dei cosiddetti **breaking changes**, cioè un cambiamento critico nel comportamento del codice della funzione che la rende non più compatibile con le convenzioni in uso (ad esempio, una funzione che prima restituiva *NULL* in caso di errore nei parametri e che ora restituisce un valore non nullo).

A differenza del *loader dinamico*, il **collegamento dinamico** prevede un supporto del SO per bypassare la protezione fra processi che condividono segmenti di codice in specifiche aree di memoria.

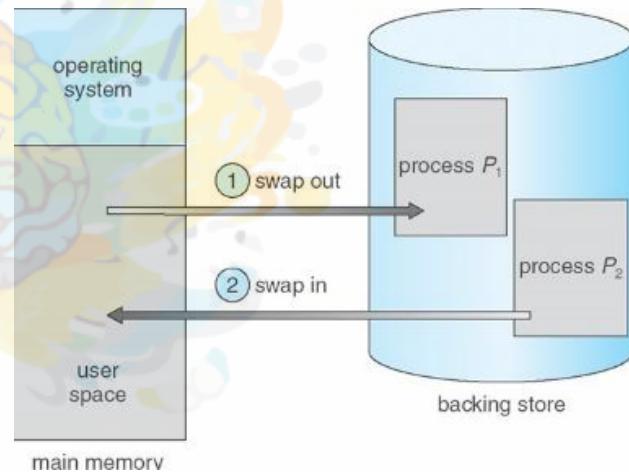
7.3 Swapping

Un processo può essere spostato temporaneamente in una partizione della memoria secondaria, detta **backing store**, dalla quale in futuro potrà essere nuovamente ottenuto e spostato in memoria centrale per l'esecuzione. Questa tecnica si chiama **swapping** (*avvicendamento dei processi in memoria*) il vantaggio è che consente di sovrascrivere la memoria fisica, in modo che il sistema possa ospitare più processi rispetto alla quantità di memoria fisica che abbiamo a disposizione. Lo swapping è controllato dallo **scheduling a medio termine**. Lo swapping è molto comune nei sistemi con schedulatore RR, in cui il processo che finisce il *quanto di tempo* subisce lo **swap-out**.

Esiste una variante dello *swapping* detta **roll out**, **roll in** che è ottimale per gli algoritmi di scheduling a priorità, poiché gli algoritmi con bassa priorità vengono spostati in memoria secondaria così da permettere ai processi di priorità maggiore di essere caricati ed eseguiti.

Lo *swapping* dei processi richiede una **memoria ausiliaria**. Tale *memoria ausiliaria* deve essere abbastanza ampia da contenere le copie di tutte le immagini di memoria di tutti i processi utenti.

Lo *swapping* aumenta il tempo di *context switching* nel caso in cui un processo viene spostato dalla memoria secondaria. I processi in fase *wait* possono essere spostati su disco solo con il **double buffering**. Lo *swapping* deve tenere conto anche di possibili I/O: se i processi sono impegnati in un *I/O asincrono* non possono essere *swappati*. In conclusione, lo *swapping semplice* è oggi poco usato poiché richiede un elevato tempo di gestione e consente un tempo di esecuzione troppo breve per i processi.



Swapping di due processi

7.4 Allokazione contigua della memoria

La memoria centrale deve contenere sia il SO che i vari processi che si vogliono eseguire. Di solito si divide in due partizioni, una per il SO e l'altra per i processi utenti. Il SO si può collocare in **memoria bassa** che in quella alta ma normalmente viene collocata in memoria bassa vicino al vettore delle interruzioni.

7.4.1 Allokazione della memoria

La memoria centrale può essere divisa in diverse partizioni:

A partizioni singole

La parte di memoria disponibile non è partizionata ed è allocata ad un unico processo, **non c'è multiprogrammazione**.

A partizioni multiple

- Partizioni fisse

Il metodo a **partizioni fisse** è il modo più semplice per l'allokazione della memoria: consiste nel suddividere la memoria in partizioni di dimensione fissa. Ogni partizione deve contenere esattamente un processo quindi il grado di multiprogrammazione è limitato dal numero di partizioni.

Quando un processo deve essere caricato in memoria, il SO cerca una partizione libera sufficientemente grande, quindi la dimensione massima dei processi è determinata dalla partizione più grande.

Nello schema a partizione fissa il SO conserva una tabella in cui sono indicate le porzioni di memoria disponibili e quelle occupate. Inizialmente tutta la memoria è a disposizione dei processi utenti, si tratta di un grande blocco di memoria disponibile, chiamato **buco** (*hole*). Quando si carica un processo che necessita di memoria, occorre cercare un *buco* sufficientemente grande da contenerlo. Se ne esiste uno si assegna solo la parte di memoria necessaria, la rimanente la si riporta nell'insieme dei buchi.

Quando i processi entrano nel sistema vengono inseriti in una **coda d'ingresso** gestita con un algoritmo di scheduling. Quando un processo termina si rilascia il blocco di memoria che viene inserito nell'insieme dei buchi liberi e se c'è un buco vicino questo viene accorpato in un unico buco più grande. Questa procedura è una dei più grandi problemi dell'allocazione dinamica della memoria.

Ci sono tre metodi per decidere quale dei buchi di memoria libera sufficientemente grandi utilizzare:

1. **First-fit**: viene allocato il primo buco sufficientemente grande;
2. **Best-fit**: viene allocato il più piccolo buco sufficientemente grande;
3. **Worst-fit**: viene allocato il più grande buco sufficientemente grande.

Best-fit e *Worst-fit* necessitano di scandire tutta la lista dei buchi e tramite l'uso delle **simulazioni** si è dimostrato che sia *first-fit* che *best-fit* sono migliori di *worst-fit*.

- **Partizioni variabili**

Il metodo a **partizioni variabili** prevede che ogni partizione sia allocata a *run-time* in base alla dimensione del processo che deve essere caricato in memoria. Il grado di multiprogrammazione è variabile e la dimensione massima è determinata dai limiti fisici del disco.

7.4.2 Frammentazione

Quando un processo viene caricato e rimosso dalla memoria, lo spazio libero viene diviso in piccoli pezzi. Questi blocchi di memoria sono di piccole dimensioni e non è possibile allocarli per lo stesso processo o qualche altro processo. Pertanto, quei blocchi di memoria rimangono inutilizzati. Questo problema è chiamato **frammentazione**. La **frammentazione** divide uno spazio di memoria libero in sezioni più piccole che non possono essere allocate a nessun processo.

Esistono due tipi di frammentazione denominati **frammentazione interna** ed **esterna**.

- 1) Entrambi i criteri **first-fit** e **best-fit** soffrono della **frammentazione esterna**; si ha quando lo spazio di memoria totale è sufficiente per risiedere in un processo, ma non è continuo, e quindi quello spazio non è utilizzato. La scelta del primo buco abbastanza grande porta uno spreco di un terzo della memoria: questa caratteristica è nota come **regola del 50%**.

Una soluzione al problema della *frammentazione esterna* è data dalla **compattazione**. Lo scopo è quello di riordinare il contenuto della memoria per riunire la memoria libera in un unico grosso blocco. Il più semplice algoritmo di *compattazione* consiste nello spostare tutti i processi verso un'estremità della memoria mentre tutti i buchi vengono spostati nell'altra direzione. Questo metodo può essere molto oneroso.

- 2) Nella **frammentazione interna**, il blocco di memoria assegnato a un processo è più grande del necessario quindi le porzioni rimanenti non possono essere utilizzate per altri processi e diventano così inutilizzate. Una soluzione consiste nell'assegnare partizioni sufficientemente grandi per i processi.

7.5 Paginazione

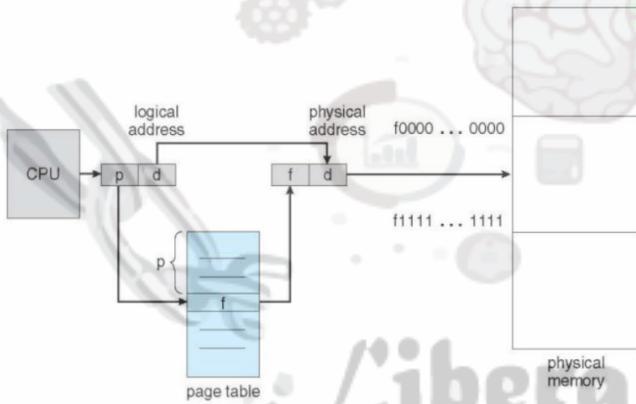
La **paginazione** è un metodo di gestione della memoria che permette che lo spazio degli indirizzi fisici di un processo non sia contiguo. La **paginazione** viene utilizzata come soluzione al problema della **fermentazione esterna**. Inoltre, elimina il problema della sistemazione di blocchi di memoria di diverse dimensioni in memoria ausiliaria. Il problema insorge perché si deve trovare lo spazio necessario in memoria ausiliaria quando alcuni frammenti di codice o dati residente in memoria centrale devono essere scaricati.

7.5.1 Implementazione della paginazione

L'implementazione della **paginazione** consiste nella:

- suddivisione della memoria fisica in blocchi di dimensione fissa, detti **frame** o **pagine fisiche**;
- suddivisione della memoria logica (ovvero lo spazio di memoria dei processi) in blocchi della stessa dimensione, detti **pagine**.

Così facendo si ottiene uno spazio di indirizzi logici totalmente separato dallo spazio di indirizzi fisici. Se un programma necessita di n **pagine** allora sono necessari n **frame**. Per la traduzione da indirizzi logici a fisici si utilizza la **tabella delle pagine**.



Architettura di paginazione

Ogni indirizzo generato dalla CPU è diviso in due parti:

- 1) un **numero di pagina (p)**: che serve come indice per la tabella delle pagine contenente l'indirizzo base in memoria fisica di ogni pagina.
- 2) un **offset (d)**: che combinato con l'indirizzo base, ottenuto mediante il numero di pagina, forma l'indirizzo fisico inviato alla memoria, infatti la paginazione non è altro che una forma di *rilocazione dinamica*.

Con la paginazione si può evitare la **frammentazione esterna**: qualsiasi **frame** libero si può assegnare a un processo che ne ha bisogno. Però c'è il problema della **frammentazione interna** in cui si ha un processo che necessita di n **pagine** + 1 **byte** e si devono allocare n + 1 **pagine** con una frammentazione interna media di mezza pagina per processo. Questo fa sì che si usino pagine di piccole dimensioni.

Un aspetto importante della paginazione è la distinzione tra la memoria vista dall'utente e la memoria fisica: il programma utente vede la memoria come un unico spazio contiguo, contenente anche altri programmi. La differenza è colmata dall'architettura di traduzione degli indirizzi.

Poiché il SO gestisce la memoria fisica, deve avere informazioni su quali **frame** sono stati assegnati e a quale processo e quindi quanti frame ci sono ancora disponibile. Le informazioni sono contenute in una struttura dati chiamata **tabella dei frame**. Il SO conserva una copia della **tabella delle pagine** per ciascun processo e una copia dei valori contenuti nel **PC** e nei registri. Questa copia si usa per tradurre gli indirizzi logici in fisici ogni volta che il SO deve associare un indirizzo fisico a uno logico. La **paginazione** quindi fa aumentare il **context switch**.

7.5.2 Architettura di paginazione e TLB

L'architettura d'ausilio alla *tabella delle pagine* può essere realizzata in modi diversi, nel caso più semplice attraverso dei **registri**.

L'uso dei registri è efficiente se la tabella stessa è piccola, massimo 256 elementi. La maggior parte dei calcolatori usa comunque tabelle di pagine molto grandi quindi non si possono utilizzare registri ma si utilizza un registro che punta alla tabella delle pagine mantenuta in memoria, il **PTBR**. Il cambio della pagina richiede solo l'aggiornamento del puntatore.

Il **PTLR** è il registro che indica la dimensione della tabella. Quindi ogni accesso alla memoria richiede in realtà due accessi uno per la tabella e uno per il dato a cui si è interessati.

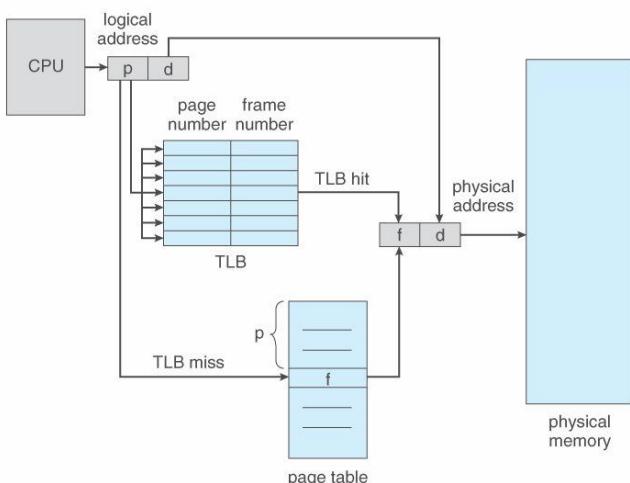
Questo metodo presenta un problema connesso al tempo di accesso a una locazione della memoria utente: per accedere alla locazione di memoria "i" occorre tener presente del valore del registro *PTBR* aumentato del numero di pagina relativo a "i". Si ottiene il numero del *frame* che produce l'indirizzo desiderato.

La soluzione tipica al problema riscontrato per il doppio accesso consiste nell'impiego di una speciale piccola cache ad alta velocità, detta **TLB**, in cui ogni suo elemento consiste di due parti: una **chiave** (*tag*) e un valore che corrisponde al frame. La *TBL* contiene una piccola parte degli elementi della tabella delle pagine, infatti, quando la CPU genera un indirizzo logico, si presenta il suo numero di pagina alla *TBL*, se tale numero è presente, il corrispondente numero del frame è disponibile e si usa per accedere alla memoria. Se non è presente nella *TBL* è noto come **insuccesso nella TBL** (*TLB miss*) e si cerca la tabella delle pagine in memoria.

Se la *TBL* è piena, il SO deve sostituire un elemento scegliendo quello meno recente. Alcune *TBL* memorizzano gli **identificatori dello spazio d'indirizzi** (*ASID*) in ciascun elemento della *TBL*.

Un'ASID identifica in modo univoco ciascun processo e si usa per fornire al processo la protezione del suo spazio di indirizzi. Se la TLB non permette l'uso di ASID diversi, ogni volta che si seleziona una nuova tabella delle pagine si deve cancellare la TLB, in modo da assicurare che il successivo processo in esecuzione non faccia uso di informazioni errate.

La percentuale di volte che un numero di pagina si trova nella TLB è detta **tasso di successi** (*hit rate*).



Architettura di paginazione con TLB

La ricerca nella TLB richiede 20 nanosecondi e sono necessari 100 nanosecondi per accedere alla memoria, allora, supponendo che il numero di pagina si trovi nella TLB, un accesso alla memoria richiede 120 nanosecondi.

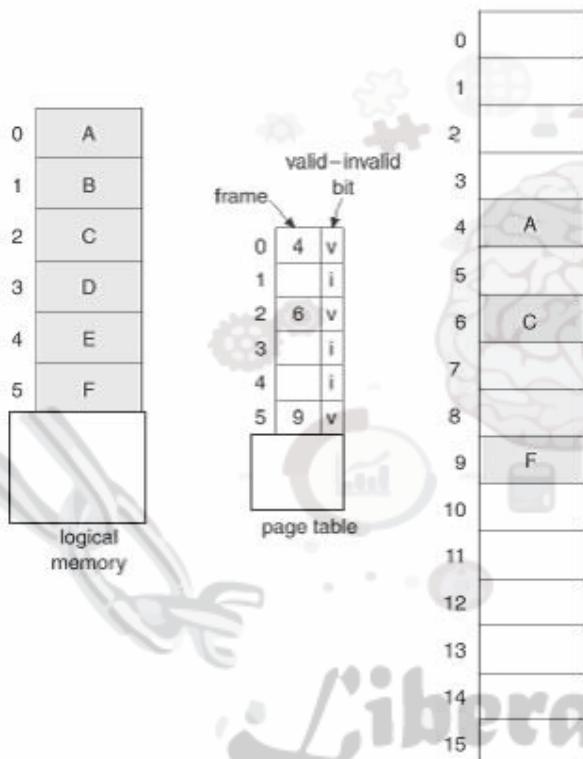
Se, invece, il numero non è contenuto nella TLB (20 nanosecondi) occorre accedere alla memoria per arrivare alla tabella delle pagine e al numero del frame (100 nanosecondi), quindi accedere al byte desiderato in memoria (100 nanosecondi); quindi in totale sono necessari 220 nanosecondi.

7.5.3 Protezione

In un ambiente paginato, la protezione della memoria è assicurata dai **bit di protezione** associati a ogni frame, tali bit si trovano nella tabella delle pagine. Un bit può determinare se una pagina si può leggere e scrivere oppure solo leggere. Mentre si calcola l'indirizzo fisico si possono controllare i bit di protezione per verificare che non si scriva in una pagina di sola lettura. Si può progettare un hardware che fornisca la protezione di sola lettura, di sola scrittura o di sola esecuzione.

Di solito si associa a ciascun elemento della tabella delle pagine un ulteriore bit, detto **bit di validità**:

- Tale bit se impostato a “*valido*”, indica che la pagina è nello spazio degli indirizzi logici del processo;
- Se è impostato a “*non valido*”, indica il contrario.



Il *bit di validità* consente di riconoscere gli indirizzi illegali e di notificarne la presenza attraverso le eccezioni. Capita raramente che un processo faccia uso di tutto il suo intervallo di indirizzi, infatti molti processi utilizzano solo una piccola frazione dello spazio di indirizzi. In questi casi è inutile creare una tabella di pagine poiché una gran parte di questa tabella resta inutilizzata e occupa solo spazio di memoria.

Alcune architetture dispongono di registri, detti **registri di lunghezza della tabella delle pagine (PTLR)** per indicare le dimensioni della tabella.

Bit di validità in una tabella delle pagine

7.5.4 Pagine condivise

Un altro vantaggio della paginazione consiste nella possibilità di condividere codice comune, utile soprattutto in un ambiente time-sharing.

La condivisione del codice avviene tramite l'uso del **codice rientrante**, detto **codice puro**: ovvero un codice non automodificante, cioè non cambia durante l'esecuzione. Quindi, due o più processi possono eseguire lo stesso codice nello stesso momento. Ciascun processo dispone di una propria copia dei registri e di una memoria dove conserva i dati necessari alla propria esecuzione.

Quindi se la memoria centrale è *paginata*, la memoria condivisa si realizza mediante pagine condivise, attraverso due modalità:

- **Codice condiviso**, una copia del *text segment* viene condivisa fra processi, quindi il codice condiviso deve apparire nella stessa locazione degli indirizzi logici di tutti i processi.
- **Codice e dati privati**, ciascun processo mantiene una copia separata dei dati e le pagine di codice e dei dati privati possono apparire ovunque nello spazio degli indirizzi logici.

7.6 Struttura della tabella delle pagine

Analizziamo alcune tecniche più comuni per strutturare la **tabella delle pagine**.

7.6.1 Paginazione Gerarchica

La maggior parte dei moderni calcolatori dispone di uno spazio d'indirizzi logici molto grande (da 2^{32} a 2^{64} elementi). Bisogna, quindi, evitare di collocare la *tabella delle pagine* in modo contiguo in memoria centrale.

Una soluzione a questo problema è la **paginazione gerarchica** che prevede la suddivisione degli indirizzi logici in più tabelle delle pagine. Questa suddivisione si può fare in vari metodi: un metodo semplice consiste nell'adottare un algoritmo di **paginazione a due livelli**, in cui la tabella stessa è paginata.

Caso a 32 bit

Un indirizzo logico a 32 bit verrebbe suddiviso normalmente in:

- **numero di pagina** a 20 bit;
- **offset** a 12 bit;

Dato che la tabella delle pagine è paginata, il numero di pagina viene ulteriormente suddiviso in:

- **numero di pagina** a 10 bit (**tabella esterna**)
- **offset** a 10 bit (**tabella interna**);

numero di pagina	offset
p_1	p_2
10	10

Questo modello è adatto solo ad architetture a **32 bit**, a **64 bit** la **paginazione gerarchica** è inadeguata a causa dei costi proibitivi di accesso alla memoria in caso di TLB miss.

Vantaggi Paginazione a più livelli

- possibilità di indirizzare spazi logici di dimensioni elevate riducendo i problemi di allocazione delle tabelle;
- possibilità di mantenere in memoria soltanto le pagine della tabella che servono.

Svantaggi Paginazione a più livelli

- tempo di accesso più elevato: per tradurre un indirizzo logico sono necessari più accessi in memoria (ad esempio, 2 livelli di paginazione -> 2 accessi).

7.6.2 Tabelle delle pagine hash

Un metodo di gestione degli spazi di indirizzi oltre i 32 bit consiste nell'impiego di una **tabella delle pagine di tipo hash**, in cui l'argomento della *funzione hash* è il numero della pagina virtuale. Per la gestione delle collisioni, ogni elemento della tabella hash contiene una lista concatenata di elementi che la *funzione di hash* fa corrispondere alla stessa locazione. Ciascun elemento è composto da tre campi:

1. il numero della pagina virtuale;
2. l'indirizzo del frame corrispondente alla pagina virtuale;
3. un puntatore al successivo elemento della lista.

L'algoritmo opera in questo modo: i numeri di pagina virtuale vengono confrontati con tutti gli elementi della catena. Se i valori coincidono si usa l'indirizzo del relativo frame per generare l'indirizzo fisico desiderato, altrimenti l'algoritmo esamina allo stesso modo gli elementi successivi della lista concatenata.

La **tabella delle pagine a gruppi** (adatta a spazi di indirizzamento a 64 bit) è simile alla *tabella delle pagine di tipo hash* ma con la differenza che ciascun elemento della tabella hash contiene i riferimenti alle pagine fisiche corrispondenti a un gruppo di pagine virtuali contigue.

Le **tabelle delle pagine a gruppi** sono utili per gli **spazi di indirizzi sparsi**, in cui i riferimenti alla memoria non sono contigui ma distribuiti per tutto lo spazio di indirizzi.

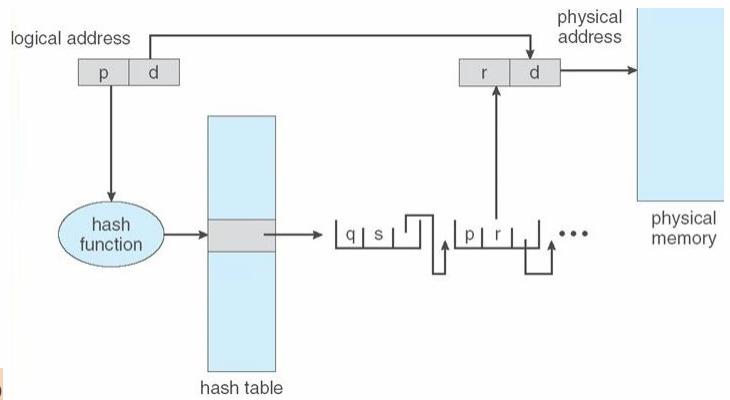


Tabelle delle pagine di tipo hash

7.6.3 Tabelle delle pagine invertita

Generalmente, esiste una tabella delle pagine per ogni processo e tale tabella contiene un elemento per ogni pagina virtuale che il processo sta utilizzando. Questo perché i processi fanno riferimento alle pagine tramite gli indirizzi virtuali delle pagine stesse, che il SO traduce in indirizzi di memoria fisica. Poiché la tabella è ordinata per indirizzi virtuali, il SO può calcolare in che punto della tabella si trova l'elemento dell'indirizzo fisico associato e usare direttamente tale valore.

Uno degli inconvenienti in questo metodo è costituito dalla dimensione di ciascuna tabella delle pagine, che può contenere milioni di elementi e occupare grandi quantità di memoria fisica. Per risolvere questo problema si può fare uso della **tabella delle pagine invertita** che prevede che la tabella delle pagine sia una sola e che ci sia un solo elemento per ogni *frame* e che per ognuno degli elementi, quindi per ogni frame, si tenga traccia dell'indirizzo virtuale della pagina memorizzata con le informazioni sul processo che la possiede.

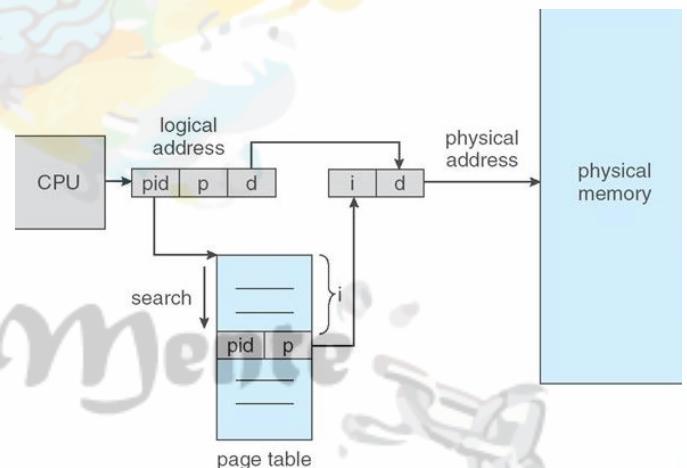


Tabelle delle pagine invertita

Ciascun indirizzo virtuale è una tripla del tipo seguente: **<id processo, numero pagina, offset>**.

Sebbene la **tabella delle pagine invertita** riduca la quantità di memoria necessaria per memorizzare ogni tabella delle pagine, aumenta però il tempo di ricerca nella tabella. Per limitare il problema si può usare una **tabella hash** che riduce la ricerca a un solo o a pochi elementi della tabella delle pagine.

L'utilizzo della *tabella delle pagine invertita* rende difficile l'implementazione della **memoria condivisa** a causa di un solo elemento indicante la pagina virtuale corrispondente a ogni pagina fisica. Una soluzione consiste nel porre nella tabella delle pagine una sola associazione fra un indirizzo virtuale e l'indirizzo fisico condiviso; ciò comporta un errore detto **page fault** che si verifica quando un processo cerca di accedere ad una pagina che è presente nel suo spazio di indirizzamento virtuale, ma che non è presente nella memoria fisica, poiché non è mai stata caricata in essa o perché è stata spostata su disco di archiviazione. Il **page fault** è trattato dal SO come una **trap**.

7.7 Segmentazione

Un aspetto importante della gestione della memoria è la separazione tra la visione della memoria dell'utente e l'effettiva memoria fisica.

La **segmentazione** è uno schema di gestione della memoria che suddivide la memoria fisica disponibile in blocchi di lunghezza fissa o variabile detti **segmenti**.

Un programma è una collezione di segmenti; un segmento è un'*unità logica* come il programma principale, le procedure e le funzioni, le variabili locali e globali, metodi, oggetti e strutture dati. La *segmentazione* riflette quindi la visione che il programmatore ha del proprio programma. Inoltre, un processo segmentato è allocato in memoria per segmenti non necessariamente adiacenti.

- Gli elementi che si trovano all'interno di un segmento sono identificati dal loro **offset**, misurato dall'inizio del segmento.

7.7.1 Architettura di segmentazione

La memoria fisica è una sequenza di byte unidimensionale per questo motivo occorre tradurre gli indirizzi bidimensionali in indirizzi fisici unidimensionali. Questa operazione si svolge tramite la **tabella dei segmenti**, dove ogni suo elemento è una coppia ordinata:

- **base del segmento**: specifica l'indirizzo fisico iniziale della memoria contenente il segmento;
- **limite del segmento**: specifica la lunghezza del segmento.

Ogni **indirizzo logico** è formato dalla coppia <numero di segmento, offset>:

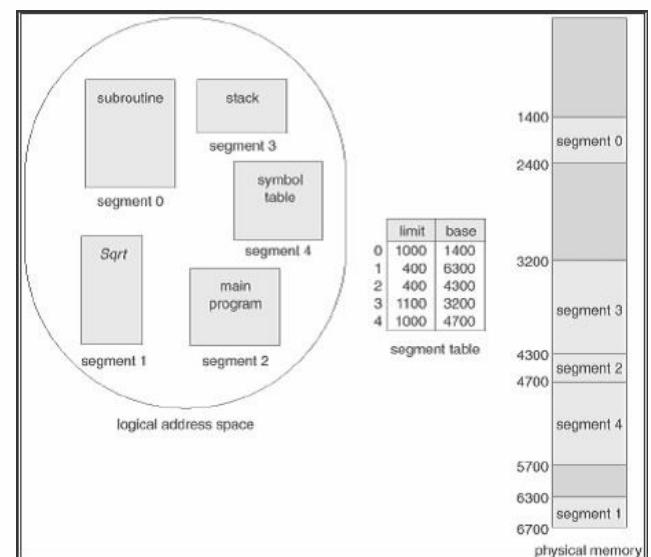
- il **numero di segmento** si usa come indice per la tabella dei segmenti;
- l'**offset** dell'indirizzo logico deve essere compreso tra 0 e il *limite del segmento*, altrimenti il SO genera una trap.

Se tale condizione è rispettata, si somma l'offset alla **base del segmento** per produrre l'indirizzo della memoria fisica dove si trova il byte desiderato. Quindi la tabella dei segmenti è un vettore di coppie di registri di base e limite.

Il registro **STBR** (*segment-table base register*) punta alla locazione in memoria della tabella dei segmenti. Il registro **STLR** (*segment-table length register*) indica il numero di segmenti utilizzati dal programma. Si associano agli elementi della *tabella dei segmenti* due valori:

- **bit di validità**, che se uguale a 0 indica che il segmento è illegale;
- **privilegi**, indica se si hanno permessi di lettura, scrittura ed esecuzione.

Nella segmentazione poiché i segmenti variano di dimensione soffre di **frammentazione esterna**.



7.7.2 Segmentazione vs Paginazione

Nel caso della **paginazione** la suddivisione del programma in blocchi avviene secondo criteri geometrici, la singola pagina non ha alcun significato per il programmatore e non c'è nessun collegamento logico tra il contenuto di un blocco e quello del blocco successivo. Quindi la paginazione è invisibile al programmatore. Nel caso della **segmentazione** un processo è formato da un unico spazio di memoria che è suddiviso in un certo numero di blocchi. Con la **segmentazione** un processo è suddiviso in *segmenti*, ognuno dei quali possiede un proprio spazio di indirizzamento.

Vantaggi segmentazione

I vantaggi della segmentazione rispetto alla paginazione sono:

- La gestione della memoria condivisa è semplificata: i dati (o il codice) da condividere possono essere inseriti in un unico segmento che viene condiviso tra i processi.
- La protezione della memoria può essere effettuata in base alle esigenze e alle caratteristiche funzionali del singolo segmento.
- È più semplice gestire strutture dati dinamiche allocandole in un segmento. Infatti, se nell'esecuzione del processo lo spazio occupato dai dati aumenta, basta aumentare la dimensione di quello specifico segmento. Nel caso della **paginazione** è invece impossibile aumentare la dimensione delle pagine ed è più complicato inserire nuove pagine tra quelle già assegnate al processo per allocare dati che possono crescere fino a superare lo spazio inizialmente previsto.

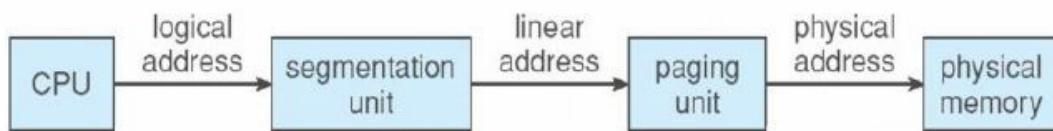
Vantaggi Paginazione

- Pagine logiche contigue che possono essere allocate su pagine fisiche non contigue: **non c'è frammentazione esterna**.
- Le pagine sono di dimensione limitata: la frammentazione interna, per ogni processo è **limitata dalla dimensione del frame**.
- È possibile caricare in memoria un sottoinsieme delle pagine logiche di un processo.

7.8 Pentium Intel

Le architetture **Intel Pentium** supportano sia la segmentazione sia la segmentazione mista a paginazione. In questi sistemi la CPU genera indirizzi logici che confluiscono nell'unità di segmentazione che produce indirizzi lineari.

L'indirizzo lineare passa all'unità di paginazione, la quale, a sua volta, genera l'indirizzo fisico all'interno della memoria centrale. Così, l'unità di segmentazione e di paginazione formano l'equivalente di una **MMU**.



7.8.1 Segmentazione in IA-32

Nelle architetture **Intel a 32 bit** i segmenti possono raggiungere la dimensione massima di 4GB, e lo spazio degli indirizzi logici viene diviso in due partizioni:

- La prima contiene fino a 8KB segmenti riservati e tali informazioni sono contenute nella **LDT: Local Descriptor Table** ;

- La seconda contiene fino a 8KB segmenti condivisi e tali informazioni sono contenute nella **GDT**: **Global Descriptor Table**.

Un indirizzo logico è una coppia

<selettore, offset>, dove il selettore è un numero a 16 bit, composto da tre parti:
numero del segmento (13 bit), indicatore di tipo **LDT** o **GDT** (1 bit) e 2 bit di **protezione**.

<i>s</i>	<i>g</i>	<i>p</i>
13	1	2

Un *indirizzo lineare* del Pentium è 32 bit e si genera come segue: il selettore punta all'elemento nella *LDT* o nella *GDT* e le informazioni alla base ed al limite di tale segmento sono utilizzate per generare l'*indirizzo lineare*.

7.8.2 Paginazione in Pentium

L'architettura Pentium prevede che le pagine abbiano misura di 4KB o 4MB. Per le prime la paginazione è a due livelli: i 10 bit più significativi puntano alla tabella delle pagine esterna, detta **directory delle pagine**, i restanti 12 indicano l'offset all'interno della tabella delle pagine.

Il limite degli indirizzi a 32 bit portò l'Intel a sviluppare l'estensione di indirizzo della pagina **PAE** (*Page Address Extension*) che permise alle applicazioni di accedere a più di 4GB di memoria utilizzando uno schema di **paginazione a 3 livelli**, dove i 2 bit più significativi dell'indirizzo si riferiscono a una tabella di puntatori alla directory delle pagine. Gli elementi della directory delle pagine sono a 64 bit, tra cui 24 bit per gli indirizzi base dei frame.

L'effetto finale è un aumento degli spazi di indirizzi a 36 bit, garantendo un supporto di 64GB di memoria fisica.

7.8.3 Intel 64 bit

Le architetture a **64 bit** rappresentano la generazione attuale dei microprocessori Intel (AMD). Nella pratica implementano una modalità di indirizzamento a 48 bit, la dimensione delle pagine è di 4Kb, 2MB, 1GB e si ha una paginazione gerarchica a 4 livelli. Si possono utilizzare anche le **PAE** cosicché si ottengono 48 bit e indirizzi fisici a 52 bit.

7.8.4 ARM-1

È una CPU a 32 bit particolarmente efficiente per il risparmio energetico, supporta pagine a 4KB, 16KB, 1MB o 16MB chiamate **sezioni**. La paginazione è a singolo livello per le sezioni e a due livelli per le pagine di dimensione minore.

7.8.5 ARM-2

Si hanno due livelli di TLB, due micro TLB, una per dati e una per le istruzioni con **AS/D** e una TLB principale: la traduzione di un indirizzo inizia a livello micro TLB; in caso di insuccesso si controlla la TLB principale e in caso di ulteriore insuccesso si consulta la tabella delle pagine.

Memoria virtuale (Capitolo 8)

8.1 Introduzione

La gestione della memoria ha come scopo tenere contemporaneamente tutti i processi in memoria per permettere la multiprogrammazione, però l'intero processo deve trovarsi in memoria prima di essere eseguito.

La **memoria virtuale** è una tecnica che permette di eseguire i processi che possono anche non essere completamente contenuti in memoria.

Vantaggi

I vantaggi di far risiedere in memoria solo una parte di un processo sono:

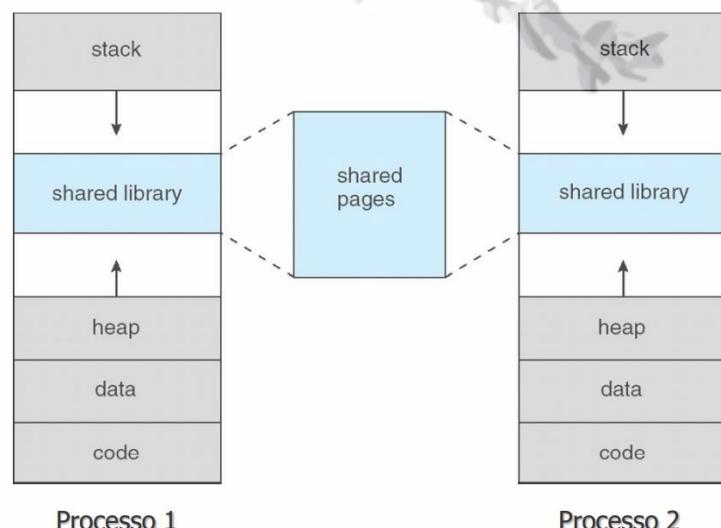
- I programmi non sono più vincolati dalla quantità di memoria fisica, quindi gli utenti possono scrivere programmi per uno **spazio degli indirizzi virtuale** molto grande;
- Possibile aumento del grado di multiprogrammazione perché ogni utente impiega meno memoria fisica quindi si possono eseguire molti più programmi contemporaneamente senza aumentare il tempo di risposta o di completamento;
- Per caricare e scaricare porzioni in memoria sono necessarie meno operazioni di I/O, quindi ogni programma utente è eseguito più rapidamente.

Il concetto di **memoria virtuale** indica la separazione della memoria logica, vista dall'utente, dalla memoria fisica, quindi i programmatori non devono preoccuparsi della quantità di memoria fisica disponibile. I vantaggi sono:

- Solo parte del programma è in memoria centrale;
- Lo spazio logico è più grande di quello dello spazio fisico;
- Porzioni di spazio fisico possono essere condivise;

Oltre questa separazione, la **memoria virtuale** offre, per due o più processi, il vantaggio di condividere i file e la memoria, attraverso la condivisione delle pagine. Ciò comporta vari vantaggi:

- Le librerie di sistema sono condivise mediante la **mappatura** delle pagine logiche di diversi processi sugli stessi frame;
- La memoria virtuale permette ai processi di condividere la memoria creando una regione di memoria condivisa da un altro processo.
- Mediante la condivisione delle pagine i processi possono essere generati rapidamente.



Condivisione delle librerie tramite memoria virtuale

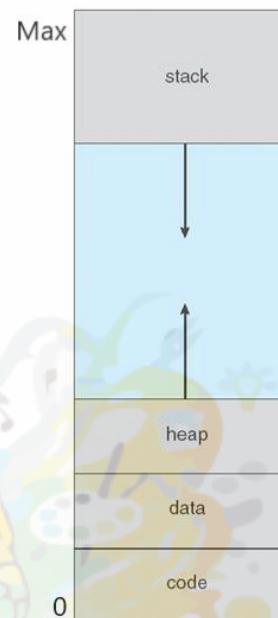
Svantaggi

Lo svantaggio è che la **memoria virtuale** è difficile da realizzare e se usata scorrettamente, può ridurre di molto le prestazioni del sistema.

Lo **spazio degli indirizzi virtuali** si riferisce alla collocazione dei processi in memoria dal punto di vista logico, in particolare: un processo inizia in corrispondenza dell'*indirizzo logico 0* e termina a *Max*. Le pagine sono sparse nei vari *frame* della memoria fisica e spetta alla MMU associare in memoria le pagine logiche alle pagine fisiche.

Lo **heap** cresce verso l'alto nello spazio di memoria, poiché esso ospita la memoria allocata dinamicamente. In modo analogo, la **pila** si sviluppa verso il basso nella memoria, a causa di ripetute chiamate di funzione. Lo **spazio vuoto** (o buco) separa lo *heap* dalla *pila* e richiede pagine fisiche realmente esistenti solo nel caso che lo *heap* o la *pila* crescano. Qualora contenga buchi, lo **spazio degli indirizzi virtuali** si definisce **sparso**.

Uno spazio del genere è utile perché consente di riempire i buchi grazie all'espansione dell'*heap* o *pila*, e di collegare dinamicamente delle librerie durante l'esecuzione del programma.



Spazio degli indirizzi virtuale

La memoria virtuale può essere implementata per mezzo di:

- ❖ Paginazione su richiesta;
- ❖ Segmentazione su richiesta.

8.2 Paginazione su richiesta

Durante l'esecuzione di un programma non è necessario caricarlo interamente in memoria, perché alcune parti del codice possono non servire. Per questo motivo si utilizza la **paginazione su richiesta** che carica le pagine in memoria solo quando vengono richieste durante l'esecuzione del programma: le pagine a cui non si accede non vengono caricate nella memoria fisica, ottimizzando lo spazio. Questa tecnica è adottata dai sistemi con memoria virtuale.

Un sistema di paginazione su richiesta è simile a un sistema paginato con swapping perché, anziché caricare in memoria l'intero processo, si usa uno **swapping pigro** che sposta una pagina in memoria solo quando è necessario. Il modulo del SO che si occupa della sostituzione delle pagine si chiama **pager**.

8.2.1 Come funziona la paginazione su richiesta

La **paginazione su richiesta** ha il seguente funzionamento: quando un processo sta per essere caricato in memoria, il *pager* ipotizza quali pagine saranno usate e anziché caricare tutto il processo trasferisce in memoria solo le pagine che ha ipotizzato essere necessarie. In questo modo si riduce il tempo di *swapping* e la quantità di memoria fisica richiesta.

Per determinare l'insieme di pagine è necessaria una **MMU** in cui:

- Se la pagina richiesta è già presente in memoria, l'esecuzione procede.
- Se la pagina deve essere trovata in memoria secondaria e trasferita in memoria centrale, l'operazione non deve alterare il flusso del programma.

Il meccanismo che permette di riconoscere se le pagine sono residenti in memoria o su disco, è un **bit di validità** presente nella tabella delle pagine:

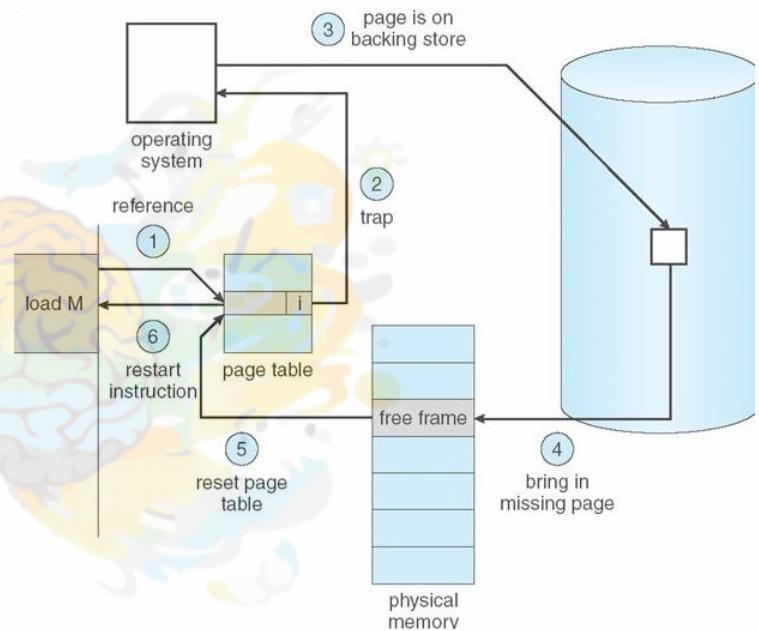
- Se il *bit di validità* è impostato come **valido** (v): la pagina è in memoria;
- Se il *bit di validità* ha valore “**non valido**” (i): la pagina è non valida o non residente.

Inizialmente tutta la tabella della pagina ha valore “*non valido*” per il **bit di validità**.

Se il processo cerca di accedere a una pagina non presente in memoria viene causato un **page fault trap** (eccezione di pagina mancante); viene inviato un segnale al SO il quale provvede a caricularla in memoria.

La **gestione di un page fault** consiste in:

1. Si controlla una tabella interna per questo processo, di solito è conservata nel PCB;
2. Se il riferimento era “*non valido*” si termina il processo. Se era “*valido*” ma la pagina non è in memoria se ne effettua l’inserimento;
3. Si individua un frame libero dove collocare la pagina;
4. Si programma un’operazione sui dischi per trasferire la pagina nel frame libero;
5. Quando la lettura dal disco è completata, si modifica la tabella interna e la tabella delle pagine
6. Si riavvia l’istruzione interrotta dal segnale di eccezione.



Fasi di gestione di una page fault trap

Il **tempo di gestione del page fault** è composto da tre componenti principali:

- Servizio del segnale di eccezione di page fault;
- Lettura pagina da disco;
- Riavvio del processo;

È anche possibile avviare un processo senza pagine in memoria e caricarle quando vengono richieste. I meccanismi d’ausilio alla *paginazione su richiesta* sono:

- ❖ **Tabella delle pagine**. Questa tabella ha la capacità di contrassegnare un elemento come non valido attraverso un bit di validità oppure un valore speciale dei bit di protezione.
- ❖ **Memoria secondaria**. Questa memoria conserva le pagine non presenti in memoria centrale in una sezione del disco chiamata **area di swap**.

Il punto cruciale della *paginazione su richiesta* è che deve essere possibile continuare l’esecuzione nel punto in cui si interrompe l’istruzione per caricare una pagina mancante e riavviare il processo esattamente nel punto in cui è stato interrotto. Il sistema di paginazione si colloca tra la CPU e la memoria di un calcolatore e deve essere trasparente al processo utente.

8.2.2 Lista dei frame liberi

Quando si verifica un *page fault*, il SO deve spostare la pagina desiderata dalla memoria secondaria alla memoria principale. La maggior parte dei SO, per risolvere i *page fault*, usa una **lista di frame liberi** allocati tramite una tecnica detta **zero-fill-on-demand**: i frame vengono “azzerati” su richiesta prima di essere allocati, cancellando così il loro contenuto precedente.

All'avvio del sistema tutta la memoria disponibile viene inserita nella *lista dei frame liberi*. Man mano che vengono richiesti frame liberi, la dimensione della lista si riduce fino a diventare vuota, oppure la sua dimensione scende al di sotto dina una certa soglia fissata: quando si verifica questo la lista deve essere ripopolata.

8.2.3 Prestazioni della paginazione su richiesta

La *paginazione su richiesta* può aver un effetto rilevante sulle prestazioni di un calcolatore basta calcolare il **tempo d'accesso effettivo** per una memoria con *paginazione su richiesta*. Finché non si verifichino *page fault*, il tempo d'accesso effettivo è uguale al **tempo d'accesso alla memoria**. Per calcolare il tempo d'accesso effettivo occorre conoscere il **tempo necessario alla gestione di page fault**.

Alla presenza di *page fault* si esegue la seguente sequenza:

1. Trap del so;
2. Salvataggio dei registri e dello stato del processo;
3. Verifica che l'interruzione sia dovuta ad un *page fault*;
4. Controllo della correttezza del riferimento alla pagina su disco;
5. Lettura dal disco e trasferimento in un frame libero;
 - attesa nella coda al dispositivo di I/O;
 - attesa dovuta al posizionamento e latenza del dispositivo;
 - trasferimento della pagina in un frame libero;
6. Durante l'attesa la CPU viene allocata ad un altro processo: questa situazione permette la multiprogrammazione, ma una volta completato il trasferimento di I/O implica una perdita di tempo per riprendere la procedura del *page fault trap*;
7. I/O completato dal disco;
8. salvataggio dei registri e dello stato dell'altro processo utente;
9. verifica della provenienza dell'interruzione dal disco;
10. aggiornamento della tabella delle pagine e di altre tabelle per segnalare che la pagina richiesta è attualmente presente in memoria;
11. attesa nella ready queue;
12. context switch di accesso alla cpu.

In un sistema con paginazione su richiesta, è importante tenere bassa la **frequenza delle assenze di pagina**, altrimenti il tempo effettivo d'accesso aumenta, rallentando molto l'esecuzione del processo. Si può permettere un'assenza di pagine ogni 399.990 accessi.

Ottimizzazione

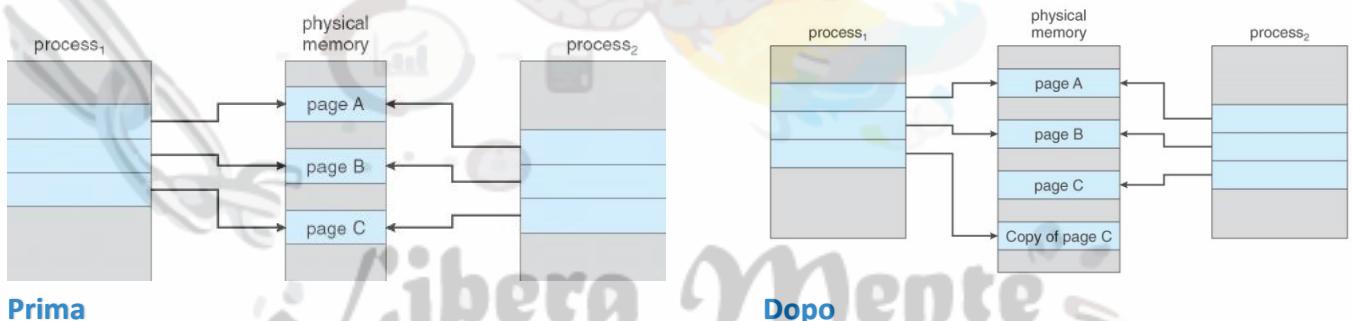
Dato che l'I/O relativo all'*area di swap* è più rapido rispetto al file system, il SO copia tutta l'immagine di un file nell'area di swap all'avvio del processo ed esegue da quel punto la paginazione su richiesta migliorando l'efficienza della paginazione.

8.3 Copiatura su scrittura

Il processo di **copiatura su scrittura** (*copy-on-write*) permette ai processi padre-figlio la condivisione delle stesse pagine di memoria: se il sistema scrive su una pagina condivisa, il sistema deve creare una **copia di tale pagina**. Con tale tecnica si copiano soltanto le pagine modificate da uno dei due processi, mentre le altre sono condivisibili dai processi genitore e figlio.

Per capire da dove si prenderà la pagina libera necessaria, molti SO forniscono un **pool di pagine libere**. L'allocazione di nuove pagine avviene tramite la tecnica di **azzeramento su richieste** (*zero-fill-on-demand*): prima dell'allocazione si riempiono di zeri le pagine, cancellandone in questo modo tutto il contenuto precedente.

Con la **vfork()**, alternativa della **fork()**, il processo genitore viene sospeso e il processo figlio usa lo spazio di indirizzi del genitore. Poiché la **vfork()** non usa la *copiatura su scrittura*, se il processo figlio modifica qualche pagina dello spazio di indirizzi del genitore, le pagine modificate saranno visibili al processo genitore non appena riprenderà il controllo. La **vfork()** è adatta nel caso in cui il processo figlio esegue una **exec()** immediatamente dopo la sua creazione.



8.4 Sostituzione delle pagine

Aumentando il grado di multiprogrammazione si **sovrassegna** la memoria. La **sovrallocazione della memoria** è un fenomeno che si verifica quando è richiesta più memoria di quella effettivamente disponibile, e quindi si verifica un *page fault* con assenza di frame liberi. Si previene la *sovrallocazione* modificando la routine di servizio del *page fault*, includendo la **sostituzione delle pagine**.

La *sostituzione delle pagine* segue il seguente criterio: se nessun frame è libero, ne può liberato uno attualmente inutilizzato. È possibile liberalo scrivendo il suo contenuto nell'area di swap e modificando la tabella delle pagine per indicare che la pagina non si trova più in memoria. Il frame liberato si può utilizzare per contenere la pagina che ha causato il *fault*. La sostituzione delle pagine ha le seguenti fasi:

1. Si individua la locazione nel disco;
2. Si individua un frame libero: se esiste viene utilizzato; altrimenti viene utilizzato un algoritmo di sostituzione per selezionare un frame vittima;
3. Si scrive la pagina richiesta nel frame appena liberato;
4. Si riavvia il processo utente.

Dato che sono necessari 2 trasferimenti di pagine, il tempo di servizio del *page fault* aumenta e di conseguenza anche il tempo effettivo d'accesso.

Questo sovraccarico si può ridurre usando se non esiste nessun frame libero, questo sovraccarico si può ridurre usando un **bit di modifica** (*dirty bit*) che se impostato a 1, riscrive su disco le pagine che vengono modificate.

La **sostituzione di una pagina** è fondamentale per la *paginazione su richiesta* perché completa la separazione tra memoria logica e memoria fisica.

Per realizzare la paginazione su richiesta, è necessario risolvere 2 problemi principali; occorre sviluppare:

- ❖ un **algoritmo di allocazione dei frame**: determina quanti frame è necessario associare a ciascun processo.
- ❖ un **algoritmo per la sostituzione delle pagine**: cerca di ridurre la **frequenza di page fault** scegliendo quali frame sostituire quando c'è necessità.

Gli algoritmi si valutano eseguendoli su una *successione di riferimenti*, detta **reference string**, e contando il numero di page fault su tale successione. Questa successione è composta dai soli numeri di pagina, i risultati ottenuti dipendono dal numero di frame a disposizione.

8.4.1 Sostituzione delle pagine secondo l'ordine d'arrivo (FIFO)

L'algoritmo di sostituzione delle pagine più semplice è quello **FIFO**. Associa ad ogni pagina l'istante di tempo in cui quella pagina è arrivata in memoria e se si deve sostituire una pagina si sceglie quella che sta da più tempo in memoria.

Le prestazioni dell'algoritmo FIFO non sono sempre buone perché la pagina sostituita potrebbe essere un modulo usato molto tempo prima e che non serve più, ma potrebbe contenere una variabile ancora in uso.

Anche se si sostituisce una pagina in uso attivo, tutto continua a funzionare correttamente; infatti, dopo aver rimosso una pagina attiva, si verifica un *page fault trap* per riprendere la pagina attiva e riportala in memoria sostituendo un'altra pagina, aumentando così la *frequenza di pagine mancanti* che rallenta l'esecuzione del processo, ma senza causare errori.

In questo algoritmo si verifica un'anomalia, detta **anomalia di Belady**, ovvero aumentando il numero di frame aumenta il numero di *page fault*.

8.4.2 Sostituzione ottimale delle pagine

L'**algoritmo ottimale** è quello che fra tutti gli algoritmi presenta la minima frequenza di page fault e non presenta mai l'anomalia di Belady.

L'algoritmo è semplice: si sostituisce la pagina che non verrà usata per il periodo di tempo più a lungo.

Sfortunatamente questo algoritmo è difficile da realizzare, perché richiede la conoscenza futura della *successione dei riferimenti*. Questa situazione è analoga all'algoritmo SJF di scheduling della CPU.

8.4.3 Sostituzione delle pagine usate meno recentemente (LRU)

La distinzione fondamentale tra l'**algoritmo ottimale** e l'**algoritmo FIFO** consiste nel fatto che il *FIFO* impiega l'istante in cui una pagina è stata caricata in memoria, mentre l'**algoritmo ottimale** impiega l'istante in cui la pagina è usata.

L'**algoritmo LRU** (*last recently used*) utilizza la conoscenza passata e sostituisce la pagina che non è stata usata per il periodo più lungo: quindi per utilizzare questo algoritmo a ciascuna pagina è associato il tempo di ultimo accesso. L'algoritmo LRU è meglio di FIFO ma peggiore di OPT.

Il problema principale consiste nel determinare un ordine per i frame definito secondo il momento dell'ultimo uso. L'algoritmo LRU può essere implementato in due modi:

- ❖ **Implementazione con contatore.** Ciascuna pagina ha un contatore, ogni volta che si fa riferimento alla pagina si copia l'orologio nel contatore; quando si deve scegliere una pagina da rimuovere si analizza il contatore per scegliere quale pagina cambiare.
- ❖ **Implementazione con pila.** Si usa una lista doppiamente concatenata che rappresenta il numero delle pagine. Ogni volta che si fa un riferimento a una pagina, la si estraе dalla pila e la si colloca in cima a quest'ultima. In questo modo in cima alla pila si trova sempre la pagina usata per ultima.

Gli algoritmi OPT e LRU sono algoritmi a pila che non soffrono della anomalia di Belady.

In particolare, in un algoritmo a pila l'insieme delle pagine in memoria per n frame è un sottoinsieme delle pagine contenute in memoria per $n+1$ frame.

Sono pochi i sistemi che utilizzano LRU. Molti sistemi offrono aiuto fornendo un **bit di riferimento**, impostato automaticamente dall'architettura del sistema ogni volta che si fa riferimento alla pagina. Inizialmente, il SO azzera tutti i bit. Quando si inizia l'esecuzione di un processo utente, l'architettura del sistema imposta a 1 il bit associato alla pagina a cui si fa riferimento. Dopo qualche tempo, è possibile stabilire quali pagine sono state usate semplicemente esaminando i bit di riferimento.

8.4.4 Sostituzione delle pagine basata su conteggio

Esistono altri algoritmi basati sul conteggio dei riferimenti fatti a ciascuna pagina:

- **Algoritmo di sostituzione delle pagine meno frequentemente usate (LFU).** Richiede che si sostituisca la pagina con il conteggio più basso, questo perché una pagina usata attivamente ha un conteggio alto. Questo è sconveniente quando la pagina viene utilizzata molto nella fase iniziale di un processo e successivamente non usata, quindi ha un conteggio alto e non può essere sostituita.
- **Algoritmo di sostituzione delle pagine più frequentemente usate (MFU).** È basato sul fatto che la pagina con il contatore più basso è stata appena inserita in memoria e quindi non è stata ancora usata.

Questi due tipi di algoritmi sono molto onerosi.

8.4.5 Algoritmi con memorizzazione transitoria delle pagine

Oltre agli algoritmi si possono usare anche altre procedure: generalmente i sistemi usano un **gruppo di frame liberi**.

Quando si verifica un page fault, si sceglie il *frame vittima* da questo gruppo, ma prima che si scriva in memoria secondaria, si trasferisce la pagina richiesta in un frame libero del gruppo. Questo permette la rapida esecuzione al processo senza che si attenda la scrittura della pagina vittima nella memoria secondaria.

Prima di accedere alla memoria di massa si controlla se la pagina richiesta è ancora presente nel pool dei frame liberi; se così fosse il processo può essere riattivato rapidamente altrimenti, in caso contrario, si deve individuare un frame libero e trasferirvi la pagina.

Ogni volta che il dispositivo di paginazione è inattivo, si sceglie una pagina modificata, la si scrive nel disco e si reimposta il suo bit di modifica così da aumentare la probabilità, che al momento della selezione per la sostituzione, di una pagina vittima questa non abbia subito modifiche e non debba essere trascritta sul disco.

8.4.6 Applicazioni e sostituzione della pagina

In alcuni casi le applicazioni che accedono ai dati tramite la memoria virtuale del SO non hanno prestazione migliori di quelle che il sistema, senza impiegare alcun buffer, potrebbe offrire.

Per risolvere tali problemi, alcuni SO permettono certi programmi di utilizzare una partizione del disco come un array sequenziale di blocchi logici, senza ricorrere alle strutture dati del file system. Un simile array è anche detto **disco di basso livello** (*raw disk*) e il relativo I/O è denominato **I/O di basso livello** (*raw I/O*).

8.5 Allocazione dei frame

Bisogna stabilire un criterio per l'allocazione della memoria libera ai diversi processi. Il SO può assegnare tutto lo spazio richiesto dalle proprie strutture dati attingendo dalla lista dei frame liberi. Quando questo spazio è inutilizzato dal SO può essere sfruttato per la paginazione utente.

Un'altra variante prevede di riservare sempre *tre frame liberi*, in modo che quando si verifica un page fault è disponibile un frame libero in cui trasferire la pagina richiesta. Durante il trasferimento della pagina, si può fare una sostituzione, la pagina coinvolta viene poi scritta nel disco mentre il processo utente continua l'esecuzione. Al processo utente si assegna qualsiasi frame libero.

8.5.1 Numero minimo di frame

L'allocazione dei frame è soggetta a parecchi vincoli. Non si possono assegnare più frame di quanti ne siano disponibili, sempre che non vi sia condivisione di pagine. Inoltre, è necessario assegnare almeno un **numero minimo di frame** (il motivo è legato alle prestazioni del sistema) ma con il diminuire del numero di frame allocati a ciascun processo aumenta la *frequenza dei page fault* portando il rallentamento dell'esecuzione del processo.

Il numero minimo di frame è definito dall'hardware del calcolatore, mentre il numero massimo è definito dalla quantità di memoria fisica disponibile.

Nei calcolatori che permettono *riferimenti indiretti a più livelli*, tutta la memoria virtuale si deve trovare in memoria fisica. Però occorre porre un limite al livello dei riferimenti indiretti impostando un contatore di livelli. Se il contatore si riduce a zero si verifica una trap.

8.5.2 Algoritmi di allocazione

Esistono principalmente due schemi di allocazione:

- ❖ **Allocazione uniforme:** se si avessero a disposizione m frame per n processi, ogni processo avrebbe m/n frame.
- ❖ **Allocazione proporzionale:** la memoria disponibile si alloca a ciascun processo secondo la propria dimensione.

Entrambe le allocazioni variano rispetto al livello di multiprogrammazione: se tale livello aumenta, ciascun processo perde alcuni frame per fornire la memoria necessaria per il nuovo processo; se la multiprogrammazione diminuisce, i frame allocati al processo allontanato si possono distribuire tra quelli che restano.

Esiste un altro tipo di allocazione: l'**allocazione con priorità**, è una sorta di *allocazione proporzionale* però è basata su priorità piuttosto che sulla dimensione, nel caso di page fault si seleziona un frame di un processo con priorità minore.

8.5.3 Allocazione globale e allocazione locale

Nel caso in cui ci siano più processi in competizione per i frame, vengono utilizzati algoritmi di sostituzione delle pagine:

- ❖ **Sostituzione globale:** il SO seleziona il frame da sostituire dall'insieme di tutti i frame, anche se quel frame è allocato ad un altro processo; un processo può, quindi, sottrarre un frame ad un altro processo;
- ❖ **Sostituzione locale:** il SO, per ciascun processo, seleziona i frame solo dal proprio insieme di frame.

L'algoritmo di sostituzione globale possiede un problema: un processo non può controllare la propria *frequenza di page fault*, infatti l'insieme di pagine che si trova in memoria per un processo non dipende solo dal comportamento di paginazione di quel processo, ma anche dal comportamento di paginazione di altri processi.

Con l'algoritmo di sostituzione locale questo problema non si presenta però la sostituzione locale può limitare un processo, non rendendogli disponibili altre pagine di memoria.

La sostituzione globale genera un maggior throughput, perciò è il metodo più usato

8.5.4 Accesso non uniforme alla memoria (NUMA)

Un sistema multiprocessore è costituito da diverse schede madri, ognuna contenente più processori e una parte di memoria. Le schede sono connesse attraverso bus di sistema oppure con connessioni di rete ad alta velocità. I processori accedono alla memoria della propria scheda madre in meno tempo rispetto a quello necessario per accedere alle altre schede del sistema.

I sistemi nei quali i tempi di accesso alla memoria variano molto sono detti **sistemi con accesso non uniforme alla memoria (NUMA)** e sono più lenti dei sistemi in cui l'accesso avviene in modo uniforme (**UMA**), cioè dove processori e memoria risiedono sulla stessa scheda madre.

Nei sistemi NUMA l'obiettivo è quello di allocare i frame di memoria "il più vicino possibile" al processore sul quale il processo è in esecuzione.

La situazione diventa complicata con l'aggiunta dei *thread*: un processo con molti *thread* in esecuzione potrebbe vedere quei *thread* schedulati su diverse schede del sistema.

- ❖ **Linux** gestisce questa situazione facendo in modo che il kernel identifichi una gerarchia di domini di scheduling. Lo scheduler CFS di Linux non consente ai thread di migrare su domini diversi e quindi incorrere in penalità nell'accesso alla memoria. Linux, inoltre, ha una lista dei frame liberi diversa per ogni nodo NUMA, garantendo in tal modo che a un thread sia allocata memoria del nodo su cui è in esecuzione.
- ❖ **Solaris** risolve il problema creando un'entità **Igroup** nel kernel, ordinati gerarchicamente sulla base del periodo di latenza tra gruppi. Ogni *Igroup* raccoglie i processori e la memoria vicini fra loro. Il SO pianifica tutti i *thread* di un processo e alloca tutta la memoria di un processo in un solo *Igroup*: l'obiettivo è di minimizzare la latenza complessiva della memoria e di massimizzare il grado di *cache hit* (successo di cache) del processore.

8.6 Thrashing

Il fenomeno del **thrashing** si verifica quando un processo è costantemente soggetto a *page fault* e quindi è occupato a spostare pagine dal disco alla memoria, spendendo più tempo per la paginazione che per l'esecuzione dei processi causando gravi problemi di prestazioni.

In particolare: se un processo non dispone di un numero minimo di frame, occorre sospendere la sua esecuzione, eliminare le pagine restanti e liberare tutti i frame allocati. A questo punto si deve sostituire qualche pagina; ma, poiché le sue pagine sono attive, si deve sostituire una pagina che sarà subito necessaria, e di conseguenza si verificano parecchi page fault.

8.6.1 Cause del thrashing

Analizziamo le **cause del thrashing**:

- Il SO controlla l'utilizzo della CPU: se questo è basso, aumenta il grado di multiprogrammazione introducendo un nuovo processo.
- Si usa un **algoritmo di sostituzione delle pagine globale**, che sostituisce le pagine senza tener conto del processo al quale appartengono.
- La memoria fisica libera è insufficiente rispetto ai frame necessari per l'esecuzione dei processi e si generano quindi numerosi page fault;
- Per effettuare il caricamento e lo scaricamento delle pagine per questi processi si usa un dispositivo di paginazione ma mentre si mettono i processi in coda per il dispositivo di paginazione, la ready queue si svuota e l'utilizzo della CPU diminuisce;
- Il SO deduce erroneamente che è necessario aumentare il grado di multiprogrammazione dato che la CPU rimane inattiva a causa dell'intensa attività di I/O;
- Vengono avviati nuovi processi che per una mancanza di frame liberi attiveranno altre intensive attività di paging.
- Una volta che l'utilizzo della CPU arriva al massimo, la paginazione degenera e fa crollare l'utilizzo della CPU. In questa situazione, per aumentare l'utilizzo della CPU, bisogna ridurre il grado di multiprogrammazione.

Per limitare gli effetti del *thrashing* si può usare un **algoritmo di sostituzione locale** o **algoritmo di sostituzione per priorità** che vieta al processo di sottrarre i frame ad un altro processo provocandone la degenerazione.

Invece, per evitare il *thrashing* occorre fornire ad un processo tutti i frame di cui necessita attraverso la creazione di un **working-set** che osserva quanti sono i frame che un processo sta effettivamente usando. Questo metodo permette alla *paginazione su richiesta* di utilizzare un **modello di località**.

Una **località** è un insieme di pagine che sono accedute insieme, e i processi, durante la loro esecuzione, possono passare da una località all'altra. Il *thrashing* avviene quando la somma delle dimensioni delle località è maggiore della memoria totale.

Le località hanno una caratteristica *spazio-temporale*. Per esempio, quando viene invocata una funzione viene definita una nuova località; quando la funzione termina il processo lascia la località corrispondente. Quindi le località sono definite dalla struttura del programma e dalle relative strutture dati.

8.6.2 Working set

Il modello del **working set** è basato sull'ipotesi di località. Il **working-set** è praticamente una finestra dove sono contenute le pagine che sono state usate più recentemente. All'interno di questa finestra saranno quindi presenti tutte le pagine che sono attive, mentre quelle non più attive non vi rientrano. Essendo un'approssimazione della località del programma, si possono individuare quelle pagine che servono al processo, non sostituendo queste ultime, ma bensì quelle inattive.

8.6.3 Frequenza di page fault

I metodo del **working-set** è considerato un metodo rozzo per controllare il **thrashing**. Un approccio più semplice è di stabilire una **frequenza di page fault** accettabile utilizzando la **sostituzione locale**. Con tale tecnica si può controllare se un processo ha un alto tasso di *page fault*, allora vuol dire che ha bisogno di più frame, nel caso in cui fosse basso allora vuol dire che a quel processo sono stati allocati più frame di quelli a lui necessari:

- Se la frequenza è troppo bassa il processo rilascia dei frame;
- Se la frequenza è molto alta il processo ottiene dei frame.

8.7 Prepaginazione

Per ridurre le innumerevoli *page fault* al momento dell'avvio di un processo, si potrebbe utilizzare la **prepaginazione**. Tale tecnica prevede di diminuire il tasso di *page fault* caricando in memoria tutte le pagine che serviranno al processo.

Il problema principale della **prepaginazione** è che potrebbe essere molto onerosa. Questo potrebbe accadere in quanto alcune pagine portate in memoria nella **prepaginazione** potrebbero non essere utilizzate dal processo, questo provocherebbe uno spreco di tempo che poteva essere impiegato per la computazione, quindi la richiesta di *page fault* poteva risultare più efficiente.

8.8 Linux

Linux gestisce la memoria del kernel usando l'**allocazione a lastre**. Vediamo ora come Linux gestisce la memoria virtuale.

Linux usa la **paginazione richiesta**, allocando pagine da una lista dei frame liberi. Per gestire la memoria Linux mantiene due tipi di liste di pagine:

- **active_list** contiene le pagine considerate in uso;
- **inactive_list** contiene le pagine a cui non è stato fatto riferimento di recente e la cui memoria può essere recuperata.

Ogni pagina ha un **bit accessed** che viene settato a ogni riferimento alla pagina. Quando una pagina viene allocata per la prima volta, il suo *bit accessed* viene impostato e la pagina viene aggiunta in fondo alla **active_list**. Ogni volta che si fa riferimento a una pagina della **active_list**, il *bit accessed* viene impostato e la pagina viene spostata in fondo alla lista.

Le due liste sono tenute bilanciate tra loro dal processo **kaswapd** del kernel Linux che controlla periodicamente la quantità di memoria disponibile nel sistema.

8.9 Windows

Windows 10 supporta sistemi a 32 e 64 bit. Nei sistemi a 32 bit, lo spazio di indirizzi virtuali per un processo è 2GB. I sistemi a 32 bit supportano 4GB di memoria fisica. Sui sistemi a 64 bit, Windows 10 ha uno spazio di indirizzo virtuali da 128TB e supporta fino a 24TB di memoria fisica.

Windows 10 realizza la memoria virtuale impiegando la *paginazione su richiesta* per gruppi, detti **cluster di pagine** (*demand paging with clustering*), una tecnica che riconosce la località dei riferimenti alla memoria e gestisce i *page fault* caricando in memoria non solo la pagina richiesta, ma più pagine a essa precedenti e successive.

Un componente importante del gestore di memoria virtuale in Windows 10 è la **gestione del working set**. Quando viene creato un processo, gli viene assegnato un *working set* minimo di 50 pagine e un *working set* massimo di 345 pagine.

Il gestore della memoria virtuale mantiene una lista dei frame liberi a cui è associato un valore che indica se è disponibile sufficiente memoria libera.

- Se un processo che ha raggiunto il suo *working set* massimo incorre in un *page fault* ed è disponibile sufficiente memoria, al processo viene assegnata una pagina libera.
- Se invece la quantità di memoria è insufficiente, il kernel deve selezionare una pagina dal *working set* del processo ed effettuare una sostituzione usando una politica LRU di sostituzione locale delle pagine.

Quando la quantità di memoria disponibile scende al di sotto della soglia, il gestore della memoria virtuale utilizza una *sostituzione globale* detta **trimming di working set automatico** per ripristinare il valore ad un livello superiore alla soglia.

8.10 Criteri di dimensionamento delle pagine

I criteri per determinare la dimensione delle pagine sono:

- **Dimensione della tabella delle pagine;**
- **Sovraccarico di I/O;**
- **Numero di page fault;**
- **Dimensione ed efficienza della TLB;**
- **Frammentazione;**
- **Risoluzione;**
- **Località:** si riduce l'I/O totale, ovvero pagine di piccole dimensioni.

Memoria secondaria e terziaria (Capitolo 9)

9.1 Introduzione

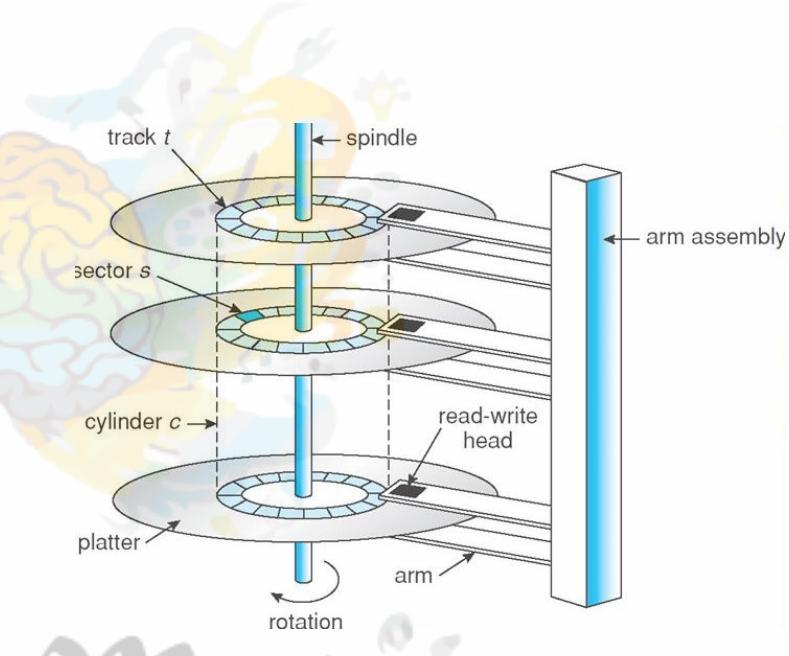
Una **memoria di massa** è un tipo di memoria che non possiede una grande velocità di accesso però ha un costo unitario decisamente più basso rispetto alla *memoria primaria*. Viene detta di massa poiché, rispetto alla *memoria primaria*, è in grado di raccogliere enormi quantità di dati.

La memoria di massa è una memoria **non volatile** cioè i dati non vengono persi allo spegnimento della macchina a differenza della memoria centrale che è **volatile**.

La **memoria secondaria** di solito comprende **dischi rigidi** (HDD) e dispositivi di memoria non volatile detti **NVM**. Alcuni sistemi dispongono, inoltre, di una **memoria terziaria** più lenta e più grande, generalmente costituita da nastri magnetici, dischi ottici o anche spazio di memorizzazione su cloud.

9.2 Dischi rigidi

I **dischi magnetici** (HDD) sono il supporto fondamentale dei sistemi elaborativi moderni. I **piatti** dei dischi hanno una forma piana e rotonda e le due superfici sono ricoperte di materiale magnetico, infatti, le informazioni vengono registrate magneticamente. Le **testine di lettura e scrittura** sono sospese su ciascuna superficie d'ogni piatto e sono attaccate al **braccio del disco** che le muove in blocco. La superficie di un piatto è divisa logicamente in **tracce circolari** che a loro volta sono suddivise in **settori**. L'insieme delle tracce corrispondenti a un braccio formano un **cilindro**.



Quando un disco è in funzione un motore lo fa ruotare ad alta velocità. La velocità di un disco è caratterizzata da due valori:

- **tempo di trasferimento**, cioè la velocità con cui i dati viaggiano dall'HDD al calcolatore;
- il **tempo di posizionamento** composto da:
 - il **tempo di ricerca** (*seek time*), il tempo necessario a spostare il braccio del disco in corrispondenza del cilindro desiderato;
 - **latenza di rotazione**, il tempo necessario affinché il settore desiderato si porti sotto la testina.

Poiché le testine di un disco sono sospese su un cuscino d'aria sottilissimo, esiste il pericolo che la testina urti la superficie magnetica, in tal caso si parla di **crollo della testina** e causa la rottura del disco che non può essere riparato.

Un disco può essere **rimovibile**, ciò permette che diversi dischi siano montanti secondo le necessità, come i **floppy disk** formati da un involucro di plastica che contiene un piatto flessibile.

I dischi possono essere collegati attraverso dei fili detti **bus di I/O** e possono essere *EIDE*, *SATA*, *serial ATA*, *SCSI*, *USB*. Il trasferimento dei dati è eseguito da *unità di elaborazione* dette **controllori**:

- **adattatori** (*adapter*) o **controllori di macchina** (*host controller*) sono i controllori posti all'estremità relativa del bus.
- **controllori dei dischi** (*disk controller*) sono incorporati in ciascuna unità disco e solitamente hanno anche una cache incorporata.

Per eseguire un'operazione di I/O, il calcolatore inserisce un comando nell'*adapter*, generalmente mediante porte di I/O mappate in memoria; l'*adapter* invia il comando al controllore del disco, che agisce sugli elementi elettromeccanici dell'unità per portare a termine il compito richiesto.

9.3 Dispositivi NVM (SSD)

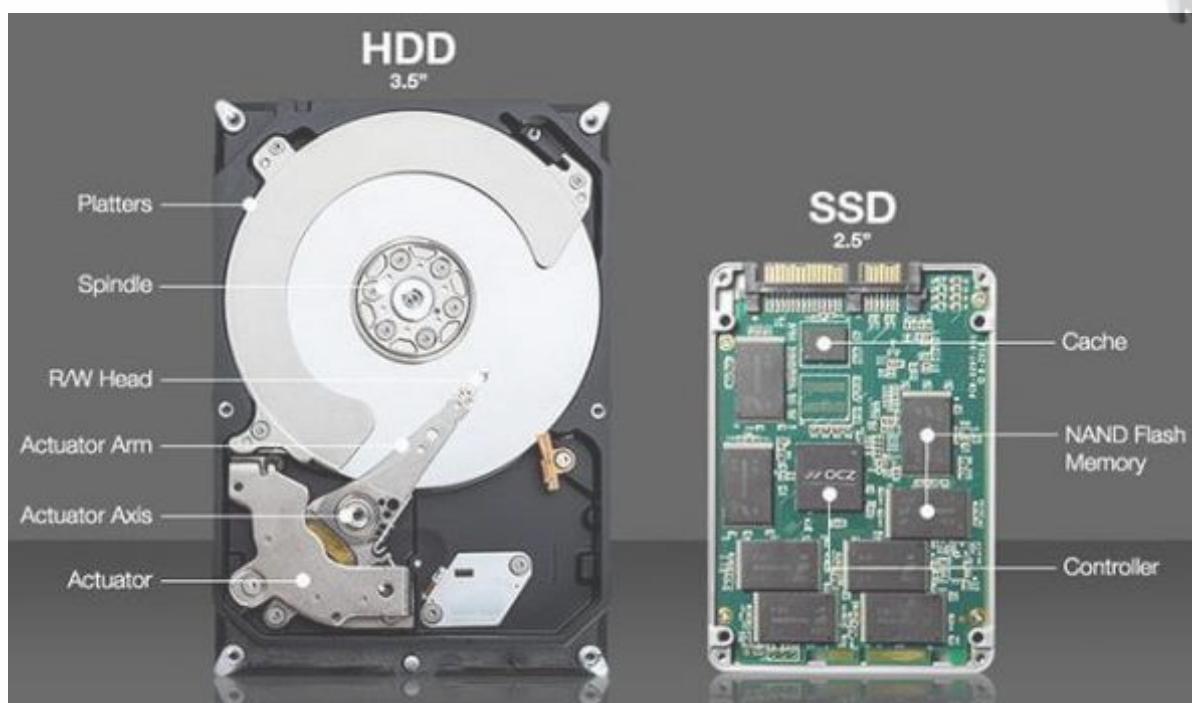
I **dispositivi NVM** sono dispositivi elettrici anziché meccanici e, nella maggior parte dei casi, sono composti da un controllore e da diversi chip di memoria a **semiconduttore NAND flash** utilizzati per memorizzare i dati. Esistono altre tecnologie NVM, come le **DRAM** dotate di una batteria di backup che permette di non perdere il contenuto.

I *dispositivi NVM* basati su **memoria flash** vengono spesso inseriti in contenitori simili a unità disco, e per questo motivo sono chiamati **dischi a stato solido (SSD)**. Un dispositivo NVM può assumere la forma di un'**unità USB** o di un modulo **DRAM**. Negli smartphone le NVM sono montate sulla superficie delle schede madri come dispositivo principale di archiviazione.

Gli **SSD** sono più affidabili dei *dischi rigidi*, perché non hanno parti mobili, e sono più veloci perché non hanno un tempo di posizionamento o latenza di rotazione, inoltre, consumano meno energia. Lo svantaggio è che sono molto costosi rispetto e hanno, in genere, una capacità inferiore rispetto agli HDD presenti sul mercato.

Poiché gli **SSD** sono più veloci dei dischi rigidi, alcuni di loro sono progettati per connettersi direttamente al bus di sistema. Alcuni sistemi usano gli **SSD** come un nuovo livello di cache per ottimizzare le prestazioni.

I **semiconduttori NAND** possono essere letti e scritti a livello di pagina ma i dati non possono essere sovrascritti senza che le celle NAND siano prima cancellate. I dispositivi flash NVM sono composti da molte piastrine, dette **die**, ciascuna con molti **datapath**, e le operazioni possono avvenire in parallelo (ciascuna su un datapath). I semiconduttori NAND, inoltre, si deteriorano a ogni ciclo di cancellazione e dopo circa 100.000 cicli le celle non sono più in grado di conservare i dati. A causa dell'usura, la durata delle NVM NAND viene misurata in numero di scritture per giorno sull'unità (DWPD): tale numero indica quante volte al giorno l'intera capacità dell'unità può essere scritta prima che si guasti.



9.3.1 Memorie flash

La **memoria flash** è un tipo di memoria a stato solido e non volatile, che può anche essere usata come *memoria a lettura-scrittura*; quando viene utilizzata come ROM viene anche chiamata *flash ROM*. Le memorie flash memorizzano le informazioni in un array e tramite una tipologia di transistor conservano i valori dei bit. Le memorie flash non hanno parti mobili quindi sono molto resistenti, inoltre risultano leggere e di dimensioni ridotte quindi di facile trasporto anche perché non hanno bisogno di un'alimentazione elettrica.

9.3.2 Algoritmi del controllore delle NAND Flash

Poiché i semiconduttori NAND non possono essere sovrascritti, sono di solito presenti pagine che contengono dati non validi. Il controllore mantiene una **tavella di traduzione flash (FTL)** che tiene traccia delle pagine fisiche contenenti blocchi logici validi e i blocchi che contengono solo pagine non valide e quindi cancellabili.

Se non ci sono blocchi disponibili per effettuare una scrittura può essere effettuata la **garbage collection** in cui i dati vengono copiati in altre posizioni, liberando i blocchi cancellabili e riutilizzabili per nuove scritture. Per migliorare le prestazioni in scrittura, l'SSD utilizza una tecnica detta **over-provisioning** in cui un certo numero di pagine viene messo da parte per fornire un'area sempre disponibile per la scrittura. I blocchi che diventano non validi vengono cancellati e inseriti nello spazio di *over-provisioning* se il dispositivo è pieno, o altrimenti nella lista dei blocchi liberi.

Il controllore cerca di evitare che i blocchi frequentemente cancellati si logorano più velocemente degli altri mediante vari algoritmi che memorizzano i dati sui blocchi meno cancellati, livellando l'usura.

Pagina valida	Pagina valida	Pagina non valida	Pagina non valida
Pagina non valida	Pagina non valida	Pagina non valida	Pagina valida

9.4 Nastri magnetici

I **nastri magnetici** sono stati i primi supporti di memorizzazione secondaria. Queste unità sono caratterizzate da un tempo d'accesso molto elevato rispetto a quello della memoria centrale e dei dischi magnetici. Gli usi principali dei nastri sono la creazione di copie di backup dei dati, la registrazione di dati poco usati e il trasferimento di informazioni tra diversi sistemi di elaborazione.

Il nastro è avvolto in bobine e scorre su una testina di lettura e scrittura. Il posizionamento sul settore richiesto può richiedere alcuni minuti, una volta raggiunta la posizione, l'unità può leggere o scrivere a una velocità simile a quella di un'unità disco.

9.5 Formattazione del disco, partizioni e volumi

Un **file system** è un meccanismo con il quale i file sono immagazzinati e organizzati su un dispositivo di archiviazione. In particolare, un *file system* è l'insieme delle strutture dati e dei metodi che ci permettono la registrazione e l'accesso a dati e programmi presenti in un sistema di calcolo.

Un dispositivo di archiviazione nuovo è *tabula rasa* cioè un insieme di uno o più piatti privi di file system.

Per prima cosa il dispositivo deve essere diviso in settori che possono essere scritti o letti dal controllore. Questa fase è detta **formattazione fisica** o **formattazione di basso livello**. Tale formattazione riempie il dispositivo con una struttura dati per ogni settore.

La struttura dati consiste di una intestazione e una coda. L'intestazione e la coda contengono informazioni usate dal controllore del disco, ad esempio il numero del settore e un codice per la **correzione degli errori (EEC)**. Quando si effettua un'operazione si aggiorna il valore *EEC* secondo il contenuto dell'area e i dati del settore. Se risulta una differenza tra *EEC* e i dati letti nel settore significa che il settore potrebbe essere difettoso. La *formattazione fisica* è effettuata di solito dai costruttori in modo che testino anche il disco e controllino il numero dei settori danneggiati.

Per usare il disco il SO deve registrare le proprie strutture dati nel disco attraverso 3 passaggi:

1. Si suddivide il dispositivo in uno o più gruppi di blocchi o pagine, dette **partizioni**. Ciascuna partizione è un'unità a sé stante. Le informazioni sulle partizioni sono scritte in un formato stabilito e in una posizione fissa sul dispositivo di archiviazione. Una volta che il SO riconosce il dispositivo, vengono lette le informazioni sulle partizioni e il SO crea delle descrizioni per tali partizioni (*in Linux si trovano in /dev*).
In Linux le partizioni vengono gestite con il comando `fdisk`.
2. Il secondo passo è la creazione e gestione del **volume**. Con il termine “volume” indichiamo un qualsiasi file system montabile, incluso un singolo file contenente un file system, come l’immagine di un CD.
3. Il terzo passo è la **formattazione logica**, cioè la creazione di un file system: il SO registra nel dispositivo le strutture dati iniziali relative al file system. La partizione etichettata per l'avvio viene utilizzata per stabilire il *root* del file system.

Il **file system** di un computer è costituito da tutti i volumi montati. In Windows, questi vengono denominati tramite lettere maiuscole dell’alfabeto (C:, D:, E:), mentre in Linux, al momento dell'avvio viene montato il file system principale e gli altri file system possono essere montati all'interno della stessa struttura ad albero.

Per una maggiore efficienza, la maggior parte dei file system accorda i blocchi in gruppi, detti **cluster**. L'I/O del dispositivo procede per blocchi, mentre l'I/O del file system procede per cluster.

Alcuni SO danno l'opportunità ad alcuni programmi speciali di impiegare una partizione del disco come un grande array sequenziale di blocchi logici. Questo array è detto **disco di basso livello (raw disk)**; l'I/O relativo si chiama **I/O di basso livello (raw I/O)**. Un sistema di questo tipo può essere utilizzato come **area di swap**. Linux non supporta il *raw I/O*, però è possibile ottenere un accesso di questo tipo utilizzando il flag `DIRECT` nella syscall `open()`.

9.5.1 Blocco d'avviamento (**bootstrap**)

Un calcolatore entra in funzione eseguendo un programma iniziale chiamato **bootstrap** che avvia il SO individuandone il kernel nei dischi, lo carica nella memoria, e salta a un indirizzo iniziale per avviare l'esecuzione del SO. Il SO avvia l'esecuzione del primo processo di elaborazione (**init**) e attende che si verifichi qualche evento.

Per la maggior parte dei calcolatori, il *bootstrap* è di solito memorizzato nel **firmware** (una memoria flash NVM). Il **firmware** è un programma integrato direttamente in un componente elettronico come le memorie a sola lettura, le **ROM**, e le memorie programmabili cancellabili elettricamente, le **EEPROM**. Questo tipo di memoria presenta il vantaggio di non dover essere inizializzata, e ha un indirizzo iniziale fisso dal quale la CPU può cominciare l'esecuzione ogni volta che si accende o si riavvia la macchina; inoltre visto che la **ROM** è a sola lettura è immune ai virus.

Un problema, generato dal *firmware*, risiede nel fatto che l'esecuzione del codice è più lenta di quanto avvenga con la RAM, per questo molti SO vengono memorizzati nel *firmware* e copiati nella RAM per essere eseguiti rapidamente.

Un altro problema è legato al fatto che se si vuole cambiare *bootstrap* bisogna sostituire i circuiti integrati **ROM**. Una soluzione è quella di memorizzare nella **ROM** un piccolo **caricatore d'avviamento** (**bootstrap loader**) in grado di caricare dalla memoria secondaria un **bootstrap completo**, registrato in una partizione del disco chiamata *blocchi d'avviamento*, posta in una locazione fissa del disco. Un dispositivo che contiene una partizione di avvio è chiamato **disco di sistema** o **boot disk**. Il *bootstrap loader* predefinito di Linux è **grub2**.

Windows consente di suddividere il disco rigido in una o più partizioni; in una di esse, detta **partizione di avviamento**, sono contenuti il SO e i driver dei dispositivi. Windows colloca il proprio *bootstrap* nel primo **blocco logico del disco fisso**, chiamato **MBR** (*master boot record*). La procedura di avviamento (**booting**) inizia con l'esecuzione del codice residente nel *firmware* del sistema. Questo codice fa sì che il sistema legga il *bootstrap* dall'MBR.

L'MBR contiene una tabella che elenca le partizioni del dispositivo e un flag che indica da quale partizione si debba avviare il sistema. Dopo avere identificato il *boot disk*, il sistema legge da tale partizione il primo settore, chiamato **settore di avviamento**, svolge le restanti procedure di avviamento, tra cui il caricamento dei vari sottosistemi e dei servizi del sistema.

9.5.2 Blocchi difettosi (**bad blocks**)

Le unità disco possono rompersi facilmente e prima di sostituirle bisogna recuperarne il contenuto da una copia di backup e trasferirla nella nuova unità disco.

La maggior parte dei dischi messi in commercio contiene già **blocchi difettosi** (**bad blocks**), trattati in diversi modi a seconda del controllore e del tipo del disco.

Nel caso di dischi semplici, i blocchi difettosi sono gestiti "manualmente". Una soluzione consiste nell'effettuare una scansione del disco durante la formattazione, per rivelare la presenza dei *bad blocks*: se lo si trova lo si marca come inutilizzabile. Se un blocco diventa difettoso durante l'uso del sistema, viene lanciato un programma (*in Linux il comando badblocks*) che individua i *bad blocks* e li isola. I dati sui *bad blocks* di solito vanno persi.

Unita a disco più complesse, come dischi SCSI, utilizzano un controllore che tiene una lista dei *bad blocks* inizializzata durante la formattazione fisica da parte del produttore e aggiornata man mano che si verificano nuovi malfunzionamenti. La formattazione mette anche a disposizione dei settori di riserva

non visibili al SO nel caso in cui uno diventi difettoso viene rimpiazzato da uno di questi settori. Questa strategia è chiamata **accantonamento dei settori** (*sector formarding* o *sector sparing*). La maggior parte di questi dischi mette a disposizione interi cilindri per questa operazione.

Un’alternativa all’accantonamento dei settori è data dai controllori capaci di sostituire i settori difettosi tramite la **traslazione dei settori** (*sector slipping*) che sposta in avanti di una posizione tutti i settori.

- Gli **errori reversibili** possono attivare un processo che copia i dati del *bad blocks* mentre questo viene messo in riserva o traslato.
- Gli **errori irreversibili** causano la perdita dei dati.

La gestione dei *bad blocks* negli SSD è più facile rispetto agli HDD poiché il controllore mantiene una tabella delle pagine non valide in modo da non renderle disponibili per la scrittura.

9.6 Gestione dell’area di swapping

Lo *swapping* sposta interi processi tra disco e memoria centrale quando la dimensione della memoria fisica si abbassa fino a raggiungere una soglia critica.

I sistemi moderni combinano lo *swapping* con la memoria virtuale spostando solo alcune pagine e non interi processi.

La **gestione dell’area di swap** usa lo spazio dei dischi come estensione della memoria centrale ma tale gestione riduce molto le prestazioni del sistema perché l’accesso al disco è molto più lento dell’accesso alla memoria centrale.

9.6.1 Uso dell’area di swapping

L’area di **swapping** è usata in modi diversi.

- I sistemi che adottano lo *swapping* dei processi nella memoria possono usare l’*area di swapping* per mantenere l’intera immagine del processo inclusi i segmenti dei dati e del codice;
- I sistemi a paginazione invece possono semplicemente memorizzare le pagine non contenute nella memoria centrale.

Un sistema che esaurisca l’area di swapping potrebbe essere costretto a terminare forzatamente i processi o ad arrestarsi completamente: alcuni sistemi consigliano di usare una quantità di **1,5 × RAM** per l’area di swap.

Linux permette l’uso di aree di swapping multiple che includono sia file che partizioni dedicate. Ogni area è formata da una serie di moduli di 4KB detti **slot di pagine** che conservano le pagine swappate. Inoltre, ogni area di swap ha una **swap map** associata, cioè un’array di contatori interi: se un contatore vale 0, la pagina che gli corrisponde è valida altrimenti lo slot è vuoto.

9.6.2 Collocazione dell’area di swapping

Le possibili collocazioni per un’area di swap sono due: all’interno del file system o in una partizione del disco a sé stante. Se l’area di swap è un file all’interno del file system, l’attraversamento delle strutture dati per l’allocazione dello spazio nei dischi richiede tempo. Le prestazioni si possono migliorare impiegando la memoria fisica come *cache* e anche usando strutture speciali per l’allocazione in blocchi contigui del file di swap. In alternativa all’area di swap si può creare una **partizione del disco non formattata** e si velocizza perché non si deve attraversare il file system ma si crea frammentazione interna ma dato che questa area ha vita breve, è un prezzo che si può pagare.

9.7 Connessione dei dispositivi di memorizzazione

I calcolatori accedono alla memoria secondaria in tre modi:

1. tramite un dispositivo collegato alla macchina attraverso porte locali di I/O. Le porte più utilizzate sono **SATA, USB, FireWire e Thunderbolt**.
2. tramite un dispositivo connesso alla rete come i server **NAS** che permettono di condividere la memoria tra più utenti connessi alla stessa rete locale. Ogni NAS è dotato di uno o più HDD che possono essere utilizzati come supporto per la memorizzazione dei dati. Il NAS è infatti collegato alla rete locale attraverso un normale cavo ethernet ed è basato su kernel Linux.
3. tramite un dispositivo cloud in cui il calcolatore si connette alla rete tramite Internet o WAN. Il **cloud storage** è basato su API perché a differenza dei NAS, la connessione WAN ha una latenza superiore rispetto alle reti LAN.

SAN è una rete o parte di una rete ad alta velocità di trasmissione costituita da dispositivi di memorizzazione di massa. Il suo scopo è quello di rendere tali risorse di immagazzinamento disponibili per qualsiasi computer connesso ad essa. I protocolli più diffusi, usati per la comunicazione all'interno di una SAN, sono **FCP** ed **iSCSI**.

Uno **storage array** è un dispositivo che può includere porte SAN, porte di rete o entrambe. Contiene inoltre unità per memorizzare dati e un controllore per gestire l'archiviazione e consentire l'accesso storage attraverso le reti. Alcuni *storage array* includono solo SSD oppure usano gli SSD come cache e gli HDD come memoria di massa.

9.8 Strutture RAID

L'evoluzione tecnologica ha reso le unità a disco più piccole e meno costose. Oggi è possibile equipaggiare sistemi con più dischi senza spendere grandi somme di denaro. Le varie tecniche per l'organizzazione dei dischi sono note col nome di **RAID** ed hanno lo scopo di affrontare i problemi di prestazioni e affidabilità.

9.8.1 Miglioramento dell'affidabilità tramite la ridondanza

La possibilità che un disco si rompa o si guasti è molto alta e può dipendere da molti fattori. La soluzione al problema dell'affidabilità sta nell'introdurre una **ridondanza**, cioè memorizzare informazioni che non sono necessarie ma che si possono usare nel caso di un guasto a un disco per ricostruire le informazioni perse. Il metodo più semplice è quello del **mirroring** in cui ogni disco logico è duplicato in due dischi fisici e ogni scrittura si effettua su entrambi i dischi così se uno si guasta i dati possono essere letti dall'altro.. Nel caso in cui si hanno interruzioni di corrente e una scrittura non andasse a buon fine, si deve cercare di non far corrompere il file system.

Una soluzione prevede la scrittura in uno solo dei dischi e solo successivamente nell'altro in modo da avere nella copia sempre scritture portate a termine.

Un'altra soluzione è aggiungere una **memoria non volatile stato solido (NVRAM)** alla batteria RAID, protetta dalla perdita di dati causati dalle cadute di alimentazione.

9.8.2 Miglioramento delle prestazioni tramite il parallelismo

L'**accesso in parallelo** di più dischi può portare grossi vantaggi tra cui migliorare la capacità di trasferimento distribuendo i dati in sezioni, su più dischi. La forma più semplice di questa distribuzione è detta **sezionamento o striping a livello dei bit** che consiste nel distribuire i bit di ciascun byte su più

dischi facendo in modo che la quantità di dati letti o scritti sia 8 volte superiore. Questo porta ad un grave svantaggio, nel caso in cui uno dei dischi si rompa, tutti i dati sono irrecuperabile.

Gli obiettivi principali del parallelismo tramite *striping* sono:

1. l'aumento del throughput per accessi multipli a piccole porzioni di dati;
2. la riduzione del tempo di risposta relativo agli accessi a grandi quantità di dati.

9.8.3 Livelli RAID

Gli schemi che realizzano diversi compromessi tra costi e prestazioni sono stati classificati in livelli chiamati **livelli di RAID**:

- ❖ **Raid 0**: si riferisce ad array di dischi con *striping* a livello di blocchi, ma senza ridondanza;
- ❖ **Raid 1**: si riferisce alla tecnica di *mirroring*;
- ❖ **Raid 4**: noto come **organizzazione con blocchi per la correzione degli errori di memoria (ECC)**. Tale livello corregge gli errori, anche se c'è un solo blocco *ECC*; per ogni bit nel settore, è possibile determinare se debba avere valore 1 o 0 calcolando la parità dei bit corrispondenti. Se un dei dischi si guasta, il blocco di parità serve per recuperare il contenuto.

Un sistema RAID 4 può servire diverse letture contemporaneamente. In lettura la capacità di trasferimento è paragonabile al RAID 0, ma la scrittura è penalizzata, perché la scrittura di ogni blocco comporta anche la lettura del valore di parità corrispondente e il suo aggiornamento (si parla in questo caso di **ciclo lettura-modifica-scrittura**): una richiesta di I/O comporta 4 accessi a disco, 2 in lettura e 2 in scrittura.

Vantaggi rispetto al RAID 1

1. resistenza ai guasti;
2. usa un solo disco per la parità dei dati memorizzati in diversi dischi di dati, anziché un solo disco di mirroring per ciascun disco di dati;
3. la velocità di trasferimento di un singolo blocco è più veloce del RAID 1 perché le scritture e le letture dei byte sono distribuite su più dischi.

Svantaggi

1. il disco usato per la parità può costituire il collo di bottiglia del sistema;
2. scrittura lenta a causa della modifica e del calcolo della parità

Tuttavia, molti RAID hanno un controllore che gestisce il calcolo della parità spostando il carico dalla CPU all'array di dischi, inoltre l'array ha anche una cache NVRAM per memorizzare i blocchi mentre viene calcolata la parità e per memorizzare le scritture evitando il **ciclo lettura-modifica-scrittura**.

- ❖ **Raid 5**: noto come **organizzazione con blocchi intercalati a parità distribuita**, differisce dal 4 per il fatto che invece di memorizzare i dati in n dischi e la parità in un disco separato, i dati e le informazioni di parità sono distribuite tra gli $n+1$ dischi. Ogni blocco memorizza la parità e gli altri i dati. È il più comune sistema raid.
- ❖ **Raid 6**: noto anche come **schema di ridondanza P+Q**, memorizza ulteriori informazioni ridondanti per potere gestire guasti contemporanei di più dischi. Invece della parità si usano codici ECC basati sulla matematica dei **campi di Galois**.

Esiste anche un RAID di livello 6 multidimensionale che organizza logicamente le unità in righe e colonne e implementando il RAID 6. Il sistema è in grado di ripristinare gli errori utilizzando blocchi di parità in una di queste posizioni.

- ❖ **Raid 0+1:** consiste in una combinazione del RAID 0 e 1. Il livello 0 fornisce le prestazioni, mentre il livello 1 l'affidabilità. In questo RAID si effettua lo **striping** su un insieme di dischi e si duplica ogni sezione con la tecnica del **mirroring**. Se si guasta un singolo disco, l'intera sezione di dati diventa inaccessibile, lasciando disponibile solo l'altra sezione.
- ❖ **Raid 1+0:** si fa prima il **mirroring** dei dischi a coppie e poi lo **striping** di queste coppie. Con un guasto in questo RAID, il singolo disco diventa inaccessibile ma il suo duplicato è ancora disponibile.

Il RAID può essere implementato sia a livello software, implementando il *RAID* nel kernel o a livello di programmi di sistema, che a livello hardware nel bus della macchina (**HBA**) o dall'array dei dischi. Può essere implementato anche da dispositivi di virtualizzazione del disco a livello di interconnessione SAN: in questo caso il dispositivo fa da intermediario tra le macchine e l'area di memorizzazione. Un'altra caratteristica del *RAID* è la previsione di **dischi di scorta** (*hot spare*) che possono sostituire immediatamente i dischi rotti.

9.8.4 Scelta di un livello RAID

Per scegliere un livello RAID bisogna considerare il **tempo di ricostruzione**, ovvero il tempo necessario a ricostruire i dati quando un disco si guasta. Tale tempo varia a seconda del livello RAID utilizzato.

- La ricostruzione più semplice si ha per il **RAID di livello 1** perché i dati possono essere copiati da un'altra unità; per gli altri livelli, è necessario accedere a tutte le altre unità dell'array e il tempo necessario aumenta. Il **RAID di livello 1** si usa nelle applicazioni che richiedono un'alta affidabilità e un rapido ripristino.
- Il **RAID di livello 0** si usa nelle applicazioni ad alte prestazioni in cui le perdite di dati non sono critiche.
- Il **RAID di livello 6** offre buone prestazioni una buona protezione senza un eccessivo overhead.

Altre caratteristiche da tenere conto sono il numero ottimale di dischi in un array per la capacità di trasferimento dei dati, e il numero di bit che ciascun bit di parità deve proteggere.

9.8.5 Problemi connessi a RAID

I sistemi RAID non assicurano sempre la disponibilità dei dati al SO e agli utenti. Per risolvere questi problemi, il file system **Solaris ZFS** usa una **checksum** interno ad ogni blocco, dati e metadati inclusi. Se si verifica un errore la **checksum** darà un valore errato e il *sistema ZFS* sostituirà il blocco errato con quello valido.

Un altro problema dell'implementazione RAID è la mancanza di flessibilità garantita da ZFS unendo i dischi in **gruppi di memorizzazione (pools of storage)** contenenti uno o più file system ZFS.

Linux ha un gestore di volume che consente l'unione logica di più dischi per creare volumi più grandi dei dischi stessi, in modo da poter contenere file system di grandi dimensioni.

Sistemi di I/O (Capitolo 10)

10.1 Introduzione

I due compiti principali di un calcolatore sono l'I/O e l'elaborazione. Il ruolo del SO nell'I/O è quello di gestire e controllare le operazioni e i dispositivi di I/O. I **driver** per il controllo dei dispositivi di I/O costituiscono il **sottosistema di I/O del kernel**.

10.2 Hardware di I/O

I dispositivi di I/O si dividono in base alla loro funzione: memorizzazione, trasmissione e interfaccia. Un dispositivo comunica con un elaboratore attraverso un punto di connessione detto **porta** e inviando segnali ad un canale di comunicazione detto **bus** che permette a tali dispositivi o periferiche di scambiarsi informazioni o dati con l'elaboratore stesso.

Il **bus PCIe** è il comune bus di sistema dei PC che connette il sottosistema *CPU-memoria* ai dispositivi veloci. Questo è un *bus* flessibile che invia i dati su uno o più canali, ciascuno composto da due coppie di fili, una coppia per la ricezione dei dati l'altra per la trasmissione.

Al **bus di espansione**, invece, si connettono i dispositivi lenti come la tastiera, le porte seriali e USB.

Un **controllore** è un insieme di componenti elettronici che può far funzionare una *porta*, un *bus* o un dispositivo, inoltre controlla i segnali presenti nei fili della porta seriale.

10.2.1 I/O memory mapped

Il processore invia comandi e dati al controller per eseguire un trasferimento di I/O tramite l'uso di speciali istruzioni di I/O che specificano il trasferimento di un byte o una parola a un indirizzo di porta I/O. In alternativa, il controllore di dispositivo può sopportare l'**I/O memory mapped** con la CPU che esegue le richieste di I/O utilizzando le istruzioni standard di trasferimento dati in memoria centrale. Tale comunicazione offre maggiore facilità di lettura e viene utilizzata comunemente con le schede video. Il controllo di un dispositivo di I/O consiste in genere in quattro registri: **status**, **control**, **data-in** e **data-out**.

- La CPU legge il registro **data-in** per ricevere dati.
- La CPU scrive nel registro **data-out** per emettere dati.
- Il registro **status** contiene alcuni bit che possono essere letti e indicano lo stato della porta.
- Il registro **control** può essere scritto per attivare un comando o per cambiare il funzionamento del dispositivo.

Certi controllori hanno circuiti integrati *FIFO* che possono contenere parecchi byte per l'immissione e l'emissione dei dati, in modo da espandere la capacità del controllore.

10.2.2 Modalità di scambio di dati

Lo scambio di dati con le periferiche può presentare alcune situazioni particolari per cui la CPU deve verificare, leggendo lo stato di un apposito registro del dispositivo, la disponibilità al dialogo. Le modalità di scambio di dati tra la CPU e le periferiche sono 3.

1) Polling

Il **polling** consiste nella scansione ciclica, da parte della CPU, di tutte le periferiche per verificare la disponibilità o meno alla comunicazione. Il **polling** si verifica quando la CPU legge ripetutamente il **bit busy** finché questo non vale 0. Il **polling** viene implementato tramite la semantica di I/O non bloccante.

Se il controllore e il dispositivo sono veloci, questo metodo è efficiente, ma se l'attesa si prolunga è meglio che la CPU si dedichi ad un'altra operazione. In molti calcolatori sono sufficienti tre istruzioni della CPU effettuare il polling di un dispositivo:

1. **read**: lettura di un registro del dispositivo;
2. **logical-and**: usata per estrarre il valore di un bit di stato;
3. **branch**: salto a un altro punto del codice se l'argomento è diverso da zero.

Il polling diventa inefficiente se le ripetute interrogazioni trovano raramente un dispositivo pronto per il servizio, mentre le altre elaborazioni attendono la CPU.

Anziché richiedere alla CPU di eseguire un'interrogazione ciclica, può essere più efficiente fare in modo che il controllore comunichi alla CPU che il dispositivo è pronto attraverso un **interrupt**.

2) Interrupt

Quando la CPU riceve un **interrupt**, sospende ciò che sta facendo e comincia ad eseguire codice a partire da una locazione fissa, che contiene l'indirizzo di partenza della *routine di interrupt*. Una volta completata la procedura richiesta, la CPU riprende l'elaborazione.

Il modo più semplice per gestire le interruzioni è quello di impiegare una **procedura generale** che esamina le informazioni presenti nel segnale di interruzione, porta a termine l'elaborazione necessaria e fa sì che la CPU ritorni nello stato in cui si trovava prima della sua interruzione attraverso l'istruzione `return from interrupt`.

Poiché la **gestione di un'interruzione** deve essere rapida, si può usare una tabella di puntatori alle specifiche procedure. L'accesso a questa sequenza di indirizzi, detta **vettore delle interruzioni**, avviene per mezzo di un indice.

La **gestione delle interruzioni** salva l'indirizzo dell'istruzione interrotta memorizzando l'indirizzo di ritorno nello *stack* di sistema. Terminato il servizio di interrupt, l'indirizzo di ritorno viene caricato nel **program counter**.

Nei SO moderni l'efficienza della gestione delle interruzioni è fornita dalla CPU e dal **controllore delle interrupt**. La maggior parte delle CPU ha due *linee di richiesta delle interruzioni*:

1. **interruzioni non mascherabili**: riservata a errori di memoria irrecuperabili.
2. **interruzioni mascherabili**: può essere disattivata dalla CPU prima dell'esecuzione di una sequenza critica di istruzioni che non deve essere interrotta.

Inoltre, poiché i calcolatori sono connessi a più dispositivi, per rendere le interrupt più rapide il SO usa il **concatenamento delle interruzioni** (*interrupt chaining*), in cui ogni elemento del vettore delle interruzioni punta alla testa di una lista di gestori delle interruzioni.

Ruolo del SO nelle interrupt

Un SO moderno interagisce con il meccanismo delle interrupt in vari modi. All'accensione della macchina esamina i bus per determinare quali dispositivi sono presenti, e installa gli indirizzi dei gestori delle interruzioni nel vettore delle interruzioni. Durante l'I/O, i vari controllori di dispositivi generano le interrupt quando sono pronti per un servizio.

I SO moderni usano le interrupt per gestire *eventi asincroni* e per eseguire procedure in kernel mode. Per fare in modo che i compiti più urgenti siano portati a termine, i calcolatori moderni usano un sistema di **priorità delle interrupt**.

3) DMA

Quando un dispositivo compie trasferimenti massicci si può generare un sovraccarico della CPU per questo si utilizza l'**accesso diretto alla memoria (DMA)**. Una volta impostati i buffer, il controllore trasferisce un intero blocco di dati direttamente nella memoria centrale senza alcun intervento da parte della CPU. Nei sistemi moderni si utilizzano gli switch, in cui più dispositivi fisici possono interagire con varie parti del sistema piuttosto che contendersi un unico bus.

La procedura di **handshaking** tra il controllore del DMA e il controllore del dispositivo si svolge grazie a una coppia di fili detti **DMA-request** e **DMA-acknowledge**. Il controllore del dispositivo manda un segnale sulla *DMA-request* quando una word di dati è disponibile per il trasferimento e il controllore DMA prende possesso del bus di memoria e manda un segnale al *DMA-acknowledge* (qui si verifica una sottrazione dei cicli della CPU). Quando il controllore del dispositivo riceve questo segnale, trasferisce in memoria la parola di dati e rimuove il segnale dalla linea *DMA-request*. Quando l'intero trasferimento termina, il controllore del DMA interrompe la CPU.

In alcune architetture la tecnica DMA è realizzata usando gli indirizzi della memoria fisica, mentre in altre s'impiega l'**accesso diretto alla memoria virtuale (DVMA)**, in questo caso si usano indirizzi virtuali che poi si traducono in indirizzi fisici.

10.3 Interfaccia di I/O delle applicazioni

Per fare in modo che un SO tratti le periferiche allo stesso modo si usa l'astrazione, l'incapsulamento e la stratificazione del software. A ciascuno di questi tipi si accede tramite un insieme standard di funzioni che ne costituiscono l'**interfaccia**, mentre le differenze sono incapsulate in moduli del kernel detti **driver del dispositivo**. Un **driver** è un software che permette ad un SO di utilizzare l'hardware senza sapere come esso funzioni, ma dialogandoci attraverso un'**interfaccia standard**.

Le funzioni di controllo diretto delle periferiche, dette **escape** o **back door**, permettono il passaggio trasparente di comandi direttamente dalle applicazioni ai driver dei dispositivi.

I dispositivi possono differire in molti aspetti:

- **Trasferimento a flusso di caratteri o a blocchi.** Un dispositivo del primo tipo trasferisce dati un byte alla volta mentre uno del secondo tipo ne trasferisce un blocco alla volta.
L'interfaccia per i dispositivi con trasferimento a blocchi si occupa di accedere ai drive di disco e ad altre periferiche a blocchi. Gli accessi da parte del SO e delle applicazioni possono trovare più conveniente trattare questi dispositivi come una semplice sequenza lineare di blocchi. In questo secondo caso si parla di **I/O grezzo**.
Le periferiche con trasferimento dati a caratteri sono ad esempio tastiere, mouse e modem, che producono dati in ingresso sottoforma di flussi sequenziali di byte, inoltre, **il trasferimento a caratteri è overhead**.
- **Dispositivi di rete.** Poiché i modi di indirizzamento e le prestazioni tipiche dell'I/O di rete sono molto diverse da quelli dell'I/O dell'unità a disco, la maggior parte dei SO fornisce un'interfaccia per l'I/O detta **socket**. Una volta creata una *socket* si possono usare le normali operazioni di I/O.

10.3.1 Orologi e timer

Nella maggior parte dei sistemi vengono utilizzati orologi (**clock**) e temporizzatori (**timer**) per tener traccia dell'ora corrente, del tempo trascorso, o per impostare un certo intervallo prima di produrre un interrupt. Quest'ultima funzione può essere utilizzata per implementare ad esempio il round-robin tra processi, o per svuotare periodicamente la cache del buffer, o per interrompere le operazioni di rete che ci stanno mettendo troppo tempo.

10.4 Sottosistema di I/O del kernel

Molti servizi di I/O sono messi a disposizione dal **sottosistema di I/O del kernel** e montati direttamente sull'hardware e sui driver delle periferiche.

10.4.1 Scheduling dell'I/O

Fare lo scheduling di un insieme di richieste di I/O significa stabilirne un ordine d'esecuzione efficace. Lo scheduling può migliorare le prestazioni del sistema, distribuire equamente gli accessi dei processi ai dispositivi e ridurre il *tempo d'attesa medio* per il completamento di un'operazione di I/O. I progettisti di SO realizzano tale scheduling mantenendo una **coda di richieste** per ogni dispositivo.

I kernel che mettono a disposizione di *I/O asincrono* devono essere in grado di tener traccia di più richieste di I/O contemporaneamente, per questo alcuni sistemi usano una **tabella dello stato dei dispositivi** alla coda dei processi in attesa. Gli elementi della tabella indicano il dispositivo e lo stato.

10.4.2 Gestione del buffer

Un **buffer** è un'area di memoria che contiene dati durante il trasferimento tra due dispositivi o tra un'applicazione e un dispositivo. Si ricorre al buffer per 3 ragioni:

- 1) la prima è la necessità di gestire la differenza di velocità tra produttore e consumatore. Un modem ad esempio che è più lento di qualsiasi altro dispositivo e ha bisogno di una **doppia bufferizzazione**.
- 2) Un secondo uso del buffer riguarda la gestione dei dispositivi che trasferiscono dati in blocchi di dimensione diversa.
- 3) Il terzo modo è per la realizzazione della **semantica delle copie** nell'ambito dell'I/O delle applicazioni. La *semantica delle copie* garantisce che la versione dei dati scritta nel disco sia conforme a quella contenuta nel buffer al momento della system call.

Interfaccia del file system (Capitolo 11)

11.1 Introduzione

Il **file system** è un meccanismo per la memorizzazione di dati e programmi appartenenti al SO e a tutti gli utenti del sistema. La maggior parte dei *file system* risiede su dispositivi di memoria secondaria. Il *file system* consiste in due parti diverse:

1. Un insieme di file;
2. Una struttura delle directory che organizza tutti i file nel sistema.

11.2 Concetto di file

Per facilitare l'uso del calcolatore si memorizza l'informazione sotto forma di un'unità logica di memorizzazione: il **File**, un insieme di informazioni, registrate in memoria secondaria, cui è assegnato un nome. Il contenuto del file è definito dal suo creatore, e può essere di diverso tipo (numerico, binario, alfanumerico, ecc..). Ogni file possiede una sua **struttura** interna la quale dipende proprio dal tipo di file.

11.2.1 Attributi dei file

Ogni file possiede un nome, il quale deve essere univoco, per poter essere identificato. Quando il file viene creato esso diventa indipendente dal processo e dal sistema. Per poter operare sul file ci devono essere diversi attributi, i quali variano da sistema a sistema, quelli generali sono: **Nome, Identificatore, Tipo, Locazione, Dimensione, Protezione, Ora, date e identificativo dell'utente**. Tutti questi attributi rappresentano il **descrittore** del file, che viene salvato dal SO.

11.2.2 Operazioni sui file

Un file è un **tipo di dato astratto**. Le operazioni che si possono effettuare su un file sono:

- **Creazione di un file**: bisogna trovare spazio nel file-system, successivamente creare il descrittore nella directory;
- **Scrittura/Lettura del file**: è indispensabile una system call che verifichi il nome e le informazioni che si vogliono scrivere, mentre per leggere c'è bisogno di una system call che specifichi anche la posizione in memoria dove collocare il successivo blocco del file;
- **Cancellazione del file**: è desiderabile cancellare un file quando non è più utile, deallocando lo spazio che occupava;
- **Troncamento di un file**: cancella solo il contenuto di un file ma non il file stesso per non doverlo ricreare una seconda volta.

Quando si devono effettuare più operazioni su un file, anziché aprire e chiudere lo stesso file, attraverso la system call `open()`, si memorizza il file in una piccola tabella detta **tabella dei file aperti** (residente in memoria centrale).

Quando si deve ricercare un file si passa l'indice di tale tabella, nel caso in cui il file sia aperto. La prima volta si accede alla memoria secondaria, con la chiamata `open()` si copia il descrittore del file all'interno della tabella. Per sapere quando rimuovere il descrittore dalla tabella si utilizza un contatore, quando è uguale a 0 si toglie dalla tabella. Alcuni file non potranno essere mai tolti da questa tabella perché sono file di sistema. Quando un processo vuole proteggere un file dall'accesso di altri processi, usa un **lock** che può essere di diversi tipi:

- **Lock shared**: più processi possono acquisire il lock contemporaneamente;
- **Exclusive**: un solo processo alla volta può acquisire il lock.

I lock hanno due tipi di approcci: **obbligatorio e consigliato**.

11.2.3 Tipi di file

Nella progettazione di un file system si deve sempre considerata la possibilità che questo ricerca e gestisca i tipi di file. La maggior parte dei SO alla fine del nome del file associa un'**estensione** costituita da un punto e da una serie di caratteri che identificano i tipi di file. *UNIX* usa un **numero magico** per identificare approssimativamente che tipo di file si ha mentre *Mac OS X* usa un attributo tipo associato al file nelle directory.

11.2.4 Struttura dei file

I tipi di file sono usati anche per identificare la **struttura interna del file**. Così facendo si potrà sapere in che modo questo file dovrà essere caricato in memoria. Se il SO gestisce molte strutture di file si possono verificare diverse problematiche:

- codice di sistema più ingombrante;
- incompatibilità di programmi con file di formato diverso;

UNIX e *DOS* considerano i file come semplici stringhe di byte e impostano un formato predefinito solo per i file eseguibili.

11.2.5 Struttura interna dei file

L'informazione all'interno dei dischi viene organizzata all'interno di un blocco, quindi quando di sposta informazione da e verso i dischi si spostano blocchi di dimensione fissa.

E' improbabile che la lunghezza di un record logico eguali un blocco fisico, dunque si **compatta** l'informazione all'interno di pacchetti. Quindi, saranno questi pacchetti ad essere memorizzati nei blocchi fisici. Tale metodo potrebbe soffrire di frammentazione interna in quanto non è detto che l'ultimo blocco sia riempito del tutto.

11.3 Metodi d'accesso

Ci sono diversi metodi per accedere ai file: se ne può scegliere uno, o più di uno, in base alle esigenze.

Accesso sequenziale

Il metodo più semplice è quello **sequenziale**, dove le informazioni vengono elaborate una dietro all'altra; questo metodo d'accesso è il più comune ed è usato dagli editor di testo e dai compilatori. Le più comuni operazioni sono di *lettura* e *scrittura*: la lettura fa avanzare il puntatore e può avvenire in qualunque posizione del file, mentre la scrittura avviene solo in coda al file.

Il *metodo di accesso sequenziale* è supportato sia sui dispositivi ad accesso sequenziale, sia su quelli ad accesso diretto.

Accesso diretto

Un altro metodo è l'**accesso diretto**. In questo caso, un file è formato da elementi logici (**record**) di lunghezza fissa; ciò consente che i file siano letti e scritti rapidamente senza un ordine particolare. Il *metodo ad accesso diretto* è molto utile quando è necessario accedere immediatamente a grandi quantità di informazioni. Spesso le basi di dati sono di questo tipo.

Si accede in base al numero di blocco che è passato dall'utente. Tale blocco è un **blocco fisico relativo**, così l'utente crede che inizi dal blocco 0 poi 1, poi 2 ecc. I blocchi assoluti possono non essere contigui e vicini. I blocchi relativi permettono al sistema di allocare il file dove vuole e non fa accedere l'utente a blocchi a cui non dovrebbe accedere.

Altri metodi di accesso

Altri tipi di accesi sono mediante gli **indici**. Quando si vuole accedere ad un blocco logico prima si cerca nell'indice e mediante un puntatore si accede al record logico.

Per file grandi, l'indice potrebbe diventare troppo lungo. Una soluzione è usare **due livelli di indici**. Si cerca il record logico tramite la prima scansione degli indici, la quale mi porta ad un altro indice. Da qui si accede al record logico voluto.

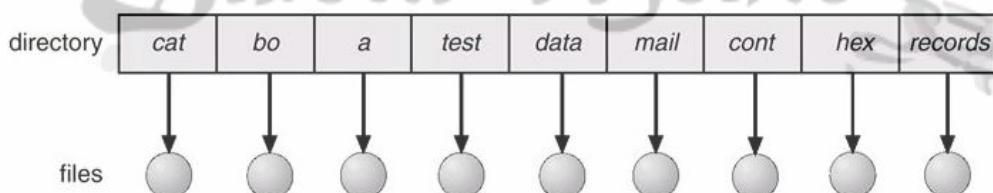
11.4 Struttura delle directory

La **directory** è un contenitore di file ma può contenere anche *sotto-directory*; in questo modo i dati sono organizzati in un sistema gerarchico di directory e sottodirectory. Le operazioni che si possono effettuare su una *directory* sono:

- **Ricerca di un file**: devo poter trovare un descrittore di file desiderato;
- **Creare un file**: e aggiungerlo alla *directory*;
- **Cancellare file**: non più necessario e rimosso dalla *directory*;
- **Elencare una directory**: elencare quindi il contenuto della directory e quindi tutti i suoi file;
- **Rinominare file**: permette anche di cambiare la posizione del file nella *directory*;
- **Attraversamento del file-system**: accedere a tutti i file che sono in una *directory*.

11.4.1 Directory a un livello

La struttura più semplice di una *directory* è quella a livello singolo dove tutti i file sono contenuti nella stessa *directory* e quindi possiedono nomi univoci. La situazione si complica quando il numero dei file aumenta oppure se il sistema è usato da più utenti poiché i nomi disponibili da attribuire ad un file scarseggiano. Fortunatamente, la maggior parte dei file system supporta nomi di file di 255 caratteri. Un altro inconveniente con una *directory* a un livello è che diventa difficile ricordare tutti i nomi presenti nella singola *directory*.



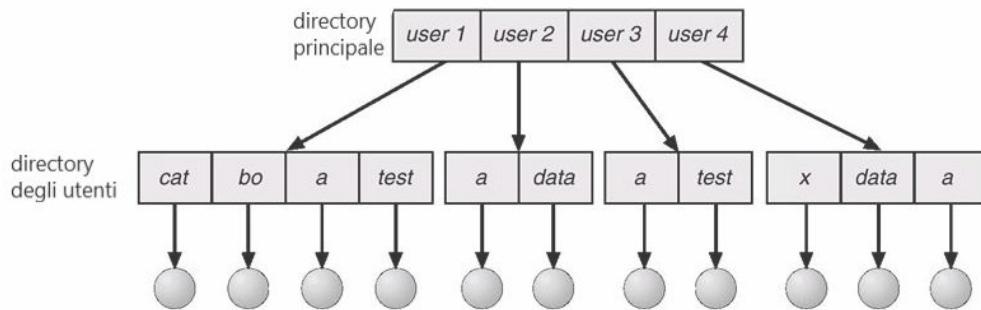
11.4.2 Directory a due livello

La struttura della directory a due livelli dispone per ogni utente una **directory personale (UFD)**. Tutte le *directory utente* hanno una struttura simile ma in ognuna sono elencate solo i file del proprietario. La *directory dell'utente* è detta **directory principale (MFD)**, indicizzata con il nome dell'utente o con il numero dell'account. Quando un utente effettua le varie operazioni su un file, il SO le svolge solo nella *directory* a lui associata. In questo modo diversi utenti possono avere file con lo stesso nome, purché tutti i nomi dei file della *directory* di un singolo utente siano unici.

La *directory a due livelli* ha lo svantaggio di isolare gli utenti qualora questi vogliono cooperare e accedere ai rispettivi file. Una *directory a due livelli* è vista come un albero di altezza 2:

- la **radice** è la *directory principale*;
- i **discendenti** sono le *directory utente*;
- le **foglie** dell'albero sono i *file*.

Il nome utente e il nome di un file definiscono il **path name**. Tramite la **search path** possiamo avviare la ricerca su un file o una directory scelta dall'utente.



11.4.3 Directory con struttura ad albero

La **struttura ad albero della directory**, con altezza arbitraria, permette di creare agli utenti delle **sottodirectory** e di accedere ai file di altri utenti specificando il **path name**. L'albero ha una **directory radice (root directory)**, e ogni file del sistema ha un unico **path name**. I **path name** possono essere di 2 tipi:

1. **percorsi assoluti**: cominciano dalla radice dell'albero
2. **percorsi relativi**: partono dalla directory corrente.

Una directory può contenere un insieme di file o sottodirectory. La distinzione tra file e directory è data da un bit di ogni elemento della directory. Ogni utente dispone di una **directory corrente** stabilita all'avvio del lavoro d'elaborazione.

Per quanto riguarda la cancellazione, se una directory è vuota la si può eliminare, se è piena o si cancella automaticamente tutto il suo contenuto oppure si devono cancellare manualmente ciascun file o sottodirectory contenuti e solo quando è vuota si può la si può cancellare.

Con la *struttura ad albero delle directory* l'accesso ai file di altri utenti è più facile da realizzare, infatti, l'utente B può accedere ai file dell'utente A specificando i **path name assoluti** o **relativi**.

11.4.4 Directory con struttura a grafo aciclico

La **struttura della directory a grafo aciclico** consente di condividere dei file o sottodirectory, in quanto con la struttura ad albero si poteva solo accedere a tale informazione. Con il **grafo aciclico** la stessa informazione è in più directory, ciò non vuol dire che i file sono duplicati, bensì sono condivisi. La condivisione può essere realizzata mediante un descrittore che funge da **link**: cioè un puntatore a un altro file o un'altra directory.

- Se si fa riferimento ad un file si cerca per prima cosa nella directory, se il descrittore di tale file è un *link* vuol dire che si fa riferimento ad un altro file. Il descrittore del *link* è in un formato diverso da quello dei normali file per poterlo identificare.
- Il secondo metodo prevede la duplicazione dei file permettendo di operare su due file diversi e quindi di avere delle copie non coerenti.

Un problema con la struttura aciclica è quando si vuole cancellare i file perché si potrebbero lasciare dei *puntatori pendenti*, oppure potrebbero far riferimento ad altri file nel caso lo spazio deallocated precedentemente venga rioccupato. Per poterlo cancellare ci dobbiamo assicurare di aver cancellato ogni riferimento di quel file, tenendo una **tabella dei riferimenti per ogni file**. Il problema è che la tabella potrebbe avere una dimensione molto grande. Per risolvere questo problema, si può usare solo un contatore all'interno di questa tabella: quando il contatore vale 0 si può cancellare il file.

11.4.5 Directory con struttura a grafo generale

Nel grafo aciclico non ci devono essere cicli al fine di poter implementare algoritmi efficienti per esplorare il grafo. La presenza dei cicli potrebbe far capitare un'esplorazione infinita. Un altro problema che potrebbe sorgere è quello di non sapere quando dover cancellare un file in quanto il contatore non si azzererebbe mai.

Si può utilizzare un **garbage collector** per stabilire quando sia stato cancellato l'ultimo riferimento e quando sia possibile riallocare lo spazio dei dischi. Questo metodo implica l'attraversamento del *file system* e ciò richiede molto tempo, per questo è usato di rado.

11.5 Protezione

Per computer che immagazzinano una grande quantità dei dati è consigliabile tenere tale informazione in un luogo sicuro. Quindi si potrebbero effettuare dei backup (**affidabilità**) e tenerli in posti lontani dalla macchina (**protezione**) per evitarli di perderli a causa di catastrofi naturali.

11.5.1 Tipi d'accesso

La protezione di un file è legata al suo metodo di accesso. I casi limite sono: far evitare di accedere al file qualsiasi utente, oppure far accedere tutti attraverso un **accesso controllato**. Controllare un file significa controllare le operazioni che possono essere svolte su tale file:

- lettura
- scrittura
- esecuzione
- accodamento
- cancellazione
- elenco

11.5.2 Controllo degli accessi

Il primo metodo di controllo dell'accesso è quello di mantenere una **lista dei controlli degli accessi (ACL)** a ogni file e directory: in questa lista si memorizzano gli utenti e il tipo di operazioni.

Il vantaggio di questo sistema è che permette metodi di accesso complessi; lo svantaggio è che è difficile realizzare tale lista in quanto la lista potrebbe essere molto lunga. Questi problemi sono risolvibili utilizzando una versione ridotta di tale lista che raggruppa gli utenti di ogni file in 3 classi diverse:

- **Proprietario**: l'utente che ha creato il file;
- **Gruppo**: l'insieme degli utenti che vi possono accedere con un sottoinsieme di privilegi del proprietario;
- **Universo**: tutti gli altri utenti del sistema.

In *UNIX*, ad esempio, un solo utente (*superuser*) può creare e modificare i gruppi. La protezione è affidata dall'uso di 3 campi, ciascuno è formato da un insieme di bit: ad esempio **rwx**.

- **r** controlla l'accesso per la lettura;
- **w** controlla l'accesso per la scrittura;
- **x** controlla l'accesso per l'esecuzione.

Gli utenti di Windows gestiscono la lista tramite un'interfaccia grafica.

11.5.3 Altri metodi di protezione

Un altro metodo per la protezione consiste nell'introdurre una **password** per quel determinato file. Cambiare spesso la password del file consente un alto grado di protezione. Lo svantaggio è che l'utente dovrebbe ricordarsi molte password in quanto potrebbe accedere anche ad altri file. Molti sistemi permettono di associare una password a una directory anziché a un singolo file.

Realizzazione del file system (Capitolo 12)

12.1 Introduzione

Il **file system** fornisce il meccanismo per la memorizzazione e l'accesso al contenuto dei file, compresi dati e programmi. Il *file system* risiede permanentemente nella memoria secondaria, progettata per ottenere in modo permanente grandi quantità di dati.

12.2 Realizzazione delle directory

La selezione degli algoritmi di allocazione e di gestione delle directory ha un grande effetto sull'efficienza, le prestazioni e l'affidabilità del *file system*.

12.2.1 Lista lineare

Il metodo più semplice per realizzare una directory è basato sull'uso di una **lista lineare** contenente i nomi dei file con puntatori ai blocchi di dati. Questo metodo è facile da programmare, ma nel caso venga usato frequentemente un file il tempo di attesa aumenta. Molti SO impiegano una cache per memorizzare le informazioni sulla directory usata più recentemente.

Un'alternativa, a tale metodo, sarebbe quella di utilizzare una lista ordinata che permette una ricerca binaria riducendo il tempo medio di ricerca ma aumenta il tempo di inserimento degli elementi. In questo caso può essere utile l'implementazione di un albero bilanciato.

Per creare un nuovo file occorre prima esaminare la directory per essere sicuri che non esista già un file con lo stesso nome, quindi aggiungere un nuovo elemento alla fine della directory. Per cancellare un file occorre cercare nella directory il file con quel nome, quindi rilasciare lo spazio che gli era assegnato. Per ridurre il tempo di cancellazione di un file si può usare anche una lista concatenata.

12.2.2 Tabella hash

La realizzazione della directory con una **tabella hash** avviene con la memorizzazione degli elementi in tale tabella. La *tabella hash* riceve un valore usando come operando il nome del file e riporta un puntatore al nome del file nella lista lineare. Attraverso questa struttura dati si può diminuire di molto il tempo di ricerca nella directory. L'inserimento e la cancellazione sono abbastanza semplici anche se occorre prendere provvedimenti per evitare **collisioni**: cioè situazioni in cui da due nomi di file si ottiene un riferimento alla stessa locazione. Per eliminare la collisione, ciascun valore della *tabella hash*, invece che un singolo valore, può essere una lista concatenata.

Il problema delle *tabelle di hash* è la dimensione, che in genere è fissa, e la dipendenza della funzione hash da tale dimensione. Tuttavia, la *tabella hash* è più veloce di una ricerca lineare nell'intera directory.

12.3 Metodi di allocazione

Esistono 3 metodi per allocare i file sul disco: **contigua**, **concatenata** o **indicizzata**. L'obiettivo è allocare lo spazio ai file in modo che lo spazio nel disco sia usato in modo efficiente e l'accesso ai file sia rapido.

12.3.1 Allocazione contigua

Per usare il metodo di **allocazione contigua**, ogni file deve occupare un insieme di blocchi contigui nel disco: si parte ad allocare dal blocco b , poi $b+1$, $b+2$, e così via. L'accesso all'informazione allocata in questo modo è semplice: infatti, il *file system* memorizza solo il primo blocco referenziato, poi avanza al successivo.

L'allocazione contigua presenta però alcuni problemi tra cui:

- soffre di **frammentazione esterna** come **l'allocazione dinamica della memoria** che usava dei metodi per scegliere quale buco libero usare: in particolare *first-fit* e *best-fit* (migliori di *worst-fit*) soffrono di *frammentazione esterna*.
La *frammentazione esterna* si ha quando lo spazio di memoria totale è sufficiente per risiedere in un processo, ma non è continuo, e quindi quello spazio è inutilizzato. Una soluzione è di utilizzare l'algoritmo di **compattazione** che consiste nel copiare il file system in un altro supporto e successivamente riscriverlo in modo compatto ma questo spreca molto tempo.
- Un altro problema più importante è quello che non si può sapere a priori quanto spazio occuperà un file durante la sua vita. Se si allocasse poco spazio per tale file non si potrebbe più estenderlo.
Esistono 2 soluzioni:
 - 1) nel momento della creazione si dà molto spazio;
 - 2) nel momento che questo cresce si copia il file in un buco libero più grande e quello precedente si libera.

Per ridurre al minimo questo problema alcuni SO usano una versione modificata: inizialmente si assegna una porzione di spazio contiguo, e se questa non è abbastanza grande si aggiunge un'altra porzione di spazio contiguo, detta **estensione**.

12.3.2 Allocazione concatenata

L'allocazione concatenata risolve tutti i problemi dell'*allocazione contigua*, infatti, ogni file è composto da una lista concatenata di blocchi del disco i quali possono essere sparsi ovunque nel disco stesso. Per creare un nuovo file si crea semplicemente un nuovo elemento della directory.

Un'operazione di scrittura nel file determina la ricerca di un blocco libero, la scrittura in tale blocco e la concatenazione di tale blocco. Per leggere un file basta scorrere tutto il file, infatti, non è necessario dichiarare a priori la grandezza del file.

Gli svantaggi dell'allocazione concatenata sono che questo tipo di allocazione può essere efficiente solo per i file ad accesso sequenziale (se voglio accedere ad un determinato blocco devo scorrere prima tutti quelli prima per poter raggiungere quello desiderato) in quanto, in quelli diretti, non è possibile sapere in modo veloce il blocco *i*-esimo. Un altro svantaggio è lo spazio richiesto per i puntatori.

Per risolvere questo problema è possibile riunire un numero di blocchi detti **cluster**, ed allocare i cluster anziché i blocchi. Questo metodo migliora il throughput del disco ma aumenta la frammentazione interna.

Un altro problema dell'allocazione concatenata riguarda l'affidabilità: poiché sono tenuti molti puntatori, se per errore uno di questi venisse cancellato, non si potrebbe accedere a tutto il file. Una variante del metodo di allocazione concatenata consiste nell'uso della **tabella di allocazione dei file (FAT)**. Per contenere tale tabella si riserva una sezione del disco all'inizio di ciascun volume; la FAT ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco; si usa come una lista concatenata.

Lo schema di allocazione basato sulla FAT, se non si usa una cache, può causare un aumento del numero di posizionamenti della testina. La testina deve spostarsi all'inizio del volume per leggere la FAT e poi leggere il blocco.