

Sistemi Operativi

Riassunto

Andrea Mansi 2018-2019 - UNIUD

Bibliografia:

Slides Prof. Marina Lenisa UNIUD

Slides Prof. Ivan Scagnetto UNIUD

Sommario

1. Introduzione	3
2. Richiami di Hardware	6
3. Struttura dei Sistemi Operativi.....	7
4. Processi & Threads.....	11
5. Scheduling dei Processi.....	18
6. Cooperazione tra Processi.....	25
7. Deadlock.....	34
8. Memory Management	38
9. Virtual Memory.....	45
10. Input / Output.....	52
11. Dischi	58
12. File System.....	65
13. Implementazione del File System.....	69

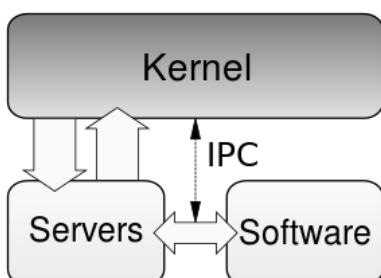
1. Introduzione

Sistema Operativo: Un sistema operativo, abbreviato S.O. è un software di sistema che agisce come intermediario tra l'utente e l'hardware del calcolatore. Un S.O. ha il compito di gestire le risorse hardware e software di quest'ultimo e di fornire servizi di base ai software applicativi installati. Inoltre, si pone l'obiettivo di nascondere tutti i dettagli tecnici legati allo specifico hardware e architettura su cui è installato, rappresentando le informazioni ad un alto livello, meglio comprensibile dall'uomo. Un sistema operativo fornisce all'utente una serie di comandi e servizi per usufruire al meglio della potenza di calcolo di un qualsivoglia elaboratore elettronico. Esso garantisce l'operatività di base di un calcolatore, coordinando e gestendo le risorse hardware di processamento, memorizzazione, input/output, le risorse/attività software (*processi*) e facendo da interfaccia con l'utente, senza il quale non sarebbe possibile l'utilizzo del calcolatore stesso. Riassumendo, un S.O. si pone principalmente l'obiettivo di creare una macchina astratta, realizzando funzionalità di alto livello, nascondendo i dettagli di basso livello; un S.O. mira quindi a soddisfare principalmente i seguenti criteri:

- Eseguire i programmi utente e semplificargli l'utilizzo del calcolatore.
- Gestire le risorse del calcolatore in modo sicuro ed efficiente.

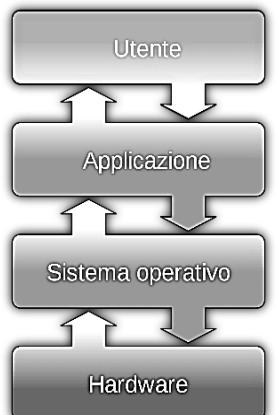
Questi obiettivi sono in contrapposizione (*avere un sistema più efficace ed efficiente possibile*). A quale obiettivo dare priorità dipende dal contesto e dalle politiche introdotte dal sistema operativo.

Kernel: Nucleo di un sistema operativo: Il kernel è un gruppo di funzioni fondamentali, strettamente interconnesse fra loro, e con l'hardware, che vengono eseguite con il privilegio massimo disponibile sulla macchina (*la modalità kernel designa proprio questo tipo di interazione*). Un kernel non è altro che un software che ha il compito di fornire ai moduli che compongono il sistema operativo e ai programmi in esecuzione sul calcolatore le funzioni fondamentali, inoltre fornisce egli un accesso controllato all'hardware, sollevandoli dai dettagli della sua gestione. Il kernel fornisce dunque le funzionalità di base per tutte le altre componenti del sistema operativo, che assolvono le loro funzioni servendosi dei servizi che esso offre, è dunque la colonna portante di un S.O..

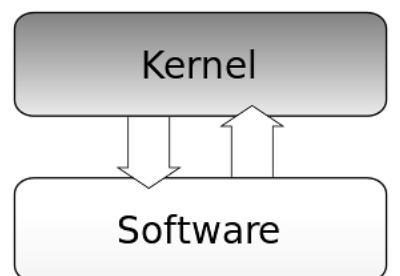


Schema di un Microkernel

A seconda del tipo di sistema operativo il kernel può inglobare altre parti (*kernel classico, monolitico o modulare*) o fornire solo funzioni base delegando più funzioni possibili a "servers" (*moduli*) esterni (*microkernel*). Un kernel tradizionale, ovvero monolitico, integra invece dentro di sé la gestione della memoria virtuale e della CPU, lo scheduler e i gestori del file system, nonché i driver necessari per il controllo di tutte le periferiche collegate. Quali funzioni sia opportuno che il kernel debba fornire e quali possano essere demandate a moduli esterni è oggetto di opinioni divergenti (*che danno appunto vita alle diverse tipologie di kernel sopracitate*). Il vantaggio di un sistema operativo con microkernel è la maggiore semplicità del suo kernel, del suo sviluppo, della possibilità di cambiare facilmente i moduli e di una certa tolleranza ai guasti in quanto se un modulo "crolla" (*crash*), solo la funzionalità del modulo in questione si interrompe, ed il sistema rimane funzionale e gestibile dall'amministratore (*che può ad esempio ripristinare la funzionalità del modulo stesso*); lo svantaggio è invece l'interazione più complessa e costosa fra kernel e le altre componenti del S.O. stesso, che spesso rallenta il sistema e/o lo rende meno stabile.



Schema di un S.O.



Schema di un Kernel Monolitico

Alcuni concetti e tipologie base di sistemi operativi:

Multiprogrammazione: In generale un singolo utente quando utilizza un calcolatore non può tenere costantemente occupati la CPU e i dispositivi di I/O. La multiprogrammazione consente di aumentare l'utilizzo della CPU organizzando il lavoro in modo tale da mantenerla in continua attività. L'idea su cui si fonda questo concetto è la seguente: il sistema operativo tiene nella memoria centrale diversi processi. Il S.O. sceglie uno tra quelli contenuti nella memoria e inizia ad eseguirlo. Esso a qualche punto potrebbe trovarsi in attesa di qualche evento, come il completamento di un'operazione di input output. In questo caso nei sistemi non multiprogrammati la CPU rimarrebbe inattiva, ma in questo caso, invece, il S.O. passa a un altro processo e lo esegue. Quando il primo processo ha terminato l'attesa, la CPU ne riprende l'esecuzione. Finché c'è almeno un lavoro da eseguire la CPU non rimarrà mai inattiva.

Multitasking: In informatica, un sistema operativo con supporto per il multitasking (*multiprocessualità*) permette di eseguire più programmi contemporaneamente. Se ad esempio viene chiesto al sistema di eseguire in contemporanea due processi A e B, la CPU eseguirà per qualche istante di tempo il processo A, poi per qualche istante successivo il processo B, poi tornerà a eseguire il processo A e così via. Il passaggio dal processo A al processo B e viceversa viene definito "*commutazione di contesto*" (*context switch*). Le decisioni riguardanti l'esecuzione di un cambio di contesto tra due programmi vengono intraprese da un componente del sistema operativo, lo scheduler processi, il quale invierà le proprie decisioni a un altro modulo del sistema operativo, il dispatcher, che eseguirà effettivamente il cambio di contesto. A seconda di quale strategia (*algoritmo di scheduling*) venga seguita, lo scheduler controlla la ripartizione del tempo di CPU tra tutti i processi attivi.

Sistemi Time-Sharing: Letteralmente "condivisione di tempo", è un approccio all'uso interattivo dei calcolatori in cui un singolo calcolatore viene utilizzato contemporaneamente da più utenti, ciascuno con un proprio terminale. Più nello specifico, l'esecuzione dell'attività della CPU viene suddivisa in quanti o intervalli temporali (*come nel multitasking*). Ogni quanto è assegnato sequenzialmente a vari processi di uno stesso utente o a processi di più utenti. La CPU del calcolatore viene quindi utilizzata per rispondere alle richieste dei singoli utenti, passando rapidamente da uno all'altro e dando l'impressione che ognuno abbia a disposizione il calcolatore centrale interamente per sé. Il time-sharing è correlato al multitasking nel senso che in ambedue i sistemi un singolo computer esegue più processi in modo che appare simultaneo. Tuttavia, il time-sharing fa riferimento ad un calcolatore che supporta più utenti simultaneamente, mentre il multitasking, come definito precedentemente è un termine più ampio che implica l'esecuzione di più processi "contemporaneamente". Nei sistemi time-sharing è più importante garantire un basso tempo di risposta rispetto al minimizzare il tempo di turnaround dei processi. Due persone che utilizzano lo stesso sistema time-sharing allo stesso momento potrebbero ricevere tempi di risposta molto differenti, infatti, il tempo di risposta dipende molto dal tipo di processo in esecuzione: un processo CPU-bound potrebbe avere un tempo di risposta molto più lungo di un processo interattivo (*processo I/O-bound*).

Sistemi Real Time: Un sistema operativo real-time è un sistema operativo specializzato per il supporto di applicazioni software real-time. Questi sistemi vengono utilizzati tipicamente in ambito industriale (*controllo di processo, pilotaggio di robot, trasferimento dati nelle telecomunicazioni*) o comunque dove sia necessario ottenere una risposta dal sistema entro un tempo prefissato. Un sistema operativo real-time non deve essere necessariamente veloce: non è importante l'intervallo di tempo in cui il sistema operativo/applicativo deve reagire; l'importante è che risponda entro un tempo massimo prefissato. In altre parole, il sistema deve essere prevedibile, nel senso che nel sistema si possa conoscere il tempismo reale (*nei migliori o peggiori dei casi*) di un determinato processo o elaborazione. Quindi un sistema real-time deve garantire che una elaborazione (*o job/task*) termini entro un dato vincolo temporale o scadenza (*detta in gergo "deadline"*). Per garantire questo è richiesto che la schedulazione delle operazioni sia fattibile. Il concetto di fattibilità di schedulazione è alla base della teoria dei sistemi real-time.

Sistemi Operativi di Rete: Sono quei sistemi operativi utilizzati quando è necessaria la distribuzione della computazione tra più processi in cui sono presenti più calcolatori che comunicano tra loro attraverso linee di comunicazione.

Sistemi Operativi Distribuiti: I sistemi operativi distribuiti vengono utilizzati quando più calcolatori sono collegati tra loro e si hanno più processi che comunicano mediante l'invio di messaggi. In questo caso si vuole ottimizzare l'utilizzo delle risorse evitando che esse risiedano in una macchina sola. Questo tipo di architettura di un sistema di elaborazione si chiama "network" ed è costituito da un insieme di calcolatori che, anche se a prima vista possono apparire come simili, possono essere dedicati a scopi diversi. L'utente ha una visione unitaria del sistema di calcolo.

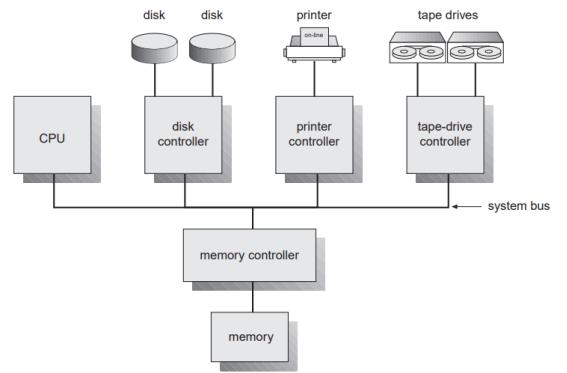
Kernel Prelazionabile: Un kernel si dice prelazionabile quando consente la prelazione di un processo anche quando opera in kernel-mode (ad esempio quando esegue il codice di una chiamata di sistema). I punti di prelazionabilità individuano delle posizioni nel codice delle chiamate di sistema in cui è possibile che il processo venga prelazionato, solitamente i punti di prelazionabilità vengono individuati in posizioni sicure del kernel, ovvero, laddove non vi siano in corso operazioni di modifica delle strutture dati di quest'ultimo (per non rischiare di creare delle inconsistenze). Tipicamente nei sistemi operativi per applicazioni tradizionali il kernel è prelazionabile in quanto non vi sono esigenze di risposte in tempi così rapidi come quelli tipici di questi sistemi. Invece nei sistemi real-time avere un kernel prelazionabile è fondamentale per evitare che una chiamata di sistema troppo lunga porti al fallimento dei vincoli temporali dei vari processi. I vantaggi di un kernel prelazionabile consistono nell'avere dei tempi di risposta molto più rapidi, raggiungendo un grado di parallelismo molto più alto rispetto ai kernel tradizionali. Gli svantaggi sono rappresentati dalla maggior complessità del kernel stesso e dal rischio di deadlock/inconsistenze in caso di errori di progettazione dei punti di prelazione o dell'utilizzo delle primitive di sincronizzazione.

2. Richiami di Hardware

Segue un riassunto delle principali nozioni richieste in ambito hardware per lo studio dei sistemi operativi. Alcune di queste nozioni verranno riprese e approfondite nei capitoli successivi.

- Struttura dei sistemi di calcolo
- Struttura dell'input/output
- Struttura della memoria
- Protezione hardware

Architettura dei calcolatori: Un sistema operativo (S.O.) è fortemente legato all'hardware del calcolatore su cui è installato. Infatti un S.O. estende l'insieme di istruzioni del calcolatore e inoltre gestisce le sue risorse hardware. Ne consegue che un S.O. deve avere un'ottima conoscenza interna dell'hardware, o almeno, di come quest'ultimo appaia ad alto livello. Questa è la ragione per cui tipicamente un S.O. è scritto in un linguaggio "vicino" alla macchina (assembly, C). Segue un'illustrazione di un modello astratto del calcolatore.



Struttura della memoria: Idealmente si vorrebbe che la memoria fosse estremamente veloce (più veloce del tempo richiesto per l'esecuzione di un'istruzione), molto capiente ed economica. In realtà la memoria si distingue in almeno due categorie: memoria principale ovvero la memoria a cui la CPU può accedere direttamente e la memoria secondaria, un'estensione della memoria principale che fornisce una memoria non volatile (e solitamente più grande).

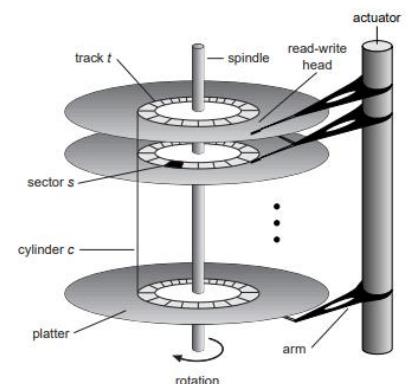
Dischi magnetici: Sono il principale dispositivo di archiviazione (memoria secondaria) utilizzato ai giorni nostri. Sono sostanzialmente dei piatti di metallo rigido in rotazione, ricoperti di materiale ferromagnetico. La superficie del disco è logicamente divisa in tracce, che sono sotto divise in settori; Il controller del disco determina l'interazione logica tra il dispositivo ed il calcolatore.

Input/Output: Con input output si indicano le fasi di scambio informazioni tra i dispositivi collegati al calcolatore e il calcolatore stesso. I dispositivi di I/O possono funzionare concorrentemente alla CPU, ciascuno di essi possiede un controller che permette l'interfacciamento con la CPU. Queste operazioni possono essere eseguite in modo sincrono, asincrono e DMA (Direct Memory Access). Tutto questo verrà spiegato in modo molto più dettagliato nei capitoli successivi.

Protezione Hardware: La condivisione di risorse di sistema richiede che il S.O. assicuri che un programma scorretto non possa portare altri programmi (*corretti*) a funzionare non correttamente. L'hardware deve fornire un supporto per differenziare almeno tra due modi di funzionamento:

- user mode: La CPU esegue codice utente
- kernel mode (o supervisor/system/monitor mode): la CPU sta eseguendo codice del sistema operativo.

Questa funzionalità richiede supporto a livello hardware: la CPU ha un mode bit che indica in quale modalità si trova. Mentre nella modalità user la CPU esegue il codice di un processo utente, nella modalità kernel essa esegue il codice del sistema operativo. L'utilità nel disporre di due modalità consiste nel fatto che tutte le operazioni potenzialmente pericolose per il sistema possono essere marcate come eseguibili soltanto in kernel mode, evitando che possano essere richiamate in modo erroneo dai processi degli utenti. Quindi soltanto il codice del sistema operativo (che si suppone essere stato scritto in modo da sfruttare correttamente l'hardware della macchina) può eseguire queste istruzioni privilegiate. I processi degli utenti potranno invocare i servizi oggetto delle istruzioni privilegiate attraverso il meccanismo delle chiamate di sistema che provocano il passaggio da user mode a kernel mode e viceversa.



Schematizzazione di un disco

3. Struttura dei Sistemi Operativi

➤ **Componenti comuni dei sistemi operativi:**

- Gestione dei processi
- Gestione della memoria principale
- Gestione della memoria secondaria
- Gestione dell'I/O (*input & output*)
- Gestione dei file (*Filesystem*)
- Sistemi di protezione (*Sicurezza*)
- Networking (*connessioni di rete*)
- Sistema di interpretazione dei comandi

Segue una breve descrizione di ogni componente: (*Gli argomenti vengono approfonditi nei cap. successivi*).

• **Gestione dei Processi:**

Un processo è un programma in esecuzione che necessita di certe risorse per assolvere il suo compito, ad esempio: tempo di CPU, memoria, file, dispositivi di I/O.

Il sistema operativo, in ambito della gestione dei processi è responsabile delle seguenti attività:

- Creazione e cancellazione dei processi
- Sospensione e resume dei processi
- Fornire meccanismi per:
 - Sincronizzazione dei processi
 - Comunicazione tra processi
 - Evitare, prevenire e risolvere i deadlock

• **Gestione della Memoria Principale:**

Il sistema operativo è responsabile della gestione della memoria principale, principalmente deve:

- Tener traccia di quali parti della memoria sono correttamente utilizzate e da chi.
- Decidere quale processo caricare in memoria quando dello spazio si rende disponibile.
- Allocare e deallocare spazio in memoria.

• **Gestione della memoria secondaria:**

Per quanto riguarda la gestione della memoria secondaria il sistema operativo deve:

- Gestire lo spazio libero
- Allocare lo spazio quando necessario
- Gestire la schedulazione dei dispositivi di archiviazione (*tipicamente dischi*)

• **Gestione dell'I/O:**

Il sistema operativo deve fornire una serie di meccanismi in ambito input/output:

- Sistemi di caching, buffering e spooling
- Una interfaccia generale ai gestori dei dispositivi
- Un ambiente di esecuzione per i driver di ogni specifico dispositivo

• **Gestione dei file:**

Mediante opportuni meccanismi (*forniti dal filesystem*) il S.O. deve essere in grado di gestire i file:

- Creazione e cancellazione dei file e delle directory
- Manipolazione di file e directory
- Allocazione dei file nella memoria secondaria

• Sistemi di protezione:

Per protezione si intende un meccanismo per controllare l'accesso da programmi, processi e utenti sia al sistema sia alle risorse. Il sistema operativo deve fornire meccanismi per:

- Distinguere uso autorizzato e non autorizzato delle risorse
- Gestire i controlli da imporre ed eseguire
- Forzare gli utenti e i processi a sottostare ai controlli richiesti

• Networking:

Un sistema distribuito, indica genericamente una tipologia di sistema informatico costituito da un insieme di processi interconnessi tra loro in cui le comunicazioni avvengono solo esclusivamente tramite lo scambio di opportuni messaggi. Il sistema operativo deve quindi fornire meccanismi per l'utilizzo di una rete di comunicazione.

• **Interprete dei comandi:** è la parte di un sistema operativo che permette agli utenti di interagire con il sistema stesso, impartendo comandi e richiedendo l'avvio di altri programmi. Insieme al kernel costituisce una delle componenti principali di un sistema operativo. Il suo nome (*dall'inglese shell, guscio*) deriva dal fatto che questa componente viene considerata l'involucro, la parte visibile del sistema ed è dunque definibile come l'interfaccia utente o il programma software che la rende possibile. La sua funzione è quindi di ricevere un comando ed eseguirlo.

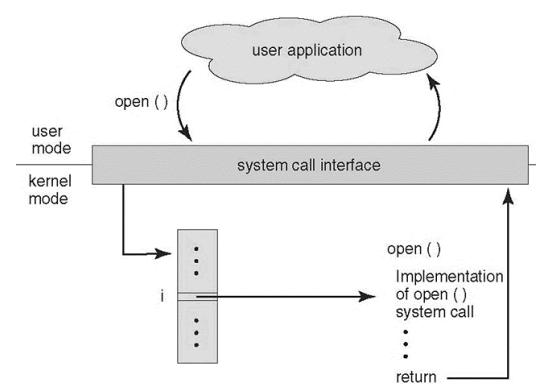
➤ Servizi dei sistemi operativi

Segue una lista dei principali servizi/funzionalità che un sistema operativo deve fornire allo user e ai processi.

- **Esecuzione dei programmi:** Caricamento dei programmi in memoria e successiva esecuzione.
- **Operazioni di I/O:** Il sistema operativo deve fornire un modo per condurre le operazioni di I/O all'utente, che non può eseguirle operando direttamente sull'hardware.
- **Manipolazione del file system:** Il S.O. deve fornire un meccanismo di gestione dei file (*file system*) che permetta di creare, cancellare, leggere, scrivere file e directory.
- **Comunicazioni:** Fornire un meccanismo di scambio informazioni tra processi in esecuzione sullo stesso computer o su diversi calcolatori collegati a una rete.
- **Individuazione degli errori:** Il S.O. deve garantire una computazione corretta individuando errori nell'hardware della CPU o della memoria, nei dispositivi di I/O o nei processi.
- **Allocazione delle risorse:** Funzionalità del S.O. che gestisce l'allocazione delle risorse tra vari utenti e processi che operano allo stesso momento.
- **Accounting:** Tiene traccia di chi è usato da cosa o da chi, a scopi statistici o di aumento delle performance.
- **Protezione:** Servizio che si pone l'obiettivo di assicurare che tutti gli accessi alle risorse di sistema siano controllate.

➤ System Calls (Chiamate di Sistema)

Le chiamate di sistema sono un meccanismo fornito dal sistema operativo per permettere agli utenti o ai processi di richiedere un servizio a livello kernel. Le system call sono tipicamente delle funzioni dei vari linguaggi di programmazione (*quelli che supportano la programmazione di sistema, ad esempio il C*) oppure particolari funzioni assembly. Le chiamate di sistema formano quindi l'interfaccia tra i programmi in esecuzione e il sistema operativo. Per motivi di sicurezza, il codice inerente alle system call del sistema operativo è eseguibile solo in kernel mode, avendo completo accesso all'hardware. Infatti, una chiamata al kernel, richiede spesso l'uso di una speciale istruzione di linguaggio macchina che provoca una commutazione di contesto del microprocessore (*da "modalità protetta" a "modalità supervisore (anche detta modalità kernel)" ovvero una context switch*).



Esistono tre metodi generali per passare parametri tra un programma e il sistema operativo:

- Passare i parametri tramite i registri della CPU.
- Memorizzare i parametri in una tabella in memoria principale, il cui indirizzo è passato tramite i registri.
- Il programma fa un “push” dei parametri sullo stack, il S.O. ne fa il “pop”.

Tipologie di chiamate di sistema: Esistono più categorie di system-call, ne seguono le principali:

- **Controllo dei processi:** Creazione e terminazione di processi. Esecuzione di programmi, allocazione e deallocazione della memoria principale. Attesa di eventi, etc...
es: load, execute, end, fork, wait...
- **Gestione dei file:** Creazione e cancellazione, apertura e chiusura, lettura e scrittura dei file.
es: create, delete, open, close, read, write...
- **Gestione dei dispositivi:** Allocazione e rilascio dei dispositivi, lettura e scrittura da/sui dispositivi.
es: request/release device, read, write, mount...
- **Informazioni di sistema:** Gestione delle informazioni di sistema, informazioni hw, software, ora etc...
es: get/set system data...
- **Comunicazione:** Gestione delle connessioni tra processi, dispositivi etc...
es: create/delete communication connection, send, receive, transfer...

➤ Programmi di Sistema

Un programma di sistema è un software che si pone fra il kernel e i normali programmi applicativi, fornendo dei servizi comuni indipendenti da questi ultimi; fanno parte del S.O. e vengono forniti insieme ad esso. I programmi di sistema forniscono un ambiente per lo sviluppo e l'esecuzione di altri programmi. Permettono (*implementandole*) di eseguire operazioni sull'hardware (*gestione dei file, modifica dei file, informazioni sullo stato del sistema e dell'utente, supporto dei linguaggi di programmazione, caricamento ed esecuzione dei programmi, comunicazione etc.*). Alcuni esempi di programmi di sistema: *compilatori, elaboratori di testo, interpreti comandi*. La maggior parte di ciò che un utente vede di un sistema operativo è definito dai programmi di sistema, non dalle reali chiamate di sistema.

- *Differenze tra Software di Sistema e Software Applicativo:*

Software di Sistema	Software Applicativo
Permette di operare sull'hardware e fornisce un ambiente per l'esecuzione e sviluppo di altri programmi.	Permette all'utente di eseguire specifiche operazioni/compiti.
In generale l'utente non interagisce con questi programmi, in quanto essi lavorano in background, e non forniscono servizi direttamente utili all'utente, ma piuttosto utili ad altri processi o al sistema operativo stesso.	Per definizione l'utente interagisce con questi programmi, sviluppati appunto per permettergli di eseguire specifici compiti.
Possono "girare" indipendentemente.	La loro esecuzione dipende dai software di sistema, che gli forniscono le operazioni di base per operare.
Esempi: Compilatori, Assembler, Debugger, IDE, driver etc.	Esempi: Word Processor, Web Browser, Videogames etc.

➤ Macchine Virtuali

Il termine macchina virtuale (*VM: Virtual Machine*) indica un software che, attraverso un processo di virtualizzazione, crea un ambiente virtuale che emula tipicamente il comportamento di una macchina fisica (*PC client o server*) grazie all'assegnazione di risorse hardware (*porzioni di disco rigido, RAM e risorse di processamento*) in cui alcune applicazioni possono essere eseguite come se interagissero con tale macchina; se dovesse andare fuori uso il sistema operativo che gira sulla macchina virtuale, il sistema di base non ne risentirebbe affatto. Tra i vantaggi vi è il fatto di poter offrire contemporaneamente ed efficientemente a più utenti diversi ambienti operativi separati, ciascuno attivabile su effettiva richiesta. Una VM fornisce una protezione completa delle risorse di sistema, dal momento che ogni macchina virtuale è isolata dalle altre. Questo isolamento non permette però una condivisione diretta delle risorse. Un sistema a macchine virtuali è un mezzo perfetto per l'emulazione di altri sistemi operativi, o lo sviluppo di nuovi sistemi operativi: tutto si svolge sulla macchina virtuale, invece che su quella fisica, quindi non c'è pericolo di fare danni.

Implementare una macchina virtuale è complesso, in quanto si deve fornire un perfetto duplicato della macchina sottostante. Può essere necessario dover emulare ogni singola istruzione macchina.

➤ Meccanismi e Politiche

Un concetto importante è quello della distinzione tra meccanismi e politiche.

- Un **meccanismo** indica come fare qualcosa.
- Una **politica** indica cosa deve essere fatto.

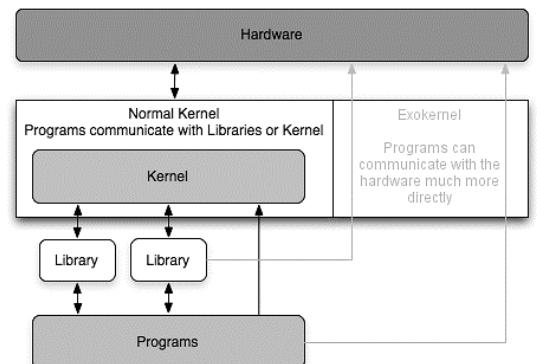
Questa distinzione permette elevata flessibilità nel kernel nel caso in cui le politiche debbano essere modificate (*e viceversa*). Un esempio classico è l'uso del timer per proteggere la CPU che rappresenta un meccanismo, mentre la decisione di quanti millisecondi assegnare ad ogni processo rappresenta una politica. Idealmente, se i meccanismi sono implementati bene, un cambiamento delle politiche comporta soltanto una riparametrizzazione dei meccanismi coinvolti (e non la loro completa sostituzione). Il sistema operativo Solaris è un esempio di questo approccio: l'utente può decidere di cambiare le politiche di scheduling del sistema, semplicemente caricando una nuova tabella di parametri che istruiscano i meccanismi sottostanti sui cambiamenti da apportare.

Rivediamo due tipologie di S.O. già accennate nell'introduzione: *exokernel* e *microkernel*.

➤ Exokernel

L'*exokernel* (*o esokernel*) permette di estendere il concetto di macchina virtuale, rendendo visibile a ciascuna di esse solo una porzione di risorse. L'idea centrale è: "separare la protezione dalla gestione". Nessuno meglio di uno sviluppatore sa come rendere efficiente l'uso dell'hardware disponibile, quindi l'obiettivo è dargli la possibilità di prendere le decisioni. Gli esokernel sono estremamente piccoli e compatti, poiché le loro funzionalità sono volutamente limitate alla protezione e all'allocazione delle risorse. I kernel "classici" (*sia monolitici che microkernel*) astraggono l'hardware, nascondendo le risorse dietro a un "livello di astrazione dell'hardware", approccio contrario a quello dell'*esokernel* che permette ad un'applicazione di richiedere aree specifiche di memoria, settori specifici su disco etc.

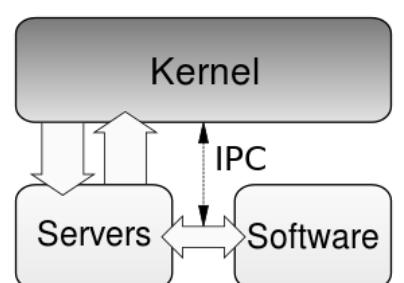
L'*esokernel* si assicura solo che le risorse richieste siano disponibili e che chi le richiede abbia il permesso di accedervi. Sostanzialmente l'*exokernel* deve solo tenere separati i domini di allocazione delle risorse dei vari sistemi operativi/VM che sono installati/e nel calcolatore.



Schema di un Exokernel

➤ Microkernel

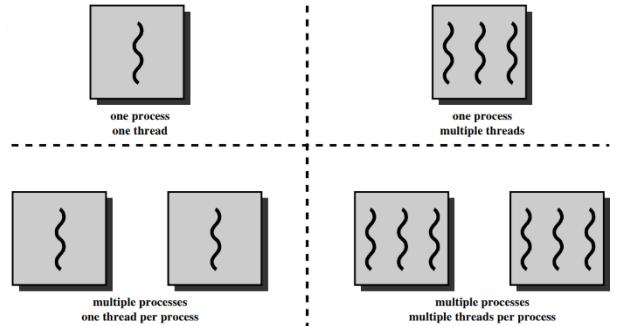
L'approccio *microkernel* consiste nel definire delle macchine astratte molto semplici sopra l'hardware, con un set di primitive o chiamate di sistema per implementare servizi minimi del sistema operativo quali gestione dei thread, spazi di indirizzamento o comunicazione interprocesso. L'obiettivo principale è la separazione delle implementazioni dei servizi di base dalle strutture operative del sistema. Per esempio, il processo di blocco (locking) dell'Input/Output può essere implementato come modulo server a livello utente. Questi moduli a livello utente, usati per fornire servizi di alto livello al sistema, sono modulari e semplificano la struttura e la progettazione del kernel. Un servizio server che smette di funzionare non provoca il blocco dell'intero sistema, e può essere riavviato indipendentemente dal resto.



Schema di un Microkernel

4. Processi & Threads

- **Processo:** Un processo è l'attività di esecuzione di un programma, comprende le operazioni che il processore deve portare a termine su richiesta dell'utente. Più precisamente è un'attività (*di passaggi sequenziali*) controllata da un programma che si svolge su un processore, in genere sotto la gestione o supervisione del rispettivo sistema operativo. Un processo comprende anche tutte le risorse di cui necessita.
- **Thread:** Il concetto di processo è associato, ma comunque distinto da quello di thread (*abbreviazione di thread of execution, filo dell'esecuzione*) con cui si intende invece l'unità granulare in cui un processo può essere suddiviso (*sotto processo*) e che può essere eseguito a divisione di tempo o in parallelo ad altri thread da parte del processore. In altre parole, un thread è una parte del processo che può essere eseguita in maniera concorrente ed indipendente internamente allo stato generale del processo stesso. Il termine inglese rende bene l'idea, in quanto si rifà visivamente al concetto di fune composta da vari fili attorcigliati: se la fune è il processo in esecuzione, allora i singoli fili che la compongono sono i thread. Un processo ha sempre almeno un thread (*sé stesso*), ma in alcuni casi un processo può avere più thread che vengono eseguiti in parallelo. Una differenza sostanziale fra thread e processi consiste nel modo con cui essi



condividono le risorse: mentre i processi sono di solito fra loro indipendenti, utilizzando diverse aree di memoria ed interagendo soltanto mediante appositi meccanismi di comunicazione messi a disposizione dal sistema, al contrario i thread di un processo tipicamente condividono le medesime informazioni di stato, la memoria ed altre risorse di sistema. Un sistema che supporta anche i thread (*più thread per processo*) viene chiamato: sistema multithread.

È quindi possibile condividere le risorse tra i vari thread, come vantaggio si ha una maggiore efficienza.

Vantaggi:

- Creare e cancellare thread è più veloce (*10-100 volte*): meno informazione da duplicare/creare/cancellare.
- Lo scheduling tra thread dello stesso processo è molto più veloce di quello tra i vari processi.
- La cooperazione di più thread nello stesso stack porta a un maggiore throughput e performance. (*es: in un file server multithread, mentre un thread è bloccato in attesa di I/O, un secondo thread può essere in esecuzione e servire un altro client*)

Svantaggi:

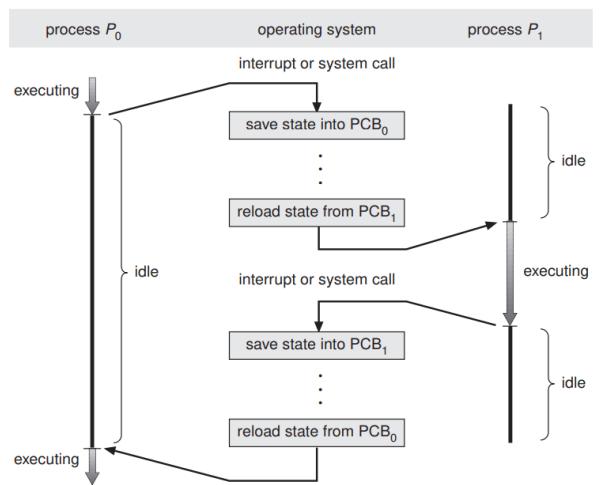
- Maggiore complessità di progettazione in fase di programmazione:
 - I processi devono essere “pensati” paralleli
 - Sincronizzare i thread è complicato
 - Si aggiunge la necessità di gestire anche lo scheduling tra i thread
 - Garantire la sicurezza è più complesso

Esempi di applicazioni multithread:

- **Lavoro foreground/background:** Mentre un thread gestisce l'I/O con l'utente, altri thread operano sui dati in background. *Esempio: Word Processor*
- **Elaborazione asincrona:** Operazioni asincrone possono essere implementate come thread. *Esempio: Salvataggio automatico su disco ogni x tempo.*
- **Task intrinsecamente parallele:** Se implementate con i thread risultano più efficienti.

➤ Context Switch (Cambio di contesto)

La commutazione di contesto (*in inglese context switch*) è una particolare operazione del sistema operativo che cambia il processo correntemente in esecuzione su una CPU. Questo avviene all'occorrenza di una qualsiasi interruzione dovuta allo scheduler dei processi, ma anche a interruzioni dovute a errori di altri processi o segnali. Viene effettuato per salvare tutte le informazioni necessarie al riavvio successivo del processo (*per esempio registri di CPU, stato del processo, indirizzo tabelle di paginazione*). Permette a più processi di condividere una stessa CPU, è utile quindi sia nei sistemi monoprocesso perché consente di eseguire più programmi contemporaneamente, sia nell'ambito del calcolo parallelo perché consente un migliore bilanciamento del carico. È importante ricordare che in corrispondenza di un cambio di contesto il S.O. deve salvare lo stato del processo corrente e caricare quello del nuovo processo, queste operazioni richiedono tempo che portano ad un certo *overhead*; il sistema non fa lavoro utile mentre esegue queste operazioni. L'*overhead* può essere anche molto elevato e portare ad un collo di bottiglia nei sistemi operativi ad alto livello di parallelismo (*con decine di migliaia di thread*). Il tempo di context switch dipende anche dal supporto hardware di questa operazione.



Schema logico di un context-switch

CPU Overhead: Definisce le risorse accessorie, richieste in eccesso rispetto a quelle strettamente necessarie per ottenere un determinato scopo in seguito all'introduzione di un metodo o di un processo più evoluto o più generale. Più in generale risulta essere il tempo in cui la CPU non esegue computazione utile. Le operazioni eseguite per il context switch sono un esempio di "spreco" di tempo CPU, che non viene usato per l'esecuzione di un processo utente ma per dei meccanismi di gestione del calcolatore.

➤ Creazione e terminazione dei processi

I processi possono venir creati in più contesti:

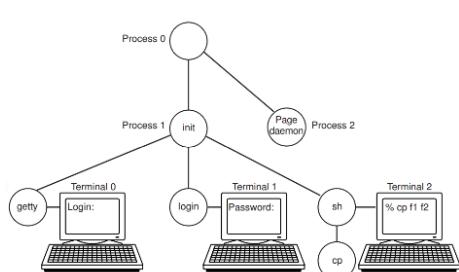
- Al boot del sistema
- Come effetto di una system call apposita
- Su richiesta dell'utente

La creazione dei processi induce una naturale gerarchia, detta albero dei processi.

I processi possono inoltre venir terminati in più contesti:

- Terminazione volontaria: fine dell'esecuzione
- Terminazione involontaria: a causa di un errore fatale o di operazioni illegali
- Terminazione da parte di un altro processo
- Terminazione da parte del kernel (*ad esempio termina il processo padre e il kernel "killa" i processi figli*).

Il sistema operativo si occupa di deallocare le risorse del processo in corrispondenza della sua terminazione.



Gerarchia dei processi: In alcuni sistemi, i processi generati (*detti figli*) rimangono collegati al processo padre (*parent, genitore*) dando vita a famiglie di processi (*gruppi*).

Esempi: In UNIX: tutti i processi discendono da init (PID=1). Se un parent muore, il figlio viene ereditato da init. Un processo non può diseredare il figlio. In Windows invece non c'è gerarchia dei processi; il task creator ha una handle del figlio, che comunque può essere passata da processo a processo.

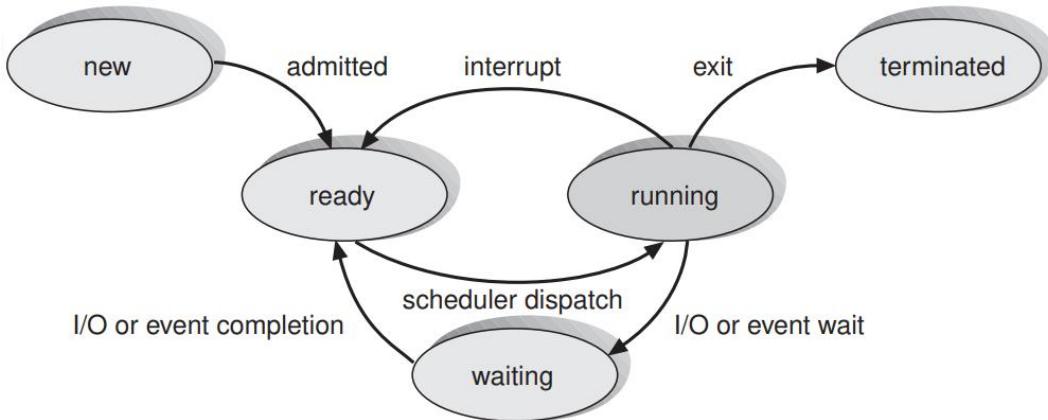
➤ Stato di un processo

Durante l'esecuzione, un processo cambia di stato. In generale si possono individuare i seguenti 5 stati più comuni:

- **new**: processo appena creato
- **running**: il processo è attualmente eseguito dalla CPU
- **waiting**: il processo è in attesa di qualche evento
- **ready**: il processo attende di essere assegnato ad una CPU
- **terminated**: il processo ha completato la sua esecuzione

Il passaggio da uno stato all'altro avviene in seguito a interruzioni, richieste di risorse non disponibili, selezione da parte dello scheduler, errori, etc...

Segue un “diagramma degli stati” raffigurante i passaggi di stato che può eseguire un processo.



➤ Process Control Block (PCB)

Il process control block o PCB (*in italiano: blocco di controllo del processo*) è la struttura dati di un processo, che contiene le informazioni essenziali per la gestione del processo stesso. In genere contiene queste informazioni:

- Stato del processo
- Dati identificativi del processo e dell'utente
- Program counter
- Registri della CPU
- Informazioni per lo scheduling della CPU
- Informazioni per la gestione della memoria
- Informazioni di utilizzo delle risorse
 - – tempo di CPU, memoria, file...
 - – eventuali limiti (quota)
- Stato dei segnali

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

PCB più nello specifico.

➤ Stati e Operazioni sui Thread

Un thread, molto similmente ad un processo può trovarsi in più stati: *running*, *ready* e *blocked*. Segue una lista delle principali operazioni effettuabili sui thread:

- **Creazione (o spawn):** viene creato un nuovo thread all'interno di un processo.
- **Blocco:** un thread si ferma, e l'esecuzione passa ad un altro thread o ad un altro processo. Può essere una azione volontaria o su richiesta di un evento.
- **Sblocco:** Un thread precedentemente bloccato viene sbloccato, passando da blocked a ready.
- **Cancellazione:** Il thread chiede di essere cancellato, il suo stack e le copie dei registri vengono deallocati.

➤ Implementazione dei Thread

I Thread possono essere implementati principalmente in due modi: a livello/spazio utente, o a livello/spazio kernel.

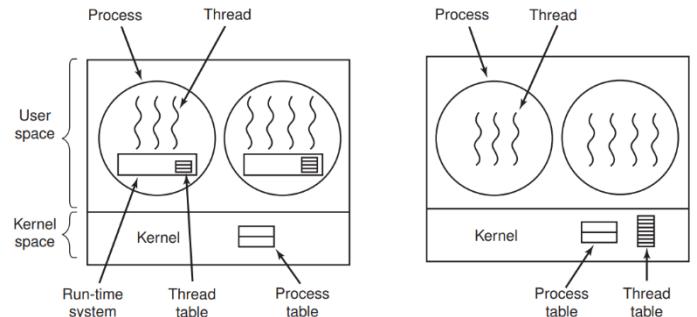
- **User-Level thread (ULT):** Stack, PC, e le operazioni sui thread sono implementate a livello utente.

- **Vantaggi:**

- Efficiente: non c'è il costo delle sys-call.
- Semplici da implementare.
- Portabilità elevata.
- Gli scheduler dei thread possono essere studiati specificatamente per ciascuna applicazione.

- **Svantaggi:**

- Non c'è scheduling automatico tra i thread, non c'è prelazione tra di loro: se un thread non passa il controllo esplicitamente monopolizza la CPU all'interno del processo. Inoltre, le system call bloccanti bloccano tutti i thread del processo.
- L'accesso al kernel diventa sequenziale
- Non sfrutta molto bene i sistemi multi-cpu.
- Poco performante su sistemi I/O bound.



- **Kernel-Level thread (KLT):** Il kernel gestisce direttamente i thread. Le operazioni sono ottenute attraverso system call appropriate.

- **Vantaggi:**

- Lo scheduling del kernel è per thread e non per processo, un thread che si blocca non blocca l'intero processo.
- Utile e molto performante per i processi I/O bound e per sistemi multi-cpu.

- **Svantaggi:**

- Meno efficiente, costo della sys-call per ogni operazione sui thread.
- Necessita l'aggiunta e la riscrittura di system call dei kernel preesistenti.
- Soluzione meno portabile.
- La politica di scheduling è fissata dal kernel, e non può essere modificata.

Esempi ULT: Mac OS < 9

Esempi KLT: Unix (Linux e Solaris), Windows, Mac OS >= 9

I Thread possono essere anche implementati con un approccio ibrido: ULT/KLT.

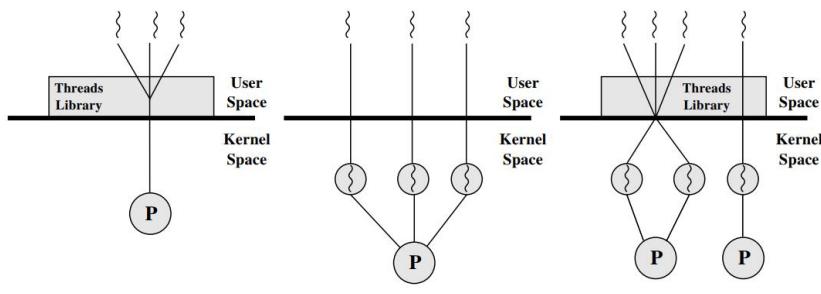
- **Sistemi Ibridi ULT/KLT:** Permettono sia thread a livello utente che a livello kernel.

- **Vantaggi:**

- Tutti quelli degli ULT e dei KLT perché grazie alla loro alta flessibilità, un programmatore può scegliere di volta in volta il tipo di thread che meglio si adatta alle sue esigenze.

- **Svantaggi:**

- Portabilità della soluzione scarsa.



(a) Pure user-level

(b) Pure kernel-level

(c) Combined

{ User-level thread

{ Kernel-level thread

P Processor

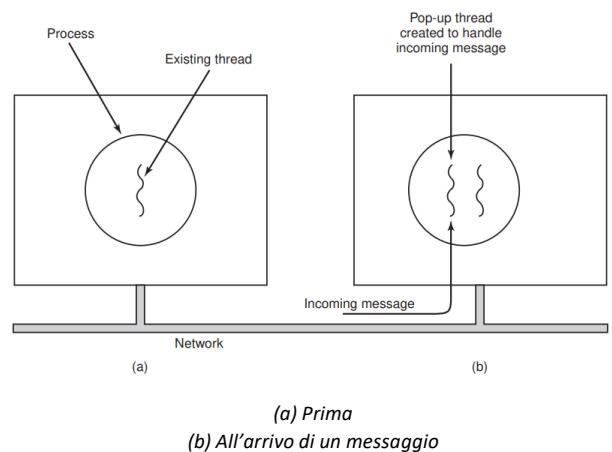
- **Thread Pop-up:** Vengono creati in modo asincrono da eventi esterni; all'arrivo di un messaggio il sistema crea un nuovo thread solo per gestire l'evento.

- **Vantaggi:**

- Molto utili per servizi a evento esterno e nei sistemi distribuiti.
- Bassi tempi di latenza (creazione rapida)

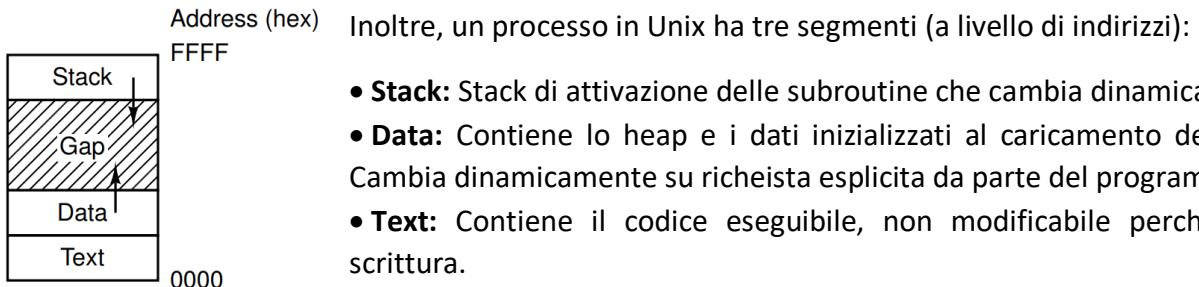
- **Svantaggi:**

- Scegliere dove eseguirli:
 - In user space risulta sicuro ma costoso
 - In kernel space viceversa.

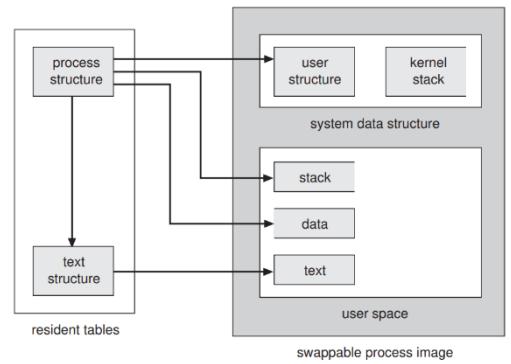


➤ **I processi in Unix tradizionale:** Segue un riassunto di come vengono gestiti i processi nel mondo UNIX.

In Unix viene assegnato, ad ogni processo, un valore identificativo detto PID (*Process Identifier*). Ogni processo è separato dagli altri, nel senso che non vede le zone di memoria dedicate agli altri processi.



In unix i PCB rappresentano i processi e sono in parte memorizzati nel kernel (*process structure e text structure*), e in parte nello spazio di memoria allocato al processo stesso (*user structure*). Le informazioni di questi blocchi sono usate dal kernel per il controllo e lo scheduling dei processi stessi. La process structure contiene: *stato del processo*, *puntatori alla memoria*, *identificatori del processo e dell'utente (PID e UID)*. La text structure è sempre residente in memoria e contiene dati relativi alla gestione della memoria virtuale per il text ed inoltre memorizza quanti processi stanno usando il segmento text. In unix quindi l'utente può creare e manipolare direttamente i processi mediante i PCB.



Process structure: Contiene le informazioni necessarie al sistema per la gestione del processo (*a prescindere dal suo stato attuale*). Contiene le seguenti informazioni:

- Un valore intero che rappresenta l'identificatore unico del processo (*Process IDentifier, PID*).
- Lo stato del processo.
- Puntatori alle varie aree dati e stack associati al processo.
- Riferimento indiretto al codice: la process structure contiene il riferimento all'elemento della text structure associata al codice del processo.
- Informazioni di scheduling (es: *priorità, tempo di CPU, etc.*).
- Riferimento al processo padre (*PID del padre*).
- Info relative alla gestione di segnali (es. *segnali inviati ma non ancora gestiti*).
- Puntatore al processo successivo nella coda di processi (*ad esempio, ready queue*).
- Puntatore alla user structure.

Text structure: Contiene informazioni riguardo la posizione del codice e chi ne condivide il contenuto.

User structure: Mantiene le informazioni sul processo richieste solo quando il processo risiede in memoria. Contiene:

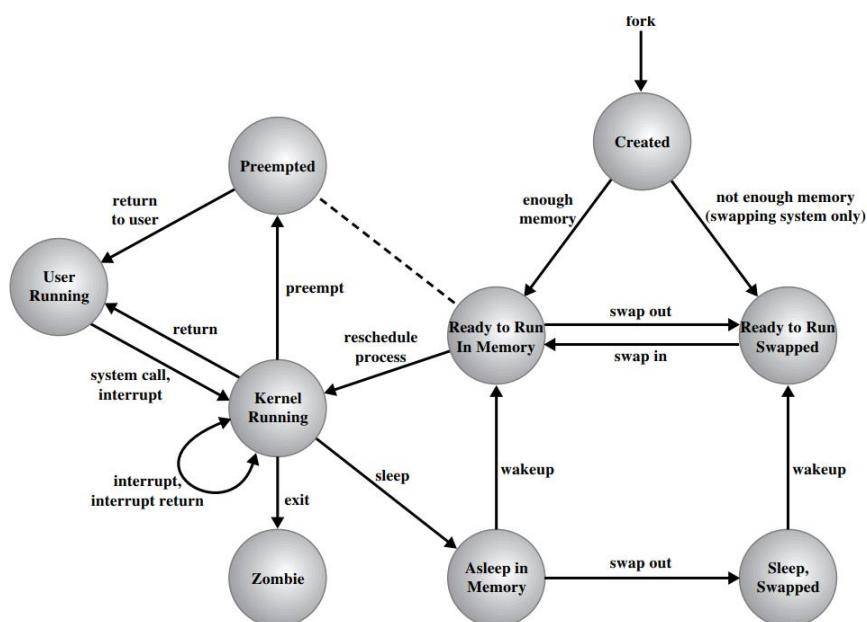
- Real UID, effective UID, real GID, effective GID.
- Gestione di ogni segnale (exit, ignore, esecuzione di una funzione).
- Terminale di controllo.
- Risultati/errori delle system call.
- Tabella dei file aperti.
- Limiti del processo.
- Mode mask (umask).

• Creazione di un processo in UNIX

Le principali system-call di UNIX utilizzate per creare un processo sono: fork(), vfork() e execve().

- La chiamata **fork()** alloca una nuova process structure per il processo figlio da lei creato. Vengono allocate nuove tabelle per la gestione della virtual memory, nuova memoria viene allocata per i segmenti data e stack del nuovo processo. Vengono copiati i segmenti data, stack e user structure preservando i file aperti, UID e GID etc.. Infine viene condiviso il text segment (entrambi i processi puntano alla stessa text structure).
- La chiamata **execve()** non crea nessun nuovo processo: semplicemente i segmenti data e stack vengono sostituiti.
- La chiamata **vfork()** non copia i segmenti data e stack, ma li condivide.

• Diagramma degli stati di un processo in UNIX



• Stati di un processo in UNIX:

Segue la lista dei possibili stati di un processo in UNIX.

- **User running:** Processo in esecuzione in modalità utente.
- **Kernel running:** Processo in esecuzione in modalità kernel.
- **Ready to run & in memory:** Processo pronto per andare in esecuzione.
- **Ready to run & swapped:** Processo eseguibile ma attualmente swappato in memoria.
- **Asleep in memory:** In attesa di un evento mentre è in memoria.
- **Sleeping & swapped:** Processo in attesa di un evento e swappato.
- **Preempted:** Processo bloccato dal kernel per mandare in esecuzione un altro processo.
- **Zombie:** Il processo non esiste più, si attende che il padre riceva l'informazione dello stato di ritorno.

- **Segnali POSIX:** Segnali utilizzati tra più processi, strumento base per la comunicazione.

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

- **Alcune System-Call UNIX:**

System call	Description
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause()	Suspend the caller until the next signal

5. Scheduling dei Processi

Lo scheduling dei processi è un'operazione molto importante per il corretto ed efficiente funzionamento del calcolatore. Infatti, non solo consente di eseguire più programmi concorrentemente, ma consente anche di migliorare l'utilizzo del processore. Ad esempio, quando è necessario eseguire un'operazione di I/O, il processore non può proseguire l'elaborazione del processo attualmente in esecuzione fino al suo completamento. Dato che le operazioni di I/O sono molto più lente del processore, sarebbe un inutile spreco di risorse se il processore rimanesse bloccato fino al loro completamento. Per evitare questo le operazioni di I/O vengono gestite unicamente dal sistema operativo che, nel frattempo, assegna l'uso del processore ad un altro processo. In questo modo si massimizza l'uso delle risorse del sistema.

È importante la distinzione tra scheduling con diritto di prelazione (*scheduling preemptive*) e scheduling senza diritto di prelazione (*scheduling non-preemptive o cooperative*). Nel primo caso lo scheduler può sottrarre il possesso del processore ad un processo anche quando questo potrebbe proseguire nella propria esecuzione. Nel secondo caso, invece, lo scheduler deve attendere che il processo termini o che cambi il suo stato da quello di esecuzione a quello di attesa (*wait*) o di pronto (*ready*), a seguito, ad esempio, di una richiesta di I/O oppure a causa di un segnale di interruzione (*interrupt*). I vantaggi di un algoritmo di schedulazione con prelazione (*preemptive*) sono essenzialmente dei tempi di risposta migliori e la garanzia che nessun processo riesca a monopolizzare la CPU senza rilasciarla. Gli svantaggi sono relativi alla condivisione dei dati fra processi, infatti, se un processo sta manipolando dei dati utilizzati anche da altri processi e viene prelazionato c'è il rischio che questi rimangano in uno stato inconsistente e generino così degli errori.

Esistono vari algoritmi di scheduling che tengono conto di varie esigenze e che possono essere più indicati in alcuni contesti piuttosto che in altri. La scelta dell'algoritmo da usare dipende principalmente dalla valutazione di cinque aspetti:

- **Utilizzo del processore:** la CPU deve essere attiva il più possibile, devono essere ridotti al minimo i possibili tempi morti. (*overhead minimo*).
- **Throughput:** il numero di processi completati in una determinata quantità di tempo.
- **Tempo di completamento (o di turnaround):** il tempo che intercorre tra la sottomissione di un processo ed il completamento della sua esecuzione.
- **Tempo d'attesa:** il tempo in cui un processo pronto per l'esecuzione rimane in attesa della CPU (*waiting time*).
- **Tempo di risposta:** il tempo che trascorre tra la sottomissione del processo e l'ottenimento della prima risposta (*non l'output ma la prima risposta*).

Per analizzare gli algoritmi che verranno successivamente presentati verrà utilizzato come criterio di valutazione e di confronto il tempo d'attesa medio dei processi presi in considerazione.

➤ Obiettivi generali di un algoritmo di scheduling

Un algoritmo di scheduling si pone i seguenti obiettivi, che possono variare da tipologia a tipologia di sistema.

- **Su tutti i sistemi (tipicamente):**
 - *Fairness (equità)*: processi dello stesso tipo devono avere trattamenti simili.
 - *Balance (bilanciamento)*: tutte le parti del sistema devono essere sfruttate in modo bilanciato.
 - Garantire che la politica scelta venga applicata.
- **Sistemi batch:**
 - *Vengono privilegiati: throughput, tempo di turnaround e utilizzo CPU.*
- **Sistemi interattivi:**
 - *Vengono privilegiati: tempo di risposta e proporzionalità.*
- **Sistemi real-time:**
 - *Vengono privilegiati: rispetto delle deadline e predicitività.*

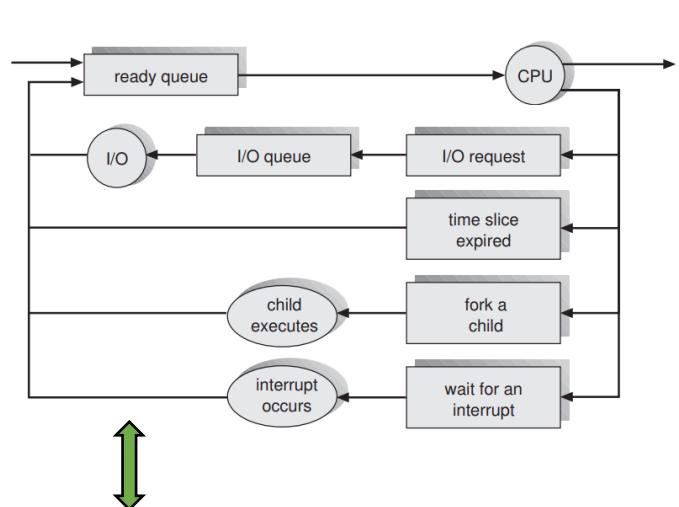
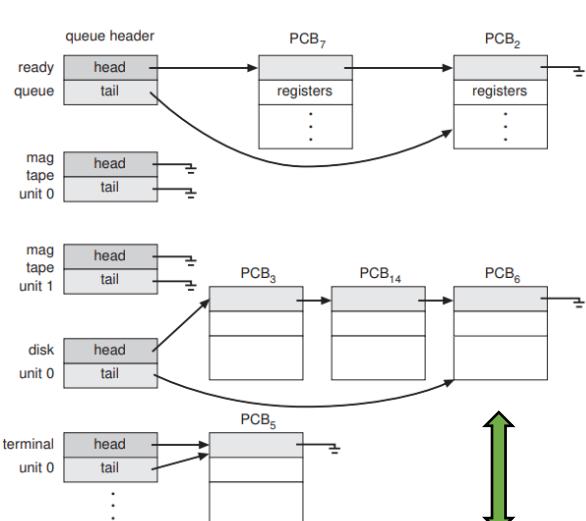
NB: In generale non esiste una soluzione ottima sotto tutti gli aspetti, molte richieste vanno in contrapposizione e bisogna bilanciarle.

➤ Code di Scheduling dei Processi

Un sistema operativo presenta più code di scheduling (*nell'ambito dei processi*), in generale abbiamo:

- **Coda dei processi:** insieme di tutti i processi nel sistema
- **Ready Queue:** insieme dei processi residenti in memoria e in attesa di essere messi in esecuzione
- **Coda dei dispositivi:** processi in attesa di un dispositivo di I/O

I processi durante l'esecuzione possono migrare da una coda all'altra, il compito di scegliere e gestire quale processo passa da una coda all'altra spetta agli scheduler.

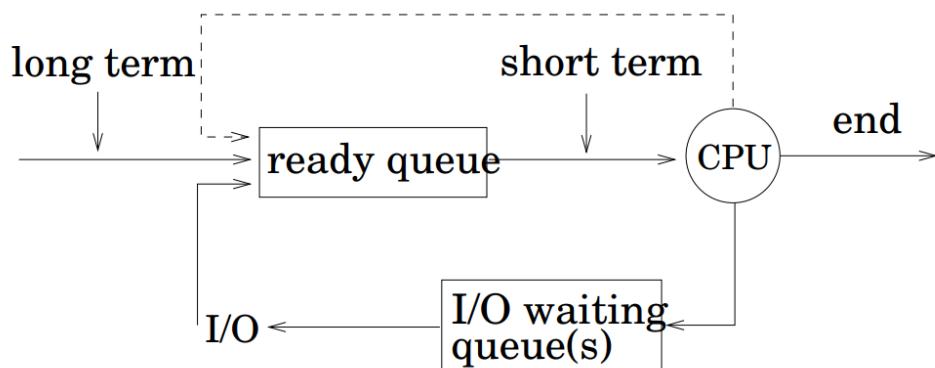


➤ Scheduler Processi

Ricapitolando quanto detto, lo scheduler è quel componente di un sistema operativo che implementa l'algoritmo di scheduling, il quale dato un insieme di richieste di accesso ad una risorsa (*tipicamente l'accesso al processore da parte di un processo da eseguire*), stabilisce un ordinamento temporale per l'esecuzione di tali richieste, privilegiando quelle che rispettano determinati parametri secondo una certa *politica di scheduling*, in modo da ottimizzare l'accesso a tale risorsa e consentire così il compimento del servizio/istruzione o processo desiderato. L'attenzione posta su alcuni parametri piuttosto che su altri differenzia la cosiddetta politica di scheduling all'interno della gestione dei processi dando vita a code di priorità.

Per quanto riguarda lo scheduler dei processi, se ne possono identificare principalmente due:

- Lo **scheduler di lungo termine** (*o job scheduler*) che seleziona i processi da portare nella ready queue, invocato raramente (*secondi o minuti*) tipicamente lento e sofisticato. Lo scheduler di lungo termine controlla il grado di multiprogrammazione e cerca di mantenere un giusto equilibrio tra processi I/O e CPU bound.



- Lo **scheduler di breve termine** (*o CPU scheduler*) che seleziona quali processi ready devono essere eseguiti, assegnandogli quindi la CPU. Viene invocato molto frequentemente (*decine di volte al secondo*). Non può quindi essere troppo complesso altrimenti rallenterebbe troppo il sistema. Lo scheduler a breve termine può prendere decisioni quando un processo:
 - passa da running a waiting.
 - passa da running a ready.
 - passa da waiting o new a ready.
 - termina.

Nel primo e quarto caso si parla di schedulazione senza prelazione, altrimenti con prelazione.

Alcuni sistemi operativi hanno anche uno scheduler aggiuntivo: lo scheduler di medio termine.

- Lo **scheduler di medio termine** (*o swap scheduler*) che sospende temporaneamente alcuni processi per abbassare il livello di multiprogrammazione.

➤ Dispatcher

Il dispatcher è il modulo che dà il controllo della CPU al processo selezionato dallo scheduler di breve termine. Questo passaggio comporta:

- Switch di contesto,
- Passaggio della CPU da modalità *supervisor* a modalità *user*,
- Salto alla locazione del programma utente per riprendere il processo.

È essenziale che il dispatcher sia veloce, perché viene chiamato di continuo. Viene chiamata *latenza di dispatch* il tempo necessario per fermare un processo e riprenderne un altro; contribuisce ad aumentare l'overhead della CPU.

Prima di elencare gli algoritmi di scheduling, vengono ripetuti e spiegati alcuni approcci/concetti.

Ricordiamo il concetto di **prelazione**:

- Con prelazione, se il processo può passare dallo stato running, waiting o new allo stato ready.
- Senza prelazione se il processo passa da running a waiting oppure se termina.

• Scheduling a priorità

Si parla di scheduling a priorità quando ad ogni processo viene associato un numero (*finito e intero*) di proprietà. La CPU viene allocata al processo con la priorità più alta.

Le proprietà possono essere definite:

- **Internamente:** in base a dei parametri misurati dal sistema operativo sul processo (*cpu time, file aperti, memoria, interattività, uso di I/O etc...*)
- **Esternamente:** importanza del processo, dell'utente proprietario e altre proprietà scelte a priori.

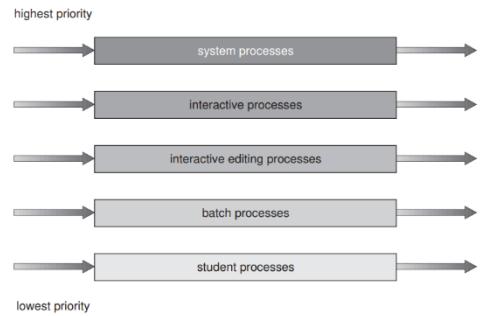
Questi algoritmi possono essere sia con che senza prelazione. Questo approccio può portare al problema della *starvation*: i processi a bassa priorità possono venire bloccati da un flusso di processi a priorità maggiore, e venir quindi eseguiti molto in ritardo. La soluzione principale è aumentare la priorità di un processo all'aumentare della loro età.

• Scheduling con code multiple

Con scheduling a code multiple si intende la suddivisione della coda ready in più sotto code, ad esempio: la coda per i processi foreground (*interattivi*) e per i processi background.

Ogni coda ha un suo algoritmo di scheduling, specifico per quel tipo di attività. Lo scheduling deve comunque avvenire su tutte le code:

- Scheduling a priorità fissa: eseguire i processi di una coda solo se le code di priorità superiore sono vuote, si ha rischio di starvation.
- Quanti di tempo per ciascuna coda: ogni coda riceve un certo ammontare di tempo CPU. La priorità di una coda rispetto ad un'altra viene caratterizzata dalla percentuale di tempo ad essa assegnata.



Si parla di scheduling a code multiple con *feedback* se i processi vengono spostati da una coda all'altra dinamicamente in base a determinati criteri. Ad esempio, vengono spostati in base alla quantità di CPU utilizzata. Uno scheduler a code multiple con feedback viene quindi definito dai seguenti parametri: numero di code, algoritmo di scheduling per ogni coda, come determinare quando promuovere un processo, come determinare quando degradare un processo e come determinare la coda in cui mettere un processo che entra nello stato di ready.

• Schedulazione garantita

Si promette ad ogni processo un certo *quality of service* che poi deve essere mantenuto. *Esempio: ci sono n utenti, ad ogni utente si promette 1/n tempo CPU.*

• Schedulazione a lotteria

È un semplice esempio di implementazione di una schedulazione garantita. Esistono un certo numero di *ticket* per ogni risorsa, ad ogni processo viene concesso un sottoinsieme di tali ticket. Lo scheduler estrae casualmente un ticket, e la risorsa viene assegnata al vincitore. Per la legge dei grandi numeri, alla lunga l'accesso alla risorsa è proporzionale al numero dei ticket posseduti. I ticket possono essere passati da un processo all'altro per cambiare proprietà.

• Cenni di scheduling dei processi in ambito multi-CPU

Lo scheduling diventa più complesso quando sono disponibili più CPU. In caso di sistemi *omogenei* è indifferente su quale di essi viene eseguito il prossimo processo. Può però essere necessario che un certo processo venga eseguito da un processore preciso (*pinning*). Per bilanciare il carico si fa sì che tutti i processi selezionano i processi dalla stessa ready queue. Si verifica però il problema di accesso condiviso alle strutture del kernel: si parla di Asymmetric multiprocessing (*AMP*) se solo un processore per volta può accedere alle strutture dati del kernel, questo approccio semplifica il problema, ma diminuisce le prestazioni (*carico non bilanciato*); mentre si parla di Symmetric multiprocessing (*SMP*): condivisione delle strutture dati. Serve hardware particolare e controlli di sincronizzazione in kernel.

• Scheduling real-time

Nei contesti real-time vanno distinti i casi di *hard real-time* e *soft real-time*.

- **Hard real-time:** Si richiede che un task critico venga completato entro un tempo ben preciso e garantito. In questo ambito è possibile la prenotazione delle risorse, inoltre, non si possono usare memorie virtuali, connessioni di rete e tutti quei meccanismi che potrebbero portare a rallentamenti non prevedibili.
- **Soft real-time:** Sistema in cui alcuni processi critici sono prioritari rispetto agli altri. Possono coesistere con i normali sistemi time-sharing, lo scheduler deve mantenere i processi real-time prioritari.

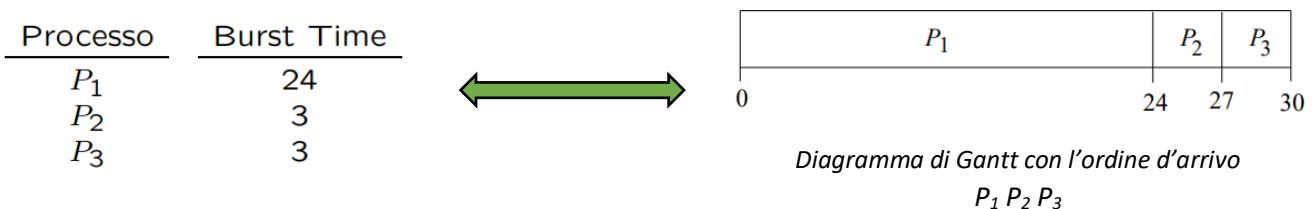
In questo contesto è anche utile categorizzare gli eventi come *aperiodici* (imprevedibili, ad esempio: segnalazione da un sensore) o *periodici* (avvengono ad intervalli di tempo regolari o prevedibili).

➤ Algoritmi di Scheduling dei processi

Esistono più algoritmi di scheduling dei processi e di diverse tipologie. Segue una lista di quelli principali e la loro successiva spiegazione.

- **FCFS: First-Come First-Served**
- **SJF: Shortest-Job-First**
 - Senza prelazione
 - Con prelazione (SRTF: Shortest-Remaining-Time-First)
- **RR: Round Robin**
- **RMS: Rate Monotic Scheduling**
- **EDF: Earliest Deadline First**

FCFS: First-Come First-Server: L'algoritmo esegue i processi in ordine di arrivo, senza prelazione. Non c'è pericolo di starvation (è quindi di tipo FIFO) Segue un esempio:



NB: Burst Time: Tempo richiesto da un processo per completare la sua esecuzione.

Tempi di attesa: $P_1 = 0$, $P_2 = 24$, $P_3 = 27$.

Tempo di attesa medio: $(0 + 24 + 27) / 3 = 27$

Effetto convoglio: i processi I/O-bound si accodano dietro un processo CPU-bound.

SJF: Shortest-Job-First: Associa ad ogni processo la lunghezza del suo prossimo burst di CPU. I processi vengono schedulati in ordine di tempi crescenti. L'algoritmo SJF risulta essere quello ottimale perché fornisce il minimo tempo di attesa per un dato insieme di processi. È un algoritmo di scheduling a priorità, dove la priorità è il prossimo tempo di burst. C'è il rischio di starvation, i processi con burst-time lungo potrebbero essere rimandati all'infinito.

Problema: Non potendo conoscere il tempo per il quale il job occuperà la CPU, il sistema operativo utilizzerà i dati delle precedenti elaborazioni per fare delle stime sulla durata del prossimo burst. Non possiamo prevedere il futuro.

Stima del burst time:

$$tn = \text{tempo dell}'n\text{-esimo burst di CPU (di quel processo)}$$

$$Tn+1 = \text{valore previsto per il prossimo burst di CPU}$$

$$\alpha = \text{parametro compreso tra 0 e 1}$$

→

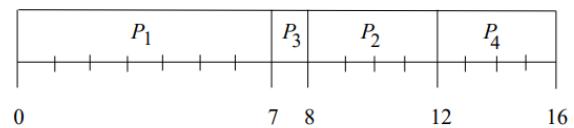
$$Tn+1 = \alpha \times tn + (1 - \alpha) \times Tn$$

È possibile implementare l'algoritmo SJF in due schemi:

- **non-preemptive (senza prelazione):** Quando l'algoritmo ha assegnato la cpu ad un processo, questo la mantiene finché non termina il suo burst.
- **preemptive (SRTF) (con prelazione):** Se nella ready queue arriva un nuovo processo il cui prossimo burst è minore del tempo rimanente per il processo attualmente in esecuzione, quest'ultimo viene prelazionato. Questo schema viene chiamato *SRTF: Shortest remaining time first*.

Esempio SJF senza prelazione:

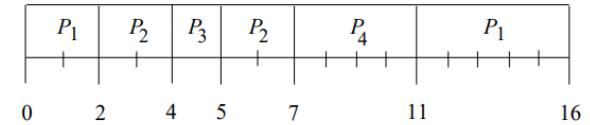
Processo	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



$$\text{Tempo di attesa medio} = (0 + 6 + 3 + 7)/4 = 4$$

Esempio SJF con prelazione (SRTF):

Processo	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

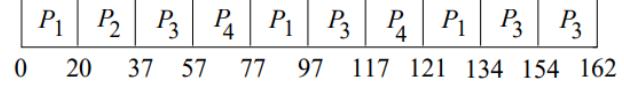


$$\text{Tempo di attesa medio} = (9 + 1 + 0 + 2)/4 = 3$$

RR: Round Robin: È un algoritmo con prelazione, specifico dei sistemi time-sharing. Molto simile al FCFS ma con prelazione quantizzata. Ogni processo riceve una piccola unità di tempo di CPU (*detto quanto*: 10-100ms). Dopo questo periodo, il processo viene prelazionato e rimesso in coda ready. Se ci sono n processi in coda ready, e il quanto è q , allora ogni processo riceve $1/n$ del tempo di CPU in periodi di durata massima q . Si ha che nessun processo attende più di $(n-1)q$. Tipicamente con scheduling round robin si ha un tempo di turnaround medio maggiore, ma minore tempo di risposta e maggiore overhead dovuto ai numerosi cambi di contesto.

Esempio RR con quanto = 20

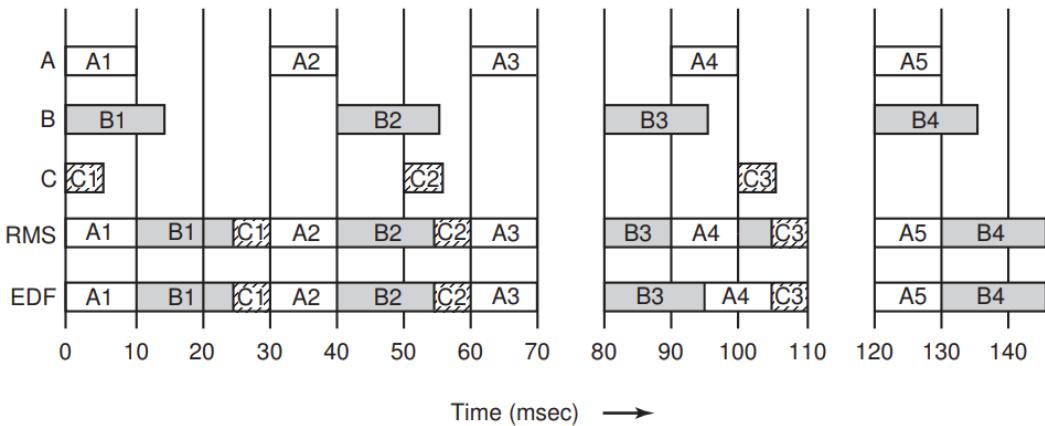
Processo	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24



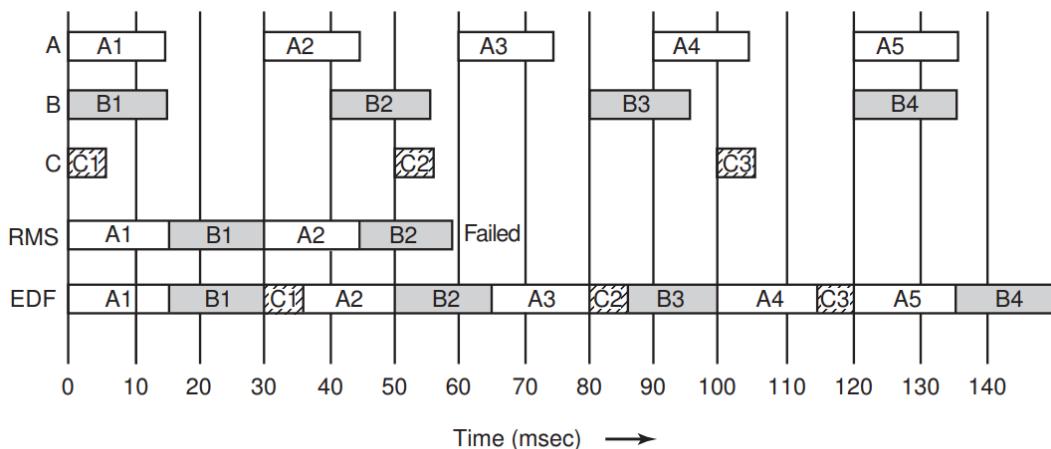
Considerazioni: con un quanto grande, degenera nell' FCFS, con un q troppo piccolo, si avrebbero troppi context switch, aumentando così l'overhead della CPU. Con algoritmo di scheduling Round Robin un eventuale processo impegnato in un'attesa attiva viene comunque prelazionato alla fine del suo quanto. In questo modo non è possibile che monopolizzi la CPU. Se invece l'algoritmo di scheduling è basato su priorità, allora è possibile (*in assenza di meccanismi di aging o trasferimento di priorità*) che ciò avvenga se il processo che lo esegue (*lo spin-lock, attesa attiva*) ha una priorità sufficientemente alta da non venire mai prelazionato e, per esempio, se l'evento atteso può essere generato soltanto da un processo a bassa priorità (*che non verrà quindi mai eseguito*), causerà un deadlock.

RMS: Rate Monotic Scheduling: È un algoritmo di scheduling con priorità tipicamente utilizzato in ambito real-time. RMS assegna a ciascun processo una priorità prefissata uguale alla frequenza con cui deve essere eseguito. Durante l'esecuzione lo scheduler esegue sempre il processo pronto a più altra priorità, prelazionando, se necessario, l'attuale processo in esecuzione.

EDF: Earliest Deadline First: In questo algoritmo, il sistema operativo mantiene una lista dei processi eseguibili rispetto alla loro scadenza temporale (ogni processo annuncia la sua presenza allo scheduler specificando la loro scadenza temporale). L'algoritmo esegue il primo processo della lista, cioè quello con scadenza temporale più vicina. Quando un nuovo processo è pronto, il sistema controlla se la sua scadenza precede quella del processo correntemente in esecuzione, in caso positivo quello in esecuzione viene prelazionato. A differenza di RMS è adatto a priorità dinamiche (in base a chi scade prima) ed è adatto anche per processi non periodici. Permette di raggiungere anche il 100% di utilizzo cpu.



Esempio di fallimento di RMS



Accenni di scheduling nei sistemi operativi odierni:

Scheduling in UNIX tradizionale: A code multiple, con approccio Round Robin. Numeri più grandi indicano priorità minore. Feedback negativo sul tempo della CPU impiegato; i processi invecchiano per prevenire la starvation.

Scheduling in UNIX moderno (SVR4): Utilizza il principio di separazione tra meccanismi e politiche. Il meccanismo principale implica 160 livelli di priorità (numero maggiore = priorità maggiore), ogni livello è gestito separatamente con politiche che possono differire una dall'altra. Per ogni classe di scheduling si può definire una politica diversa (intervallo delle priorità che definisce la classe, algoritmo per il calcolo della priorità, assegnazione dei quanti, migrazione dei processi da un livello all'altro etc...).

Scheduling in Linux: L'algoritmo di scheduling di Linux gira in tempo costante $O(1)$. Adatto per SMP (Symmetric Multi Processing). Scheduling con prelazione basato su priorità. Esistono due priorità: real time (0-99) e nice (100-140), valori bassi indicano priorità più alte. A priorità più basse corrispondono quanti più lunghi (a differenza di Unix e Windows). Ogni CPU mantiene una coda di esecuzione con due liste di task: attivi e scaduti. Un task è attivo se non ha terminato il suo quanto, altrimenti è scaduto. Le due liste sono ordinate secondo la priorità dei task. Quando l'array attivo si svuota i due array si scambiano di ruolo.

Scheduling in Windows: Un thread esegue lo scheduler quando viene eseguita una chiamata bloccante, si comunica con un oggetto e alla scadenza del quanto di thread. Inoltre lo scheduler si esegue in modo asincrono al completamento di un I/O e allo scadere di un timer. Lo scheduler sceglie sempre il thread a priorità maggiore, ovvero andando a guardare nella coda di priorità di maggiore priorità.

6. Cooperazione tra Processi

Si parla di cooperazione tra processi quando due o più processi possono modificare o essere modificati tra di loro. In questo caso i processi vengono detti cooperanti, altrimenti, vengono chiamati indipendenti, cioè che non possono essere modificati o modificare l'esecuzione di altri processi.

I principali vantaggi della cooperazione tra processi sono:

- La condivisione delle informazioni
- L'aumento della computazione: parallelismo
- Maggiore modularità
- Praticità implementativa e di utilizzo

Perché dei processi cooperino, è necessario che essi possano comunicare. In inglese, la comunicazione tra processi viene detta *inter-process communication (IPC)* e si riferisce a tutte quelle tecnologie software che consentono, appunto, a diversi processi (*o thread*) di comunicare, scambiandosi dati e informazioni tra di loro.

Quando si parla di comunicazione tra processi è importante considerare le seguenti problematiche:

- Come può un processo comunicare con altri processi?
- Come si può evitare accessi inconsistenti a risorse condivise?
- Come sequenzializzare gli accessi alle risorse?

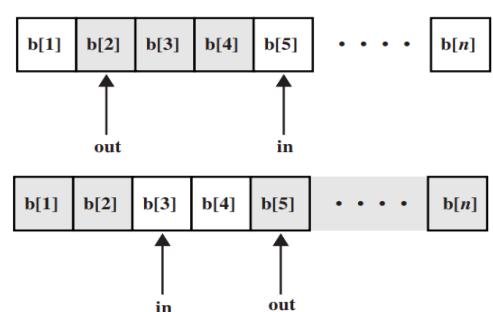
I due principali modelli di comunicazione tra processi sono:

- **memoria condivisa** (*utilizzabile se i processi risiedono nello stesso sistema*): i processi che vogliono scambiare dei dati li scrivono e leggono in un'area di memoria allocata e destinata a questo scopo per mezzo di normali operazioni di lettura/scrittura della memoria.
- **scambio di messaggi** (*più generale in quanto applicabile anche fra processi che risiedono su sistemi distinti, ovvero, in sistemi distribuiti connessi da un canale di comunicazione*): permette a due o più processi di scambiare dei dati su un canale (*realizzato solitamente mediante delle socket*) attraverso due primitive, ovvero, *send()* e *receive()*.

Mantenere la consistenza dei dati è importante e richiede meccanismi per assicurare che l'esecuzione di vari processi cooperanti su di essi sia ordinata e coerente.

Esempio: Problema del produttore-consumatore

Si considerino due processi detti “*produttore*” e “*consumatore*”. Il produttore produce delle informazioni che vengono utilizzate dal processo consumatore. Soluzione immediata: tra i due processi si pone un buffer di comunicazione di dimensione fissata, utilizzato per scambiare i dati, un processo scrive da un lato, e l'altro legge dall'altro.



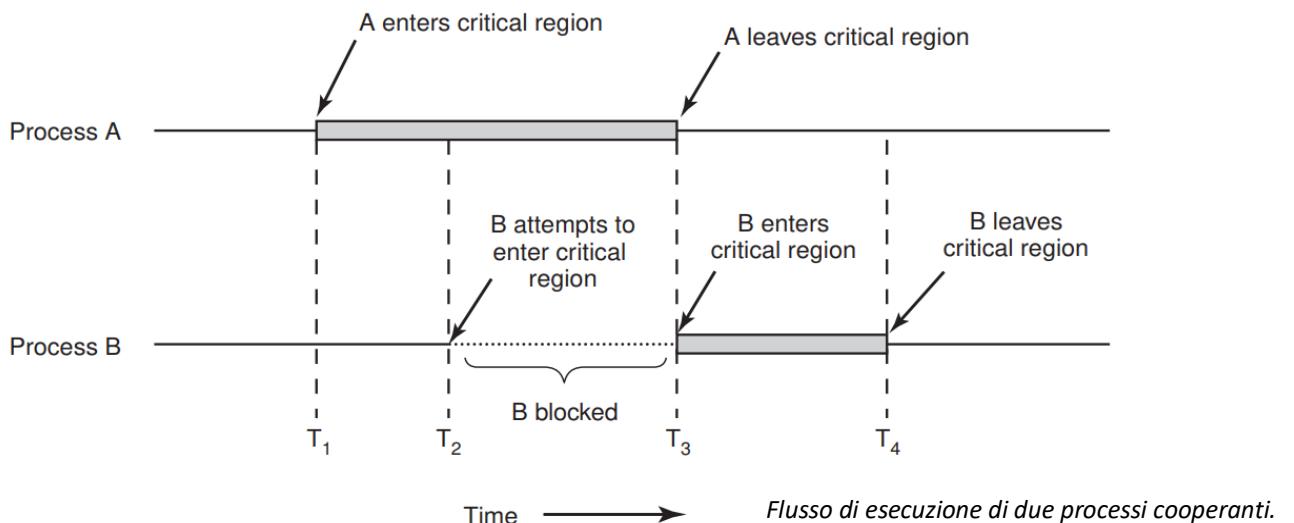
Alcuni fenomeni e concetti legati alla cooperazione tra processi:

- **Race condition (corsa critica):** Le race conditions sono quel fenomeno in cui più processi accedono concorrentemente agli stessi dati, e il risultato dipende dall'ordine di interleaving dei processi. Questo fenomeno è molto frequente nei sistemi operativi multitasking, sia per dati in user che in kernel space. Il verificarsi di questa situazione risulta essere di estremo pericolo perché può portare al malfunzionamento dei processi cooperanti, o addirittura dell'intero sistema se riguardano dati sensibili in kernel space. Sono situazioni difficili da individuare e da riprodurre perché non dipendono tanto dai dati ma da informazioni astratte quali le decisioni prese dallo scheduler, dal carico del sistema, dal numero e dallo stato dei processi, dall'utilizzo della memoria etc. etc.

- **Problema della sezione critica:** Quando sono presenti n processi che competono per utilizzare dei dati condivisi (*race condition*), viene detta *sezione critica* quel segmento di codice in cui essi accedono a questi dati. Per evitare il verificarsi di race condition, serve assicurare che quando un processo esegue la sua sezione critica, nessun altro processo possa entrare nella propria (*ovvero mentre un processo lavora sui dati condivisi, deve essere l'unico a poterlo fare*). Si noti che se lo scheduling della CPU è senza prelazione, le corse critiche non si possono verificare in quanto il processo correntemente in esecuzione non può essere costretto a rilasciare la CPU e quindi ad alternarsi con altri processi.

```
while (TRUE) {
    entry section
    sezione critica
    exit section
    sezione non critica
};
```

Dopo che il processo P passa la sua *entry section*, nessun altro deve poterlo fare finché P non supera la sua *exit section*.



➤ Condizioni per una soluzione al problema della sezione critica ed esempi di soluzione

Perché il problema della sezione critica sia risolto è importante che vengano rispettati i seguenti punti:

- **Mutua esclusione:** se il processo P sta eseguendo la sua sezione critica, allora nessun altro processo può eseguire la propria sezione critica.
- **Progresso:** nessun processo in esecuzione fuori dalla sua sezione critica può bloccare processi che desiderano entrare nella propria sezione critica.
- **Attesa limitata:** se un processo P ha richiesto di entrare nella propria sezione critica, allora il numero di volte che si concede agli altri processi di accedere alla propria sezione critica prima del processo P deve essere limitato. Un processo può subire una starvation all'entrata di una sezione critica se l'implementazione della sezione critica non soddisfa la condizione di attesa limitata.

NB: Perché le condizioni siano sufficienti si considera (sebbene ovvio) che ogni processo non venga eseguito ad una velocità nulla.

- **Soluzione Hardware: Controllo degli interrupt:** Il processo può disabilitare tutti gli interrupt hardware all'ingresso della sezione critica e ristabilirli all'uscita. Risulta essere una soluzione semplice per garantire la mutua esclusione, ma risulta pericolosa perché il processo potrebbe non riabilitare più gli interrupt, acquisendo pieno controllo della macchina. Non si estende molto bene a macchine multi-cpu (*a meno che non le blocchi tutte*). Risulta quindi adatto per brevi segmenti di codice affidabile (*ad esempio codice del kernel, che si suppone sia scritto correttamente*).

- **Soluzione Software:** Si suppone che i processi possano condividere alcune variabili dette di lock per sincronizzare le proprie azioni. Il codice del processo avrà una struttura come la seguente:

```
while (TRUE) {
    entry section
    sezione critica
    exit section
    sezione non critica
}
```

Lo schema sottostante indica un tentativo errato: lo scheduler può agire dopo il ciclo nel punto indicato.

```
while (TRUE) {
    ↓
    while (occupato ≠ 0);   occupato := 1;
    sezione critica
    occupato := 0;
    sezione non critica
};
```

Seguono una lista di alcune soluzioni software sviluppate nel corso dei decenni.

- **Algoritmo di Peterson:** È un esempio che soddisfa tutti i requisiti, e si basa su una combinazione di richiesta e accesso. È basato su *spinlock* (*si dice che un processo è in spin lock quando attende attivamente su una variabile*).

```
#define FALSE 0
#define TRUE 1
#define N      2           /* number of processes */

int turn;                  /* whose turn is it? */
int interested[N];         /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;               /* number of the other process */

    other = 1 - process;    /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;          /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

Spinlock: Uno spinlock è una tecnica di programmazione che utilizza un ciclo di attesa attiva (*busy waiting*) per implementare la routine di attesa di un lock. La tecnica consiste nel verificare periodicamente se il lock è stato sbloccato, effettuando un test che può aver luogo ad intervalli di tempo prestabiliti, oppure nel tempo più breve possibile consentito dal sistema. Un lock è un meccanismo di sincronizzazione per limitare l'accesso ad una risorsa condivisa in un ambiente multitasking ad un solo thread o ad un solo tipo di thread alla volta.

Inversione di priorità: Si verifica quando un processo a bassa priorità detiene una risorsa indispensabile ad un processo ad alta priorità pronto ad eseguire. Di conseguenza si può creare una situazione in cui un processo ad alta priorità deve attendere (passando in stato waiting) un processo a bassa priorità per proseguire e quest'ultimo non può andare in esecuzione in quanto esiste un terzo processo a priorità intermedia fra i due che ottiene tutto il tempo della CPU.

- **Algoritmo del fornaio:** È in grado di risolvere la sezione critica per n processi. Prima di entrare nella sezione critica, ogni processo riceve un numero. Chi ha il numero più basso entra nella sezione critica. Eventuali conflitti vengono risolti da un ordine statico: se ad esempio i processi P_i e P_j ricevono lo stesso numero: se $i < j$, allora P_i è servito per primo, altrimenti P_j è servito per primo. Lo schema di numerazione genera numeri in ordine crescente.
- **Istruzioni Test & Set:** Le istruzioni di *Test-and-Set-Lock* testano e modificano il contenuto di una parola atomicamente (*in un unico ciclo*). Risulta essere una soluzione corretta e semplice, comunque basata sullo spinlock (*busywait*).

```
function Test-and-Set (var target: boolean): boolean;
begin
    Test-and-Set := target;
    target := true;
end;
```

- **BusyWait:** Per busy wait (attesa attiva) si intende l'attendere il verificarsi di un certo evento "sprecando" tempo di CPU per controllare ciclicamente se sia avvenuto o meno (ad esempio controllando il valore di una variabile). È un meccanismo semplice da implementare, ma può portare a consumi inaccettabili di CPU, pertanto sarebbe da utilizzare solo in caso di attese molto brevi. Considerando quanto detto riguardo lo spinlock e riguardo i precedenti algoritmi (che ne fanno uso) risulta chiaro che le soluzioni basate su questo meccanismo portano ad un alto consumo della CPU. Inoltre si può verificare il problema dell'inversione di priorità: un processo a bassa priorità che blocca una risorsa può essere bloccato all'infinito da un processo ad alta priorità in busy wait sulla stessa risorsa. Risulta necessario quindi evitare il busy wait. L'idea migliore è di impostare un processo in *wait* quando un processo deve attendere un evento, e impostarlo in *ready* quando l'evento avviene. Servono però specifiche system-call o funzioni del kernel, come ad esempio: *sleep()* che autosospende un processo e *wakeup(P)* che risveglia il processo P (mettendolo dalla coda *wait* a *ready*).

- **Produttore consumatore con sleep e wakeup:** Risolve il problema dell'busy wait (*spinlock*) ma non risolve la corsa critica (sulla variabile *count*). Inoltre i segnali possono andare perduti con conseguenti deadlock, soluzione: salvare i segnali in attesa in un contatore/buffer.

```
#define N 100                                     /* number of slots in the buffer */
int count = 0;                                    /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                                /* repeat forever */
        item = produce_item();                     /* generate next item */
        if (count == N) sleep();                   /* if buffer is full, go to sleep */
        insert_item(item);                        /* put item in buffer */
        count = count + 1;                         /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);          /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                                /* repeat forever */
        if (count == 0) sleep();                   /* if buffer is empty, got to sleep */
        item = remove_item();                    /* take item out of buffer */
        count = count - 1;                        /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer);     /* was buffer full? */
        consume_item(item);                      /* print item */
    }
}
```

➤ Semafori e implementazione dei semafori

Il semaforo è uno strumento di sincronizzazione che si generalizza bene a tutti i casi dove è richiesta. Inventato da Dijkstra nel 1965, un semaforo è un dato astratto gestito dal sistema operativo usato appunto per sincronizzare l'accesso a una determinata risorsa in un contesto multitasking. A livello implementativo si tratta di una variabile intera con una sua interfaccia, che ci limita nell'interazione con essa. Risulta infatti possibile eseguire solo due operazioni:

- ***up(S)***: incrementa S di 1
- ***down(S)***: attendi finchè S è maggiore di 0, quindi decrementa S.

Normalmente, l'attesa è implementata spostando il processo in stato di wait, mentre la *up(S)* mette uno dei processi eventualmente in attesa nello stato di ready.

Esempio di gestione della sezione critica per n processi con semaforo: È presente la variabile ***mutex*** condivisa tra gli *n* processi. La variabile in questione è il semaforo ed è inizialmente di valore pari a 1.

```

while (TRUE) {
    down(mutex);
    sezione critica
    up(mutex);
    sezione non critica
}

```

Per quanto riguarda l'implementazione di questo meccanismo bisogna fare alcune considerazioni. Come si implementa l'attesa del processo che trova a 0 il semaforo quando esegue la chiamata *down*?

L'implementazione classica/standard utilizzava uno spinlock, risultava facile da implementare ma poco efficiente. L'alternativa di base è mettere in wait un processo che attende. L'implementazione quindi diventa la seguente:

```

type semaphore = record
    value: integer;
    L: list of process;
end;

```

Il semaforo viene visto come una variabile di tipo *record*. Assumiamo inoltre che il sistema operativo fornisca le due operazioni di base *sleep()* e *wakeup(P)* per implementare lo stato di wait di un processo. Ricordiamo:

- ***sleep()*** sospende il processo che la chiama rilasciando la CPU,
- ***wakeup(P)*** pone in stato ready il processo P che è stato precedentemente sospeso da una *sleep()*.

Le operazioni sui semafori vengono implementate in questo modo:

```

down(S):  $S.value := S.value - 1;$ 
    if  $S.value < 0$ 
        then begin
            aggiungi questo processo a  $S.L$ ;
            sleep();
        end;
up(S):    $S.value := S.value + 1;$ 
    if  $S.value \leq 0$ 
        then begin
            togli un processo  $P$  da  $S.L$ ;
            wakeup(P);
        end;

```

Le operazioni suddette sono corrette soltanto se non vengono interrotte da altri task. Supponiamo invece che due task si "accavallino" ed eseguano "quasi" contemporaneamente l'operazione *down()* su un semaforo che ha valore 1. Subito dopo che il primo task ha decrementato il semaforo da 1 a 0, il controllo passa ora al secondo task, che decremente il semaforo da 0 a -1 e quindi si pone in attesa. Allora a questo punto il controllo torna al primo task che, siccome il semaforo ha ora un valore negativo, si pone anche lui in attesa. Il risultato è che entrambi i task sono bloccati, mentre il semaforo a 1 avrebbe consentito a uno dei due task di procedere.

Una corretta implementazione si può ottenere disabilitando gli interrupt, e quindi i cambi di contesto, prima delle operazioni *down()* e *up(S)*, e riabilitandoli subito dopo. Tale implementazione ha il difetto di richiedere due istruzioni aggiuntive. Su alcuni processori esiste l'istruzione *Test-and-set* spiegata precedentemente, usando tale istruzione è possibile implementare le operazioni sui semafori in modo sicuro ed efficiente (*eseguendole in modo atomico*).

Intuitivamente, il significato delle operazioni è il seguente: Con l'inizializzazione si dichiarano quante unità di un tipo di risorsa sono disponibili. Con l'operazione *down()* ("Pazienta"), un task chiede di riservare un'unità della risorsa. Se non sono disponibili unità, il task viene messo in attesa per essere risvegliato solo quando gli sarà assegnata l'unità richiesta. Con l'operazione *up(S)* ("Vai!"), un task rilascia l'unità di cui non ha più bisogno e per la quale altri task potrebbero già essere in attesa.

➤ Mutex

Il termine mutex indica un procedimento di sincronizzazione fra processi o thread concorrenti, utilizzato per impedire ad essi che accedano contemporaneamente a una risorsa condivisa (*soggetto a corsa critica*). A livello implementativo possono essere visti come dei semafori con due soli possibili valori: *bloccato* o *non bloccato*. Una volta creato un oggetto di tipo *mutex* sono possibili solo due operazioni primitive: *mutex_lock* e *mutex_unlock*. Quando un processo o un thread vuole accedere a una risorsa si trova davanti due possibili scenari. Nel primo, il mutex risulta *unlocked*, quindi il processo procede a bloccarlo (riservarlo) con una *mutex_lock*, utilizza la risorsa, e quando ha finito fa una chiamata a *mutex_unlock*. Nel secondo caso, quando vi accede, il mutex risulta *locked* il che significa che la risorsa condivisa è attualmente utilizzata da un altro processo, il processo dovrà quindi aspettare che si liberi.

➤ Monitor

Un'altra possibile soluzione per garantire la mutua esclusione nell'ambito della sincronizzazione fra processi è quella di utilizzare i *monitor*. Un monitor è un dato astratto, formato da:

- una collezione di variabili che definiscono lo stato dell'istanza del monitor stesso,
- una serie di procedure di inizializzazione,
- una serie di procedure e funzioni che realizzano le operazioni e i meccanismi del monitor.

Le caratteristiche di base del monitor, che garantiscono la mutua esclusione sono il fatto che i processi non possono accedere direttamente alle sue variabili e che un solo processo alla volta può eseguire il suo codice. Un processo entra in un monitor invocando una delle sue procedure. All'interno del monitor può essere attivo un solo processo per volta, infatti, quando un processo chiama una procedura, la richiesta viene accodata e soddisfatta non appena il monitor è libero. Il monitor però non è solo una raccolta di funzioni, senza un rigido controllo del flusso da parte sua, la mutua esclusione non potrebbe essere garantita.

Variabili condizionali e controllo del flusso: Il monitor, possiede delle variabili condizionali, di tipo *condition*, e delle procedure:

- **wait(c)**: il processo che la esegue si blocca sulla condizione c.
- **signal(c)**: uno dei processi in attesa su c viene risvegliato.

Il successivo processo a prendere il controllo del monitor viene scelto dallo scheduler di sistema.

```

monitor example
integer i;
condition c;

procedure producer( );
.
.
.
end;

procedure consumer( );
.
.
.
end;
end monitor;
```

➤ Comunicazione con scambio di messaggi (shared nothing)

Lo scambio di messaggi è un metodo di comunicazione tra processi che non è basato sull'utilizzo di memoria condivisa con controllo d'accesso, ma è basato sullo scambio di messaggi tra i due o più processi.

Lo scambio di messaggi può essere essenzialmente di tre tipi:

- **comunicazione asincrona**: il mittente spedisce il messaggio e continua ad effettuare le proprie operazioni. I messaggi spediti ma non ancora consumati vengono automaticamente bufferizzati in una *mailbox* (*mantenuta in kernel o dalle librerie*). L'oggetto delle send e receive è la mailbox, la prima si blocca se la mailbox è piena, la seconda si blocca se è vuota.
- **comunicazione sincrona**: il mittente spedisce il messaggio ed attende sino a quando il ricevente non ha ricevuto il messaggio, elaborato la risposta ed inviata al mittente. I messaggi sono quindi inviati direttamente al processo destinazione, l'oggetto delle send e receive sono i processi stessi.
- **remote invocation**: il mittente aspetta che il ricevente sia pronto per ricevere, e solo dopo che il ricevente si è dato disponibile per ricevere, il mittente invia il messaggio.

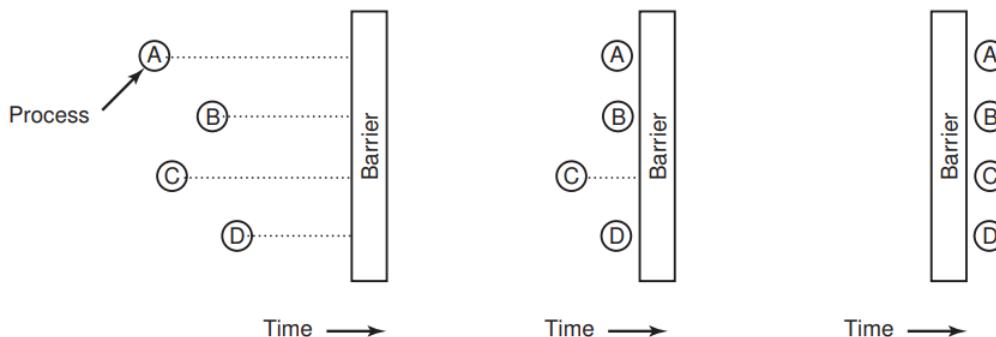
Le varie implementazioni si basano, tipicamente, su due funzioni primitive (*tipicamente sys-call o funzioni di libreria*).

- **send(destinazione, messaggio)**: spedisce un messaggio ad una certa destinazione, non bloccante.
- **receive(sorgente, messaggio)**: riceve un messaggio da una sorgente, solitamente bloccante.

Questo approccio porta però a delle problematiche: I canali utilizzati potrebbero essere inaffidabili e intercettati (*ad esempio le reti*). Bisogna implementare appositi protocolli di autenticazione e controllo degli errori, inoltre, questo approccio, se prende luogo nella stessa macchina, risulta meno efficiente dell'utilizzo di approcci con memoria condivisa.

➤ Barriere

Un altro meccanismo di sincronizzazione per gruppi di processi sono le *barriere*, utilizzato soprattutto in ambito del calcolo parallelo con memoria condivisa. Ogni processo alla fine della sua computazione, chiama la funzione *barrier()* e si sospende. Quando tutti i processi hanno raggiunto la barriera, la superano tutti insieme (sbloccandosi).



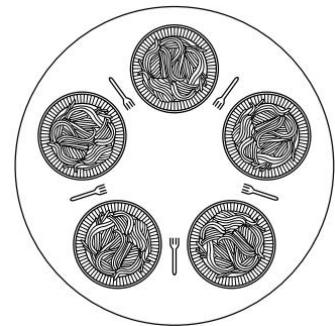
➤ I Grandi classici: esempi di problemi legati alla sincronizzazione tra processi

Esiste una serie di problemi classici nella programmazione concorrente. Essi possono venire utilizzati per dimostrare l'efficienza di determinate teorie od algoritmi e forniscono una base comune per poter effettuare dei paragoni. Tra quelli più famosi si elencano:

- *Produttore consumatore a buffer limitato*
- *I filosofi a cena*
- *Lettori-Scrittori*
- *Il barbiere che dorme*

Segue la spiegazione e illustrazione dei seguenti problemi:

- **I filosofi a cena:** formulato da Edsger Dijkstra come *dining philosophers problem*. Alcuni filosofi (5 nel testo originale) sono seduti a tavola di fronte al loro piatto e a due forchette. I filosofi alternano momenti durante i quali meditare e momenti durante i quali mangiare. Per mangiare devono prendere le due forchette accanto al loro piatto e mangiare mentre durante la meditazione devono tenere le forchette sul tavolo. Risulta evidente che il numero di forchette impedisce a tutti i filosofi di mangiare contemporaneamente quindi una corretta programmazione concorrente deve essere in grado di far mangiare alternativamente tutti i filosofi evitando che qualcuno in particolare soffra la fame ed evitando che si verifichino stalli in fase di "acquisizione delle forchette". Ricapitolando, bisogna programmare i filosofi in modo da garantire l'assenza di deadlock e assenza di starvation (un filosofo che vuole mangiare prima o poi deve poterlo fare).
 - Mentre pensa, un filosofo non interagisce con nessuno
 - Quando gli viene fame, cerca di prendere le forchette più vicine, una alla volta.
 - Quando ha due bacchette, un filosofo mangia senza fermarsi.
 - Terminato il pasto, lascia le bacchette e torna a meditare.



Soluzioni possibili:

1. Introdurre un semaforo *mutex* per proteggere la sezione critica (*ovvero il processo che fa prendere in modo sequenziale le due forchette a un filosofo*). Funziona, ma solo un filosofo può mangiare, quando in teoria possono farlo $n/2$ al massimo (*contemporaneamente*).
2. Tenere traccia dell'intenzione di un filosofo di voler mangiare. Ad ogni filosofo verrà quindi assegnato uno stato tra: *thinking,hungry,eating*, mantenuto in un vettore *state*. Un filosofo può entrare nello stato di *eating* solo se è *hungry* e i due vicini sono nello stato *thinking*. Soluzione che permette il massimo parallelismo.

- **Lettori-Scrittori:** Su dei dati agiscono processi *scrittori*, che ne modificano il contenuto, e processi *lettori* che ne recuperano il contenuto. Una corretta programmazione concorrente prevede l'accesso di un solo scrittore in fase di modifica (*e di nessun lettore onde evitare inconsistenza*) mentre l'accesso di tanti lettori è possibile a patto che non ci siano scrittori attivi contemporaneamente. Quindi:

- Due o più lettori possono accedere contemporaneamente ai dati se nessun scrittore sta scrivendo.
 - Ogni scrittore deve accedere ai dati in modo esclusivo, uno alla volta.

Soluzione: Implementazione con un semaforo. Si tiene conto dei lettori in una variabile condivisa, finchè ci sono lettori in azione, gli scrittori non possono accedere. Sostanzialmente o lavorano gli scrittori, o i lettori. Una soluzione del genere da priorità ai lettori.

- **Il barbiere che dorme:** un barbiere possiede un negozio con una sola sedia da lavoro e un certo numero limitato di posti per attendere. Se non ci sono clienti il barbiere dorme. All'arrivo del primo cliente il barbiere si sveglia ed inizia a servirlo. Se dovessero sopraggiungere clienti durante il periodo di attività del barbiere, essi si mettono in attesa sui posti disponibili. Una corretta programmazione concorrente deve far "*dormire*" il barbiere in assenza di clienti, attivare il barbiere sul primo cliente al suo arrivo e mettere in coda tutti i successivi clienti tenendoli inattivi. Ricapitolando:

- Quando non ci sono clienti, il barbiere dorme sulla sedia.
 - Quando arriva un cliente, questo sveglia il barbiere se sta dormendo.
 - Se la sedia è libera e ci sono clienti, il barbiere fa sedere un cliente e lo serve.
 - Se un cliente arriva e il barbiere sta già servendo un cliente, si siede in una sedia d'attesa se ce ne sono libere, altrimenti va via.

7. Deadlock

Con *deadlock* si indica quella situazione di stallo che si verifica quando due o più processi si bloccano a vicenda, aspettando che uno esegua una certa azione che serve all'altro e viceversa (ad esempio rilasciare il controllo di un file, una porta di input/output, il risultato di una funzione etc.). Un esempio è rappresentato da due persone che vogliono disegnare. Per disegnare hanno a disposizione solo una riga e una matita. Per disegnare hanno bisogno di entrambe. Potendo prendere un solo oggetto per volta, se uno prende la matita e l'altro prende la riga, e se entrambi aspettano che l'altro gli dia l'oggetto che ha in mano, i due generano un deadlock. Un deadlock è una situazione inrisolvibile (o meglio: non facilmente risolvibile), possono essere però attuate delle tecniche di prevenzione, messe in atto dal sistema operativo. Innanzitutto è importante chiarire alcuni concetti sulle risorse: quando si parla di risorse condivise e possibilmente richieste da più processi possiamo categorizzarle in *prerilasciabili* e *non prerilasciabili*.

- **prerilasciabili:** possono venire sottratte al processo che le possiede, senza effetti dannosi (*ad esempio la ram*).
- **non prerilasciabili:** non possono essere cedute dal processo che le possiede, pena il fallimento dell'esecuzione.

I deadlock intuitivamente si possono verificare con le risorse non prerilasciabili.

Quattro condizioni necessarie perché si possa verificare un deadlock

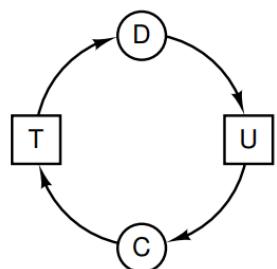
- **Mutua esclusione:** ogni risorsa è assegnata ad un solo processo, oppure è disponibile.
- **Hold & Wait:** i processi che hanno richiesto ed ottenuto delle risorse ne possono richiedere altre.
- **Mancanza di prerilascio:** le risorse che un processo detiene possono essere rilasciate dal processo solo volontariamente.
- **Catena di attesa circolare di processi:** esiste un sottoinsieme di processi tali che ciascuno di essi è in attesa di una risorsa assegnata al processo successivo.

NB: Se anche solo una di queste condizioni manca, il deadlock **NON** può verificarsi. Per ciascuna di queste condizioni è possibile adottare o meno alcune contromisure. [Coffman 1971]

Fenomeno della starvation: Nonostante il problema della starvation sia strettamente correlato a quello del deadlock, si tratta di due cose differenti. Infatti, con il termine starvation, si indica un'assenza di progresso per cui un programma in esecuzione non riesce ad ottenere una risorsa o un servizio (*a causa ad esempio della politica di allocazione della risorsa/servizio in questione adottata nel sistema*), nonostante non venga mai bloccato. Invece con il termine deadlock come precedentemente spiegato, si intende una situazione in cui un insieme di processi è bloccato in quanto ogni processo è in attesa di un evento che soltanto un altro processo dell'insieme può provocare.

Grafo di allocazione risorse

È possibile modellare uno schema rappresentante l'allocazione delle risorse, e le varie dipendenze tra processi mediante un grafo orientato. Un insieme di vertici V e un insieme di archi E .



- V è partizionato in P ed R , rispettivamente l'insieme di tutti i processi di sistema e l'insieme di tutte le risorse di sistema.
- Ogni arco indica o una richiesta (da un p ad un r) o una richiesta (da un r ad un p). (con p elemento di P e r elemento di R).
- Uno stallo, è un ciclo nel grafo di allocazione delle risorse

Si ha quindi che se il grafo non contiene cicli, non è possibile il verificarsi di un deadlock. Viceversa, se sono presenti cicli allora il deadlock può verificarsi. Questi grafi sono uno strumento per verificare se una sequenza di allocazione porta ad un deadlock: Il sistema operativo ha a disposizione molte sequenze di scheduling dei processi, per ciascuna di esse può eseguire una simulazione sul grafo e controllare se porta ad un deadlock; questo meccanismo gli permette di scegliere sequenze di allocazione teoricamente più sicure. Risulta comunque un'operazione costosa da eseguire.

Gestione dei Deadlock

In generale, è possibile seguire uno dei seguenti quattro approcci per gestire un deadlock:

- Ignorare il problema (*pericoloso*)
- Permettere che il sistema entri in un deadlock, riconoscerlo e risolverlo (*costoso*)
- Cercare di evitare dinamicamente il verificarsi di un deadlock (*difficile e costoso*)
- Assicurare che il sistema non possa mai entrare in uno stato di deadlock, negando una delle quattro condizioni necessarie.

Segue un approfondimento dei quattro approcci:

- **Primo approccio: Ignorare il problema**

Il principio alla base di questo approccio è che assicurare l'assenza di deadlock impone dei costi molto alti, che possono essere sopportati in alcuni contesti, ma insostenibili in altri. Viene quindi considerato il rapporto costo/benefici nel caso che la probabilità che accada un deadlock sia sufficientemente bassa. Si preferisce che l'utente debba riavviare il sistema in situazioni come queste (*molto rare*) piuttosto che rallentare di molto il sistema per prevenirle. (Unix e Windows, e molti altri sistemi addottano questo sistema).

- **Secondo approccio: Identificazione e risoluzione dei deadlock**

Si lascia che il sistema possa entrare in un deadlock, lo si riconosce con opportuni algoritmi di identificazione e si attua una politica di risoluzione del deadlock (*recovery*). Segue una lista di alcuni metodi di identificazione e risoluzione.

- **Una risorsa per classe:** Esiste una sola istanza per ogni classe, si mantiene e controlla un grafo di allocazione delle risorse per cercare eventuali cicli che porterebbero al verificarsi di un deadlock.
- **Prerilascio:** A volte basta sottrarre una risorsa allocata (se di tipo prerilasciabile) ad uno dei processi in deadlock per permettere agli altri di continuare con l'esecuzione.
- **Rollback:** Vengono inseriti nei programmi dei check-point, in cui tutto lo stato dei processi viene salvato su un file. Quando si scopre un deadlock si conoscono le risorse ed i processi coinvolti. Uno o più processi coinvolti vengono riportati ad uno dei checkpoint precedenti, con conseguente rilascio delle risorse allocate da quel punto in poi. Si perde il lavoro fatto da quel check-point al punto di deadlock.
- **Terminazione:** A volte basta terminare, se possibile, uno dei processi coinvolti nel deadlock per far riprendere la computazione.

Va considerato che gli algoritmi di identificazione dei deadlock sono costosi e molto spesso è difficile decidere quando è opportuno farli agire. A grandi linee è possibile richiamare l'algoritmo ad ogni richiesta di risorse o chiamarlo ad un intervallo di tempo prestabilito.

- **Terzo approccio: Evitare dinamicamente i deadlock**

Questo approccio si basa sull'idea, che a patto di conoscere a priori alcune informazioni, è possibile decidere velocemente se assegnare una risorsa o meno, garantendo che questa decisione non possa introdurre un possibile deadlock. Il modello più semplice ed utile richiede che ogni processo dichiari fin dall'inizio il numero massimo di risorse di ogni tipologia di cui avrà bisogno nel corso della computazione, così che il sistema operativo possa fare le sue valutazioni. L'algoritmo (tipicamente chiamato deadlock-avoidance) esamina dinamicamente le risorse per assicurare che non ci siano mai code circolari / stalli.

Quando un processo richiede una risorsa, si deve decidere se l'allocazione lascia il sistema in uno stato sicuro. Lo stato è considerato sicuro se esiste una sequenza sicura per tutti i processi. Quindi, intuitivamente, se il sistema è in uno stato sicuro, non si possono verificare deadlock, viceversa potrebbero verificarsi se non si trova in uno stato sicuro.

Esempio di deadlock-avoidance: Algoritmo del banchiere (Dijkstra 1965): Controlla se una richiesta può portare ad uno stato non sicuro; in tal caso, la richiesta non è accettata. Ad ogni richiesta, l'algoritmo controlla se le risorse rimanenti sono sufficienti per soddisfare la massima richiesta di almeno un processo; in tal caso l'allocazione viene accordata, altrimenti viene negata. Funziona sia con istanze multiple che con risorse multiple. Ogni processo deve dichiarare a

priori l'uso massimo di ogni risorsa, inoltre quando richiede una risorsa, può essere messo in attesa. Quando un processo ottiene tutte le risorse che vuole, deve restituirle in un tempo finito.

- **Quarto approccio**

Il quarto approccio, il più drastico ma il più sicuro si basa sul fatto che negando una delle quattro condizioni necessarie al verificarsi di un deadlock, quest'ultimo non possa più verificarsi.

- **Approccio combinato: Blocco a due fasi (two-phase locking)**

I tre approcci di gestione non sono esclusivi, possono essere combinati: rilevamento, prevenzione e avoidance. Si può così scegliere l'approccio ottimale per ogni classe di risorse del sistema.

Blocco a due fasi: Protocollo in due passi, molto usato nei database: Prima il processo prova ad allocare tutte le risorse di cui ha bisogno per la transazione e se non ha successo, rilascia tutte le risorse e riprova. Se ha successo, completa la transazione usando le risorse.

➤ **Corollario pt.1:**

1. ***Un processo può subire una starvation all'entrata di una sezione critica se l'implementazione della sezione critica non soddisfa la condizione di attesa limitata.***

Vero, infatti la condizione di progresso afferma che “nessun processo che voglia entrare nella prossima sezione critica può essere bloccato da processi in esecuzione fuori dalle proprie sezioni critiche”.

2. ***Attese attive indefinite si possono verificare in un SO che utilizza uno scheduling basato su priorità, ma non possono verificarsi in un SO che utilizza uno scheduling Round Robin.***

Per attesa attiva indefinita si attende uno “spin-lock” indefinito, ovvero, saturare l'utilizzo della CPU con un ciclo busy-wait che non termina. Se l'algoritmo di scheduling è RR, allora un eventuale processo che fosse impegnato in un'attesa attiva di questo tipo verrebbe comunque prelazionato alla fine del suo quanto. In questo modo non sarebbe possibile monopolizzare la CPU. Se invece l'algoritmo di scheduling è basato sulla priorità, allora è possibile (in assenza di meccanismi di aging o trasferimento di priorità) che ciò avvenga se il processo che esegue lo spin-lock ha una priorità sufficientemente alta da non venire mai prelazionato e, per esempio, l'evento atteso può essere generato soltanto da un processo a bassa priorità (che non verrà quindi mai eseguito).

3. ***In un sistema lettore-scrittori che favorisce gli scrittori, alcuni processi lettori che vogliono leggere i dati condivisi non possono bloccarsi anche quando altri processi lettori stanno leggendo i dati condivisi.***

Vero, l'accesso concorrente dei dati in lettura è consentito, dato che non produce cambiamenti nei dati stessi e quindi non può essere fonte di race condition.

4. ***Un deadlock non può verificarsi nel problema dei filosofi a cena se un filosofo può mangiare solo con una forchetta.***

Vero, ci sono tante forchette quanti i filosofi e quindi non c'è la possibilità che si verifichi un deadlock nel caso sia sufficiente una singola forchetta per mangiare.

5. ***Si descriva il funzionamento dell'istruzione assembler Test-and-Set-Lock TSL RX , LOCK.***

L'istruzione TSL RX , LOCK copia il contenuto della locazione di memoria LOCK nel registro RX ed imposta con un valore diverso da zero. Il tutto viene compiuto atomicamente, bloccando il bus della memoria.

6. Si diano esempi di algoritmi di scheduling della CPU che privilegiano i processi I/O-bound.

In generale una classe di algoritmi di scheduling della CPU che privilegiano i processi I/O bound è costituita dagli scheduling con code multiple e con retroazione (feedback) in cui la coda a maggior priorità viene riservata ai processi interattivi, mentre quelli CPU-bound vengono posizionati nelle code a minor priorità. Fra i sistemi operativi di uso comune l'algoritmo di scheduling di è un esempio di algoritmo che favorisce i processi interattivi (I/O-bound) dato che assegna un bonus di priorità dinamica a questi ultimi. Anche lo scheduling di Windows favorisce i processi I/O-bound dato che sceglie sempre dalla coda a priorità maggiore, ma la priorità di un thread utente può essere temporaneamente maggiore di quella base (per effetto delle "spinte") per thread che attendono dati di I/O (spinte fino a +8) e per dare maggiore reattività a processi interattivi (+2).

7. Quali delle seguenti transizioni di stato per un processo possono causare la transizione di stato waiting → ready per uno o più degli altri processi? (a) Un processo avvia un'operazione di I/O e passa nello stato waiting. (b) Un processo termina. (c) Un processo effettua la richiesta per una risorsa e passa nello stato waiting. (d) Un processo invia un messaggio. (e) Un processo effettua una transizione di stato waiting → waiting swapped

(a) No, non influenza il passaggio da waiting a ready di un altro processo.(b) Sì se, per esempio, si tratta della terminazione di un processo figlio di cui il padre era in attesa o se un altro processo era in attesa di una risorsa detenuta dal processo terminato. (c) No, non influenza il passaggio da waiting a ready di un altro processo. (d) Sì: un altro processo poteva essere in attesa di ricevere un messaggio. (e) No, lo swapping su disco di un processo in stato waiting non può influenzare il passaggio da waiting a ready di un altro processo.

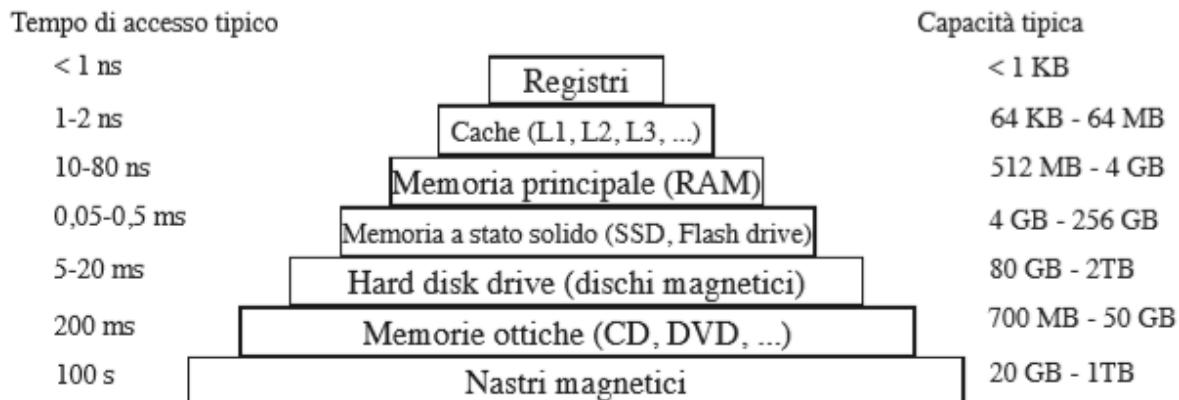
8. I monitor ed i semafori sono costrutti applicabili anche a sistemi distribuiti?

No, monitor e semafori necessitano di una forma di memoria condivisa e quindi non sono costrutti applicabili a sistemi distribuiti. Per questi ultimi si può utilizzare il modello dello scambio di messaggi. Tali funzioni non necessitano di nessuna forma di condivisione di memoria e quindi sono applicabili anche per mettere in comunicazione dei sistemi distribuiti.

8. Memory Management

La memoria, in informatica, è un elemento di un calcolatore deputato alla memorizzazione dei dati, la cui implementazione fisica dà vita ai vari supporti di memorizzazione esistenti. La memorizzazione di informazioni in memoria e il loro successivo recupero è una funzione fondamentale nel processo di elaborazione di dati. Nell'architettura di Von Neumann la memoria contiene sia i dati su cui/con cui la computazione opera, sia i programmi che istruiscono il processore riguardo quali operazioni effettuare.

➤ Gerarchia delle memorie:



Memoria Principale: chiamata anche memoria centrale, contiene dati ed istruzioni prelevati dalla memoria di massa in attesa che questi siano a loro volta prelevati ed elaborati dal microprocessore. Risorsa importante e limitata da cui dipendono le prestazioni in termini di tempo di un calcolatore. La memoria principale tipicamente è memoria RAM o cache. I processi memorizzati si “espandono” fino a riempire la memoria disponibile. Memoria illimitata, infinitamente veloce ed economica non esiste.

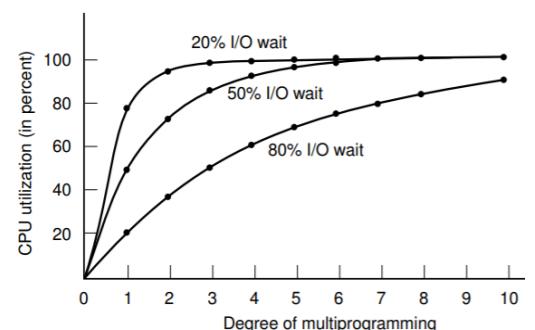
Il Sistema operativo deve occuparsi della gestione della memoria principale, al fine di garantire alcuni requisiti:

- *Organizzazione logica:* offre una visione astratta della gerarchia della memoria, permettendo di allocare e deallocare memoria ai processi su richiesta, nascondendone i dettagli implementativi.
- *Organizzazione fisica:* tener conto a chi è allocato cosa, ed effettuare gli scambi di dati con il disco.
- *Rilocazione:* predisporre di un meccanismo che consente di mettere in corrispondenza indirizzi logici con quelli fisici.
- *Protezione:* tra i processi, e per il sistema operativo.
- *Condivisione:* aumentare l'efficienza condividendo parti di dati (tipicamente di sola lettura) tra processi.

➤ Monoprogrammazione vs Multiprogrammazione.

Nella *monoprogrammazione* un solo processo viene eseguito per volta, la memoria è in parte allocata al S.O. e la parte rimanente al processo in esecuzione. La CPU non viene sfruttata al 100% in quanto nelle fasi di I/O con la memoria, quest'ultima è in attesa, e non produce computazione utile.

Nella *multiprogrammazione* più processi vengono eseguiti per volta (in realtà il S.O. esegue un po' per volta i vari processi; in una singola CPU un solo processo verrà eseguito realmente in un unico istante di tempo). Più parti di memoria sono quindi allocate a più processi, oltre che al sistema operativo. Nella multiprogrammazione è presente una coda di input, dove sono memorizzati i programmi in attesa di essere portati dal disco alla memoria primaria ed essere quindi eseguiti. La selezione è gestita dallo scheduler a lungo termine. Si ha come risultato un maggiore utilizzo della CPU.



Alcuni concetti:

Caricamento dinamico: Consiste nel caricare in memoria solo le parti di programma utili in un determinato istante di esecuzione. Inutile caricare parti di codice che vengono eseguite di rado. Si ha un migliore utilizzo della memoria.

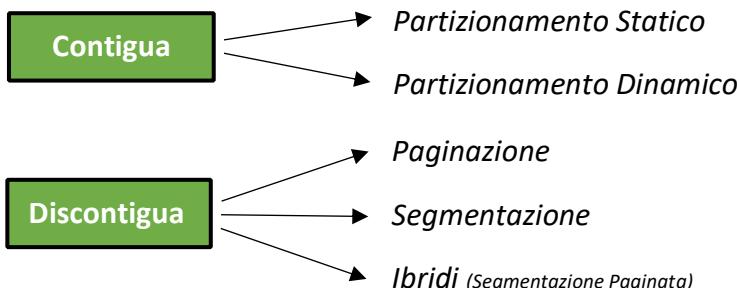
Indirizzo di memoria logico e fisico: La CPU genera indirizzi di memoria logici, che poi corrispondono realmente a indirizzi di memoria fisici (che possono coincidere nel caso di binding a compile e load time) trattati dalla MMU (Memory Management Unit). L'indirizzo logico è un riferimento che la CPU fa a una determinata locazione di memoria. L'indirizzo fisico è l'effettivo indirizzo a cui l'indirizzo logico fa riferimento.

Binding degli indirizzi: Con binding degli indirizzi si intende il processo tramite cui viene effettuato il collegamento fra un'entità di un software ed il suo corrispettivo valore localizzato in un preciso indirizzo di memoria fisico. L'associazione di istruzioni e dati ai corrispettivi indirizzi di memoria può avvenire principalmente in tre modi:

- Compile time: Se le locazioni di memoria sono note a priori si può produrre del codice assoluto.
- Load time: Il compilatore genera codice rilocabile la cui posizione in memoria viene decisa al momento del caricamento.
- Execution time: L'associazione viene effettuata durante l'esecuzione.

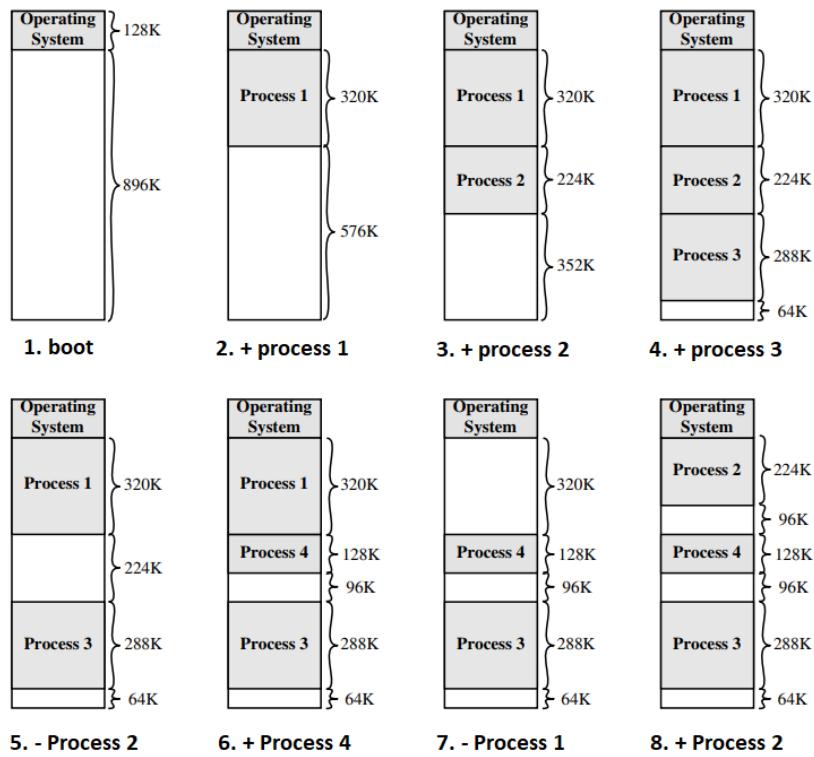
MMU (Memory Management Unit): Componente Hardware che gestisce le richieste di accesso generate dalla CPU. Traduce/Associa al run-time gli indirizzi logici a quelli fisici.

➤ **Allocazione contigua vs discontigua:** Esistono più approcci di allocazione della memoria:



Allocazione contigua: L'allocazione della memoria di tipo contiguo può essere statica o dinamica. Nell'allocazione contigua, la memoria è divisa in almeno due partizioni, una per il S.O. e la seconda per i processi utente. Ogni processo è contenuto in una sola partizione (che è contigua) di memoria.

- **Partizionamento Statico:** Nel caso di Allocazione contigua a partizionamento statico si ha che la memoria disponibile ai processi utente è divisa in partizioni fisse (che possono essere uguali o no) definite in fase di boot dal S.O. All'arrivo di un processo, viene scelta la partizione di memoria più indicata (se presente) e gli viene allocata. Questo approccio porta a frammentazione interna: La memoria allocata ad un processo è superiore a quella necessaria, la parte rimanente di quella partizione è sprecata!
- **Partizionamento Dinamico:** Nel caso di Allocazione contigua a partizionamento dinamico le partizioni di memoria vengono create a runtime. Le partizioni libere di memoria, spesso sparpagliate lungo la memoria vengono chiamate "holes/buchi". All'arrivo di un processo gli viene allocata una partizione di memoria all'interno di un "buco" sufficientemente grande. Questo approccio non causa frammentazione interna ma frammentazione esterna, ovvero che è possibile il verificarsi della seguente situazione: la memoria libera è sufficiente per un nuovo processo ma non è contigua. Il problema viene risolto con la compattazione: riordinamento della memoria, consiste nel compattare tutti i buchi in un solo unico buco.



Simulazione di esecuzione di allocazione e deallocazione di memoria
con allocazione contigua con partizionamento dinamico con approccio first-fit.

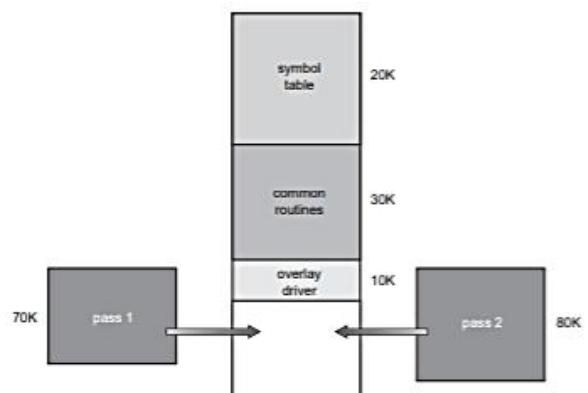
Approcci di allocazione nel partizionamento dinamico (alloc. contigua):

- **First-fit:** Alloca il primo buco sufficientemente grande.
- **Next-fit:** Alloca il primo buco sufficientemente grande a partire dall'ultimo usato.
- **Best-fit:** Alloca il più piccolo buco sufficientemente grande. Deve scandire l'intera lista (a meno che non sia ordinata). Produce il più piccolo buco di scarto.
- **Worst-fit:** Alloca il più grande buco sufficientemente grande. Deve scandire l'intera lista (a meno che non sia ordinata). Produce il più grande buco di scarto.

In generale, gli algoritmi migliori sono il *first-fit* e il *next-fit*. *Best-fit* tende a frammentare molto. *Worst-fit* è più lento.

Swapping: Un processo in esecuzione può essere temporaneamente rimosso dalla memoria e swappato (spostato) in una memoria secondaria (swap area / backing store) per fare spazio ad un altro processo, in seguito viene ri-swappato in memoria quando dovrà riprendere l'esecuzione. Lo swapping è gestito dallo scheduler di medio termine. Un processo per essere swappato deve essere "inattivo". Attualmente lo swapping standard non viene impiegato in quanto troppo costoso in termini di tempo. Questo processo è gestito dallo scheduler di medio termine. Allo swap-in il processo deve essere ricaricato esattamente nelle stesse regioni di memoria, a meno che non ci sia un binding dinamico degli indirizzi.

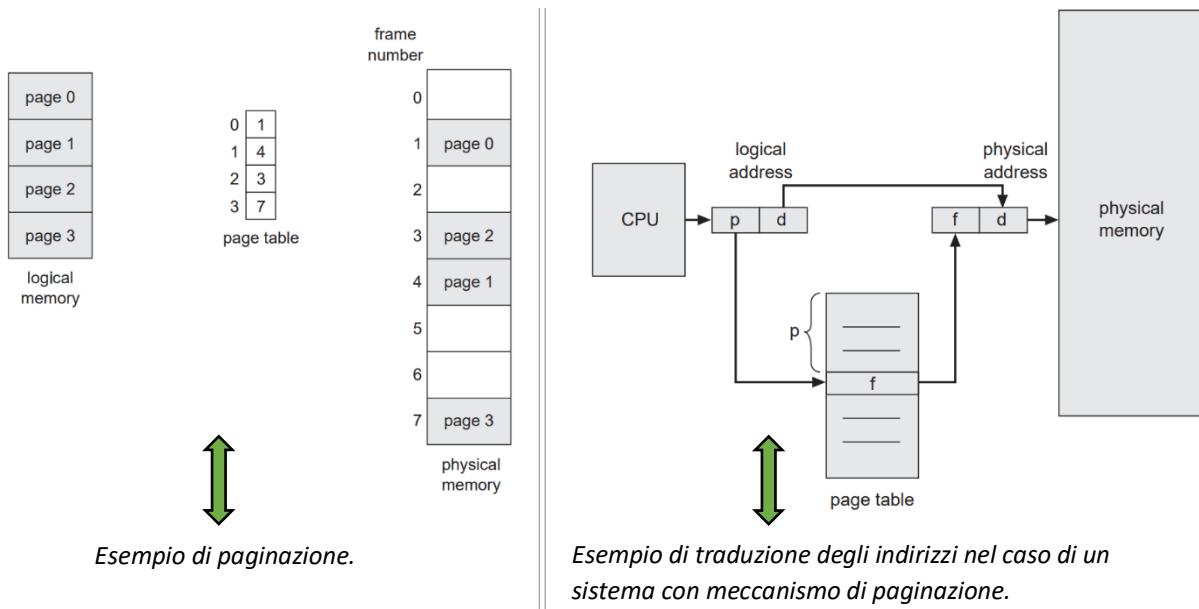
Overlay: Consiste nel mantenere in memoria solo le istruzioni e dati che servono in un determinato istante, utilizzato quando un processo è più grande della memoria allocatagli.



Allocazione NON contigua:

- **Paginazione:** Permette di allocare memoria fisica non contigua ad un processo.

Consiste nel suddividere la memoria in frame (pagine fisiche) ovvero blocchi di dimensione prestabilite (tipicamente potenze di 2 comprese tra 512byte e 16Mbyte). La memoria logica viene divisa in pagine della stessa dimensione. Il S.O. tiene traccia di tutti i frame liberi e non. Per eseguire un programma di dimensione n pagine serviranno quindi n frame liberi. È presente una page table, utilizzata per tradurre gli indirizzi logici in indirizzi fisici. Questa tecnica permette di non avere frammentazione esterna ma presenta solo una insignificante frammentazione interna quando l'ultima pagina non è completamente "riempita" (nel caso in cui un processo p non abbia dimensione multipla della dim. dei frame).

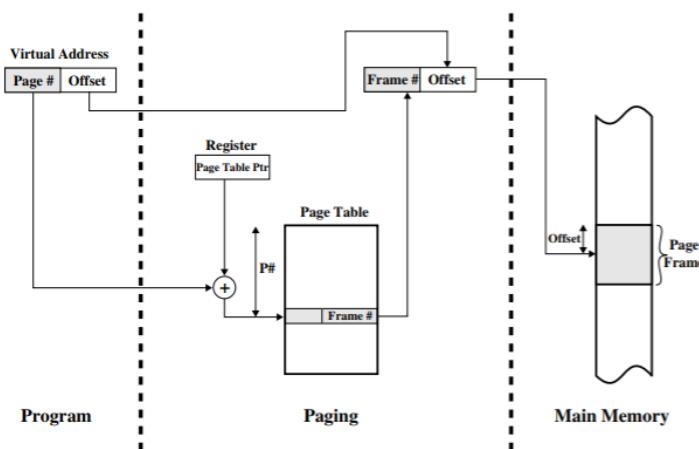


L'indirizzo logico generato dalla CPU consiste in una coppia (p,d), dove: (vedi schema sopra a destra)

< Frame#, Offset >

- *numero di pagina p*: usato come indice nella page table che contiene il numero del frame contenente la pagina p .
- *offset di pagina d*: combinato con il numero di frame fornisce l'indirizzo fisico da inviare alla memoria.

La paginazione permette di condividere parti di memoria: pagine in cui sono memorizzate parti di codice read-only possono essere condivise tra processi, e quindi più indirizzi logici possono far riferimento ad un unico indirizzo fisico. Con la paginazione si può introdurre il concetto di protezione della memoria: Ad ogni frame/pagina viene associato un bit di protezione di nome *Valid-bit* che indica se la pagina associata è nello spazio logico del processo che ne richiede l'accesso oppure no.



Implementazione della page table

La page table viene tenuta in memoria principale (idealmente dovrebbe stare in MMU su memorie molto veloci):

Sono presenti due registri importanti:

- *Page-table base register (PTBR)* punta all'inizio della page table
- *Page-table length register (PTLR)* indica il numero di entry della page table

Salvare la page table in memoria consuma comunque una grande quantità di spazio, inoltre, ogni accesso richiede 2 accessi alla memoria: uno per accedere alla page table (che risiede in memoria) e uno per accedere ai dati, si ha quindi un degrado del 100%. Costo per accedere a un dato in memoria: $2t$ (t =tempo di accesso alla memoria principale).

Il doppio accesso alla memoria si risolve utilizzando una cache dedicata per le entry della page table: registri associativi TLB (Translation look-aside buffer).

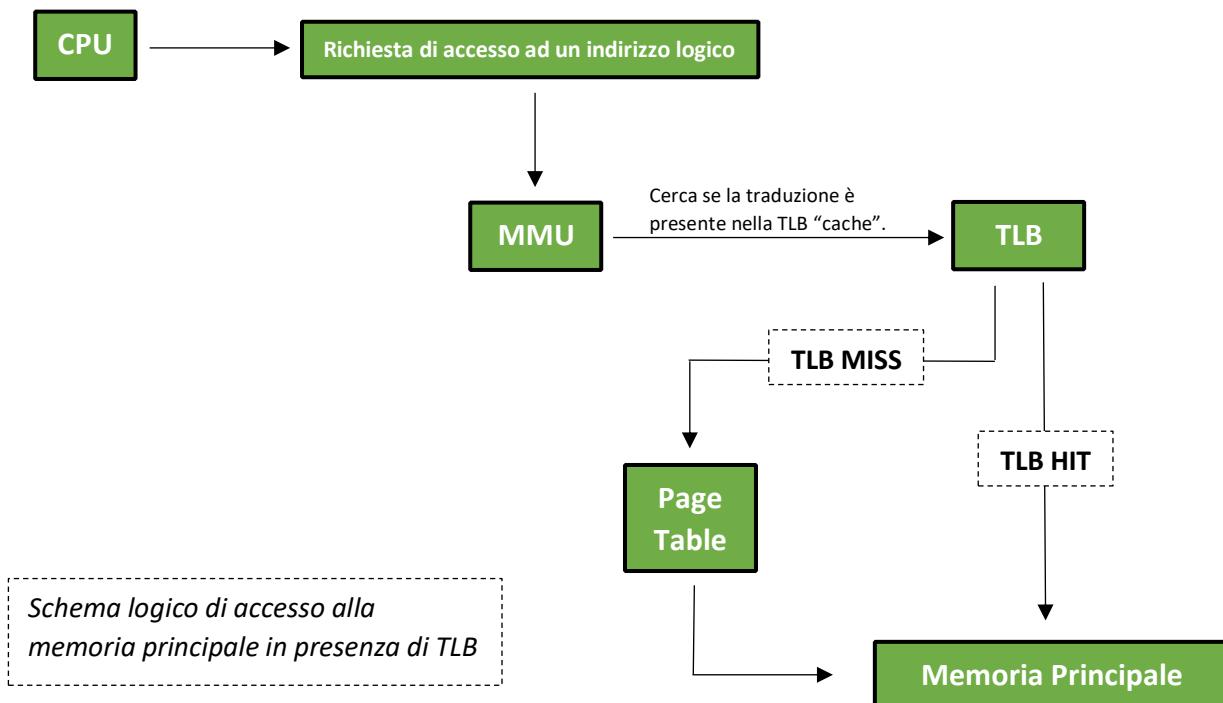
Esempio indirizzi page table: Indirizzo di 32bit, dimensioni pagine 4KB. Memoria fisica: 512MB. Page Table convenzionale.

In una tabella delle pagine convenzionale a singolo livello ci sono 2^{20} voci (infatti $32 - 12 = 20$, ovvero, 20 bit sono dedicati a rappresentare il numero di pagina logica) e 12 per l'offset della pagina ($2^{12} = 4KB$).

➤ TLB (Translation lookaside buffer) Registri associativi

In linea di principio, con paginazione ogni accesso alla memoria principale richiede due tempi di accesso. Uno per accedere alla page table, e un secondo per poi accedere ai dati. Una tecnica per migliorare le prestazioni è quella di sfruttare la località dei riferimenti mediante i TLB (Translation lookaside buffer, in italiano: registri associativi).

Il TLB è una memory cache, parte integrante della MMU. Il TLB tiene in memoria le traduzioni recenti di indirizzi logici in fisici nel caso servano in un istante successivo.



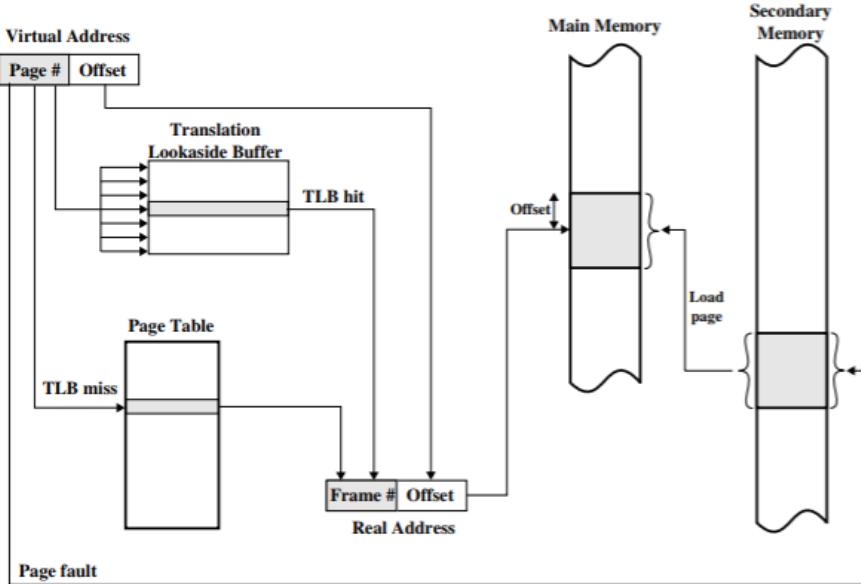
Tempo effettivo di accesso con TLB (tempo medio di accesso)

ϵ = tempo di lookup associativo (Tempo per controllare se una traduzione è in cache TLB)

α = Hit Ratio: percentuale dei page # reperiti nel TLB (compreso tra 0 e 1)

t = tempo di accesso alla memoria.

$$\text{EAT} = (t + \epsilon) \alpha + (2t + \epsilon)(1 - \alpha)$$



I TLB Miss possono essere gestiti via hardware dalla MMU, o via software dal sistema operativo. Per il principio di località, si ha che in media l'hit ratio è del 98% con cache di dimensioni nontroppo elevate, rimuovendo così quasi completamente i doppi accessi. Più grande è il buffer, maggiore sarà l'hit ratio.

Schema di funzionamento di traduzione di un indirizzo logico a uno fisico, con presenza di TLB su paginazione con page table in memoria.

• Paginazione a più livelli

Per ridurre l'occupazione della page table, si pagina la page table stessa. Solo le pagine effettivamente usate sono allocate in memoria principale.

Dato che ogni livello è memorizzato in memoria principale, la conversione dell'indirizzo logico in indirizzo fisico può necessitare di diversi accessi alla memoria. Il caching degli indirizzi di pagina visto precedentemente permette di ridurre drasticamente l'impatto degli accessi multipli.

Esempio: Nel caso di indirizzo logico a 32bit con pagine da 4K, esso viene diviso in una parte da 20bit per il numero pagina e in una parte da 12bit per l'offset. La page table è paginata, quindi il numero pagina (da 20bit) è a sua volta diviso in directory number e page offset, entrambi da 10bit. La page table, essendo salvata in memoria, richiede numerosi accessi per la conversione dell'indirizzo logico in fisico. Con page table ad un livello avevamo riscontrato un tempo pari a $2t$, (con t =tempo di accesso alla memoria). Con più livelli avremo un costo di nt (con n =numero di livelli) + 1. Nel caso di paginazione a due livelli avremmo quindi costo $3t$.

• Page Table invertita

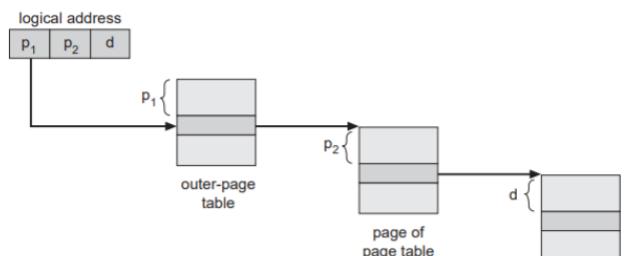
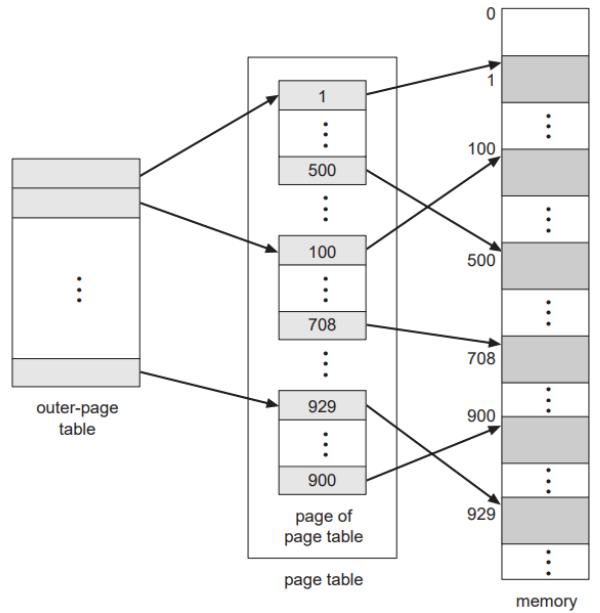
La page table invertita consiste nella presenza di un'unica tabella con una entry per ciascun frame della memoria. Ogni entry consiste nel numero della pagina (virtuale) memorizzata in quel frame. Diminuisce la memoria necessaria per memorizzare le page table, ma aumenta il tempo di accesso alla tabella.

Esempio: Indirizzo di 32bit, dimensioni pagine 4KB. Memoria fisica: 512MB. Page Table invertita.

$$512\text{MB} = 2^{29} \text{ bytes}$$

$$4\text{KB} = 2^{12} \text{bytes}$$

29-12=17 -> 17 bit dedicati al numero di pagina fisica, ovvero 2¹⁷ voci nella tabella mentre 12 bit sono usati per l'offset di pagina.



- **Segmentazione:** È una tecnica di gestione della memoria che suddivide la memoria fisica disponibile in blocchi (di lunghezza fissa o variabile) chiamati segmenti. La segmentazione presuppone una suddivisione logica dei programmi e dei dati. Tale suddivisione prevede l'individuazione di unità logiche di dimensioni diverse, chiamate segmenti, che possono ad esempio essere (nel caso di codice) una funzione, una routine, procedure etc. ovvero parti di codice indipendenti, che possono essere caricate in memoria solo nell'istante in cui esse vengono richieste. Per riferirci a uno di questi "oggetti" serve conoscere il numero di segmento che lo contiene, ed inoltre il suo indirizzo all'interno del segmento, detto offset. Quindi ogni indirizzo logico generato dalla CPU sarà del tipo:

<segment-number, offset>

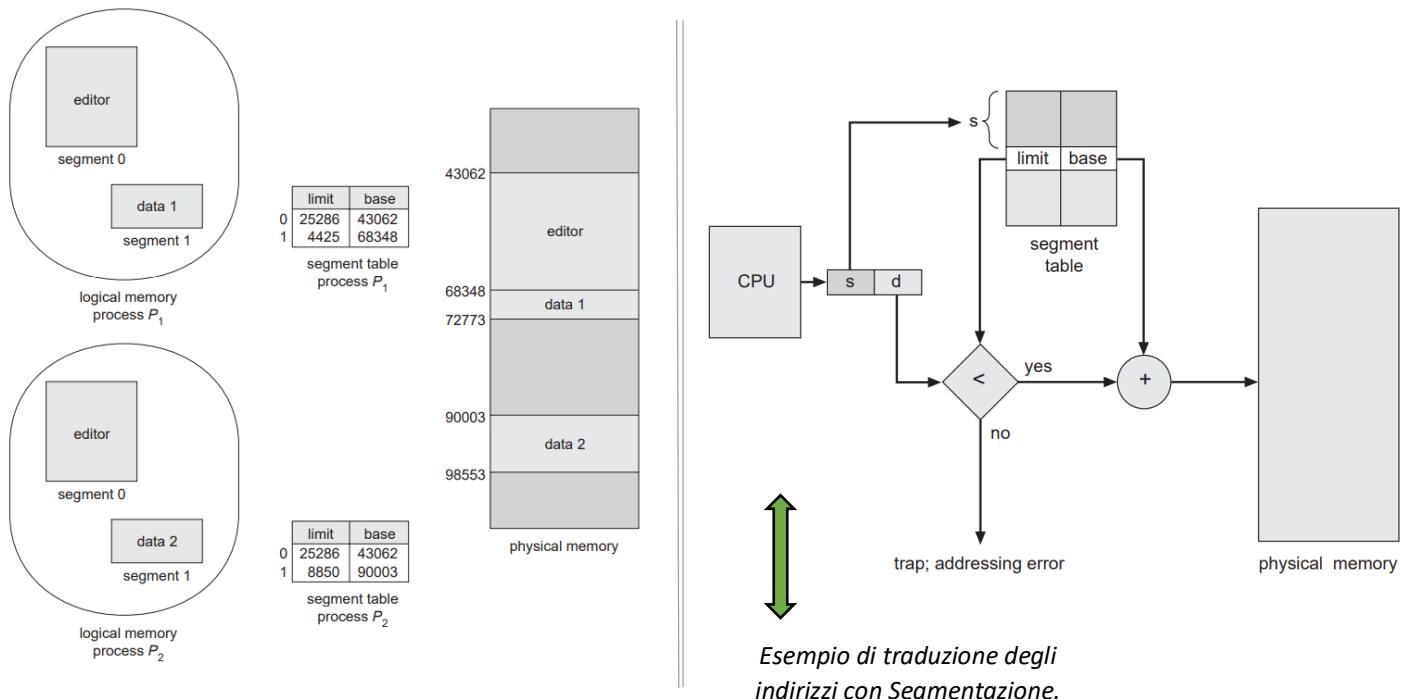
È presente una **segment table**, che mappa gli indirizzi bidimensionali dell'utente negli indirizzi fisici unidimensionali. Ogni entry nella segment table ha:

- *base*: indirizzo fisico di inizio segmento.
- *limit*: lunghezza/dimensione del segmento.

Implementazione della segment table:

La segment table viene tenuta in memoria principale. Sono presenti due registri importanti:

- *Segment-table base register (STBR)* punta all'inizio della tabella dei segmenti.
- *Segment-table length register (STLR)* indica il numero di segmenti usati dal programma, un segment number *s* è legale se *s* < STLR.



La segmentazione non porta a frammentazione interna, inoltre la segment table occupa meno memoria della page table per la paginazione. Causa però frammentazione esterna di grandi blocchi di memoria, quando un segmento viene deallocated.

• Segmentazione paginata

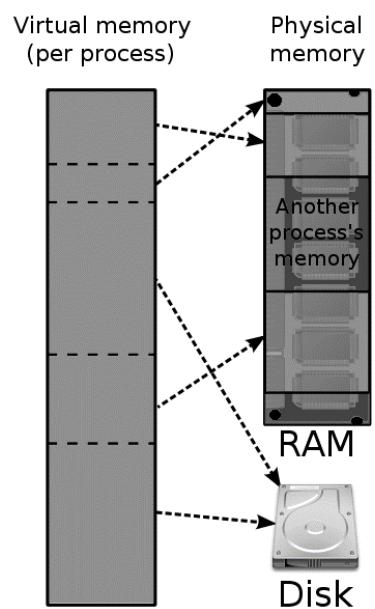
I segmenti di memoria vengono paginati. Questa tecnica permette di combinare i vantaggi di entrambi gli approcci. A differenza della pura segmentazione, nella segment table ci sono gli indirizzi base delle page table dei segmenti.

9. Virtual Memory

La memoria virtuale è una tecnica che permette di simulare uno spazio di memoria centrale maggiore di quello fisicamente presente o disponibile. Questo risultato si raggiunge utilizzando spazio di memoria secondaria, ad esempio il disco. In un sistema dotato di memoria virtuale, la CPU e i processi si riferiscono alla memoria principale con indirizzi logici che vengono tradotti dalla MMU in indirizzi fisici effettivi.

La MMU all'arrivo di una richiesta di accesso alla memoria esegue i seguenti passi:

- Traduce l'indirizzo logico in indirizzo fisico.
- Controlla che l'indirizzo fisico corrisponda a una zona di memoria fisicamente presente nella memoria centrale.
- Se invece la zona in questione è nello spazio di swap, la MMU solleva una eccezione di *page fault* e il sistema operativo si occupa di caricarla in memoria centrale, scartando una pagina già presente.



Questo meccanismo ha un prezzo in termini di prestazioni: la MMU impiega del tempo per tradurre l'indirizzo logico in indirizzo fisico, inoltre ce ne vuole molto di più per caricare una zona di memoria dallo spazio di swap nel caso si verifichi un *page fault* (*si passa da ns (10⁻⁹) a ms (10⁻³)*).

L'utilizzo di questo approccio porta ad alcuni vantaggi, principalmente permette allo spazio logico di essere più grande di quello fisico. Inoltre permette un minore utilizzo di memoria e quindi la presenza di più processi aumentando così il grado di multiprogrammazione.

La memoria virtuale porta alla necessità di caricare e salvare parti di memoria dei processi da/per il disco durante l'esecuzione (a runtime), questo ha un costo elevato, nasce così l'esigenza di utilizzare degli algoritmi più efficienti possibili.

La memoria virtuale può essere implementata per “*paginazione su richiesta*” oppure “*segmentazione su richiesta*”.

- **Paginazione su richiesta**

Schema a paginazione, con l'unica differenza che si carica una pagina in memoria solo quando richiesto, le altre rimangono su disco. Si ha meno utilizzo di memoria e la possibilità di avere più utenti/processi in contemporanea. Quando si fa riferimento ad una pagina non presente in memoria, la MMU si occuperà di sollevare una eccezione di *page fault* e successivamente il sistema operativo recupererà la pagina richiesta dal disco.

Alcuni concetti:

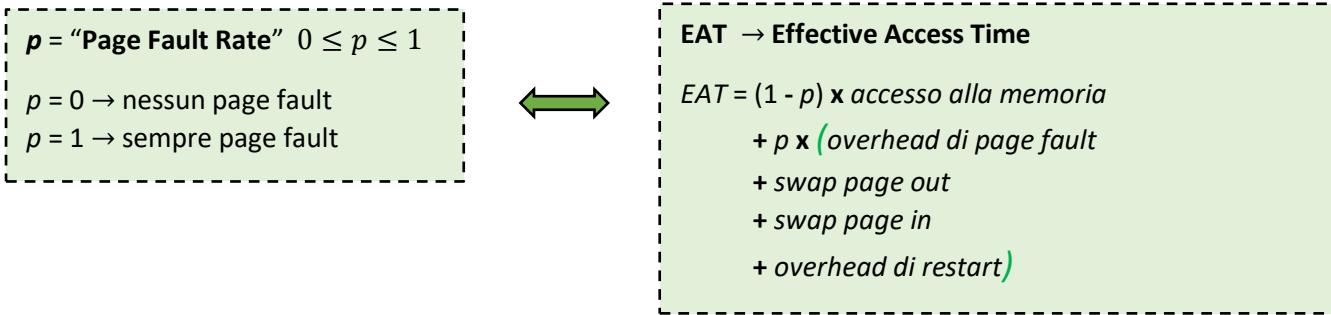
Paging: Il paging consiste nello scambio di pagine da/per il backing store. Attenzione a non confondersi con lo swapping che è lo scambio di interi processi da/per il backing store.

Valid-Invalid Bit: Ad ogni entry nella page table, si associa un bit di validità. 1: *in memoria*, 0: *non in memoria*. Inizialmente il bit di validità è a zero per tutte le pagine, in quanto si trovano tutte su disco. La prima volta che si fa riferimento ad una pagina non presente in memoria la MMU invia un segnale di interrupt alla CPU: *page fault*.

Gestione dei page-fault: All'arrivo di una entry nella page table il S.O. controlla che sia stato un accesso valido (se non è fuori dallo spazio di indirizzi virtuali associati al processo) in caso negativo chiude il processo (segmentation fault). Se l'accesso è valido ma la pagina non è in memoria il S.O. deve:

- Trovare una pagina in memoria non utilizzata e swapparla nel disco.
- Caricare la pagina richiesta nel frame liberato.
- Aggiornare le tabelle delle pagine.

Performance paginazione su richiesta:



Esempio:

Tempo di accesso alla memoria (comprensivo del tempo di traduzione) $\rightarrow 60\text{ns}$

Page fault rate $\rightarrow 0.5 \rightarrow 50\%$

Swap page time $\rightarrow 5\text{ms} (5 \times 10^6)$

$$\begin{aligned} EAT &= 60(1 - p) + (5 \times 10^6) \times 1.5 \times p \\ &= 60 + (7.5 \times 10^6 - 60)p \end{aligned}$$

Considerazioni sulla paginazione su richiesta

È necessario, al fine di ottimizzare le performance, avere un algoritmo di rimpiazzamento che porti al minor numero di page fault possibile.

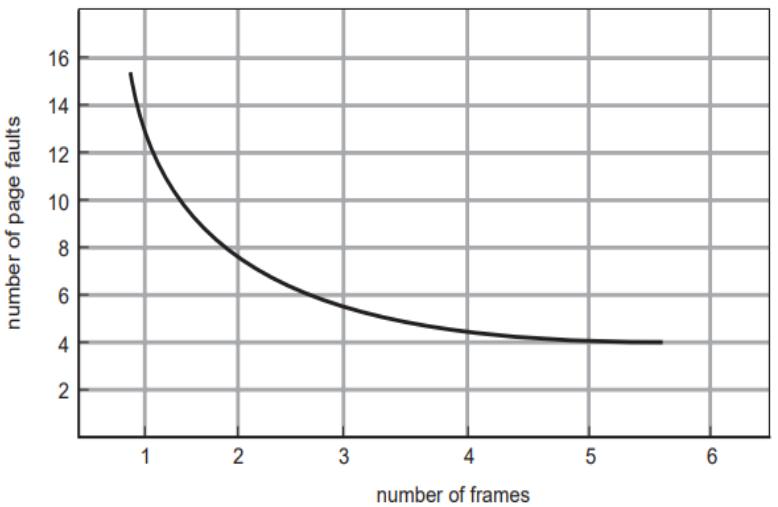
Inoltre, sempre per questioni di velocità, l'area di swap deve essere il più veloce possibile al fine di ridurre al minimo i tempi di recupero in caso di “swap-in”.

La creazione di processi riceve dei miglioramenti nel caso di paginazione su richiesta:

- **Creazione dei processi - Copy on Write (COW)** \rightarrow Permette a un processo padre di condividere in fase iniziale le stesse pagine in memoria con il processo figlio appena creato. Immediatamente dopo la duplicazione e per tutto il tempo in cui nessuna delle due risorse viene modificata, esse sono, di fatto, indistinguibili; durante questo lasso di tempo la reale esistenza di due copie indipendenti non è strettamente necessaria: il sistema può limitarsi a simulare l'operazione di duplicazione, mantenendo l'esistenza di un'unica copia e gestendo attraverso di essa, in modo del tutto trasparente ai richiedenti, tutte le operazioni di lettura destinate ad una qualsiasi delle due. La vera e propria duplicazione delle risorse può essere posticipata fino al momento in cui l'esistenza di due risorse indipendenti si rende realmente necessaria, cioè in corrispondenza di un'operazione di modifica dello stato (generalmente la scrittura di un nuovo contenuto, da cui il nome di *copy-on-write*) di una qualsiasi delle copie fittizie.
- **Memory-Mapped I/O** \rightarrow Permette di gestire l'I/O di file come accessi in memoria, ogni blocco di un file viene mappato su una pagina di memoria virtuale. Un file (es. DLL, .so) può essere così letto come se fosse in memoria, con demand paging. Dopo che un blocco è stato letto una volta, rimane caricato in memoria senza doverlo rileggere. La gestione dell'I/O è molto semplificata. Più processi possono condividere lo stesso file, condividendo gli stessi frame in cui viene caricato.

- **Sostituzione delle pagine**

All'aumentare del grado di multiprogrammazione, la memoria primaria verrà sovrallocata (la somma degli spazi logici dei processi in esecuzione è superiore alla dimensione della memoria fisica.) Al verificarsi di un page fault, potrebbero non esserci frame liberi, in quel caso il S.O. deve occuparsi di sostituire una pagina (la vittima) rimuovendola e dando spazio alla pagina da caricare. Il rimpiazzamento di pagina completa la separazione tra memoria logica e memoria fisica: una memoria logica di grandi dimensioni può essere implementata con una piccola memoria fisica. Esistono vari algoritmi di sostituzione delle pagine, implementati dal sistema operativo, che ovviamente mirano a scegliere la pagina vittima in modo più intelligente possibile, al fine di diminuire i page fault futuri. Generalmente all'aumentare della memoria primaria, ci si aspetta che il numero di page faults si riduca, come logicamente ci si aspetta, aggiungendo nuovi moduli di memoria, aumenta il numero di frame disponibili.



Principali algoritmi di sostituzione delle pagine:

- OPT (o MIN) *Limite teorico*
- FIFO (*First In - First Out*)
- LRU (*Last Recently Used*)
- Reference Bit
- NFU (*Not Frequently Used*)
- Aging
- CLOCK (o “second chance”)
- CLOCK Migliorato

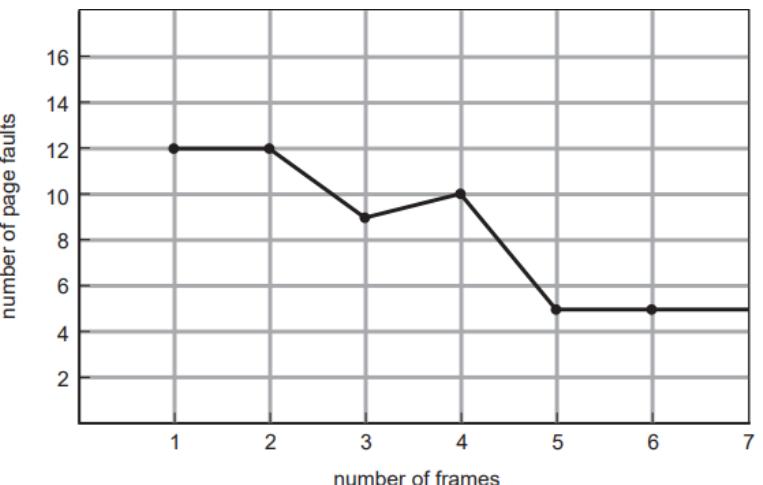
OPT (o MIN) *Limite teorico*: Consiste nel rimpiazzare la pagina che non verrà riusata per il periodo più lungo. E' l'algoritmo migliore, ma non è implementabile, in quanto non è possibile sapere quale sarà la pagina che verrà usata più tardi. Impossibile prevedere il futuro.

FIFO (*First In - First Out*): Si rimpiazza la pagina che è da più tempo in memoria. L'approccio FIFO soffre dell'anomalia di Belady.

Anomalia di Belady:

“Più memoria fisica non implica minori page fault.” Ovvero aumentando la memoria fisica, potremmo incorrere in più page fault. Come nell'esempio del grafico a destra.

Un algoritmo che non soffre di questa anomalia, viene chiamato “*algoritmo di stack*”



Algoritmo di stack:

Un algoritmo di rimpiazzamento pagine si dice di stack se per ogni reference string r , per ogni memoria m vale la seguente relazione.

Gli algoritmi di stack non soffrono dell'anomalia di Belady.

$$\longleftrightarrow \boxed{M(m,r) \subseteq M(m+1, r)}$$

LRU (Last Recently Used): Approssimazione dell' algoritmo OPT. Si "studia" il passato per "prevedere" il futuro. Si rimpiazza la pagina che da più tempo NON viene usata. Non soffre dell'anomalia di Belady. È uno dei migliori algoritmi ma necessita di notevole supporto hardware.

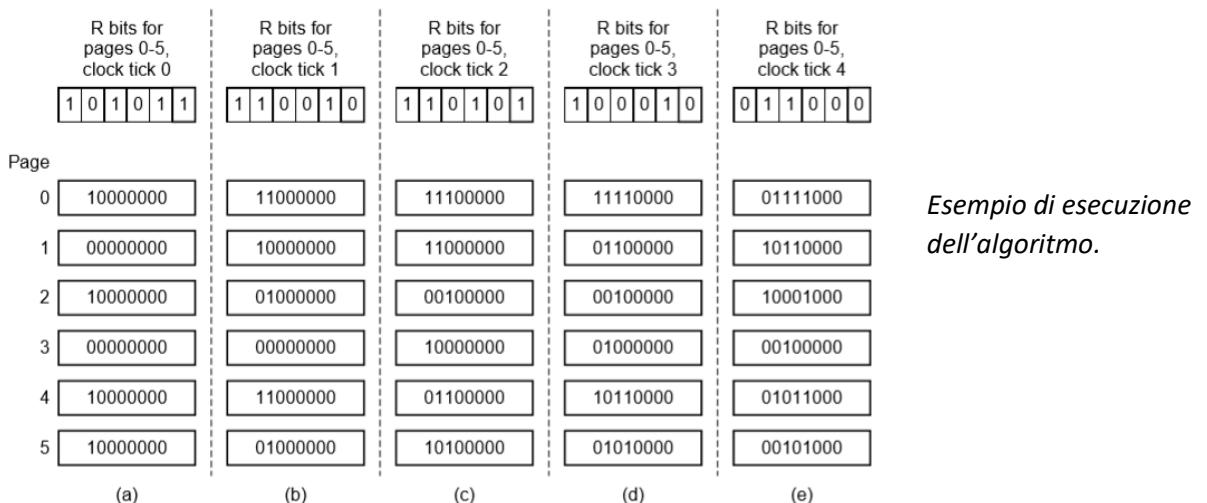
LRU può essere implementato con dei contatori sulla MMU che vengono automaticamente incrementati ad ogni accesso in memoria. Ogni entry nella page table ha un registro reference time. Ad ogni riferimento ad una pagina si copia il contatore nel registro della entry corrispondente, quando si deve liberare una pagina si cerca il reg. con il contatore più basso.

LRU può essere anche implementato a stack: Si tiene uno stack di numeri di pagina in una lista *double-linked*. Quando si fa riferimento ad una pagina, la si sposta in cima allo stack (Richiede la modifica di 6 puntatori). In caso di necessità di liberare una pagina, la vittima sarà quella in fondo allo stack.

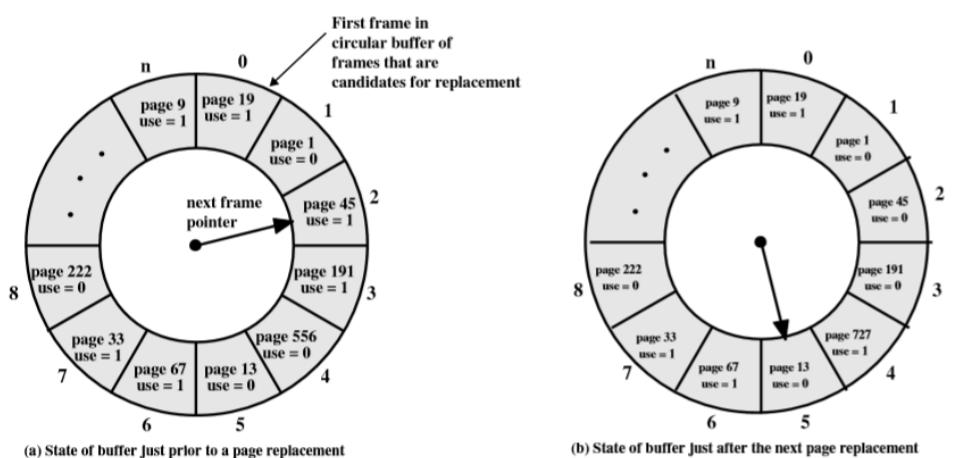
Reference Bit: È una semplificazione dell' LRU. Si associa ad ogni pagina un bit *R* inizialmente = 0. Quando si fa riferimento alla pagina, *R* viene settato a 1. Si rimpiazza la pagina che ha *R* = 0 (se c'è). Non c'è informazione sull'ordine. Se due pagine hanno *R* = 0 va rimpiazzata quella più "vecchia", ma qual è?

NFU (Not Frequently Used): Variante dell' LRU. Ad ogni pagina si associa un contatore. Ad intervalli regolari chiamati "tick" (ad esempio 10ms) ad ogni entry si somma il reference-bit al contatore.

Aging: Si aggiungono dei bit supplementari di riferimento, con peso diverso. Ad ogni pagina si associa un array di bit. Inizialmente tutto = 0. Ad intervalli regolari un interrupt del timer fa partire una routine che shifta gli array di tutte le pagine mettendovi i bit di riferimento nel primo slot di ciascun array (bit più significativo) che vengono resettati a 0.



CLOCK (o "second chance"): L'algoritmo si basa sull'ipotesi che se una pagina è stata usata pesantemente di recente, allora la probabilità che verrà usata prossimamente è molto elevata. Si utilizza il reference bit. Si segue un ordine "ad orologio" da cui deriva il nome clock. Se la pagina candidata ha il reference bit a 0 si rimpiazza, altrimenti se ha il bit a 1 si imposta il ref-bit a 0 e lascia la pagina in memoria, l'algoritmo passa alla pagina successiva seguendo le stesse regole. Per logica alla pagina viene data quindi una seconda possibilità, perché verrà sostituita solo in un successivo giro di "clock". Se tutti i bit sono a 1, l'algoritmo si comporta in modo FIFO. Algoritmo che approssima molto bene l' LRU. Viene utilizzato in molti S.O. (con opportuni miglioramenti).



CLOCK Migliorato: Nel clock migliorato vengono usati due bit per pagina. il reference bit r e il dirty bit d . Ogni pagina potrà quindi essere del seguente tipo (in ordine di bontà di essere la vittima):

- $[r=0 \ d=0]$ → non usata recentemente, non modificata.
- $[r=0 \ d=1]$ → non usata recentemente, modificata.
- $[r=1 \ d=0]$ → usata recentemente, non modificata.
- $[r=1 \ d=1]$ → usata recentemente, modificata.

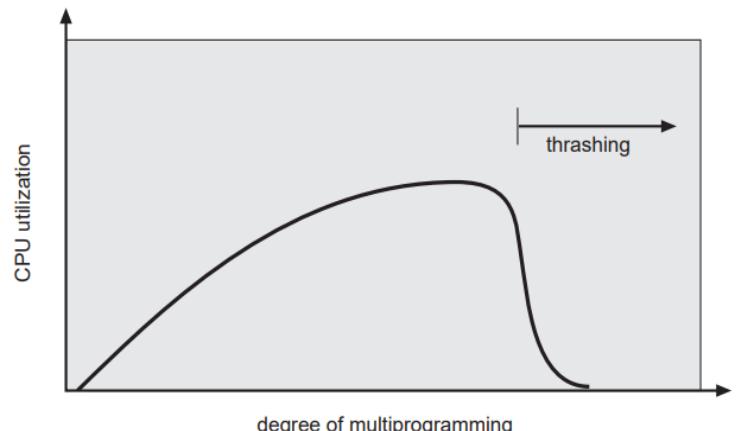
L'algoritmo scandisce la coda dei frame più volte (sempre a orologio). Cerca una pagina con (0,0) senza modificare i bit. Se la trova, è la vittima. Altrimenti cerca una pagina con (0,1) azzerando i reference bit, se la trova è la vittima. Ripete.

Matrice di memoria: Dato un algoritmo di rimpiazzamento pagine e una reference string, si definisce matrice di memoria $M(m,r)$ l'insieme delle pagine caricate all'istante r avendo m frame. Esempio di matrice di memoria per LRU:

Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	3	1	7	1	3	4	
	0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3			
	0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	5	7	7			
	0	2	1	1	5	5	5	5	6	6	6	4	4	4	4	4	4	4	4	4	5	5			
	0	2	2	1	1	1	1	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6			
	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2			
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
Page faults	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P			
Distance string	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3							

➤ Thrashing

Se un processo non ha abbastanza pagine allocate, il page fault rate è molto alto. Questo fenomeno viene chiamato *thrashing*. Il thrashing porta ad un basso utilizzo della CPU perché i processi sono impegnati in un continuo I/O (swapping delle pagine), inoltre il sistema operativo, potrebbe erroneamente aumentare il grado di multiprogrammazione (lanciando processi) perché riscontra un basso utilizzo della cpu. Più nello specifico, il thrashing avviene se a un processo non viene allocata abbastanza memoria per coprire la sua *località*. Quindi avviene quando la memoria fisica è inferiore alla somma delle località dei processi in esecuzione.



Principio di località

"Durante l'esecuzione di una data istruzione presente in memoria, con molta probabilità le successive istruzioni saranno ubicate nelle vicinanze di quella in corso. Nell'arco di esecuzione di un programma si tende a fare riferimenti continui alle stesse istruzioni."

La località di un processo è quindi un insieme di pagine che vengono utilizzate attivamente da un processo durante un determinato lasso di tempo. Un processo durante la sua esecuzione "migra" da una località all'altra.

Impedire il thrashing con il modello del Working-Set (WS)

$$\Delta = \text{Working-Set window} = \text{numero fisso di riferimenti a pagine}$$

Si sceglie ad esempio un working set che comprende le pagine a cui hanno fatto riferimento le ultime 10.000 istruzioni. Se il working-set è troppo piccolo, non copre l'intera località di un processo, al contrario se è troppo grande, copre più località. È possibile utilizzare il working-set come informazione per l'allocazione delle pagine. Il sistema operativo memorizza il WS di un processo p , allocandogli frame sufficienti per coprirlo. Alla creazione di un processo, questo viene ammesso nella coda ready solo se ci sono abbastanza frame liberi per coprire il suo WS. In caso contrario si sospende uno dei processi (sarà lo scheduling di medio termine a prendere la decisione corretta).

Implementazioni approssimative del Working Set:

- **Registri a scorrimento:** Si può implementare il working set con dei registri a scorrimento: Si mantengono 2 bit per pagina (oltre al reference bit). Un timer manda un interrupt ogni n unità di tempo. Al suo arrivo si shifta il reference bit di ogni pagina nei due bit in memoria e lo si cancella. Quando si deve scegliere una vittima, se uno dei 3 bit è = 1 allora la pagina è nel WS. Man mano che si diminuisce il lasso di tempo tra un interrupt e l'altro, aumenta la precisione, ma anche il dispendio di risorse.
- **Tempo virtuale corrente:** Il modello del Working set si può anche implementare mediante "un tempo virtuale corrente". Si mantiene un *tempo virtuale corrente* del processo. Ad ogni pagina viene associato un registro contenente il tempo di ultimo riferimento. Ad ogni page fault si controlla la tabella alla ricerca di una vittima. Se il reference bit è = 1, viene settato a 0 e si passa alla pagina successiva. Le pagine con reference bit = 0 possono avere età $> T$ o $< T$ (T = soglia secondo cui una pagina è vecchia o no). Se la pagina ha età $> T$ viene rimossa, altrimenti no. Se tutte le pagine hanno età $< T$ si cancella quella con età maggiore.
- **WSClock:** Variante dell'algoritmo di rimpiazzamento page "Clock" che inoltre tiene conto del Working Set. Invece di contare i riferimenti, si tiene conto di una finestra temporale T fissata. Si mantiene un contatore T del tempo CPU impiegato da ogni processo. Le pagine sono organizzate ad orologio, come nel "Clock". Ad ogni entry sono presenti i reference e i dirty bit (r e d) più un registro "Time of last use" che viene copiato dal contatore durante l'algoritmo. La differenza tra questo registro e il contatore è l' età della pagina. Ad un page fault ogni pagina può essere:

- $[r = 1] \rightarrow$ si setta $r = 0$, si copia $TLU = T$ e si passa alla pag. successiva.
- $[r = 0 \text{ età} > T] \rightarrow$ La pagina è nel WS, si passa avanti.
- $[r = 0 \text{ età} < T] \rightarrow$ Se $d = 0$ si libera la pagina, altrimenti si schedula un pageout e si passa avanti.

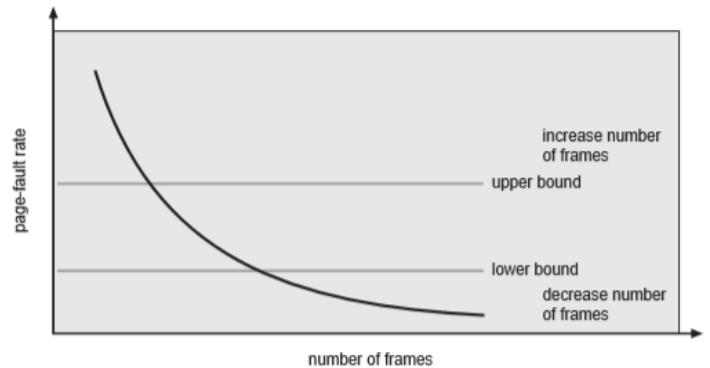
Cosa succede se si fa un giro completo? Se almeno un pageout è stato schedulato, si continua a girare (aspettando che le pagine scheduled vengano salvate). Altrimenti, significa che tutte le pagine sono nel working set. Soluzione semplice: si rimpiazza una qualsiasi pagina pulita. Se non ci sono neanche pagine pulite, si rimpiazza la pagina corrente.

Frequenza dei page-fault

È usanza stabilire un page-fault-rate accettabile.

- Se quello attuale è troppo basso, il processo perde un frame di memoria.
- Se quello attuale è troppo alto, il processo guadagna un frame di memoria.

Questo processo può essere distinto in sostituzione locale o globale.



Sostituzione locale: Ogni processo può rimpiazzare solo i propri frame. Si mantiene fisso il numero di frame allocati ad un processo (anche in presenza di frame liberi). Il comportamento di un processo quindi non è influenzato dagli altri processi.

Sostituzione globale: Un processo sceglie un frame tra tutti quelli del sistema, quindi può sottrarre un frame ad un altro processo. Questo approccio sfrutta meglio la memoria fisica, inoltre, un processo si comporterà quindi anche in base al comportamento degli altri processi.

Algoritmi di allocazione dei frame (pagine)

Ogni processo necessita di un numero minimo di pagine imposto dall'architettura. Esistono quindi diversi modi di assegnare i frame ai vari processi.

- Allocazione libera: dare a qualsiasi processo quanti frame desidera.
- Allocazione equa: dare ad ogni processo lo stesso numero di frame, porta a sprechi in quanto diversi processi hanno diverse necessità.
- Allocazione proporzionale: un numero di frame in proporzione alla dimensione del processo e della sua priorità

Esempio allocazione proporzionale: due processi da 10 e 127 pagine, su 62 frame.

$$p1 \rightarrow (10/(127+10)) * 62 = 4 \text{ frame allocati.}$$

$$p2 \rightarrow (127/(127+10)) * 62 = 57 \text{ frame allocati.}$$

Alcune considerazioni:

Buffering di pagine: si aggiunge un insieme chiamato *free list* di frame liberi agli schemi visti precedentemente. Il sistema cerca di mantere sempre un po' di frame sulla free list così quando si libera un frame, se è stato modificato lo si salva su disco, si mette il dirty bit a zero e si tiene in caso di riutilizzo futuro nella free list.

Quando un processo produce un page fault, si vede se la pagina è per caso sulla free list (soft page fault) altrimenti si prende dalla free list un frame e vi si carica la pagina richiesta da disco.

Prepaging: Caricare in anticipo le pagine che forse verranno usate. Si cerca di predirre cosa servirà.

Selezione della dimensione della pagina: solitamente imposta dall'architettura. Dimensione tipica: 4K-8K. Influenza:

- Frammentazione: meglio piccola.
- Dimensioni della page table: meglio grande.
- Quantità di I/O: meglio piccola.
- Tempo di I/O: meglio grande.
- Località: meglio piccola.
- Numero di page fault: meglio grande.

10. Input / Output

Esistono una varietà immensa di dispositivi I/O. Ad ogni modo essi possono essere categorizzati in modo molto grossolano in tre grandi famiglie.

- **Human readable:** orientati all'iterazione con l'utente (terminale, mouse, tastiera...).
- **Machine readable:** adatti alla comunicazione con il calcolatore (disco, nastro...).
- **Comunicazione:** adatti alla comunicazione tra calcolatori (modem, schede di rete....).

I dispositivi di I/O possono essere visti attraverso più livelli d'astrazione. Un ingegnere elettronico avrà una visione fisica, un dispositivo sarà un insieme di circuiteria, per il programmatore è più importante la visione funzionale. È importante sapere cosa fa e non come lo fa.

Una categorizzazione dei dispositivi di I/O più rigorosa è quella che tiene conto della loro logica d'accesso:

- **Dispositivi a blocchi:** permettono l'accesso diretto ad un insieme finito di blocchi di dimensione nota e costante. Il trasferimento è strutturato a blocchi. *Esempio: dischi.*
- **Dispositivi a carattere:** generano e/o accettano uno stream di dati, non strutturati, Non permettono indirizzamento. *Esempio: tastiera.*

Esistono dispositivi che esulano da queste categorie (ad es. il timer) che sono difficili da classificare. Ai giorni d'oggi la CPU non comunica direttamente con i dispositivi di I/O ma con i suoi "controller". Esistono due modi principali che la CPU utilizza per comunicare.

- **I/O mappato in memoria:** Usa lo stesso bus per indirizzare sia la memoria che i dispositivi di I/O, inoltre le stesse istruzioni della CPU che vengono utilizzate per leggere e scrivere la memoria sono utilizzate anche per accedere ai dispositivi di I/O.
- **I/O separato in memoria:** Un segmento a parte distinto dallo spazio indirizzi (bus extra) è collegato ai registri del controller.

Modi di I/O:

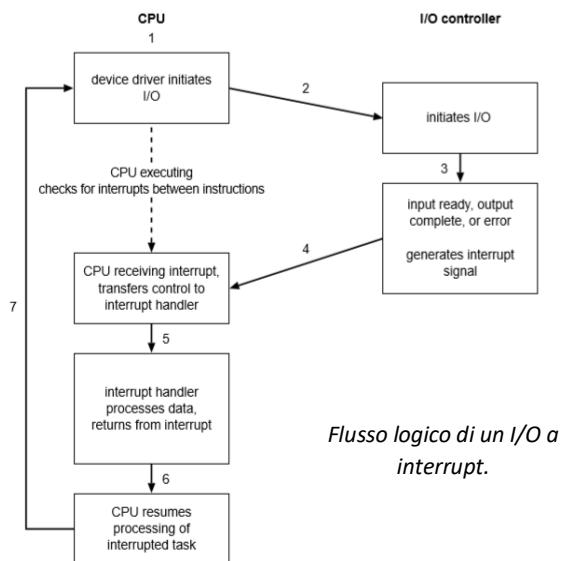
- **Programmed I/O (a interrogazione ciclica):** La CPU manda un comando di I/O e poi attende che l'operazione sia terminata, testando lo stato del dispositivo con un loop busy-wait (polling). Efficiente solo se la velocità del dispositivo è paragonabile a quella della CPU. Durante l'attesa la CPU non esegue computazione utile.

	Senza interrupt	Con interrupt
trasferimento attraverso il processore		
trasferimento diretto I/O-memoria	Programmed I/O	Interrupt-driven I/O
		DMA, DVMA

- **I/O a interrupt:** La CPU manda un comando di I/O e il processo viene sospeso. Quando l'I/O è terminato, il controller del dispositivo lancia un interrupt. Nel frattempo la CPU può mandare in esecuzione altri processi o altri thread dello stesso processo.

Vettore di Interrupt: Tabella che associa ad ogni interrupt l'indirizzo di una corrispondente routine di gestione.

- **DMA (Direct Memory Access):** meccanismo di una CPU che permette ad altri sottosistemi, come ad esempio i dispositivi di I/O di accedere direttamente alla memoria interna per scambiare dati, in lettura e scrittura, senza però coinvolgere l'unità di controllo per ogni byte trasferito tramite l'usuale meccanismo dell'interrupt. Il trasferimento quindi avviene direttamente tra il dispositivo di I/O e la memoria fisica, bypassando la CPU. Variante: DVMA (DMA con mem. virtuale). L'accesso diretto avviene nello spazio virtuale del processo e non in quello fisico.



Trasferimento dati tra una periferica e memoria principale tramite DMA: Il trasferimento avviene direttamente fra il dispositivo di I/O e la memoria fisica senza l'intervento della CPU (*se si escludono il momento iniziale e quello finale dell'operazione*). Inizialmente il controller DMA viene inizializzato specificando la periferica da cui prelevare i dati, l'indirizzo fisico X della memoria in cui memorizzare questi ultimi ed il numero C dei byte complessivi da trasferire. A questo punto il controller della periferica inizia il trasferimento, inviando ogni singolo byte al controller DMA. Quest'ultimo acquisisce il controllo del bus della memoria memorizzando il byte all'indirizzo X ; in seguito incrementa l'indirizzo per il prossimo byte e decrementa il contatore C . Quando quest'ultimo raggiunge il valore 0, il trasferimento si è concluso e viene inviato un interrupt alla CPU per segnalare l'evento. Il meccanismo di DMA non può funzionare senza un meccanismo di gestione degli interrupt in quanto alla fine dell'operazione di I/O deve comunque avvisare (interrompere) la CPU.

Fenomeno del Cycle Stealing (Sottrazione di cicli): Dato che il DMA deve trasferire i dati da/verso la memoria, deve anche acquisire e bloccare il bus. Quindi se la CPU nel frattempo ha necessità di accedere alla RAM, deve attendere che il DMA termini e liberi l'accesso al bus: questo fenomeno prende il nome di cycle stealing, dato che la CPU perde dei cicli utili a causa dell'attesa.

➤ Gestione degli interrupt

All'arrivo di un interrupt, viene fatta partire una specifica routine di gestione (per quel determinato interrupt) chiamata "interrupt handler", che si occuperà della gestione di quest'ultimo. Prima di fare ciò bisogna salvare lo stato della CPU. Risulta quindi necessario che tutte le azioni svolte dalla routine di gestione siano trasparenti rispetto al programma interrotto, cioè che al termine venga ripristinato tutto come era prima dell'interrupt. Lo stato della CPU può essere salvato in diversi modi:

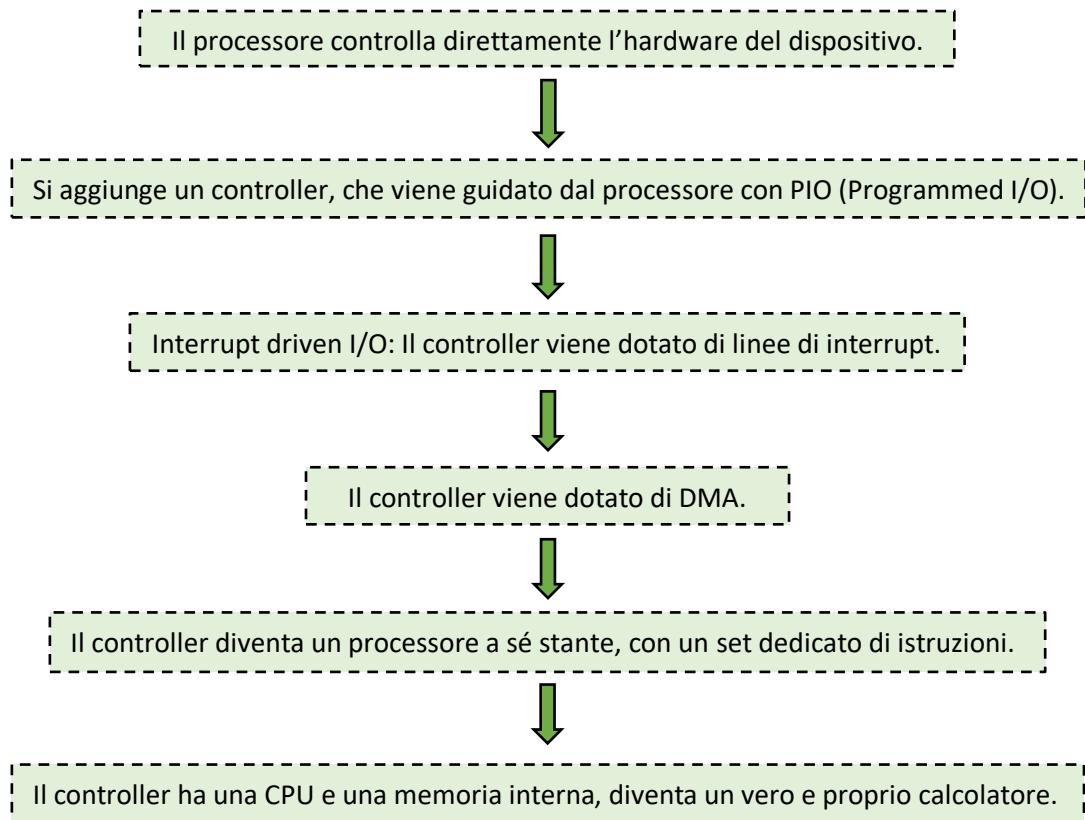
- sui registri della CPU: in questo caso gli interrupt non possono essere annidati.
- su uno stack che:
 - Se salvato su spazio utente può portare a problemi di sicurezza e efficienza (se si verifica un page fault? recuperare le informazioni da disco richiede molto tempo).
 - Se salvato sulla porzione di memoria del kernel è più efficiente ma può portare a sovraccarico della MMU.

Considerazione: Per le CPU con pipeline il PC (Program Counter) non identifica nettamente il punto in cui è arrivata l'esecuzione, anzi punta alla prossima istruzione da mettere nella pipeline. Ancora peggio per le CPU superscalari, le istruzioni possono essere già state eseguite ma non in ordine, il PC cosa indica allora? Da qui la necessità di definire interruzioni precise e interruzioni imprecise.

Interruzioni precise e imprecise

- **Precise:** Il PC è salvato in un posto noto. Tutte le istruzioni precedenti a quella dal pc vengono completate. Nessuna istruzione successiva al PC viene o è stata eseguita. Si ricava che lo stato dell'esecuzione puntata dal PC è noto. Questo approccio è più costoso, la CPU deve tener traccia del suo stato interno. È richiesto hardware più complesso, inoltre una parte di cache e registri viene utilizzata a questo scopo.
- **Imprecise:** Portano a difficoltà di ripresa precisa in hardware, la CPU riversa tutto lo strato interno sullo stack e lascia che sia il SO a decidere e capire cosa fare. Implica un hardware più semplice, meno costoso.

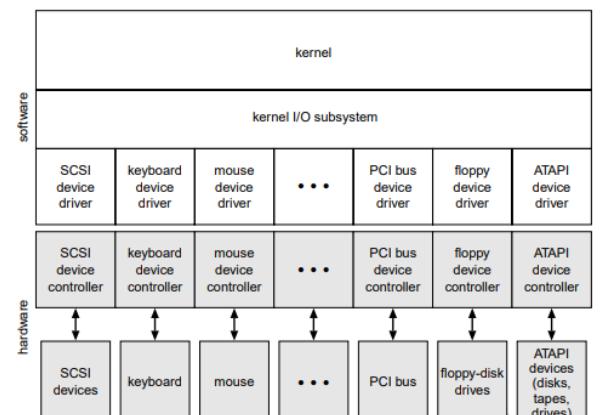
Evoluzione dell' I/O:



Interfaccia di I/O per le applicazioni:

È necessario avere un trattamento uniforme dei dispositivi I/O. Le chiamate di sistema I/O incapsulano il comportamento dei dispositivi in alcuni tipi generali. Le effettive differenze tra i dispositivi sono contenute nei driver e moduli del kernel dedicati a controllare ogni diverso dispositivo. Le chiamate di sistema raggruppano tutti i dispositivi in poche classi generali, uniformando i modi di accesso. Solitamente le modalità di accesso sono:

- I/O a blocchi
- I/O a carattere
- Accesso mappato in memoria
- Socket di rete



I/O bloccante, non bloccante e asincrono:

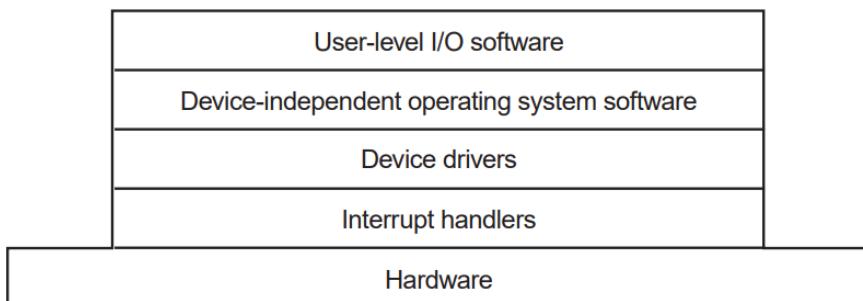
Un'altra caratteristica è quella della tipologia di input/output, esso può essere:

- **Bloccante:** Il processo viene sospeso finché l'I/O non è completato.
- **Non bloccante:** La chiamata ritorna non appena possibile, anche se l'I/O non è ancora terminato. ad esempio l'interfaccia utente che attende il movimento del mouse.
- **Asincrono:** Il processo continua mentre l'I/O viene eseguito (*ad esempio interfaccia utente in attesa del mouse*).

Sottosistema di I/O del kernel: Il sottosistema I/O del kernel deve fornire molte funzionalità:

- **Scheduling:** Si occupa di decidere in che ordine le system call devono essere eseguite, tipicamente le politiche first-come e first-served non sono molto efficaci, è quindi necessario adottare qualche politica per ogni dispositivo, per aumentare l'efficienza.
- **Buffering:** Mantenere i dati in memoria mentre sono in transito, per gestire differenti velocità, e diverse dimensioni dei blocchi di trasferimento.
- **Caching:** Mantenere una copia dei dati più usati in una memoria più veloce.
- **Spooling:** Buffer per dispositivi che non supportano I/O interleaved (*stampante*).
- **Accesso esclusivo:** Alcuni dispositivi possono essere usati solo da un processo alla volta, bisogna avere la possibilità di allocare e deallocare i dispositivi mediante apposite chiamate di sistema.
- **Gestione degli errori:** Il sistema operativo deve potersi difendere dal malfunzionamento dei dispositivi, gli errori potrebbero essere transitori o permanenti. Nel caso di situazioni transitorie, il S.O. può tentare di recuperare la situazione richiedendo di nuovo l'I/O, nel caso una system call ritorni un segnale d'errore. Spesso i dispositivi di I/O sono in grado di fornire dettagliate spiegazioni di cosa è successo. Il kernel deve inoltre registrare queste diagnostiche in appositi log di sistema.

I livelli del software I/O: Per raggiungere gli obiettivi sopra elencati, il software di I/O viene stratificato al fine di aumentare la modularità e l'astrazione.



- **Driver dei dispositivi**

Sono software di terze parti che accedono al controller dei device. Hanno la vera conoscenza di come far funzionare un dispositivo, inoltre implementano le funzionalità standardizzate. Vengono eseguiti in spazio kernel. Alla richiesta di un I/O il driver esegue i seguenti passi:

- Controllare la validità dei parametri passati.
- Accodare la richiesta in una coda di operazioni (soggetta a scheduling).
- Eseguire le operazioni accedendo al controller del device.
- Impostare il processo in wait o attendere la fine dell'operazione in busy-wait.
- Controllare lo stato dell'operazione nel controller.
- Restituire il risultato.

I driver devono essere rientranti: a metà di una esecuzione può essere lanciata una nuova esecuzione. I driver non possono eseguire system-call (sono in uno strato sottostante), ma possono accedere ad alcune funzionalità del kernel.

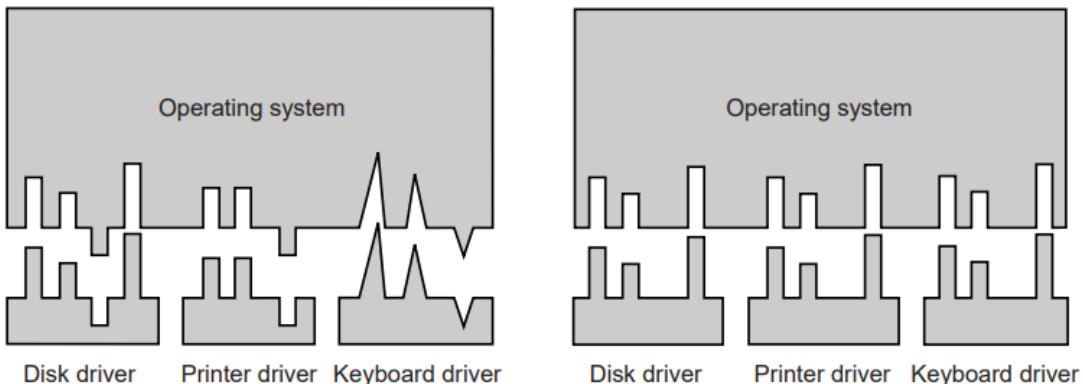
Software di I/O indipendente dai dispositivi: Implementa le funzionalità comuni a tutti i dispositivi di una certa classe:

- Forniscono un'interfaccia uniforme per i driver/software ai livelli superiori (file system, user-software).
- Permettono la bufferizzazione dell' I/O.
- Segnalazione degli errori.
- Allocazione e rilascio di dispositivi ad accesso dedicato.
- Uniformizzazione della dimensione dei blocchi (blocco logico).

Interfacciamento uniforme tra SO e driver

È buona pratica dai produttori di sistemi operativi fornire delle specifiche sulla scrittura dei driver, al fine di dare delle linee guida sulla loro struttura, così da ottenere un interfacciamento tra S.O. e driver più standardizzato possibile.

- Viene facilitato se anche l'interfaccia dei driver è standardizzata.



- Gli scrittori dei driver hanno una specifica di cosa devono implementare.
- Deve offrire anche un modo di denominazione uniforme flessibile e generale.
- Implementare un meccanismo di protezione per gli strati utente.

Alcuni concetti:

Bufferizzazione: Tecnica in cui viene utilizzato un buffer (memoria tampone/di transito) per compensare le differenze di velocità nel trasferimento e nella trasmissione dati. Permette quindi di disaccoppiare la chiamata di sistema di scrittura con l'istante di effettiva uscita dei dati (output asincrono).

Gestione degli Errori di I/O: Gli errori di I/O possono venire causati da molteplici cause:

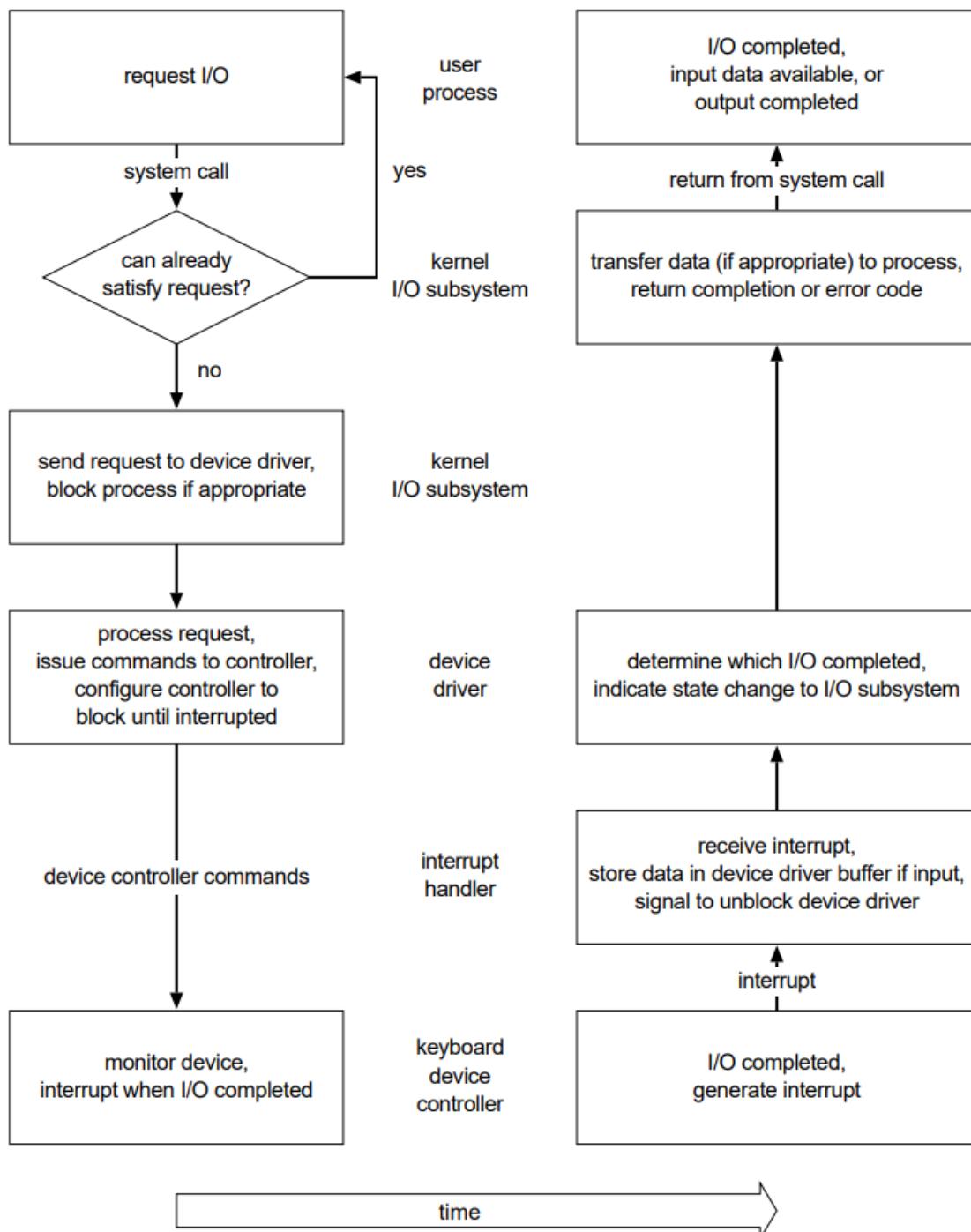
- *Errore di programmazione:* Viene chiesto qualcosa di impossibile, ad esempio la scrittura su tastiera. La chiamata di sistema viene abortita e viene segnalato un errore.
- *Errore del dispositivo:* Se transitori: cercare di ripetere le operazioni fino a che l'errore viene superato (rete congestionata). Abortire la chiamata: adatto per situazioni non interattive, o per errori non recuperabili. Far intervenire l'utente/operatore: adatto per situazioni riparabili da intervento esterno (es.: manca la carta)

Performance in ambito di I/O:

L'I/O è un fattore predominante nelle performance di un sistema, consuma tempo di CPU per eseguire i driver e il codice kernel di I/O. Inoltre porta a continui cambi di contesto all'avvio dell'I/O e alla gestione degli interrupt. Trasferimenti dati da/per i buffer consumano cicli di clock e spazio in memoria.

Per ridurre le performance vengono adottati diversi accorgimenti:

- Ridurre il numero di context switch.
- Ridurre spostamenti di dati tra dispositivi e memoria, e tra memoria e memoria.
- Ridurre gli interrupt preferendo grossi trasferimenti, utilizzando controller intelligenti, e interrogazione ciclica.
- Utilizzare dei canali di DMA o bus dedicati.
- Aumentare il parallelismo implementando le primitive in hardware.
- Bilanciare le performance di CPU, memoria, bus e dispositivi di I/O al fine di evitare il sovraccarico di uno di essi che causerebbe l'inutilizzo degli altri.

Schema di gestione di una richiesta di I/O:

11. Dischi

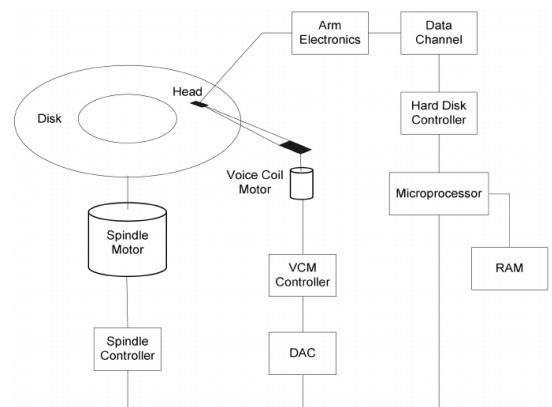
Struttura dei dischi

I "dischi" sono dei dispositivi di memoria di massa (memoria secondaria) di tipo magnetico, che utilizzano uno o più dischi magnetizzati per l'archiviazione di dati. A livello logico sono indirizzati come dei grandi array monodimensionali di blocchi logici, dove il blocco logico è la più piccola unità di trasferimento con il controller. L'array monodimensionale è mappato sui settori del disco in modo sequenziale.

Il settore 0 è il primo settore della prima traccia del cilindro più esterno. Successivamente la mappatura procede in ordine sulla traccia, poi su quelle più interne. Il SO è responsabile dell'uso efficiente e corretto dei dischi, raggiunto mettendo in atto politiche di scheduling più efficienti possibili.

Funzionamento dei dischi rigidi

Un disco rigido è costituito fondamentalmente da uno o più piatti in rapida rotazione, realizzati in alluminio o vetro, rivestiti di materiale ferromagnetico e da due testine per ogni disco (una per lato), le quali, durante il funzionamento "fluttuano" alla distanza di poche decine di nanometri dalla superficie del disco leggendo o scrivendo i dati. La testina è tenuta sollevata dall'aria mossa dalla rotazione stessa dei dischi la cui frequenza o velocità di rotazione può superare i 15.000 giri al minuto; attualmente i valori standard di rotazione sono 4.200, 5.400, 7.200, 10.000 e 15.000 giri al minuto. La memorizzazione o scrittura dell'informazione sulla superficie del supporto ferromagnetico consiste sostanzialmente nel cambiamento di stato di un determinato settore del disco (magnetizzato o smagnetizzato). Ad un certo stato (verso) di magnetizzazione è associato un bit di informazione (1 o 0).



Schedulazione dei dischi

Il tempo di accesso di un determinato dato memorizzato su disco ha due componenti principali date dall'hardware:

- *Seek time* = il tempo medio per spostare le testine sul cilindro contenente il settore richiesto.
- *Latenza rotazionale* = il tempo aggiuntivo necessario affinché il settore richiesto passi sotto la testina.

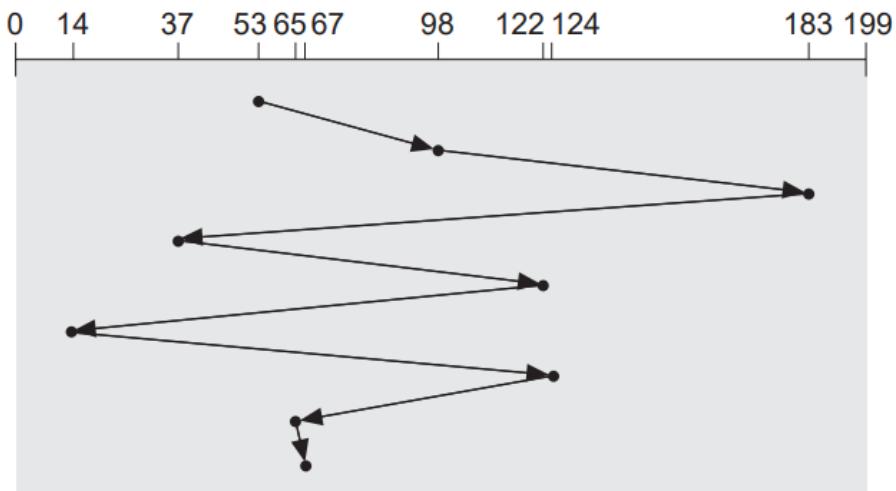
NB: *Tenere traccia della posizione angolare dei dischi è difficile, mentre si sa bene su quale cilindro si trova la testina. Inoltre: la latenza rotazionale risulta, in media, metà del tempo per effettuare una singola rotazione.*

Si ha come obiettivo quello di minimizzare il tempo speso in seek, se ne deduce che l'organizzazione dei dati all'interno del disco e gli algoritmi di ricerca e schedulazione delle richieste di lettura dei dati andranno a impattare in modo significativo sulle performance di questi dispositivi.

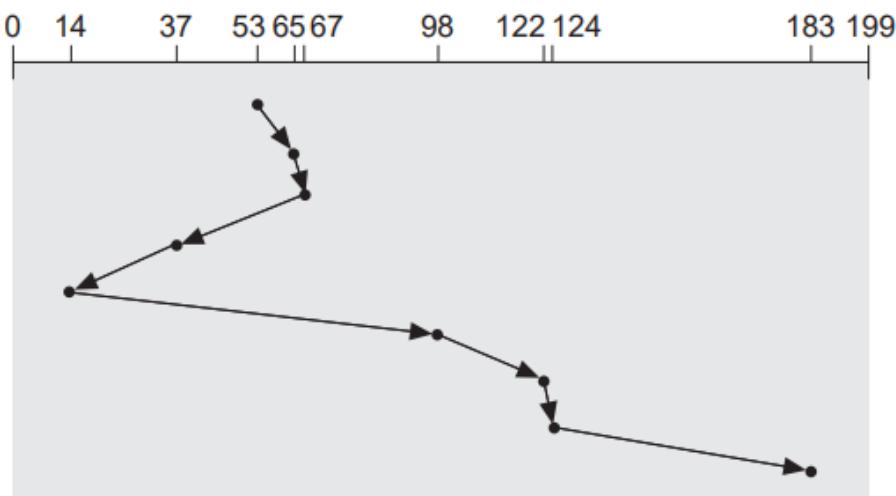
Esistono molti algoritmi di schedulazione delle richieste di I/O per un disco. Seguono le illustrazioni di alcuni di questi algoritmi con la seguente coda di richieste: 0–199: 98, 183, 37, 122, 14, 124, 65, 67 partendo dalla pos. 53.

- SSTF è molto comune e semplice da implementare, è abbastanza efficiente.
- SCAN e C-SCAN sono migliori per i sistemi con un grande carico di I/O con dischi.
- SSTF e C-LOOK sono scelte ragionevoli come scelte di default.

Si noti che le performance dipendono molto da come vengono allocati i file, ovvero da come è implementato il filesystem. Inoltre le performance sono influenzate molto dal numero e dal tipo di richieste. L'algoritmo di scheduling dei dischi dovrebbe essere un modulo separato dal resto del kernel, facilmente rimpiazzabile se necessario.

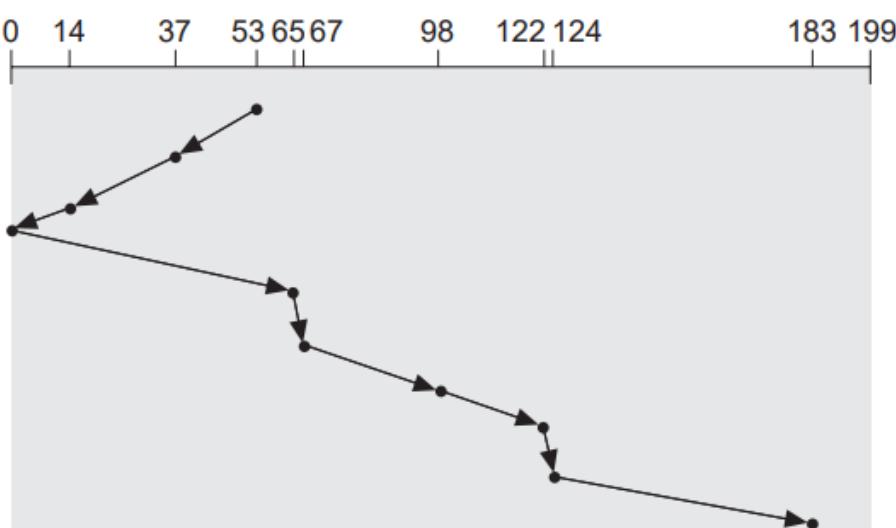
FCFS: First Come First Served

Come suggerisce il nome viene servita la prima richiesta di I/O ricevuta. Le richieste vengono quindi servite in ordine cronologico di arrivo.

SSTF: Shortest Seek Time First

Si seleziona la richiesta con il minor tempo di seek dalla posizione corrente, ovvero, la richiesta più vicina.

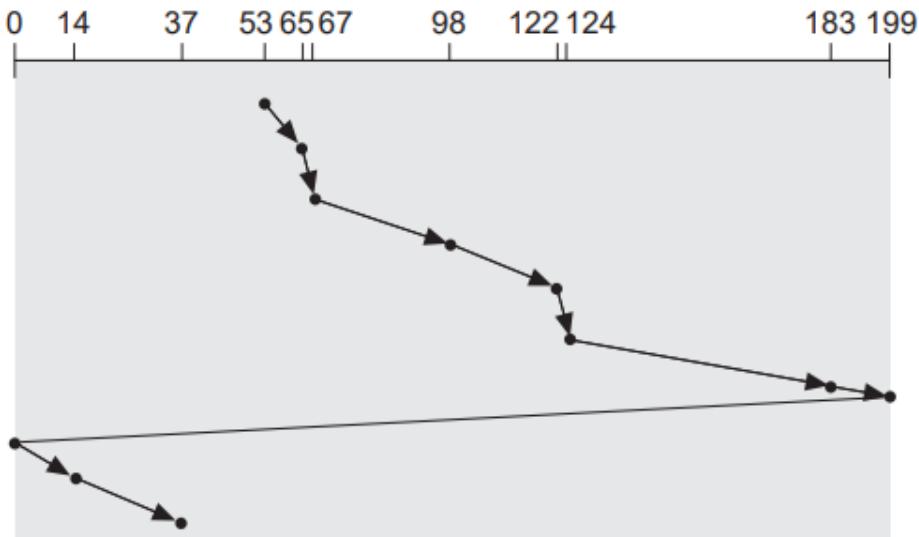
NB: può causare starvation. Su questo esempio 36,8% della distanza percorsa dal FCFS.

SCAN

Il braccio scandisce l'intera superficie del disco, da un estremo all'altro, servendo le richieste man mano. Agli estremi inverte la direzione.

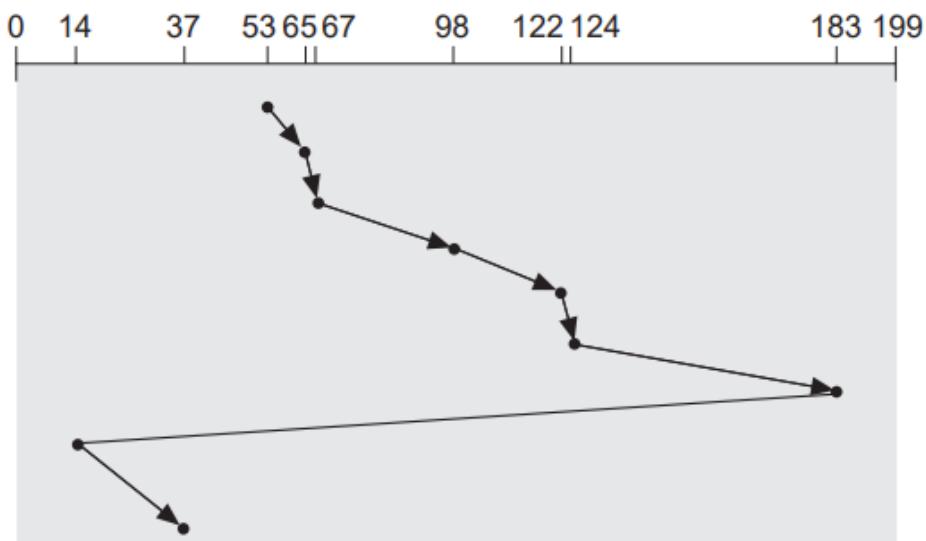
Variante: LOOK: come SCAN ma inverte la direzione non agli estremi ma quando "vede" che procedendo in quella direzione non incontrerà richieste.

C-SCAN



Garantisce un tempo di attesa più uniforme ed equo di SCAN. Tratta i cilindri come in lista circolare. Si muove da un estremo all'altro del disco, quando arriva alla fine ritorna immediatamente all'inizio del disco senza servire niente durante il rientro.

C-LOOK



Miglioramento del C-SCAN e del LOOK, il braccio si sposta solo fino alla richiesta attualmente più estrema e non fino alla fine del disco e poi inverte immediatamente la sua direzione.

Gestione dell'area di swap: Una possibilità è quella di gestire l'area di swap come un file del file system con il vantaggio che questo può crescere e decrescere a seconda delle necessità del sistema operativo, ma con lo svantaggio che, a lungo andare, si frammenterà e l'accesso diverrà inefficiente (*i sistemi operativi della famiglia Windows adottano questa soluzione*). L'alternativa è quella di dedicare all'area di swap un'intera partizione del disco, gestita in modalità diretta dal sistema operativo (ovvero, senza passare dal file system). Il vantaggio che ne risulta è una maggiore velocità d'accesso e l'assenza di problemi di frammentazione. Lo svantaggio principale è che, una volta deciso quanto spazio dedicare all'area di swap, quest'ultimo è "perso", ovvero, non può essere restituito alle partizioni con normale file system per la memorizzazione dei file ordinari. (*Linux ed i sistemi basati su Unix operano in questo modo*).

Affidabilità e performance dei dischi

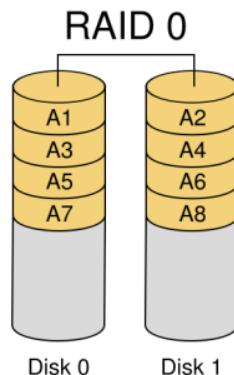
Concetto chiave: tempo medio fra guasti: *Mean time between failures: MTBF*

$$MTBF_{array} = \frac{MTBF_{disco}}{\#dischi}$$

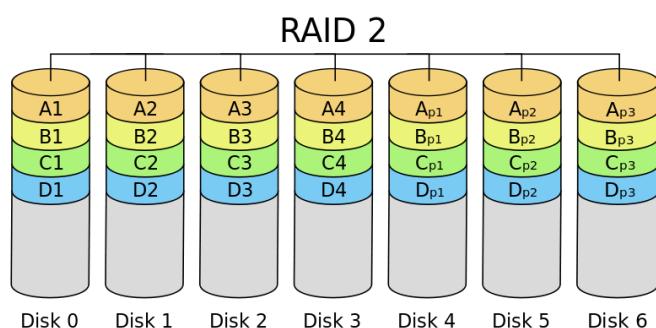
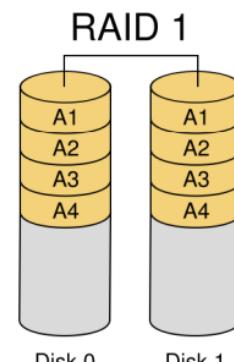
RAID: Redundant Array of Inexpensive/Independent Disks. È una tecnica di installazione raggruppata di diversi dischi che fa sì che gli stessi, nel sistema appaiano e siano utilizzati come se a livello logico fossero un unico dispositivo di memorizzazione. Gli scopi del RAID sono: aumentare le performance, e/o rendere il sistema resiliente alla perdita dei dati nel caso di guasto di uno o più dischi. Alcune configurazioni RAID inoltre permettono la sostituzione di un disco guasto senza l'interruzione del servizio. La gestione del RAID viene effettuata dal controller (RAID hardware) o dal software del driver dispositivo (RAID software).

Tipologie di RAID:

RAID 0: I dati vengono suddivisi in blocchi (stripes) ed ognuno di questi viene memorizzato in un disco dell'array a disposizione secondo lo schema indicato in figura (in modo alternato). Questa configurazione è l'unica che non implementa rindondanza di dati. Non c'è perdita di "spazio", ma solo guadagno di velocità, in quanto il carico di lavoro viene suddiviso fra più dischi. Aumenta la probabilità di guasto. Nel caso uno dei due dischi fallisca, fallirà l'intero sistema. Non c'è modo di recuperare i dati, in quanto questi sono "spezzati" in più parti e nessun disco contiene dati interi. Può essere effettuato su un minimo di due dischi. ($MTBF = F_1 + F_2 + \dots + F_n$)



RAID 1: Questa tipologia di RAID predilige la sicurezza, implementando la rindondanza tra i dati. I dati vengono duplicati (mirroring) su più copie di dischi. La lettura dei dati raddoppia (nel caso di 2 dischi), mentre quella di scrittura rimane quella di un singolo disco. In caso di richiesta di lettura di n blocchi, nel caso di raid a 2 dischi, le n richieste possono essere suddivise tra disco 1 e disco 0. Nel caso di guasto di un disco si può immediatamente utilizzare l'altro, senza bisogno di procedure di ricostruzione dei dati. Il principale aspetto negativo è lo spreco di spazio, nel caso di raid 1 a 4 dischi, avremmo lo spazio di un disco unico, sprecando quindi 75% della memoria secondaria. ($MTBF = F_1 \times F_2 \times \dots \times F_n$)



RAID 2: Un sistema RAID 2 divide i dati al *livello di bit* (invece che a blocchi). Usa un codice di Hamming per la correzione degli errori, che permette di correggere errori su singoli bit e di rilevare errori doppi. Questi dischi sono sincronizzati dal controllore, in modo tale che la testina di ciascun disco sia nella stessa posizione in ogni disco. Questo sistema si rivela molto efficiente in ambienti in cui si verificano numerosi errori di lettura o scrittura, ma in ambienti più prestanti, data l'elevata affidabilità dei dischi, il RAID 2 non viene utilizzato.

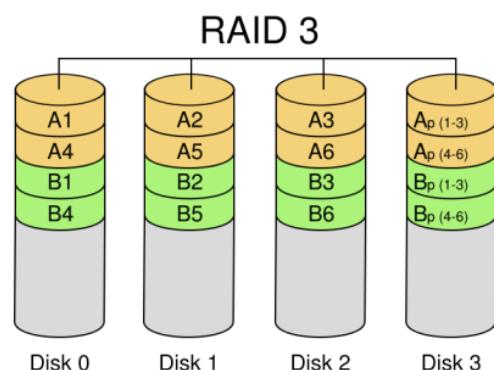
RAID 3: Un sistema RAID 3 usa una divisione al *livello di byte* con un disco dedicato alla parità. In caso di guasto, si accede al disco di parità e i dati vengono ricostruiti. Una volta che il disco guasto viene rimpiazzato, i dati mancanti possono essere ripristinati e l'operazione può riprendere. La ricostruzione dei dati è piuttosto semplice. Si consideri un array di 5 dischi nel quale i dati sono contenuti nei dischi X0, X1, X2 e X3 mentre X4 rappresenta il disco di parità. La parità per l'i-esimo bit viene calcolata come segue:

$$X4(i) = X3(i) \text{ XOR } X2(i) \text{ XOR } X1(i) \text{ XOR } X0(i)$$

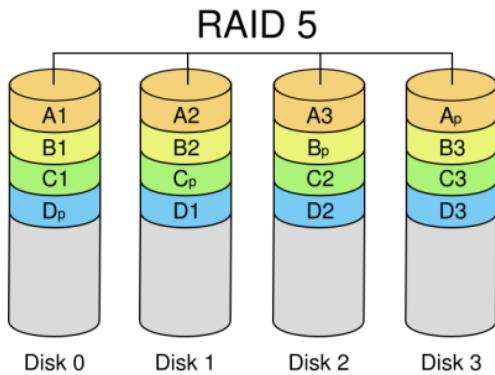
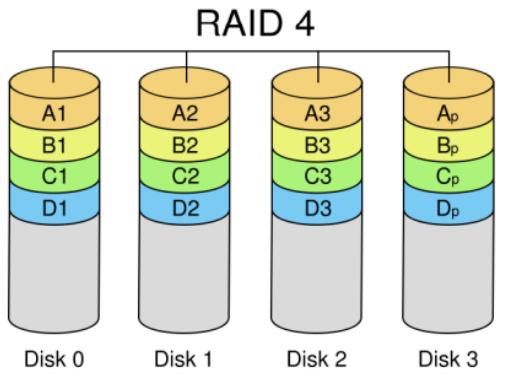
Si supponga che il guasto interessi X1. Se eseguiamo l'OR esclusivo di $X4(i)$ $\text{XOR } X1(i)$ con entrambi i membri della precedente equazione otteniamo:

$$X1(i) = X4(i) \text{ XOR } X3(i) \text{ XOR } X2(i) \text{ XOR } X0(i)$$

Così, i contenuti della striscia di dati su X1 possono essere ripristinati dai contenuti delle strisce corrispondenti sugli altri dischi dell'array. Questo principio persiste nei livelli RAID superiori.

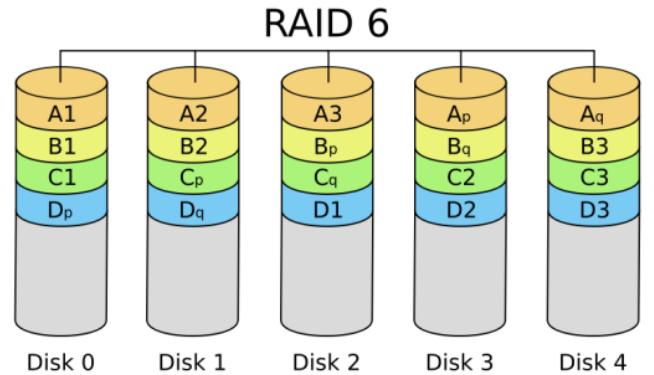


RAID 4: Il sistema RAID 4 usa una divisione dei dati a livello di blocchi e mantiene su uno dei dischi i valori di parità, in maniera molto simile al RAID 3, dove la suddivisione è a livello di byte. Questo permette ad ogni disco appartenente al sistema di operare in maniera indipendente quando è richiesto un singolo blocco. Se il controllore del disco lo permette, un sistema RAID 4 può servire diverse richieste di lettura 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.



RAID 5: Un sistema RAID 5 usa una suddivisione dei dati a livello di blocco, distribuendo i dati di parità uniformemente tra tutti i dischi che lo compongono. È una delle implementazioni più popolari, sia in software, sia in hardware. Nell'esempio sottostante, una richiesta al blocco A1 potrebbe essere evasa dal disco 1. Una simultanea richiesta per il blocco B1 dovrebbe aspettare, ma una richiesta simultanea per il blocco B2 potrebbe essere evasa in contemporanea. Il principio di funzionamento è quello del RAID 4, ma i blocchi di parità sono memorizzati in modo distribuito sui dischi dell'array

RAID 6: Il principio di funzionamento è quello del RAID 5, ma vengono utilizzati due tipi di controllo d'errori. Un sistema RAID 6 usa una divisione a livello di blocchi con i dati di parità distribuiti due volte tra tutti i dischi. Nel RAID-6, il blocco di parità viene generato e distribuito tra due stripe di parità, su due dischi separati, usando differenti stripe di parità nelle due direzioni. Il RAID-6 è più ridondante del RAID-5, ma è molto inefficiente quando viene usato in un numero limitato di dischi. Come vantaggio ha una elevatissima tolleranza ai guasti, ma pessime performance in scrittura.

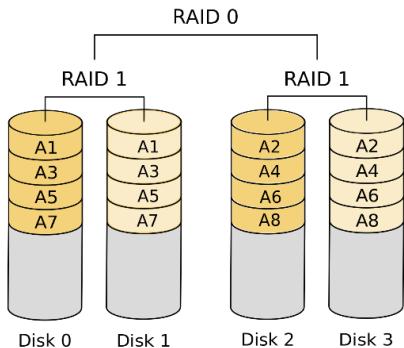


RAID Composti a più livelli:

È possibile combinare fra loro diversi livelli RAID . Esempi classici sono:

(L'ordine dei livelli si legge da destra a sinistra, ad esempio raid 10 equivale a un raid 0 di raid 1)

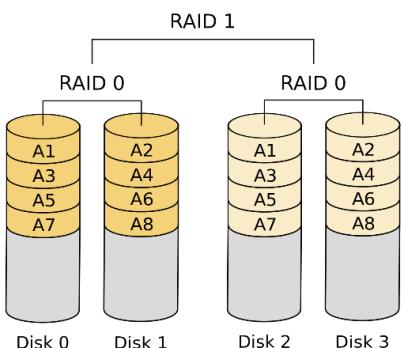
RAID 1+0



RAID 1+0: un insieme di dischi in mirror (RAID 1) vengono suddivisi in “stripes” in un secondo insieme di dischi (RAID 0). Ogni disco di ogni sistema RAID 1 può danneggiarsi senza far perdere dati al sistema. Comunque, se il disco danneggiato non viene sostituito, il disco rimasto nel RAID 1 rimane il punto critico del sistema. Se il disco rimasto si dovesse rompere, tutte le informazioni contenute nell'intero sistema andrebbero perse.

$$MTBF = (F \times F) + (F \times F) = 2F^2$$

RAID 0+1



RAID 0+1: le immagini di un insieme di dischi a livello 0 vengono replicate in un insieme di dischi a livello 1. Il sistema non è così robusto come il RAID 1+0 e non può sopportare la rottura simultanea di due dischi, se non appartengono alla stessa famiglia di RAID 0. Cioè, se un disco si rompe, ogni altro disco dell'altra stripe (riferita al RAID 1) è un elemento critico per il sistema. Inoltre, se un disco viene sostituito, per ricostruire il sistema devono partecipare tutti i dischi dell'insieme RAID 0 ancora funzionante.

$$MTBF = (F+F) \times (F+F) = 4F^2$$

Solid State Drive (SSD)

Un' unità di memoria a stato solido (in acronimo SSD dal corrispondente termine inglese solid-state drive) in elettronica e informatica, è un dispositivo di memoria di massa basato su semiconduttore, che utilizza memoria allo stato solido (in particolare memoria flash) per l'archiviazione dei dati. Le unità allo stato solido si basano su memoria flash solitamente di tipo NAND per l'immagazzinamento dei dati. Essi non richiedono parti meccaniche in movimento (dischi, motori e testine), né componenti magnetici, il che comporta notevoli vantaggi alla riduzione dei consumi elettrici e dell'usura. Il controller è costituito da un microprocessore che si occupa di coordinare tutte le operazioni della memoria di massa. Il software che governa questo componente è un firmware preinstallato dal produttore. Gli SSD inoltre possiedono una memoria cache di qualche MB o GB a seconda del tipo di sistema, generalmente proporzionale alla capienza dell'SSD, utilizzata dal processore per immagazzinare temporaneamente informazioni che verranno richieste in seguito dal sistema.

Vantaggi:

- Rumorosità assente, non essendo presente alcun componente di rotazione, al contrario degli HDD tradizionali.
- Minore possibilità di rottura: le unità a stato solido hanno mediamente un tasso di rottura inferiore a quelli dei dischi rigidi.
- Minori consumi elettrici durante le operazioni di lettura e scrittura.
- Tempi di accesso e archiviazione ridotti: si lavora nell'ordine dei decimi di millisecondo, il tempo di accesso dei dischi magnetici è oltre 50 volte maggiore, attestandosi invece tra i 5 e i 10 ms.
- Non necessitano di deframmentazione.
- Maggiore velocità di trasferimento dati.
- Maggiore resistenza agli urti.
- Minore produzione di calore.

Svantaggi:

A fronte di una maggiore resistenza agli urti e a un minor consumo, le unità a stato solido hanno due svantaggi principali:

- Maggiore prezzo.
- Peggiore permanenza dei dati quando non alimentati e in modo differente a seconda della temperatura d'esposizione.

Prestazioni di un SSD (Comando TRIM e Write Amplification): Un elemento che viene immediatamente alla luce analizzando le prestazioni di un dispositivo SSD è la minor velocità in scrittura rispetto a quella in lettura e la sua forte variabilità in dipendenza della dimensione dei file che si vogliono scrivere. Ciò dipende dal fatto che mentre i File System dei sistemi operativi solitamente usano blocchi di celle dalla dimensione di 4 KiB, nei dispositivi SSD la dimensione dei blocchi è molto superiore (per esempio 4 MiB).

Questo comporta che per scrivere una cella dobbiamo leggere prima l'intero blocco, quindi scrivere sopra la cella desiderata lasciando le altre inalterate e infine salvarlo. Ne deriva che se dobbiamo scrivere più celle (file più grandi) le prestazioni migliorano, perché a fronte della lettura e poi del salvataggio di un blocco, possiamo scriverci dentro contemporaneamente tante celle quanto sono quelle libere disponibili.

Un modo per migliorare le prestazioni è quello di conoscere i blocchi liberi (con nessuna cella utilizzata); per ottenere ciò i sistemi operativi di ultima generazione mettono a disposizione il comando TRIM, che comunica al controller dell'SSD quali blocchi sono inutilizzati e cancellano le celle direttamente in fase di cancellazione dei file, migliorando dunque le prestazioni.

12. File System

Il File System è un meccanismo con il quale i file sono posizionati e organizzati su un dispositivo di archiviazione. Un File System, più nello specifico è l'insieme dei meccanismi che permettono l'organizzazione gerarchica, la manipolazione, la lettura e scrittura dei dati. I dispositivi di archiviazione si presentano al sistema operativo come array di blocchi di dimensione fissa, (*tipicamente di 512 byte l'uno*). Le operazioni disponibili sono la lettura e la scrittura di un blocco arbitrario, o talvolta di un insieme di blocchi. Basandosi su queste operazioni di base fornite dai dispositivi a blocchi, il file system realizza un meccanismo di astrazione, che rende le risorse di memorizzazione di massa facilmente utilizzabili dagli utenti.

Iniziamo con delle premesse, elencando alcune necessità dei processi e cosa ne consegue.

- I processi devono poter memorizzare e trattare grandi quantità di dati (maggiori della quantità di memoria principale.)
- Più processi devono avere la possibilità di accedere alle informazioni in modo concorrente e coerente, nello spazio e nel tempo.
- Si deve garantire integrità, indipendenza, persistenza e protezione dei dati.

Ne consegue che l'accesso diretto ai dispositivi di memorizzazione non è sufficiente a garantire queste necessità. Da qui nasce la soluzione che porta alla nascita del "file".

➤ Files & Metadata

Un file è un insieme di informazioni correlate, a cui viene assegnato un nome. Un file è la più piccola porzione unitaria di memoria logica secondaria allocabile dall'utente o dai processi di sistema. Il S.O., mediante il File System realizza questa astrazione, nascondendo i dettagli implementativi legati ai dispositivi sottostanti. Esteriormente, il file system, è spesso la parte più visibile di un sistema operativo (nei s.o. "documentocentrici"). Internamente, il file system si appoggia alla gestione dell' I/O per implementare ulteriori funzionalità.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Ogni file ha degli attributi, chiamati METADATA:

Nome: identificatore del file. L'unica informazione human readable.

Tipo: presente nei sistemi che supportano più tipi di file.

Locazione: puntatore alla posizione del file sui dispositivi di memorizzazione.

Dimensioni: indica l'attuale dimensione ed eventualmente quella massima consentita.

Protezioni: controllo dei permessi di lettura, modifica, creazione ed esecuzione del file.

Identificatori dell'utente: possessore e creatore del file.

Date: di creazione, modifica, etc.

I metadati (dati sui dati) sono solitamente mantenuti in apposite strutture dette directory, residenti in memoria secondaria.

I file sono un meccanismo di astrazione, quindi ogni oggetto necessita di un nome che viene associato dall'utente ed è solitamente necessario (ma non sufficiente) per accedere ai dati del file.

Le regole per denominare i file sono fissate dal file system, e sono molto variabili: supportano varie lunghezze, tipologie di caratteri, inoltre possono essere case sensitive, insensitive o preserving etc. etc. (Variano da tipologia a tipologia di f.s.).

Esempi di tipologie dei file:

Tipo	Estensione	Funzione
Esegibile	exe, com, bin o nessuno	programma pronto da eseguire, in linguaggio macchina
Oggetto	obj, o	compilato, in linguaggio macchina, non linkato
Codice sorgente	c, p, pas, f77, asm, java	codice sorgente in diversi linguaggi
Batch	bat, sh	script per l'interprete comandi
Testo	txt, doc	documenti, testo
Word processor	wp, tex, doc	svariati formati
Librerie	lib, a, so, dll	library di routine
Grafica	ps, dvi, gif	FILE ASCII o binari
Archivi	arc, zip, tar	file correlati, raggruppati in un file, a volte compressi

In genere, un file è una sequenza di bit, byte, linee o record il cui significato è assegnato dal creatore. A seconda del tipo, i file possono avere diverse tipi di struttura:

- **Nessuna:** sequenza di parole, byte.
- **Sequenza di record:** linee, blocchi di lunghezza fissa/variabile.
- **Strutture più complesse:** documenti formattati, archivi (ad albero, con chiavi, . . .).
- **Eseguibili rilocabili:** (ELF, COFF), i file strutturati possono essere implementati con quelli non strutturati, inserendo appropriati caratteri di controllo.

A imporre la struttura dei file è il sistema operativo, oppure l'utente:

- **S.O:** specificato il tipo, viene imposta la struttura e modalità di accesso. Più astratto.
- **Utente:** tipo e struttura sono delegati al programma, il sistema operativo implementa solo file non strutturati.

Principali operazioni sui file: Segue una lista delle principali operazioni base sui file che un SO tipicamente implementa:

- **Creazione:** due passaggi: allocazione dello spazio sul dispositivo, e collegamento di tale spazio al file system.
- **Cancellazione:** due passaggi: "staccare" il file dal file system e deallocare lo spazio assegnato al file.
- **Apertura:** caricare alcuni metadati dal disco nella memoria principale, per velocizzare le chiamate seguenti.
- **Chiusura:** deallocate le strutture allocate all'apertura.
- **Lettura:** dato un file e un puntatore di posizione, i dati da leggere vengono trasferiti dal media in un buffer in memoria.
- **Scrittura:** dato un file e un puntatore di posizione, i dati da scrivere vengono trasferiti sul media.
- **Riposizionamento (seek):** non comporta operazioni di I/O, sposta il puntatore al file.
- **Troncamento:** azzerare la lunghezza di un file, mantenendo tutti gli altri attributi.
- **Concatenazione:** append: scrittura in "fondo" al file.
- **Lettura dei metadati:** leggere le informazioni come nome, timestamp, ecc.
- **Scrittura dei metadati:** modificare informazioni come nome, timestamp, protezione, ecc

Queste operazioni richiedono la conoscenza delle informazioni contenute nelle directory. Per evitare di accedere continuamente ad esse, si mantiene in memoria una tabella dei file aperti. Si ri-definiscono così due operazioni sui file per permettere questo meccanismo:

- **Apertura:** allocazione di una struttura in memoria (file descriptor o file control block) contenente le informazioni riguardanti un file.
- **Chiusura:** trasferimento di ogni dato in memoria al dispositivo, e deallocazione del file descriptor/file control block.

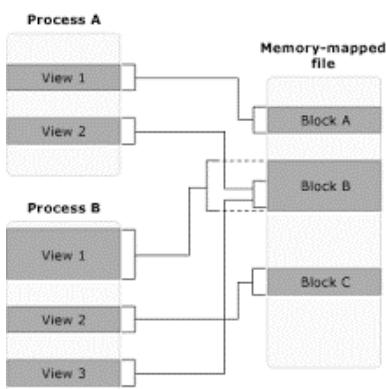
A ciascun file aperto si associa:

- **Puntatore al file:** memorizza la posizione raggiunta durante la lettura/scrittura.
- **Contatore dei file aperti:** tiene conto di quanti processi stanno utilizzando il file.
- **Posizione su disco:** informazioni riguardo la posizione del file sul disco.

Ne consegue che esistono diversi metodi di accesso ai file:

- **Sequenziale:** Un puntatore mantiene la posizione corrente di lettura/scrittura. Si può accedere solo progressivamente, o riportare il puntatore all'inizio del file. Operazioni: *read next, write next, reset, rewrite*. Adatto a dispositivi intrinsecamente sequenziali.
- **Diretto:** Il puntatore può essere spostato in qualunque punto del file. Operazioni: *read n, write n, seek n, read next, write next, rewrite n* con *n* posizione relativa a quella attuale. L'accesso sequenziale viene simulato con l'accesso diretto. Usuale per i file residenti su device a blocchi.
- **Indicizzato:** Un secondo file contiene solo parte dei dati, e puntatori ai blocchi (record) del vero file. La ricerca avviene prima sull'indice (corto), e da qui si risale al blocco. Implementabile a livello applicazione in termini di file ad accesso diretto. Usuale su mainframe e database.

Differenze fra accesso sequenziale ed accesso diretto: Con la modalità di accesso sequenziale un puntatore mantiene la posizione corrente di lettura/scrittura, consentendo di accedere ai dati soltanto progressivamente o riportando il puntatore all'inizio del file. Le tipiche operazioni che un sistema operativo deve implementare in tal senso sono: *read next, write next, reset*. Un tipico esempio di periferiche intrinsecamente sequenziali sono i nastri (tipicamente usati per funzionalità di backup). Invece, con la modalità di accesso diretto (detto anche random), il puntatore può essere spostato in qualunque punto del file. Le tipiche operazioni che un sistema operativo deve implementare in questo caso sono: *read n, write n, seek n*, (e *read next, write next, rewrite n*), dove *n* rappresenta la posizione relativa a quella attuale. L'accesso diretto permette di simulare l'accesso sequenziale ed è particolarmente indicato per i dispositivi a blocchi come i dischi.



File mappati in memoria: Mappare i file in memoria semplifica l'accesso ai file, rendendoli simili alla gestione della memoria. Risulta essere un meccanismo relativamente semplice da implementare in sistemi segmentati (con o senza paginazione): il file viene visto come area di swap per il segmento mappato. Non servono chiamate di sistema *read* e *write*, solo una *mmap* (memory map). Un file mappato in memoria include il contenuto di un file nella memoria virtuale. Questo mapping tra un file e uno spazio di memoria consente a un'applicazione, inclusi più processi, di modificare il file leggendo e scrivendo direttamente nella memoria.

Problemi: lunghezza del file non nota al sistema operativo, accesso condiviso con modalità diverse, lunghezza del file maggiore rispetto alla dimensione massima dei segmenti.

• Directory

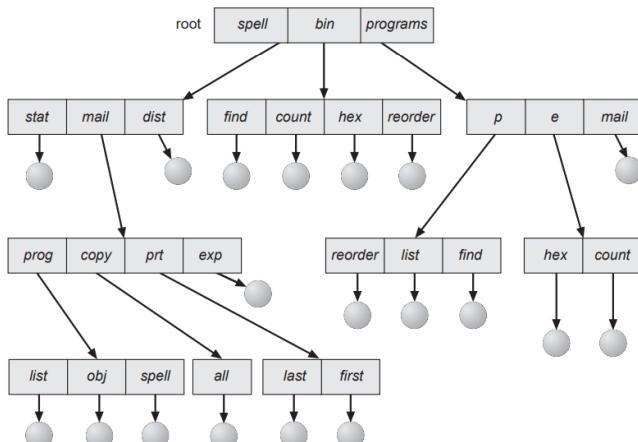
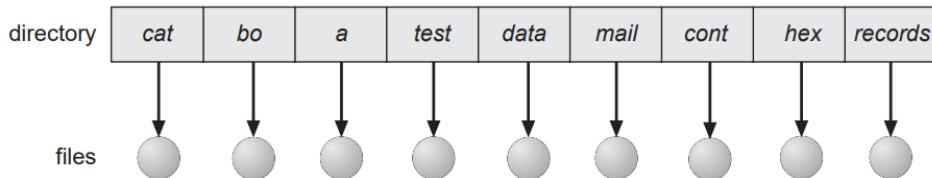
Una directory è una collezione di nodi contenente informazioni sui file (metadati). Sia la directory che i file risiedono su disco. Segue una lista di operazioni base eseguibili su una directory:

- Ricerca di un file.
- Creazione di un file.
- Cancellazione di un file.
- Listing dei file.
- Rinomina di un file.
- Navigazione del file system.

Le directory devono essere organizzate in modo da ottenere un sistema efficiente in fase di ricerca dei file. La directory deve inoltre permettere di localizzare e gestire i file con nomi mnemonici: comodi per l'utente (file differenti possono avere lo stesso nome o più nomi possono essere dati allo stesso file). Inoltre una directory si occupa di organizzare un meccanismo di raggruppamento: file logicamente collegati devono essere raccolti assieme (e.g., i programmi in C, i giochi, i file di un database, etc.). Esistono più tipologie di directory.

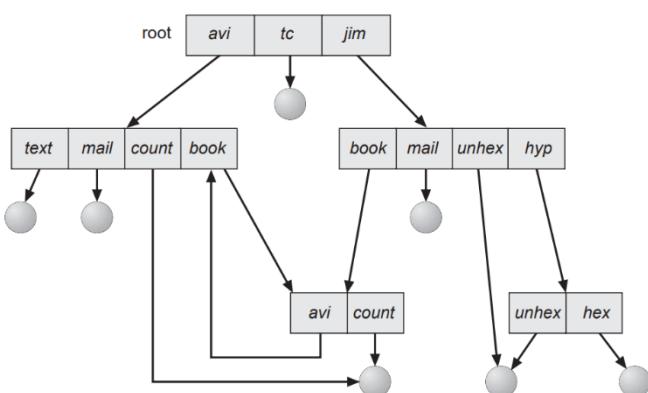
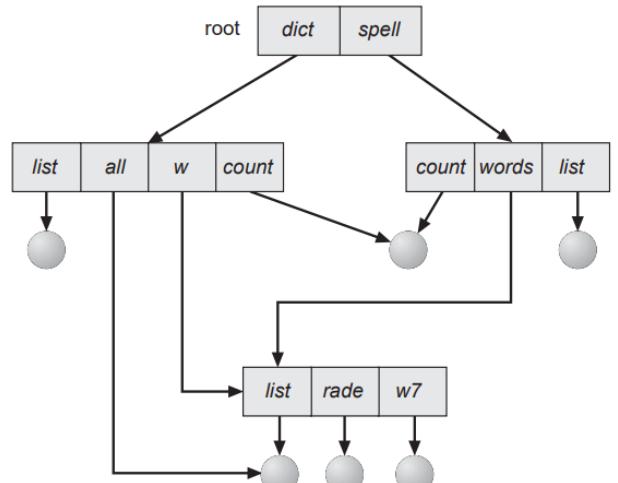
Tipologie di directory:

- **Unica (Flat):** Una sola directory condivisa da tutti gli utenti. Presenta problemi di raggruppamento e denominazione. *Variante:* un livello per ciascun utente. Obsoleta e rudimentale. Facile da implementare.



- **Ad Albero:** Le directory ad albero offrono elevata efficienza nelle operazioni di ricerca. Permettono il raggruppamento e la presenza di nomi assoluti o relativi. I processi possono avere una proprietà che indica la directory corrente su cui operano (*working directory*). Le operazioni su file e directory eseguite da un processo sono quindi relative alla directory corrente.

- **A grafo aciclico (DAG):** File e sottodirectory possono essere condivise da più directory. Due nomi differenti per lo stesso file (*aliasing*). Possibilità di puntatori "dangling". Soluzioni:
 - Puntatori all'indietro.
 - Puntatory daisy chain.
 - Contatori di puntatori per ogni file (UNIX).



- **A Grafo:** La presenza di cicli può portare a diverse problematiche: Algoritmi di visita costosi per evitare loop infiniti. Cancellazione costosa per via di creazione di garbage (parti di dati non indicizzati dalla directory). Per risolvere questo problema è possibile:
 - Limitare il numero di nodi attraversabili.
 - Permettere solo link ai file.

Protezione (Access Control List)

object domain \	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

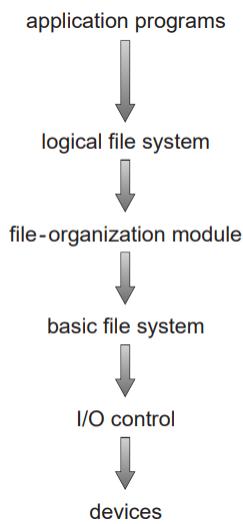
La protezione dei file è importante in ambienti multiuser dove si vuole condividere i file. Il creatore/possessore (non sempre coincidono) deve essere in grado di controllare cosa può essere fatto e da chi (in un sistema multutente). Da qui la necessità di definire quali tipologie di accesso ai file sono soggette a controllo: *Read, Write, Execute, Append, Delete, List*.

Soluzione generale: Matrici di accesso che per ogni coppia (processo,oggetto), associano le operazionimesse.

13. Implementazione del File System

I dispositivi tipici su cui viene implementato un file system sono i dischi meccanici o gli SSD. Il motivo è semplice: sono dispositivi a blocchi che permettono l'accesso diretto su tutta la superficie, sia in lettura che in scrittura.

Struttura gerarchica dei FileSystem:



- **Programmi applicativi.**
- **File System logico:** Presenta i diversi file system come un'unica struttura. Implementa i controlli di protezione.
- **Organizzazione dei file:** Controlla l'allocazione dei blocchi fisici e la loro corrispondenza con quelli logici. Effettua la traduzione da indirizzi logici a fisici.
- **File System di base:** Usa i driver per accedere ai blocchi fisici sull'appropriato dispositivo.
- **Controllo dell' I/O:** Driver dei dispositivi.
- **Dispositivi:** I controller hardware dei dischi, dispositivi di memoria secondaria.

Tabella dei file aperti (citata nel cap. precedente):

Per accedere ad un file è necessario conoscere informazioni riguardo la sua posizione, protezione etc.. Questi dati sono accessibili attraverso la directory. Per evitare continui e ripetuti accessi al disco, si mantiene in memoria una tabella dei file aperti, in cui ciascun suo elemento descrive un file aperto (file control block).

- Alla prima open di un file si caricano in memoria i metadati relativi al file aperto.
- Ogni operazione viene effettuata riferendosi al file control block in memoria.
- Quando il file viene chiuso da tutti i processi che vi accedevano, le informazioni vengono copiate su disco e il blocco deallocated.

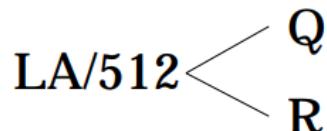
Questo approccio porta a problemi di affidabilità: Nel caso di mancata corrente? le informazioni presenti sul file control block potrebbero non essere state salvate su disco.

Mounting del file system: Ogni file system fisico, prima di essere utilizzabile, deve essere “montato” nel file system logico. Il montaggio può avvenire al boot, o dinamicamente (supporti usb, remoti). Il punto di montaggio può essere fissato oppure configurabile in qualsiasi punto del file system logico (ad es. su Unix).

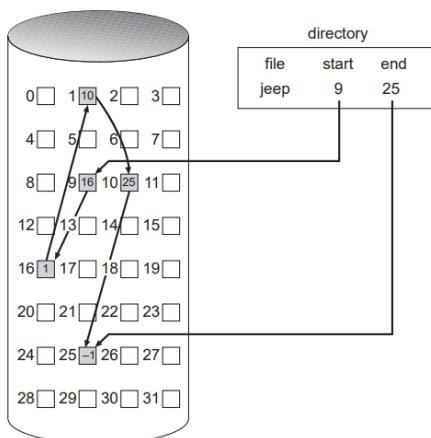
Il Kernel esamina il file system fisico per riconoscerne la struttura ed il tipo. Prima di spegnere o rimuovere il dispositivo, il file system deve essere smontato (pena il rischio di gravi inconsistenze).

- **Allocazione dei file:** I file possono essere allocati nel disco mediante i meccanismi forniti dal FileSystem in diversi modi, l'allocazione dei file può quindi essere:

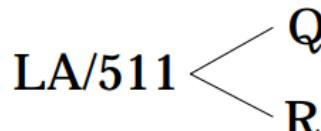
- **Contigua:** Ogni file occupa un insieme di blocchi contigui sul disco. Questa tecnica è facile da implementare ma porta a frammentazione esterna e interna. Traduzione dall'indirizzo logico a quello fisico (blocchi da 512 byte):



(Blocco da accedere = Q +Blocco di partenza, Offset all'interno del blocco = R)



- **Concatenata:** Ogni file è una linked list di blocchi, che possono essere sparpagliati ovunque sul disco. L'allocazione avviene su richiesta, i blocchi vengono semplicemente collegati alla fine del file. Non presenta frammentazione esterna e non supporta l'accesso diretto (seek). Traduzione indirizzo logico:

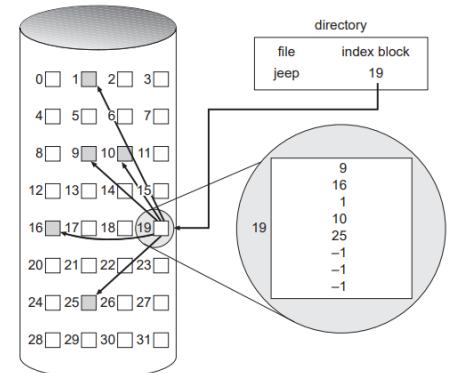
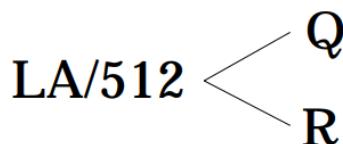


(Q è il blocco da accedere, Offset nel blocco = $R+1$)

Variante: File-allocation table (FAT) di MS-DOS e Windows. Mantiene la linked list in una struttura dedicata, all'inizio di ogni partizione.

- **Indicizzata:** Si mantengono tutti i puntatori ai blocchi di un file in una tabella indice. Supporta l'accesso random e implementa allocazione dinamica senza frammentazione esterna.

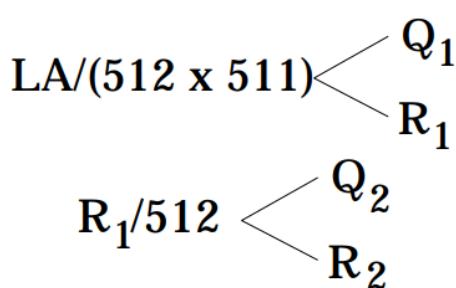
Traduzione di un indirizzo logico:



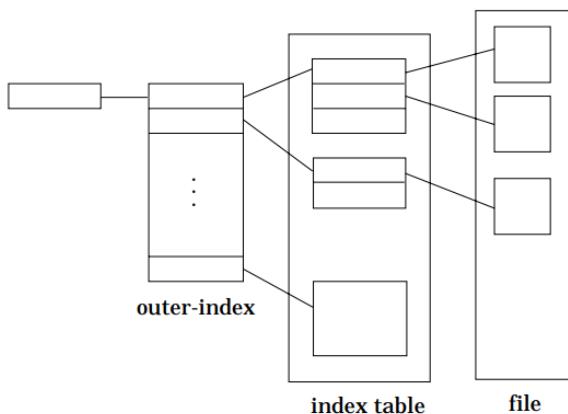
Problema dell'allocazione indicizzata: Come implementare il blocco

indice? È una struttura supplementare. Dobbiamo supportare anche file grandi. L'indice concatenato: L'indice è composto da blocchi concatenati. Nessun limite sulla lunghezza, maggior costo di accesso.

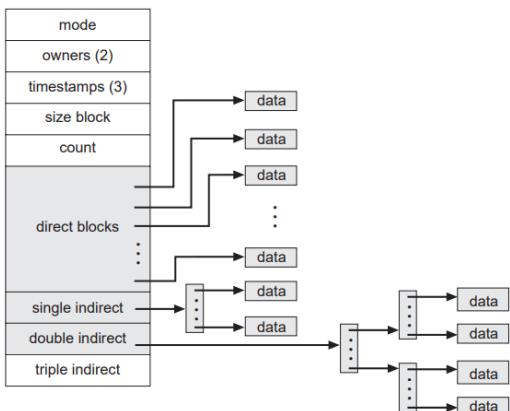
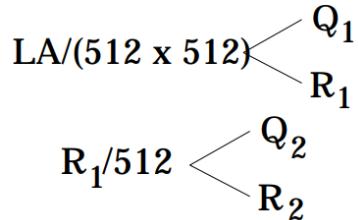
(Q_1 = blocco dell'indice da accedere Q_2 = offset all'interno del blocco dell'indice R_1 = offset all'interno del blocco del file).



- **Indicizzata a più livelli:** Indice a due (o più) livelli.



(Q_1 = offset nell'indice esterno Q_2 = offset nel blocco della tabella indice
 R_2 = offset nel blocco del file).



Inodes (UNIX): Un file in Unix è rappresentato da un inode (index node). Gli inode sono allocati in numero finito alla creazione del file system.

Gestione dello spazio libero: I blocchi non utilizzati sono indicati da una lista di blocchi liberi.

010111010101011111011000000101000000010110101011

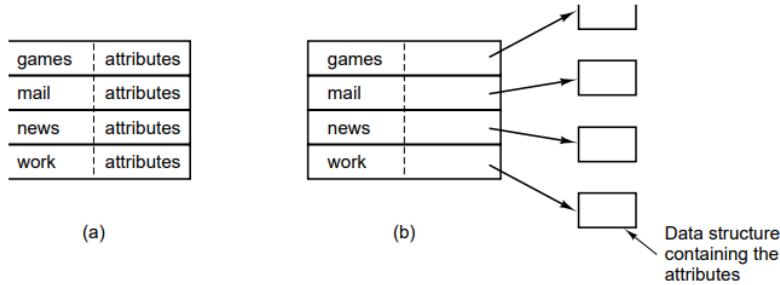
$$\text{bit}[i] = \begin{cases} 0 & \Rightarrow \text{block}[i] \text{ occupato} \\ 1 & \Rightarrow \text{block}[i] \text{ libero} \end{cases}$$

Usare un vettore di bit (block map) per indicare blocchi liberi o non liberi: 1 bit per ogni blocco. È una soluzione comoda e veloce a livello di operazioni assembler. La bit map però consuma spazio.

Un'alternativa è implementare una lista concatenata (linked list) dei blocchi liberi. Questa occuperà soltanto lo spazio strettamente necessario (proporzionale al numero dei blocchi liberi), ma sarà molto inefficiente per la ricerca di blocchi liberi contigui (in quanto tale operazione potrebbe richiedere di scorrere gran parte della lista).

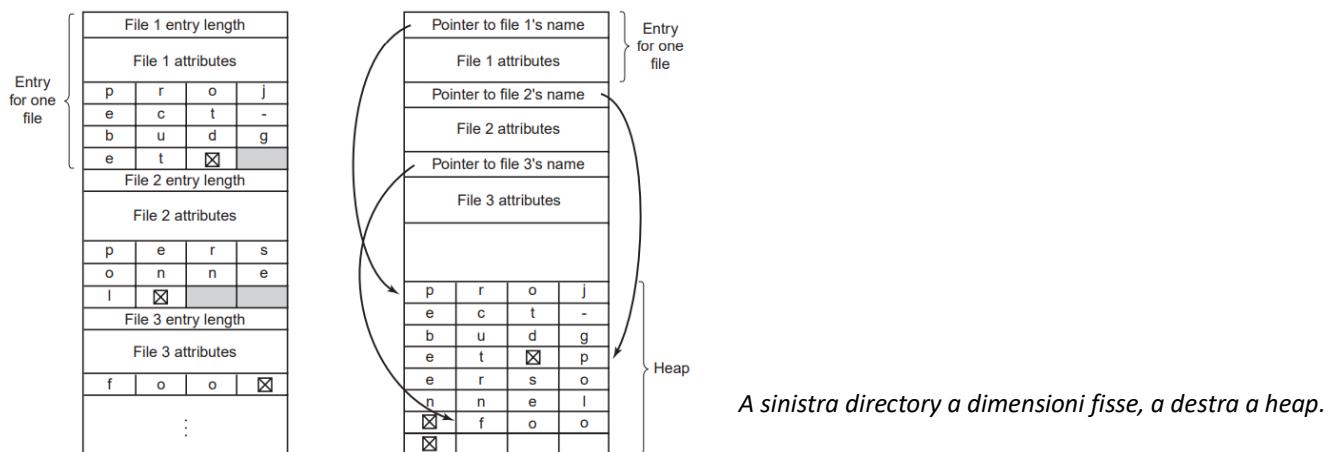
➤ Implementazione delle directory

Le directory sono essenziali per passare dal nome del file ai suoi attributi e dati. Gli attributi risiedono nella directory stessa o in strutture esterne.



a) Gli attributi risiedono nelle entry stesse della directory (MS-DOS, Windows).

b) Gli attributi risiedono in strutture esterne (es. inodes), e nelle directory ci sono solo i puntatori a tali strutture (UNIX).

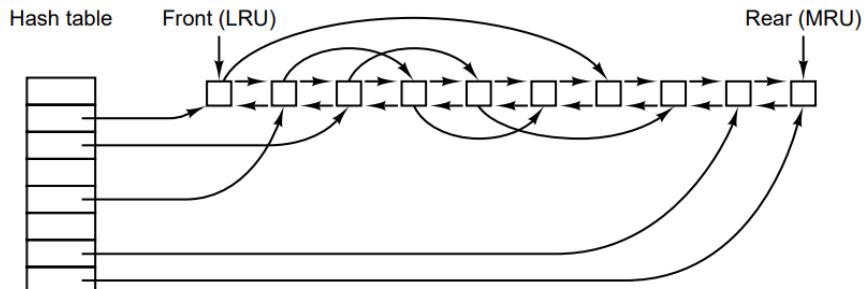


Altri tipi di directory:

- **Lista lineare** di file names con puntatori ai blocchi dati:
Semplice da implementare.
Lenta nella ricerca, inserimento e cancellazione di file.
Può essere migliorata mettendo le directory in cache in memoria.
- **Tabella hash**: lista lineare con una struttura hash per l'accesso veloce:
Si entra nella hash con il nome del file
Abbassa i tempi di accesso.
Bisogna gestire le collisioni: ad es., ogni entry è una lista
- **B-tree**: Generalizzazione di un albero di ricerca binario:
Ricerca, accesso, inserimenti e cancellazioni in tempi logaritmici.
Abbassa i tempi di accesso.
Bisogna mantenere il bilanciamento.

L'efficienza e le performance di una directory dipendono principalmente dagli algoritmi di allocazione spazio disco, dai tipi di dati contenuti nella directory, e dalla grandezza dei blocchi (piccoli per aumentare l'efficienza, causando meno frammm. interna. Grandi per aumentare le performance).

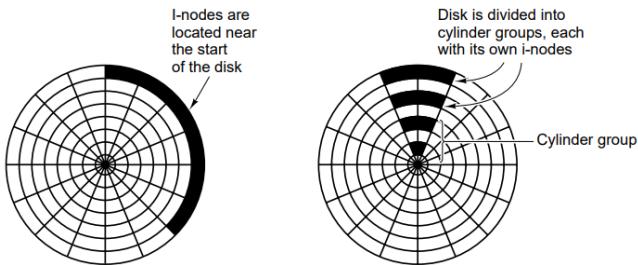
Caching: Consiste nell'utilizzo di memoria principale per bufferizzare i blocchi più usati. Può essere implementata sul controller del disco come buffer di traccia per ridurre la latenza o sulla memoria principale, prelevando pagine dalla free list. I buffer sono organizzati in una coda con accesso hash, che viene gestita LRU, Clock etc. Un blocco viene salvato su disco quando deve essere liberato dalla coda. Se blocchi critici non vengono salvati a causa di un crash dopo aver ricevuto modifiche si rischiano gravi inconsistentezie sui file.



La variante LRU consiste nel suddividere i blocchi in categorie a seconda che il blocco verrà usato a breve (posizionato in fondo alla lista) o il blocco è critico per la consistenza del file system, allora ogni sua modifica viene immediatamente scritta su disco.

Altri accorgimenti sull'aumento di performance:

Read-Ahead: Leggere blocchi in cache prima che siano realmente richiesti. Aumenta il throughput del device. Molto adatto per file che vengono letti in modo sequenziali. Inadatto per file ad accesso casuale. Il file system può tener traccia del modo di accesso dei file per migliorare le scelte



Riduzione movimento del disco: Durante la scrittura di un file, sistemare vicini i blocchi a cui si accede di seguito può aiutare a ridurre il movimento della testina del disco.

Si può ad esempio raggruppare (e leggere) i blocchi in gruppi (detti cluster). Collocare i blocchi con i metadati presso i rispettivi dati.

Affidabilità del File System

L'affidabilità del file system è fortemente legata a quella dei dispositivi di memoria di massa, che hanno relativamente un MTBF molto breve. Inoltre i crash di sistema possono essere causa di perdita di dati in cache non ancora trasferite nel disco. Esistono dunque due tipologie di affidabilità quando si parla di un file system:

- Affidabilità dei dati: Avere la certezza che i dati salvati possano venir recuperati.
- Affidabilità dei metadati: Garantire che i metadati non vadano perduti/alterati.

Perdere dati è costoso, ma lo è ancor di più la perdita dei metadati: Può portare alla completa inconsistenza del file system, che può causare la perdita di tutti i dati. È possibile aumentare l'affidabilità dei dati mettendo in atto tecniche di RAID o eseguendo backup.