

# Introduzione:

Il sistema operativo è quella componente dei calcolatori che ne permette l'utilizzo da parte degli utenti in modo completamente trasparente. Un sistema elaborativo complesso come i moderni Pc o i cellulari si compone, come si fede in figura, di vari livelli che partendo dai dispositivi fisici arrivano all'utente finale

Si parte dai **dispositivi fisici**, ovvero le componenti elettroniche del calcolatore vero e proprio, salendo attraverso la **micropogrammazione** che mi consente di programmare le operazioni della CPU attraverso il suo **linguaggio macchina** (NON esportabile)

Il **sistema operativo** vero e proprio che nasconde la complessità dei livelli inferiori e fornisce al programmatore un insieme di istruzioni di alto livello (*KERNEL MODE*) che si differenziano da quello che è il software di sistema (*USER MODE*). I programmi si dividono in **programmi di base** (ovvero i programmi minimi a disposizione all'acquisto della macchina) e **programmi applicativi** (ovvero esterni al sw di base)



Figura 1: Schema base di un calcolatore

Un moderno calcolatore comprende:

- **CPU**

E' l'elemento che riceve effettivamente le istruzioni da eseguire e le esegue

- **Memoria centrale**

Nel senso della memoria in riferimento all'architettura di Von Neumann

- **Cache**

Memoria intermedia trasparente all'utente utilizzata dal s.o. per interazioni veloci

- **Clock**

E' il sincro delle operazioni effettuate dal sistema, esiste anche senza alimentazione esterna tramite batteria

- **Terminali**

Possono fare info riguardo al sistema stesso e permettono l'interazione attraverso comandi testuali con il s.o.

- **Dischi**

Conservano i dati e lo stato del sistema

- **Interfacce di rete**

- **I/O testuale, multimediale e altri sistemi**

Permettono la gestione da parte dell'utente di tutte le operazioni, dalle più basilari alle più avanzate, del s.o.

Esistono, in ordine cronologico di sviluppo dal meno recente, i seguenti tipi di sistemi operativi:

- *Dedicati*

- *A lotti*

- *Multiprogrammati*

- *Time-Sharing*

- *Quarta Gen.*

- *Quinta Gen.*

# Evoluzione dei S.O.

A partire dagli inizi del calcolo automatizzato si sono susseguiti vari tipi di sistemi operativi, prima di giungere ai moderni;

## Prima Gen: 1945-55:

Sono stati i primissimi esempi di calcolo automatizzato ed erano sistemi estremamente costosi dedicati quali esclusivamente al calcolo scientifico. Era presente un unico staff per progetto, costruzione, programmazione, esecuzione etc e tutti i programmi erano scritti in **linguaggio macchina**(NON assembly) Vi era inoltre un obbligo di controllo integrità delle valvole e una totale assenza di sistemi operativi (soltanto negli ultimi anni iniziarono a comparire le prime *schede perforate*

## Seconda Gen. : 1955-65:

Con la seconda generazione di s.o. si assiste all'introduzione dei s.o. **a lotti** determinata dall'introduzione dell'innovazione tecnologica dovuta ai transistor e dalla necessità di risolvere i problemi dovuti ai tempi morti tra i programmi (vedi *job*). Anche in questo caso si tratta di macchine prevalentemente dedicate a studi e università, con la differenza che possono essere prodotti e acquistati da clienti che ne fanno richiesta

### Job:

sono i *programmi* di questa generazione; un programmatore scrive il programma su carta (Fortran, Assembly) e perfora le schede. Le riceve un operatore che le inserisce in una coda dei programmi, le immette nel calcolatore e se necessario carica il compilatore opportuno. L'operatore poi consegna al programmatore l'output finale.

Il problema principale di questa generazione è lo spreco di risorse dovuto al **setup-time**, ovvero il tempo necessario al caricamento sulla macchina dei nastri e del compilatore e al loro scaricamento una volta letti i risultati. La soluzione a questo problema di spreco di risorse è l'utilizzo di **s.o. batch**

### Batch:

Nati per sfruttare al meglio la velocità messa a disposizione dal calcolatore eliminando i tempi morti tra programmi successivi di utenti differenti, automatizzando il più possibile le operazioni manuali. Un insieme di lavori (*jobs*) viene accorpato in un lotto (*batch*) tramite un calcolatore ausiliario e trasferito su una unità di ingresso veloce come può essere un nastro

Ogni lavoro, formato da piu' batch, viene caricato da un operatore ed eseguito in sequenza senza interruzione fino al termine. L'output viene scritto su un altro nastro invece di essere stampato, risparmiando anche sui tempi di stampa (notevolmente più lenti di quelli di scrittura di un nastro) Nonostante ciò la CPU viene comunque sottoutilizzata poichè durante le operazioni di I/O deve adeguarsi alla bassa velocità delle periferiche.

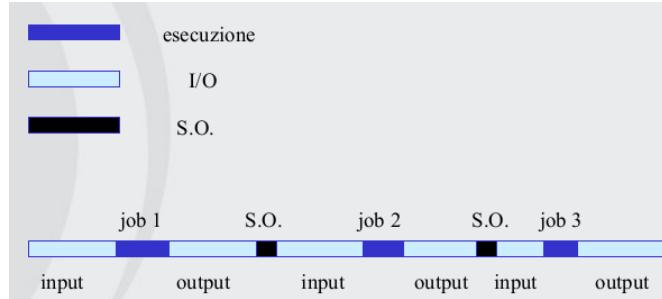


Figura 2: Utilizzo delle risorse in s.o. **batch**

## Terza Gen: 1965-80

È la generazione che vede l'introduzione dei s.o. multiprogrammati grazie all'avvento dei *circuiti integrati*. Questo consente di velocizzare le operazioni di I/O e eseguire queste ultime mentre è in esecuzione un altro programma grazie all'introduzione del **canale di gestione I/O**.

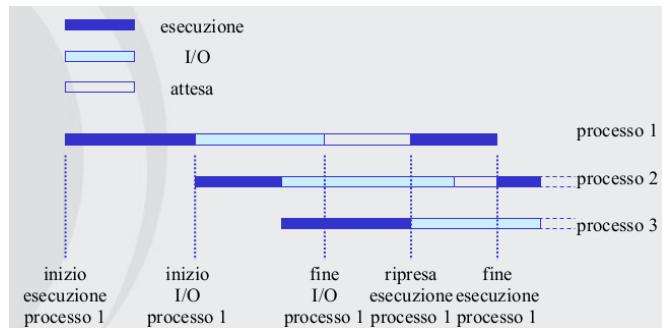


Figura 3: Utilizzo delle risorse in s.o. **multiprogrammati**

Il difetto di questi s.o. è che il multitasking **non** è in grado di distinguere tra programmi che richiedono un uso frequente delle periferiche di I/O rispetto a quelli che richiedono spesso l'intervento della CPU senza necessità di periferiche: questo porta ad avere il rischio che un programma che ottiene la CPU non la rilascia spontaneamente, andando a bloccare di fatto gli altri programmi

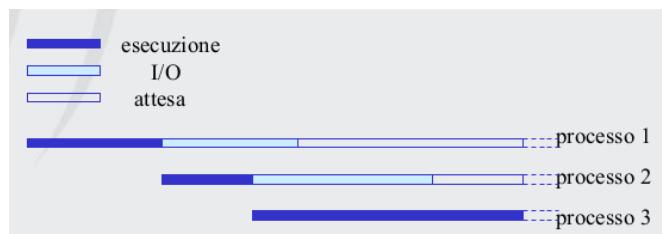


Figura 4: **stallo** dovuto alla non perfetta gestione delle risorse

## Sistemi Time Sharing

Sono sistemi operativi in cui l'utilizzo della CPU è suddiviso in "fette" di tempo definite *time slice* della durata di 50-500 ms. Ogni processo in memoria riceve a turno l'uso della CPU per una unità di tempo e al termine dello slice il controllo torna comunque al sistema operativo che decide a chi affidare la CPU. Sono sistemi operativi utilizzati per lo sviluppo software in ambiente multiutente, con l'utente singolo che ha l'impressione di avere la macchina a sua completa disposizione (con time slice molto piccoli o numerosi processi in esecuzione l'overhead può diventare significativo).

A differenza dei sistemi a lotti l'obiettivo non è massimizzare l'uso del processore ma minimizzare i tempi di risposta, utilizzando invece che il linguaggio Job Control dei comandi da terminale

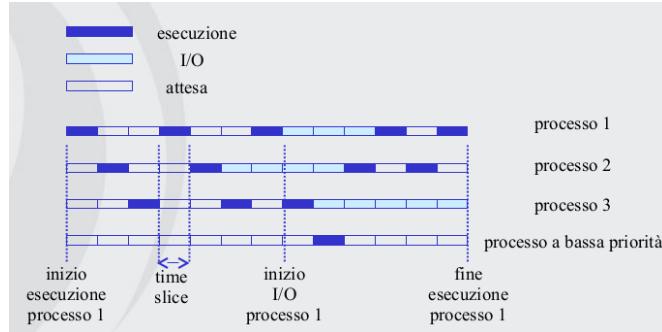


Figura 5: gestione delle risorse di un s.o. timesharing

## Quarta Gen: 1980

Sono sistemi operativi nati con la diffusione dei transistor su larga scala nell'elettronica, a livello di wafer di silicio, e si basano sulla **Very Large Scale Integration**. Le caratteristiche di tali sistemi operativi hanno dato il via allo sviluppo dei personal computer e delle workstation, in particolare con l'utilizzo della CPU che non è più critico (un solo utente) e senza concorrenza d'utilizzo (almeno nei primi esemplari)

## Quinta Gen: 1990

Sono i sistemi più moderni, da cui sono nati anche i sistemi operativi mobile e i telefoni cellulari. Essi tuttavia hanno riportato alla luce problemi non di facile gestione come:

- Memoria limitata
- Batteria
- Schermo piccolo

## Reti

Con lo sviluppo sempre più capillare di sistemi operativi mobile e di dispositivi portatili si è venuta a creare la necessità di connettere i calcolatori tra di loro attraverso **la rete**. Nascono pertanto i **network operating systems**, ovvero sistemi dove l'utente vede più calcolatori e può accedere a macchine remote, con ogni macchina che ha il suo sistema operativo locale

Vengono inoltre sviluppati i **Distributed operating systems**: sono sistemi che appaiono all'utente come un tradizionale sistema monoprocesso anche se composto da più processori grazie ai quali l'operazione da eseguire viene ripartita a carico degli N processori, secondo due possibili scenari; il primo in cui gli N processori *non* condividono il clock e la memoria e un secondo caso in cui essi condividono il clock e la memoria.

## Sistemi Real Time:

Sono sistemi operativi al servizio di una specifica applicazione che ha vincoli sui tempi di risposta di un sistema alla variazione di uno o più parametri misurati. In generale si ha un sistema *real time* quando il tempo di risposta dalla richiesta di esecuzione di un processo al completamento della stessa è sempre minore del tempo prefissato. Eventi di trigger possono essere:

- Gestione di strumentazione
- Controllo processo
- Gestione di allarme
- etc...

# Struttura di un Sistema Operativo

## Gestione dei processi:

Un **processo** è un qualunque programma in esecuzione, sia esso in background o foreground; esso ha bisognosi alcune risorse come tempo di CPU, una memoria, file ed eventualmente dispositivi di I/O.

La responsabilità del sistema operativo è quella di assegnare ai processi le risorse necessarie, creare e distruggere gli stessi processi, sospendere e ripristinarli se necessario e soprattutto mettere a disposizione meccanismi per la sincronizzazione e la comunicazione di processi per fare in modo che possano comunicare tra di loro e condividere le informazioni necessarie alla loro esecuzione

## Memoria centrale:

E' condivisa tra CPU e dispositivi I/O ed è l'unico dispositivo di memorizzazione che la CPU è in grado di indirizzare direttamente; le istruzioni possono essere eseguite dalla CPU *solo se si trovano effettivamente in memoria*. Per ovviare ai vincoli dovuti alla memoria un calcolatore deve necessariamente gestire più programmi in memoria, attraverso diversi schemi di gestione della stessa che dipendono dall'implementazione e dall'hw disponibile.

## Memoria secondaria:

il sistema operativo è responsabile di gestire lo spazio libero, allocare lo spazio necessario e fare lo scheduling del disco; dato che l'utilizzo del disco è molto frequente la gestione deve essere efficiente affinchè non venga introdotto un overhead eccessivo nell'utilizzo della CPU per continue letture e scritture in memoria secondaria.

## Gestione I/O:

La gestione delle periferiche di I/O da parte del s.o. ha lo scopo principale di rendere il sistema operativo trasparente all'utente, attraverso il buffer caching e driver specifici per ogni hardware. SOLO il driver conosce le caratteristiche del dispositivo a cui è assegnato

## Gestione dei file:

Il s.o. è responsabile della gestione di file e in generale dei supporti con cui lavora; deve essere in grado di gestire nastri, dischi magnetici, ottici etc e il tutto in relazione alla capacità e velocità di ciascuno. Il sistema operativo ha una visione logica uniforme del processo di memorizzazione dell'informazione → esiste solo un tipo di oggetto.

Il file è la sola unità logica di memoria e il sistema operativo deve essere in grado di creare e cancellare i file, creare e cancellare le directory, supportare primitive per la manipolazione di file e directory, mappare file sulla memoria ed eventualmente eseguire backup dei file

## Reti:

Il sistema operativo deve gestire anche eventuali **sistemi distribuiti** (ovvero insieme di unità che non condividono la memoria, i dispositivi periferici o un clock. Queste unità comunicano tra loro attraverso la rete, utilizzando protocolli specificati. Dal punto di vista dell'utente un sistema distribuito è un insieme di sistemi fisicamente separati organizzato in modo tale da apparire come un unico sistema coerente.

L'accesso al sistema distribuito permette di accelerare il calcolo, aumentare la disponibilità di dati e incrementare l'affidabilità.

### Gestione delle protezioni:

Un sistema operativo con più utenti consente che i processi vengano eseguiti in **concorrenza**; ciò rende necessario che i processi siano protetti dalle attività di altri processi e occorrono quindi *meccanismi di autenticazione*

**PROTEZIONE:** è il meccanismo di controllo dell'accesso da parte di programmi, processi o utenti alle risorse del sistema di calcolo.

### Interprete dei comandi:

L'interfaccia tra utente e sistema può avere diversi aspetti:

- Istruzioni di controllo
- Shell
- Interfaccia grafica

e il suo compito è quello di accettare comandi da parte dell'utente per l'esecuzione

## Il Sistema Operativo

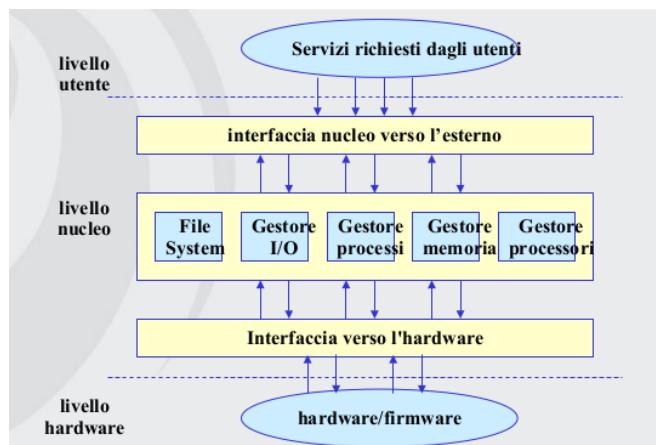


Figura 6: Un s.o. è composto da **nucleo (o kernel)** + **interfaccia con l'utente**

### System Call:

L'interfaccia tra il sistema operativo e i programmi degli utenti è definita da un insieme di istruzioni estese definite **system call**, ovvero chiamate a sistema. Esse creano, cancellano e utilizzano gli oggetti software gestiti dal sistema operativo stesso.

La loro implementazione può essere differente a seconda del sistema operativo preso in considerazione. Esistono varie tipologie di chiamate a sistema:

- Controllo dei processi
- Gestione dei file
- Gestione dei dispositivi
- Gestione dell'informazione
- Comunicazioni

I programmi dell'utente comunicano con il sistema operativo e gli richiedono servizi per mezzo delle chiamate di sistema; ad ogni systemcall corrisponde una **procedura di libreria** (che il programma può invocare) che ha i seguenti compiti:

- mettere i parametri della systemcall in un sito predefinito (e.g. registro)
- istanziare un'istruzione TRAP per mandare un interrupt al sistema operativo
- nascondere i dettagli dell'istruzione TRAP (eccezione)
- rendere le systemcall come chiamate di procedura normali, ad esempio da programmi in C

UNIX	Win32	Descrizione
fork	CreateProcess	Crea un nuovo processo
waitpid	WaitForSingleObject	Aspetta che un processo termini
execve	[nessuna]	CreateProcess = fork + execve
exit	ExitProcess	Termina l'esecuzione
open	CreateFile	Crea un file o apre un file esistente
close	CloseHandle	Chiude un file
read	ReadFile	Legge dati da un file
write	WriteFile	Scrive dati in un file
lseek	SetFilePointer	Muove il file pointer
stat	GetFileAttributesEx	Restituisce gli attributi di un file
mkdir	CreateDirectory	Crea una nuova directory
rmdir	RemoveDirectory	Rimuove una directory vuota
link	CreateSymbolicLink	Crea i collegamenti
unlink	Deletefile	Elimina un file
mount	[nessuna]	Win32 non supporta il mount
umount	[nessuna]	Win32 non supporta il mount
chdir	SetCurrentDirectory	Cambia la directory di lavoro
chmod	SetFileAttributes	Modifica gli attributi dei file
kill	TerminateProcess	Il comportamento differisce nei due S.O.
time	GetLocalTime	Restituisce data e ora corrente

Figura 7: Elenco delle chiamate a sistema delle due famiglie di s.o. principali

### Programmi di sistema:

I programmi di sistema offrono un ambiente per lo sviluppo e l'esecuzione dei programmi e possono essere classificati in programmi per la gestione dei file, l'informazione di stato, la modifica di file, ambienti di sviluppo, caricamento ed esecuzione programmi, programmi applicativi etc ect

Dal punto di vista dell'utente la maggior parte delle operazioni è *vista come esecuzione di programmi, non come chiamate a sistema*

### Struttura dei sistemi operativi:

I principali tipi di sistemi operativi sono classificabili in:

- Sistemi monolitici
- Sistemi multilivello
- Modello client-server
- Macchine virtuali

## Monolitici:

Il sistema operativo è costituito da un insieme di procedure ognuna delle quali può chiamare qualsiasi altra; l'unica struttura presente sono le *system call* che comportano il salvataggio dei parametri e l'esecuzione di una TRAP speciale, detta **kernel call** (o **supervisor call**) utilizzata per causare un interrupt per richiedere un servizio al sistema operativo. Una chiamata a sistema è svolta attraverso una serie di

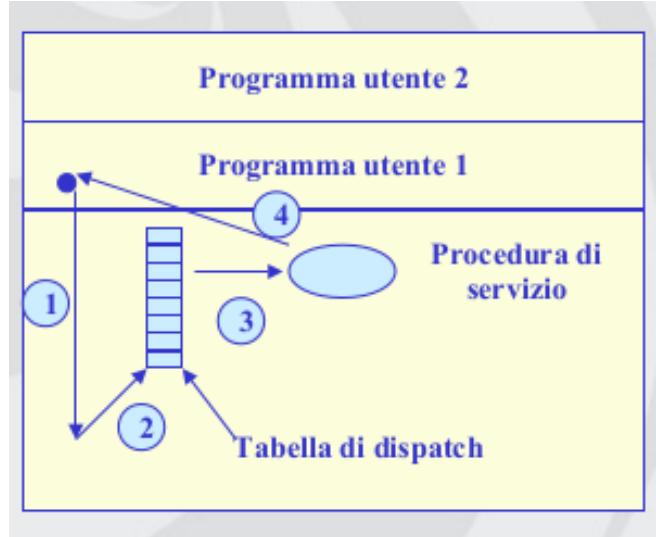


Figura 8: Schema di una chiamata di sistema

passaggi definiti:

1. il programma utente esegue una TRAP verso il kernel
2. il sistema operativo determina il numero del servizio richiesto
3. il sistema operativo individua e chiama la procedura di servizio
4. viene restituito il controllo al programma utente

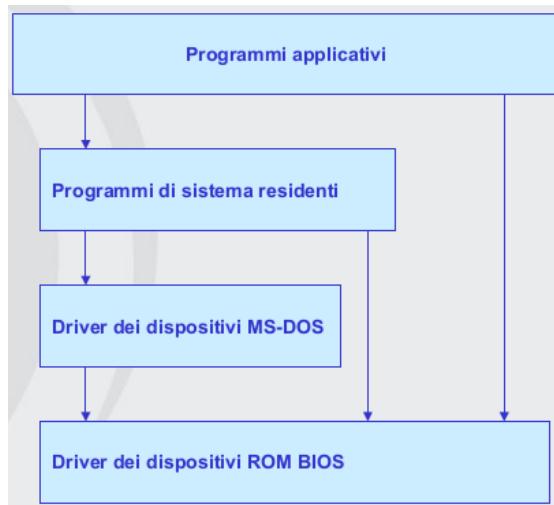


Figura 9: Struttura di MS-DOS, dove non esisteva distinzione tra le modalità e tutti i programmi potevano accedere a tutte le risorse

## Duplice modo di funzionamento:

I sistemi che condiviscono le risorse devono garantire che i programmi non corretti non interferiscano con gli altri processi in esecuzione, pertanto l'hardware (*mode bit*) permette al sistema di operare in due modalità differenti:

- **User mode** → normale funzionamento dei programmi utente, con un numero relativamente ristretto di permessi verso hardware, memoria e risorse in generale; le applicazioni generalmente girano in modalità utente e possono passare in kernel mode a seconda delle necessità
- **Monitor mode (o Kernel mode/System mode)** → operazioni effettuate dal sistema operativo (si indica solitamente lo stato di privilegio massimo, riservato all'esecuzione del kernel)

Quando avviene un interrupt si passa al kernel mode.

### Protezione dell'I/O:

Alcune istruzioni (**privileged instructions**) possono essere eseguite *solo* in kernel mode. Rientrano in questa categoria ad esempio tutte le istruzioni di I/O; occorre tuttavia garantire che nessun programma possa essere eseguito in kernel mode, per evitare possibili danni al sistema operativo

### Struttura UNIX:



Figura 10: Struttura di Unix

La versione originale rientrava nella categoria di sistemi operativi con tutti i file all'interno del nucleo (**sistema monolitico**). Le differenze vengono fatte dalla chiamate a sistema.

### Strati e livelli:

Per semplificare la gestione del sistema, sia dal punto di vista dello sviluppo che della manutenzione, si è resa necessaria la divisione del sistema in strati.

### Struttura a tre strati:

Un programma principale che richiama la procedura di servizio richiesta, con un **insieme di procedure di servizio** che eseguono le chiamate di sistema e un insieme di **procedure di utilità** che forniscono il supporto alle procedure di servizio

### Sistemi multilivello:

Il sistema operativo è organizzato in una gerarchia di livelli (primo sistema con una struttura di questo tipo è stato il THE, realizzato in Olanda con 6 livelli) Esso aveva 32k di parole a 27 bit ed un tamburo di 512k parole usato per memorizzare parti di un programma) Sistemi multilivello sono stati introdotti anche per motivi di sicurezza, al fine di proteggere i dati tra i vari utenti e i vari processi. In un sistema multilivello è più semplice gestire le priorità tra i vari livelli e i vari utenti

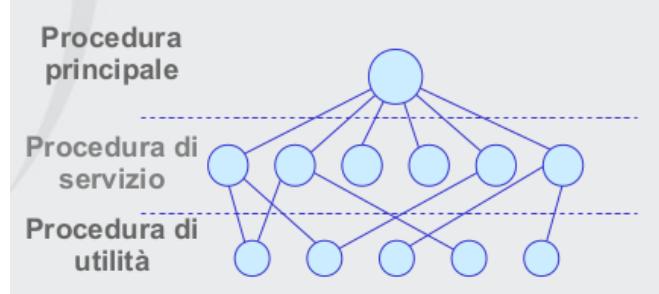


Figura 11: Struttura a tre strati

Strato	Funzionalità
5	Operatore
4	Programmi Utente
3	Gestione I/O
2	Comunicazione processo-console
1	Gestione della memoria e del tamburo
0	Allocazione della CPU e multiprogrammazione

Figura 12: Schema a livelli del THE; come si vede dal lievlo 0 si tratta di un sistema multiprogrammato

## Sistema a Microkernel:

Per limitare i problemi e i bachi più gravi si è puntato sull'avere il nucleo più piccolo possibile, per avere codice ridotto più semplice da controllare (in questo modo si possono individuare meglio le criticità e risolvere il prima possibile)

Per ridurre a microkernel si affidano a processi in esecuzione nello spazio utente *tutto* ciò che non è strettamente necessario all'interno dello spazio kernel (così facendo se avviene qualche bug questo blocca soltanto il processo e non l'intero sistema) Un altro vantaggio è la possibilità di aggiungere nuovi processi e funzionalità :andare a modificare un kernel monolitico è più complesso e rischioso, mentre un microkernel per l'aggiunta di funzionalità a livello di spazio utente non viene modificato. Questo rende anche il microkernel anche estremamente più portabile rispetto al monolitico e più affidabile. Lo **svantaggio** principale è che viene introdotto un overhead dovuto alle comunicazioni tra spazio utente e kernel

Sistemi di questo tipo sono basati sul modello di **scambio di messaggi**: una chiamata di sistema è vedibile dal punto di vista logico come un messaggio che viene mandato da un processo ad un processo differente. In questo caso pertanto il kernel non è altro che un intermediario dei messaggi tra processi, attraverso la comunicazione di un indirizzo di memoria dove si trovano le chiamate, le istruzioni o i messaggi.

## Client-Server:

Un altro modello utilizzabile si basa sull'idea di portare codice ai livelli superiori, lasciando il kernel al minimo possibile. L'approccio è quello di implementare la maggior parte delle funzioni di sistema operativo nei processi utente:

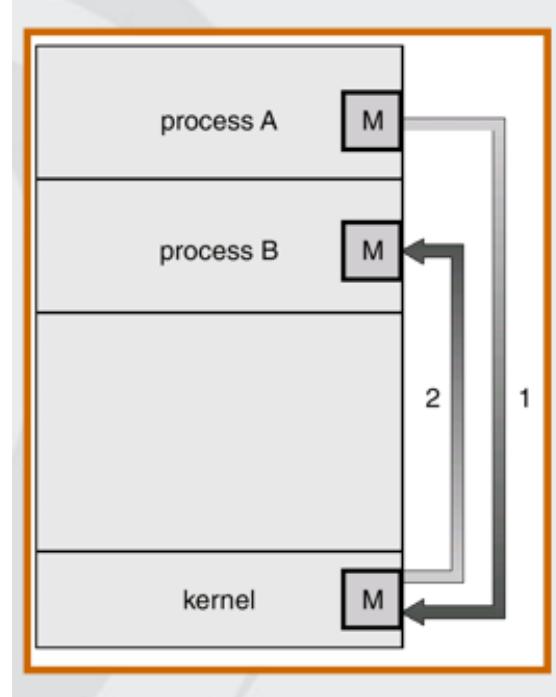


Figura 13: Struttura a microkernel

- **Processo client:** richiede un servizio
- **Processo server:** gestisce le richieste del client
- **Kernel:** gestisce la comunicazione tra client server

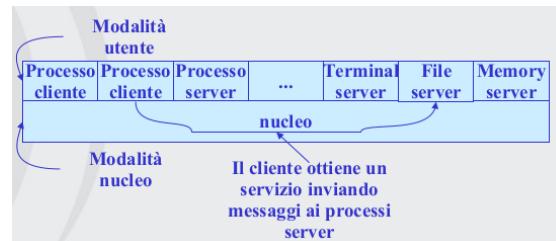


Figura 14: Struttura di un sistema client-server

Il sistema operativo viene suddiviso in parti più piccole e maneggevoli, ognuna delle quali si riferisce ad un aspetto del sistema (memoria, file...)

Uno dei primi sistemi di questa tipologia è stato **windows NT**. I vantaggi principali dell'approccio **client-server**

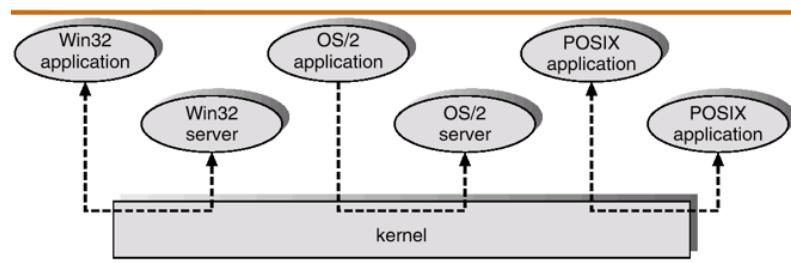


Figura 15: Struttura di windowsNT

**server** è la possibilità di creare sistemi distribuiti, dove ogni macchina ha il suo nucleo che comunica con quelli delle altre macchine. Alcune funzioni di sistema operativo non sono eseguibili da programmi nello spazio utente, pertanto si rendono necessarie delle soluzioni:

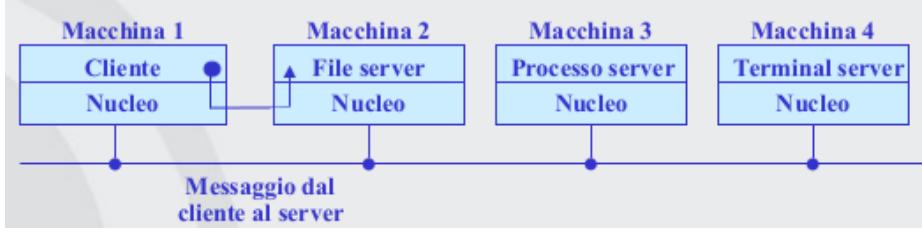


Figura 16: Struttura base di sistemi distribuiti

- processi server critici vengono eseguiti in modalità kernel
- divisione tra *meccanismo*, minimo e implementato nel kernel, e *decisioni di strategia*, lasciate ai processi server nello spazio utente

## Moduli:

Derivata dalla programmazione ad oggetti, è una struttura che implementa un approccio modulare al kernel; in questa tipologia di approccio ogni componente è separata e comunica con le altre attraverso interfacce definite. Il tutto può essere caricato a sistema avviato (approccio simile ai livelli ma più flessibile). Il vantaggio di avere i moduli separati è che se viene a cadere un modulo non viene intaccato tutto il sistema ma soltanto il modulo e ciò che ad esso è associato (in generale il sistema continua ad essere operativo). Lo scambio di informazioni tra i vari moduli deve essere ben definito attraverso interfacce che descrivano come vengono scambiati informazioni tra i moduli e il kernel.

## Macchine virtuali:

Il primo sistema si basa sulla separazione netta tra le funzioni di multiprogrammazione e di implementazione della macchina estesa. Il monitor della macchina virtuale gestisce la multiprogrammazione fornendo più macchine virtuali; Ogni macchina virtuale è una copia esatta dell'hardware e diverse macchine virtuali possono supportare sistemi operativi diversi. **CMS**, ovvero Conversational Monitor System ,è un sistema interattivo per utenti time sharing dove ogni chiamata viene spezzata in due parti: sistema operativo e utente. I *vantaggi* sono la semplicità di ogni parte, più flessibile e facile da mantenere; ogni macchina virtuale permette una completa protezione delle risorse di sistema e rende facile lo sviluppo di un sistema operativo (le normali operazioni non vengono bloccate). Lo *svantaggio* principale è che l'isolamento delle macchine virtuali rende complessa la condivisione delle risorse

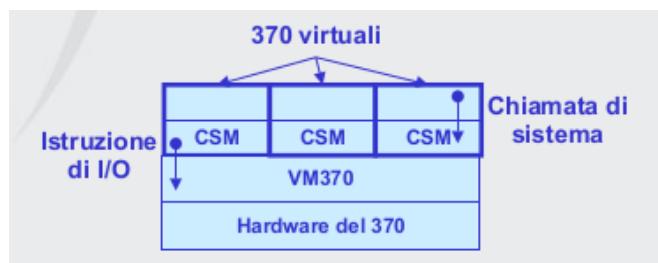


Figura 17: Struttura di una vm con CMS

## Modalità 8086:

Un approccio simile è utilizzato nei sistemi Windows con processore Pentium: esso può funzionare in modalità virtuale 8086 cosicchè la macchina si comporti effettivamente come processore 8086 per eseguire vecchi programmi MS-Dos (finchè eseguono istruzioni normali lavorano direttamente sull'hardware della macchina, nel momento in cui fanno richieste al sistema operativo o istruzioni dirette di I/O l'istruzione viene intercettata dalla macchina virtuale).

## Hypervisor:

conosciuto anche come **Virtual Machine Monitor**, deve controllare al di sopra di ogni sistema virtualizzato, senza pesare sulle prestazioni, come avviene la virtualizzazione. Deve inoltre permettere il debug e il monitoring delle attività dei sistemi operativi e delle applicazioni, in modo da scoprire eventuali malfunzionamenti ed intervenire il prima possibile.

L'hypervisor può allocare le risorse dinamicamente quando e dove necessario, ridurre in modo drastico il tempo necessario alla messa in opera di nuovi sistemi, isolare l'architettura nel suo complesso da problemi a livello di sistema operativo ed applicativo, abilitare ad una gestione più semplice di risorse eterogenee e, come già accennato, facilitare collaudo e debugging di ambienti controllati. Esistono due tipologie di hypervisor:

### Tipo 1:

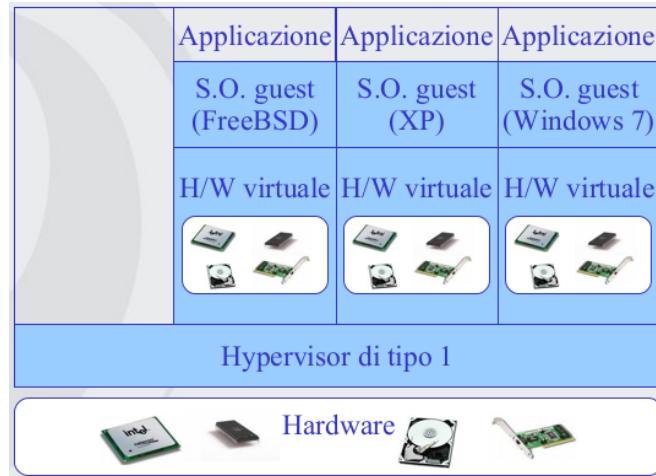


Figura 18: Hypervisor tipo 1

Tipo 1 o (**Nativo**) sistemi che si appoggiano direttamente all'hardware e consentono l'installazione dei sistemi guest su un livello superiore

### Tipo 2:

Tipo 2 o **Hosted**: software che si appoggiano ad un sottostante sistema operativo. In pratica sono delle normali applicazioni. Esempi sono la Java Virtual Machine e il Common Language Runtime di .NET. Per la virtualizzazione vera e propria è frequente il tipo Ibrido nel quale il sistema operativo host è allo stesso livello delle macchine virtuali

## Requisiti per la virtualizzazione:

Le istruzioni vengono divise in tre gruppi:

- **privilegiate**: generano una trap solo se si è in user mode
- **controllo**: modificano le risorse di sistema, e.g. I/O

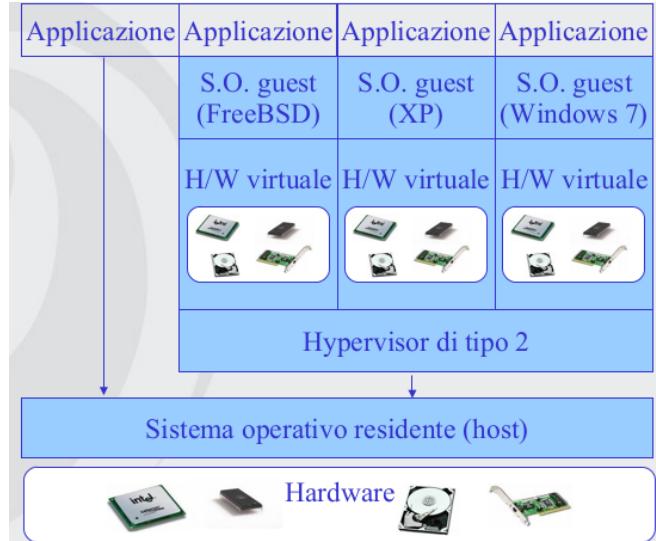


Figura 19: Hypervisor tipo 2

- **dipendenti** dalla configurazione del sistema

Le istruzioni di controllo devono inoltre essere un sottinsieme delle istruzioni privilegiate (e tutte le istruzioni che possono influenzare l'esecuzione del virtual machine monitor devono generare una trap ed essere gestite dal vmm stesso. Le istruzioni non privilegiate possono essere eseguite nativamente.

#### **Requisiti:**

I requisiti per la virtualizzazione sono i requisiti di **Goldberg** e **Popek**:

1. Ambiente di esecuzione per i programmi sostanzialmente identico a quello della macchina reale.
2. Garantire un'elevata efficienza nell'esecuzione dei programmi
3. Garantire la stabilità e la sicurezza dell'intero sistema

Esistono varie tipologie di virtualizzazione:

- **Emulazione:** permette l'esecuzione di un SO su una CPU completamente differente
- **Virtualizzazione piena:** esegue copie di SO completi
- **Paravirtualizzazione:** il sistema operativo ospitato deve essere modificato

## **Binary rewriting:**

Vengono esaminate le istruzioni del flusso del programma a runtime e vengono individuate le istruzioni privilegiate; vengono poi riscritte queste istruzioni con le versioni emulate e permettono la virtualizzazione in spazio utente. E' tuttavia un sistema lento e si deve usare il caching delle locazioni di memoria (le prestazioni tipiche vanno dall'80% al 97% di una macchina non virtualizzata. Può essere implementato con meccanismi di debugging come i breakpoint.

## **Paravirtualizzazione:**

Se un'istruzione del sistema guest genera una trap, questo ne deve gestire le conseguenze e il sistema guest deve essere modificato. Concettualmente è simile al binary rewriting, ma il rewriting avviene a tempo di compilazione. Quando un'applicazione genera una syscall questa viene intercettata dal sistema operativo guest; quando il guest prova ad eseguire istruzioni privilegiate il VMM intrappola l'operazione e le esegue correttamente. Per interagire con le risorse del sistema i so guest effettuano delle *Hypercall*

## JVM:

E' un calcolatore astratto che ben rappresenta un hypervisor del secondo tipo; esso consiste di un **caricatore delle classi**, un **verificatore delle classi** (*controlla gli accessi alla memoria*) e un **interprete**.

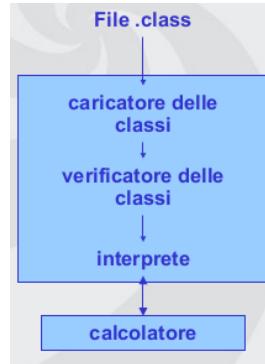


Figura 20: Struttura JVM

Java compila il proprio codice sorgente in un codice intermedio detto **bytecode**, indipendente dalla piattaforma e dell'hw su cui deve essere eseguito. E' la JVM (*dipendente dalla piattaforma*) attraverso meccanismi di virtualizzazione ad eseguire il codice precompilato

# Processi:

In ogni sistema operativo la CPU continua a switchare i compiti eseguiti eseguendo centinaia di compiti differenti al secondo. E' necessario che sia permesso ad un utente di creare processi e interlacciare l'esecuzione di diversi processi per massimizzare l'utilizzo del processore, allocando le risorse necessarie e permettendo la comunicazione tra processi.

**Processo:** è il concetto principale di ogni Sistema Operativo ed è un'astrazione di un programma in esecuzione (*NB: non è l'eseguibile*).

Nei moderni sistemi si parla di **pseudoparallelismo**: questo per indicare il rapido cambio tra programmi da parte della CPU, in modo completamente trasparente all'utente; tecnicamente ogni CPU esegue un solo comando alla volta, ma con l'impressione di eseguire più processi ogni istante.

## Modello del processo:

Se ho soltanto una CPU il program counter è unico e ad ogni cambio di contesto deve essere salvato il PC del processo in esecuzione, copiato nel PC quello del processo successivo ed eseguito e in seguito ripristinato il processo precedentemente salvato. In questo modello tutto il software eseguibile su un

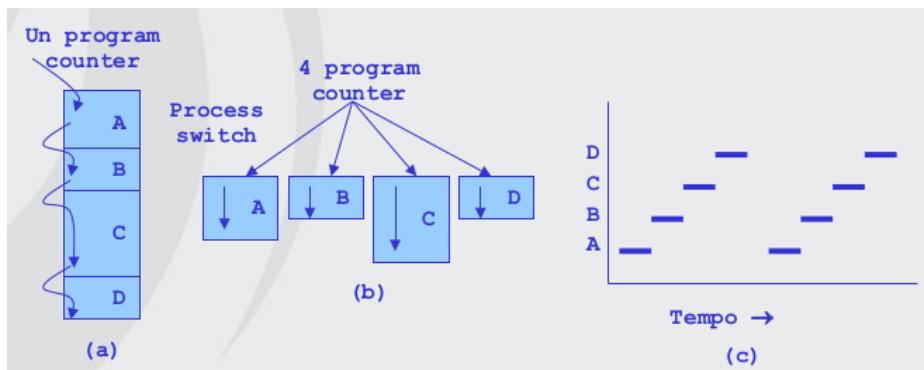


Figura 21: Program counter unico con context switching (a), visto concettualmente come 4 PC (b) ed eseguiti temporalmente (c)

calcolatore è organizzato in un certo numero di **processi sequenziali**. Un processo comprende al suo interno il valore attuale del PC, registri e variabili. Il continuo switch tra processi da parte della CPU è definito **Multiprogrammazione**.

## N.B.

La durata di un processo non è una quantità deterministica stabilita a priori, non è uniforme e soprattutto non è identica ad ogni esecuzione → i processi non devono essere programmati con assunzioni a priori rispetto al tempo (per questo motivo nel caso di processi real time con vincoli stringenti riguardo al tempo di esecuzione nell'ordine dei millisecondi è necessario che vengano adottate delle contromisure per rendere la durata del processo il più deterministica possibile)

## Stati di un processo:

1. Quando un processo in esecuzione chiede un servizio di I/O al sistema operativo va in stato di *blocked* e rimane in attesa del risultato

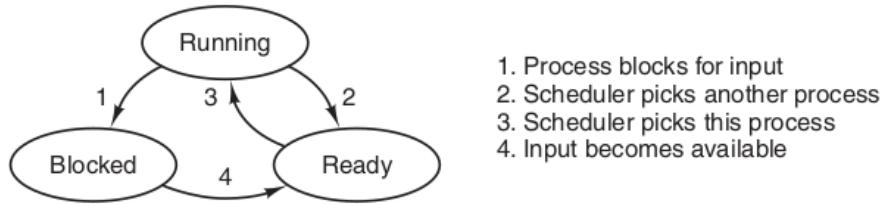


Figura 22: Stati di un processo

2. Al termine del time slice il controllo torna al s.o. e lo scheduler decide a chi affidare la CPU
3. Uno dei processi in attesa ottiene la CPU
4. Quando un processo ha completato le operazioni di I/O viene rimesso dal s.o. nella coda dei processi in attesa

**Coda dei processi in attesa:** E' una coda (FIFO nella forma più semplice) che contiene i processi in *ready* in attesa di essere eseguiti.

Possono talvolta esistere processi che nonostante abbiano terminato la loro esecuzione posseggano ancora un PID e un PCB, necessario per permettere al proprio processo padre di leggere il valore di uscita attraverso la chiamata a sistema `WAIT()`.

**Scheduler:** Con un modello di questo tipo il livello più basso del sistema operativo è lo **scheduler**, gli altri processi sono al di sopra di esso. All'interno dello scheduler sono presenti le gestioni delle interruzioni e i dettagli sulla gestione dei processi.

## Context switch:

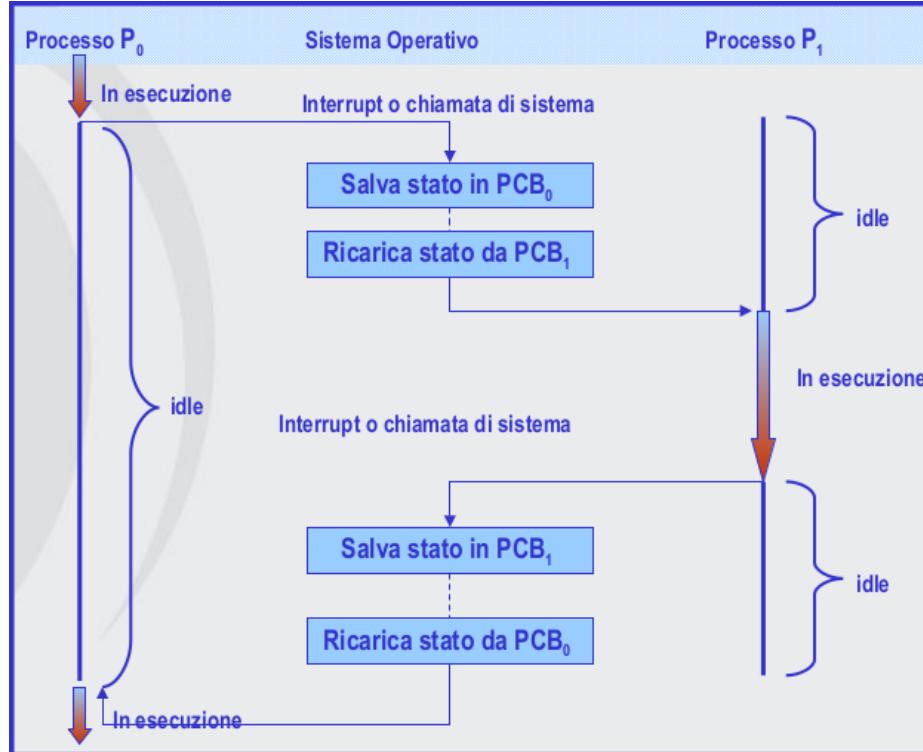


Figura 23: Context switch

Dall'immagine si nota come avvenga il passaggio di processo da un  $P_0$  in esecuzione che alla chiamata di sistema viene salvato in PCB<sub>0</sub>, successivamente viene ricaricato lo stato di PCB<sub>1</sub>, eseguito  $P_1$  per un tempo definito mentre  $P_0$  è in idle e al termine dello slice per  $P_1$  viene eseguita l'operazione opposta per riportare in esecuzione  $P_0$

**PCB:** il Process Control Block è la struttura dati atta a contenere le informazioni di un processo necessarie per la sua gestione. Esse variano da sistema operativo a sistema operativo ma in generale comprendono:

- Program counter
- Area di salvataggio per registri general purpose
- Area per il salvataggio di registri di stato e flag
- Stato corrente di avanzamento
- PID
- Puntatore ai processi figli
- Livello di priorità

Il s.o. ha una **tabella dei processi**, le cui entries sono i PCB dei processi.

## La shell:

Generalmente la shell non fa parte del sistema operativo ma è un interprete di comandi. Alla login viene avviata una shell e il terminale è il dispositivo di I/O che inizia stampando il prompt. All'immissione di un comando la shell crea un processo figlio di se stessa che fa eseguire un programma.

## Processi padri:

Il processo padre crea processi figli che a loro volta possono creare altri processi.

- *Condivisione risorse*: padre e figli possono condividere tutte le risorse, non condividerne alcuna o il figlio può condividere un sottinsieme delle risorse del padre.
- *Esecuzione*: padre e figli sono in esecuzione in modo concorrente oppure il padre attende che i figli terminino l'esecuzione
- *Spazio degli indirizzi*: Il figlio è un duplicato del padre o un nuovo programma

## Fork()

In Unix un processo figlio si genera con il comando *fork()* che duplica l'immagine del padre creando un processo figlio identico; utilizzato dalla shell alla chiamata di un comando, dove viene generato un processo shell figlio che esegue il comando.

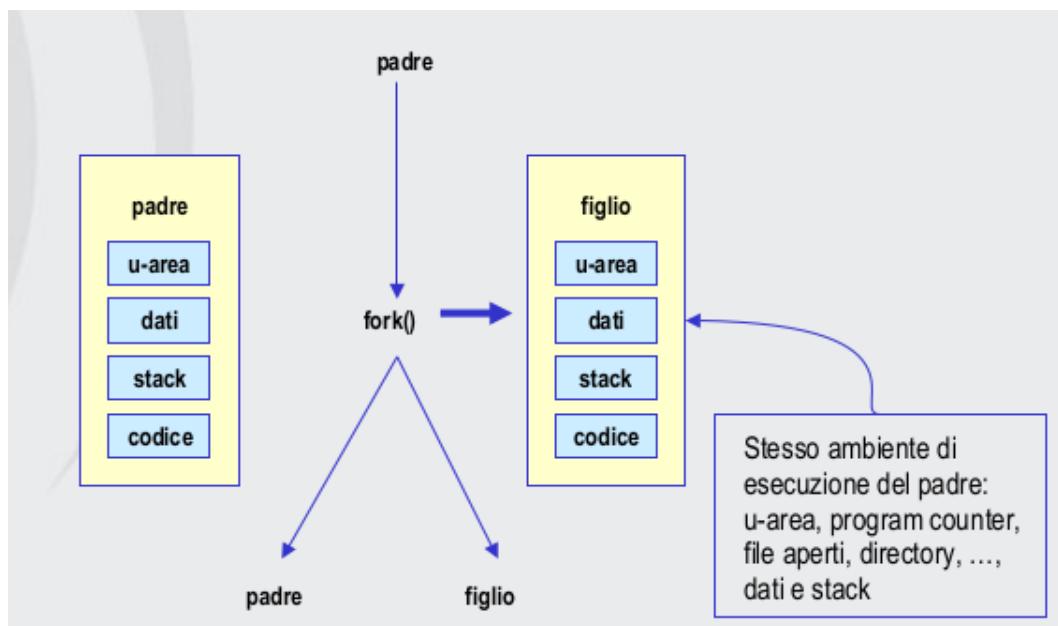


Figura 24: Struttura di un fork

## Memoria e processi:

Un processo Unix è costituito da quattro parti principali che costituiscono la sua immagine:

- **U-Area:** dati relativi al processo di pertinenza del sistema operativo (tabella dei file, working directory...)
- **Dati:** dati globali del processo
- **Stack:** pila del processo
- **Codice:** il codice eseguibile del processo

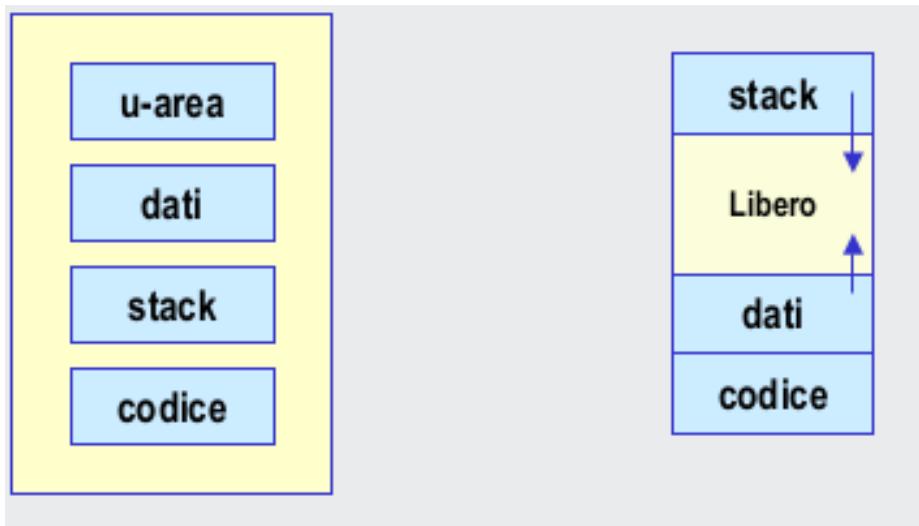


Figura 25: Struttura di processi unix

La *fork()* restituisce il valore 0 al processo figlio e il PID del processo figlio al genitore. Il codice non modificabile viene condiviso tra genitore e figlio. Essi hanno la stessa immagine di memoria, le stesse stringhe d'ambiente e gli stessi file di gestione.

Se si vuole modificare il codice di un processo è necessario chiamare il comando *exec(pathname, args)* che sostituisce l'immagine del chiamante con il file eseguibile pathname e lo manda in esecuzione passandogli gli argomenti (**NB: non crea un nuovo processo**) dove pathname è il percorso per il nuovo eseguibile.

Exec sostituisce i segmenti codice e dati del processo correntemente in esecuzione nello stato di utente con quelli di un altro programma contenuto in un file eseguibile specificato agendo solo su processi in stato di utente (la exec non interviene sul segmento di sistema e sui file utilizzati dal processo che la invoca). Durante la chiamata ad exec e al successivo caricamento nella Ram del nuovo codice compilato il processo mantiene lo stesso pid.

La funzione exec deve passare dei parametri al nuovo programma che viene eseguito. Essi vengono letti dal programma tramite il meccanismo di passaggio usuale argc, argv.

## Creazione di un processo:

Un processo speciale detto INIT viene avviato al boot, legge un file di configurazione ed avvia i terminali. tutti i processi appartengono perciò ad un albero che ha INIT come radice.

## Terminazione:

Il proesso quando ha eseguito la sua ultima istruzione (sia essa exit o nel caso del main la return) avvengono due cose:

- Per mezzo di WAIT alcune informazioni possono essere pasate al processo padre

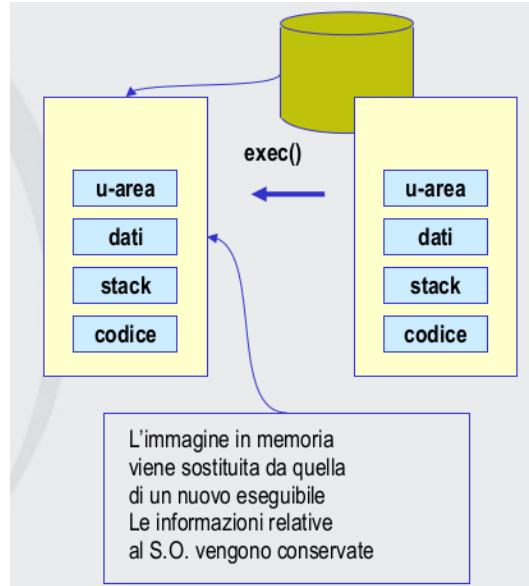


Figura 26: Modello esecutivo della exec()

- Le risorse del processo sono liberate dal sistema operativo

Il processo padre può inoltre porre forzatamente termine ad alcuni processi figli (ABORT) e generare condizioni d'errore se:

- Se il processo figlio a richiesto troppe risorse
- Se il compito affidatogli non è più necessario
- Se il processo padre termina (e.g. un utente si scollega). In ambiente Unix posso lanciare processi in background che sopravvivono al processo che li ha generati e sono gestiti da init, in altri ambienti non possono sopravvivere al padre

Se la terminazione va a buon fine viene ritornato un codice = 0, nel caso in cui invece il processo non sia stato terminato correttamente viene ritornato un valore identificativo dell'errore occorso. **N.B.** la terminazione del main non sempre corrisponde alla terminazione del processo (e.g. le GUI continuano la loro esecuzione anche a main terminato)

### Task Manager:

Mostra quali processi, considerando anche il background, sono attivi nel sistema. I processi che non sono in background sono in pratica le finestre aperte. Offre la possibilità di terminare forzatamente un processo attivo. Mostra anche l'utilizzo di risorse per la singola attività nella lista e le prestazioni dei dispositivi nel complesso. Su sistemi UNIX uso il comando ps o top, che hanno le stesse funzionalità del task manager di Windows

### Sostituzione di codice:

Quando creo un processo modifco in memoria il processo in esecuzione con un altro programma; la sostituzione di codice avviene attraverso EXECVE

*nota:* envp è un vettore con tutte le variaibl di ambiente definito all'avvio del programma, terminato con un puntatore nullo. Nel codice è un puntatore a puntatore. Oltre alle exeve esistono altre primitive:

- **exec1:** indica che i programmi sono elencati in una lista specificata nella chiamata alla funzioine. Con NULL in coda (c'è la possibilità che il pathname sia diverso da argv0 → il programma ha un nome diverso permettendogli di comportarsi in maniera differente).
- **execle:** la -e equivale all'envp, se non la uso allora l'ambiente del processo figlio è lo stesso del processo padre.

```

    • int execve(
        const char *pathname,
        char *const argv[],
        char *const envp[]);
    • esegue il file di pathname specificato, passandogli gli argomenti argv, e l'environment envp

    ○ #include <stdio.h>
    ○
    ○ int main(int argc, char *argv[], char *envp[])
    ○ {
    ○     for( ; *envp; envp++)
    ○         printf("%s\n", *envp);
    ○     return 0;
    ○ }

```

Figura 27: Esempio di sostituzione codice

- **execp**: la -p vuol dire che faccio riferimento alla variabile di shell \$PATH che contiene una lista di directory in cui cercare (nota: su UNIX uso \$nome e separo con ":", mentre su Windows uso %nome% e separo ";"). Questo perchè su UNIX ";" indica la fine di un comando, mentre su windows ":" è un carattere riservato ai Drive.
- **execv**: i parametri non sono su riga di comando ma in un vettore di parametri (numero finito), terminato con NULL
- **execve**: si passa l'informazione in ambiente
- **execvp**: fa riferimento alla \$PATH per cercare l'eseguibile. Il sistema manda in esecuzione la prima versione

Se creo un programma in UNIX posso usare ./nome per fare riferimento ad un programma nella cartella locale in cui sono posizionato. In ambiente Windows invece viene cercato di default nella directory corrente. Si possono creare processi con un numero di parametri arbitrario, ma in realtà il compilatore genera un vettore provvisorio e chiama la creazione del programma a dimensione finita passandogli quel vettore.

In C esiste la funzione SYSTEM(COMANDO) → il comportamento del system dipende dal sistema su cui viene eseguito. Attende che il comando lanciato termini. E.g. Java: Non tutti i comandi di shell

Method Summary	
<a href="#">Process</a>	<a href="#"><code>exec(String command)</code></a> Executes the specified string command in a separate process.
<a href="#">Process</a>	<a href="#"><code>exec(String[] cmdarray)</code></a> Executes the specified command and arguments in a separate process.
<a href="#">Process</a>	<a href="#"><code>exec(String[] cmdarray, String[] envp)</code></a> Executes the specified command and arguments in a separate process with the specified environment
<a href="#">Process</a>	<a href="#"><code>exec(String[] cmdarray, String[] envp, File dir)</code></a> Executes the specified command and arguments in a separate process with the specified environment and working directory.
<a href="#">Process</a>	<a href="#"><code>exec(String cmd, String[] envp)</code></a> Executes the specified string command in a separate process with the specified environment.
<a href="#">Process</a>	<a href="#"><code>exec(String command, String[] envp, File dir)</code></a> Executes the specified string command in a separate process with the specified environment and working directory.

Figura 28: Esempio di sostituzione codice

hanno un programma associato (ed esempio: cd), perchè inutile (un processo figlio può eseguire qualcosa ma il processo padre non vedrebbe nessun cambiamento di stato), quindi vengono eseguiti da shell direttamente; altre volte questo avviene per motivi di ottimizzazione (ad esempio echo). Per eseguire

un processo in background devo aggiungere in coda al comando "&", cosicche non venga attesa la sua terminazione.

# Cheduling dei processi:

Se più processi sono eseguibili in un certo istante il sistema deve essere in grado di decidere quale eseguire per primo e la parte che prende questa decisione è definito **scheduler** attraverso un **algoritmo di scheduling**.

Ogni processo è unico e imprevedibile quindi per evitare tempi troppo lunghi di esecuzione di un processo la loro gestione è dipendente da un *Timer* ( o *Clock*)

Spesso la CPU processa per un po' senza fermarsi e successivamente viene eseguita una syscall per lettura o scrittura dei file. Il processi di I/O sono quelle situazioni in cui il processo entra in stato di blocked aspettando che un device esterno completi il suo lavoro. Si divide in compute-bound (o CPU-buond,

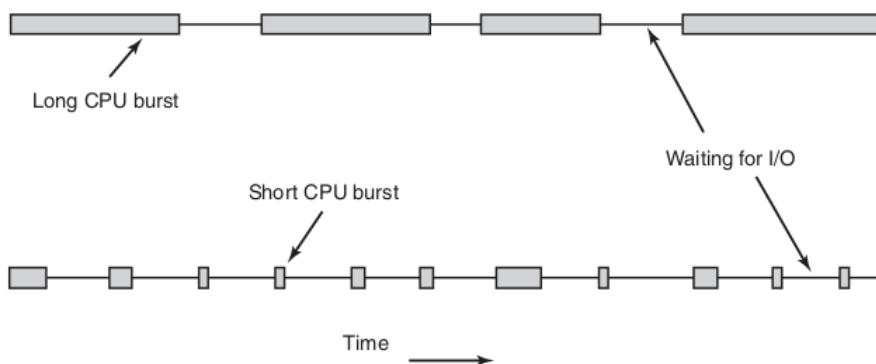


Figura 29: Esempio di burst e waiting della CPU

ovvero compiti che interessano la CPU) e I/O-bound, ovvero i compiti di I/O in cui la CPU non sta effettivamente eseguendo calcoli. Esistono di conseguenza due tipologie di algoritmi di scheduling:

- **Preemptive:** la strategia che consente di interrompere un processo dopo un tempo fissato a prescindere che il suddetto processo abbia finito la sua esecuzione (nel caso non abbia finito viene sospeso e lo scheduler prende un altro processo da eseguire). Esempio: passa da in esecuzione a pronto o da in attesa a pronto
- **Non - preemptive:** prede un processo e lo esegue fino al suo blocco (sia esso dovuto a I/O o ad un altro processo) o fino al rilascio volontario della CPU. Durante gli interrupt non vengono prese decisioni di scheduling. Esempio: passa da in esecuzione a in attesa o Termina
- **BATCH:** non ci sono utenti che attendono per il loro terminale per una risposta veloce e pertanto si preferiscono algoritmi non predittivi con tempi lunghi per ogni processo
- **INTERACTIVE:** in un ambiente con utenza interattiva è essenziale l'algoritmo predittivo per fare in modo che gli utenti abbiano apparentemente sempre la shell a loro disposizione.
- **REAL-TIME:** non è necessaria la schedulazione preventiva dei processi in quanto non sono attivi per periodi lunghi di tempo e solitamente si bloccano rapidamente. Essi infatti eseguono soltanto programmi intesi per uno scopo specifico, a differenza degli interattivi

Gli scopi che si pone un algortimo di scheduling sono molteplici:

- **Equità:** ogni processo deve avere a disposizione un tempo corretto di CPU

- **Efficienza:** la CPU dovrebbe essere utilizzata il 100%
- **Tempo di completamento**
- **Tempo di risposta:** da minimizzare per le utenze interattive
- **Throughput:** occorre massimizzare il numero di job processati per unità di tempo.

### Dispatcher:

è il modulo che da il controllo della CPU al processo selezionato dallo scheduler e coinvolge, nella sua operazione, il cambio di contesto, il passaggio al modo utente e il salto alla giusta posizione del programma utente. Il tempo necessario al cambio è chiamato *tempo di dispatch* e deve essere il più piccolo possibile

### FCFS (First Come First Served):

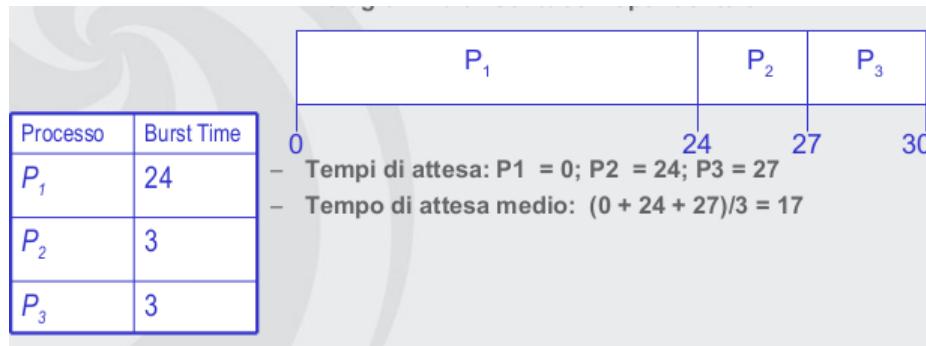


Figura 30: Schema di FCFS scheduling

Probabilmente l'algoritmo di scheduling più semplice in cui i processi ricevono l'utilizzo della CPU in ordine secondo le loro richieste. Sostanzialmente esiste una coda con l'insieme dei processi e man mano che si aggiungono processi essi vengono aggiunti in coda. Come si vede da un conto dei tempi medi di attesa se vengono eseguiti prima i processi più rapidi i tempi medi di attesa diminuiscono. E' pertanto necessario implementare un algoritmo che permetta di sfruttare questa caratteristica dei processi e che permetta di eseguire prima i processi più brevi in durata di burst.

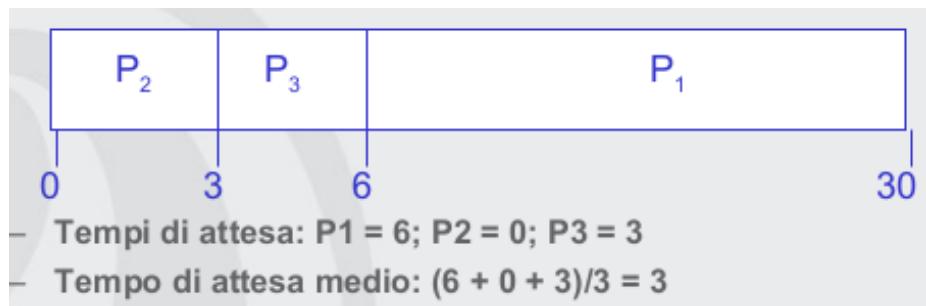


Figura 31: Esempio di caso in cui giungono prima i processi rapidi

### Shortest Job First:

Algoritmo particolarmente indicato per l'esecuzione batch (è algoritmo non predittivo) che presuppone di conoscere in anticipo il tempo di run dei processi. Ottimale quando sono disponibili quasi contemporaneamente i vari processi e hanno indicativamente tutti la stessa importanza. Applicabile anche al caso dei processi interattivi usando stime basate sul comportamento passato ed eseguendo il processo con il minor tempo stimato.

### Short Remaining Time First:

è una variante del SJF in cui se arriva un processo con un CPU burst atteso più breve del tempo rimanente per il processo corrente il nuovo processo ottiene la CPU. In questo caso l'algoritmo diventa di tipo *preemptive*



Figura 32: Esempio di preem

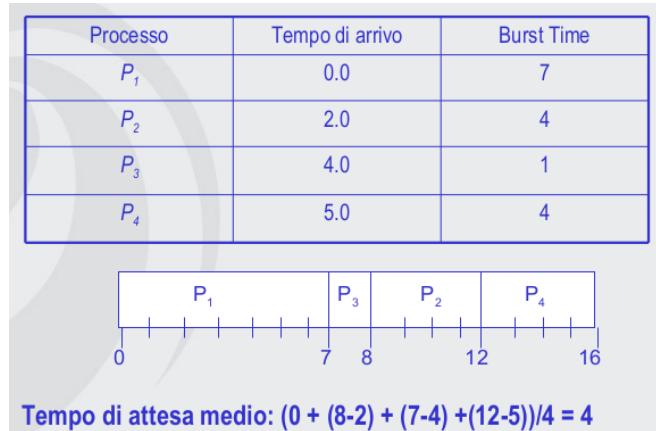


Figura 33: Esempio di non preem

## Round-Robin Scheduling:

Ad ogni processo viene assegnato un intervallo di tempo di esecuzione prefissato denominato **Quanto** (o **Time Slice**) durante il quale esso è autorizzato a processare. Se al termine di questo intervallo il processo non ha ancora terminato l'esecuzione l'uso della CPU viene comunque addifato ad un processo diverso. Il difetto principale del RR è la lunghezza del quanto. Il passaggio di processo infatti richiede del tempo.

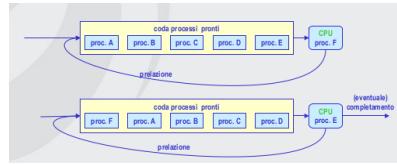


Figura 34: Sched. RR, si vede che ogni processo ha la stessa priorità

per il salvataggio dello stato corrente e l'aggiornamento di tabelle e liste. Il passaggio dell'esecuzione da un processo ad un altro è definita **Context Switch** e la durata del quanto di tempo non deve essere troppo breve per evitare molti context switch tra i processi con riduzione dell'efficienza della CPU conseguente e non deve essere troppo lunga per evitare tempi di risposta lunghi a processi interattivi con tempi di esecuzione brevi. Normalmente si ottiene un tempo di turnaround medio più alto che nel SJF



Figura 35: Diagramma di Grant di RR con time slice 20

ma una migliore risposta negli interattivi

## Priority Scheduling:

Round robin assume implicitamente che tutti i processi sono egualmente importanti. Per evitare che processi con maggior priorità vengano vincolati da processi con priorità più bassa o che processi a priorità alta vengano eseguiti indefinitamente lo scheduler decide la priorità ad ogni interrupt del clock. Avviene con un context switch quando la priorità del processo è minore di quella del processo successivo con priorità più alta.

L'assegnazione delle priorità può avvenire staticamente o dinamicamente ed è utile raggruppare i processi in **Gruppi di priorità** e utilizzare uno scheduling a priorità tra le classi, unito ad uno RR all'interno della classe

## Code multilivello:

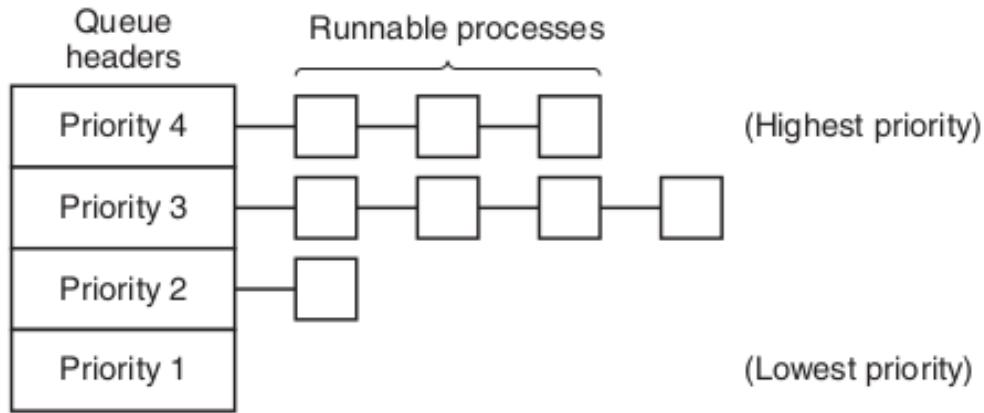
le code dei processi pronti possono essere divise in code separate, ad esempio dividendo in *processi in foreground* e *processi in background*. Ogni coda può utilizzare i propri algoritmi e successivamente viene implementato un algoritmo di scheduling tra le code. Esistono anche casi di **Code multilivello dinamiche** in cui un processo può spostarsi tra una classe e l'altra.

I processi nella classe più alta vengono eseguiti per un quanto, quelli nella seconda per que quanti, etc etc Quando un processo ha utilizzato i quanti ad esso assegnati viene passato alla classe inferiore, così facendo i processi lunghi scendono nelle code di priorità per dare precedenza all'esecuzione di processi interattivi brevi

## Scheduling garantito:

Un approccio di scheduling differente è quello di fare promesse reali all'utente e poi lasciare che si gestiscano, e.g. *se ci sono N utenti connessi ogni utente riceverà 1/N della potenza della CPU*.

Per mantenere la promessa il sistema deve tenere traccia di quanto tempo di CPU un utente ha utilizzato



**Figure 2-43.** A scheduling algorithm with four priority classes.

Figura 36: Esempio di scheduling con priorità a 4 classi

per i suoi processi dopo la procedura di login e anche quanto tempo è passato dal login. La priorità sarà calcolata in base al rapporto tra il tempo di CPU effettivamente utilizzato da un utente e il tempo che gli sarebbe spettato, con l'algoritmo che consiste nell'eseguire il processo con il rapporto minore finché il suo rapporto raggiunge quello del più vicino competitore. Questa idea è applicabile ai sistemi **real time** nei quali ci sono vincoli di tempo stretti da rispettare e l'algoritmo consiste nell'eseguire il processo che rischia maggiormente di non rispettare le scadenze

### Scheduling ad estrazione:

vengono distribuiti *biglietti* della lotteria (tempo di CPU) con l'idea di sostituire la priorità dando ai processi più importanti dei ticket extra per migliorare la loro possibilità di "vittoria". Sul lungo periodo la distribuzione dei biglietti determinerà la proporzione di CPU utilizzata dai singoli processi. E' un algoritmo **highly responsive** in realazione all'idea che se un nuovo processo viene inizializzato esso avrà comunque delle possibilità di "vittoria", dipendentemente dal numero di ticket che possiede.

### Scheduling Real Time:

un sistema real time è un sistema in cui il tempo di risposta è fondamentale, solitamente in relazione a stimoli esterni di sensori o in generale eventi. Esistono due tipi di real time systems:

- HARD REAL TIME: sono sistemi di gestione di processi critici che devono reagire entro un tempo stabilito senza possibilità di fallimento → i processi critici terminano entro un tempo stabilito.
- SOFT REAL TIME: sono sistemi in cui il tempo di reazione è fondamentale ma non è presente la criticità e la non tolleranza al fallimento dei sistemi hard rt → i processi critici hanno semplicemente una priorità maggiore degli altri

I sistemi real time generalmente devono rispondere a stimoli che possono essere **periodici** o **A-periodici**. In quest'ultimo caso se ci sono m eventi periodici e l'evento i avviene con periodo  $P_i$  richiedendo  $C_i$  secondi di CPU:

$$\sum_{i=1}^m \frac{T_{max}(P_i)}{T_{per}(P_i)} \leq 1$$

## Scheduling a due livelli:

Se la memoria principale è insufficiente alcuni processi eseguibili devono essere mantenuti sul disco e in questo caso lo scheduling dei processi comporta situazioni di switching molto differenti per i processi che risiedono sul disco e i processi che risiedono in memoria. E' il caso in cui avviene lo *swap*, ovvero quando vengono salvati su disco dei processi (non necessariamente processi in attesa di I/O) poichè la memoria principale non è sufficiente. E' necessario decidere quali processi mandare in esecuzione e quali salvare su disco, e può essere fatto attraverso uno scheduler a due livelli con due tipi di scheduler:

- **Scheduler a medio termine:** si occupa degli spostamenti dei processi tra memoria e disco, rimuovendo i processi che sono stati in memoria per un tempo sufficiente e caricando in memoria i processi che sono stati su disco a lungo.
- **Scheduler a breve termine:** si occupa dell'esecuzione dei processi che sono effettivamente in memoria

Nei sistemi batch talvolta si può parlare anche di un terzo tipo di scheduler, a **lungo termine**, che sceglie quali lavori (job) mandare in esecuzione. Sono molteplici i criteri di decisione dello scheduler di

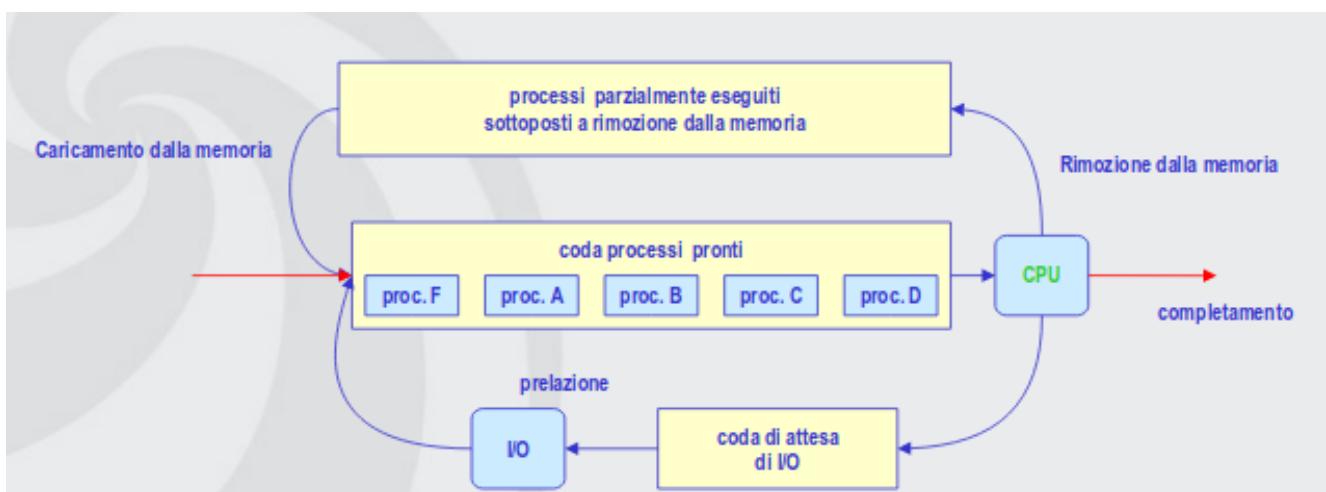


Figura 37: Scheduling a medio termine

alto livello:

- Il tempo passato dall'ultimo spostamento da/in memoria
- Quanto tempo di CPU è stato assegnato al processo
- La grandezza del processo (non conviene spostare processi piccoli)
- Priorità del processo

(L'algoritmo di scheduling può essere uno di quelli visti come RR, priorità, lotteria etc etc)

### Esempi:

- **WINDOWS 2000:** usa un algoritmo basato su priorità e prelazione con 32 livelli di priorità (una coda per ogni livello) e una classe "real time" (16-31); la classe 0 è usata solo dal thread di gestione della memoria. Il dispatcher sceglie il processo (**thread**) a priorità più alta pronto per l'esecuzione e se nessuno è pronto viene eseguito lo *idle thread* (ciclo idle del sistema). Se un processo a priorità variabile esaurisce il suo tempo gli viene tolta la CPU e la sua priorità abbassata (se invece rilascia spontaneamente la CPU la priorità viene innalzata, soprattutto se è in attesa di un evento). Vengono inoltre distinti i processi in primo piano dai processi in background.  
Un processo può sempre essere interrotto da un nuovo processo real time a priorità più alta.

- **LINUX:** Utilizza 3 diversi criteri di scheduling (FIFO, Round Robin, Timesharing) e i processi della prima classe sono interrotti solo da processi della stessa classe a priorità elevata, quelli della seconda sono eseguiti per un quanto di tempo e rimessi in coda. I processi normali sono eseguiti in base ai loro **crediti** (possibilità di eseguire un processo); ad ogni interruzione generata dal clock il processo in esecuzione perde un credito, se i suoi crediti sono 0 viene sostituito da un nuovo processo pronto e se nessun processo *pronto* dispone di crediti vengono riadattati i crediti di TUTTI i processi secondo:

$$\text{crediti} = \frac{\text{crediti}}{2} + \text{priorita'}$$

Un processo cede la CPU quando un processo a più alta priorità prima bloccato diventa pronto e dopo una `fork()` ogni processo eredita metà dei crediti.

I livelli di priorità sono 140, 0-99 per i processi realtime, 100-139 per i processi normali; Si definisce **Static Priority** =  $100 + \text{nice} + 20$  e tramite questo viene assegnato il quanto secondo la formula:

$$\text{quanto} = \begin{cases} (140 - SP) * 20 & SP < 0 \\ (140 - SP) * 5 & SP \geq 120 \end{cases}$$

**Nice** è un parametro di priorità definito dal sistema a seconda del contesto e del processo in considerazione.

```
struct runqueue {
    struct prioarray *active;
    struct prioarray *expired;
    struct prioarray arrays[2];
};

struct prioarray {
    int nr_active; /* # Runnable */
    unsigned long bitmap[5];
    struct list_head queue[140]; // double linked list
};
```

Quando il numero di processi scende a 0 i due array vengono scambiati con le priorità dei task aggiornate, con una coda differente per ogni processore. Ogni struttura possiede una lista doppiamente collegata per ogni livello di priorità.

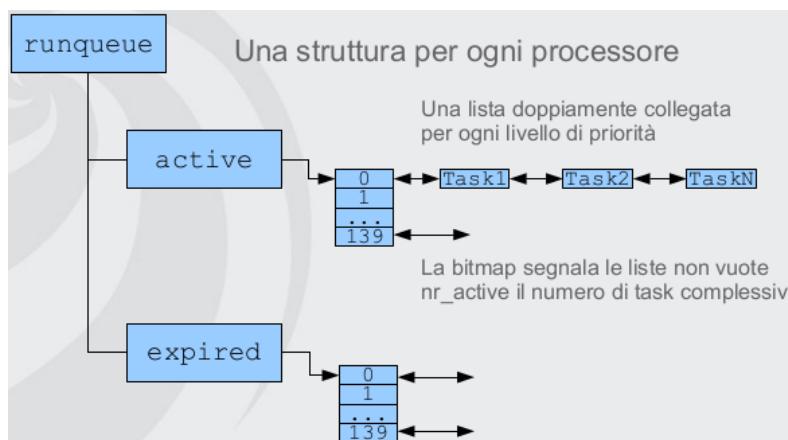


Figura 38: Struttura della runqueue

## Esercizi:

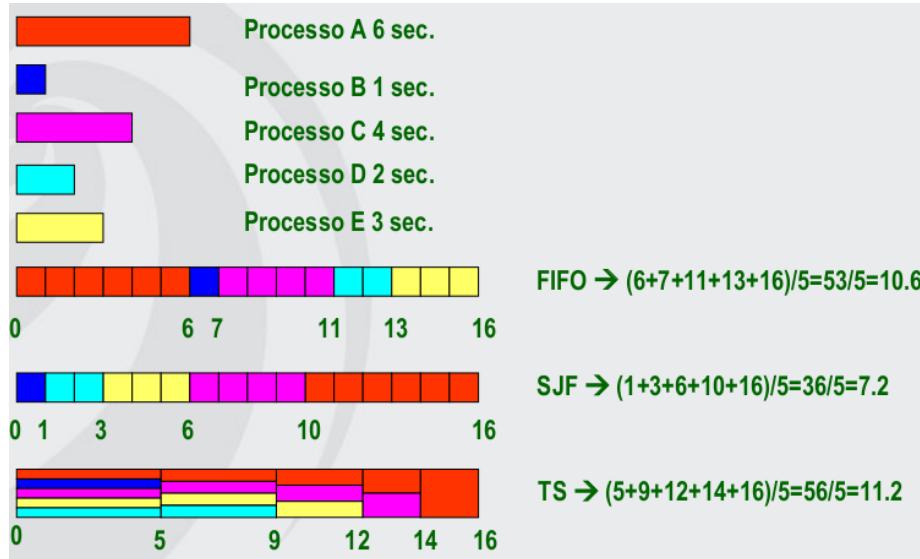


Figura 39: Calcolare tempo medio di completamento

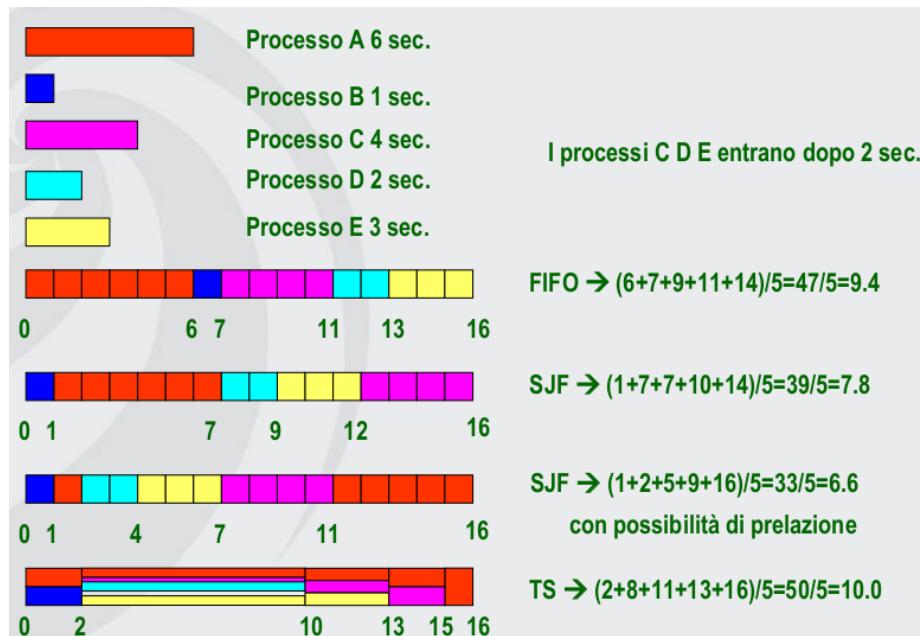


Figura 40: Calcolare il tempo medio di completamento contando i context switch

# I Thread:

Nei classici sistemi operativi ogni processo ha un indirizzo e un singolo thread di controllo. E' tuttavia possibile che si renda necessario avere thread multipli sullo stesso spazio di indirizzi che vengono eseguiti in pseudo-parallelo. I molte applicazioni infatti attività multiple vengono eseguite in contemporanea e scomporre un'applicazione in **threads** multipli sequenziali che vengono eseguiti in pseudoparallelo semplifica il modello di programmazione. Un sinonimo di threads è **processi leggeri**.

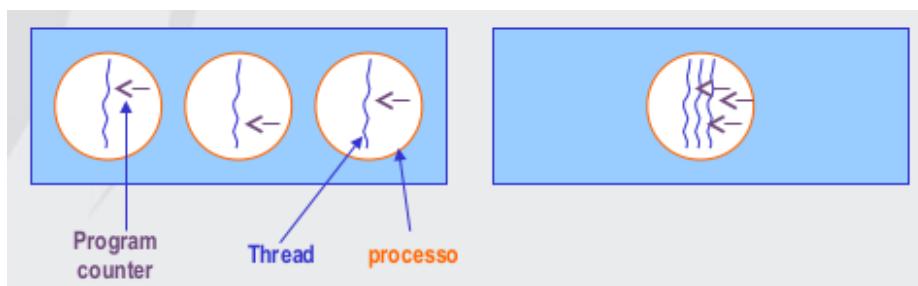


Figura 41: Struttura schematica di composizione del processo da parte dei threads

Un esempio pratico dell'utilità dei thread si può avere pensando ad un programma di scrittura: se si ipotizza una possibile modifica su 800 pagine di libro in un unico thread si nota come i passi da eseguire comprendono la modifica, la verifica della correttezza della modifica e l'effettivo salvataggio della modifica. Il tutto in un unico thread porterebbe ad avere notevoli rallentamenti e non reattività del programma. E' per questo possibile pensare a 3 thread concorrenziali dove i thread sono così suddivisi:

- Un thread si occupa di mostrare il contenuto di una pagina del libro
- Un thread si occupa di modificare il libro e riformattarlo, facendo in modo che il processo sia trasparente all'utente
- Un thread si occupa del salvataggio (anche automatico in background) del documento, sempre in modo trasparente.

**N.B.** non sarebbe stato possibile avere tre processi in quanto tutti devono aver accesso allo stesso documento e poter lavorare nella stessa area di memoria. Va sottolineato inoltre come con i moderni processori multicore il fatto di avere più thread per ogni processo permette di avere un parallelismo vero nei processi. Anche ad esempio nel caso di analisi di dati di grandi dimensioni: invece di leggere e aspettare quando non vi sono dati si possono dividere in thread i compiti con un thread per la lettura, un thread per l'input e uno per l'output, con la CPU che viene sfruttata nel migliore dei modi.

Il concetto di lasciare più thread nello stesso processo è detto **multithreading**. Come i processi un thread può essere in vari stati: *running*, *blocked*, *ready*, *terminated* e il passaggio di stati di un thread è esattamente come quello di un processo a se stante. Tuttavia ogni thread necessita del proprio stack poichè generalmente chiama procedure differenti e ha uno storico esecutivo differente.

## Gestione dei Thread:

Quando un multithreading è presente i processi solitamente iniziano con un singolo thread e questo ha la possibilità di creare nuovi thread chiamando funzioni di libreria *thread\_create*, con un parametro ch

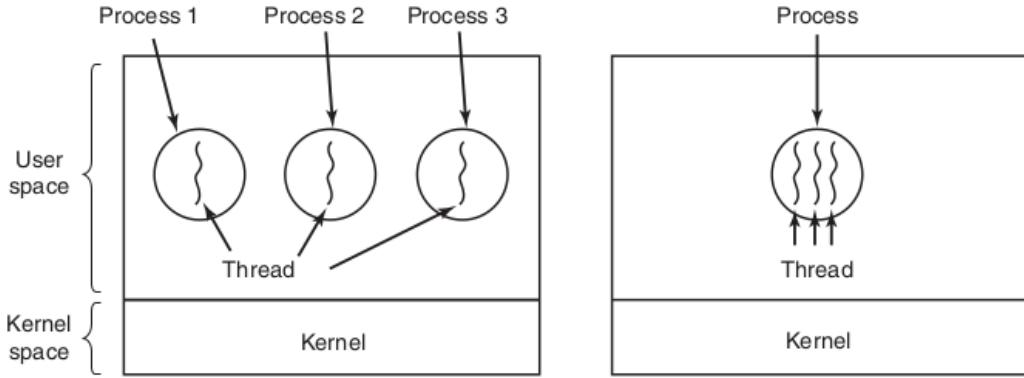


Figura 42: Tre processi ognuno con 1 thread // Un processo con 3 thread

specifica il nome della procedura per il nuovo processo. Le relazioni tra thread possono essere gerarchiche o, come avviene solitamente, non avere relazioni. Quando un thread ha finito il suo job può uscire chiamando la *thread\_exit*. Altre chiamate spesso utilizzate sono la *thread\_join* e la *thread\_yield*. E' tuttavia necessario prestare attenzione alla concorrenza dei thread per evitare situazioni ddi contrasto, ad esempio di un thread che chiude un file mentre un altro thread lo sta ancora utilizzando leggendo.

### Thread a livello utente:

I thread a livello utente sono indipendenti dal kernel, che non sa nulla di loro e quando chiamato gestisce un thread alla volta. Con questo approccio i thread sono implementati da libreria, la commutazione non richiede l'intervento del nucleo, lo scheduling dei thread è indipendente da quello del nucleo e può essere ottimizzato per la speciaia applicazione e i thread sono indipendenti dal sistema operativo. Hanno tuttavia degli *svantaggi*: le chiamate a sistema infatti sono bloccanti e non si sfrutta eventuale parallelismo hardware.

Quando i thread sono gestiti a livello utente ogni processo necessita di una sua **thread table** privata per tenere traccia dei thread in quel processo. E' analoga alla process table contenuta nel kernel se non per il fatto che tiene conto solo delle caratteristiche del thread (program counter, stack pointer...)

### POSIX Thread:

```
#include <stdio.h>
#include <pthread.h>
void* test_thread(void* pt) {
    printf("thread: %s\n", (char *) pt);
    pthread_exit(pt);
}
int main(int argc, char *argv[]) {
    pthread_t thread;
    pthread_attr_t attr;
    void *result;
    pthread_attr_init(&attr);
    if(pthread_create(&thread, &attr, test_thread, argv[1])) exit(1);
    pthread_join(thread, &result);
    if(result) printf("%s\n", (char*) result);
    return 0;
}
```

I thread sono definiti all'interno del package **Pthread** e presenta diverse chiamate possibili:

### Thread a livello Kernel:

E' possibile implementare i thread all'interno del kernel con la non necessità di avere thread table per ogni thread, con il kernel che ha una thread table che tiene traccia ddi tutti i thread nel sistema. Ogni

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figura 43: Alcune delle chiamate della libreria Pthread

volta tuttavia che è necessario creare un thread attraverso una chiamata a kernel. Essi sono infatti gestiti direttamente dal sistema operativo con i vantaggi di assegnare thread dello stesso processo a CPU diverse e in caso di chiamate di sistema si blocca il singolo thread. Il problema principale è la commutazione costosa, che va a minare uno dei vantaggi principali dell'uso dei thread. **Problemi di schedulazione:** Se ho due processi, un A con 1 thread e un B con 100 thread nel caso di *thread utente* A e B ottengono lo stesso tempo macchina con ogni thread di B che ha un centesimo dell'unico thread di A, nel caso di *thread di sistema* ogni thread ottiene lo stesso tempo, con A che ottiene un centesimo di tempos di B

## Modelli di programmazione multithread:

Esistono vari algoritmi di programmazione per il multithread per permettere la connessione tra i thread a livello utente e i thread a livello kernel.

- Uno-a-Uno
- Molti-a-uno
- Molti-a-Molti

### Uno a Uno:

Mappa ciascun thread utente in un thread kernel con il *vantaggio* di una maggiore concorrenza (possibili thread in parallelo e chiamate bloccanti che non bloccano tutti i thread) ma ha *svantaggi* nella creazione di molti thread a livello kernel che compromette le prestazioni dell'applicazione (tipicamente le realizzazioni di questo modello limitano il numero di thread a livello kernel)

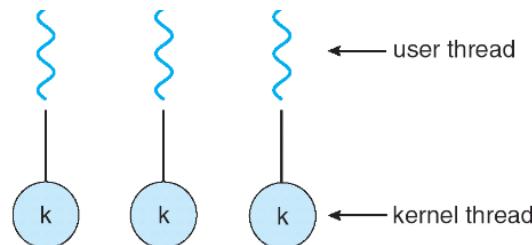


Figura 44: Schema uno a uno

### Molti a uno:

Il modello molti a uno riunisce molti thread di livello utente in un unico kernel thread. I *vantaggi* sono la gestione dei thread efficiente e l'intero processo bloccato se un thread invoca una chiamata bloccante di sistema, lo *svantaggio* maggiore è che un solo thread può accedere al kernel ed impossibile pertanto eseguire thread in parallelo.

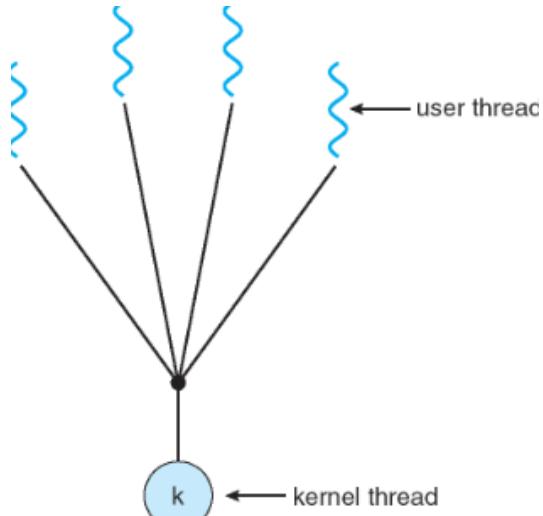


Figura 45: Schema molti a uno

### Molti a molti:

permette di aggregare molti thread a livello utente verso molti thread a livello kernel (solitamente un numero più piccolo o equivalente). Risolve le limitazioni dei modelli precedenti, con il numero massimo di processi a livello kernel che può essere personalizzato in base all'architettura.

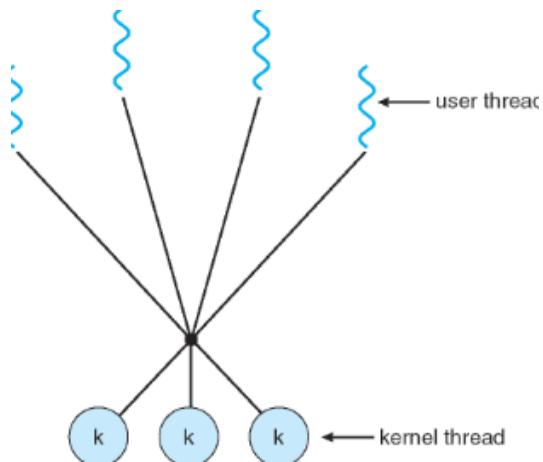


Figura 46: Struttura molti a molti

### Modello a più livelli:

Esiste anche il modello a più livelli, simile al molti a molti eccetto per il fatto che permette anche di associare un thread di livello utente ad un kernel thread

### Problemi di programmazione:

In programmazione è necessario tenere in considerazione aspetti dei thread ed evitarne i conflitti. Ad esempio la `fork()` crea un duplciato SOLTANNO del thread del processo che è in esecuzione durante la chiamata della funzione, non di tutti i thread del processo

Per la **cancellazione** esistono due differenti tecniche:

- *Asincrona*: il thread interessato viene fermato immediatamente
- *Differita*: ogni thread controlla periodicamente se deve terminare

## Segnali:

i segnali in UNIX sono notifiche asincrone inviate ad un processo o ad uno specifico thread per notificare dell'avvenuta di un evento. La differenza con gli *interrupt* è che i primi sono mediati dal kernel e gestiti dai processi, questi ultimi invece sono gestiti dal kernel e mediati dai processi. Tutti i segnali seguono lo stesso schema:

1. Il segnale è generato da un particolare evento
2. Il segnale è inviato ad un processo
3. Il segnale viene gestito

La gestione del segnale può avere invece molteplici vie, con il segnale inviato solo al thread di interesse, a tutti i thread, solo ad alcuni o addirittura mediante un thread per gestire i segnali.

Si possono inoltre creare dei **gruppi di thread** che attendono di lavorare, con i vantaggi di limitare il numero di thread esistenti e che è più veloce usare un thread esistente che non crearne uno nuovo. E' anche possibile assegnare **dati specifici ad ogni thread**, ad esempio associando un identificatore diverso ad ogni thread (N.B. attenzione se questa tecnica viene usata con i gruppi di thread)

## Esempi:

**Windows XP:** Utilizza un modello uno-a-uno con ogni thread che contiene un ID, un insieme di registri, pile separate per il modo utente e il modo kernel e deti privati; registri, pile e memoria sono definiti come **contesto** del thread

**Linux:** Nella terminologia Linux si parla di **task** invece che di thread e la creazione viene fatta attraverso la chiamata di sistema **clone()**, che permette ad un task figlio di condividere lo spazio indirizzi del task genitore → la **fork()** diventa così un caso particolare della **clone()**

## Thread in Java:

I thread in Java sono gestiti dalla JVM e possono essere creati in ddui modi: estendendo la classe Thread o implementando l'interfaccia Runnable.

### La classe Thread:

Esistono varie chiamate per la classe Thread

- **Thread( threadName )** → crea un nuovo thread con il nome specificato
- **Thread()** → crea un nuovo thread con il nome predefinito
- **run()** → metodo che effettivamente fa svolgere il lavoro al thread e deve essere invocato nella sottoclasse di chiamata
- **start()** → lancia l'esecuzione del thread permettendo quindi il proseguimento dell'esecuzione del chiamante, richiamando il metodo **run()**; NB è un errore chiamare due volte **start()** riguardo allo stesso thread

```
class Worker1 extends Thread {
    public void run() {
        System.out.println("I Am a Worker Thread");
    }
}
public class FirstThreadTester {
    public static void main(String args[]) {
        Worker1 runner = new Worker1();
```

```

    runner.start();
    System.out.println("I Am The Main Thread");
}
}

```

## Interfaccia Runnable:

```

public interface Runnable {
    public abstract void run();
}

```

E' un'interfaccia implementata anche dalla classe Thread che permette di gestire i thread in Java.

```

class Worker2 implements Runnable {
    public void run() {
        System.out.println("I Am a Worker Thread ");
    }
}
public class SecondThreadTester {
    public static void main(String args[]) {
        Runnable runner = new Worker2();
        Thread thread = new Thread(runner);
        thread.start(); //Creo un Thread a partire da un oggetto Runnable
        System.out.println("I Am The Main Thread");
    }
}

```

**Priorità dei thread:** Ogni applicazione o Thread in Java è multithread e la priorità dei thread va da 1 a 0, con ogni nuovo thread che *eredita la priorità di chi l'ha creato*.

```

Thread.MIN_PRIORITY // setta la priorità a 1, minimo per il thread
Thread.NORM_PRIORITY // setta la priorità 5, default per il thread
Thread.MAX_PRIORITY // setta la priorità a 10, massimo per il thread

```

Ogni thread ottiene poi un quanto del tempo di processore per l'esecuzione, detto **time slice**, e al termine il processore passa al prossimo thread di pari priorità (se esiste) → schedulazione di tipo round robin.

I metodi per controllare la priorità sono, ad esempio:

```

setPriority(int priorityNumber)
getPriority()

```

Se invece volessi controllare lo scheduling potrei usare

```
yield()
```

Usato come segue:

```

public void run() {
    while(true {
        FaiQualcosa(2000); // vedi slide successiva
        yield();
    }
}

public void FaiQualcosa(long time) {
// aspetta per un certo tempo
// time = (long) (Math.random()*time); tempo casuale
try {
    Thread.sleep(time);
}

```

```

    } catch(InterruptedException e) {}
}

```

Il metodo statico `yield()`, mette temporaneamente in pausa il thread corrente e consente ad altri thread in stato Runnable (qualora ve ne siano) di avere una chance per essere eseguiti. Nel caso non vi fossero altri thread con tali requisiti, il metodo `yield()` non avrà alcun effetto.

Il metodo `sleep(milliseconds)` invece sospende il thread per un numero specificato di millisecondi e nel frattempo possono andare in esecuzione thread a priorità più bassa

Altri metodi specifici per la gestione dei Thread sono:

```

void interrupt() // il thread viene bloccato
boolean isInterrupted()
static Thread currentThread() // restituisce il th. in esecuzione
boolean isAlive() // verifica se il thread è attivo
void join()

```

**Uso di Join:** La Join viene così utilizzata su un thread specifico e ha lo scopo di mettere in attesa il thread attualmente in esecuzione fino a quando il thread su cui è stato invocato il metodo `join()` non termini

```

class JoinableWorker implements Runnable {
    public void run() {
        System.out.println("Worker working");
    }
}
public class JoinExample {
    public static void main(String[] args) {
        Thread task = new Thread(new JoinableWorker());
        task.start();
        try {
            task.join();
        }
        catch (InterruptedException ie) { }
        System.out.println("Worker done");
    }
}

public class TestJoin {
    public static void main(String[] args)
        throws InterruptedException {
    Thread[] threads = new Thread[10];
    for(int i=0; i<threads.length; i++) {
        threads[i] = new Thread(){ public void run()
            { System.out.println(this);}};
        threads[i].start();
    }
    if(args.length==0) {
        for(int i=0; i<threads.length; i++) {
            threads[i].join();
        }
    }
    System.out.println(Thread.currentThread());
}

```

# Comunicazioni interprocesso:

I processi spesso necessitano di comunicare con altri processi, ad esempio in una pipeline della shell l'output del primo processo deve essere passato al secondo e così via. E' importante evitare che durante la comunicazione di processi avvenga inconsistenza dei dati, che ci sia concorrenza tra i processi senza un controllo e che i tempi dei vari processi siano rispettati (ovvero che il processo che eredita le informazioni non inizi prima che il precedente abbia effettivamente concluso).

**Bounded-buffer problem:** il problema produttore-consumatore è un tipico esempio di comunicazione interprocesso che mostra i possibili problemi che possono generarsi da essa; vi sono due processi, un processo *produttore* e un processo *consumatore*, che comunicano usando un unico buffer di dimensione fissata. E' necessario che il produttore non inserisca informazioni all'interno del buffer quando questo è ancora pieno, e allo stesso tempo che il consumatore non cerchi di prelevare dati dal buffer mentre questo è vuoto e non contiene alcuna informazione.

```
class Produttore extends Thread {  
    public void run() {  
        int appenaProdotto = 0;  
        int inserisci = 0;  
        while(running) {  
            appenaProdotto++;  
            aspetta(milliProduttore); // produco un dato  
            while(contatore == vettore.length) ;  
            vettore[inserisci] = appenaProdotto;  
            inserisci=(inserisci+1)%vettore.length;  
            incrementaContatore(); // contatore++  
        }  
    }  
}  
  
class Consumatore extends Thread {  
    public void run() {  
        int preleva = 0;  
        int daConsumare = 0;  
        while(running || contatore>0) {  
            while(contatore == 0) ;  
            daConsumare = vettore[preleva];  
            preleva = (preleva+1)%vettore.length;  
            decrementaContatore(); // contatore--  
            aspetta(milliConsumatore); // uso il dato  
        }  
    }  
}
```

**Race Condition:** Le istruzioni `contatore++` e `contatore--` devono essere eseguite in modo atomico (**Operazione atomica:** operazione indivisibile dal punto di vista logico, eseguibile pertanto senza che avvengano interruzioni). In realtà le due operazioni coinvolgono più registri e, ad esempio l'incremento, è traducibile come:

```
register1 = contatore
register1 = register1 + 1
contatore = register1
```

Se pertanto sia il produttore che il consumatore tentano di aggiornare il buffer concorrentemente le istruzioni assembly possono risultare intercalate e la sequenza effettiva dipende dallo scheduling dei due processi. Ipotizzando ad esempio che il contatore valga 5 ad inizio programma e che si verifichino le seguenti istruzioni, secondo una possibile sequenza:

```
producer: register1 = contatore (register1 = 5)
producer: register1 = register1 + 1 (register1 = 6)
consumer: register2 = contatore (register2 = 5)
consumer: register2 = register2 - 1 (register2 = 4)
producer: contatore = register1 (contatore = 6)
consumer: contatore = register2 (contatore = 4)
```

Il valore finale del contatore può valere 4 o 6, invece che 5, a seconda dello scheduling. Quando due o più processi stanno leggendo/scrivendo qualche dato condiviso e il risultato dipende da chi esegue quando si verifica una **Race Condition**. Per prevenire le race condition i processi concorrenti devono essere sincronizzati

### Sezioni critiche:

Il problema delle sezioni critiche si presenta così:

- N processi competono per usare dati condivisi
- Ogni processo ha un segmento di codice (**sezioni critiche**) in cui i dati sono utilizzati
- E' necessario garantire che quando un processo sta eseguendo la sua sezione critica nessun altro processo possa entrarvi

E' fondamentale nella progettazione di un s.o. la scelta di appropriate operazioni primitive per ottenere la mutua esclusione delle sezioni critiche. Alcune buone norme da rispettare:

1. due processi non devono mai essere simultaneamente in una sezione critica
2. nessuna assunzione a priori può essere fatta a proposito della velocità o del numero di CPU
3. nessun processo in esecuzione fuori dalla sua sezione critica può bloccare altri processi
4. nessun processo deve aspettare ad oltranza per entrare nella sua sezione critica

**Mutua esclusione:** è la condizione che si vuole raggiungere riguardo i processi e le loro sezioni critiche. Per ottenerla sono state proposte varie soluzioni, ognuna distintiva per qualche implementazione di sistema operativo.

Prima di usare le variabili condivise ciascun processo chiama una `enter_region` con il proprio numero di processo (0 o 1) e può dover aspettare fino a che risulti sicuro entrare nella propria ragione critica. Dopo aver finito di lavorare con le variabili condivise il processo chiama una `leave_region` per permettere ad un altro processo di entrare.

```
void processo(int proc) {
    while(true) {
        enter_region(proc);
        critical_region();
        leave_region(proc);
```

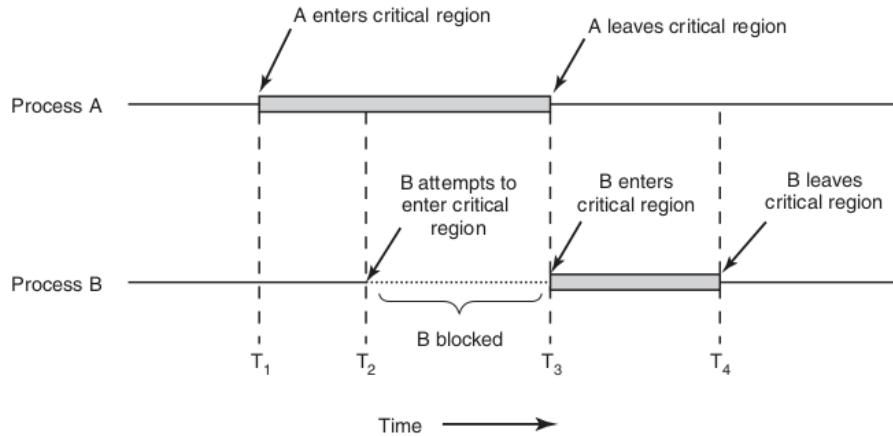


Figura 47: Diagramma temporale di una mutua esclusione con regioni critiche

```
non_critical_region();
}
}
```

**Variabili di lock:** se un processo vuole entrare nella sua regione critica esegue un test su una singola variabile lock, condivisa da tutti i processi e inizializzata a 0.

- Se il lock è a 0 (nessun processo in regione critica) il processo lo imposta a 1 ed entra nella sua regione critica
- Se il lock è 1 (c'è qualche processo in regione critica) il processo attende finché non diventi 0

```
volatile boolean lock = false;
while(true) {
    while(lock);
    lock = true;
    critical_region();
    lock = false;
    non_critical_region();
}
```

**PROBLEMI:** due processi possono entrare contemporaneamente nella loro regione critica e fallire il test di lock (se un processo ad esempio mentre sta settando ad 1 la variabile di lock venisse anticipato da un altro processo).

**Alternanza stretta:** Si tratta di un approccio differente rispetto alle variabili di lock, con il codice che implementa la stretta alternanza nell'esecuzione della regione critica da parte dei processi attraverso un'**attesa attiva**: test continuo su una variabile nell'attesa di un certo valore, è estremamente dispendioso per la CPU. In questa implementazione esiste una variabile che tiene traccia dei turni per l'ingresso nella regione critica, con i processi che continuano a controllare tale variabile finché non è il loro turno di ingresso.

**PROBLEMI:** Il problema principale di questa implementazione è che un processo può essere bloccato da un altro processo che non si trova nella sua regione critica (se infatti un processo è impegnato nella sua regione non critica e l'altro termina l'esecuzione critica e ritorna pronto prima che il primo finisca la sua esecuzione al di fuori della regione critica il processo 2 si trova bloccato (nonostante sia in ready) da un processo che non si trova nella regione critica) → violazione terza condizione di mutua esclusione. Oltretutto se un processo si ferma anche l'altro è fermo e se si generalizza la soluzione ad N processi anziché due si vedono ancora meglio le criticità.

```

final static int N=2;
// numero di processi
boolean[] pronto = new boolean[N]; // inizializzato a false
void enter_region(int processo) { // processo 0 o 1
    int altro = 1 - processo; // l'altro processo 1 o 0
    pronto[processo] = true;
    while(pronto[altro]);
    // segnala che si e' interessati
}
void leave_region(int processo) {
    pronto[processo] = false;
}

```

**Soluzione di Peterson:** unisce l'idea dei turni con l'idea di variabili di lock e variabili di warning. Prima di utilizzare le variabili condivise ogni processo chiama la `enter_region` con il proprio numero di processo. Questa chiamata pone il processo in attesa finché non è sicuro entrare nella regione critica. Finito il lavoro in regione critica il processo chiama una `leave_region` per indicare il suo aver completato. Un possibile scenario:

1. Nessun processo è in regione critica
2. Il processo 0 chiama la `enter_region`
3. Indica il suo interesse mettendo a *true* il proprio elemento nell'array
4. Mette turn a 1
5. Se il processo 1 non è interessato, `enter_region` termina subito
6. Se il processo 1 chiama `enter_region` rimane in attesa fino a che `interested[0]` è *false* (ovvero fino a quando il processo 0 non chiama la `leave_region`)
7. Se entrambi i processi chiamano la `enter_region` modificano il valore di turn (**il primo viene sovrascritto**)
8. Se il processo 1 scrive per ultimo nella variabile turn, quando entrambi i processi arrivano al ciclo while, il processo 0 non esegue il ciclo ed entra immediatamente in sezione critica, il processo 1 al contrario esegue il ciclo rimanendo in attesa di entrare in sezione critica

```

final static int N=2;
int turno;
// numero di processi
boolean[] pronto = new boolean[N]; // inizializzato a false
void enter_region(int processo) { // processo 0 o 1
    int altro = 1 - processo; // indice dell'altro processo 0 o 1
    pronto[processo] = true;
    // segnala che si e' interessati
    turno = altro;
    while(pronto[altro] && turno == altro);
}
void leave_region(int processo) {
    pronto[processo] = false;
}

```

### Algoritmo del fornaio:

E' un algoritmo per evitare le race conditions

```
final static int N = Numero_di_processi;
boolean[] scelta = new boolean[N]; // inizializzato a false
int[] numero = new int[N]; // inizializzato a 0

void enter_region(int processo) {
    scelta[processo] = true;
    numero[processo] = max(numero) + 1; //max massimo del vettore
    scelta[processo] = false;
    for (int j=0; j<N; j++) {
        while( scelta[j]);
        while( numero[j]!=0 && compare(j , processo));
    }
}
void leave_region(int processo) {
    numero[processo] = 0;
}

boolean compare(int i , int j) {
    return numero[i]<numero[j] || (numero[i]==numero[j] && i<j);
}
```

E' possibile che più thread ricevano lo stesso numero di turno. Per ovviare a questa circostanza si introduce l'indice del thread come secondo argomento di confronto. Se più thread ricevono lo stesso numero di turno, si sceglie di assegnare la precedenza al thread con l'indice più basso.

Va notato che l'indice di ciascun thread deve essere unico: esso può venire assegnato al momento della creazione o passato come parametro. Nell'esempio schematico riportato sopra, la costante N rappresenta il numero massimo di thread concorrenti. Dopo aver ricevuto il suo indice unico, ogni thread scrive solo nelle sue slot (scelta[j] e numero[j]). La serializzazione è assicurata dalle due iterazioni while consecutive. Questi cicli vengono ripetuti da ogni thread per tutti i thread, incluso quello in esecuzione e i thread non attivi. Solo quando non ci sono più altri thread con priorità più alta è possibile l'accesso alla sezione critica.

### Disabilitazione degli interrupt:

Ogni processo disabilita tutti gli interrupt dopo essere entrato nella sua regione critica e li riabilita prima di lasciarla (se sono disabilitati nessun interrupt del clock può essere attivato). Questo metodo da la possibilità di neutralizzare gli interrupt: tale procedura è utile se a carico del SO (NON può essere lasciata all'utente), ma non è un buon approccio come meccanismo generale per risolvere il problema della *mutua esclusione*.

```
while(true) {
    /* disabilita gli interrupt */
    critical_region();
    /* abilita gli interrupt */
    non_critical_region();
}
```

### Istruzione TSL:

E' un'istruzione presente soprattutto nei computer disegnati con l'idea di essere multiprocessore. E' formata similmente a TSL RX, LOCK e legge il contenuto della parola di memoria *lock* all'interno del registro RX e salva un valore diverso da zero all'interno dell'indirizzo di lock. Si noti che le read e write sono operazioni atomiche. Come si vede dal seguente blocco di codice:

```

enter_region:
    tsl register,lock
    cmp register,#0
    jne enter_region;
    ret

leave_region:
    move lock,#0
    ret

```

Prima di entrare in regione critica un processo chiama la *enter\_region* che attende finchè non è libero. A quel punto accquisisce il lock e ritorna. Dopo aver lasciato la regione critica il processo chiama la *leave\_region* che salva 0 in lock. Se tuttavia un processo imbroglia nella gestione la mutua esclusione fallisce (funziona soltanto se i processi collaborano). Un'alternativa utilizzata anche da x86 è la **XCHG**:

```

enter region:
    MOVE REGISTER,#1
    XCHG REGISTER,LOCK
    CMP REGISTER,#0
    JNE enter_region
    RET

leave region:
    MOVE LOCK,#0
    RE

```

## Sleep and Wakeup:

Le soluzioni di Peterson sono corrette ma hanno il difetto di richiedere necessariamente l'**attesa attiva** (*busy waiting*) che implica uno spreco di tempo di CPU e un problema dell'inversione di priorità (attesa di un processo a bassa priorità) → **priority inversion problem**.

La soluzione ideale a questo tipo di problema è l'utilizzo delle *system\_call sleep* e *wakeup*, che bloccano i processi invece di sprecare tempo di CPU quando non possono eseguire la loro regione critica.

- **sleep**: system call che blocca il chiamante finchè un altro processo lo risveglia.
- **wakeup**: system call che ha come unico parametro il processo da risvegliare.

## Problema *Produttore-Consumatore*:

E' un esempio dell'utilizzo delle chiamate primitive **sleep** e **wakeup**:

- Se il produttore vuole mettere un nuovo elemento nel buffer pieno viene messo in attesa per essere risvegliato quando il consumatore ha prelevato qualche elemento
- Se il consumatore vuole prelevare un elemento dal buffer vuoto viene messo in attesa per essere risvegliato quando il produttore ha messo qualcosa nel buffer.

```

static final int N=100;
int count = 0;

void producer() {
    while(true) {
        produce_item();
        if (count==N)
            sleep();
        enter_item();
        count++;
    }
}

```

```

    if (count == 1)
        wakeup (consumer);
}

void consumer () {
while (true) {
    if (count==0)
        sleep ();
    remove_item ();
    count--;
    if (count == N-1)
        wakeup (producer);
    consume_item ();
}
}

```

Si ha tuttavia una corsa critica su *count* poichè il consumatore legge *count* e viene sospeso dal SO, il produttore immette un dato e lancia un *wakeup*, riprende il consumatore, fa il test su 0 e si sospende. Quando il buffer si sarà riempito anche il produttore si sospende ed entrambi i processi saranno fissi in sospensione.

## Semafori:

Djikstra introdusse una nuova proposta per la soluzione dei problemi relativi alla sincronizzazione tra processi e alle condizioni di gara, introducendo i cosiddetti **semafori**.

Essi sono una variabile intera per il conteggio del numero di *wakeup* pendenti che può avere valore 0 se non è stato salvato alcun *wakeup* (e il processo viene messo in *sleep*) o positivo se ci sono *wakeup* pendenti. Il controllo, la modifica e lo *sleep* sono tutti eseguiti come azione atomica unica. E' inoltre presente la garanzia che una volta che un'operazione di semaforo è iniziata nessun'altra operazione può accedere al semaforo finchè la precedente non è completata.

```

public interface Semaforo {
    public void up();
    public void down();
}

```

Le due operazioni principali sono **UP()** e **DOWN()** che rispettivamente incrementano e decrementano l'indirizzo del semaforo. Se, dopo il decremento, il semaforo ha un valore negativo, il task viene sospeso e accodato, in attesa di essere riattivato da un altro task. In caso di *down* se ci sono task in coda, uno dei task in coda (il primo nel caso di accodamento FIFO) viene tolto dalla coda e posto in stato di *ready* (sarà perciò eseguito appena schedulato dal sistema operativo). Sostanzialmente il **down()** controlla se è zero il valore del semaforo, se non lo è lo decrementa, se è zero il processo è posto in *sleep* senza completare il *down* per un momento.

## Produttore-Consumatore:

E' possibile risolvere il problema del produttore/consumatore attraverso l'utilizzo dei semafori, usanto 3 semafori per mutua esclusione e sincronizzazione.

```

static final int N=100;
// numero di posizioni nel buffer */
Semaforo mutex = getDefaultValue(1); // controlla l'accesso in s.c.
Semaforo empty = getDefaultValue(N); // conta il numero di pos. vuote
Semaforo full = getDefaultValue(0); // conta il numero di elementi
void producer() {
    while(true) {
        int item = produce_item();

```

```

empty.down();
// decrementa il numero delle posizioni vuote
mutex.down();
// entra in sezione critica
enter_item(item);
// mette un nuovo elemento nel buffer
mutex.up();
// abbandona la sezione critica
full.up();
// incrementa il numero delle posizioni piene
}

}

void consumer() {
    int item;
    while(true) {
        full.down();
        mutex.down();
        item=remove_item();
        mutex.up();
        empty.up();
        consume_item(item);
    }
}
}

```

I tre semafori binari utilizzati sono il semaforo per il full, il semaforo per l'empty e il semaforo per garantire che venditore e consumatore non accedano al buffer nello stesso istante.

## Monitor:

L'utilizzo di programmi con l'utilizzo di semafori può portare ad errori gravi come **Deadlock**, ovvero una situazione in cui più processi sono definitivamente bloccati e non può essere eseguito alcun lavoro. Una soluzione che consente di realizzare programmi corretti in modo semplice è l'utilizzo di una primitiva di sincronizzazione di più alto livello definita **monitor**.

Essa è una collezione di procedure, variabili e strutture dati raggruppate in un pacchetto di tipo speciale; i processi possono richiamare le procedure contenute in un monitor ma non possono accedere direttamente alle sue strutture dati interne da procedure dichiarate esterne al monitor. Oltre tutto i monitor sono costruiti di un linguaggio di programmazione pertanto il compilatore gestisce diversamente le chiamate alle procedure del monitor dalle chiamate delle altre procedure.

**Utilità:** I monitor sono utili per ottenere la mutua esclusione, infatti in ogni istante solo un processo può essere attivo all'interno del monitor. Affinchè due processi non entrino in contemporanea nella loro regione critica è sufficiente inserire le loro sezioni critiche nelle procedure del monitor: così facendo la realizzazione della mutua esclusione dei monitor è demandata al compilatore e non al programmatore, rifiutando le possibilità di errore. **Come blocco processi quando non possono procedere?** Attraverso variabili con due operazioni su di esse, ovvero **wait** e **signal**. La prima è un'operazione su una variabile condizione che blocca il processo chiamante, la seconda un'operazione su una variabile condizione che risveglia il processo relativo. **N.B.** una signal deve essere l'ultima operazione in una procedura di un monitor per evitare che due processi siano attivi contemporaneamente nel monitor (ovvero nella sezione critica)

```

monitor ProducerConsumer
    integer count;
    condition full, empty;
    procedure enter;
        if count=N then wait(full);

```

```

enter_item;
count := count + 1;
if count=1 then
    signal(empty);
end;
procedure remove;
if count=0 then wait(empty);
remove_item;
count := count - 1;
if count=N-1 then
    signal(full);
end;
count := 0;
end monitor;

```

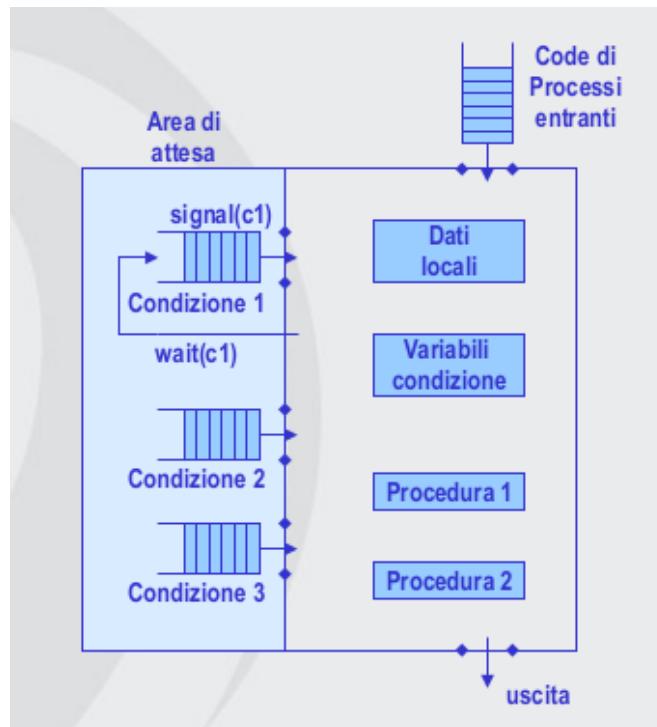


Figura 48: Schema di un monitor e del suo funzionamento

## Problemi di monitor e semafori:

Molti linguaggi come C e Pascal non prevedono i monitor e i semafori e le primitive utilizzate per l'implementazione non sono applicabili in ambiente distribuito, dove più CPU, ognuna con la relativa memoria privata, sono connesse da una rete locale. L'unica soluzione è il **passaggio di messaggi**.

**Passaggio di messaggi:** processo di comunicazione tra processi che utilizza due primitive, **send** e **receive**, ovvero systemcall che possono essere inserite i procedure di libreria.

**Produttore-Consumatore:** Con il passaggio di messaggi sono possibili molte varianti per l'indirizzamento dei messaggi:

- Assegnare ad ogni processo un indirizzo unico ed indirizzare i messaggi ai processi
- *Mailbox*: struttura dati speciale che consente di bufferizzare un numero di messaggi definito alla creazione della mailbox. I parametri si indirizzano nelle chiamate send e receive sono le mailbox, non i processi.

I due estremi con l'utilizzo della Mailbox sono i seguenti:

- Produttore e consumatore possono creare la mailbox in grado di contenere N messaggi
- Rendezvous, ovvero nessuna bufferizzazione e produttore e consumatore lavorano forzatamente allo stesso passo.

## Barriere:

Se un'applicazione applica la regola per cui nessun processo può proseguire alla fase successiva finché non sono pronti tutti i processi è possibile implementare le **barriere**

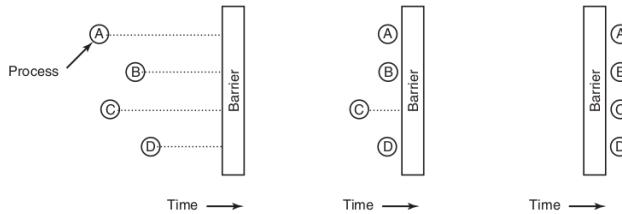


Figura 49: (a) arrivo dei processi, (b) blocco finchè non son tutti pronti e (c) passaggio quando sono tutti pronti

## Problemi di IPC:

Esistono vari problemi noti di IPC, i più famosi sono:

### Filosofi a cena:

E' un problema di sincronizzazione in cui N filosofi stanno seduti intorno ad un tavolo e ciascun filosofo ha un piatto di spaghetti: per mangiarli ogni filosofo ha bisogno di usare due forchette. Tra ognuno dei piatti vi è una forchetta. La vita dei filosofi è fatta di periodi alternati in cui essi *pensano o mangiano*. Quando un filosofo comincia ad avere fame cerca di prendere possesso della forchetta di sinistra e di quella di destra, una alla volta in ordine alfabetico. Quando ha a disposizione entrambe le forchette incomincia a mangiare, quando si è sfamato depone le forchette e inizia a pensare. Come scrivere un programma per ciascuno dei filosofi che non si fermi mai? E' importante evitare che i filosofi prendano, ad esempio, tutte la forchetta di sx nello stesso istante → nessuno di loro sarà in grado di prendere quella di destra e avrei uno stallo.

E' possibile un *secondo approccio* per cui un filosofo prende la prima forchetta, se non è disponibile la seconda forchetta rimette la prima sul tavolo e aspetta un certo intervallo di tempo. Se tutti i filosofi iniziano contemporaneamente con una forchetta e l'altra non è disponibile la ripongono, aspettano e riprovano da capo.

**Starvation:** situazione in cui ogni programma è in esecuzione senza che ottenga effettivamente alcun progresso

Esiste anche un *terzo approccio*, benchè molto costoso: si fa uso di un semaforo binario:

- prima di prendere possesso delle forchette un filosofo esegue una **down** sulla variabile mutex.
- dopo aver riposto le forcchette esegue una **up** sulla variabile mutex

La soluzione è corretta ma costosa per le prestazioni: solo un filosofo alla volta può mangiare, benchè con 5 forchette 2 filosofi potrebbero mangiare contemporaneamente. La soluzione migliore è la seguente → consentire il grado di parallelismo per un numero arbitrario di filosofi (impiegando un vettore di stati/semafori per ogni filosofo: un filosofo inizia a mangiare solo se nessuno dei vicini sta mangiando).

### Lettori e scrittori:

Modella l'accesso ad un database molto grande con tanti processi in competizione per leggere o scrivere (molti possono leggere contemporaneamente ma se uno sta scrivendo nessuno può leggere). Una soluzione possibile è l'implementazione di un **albero binario**: un lettore esegue una **down** sul semaforo db, i lettori successivi incrementano (entrando) e decrementano (uscendo) un contatore. L'ultimo a uscire esegue una **up** sul semaforo lasciando via libera ad uno scrittore. Sostanzialmente ci sono due semafori, DB e RC. Il secondo incrementato all'ingresso di un lettore e decrementato all'uscita, il primo quando poi esce l'ultimo lettore viene incrementato permettendo ad uno scrittore bloccato di entrare.

### Barbiere dormiente:

In un negozio di barbiere c'è un unico barbiere, una poltrona da lavoro e N clienti. Se non ci sono clienti il barbiere si riposa sulla poltrona, quando arriva un cliente sveglia il barbiere; se ne arrivano altri si accomodano sulle sedie d'attesa fino al loro esaurimento. Come vengono evitate le corse critiche?

Attraverso 3 semafori, uno conta i clienti in attesa, uno verifica lo stato di attività del barbiere e un mutex che controlla la mutua esclusione sulla poltrona. A questa si aggiunge una variabile che conta i clienti.

# Sincronizzazione con Java:

Java implementa un meccanismo simile al monitor per garantire la sincronizzazione fra thread dove ogni oggetto ha un lock associato ad esse; nelle classi possono essere definire metodi **synchronize** con un solo thread alla volta che può eseguire metodi synchronized. Se un oggetto ha più di un metodo synchronize comunque solo uno può essere attivo.

**N.B.** non vi è nessun controllo sui metodi non synchronized.

La chiamata ad un metodo synchronized richiede il possesso del lock e se un thread chiamante non possiede il lock (un altro thread ne è già in possesso) il thread viene sospeso in attesa del lock.

Il lock è rilasciato quando un thread esce da un metodo sincronizzato Un thread può decidere di non pre-

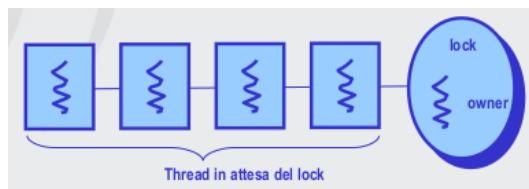


Figura 50: Lock e thread sincronizzati

oseguire l'esecuzione e chiamare volontariamente il metodo `wait()` all'interno del metodo sincronizzato (chiamarlo in un altro contesto genera un errore). Quando viene generato un wait si generano i seguenti eventi:

1. Il thread rilascia il lock
2. Il thread viene bloccato
3. Il thread è posto in una coda d'attesa
4. Gli altri thread possono ora competere per il lock

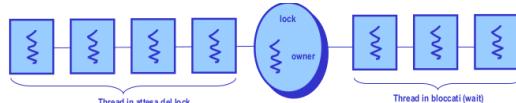


Figura 51: Schema con thread in attesa e thread in wait

## Notify:

Nel momento in cui una condizione che ha provocato una attesa si è modificata si può riattivare un singolo thread in attesa tramite `notify()`:

1. viene selezionato un thread arbitrario T fra i threads bloccati
2. T torna fra i thread in attesa del lock
3. T diventa eseguibile

E' possibile inoltre chiamare sincronizzazioni multiple attraverso diversi metodi:

- `notify()` → seleziona un thread arbitrario e non sempre è il comportamento desiderato
- `notifyAll()` → risveglia tutti i thread in attesa. In questo modo si permette ai thread stessi di decidere chi deve proseguire. E' una strategia conservativa utile quando più thread sono in attesa

<ul style="list-style-type: none"> <li>• Monitor</li> <li>• Un solo processo può entrare nel monitor ed eseguire una sua procedura</li> <li>• Variabili condizione</li> <li>• <code>wait(variabile_condizione)</code></li> <li>• <code>signal(var. condizione)</code></li> </ul>	<ul style="list-style-type: none"> <li>• (Qualunque) oggetto</li> <li>• Un solo thread alla volta può eseguire uno dei metodi <code>synchronized</code></li> <li>• Una sola implicita (<code>this</code>)</li> <li>• <code>this.wait()</code></li> <li>• <code>this.notify() / notifyAll()</code></li> </ul>
--	--

Figura 52: Confronto tra monitor e Java

### Confronto Monitor/Java:

**Sleep():** esiste anche il metodo `sleep(ms)` con il thread che si sospende senza (non richiede l'utilizzo del processore) e nel frattempo possono andare in esecuzione thread a bassa priorità.

**N.B.** sleep non restituisce il processo del lock, è un metodo statico della classe Thread mentre Wait è un metodo della classe Object.

### Sincronizzazione di un blocco:

Si definisce **scope** di un lock il tempo tra la sua acquisizione ed il suo rilascio. Tramite la parola chiave `synchronized` si possono indicare blocchi di codice piuttosto che un metodo intero. Ne risulta uno scope generalmente più piccolo rispetto alla sincronizzazione dell'intero metodo.

### Deamon Threads:

Sono thread che effettuano operazioni ad uso di altri thread, e.g. GarbageCollector. Sono eseguiti in background e usano cioè il processore solo quando altrimenti il tempo macchina andrebbe perso.

A differenza degli thread non impediscono la terminazione di una applicazione e quando sono attivi solo thread\_daemon il programma termina. Occorre specificare che un thread è un deamon prima dell'invocazione dello `start()`. Esiste inoltre il metodo `isDeamon()` che restituisce true se il thread è un deamon\_thread.

### Classe Timer:

La classe `Java.util.Timer` permette la schedulazione di processi nel futuro, attraverso il costruttore `Timer()`; permette di eseguire un task periodico dopo un ritardo specificato. Esistono anche altri costruttori ed altri metodi, come ad esempio `scheduleAtFixedRate`

# Chiamate di sistema:

Esempio di system call:

```
#include <stdio.h>
#include <sys/fcntl.h>
int main( int argc , char ** argv )
{
    int fd; // file descriptor
    fd = open("nuovo-file", O_WRONLY |
O_CREAT, 0666);
    write(fd, "Hello World\n", 12);
    close(fd);
}
```

Prima che il file possa essere effettivamente letto o scritto deve essere aperto e devono essere controllati i permessi. Se i permessi sono verificati viene ritornato un intero definito **file descriptor**, se l'accesso è negato viene ritornato un codice d'errore.

**Pipe:** una pipe è una specie di pseudofile che può essere usato per collegare due o più processi e appare come la scrittura-lettura da file, gestito come una coda FIFO (un processo produttore che deposita i dati e resta in attesa se la pipe è piena, un processo consumatore che legge i dati e resta in attesa se la pipe è vuota). L'implementazione ricorda la lettura/scrittura file poichè il processo consumatore può leggere i dati scritti dal processo produttore sulla pipe, esattamente come se vi fosse un file in cui un processo scrive e l'altro legge. Una *pipe* senza nome può essere creata ed aperta dal comando `int pipe(int fd[2]);`

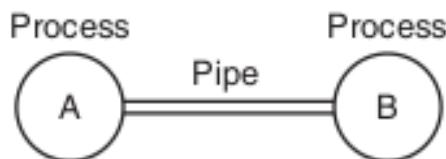


Figura 53: due processi connessi da una pipe

che crea una pipe aprendola in lettura e scrittura e restituendo l'esito dell'operazione, 0 o -1. Assegna a `fd[0]` il file descriptor del lato in lettura e a `fd[1]` quello del lato aperto in scrittura. Si possono inoltre creare pipe con nomi specifici, ma devono essere create attraverso `mknod` e aperte successivamente con `open`. La pipe è infatti analoga ad `open` ma non specifica il nome e produce **file descriptor**, con `read` e `write` che sono tuttavia le stesse in entrambi i casi.

**Uso tipico della pipe:** generalmente una pipe viene usata per far comunicare un processo padrea con un suo figlio, che eredita i file aperti e pertanto anche le pipe (la comunicazione tuttavia è generalmente unidirezionale)

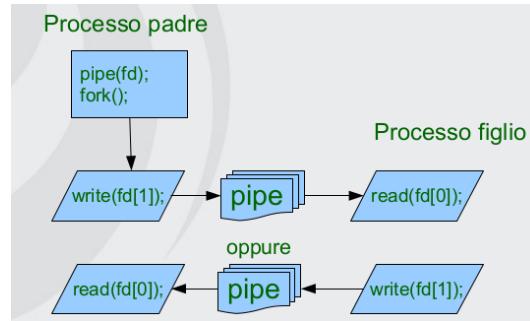


Figura 54: Esempio di pipe tra padre e figlio

```
#include <stdio.h>
int main(int argc, char ** argv)
{
    int fd [2], pid, status;
    char i, j, i1, j1;
    pipe(fd);
    pid = fork();
    if (pid!=0) { // padre
        for (i=0; i<10; i++) write(fd [1], &i, 1);
        printf("scritto!\n");
        waitpid(-1, &status, 0);
        printf("pid=%d i1=%d j=%d\n", pid, i1, j );
    } else { // figlio
        for (j=0; j<10; j++) {
            read(fd [0], &j1, 1); printf("j1=%d\n", j1 );
        }
        printf("pid=%d i=%d\n", pid, i );
    }
}
```

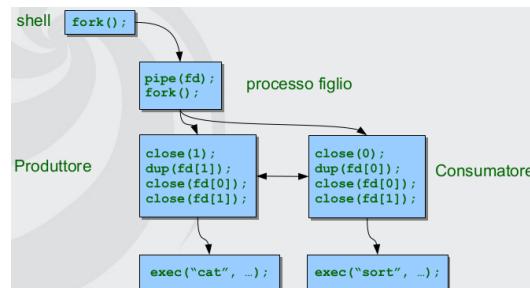


Figura 55: Esempio di pipe (*vedere slide per realizzazione*)

E' possibile anche analizzare la pipe di una redirezione: **NOTA:** è corretto utilizzare come file descriptor STDIN\_FILENO e STDOUT\_FILENO definiti in <unistd.h> al posto di 0 e 1; stdin e stdout non vengono mai aperti né chiusi. Vedere slides per esempi codice

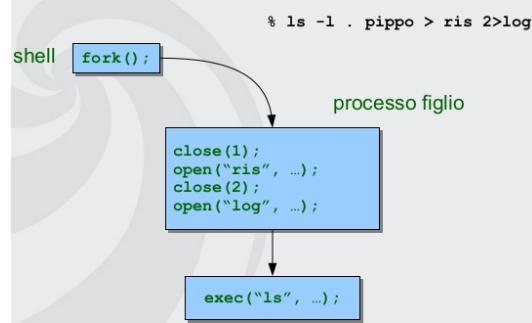


Figura 56: Esempio di pipe in redirezione

# Deadlock:

I sistemi informatici sono pieni di risorse che possono essere usati soltanto da un processo alla volta (e.g. stampanti, dischi etc). Le risorse condivise, in generale, possono generare conflitti (non solo I/O, anche ad esempio database in rete). I deadlock in generale possono verificarsi quando i processi ottengono accessi esclusivi a dispositivi (o, in generale, **risorse**). Esse sono classificate in genere in due categorie:

- Risorse con rilascio → possono essere tolte ai processi senza problemi, e.g. memoria, CPU
- Risorse senza prerilascio → se vengono tolte ai processi si ha il fallimento dell'elaborazione, e.g. stampanti o lettori di dati.

I deadlock normalmente coinvolgono risorse senza prerilascio; la sequenza normale per l'utilizzo di una risorsa è il seguente:

- Richiesta della risorsa
- Utilizzo della risorsa
- Rilascio della risorsa

Se la risorsa non è disponibile il processo viene fatto attendere o, in alcuni sistemi, bloccato automaticamente e poi risvegliato quando la risorsa è nuovamente disponibile. Talvolta può capitare anche che sia il processo stesso a dover gestire la situazione.

**Deadlock:** un insieme di processi è in deadlock se ogni processo dell'insieme è in attesa di un evento che solo un altro processo appartenente allo stesso insieme può causare. Un processo in dl non procede mai e non finisce mai la sua esecuzione, potendo portare in casi estremi anche al blocco dell'intero sistema. Sono state definite 4 condizioni necessarie (*non sufficienti*) affinchè si generi un deadlock:

- **Mutua esclusione:** un solo processo alla volta può utilizzare la risorsa
- **Hold and wait:** i processi che detengono risorse possono chiederne altre
- **No Preemption:** le risorse possono essere rilasciate solo dal processo che le detiene, non può essere forzato
- **Circular Wait:** deve esistere una lista circolare di processi/risorse

**Modellazione:** è stata introdotta una modellazione delle precedenti condizioni attraverso grafi orientati rappresentando come segue:

- QUADRATI: rappresentano le risorse
- CERCHI: rappresentano i processi
- ARCHI: possono connettere solo nodi di tipo diverso; *uscente da processo* indica che il processo ha richiesto la risorsa, *uscente da risorsa* indica che la risorsa è allocata al processo.

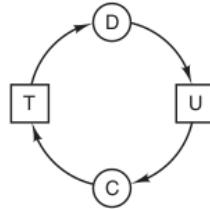


Figura 57: Rappresentazione grafica di un deadlock secondo i grafi orinetati

E' possibile modellizzare anche problemi noti: **N.B.** affinchè si generi un deadlock è necessario che vi

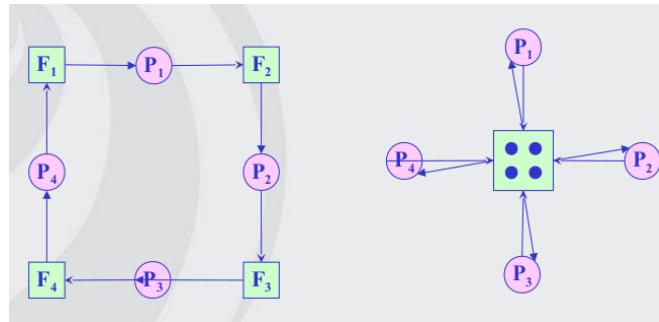


Figura 58: Problema dei filosofi rappresentato con i grafi, a risorse singole o multiple

siano dei *cicli* (sono buoni indicatori di DL), se non vi sono cicli è impossibile che si generi un deadlock, e che non vi siano sufficienti risorse per tutti i processi che le richiedono.

**Strategie:** in generale sono utilizzate 4 strategie per trattare i deadlock:

- Non porsi il problema
- Individuare e risolvere il deadlock
- Prevenire il deadlock dinamicamente
- Prevenire il deadlock impedendo una delle 4 condizioni necessarie

#### Algoritmo dello struzzo:

è l'approccio più semplice e consiste nel non porsi il problema; si può utilizzare quando si è certi che l'eventuale condizione di DL si verifichi con frequenza così bassa da non giustificare implementazioni per la sua gestione o controllo

#### Prevenzione:

la seconda strategia consiste nel rendere impossibile il DL andando ad agire su una delle 4 condizioni necessarie per il suo verificarsi (se infatti si elimina anche solo una condizione il DL diventa impossibile).

### Mutua esclusione:

se non vi sono risorse assegnate esclusivamente ad un singolo processo non avremo mai deadlock. Per questo l'idea è di rendere possibile l'accesso simultaneo → devo gestire casi dove non è banale, e.g. scrittura su file. Per questi casi si utilizza una tecnica detta **Spooling** che si articola come segue:

- Un processo deamon e una directory di spooling vengono create
- Per la stampa su file, ad esempio, il processo genera l'intero file e lo memorizza nella directory di spoofing
- Il deamon è l'unico processo ad avere accesso alla directory di spooling e ha il compito di avviare la stampa.

Si genera tuttavia un possibile problema di deadlock su disco: se infatti due processi provano in contemporanea a scrivere nello spazio di spooling e scrivono metà buffer ciascuno nessuno dei due avrà finito la scrittura ma nessuno può continuare, pertanto avremo deadlock sul disco.

### Hold & Wait:

occorre evitare che processi che detengono risorse rimangano in attesa di ulteriori risorse e si può raggiungere attraverso varie soluzioni:

- Un processo richiede immediatamente tutte le risorse di cui ha bisogno e se non disponibili attende (ci sono però nuovi problemi: un processo non sempre sa quello di cui ha bisogno – l'uso delle risorse non è ottimizzato – vi è il rischio che un processo che ha bisogno di molte risorse rimanga in attesa per tempi molto lunghi)
- Si può imporre anche che un processo rilasci temporaneamente le risorse detenute prima di una nuova richiesta.

### No-Preemption:

la condizione di nessun prerilascio di fatto non è utilizzabile

### Attesa circolare:

l'attesa circolare può essere eliminata in vari modi:

- Un processo può richiedere una sola risorsa per volta
- Le risorse sono numerate e possono essere richieste solo secondo l'ordine numerico
- Una variante richiede semplicemente che la nuova risorsa debba avere una etichetta maggiore di tutte quelle detenute

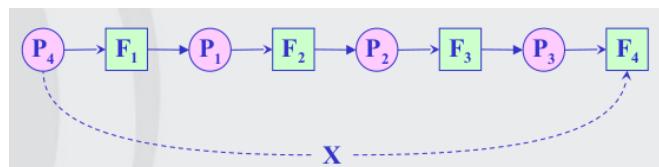


Figura 59: Possibile risoluzione dell'attesa circolare: il filosofo 4 non può richiedere la forchetta 4 prima della forchetta 1

## Evitare il deadlock:

Se si hanno a disposizione determinate informazioni è possibile impedire il verificarsi di un deadlock.

- Il modello più semplice e più utile richiede che ciascun processo dichiari il numero massimo di risorse necessario
- L'algoritmo esamina dinamicamente lo stato di allocazione delle risorse per garantire che non accada mai una attesa circolare
- Lo stato è definito dal numero di risorse disponibili e allocate e dal numero massimo di risorse richieste

### Stato sicuro:

uno stato si dice sicuro se esiste una sequenza di altri stati che porta tutti i processi ad ottenere le risorse necessarie (e quindi terminare), altrimenti è detto non sicuro. Ovvero se c'è un qualche scheduling nel quale ogni processo può eseguire fino al completamento anche se tutti gli altri improvvisamente chiedono il massimo numero di risorse. Per evitare i DL è necessario quindi evitare gli stati non sicuri.

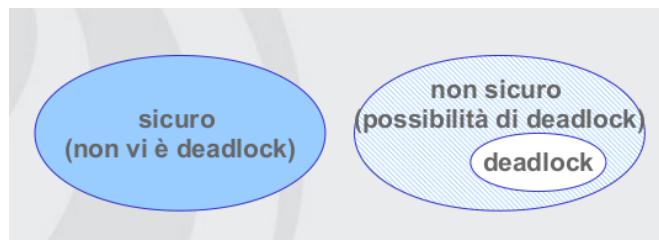


Figura 60: Immagine di come un DL possa verificarsi solo in uno stato non sicuro

	Has	Max												
A	3	9	A	3	9	A	3	9	A	3	9	A	3	9
B	2	4	B	4	4	B	0	-	B	0	-	B	0	-
C	2	7	C	2	7	C	2	7	C	7	7	C	0	-
Free: 3			Free: 1			Free: 5			Free: 0			Free: 7		

Figura 61: Lo stato A è sicuro

	Has	Max									
A	3	9	A	4	9	A	4	9	A	4	9
B	2	4	B	2	4	B	4	4	B	-	-
C	2	7	C	2	7	C	2	7	C	2	7

Figura 62: Lo stato A è insicuro

Si aggiunge alle rappresentazioni anche la rappresentazione della possibile richiesta di un processo ad una risorsa.

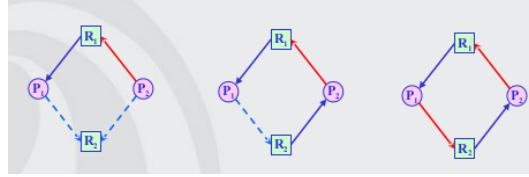


Figura 63: Stati con richiesta

## Algoritmo del banchiere:

E' un algoritmo introdotto da Djikstra che imita il comportamento di un banchiere nei confronti delle richieste dei clienti. Tratta di risorse multiple dove ogni processo deve dichiarare il numero massimo di risorse necessarie e se un processo richiede una risorsa e questa non è disponibile rimane in attesa. Dopo che un processo ha ottenuto le risorse necessarie le deve rilasciare in un tempo finito Ad ogni cliente è concesso un certo numero di unità di credito (e.g. 1000 euro), ovvero le risorse Massime per portare a termine i propri affari. Lo stato del sistema è la quantità di credito Usato in ogni istante. La complessità computazionale di questo algoritmo è  $O(n^2m)$  dove n è il numero di processi e m il numero di tipi di risorse

Nome	U	M	N
Anna	1	6	5
Barbara	1	5	4
Carlo	2	4	2
Davide	4	7	3
Disp.	2		

Figura 64: Esempio di stato sicuro con banchiere

Uno stato si dice sicuro se esiste una sequenza di altri stati che porta tutti i clienti ad ottenere prestiti fino al loro massimo credito (e pertanto a terminare). Nel caso dell'immagine abbiamo uno stato sicuro poichè con due unità Carlo può portare a termine i suoi affari, liberando quattro unità e permettendo la risoluzione dello stato. Se invece che a Carlo si concedesse a Barbara non avremmo più uno stato sicuro poichè nessun cliente riuscirebbe a portare a termine i suoi affari.

## Algoritmo:

una richiesta viene evasa solo se porta in uno stato ancora sicuro, altrimenti il processo deve attendere. L'algoritmo descritto è applicabile ad un sistema con una singola risorsa multipla, ma può essere generalizzato al caso di un sistema complesso con molte classi di risorse; soffre tuttavia del solito problema, ovvero presuppone una conoscenza completa del sistema.

In caso di risorse multiple il ragionamento è il medesimo con l'unica differenza che vengono utilizzati *stato corrente, Massima richiesta, Richiesta necessaria*.

Processi	Corrente	Massima	Necessaria
P0	1 0 1 0 0	2 0 2 1 3	1 0 1 1 3
P1	1 1 0 1 0	1 2 1 2 2	0 1 1 1 2
P2	0 0 0 0 1	2 1 3 1 2	2 1 3 1 1
P3	0 1 1 1 1	0 1 2 1 2	0 0 1 0 1
P4	0 0 1 0 0	1 1 1 2 2	1 1 0 2 2
Disponibili	1 0 1 0 1		

Figura 65: Algoritmo con risorse multiple, solo P3 può ricevere le risorse necessarie

Si liberano poi altre risorse

Processi	Corrente	Massima	Necessaria
P0	1 0 1 0 0	2 0 2 1 3	1 0 1 1 3
P1	1 1 0 1 0	1 2 1 2 2	0 1 1 1 2
P2	0 0 0 0 1	2 1 3 1 2	2 1 3 1 1
P3	0 1 2 1 2	0 1 2 1 2	0 0 0 0 0
P4	0 0 1 0 0	1 1 1 2 2	1 1 0 2 2
Disponibili	1 0 0 0 0		

Quando un processo termina le rilascia e le risorse disponibili aumentano

Disponibili	1 1 2 1 2		
-------------	-----------	--	--

Figura 66: Risorse liberate man mano

E così via

Processi	Corrente	Massima	Necessaria
P0	1 0 1 0 0	2 0 2 1 3	1 0 1 1 3
P1	1 1 0 1 0	1 2 1 2 2	0 1 1 1 2
P2	0 0 0 0 1	2 1 3 1 2	2 1 3 1 1
P3	0 0 0 0 0	0 1 2 1 2	2 1 3 1 1
P4	0 0 1 0 0	1 1 1 2 2	1 1 0 2 2
Disponibili	1 1 2 1 2		

Figura 67: P1 può terminare e liberare le risorse

Procedendo così può poi terminare anche P4 e successivamente P2 e P0. Se esiste una sequenza in cui tutti i processi ottengono tutte le risorse necessarie allora il sistema è in uno stato sicuro.

### Rilevamento dei deadlock:

Si ammette che il sistema possa entrare in uno stato di deadlock e vengono definiti un algoritmo di rilevamento ed un sistema di recovery.

### Rilevamento con risorse singole:

nel caso vi sia soltanto una risorsa per tipo è sufficiente costruire un grafo delle risorse e controllarlo periodicamente. Se sono presenti dei cicli esiste la possibilità di un deadlock (si può anche rimuovere un processo se bloccato da troppo tempo, ma è talvolta impossibile e non sempre porta ad effetti desiderati sul sistema). Un generico algoritmo di questo tipo, con controllo del grafo delle allocazioni richiede una complessità nell'ordine di  $O(n^2)$  operazioni, con  $n = \text{numero di processi}$  del **grafo di attesa**.

Il grafo di attesa si ricava dal grafo di allocazione, rimuovendo le risorse e lasciando soltanto i processi:

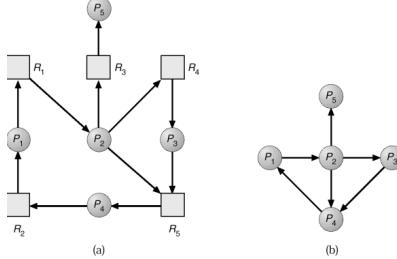


Figura 68: Esempio di grafo delle attese: la freccia indica che il processo ha bisogno della risorsa tenuta dal processo indicato

Il controllo sulla lista avviene in questo modo, per ogni nodo N del grafo:

1. Inizializza L alla lista vuota e segna tutti gli archi come non segnati
2. Aggiunge il nodo corrente alla fine di L e controlla che non compaia due volte. Se ciò avviene siamo in un ciclo
3. Dal nodo controlla se ci sono archi non segnati in uscita, se si passo 5, altrimenti 6
4. Prende un arco non segnato random e lo segna. Segue l'arco fino al nuovo nodo corrente e ricomincia da 3
5. Se il nodo è il nodo iniziale il grafo non contiene cicli e l'algoritmo termina. Altrimenti si è raggiunta una fine. Essa viene rimossa e si continua dal nodo precedente con il passo 3.

### Rilevamento con risorse multiple:

Si utilizza una matrice per rilevare deadlock tra n processi, da  $P_1$  a  $P_n$ . Vengono definite  $E_i$  le risorse  $1 \leq i \leq m$ . E è il vettore delle risorse esistenti e da un numero totale di istanze per ogni risorsa esistente. Ad esempio  $E_1 = 3$  significa che ci sono 3 istanze per la risorsa 1. A è il vettore delle risorse disponibili. Si generano così le matrici di allocazione corrente e la matrice di richiesta. Si utilizza un algoritmo simile a quello del banchiere per rilevare i deadlock che richiede una complessità dell'ordine di  $m * n^2$  con m tipo di risorse e n numero di processi. La frequenza di attivazione dell'algoritmo dipende dalla probabilità che si verifichi un deadlock e dal numero di processi coinvolti.

### Strategie di recovery:

Nel caso di implementazione di un algoritmo di rilevamento deadlock è necessario implementare anche un algoritmo di gestione e recovery da esso. Sono 3 le possibilità principali:

- **Preemption:** talvolta è possibile togliere una risorsa da un processo temporaneamente, assegnarla ad un altro processo dopo aver sospeso il primo e ricominciare il suddetto da dove era stato interrotto una volta finito il secondo. L'implementazione di un sistema di recovery così definito è spesso difficoltoso se non impossibile.
- **Rollback:** se si sa che il sistema può avere deadlock frequenti i processi vengono controllati periodicamente. Controllare un processo significa scrivere il suo stato all'interno di un file così che possa essere riavviato successivamente. Il checkpoint contiene non solo l'immagine della memoria, ma anche le risorse assegnate al processo. Quando viene rilevato un deadlock è così semplice quali

risorse sono necessarie e per eseguire una recovery un processo che possiede una risorsa necessaria è riportato allo stato di tempo precedente a quando ha acquisito tale risorsa.

- **Terminazione processi:** si può implementare un metodo di uccisione di tutti i processi quando si verifica un dl. Non essendo banale l'uccisione di tutti i processi (dal punto di vista delle conseguenze sul sistema) talvolta si preferisce implementare un algoritmo di uccisione di processi iterata fino all'apertura del ciclo che ha generato dl. Il problema maggiore di questa soluzione è l'ordine di uccisione dei processi da scegliere (priorità, tempo d'esecuzione, risorse, numero di processi, processi interattivi...)

**Starvation:** nei sistemi dinamici le richieste di risorse da parte dei processi avvengono continuamente e talvolta può verificarsi una condizione simile al deadlock definita **starvation**. In questa situazione si ha un processo che *idealmente* potrebbe ricevere le risorse ma concretamente non le riceve mai per diversi motivi:

- bassa priorità
- temporizzazione errata
- errato algoritmo di allocazione risorse

Possibili soluzioni alla starvation sono basate su *opportunità* (ogni processo ha una percentuale adeguata di tutte le risorse) o *tempo* (la priorità del processo viene aumentata se il processo attende troppo a lungo). L'introduzione di un semaforo, ad esempio, può risolvere il problema

# Gestione della memoria:

E' necessario, in un sistema informatico, gestire al meglio la memoria. Un programmatore infatti vorrebbe una memoria che sia infinita, veloce, non volatile e poco costosa. Spesso questi punti non possono essere realizzati e anzi, talvolta, sono in contrasto tra di loro. Si sfrutta per questo una **gerarchia della memoria**.



Figura 69: Gerarchia di memoria, ordinata in base alla velocità dell'accesso

Dispositivo di memoria	Velocità di accesso	Capacità	Costo	Volatilità
Registri	1 ns	1 KB	molto alto	alta
Cache	qualche ns	> 1 MB	alto	alta
Memoria principale	10 ns	> 1 GB	10\$/GB	alta
Dischi magnetici	10 ms	1 TB	100\$/TB	bassa
Nastri magnetici	decine di s	< 1 TB	basso	bassa

Figura 70: Tabella di confronto delle memorie in gerarchia

La parte di sistema operativo che si occupa della gestione della memoria è il **gestore della memoria**; il suo compito principale è quello di gestire le differenti memorie (sono infatti implementate all'interno di un sistema operativo diverse tipologie di memoria, le più capienti e lente devono comunicare con le più piccole e veloci). I compiti del gestore sono i seguenti:

- Tenere traccia di quali parti di memoria sono in uso e quali non lo sono
- allocare memoria ai processi che la necessitano e deallocarla
- gestire lo swapping tra la memoria principale e il disco quando la memoria principale non è sufficientemente grande per mantenere tutti i processi
- cercare di sfruttare al meglio la gerarchia delle memorie per ottenere dal sistema le massime prestazioni

Il problema fondamentale da risolvere è il passaggio da **programma eseguibile** su memoria di massa a **processo in esecuzione** in memoria di lavoro.

## Address binding:

L'address binding è il processo di mappatura della memoria logica (virtuale) con il corrispondente indirizzo di memoria fisica (principale). In altre parole un dato indirizzo logico viene mappato ad un indirizzo fisico. Questo può essere fatto in momenti differenti:

- Al momento della compilazione (attraverso un indirizzamento assoluto); se ad un dato momento l'indirizzo del programma deve cambiare è necessario ricompilare interamente il programma (schema utilizzato nei programmi .com del DOS)
- Al momento del caricamento (il codice generato dal compilatore viene detto *rilocabile* ed è il loader che fa le traduzioni opportune)
- Durante l'esecuzione, con il programma che può essere spostato durante la stessa.

Nel caso ad esempio di allocazione a runtime la CPU genera un indirizzo virtuale per un'istruzione/dato da essere collocato in RAM. L'indirizzo logico viene tradotto dall'MMU e l'output è l'indirizzo fisico appropriato della locazione di quella porzione di istruzioni/dati in RAM.

**Dynamic loading:** non sempre viene caricato l'intero programma in memoria; nel caso del dynamic loading, ad esempio, una funzione viene caricata soltanto se viene eseguita a run time: essa viene caricata se necessaria, il programma legge/gestisce le risorse necessarie contenute in essa e terminata la sua necessità libera la sua porzione di memoria.

## Librerie dinamiche:

Alcuni sistemi operativi permettono l'uso delle librerie dinamiche (e.g. le dll in Windows, le .so in Unix): in questo caso le librerie sono linkate tramite il *linker* solo al momento dell'esecuzione del programma. Questa tecnica consente di avere numerosi vantaggi:

- i programmi effettivi (eseguibili) sono molto più piccoli
- l'aggiornamento delle librerie è più semplice (basta sostituirle, non c'è bisogno di ricompilare interamente il programma)
- vi può essere la possibilità che se una procedura è utilizzata da più di un processo se ne possa conservare una sola copia effettiva in memoria.

## Monoprogrammazione:

Lo schema più semplice di gestione della memoria è quello di avere un solo processo alla volta in memoria e consentire al processo di utilizzarla tutta. Questo schema non è utilizzato in quanto implica che ogni processo contenga i driver di periferica per ogni dispositivo I/O utilizzato.

La memoria viene pertanto divisa tra SO e un singolo processo utente, seguendo uno degli schemi dell'immagine:



Figura 71: Tre possibili implementazioni della monoprogrammazione

Possono verificarsi diverse situazioni:

- SO nella RAM in memoria bassa
- SO nella ROM in memoria alta
- Driver di periferica nella ROM e il resto della SO nella RAM

Quando il sistema è organizzato in questo modo solo un processo alla volta può essere in esecuzione

### Multiprogrammazione:

E' vantaggiosa poichè rende più semplice programmare un'applicazione dividendola in due o più processi; fornisce inoltre un servizio interattivo a più utenti contemporaneamente. Viene anche evitato uno spreco di tempo di CPU dato che la maggior parte dei processi passa gran parte del tempo aspettando che vengano completate azioni di I/O del disco, durante questi intervalli in un sistema monoprogrammato la CPU non lavora.

Ipotesi base: supponendo che il processo medio sia in esecuzione per il 20% del tempo che risiede in memoria, con 5 processi in memoria la CPU dovrebbe essere sempre occupata (assumendo ottimisticamente che non siano mai in attesa di I/O i 5 processi). Come organizzo la memoria per poter gestire la multiprogrammazione? La soluzione più semplice è la **multiprogrammazione con partizioni fisse**, con la memoria divisa in N partizioni di grandezza eventualmente diversa.

L'utilizzo della CPU è  $CPU = 1 - p^n$  con  $p = \%$  di I/O

### Code separate:

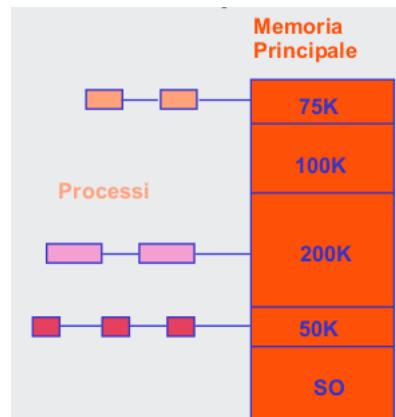


Figura 72: Schema delle code separate in multiprogrammazione

Quando arriva un job viene messo nella coda di input della più piccola partizione in grado di contenarlo.

Questo tuttavia può generare dei problemi: le partizioni infatti sono fisse e quindi solo parzialmente occupate quindi lo spazio non utilizzato dal processo è perso. Inoltre quando la coda di input per una partizione piccola è piena mentre quella per una partizione grande è vuota si ha un grande spreco di risorse.

### Coda unica:

Si può pensare pertanto a partizioni fisse che tuttavia utilizzano una coda di input singola; quando si libera una partizione, il job più prossimo all'uscita della coda e con dimensione inferiore alla partizione viene caricato e poi eseguito. E' necessario tuttavia cercare in tutta la coda il più grande job che può essere contenuto nella partizione, per evitare di sprecare partizioni grandi per job piccoli.

Anche questa soluzione non è esente da problemi: vi è infatti una discriminazione per i job piccoli perché sprecano spazio; supponendo tuttavia i job piccoli come interattivi essi dovrebbero avere il miglior servizio disponibile. Si può risolvere questo problema attraverso due soluzioni, che tuttavia complicano notevolmente l'algoritmo di scheduling:

- Avere almeno una partizione piccola per consentire l'esecuzione dei job piccoli
- Una regola che consenta di stabilire che un job in attesa di esecuzione non venga ignorato più di K volte. Ogni colta che viene ignorato acquisisce un punto e quando ha k punti non può essere ignorato.

### Rilocazione:

Ogni volta che un programma viene *linkato* è necessario che il linker conosca l'indirizzo di memoria corrispondente all'indirizzo del programma; Infatti gli indirizzi generati dal processore su cui è in esecuzione

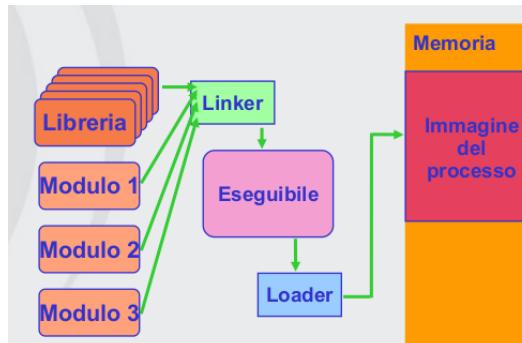


Figura 73: Schema che mostra il caricamento di un programma con linker e loader

un processo sono gli indirizzi logici. Non è detto che corrispondano agli indirizzi fisici dove sono effettivamente disponibili tali informazioni, per questo è necessario un meccanismo che metta in corrispondenza gli indirizzi fisici con gli indirizzi logici. La **rilocazione** statica prevede appunto che all'atto del caricamento del processo il loader provveda a sostituire gli indirizzi logici con i fisici corrispondenti. E' necessario tuttavia prevedere una lista o bit-map che specifichi quali elementi vanno rilocati e quali no. Un altro problema da tenere in considerazione è la **protezione**: i programmi, definendo indirizzi assoluti, possono costruire un'istruzione che legge o scrive qualsiasi parola in memoria e in sistemi multiutente non è accettabile. Per prevenire questo problema l'IBM implementò un controllo attraverso divisione della memoria in blocchi con codice di protezione di 4 bit, con il processo utente che poteva accedere soltanto alla memoria di cui possedeva il codice di protezione.

**Registri base e limite:** quando un processo viene selezionato dallo scheduler per essere eseguito:

- nel registro *base* viene caricato l'indirizzo di inizio della sua partizione

- nel registro *limite* viene caricata la lunghezza del programma.

Ogni volta che un processo fa riferimento alla memoria viene automaticamente sommato alla base l'indirizzo generato dal processo. Viene così controllato che esso non ecceda la lunghezza del registro limite, caso nel quale viene generato un errore e l'accesso è negato.

Il grosso svantaggio di questa gestione è che ad ogni riferimento a memoria è necessario eseguire una somma e un controllo comparativo, e se quest'ultimo è rapido la prima non lo è in termini di tempo della CPU poiché è necessario propagare i riporti (a meno di implementare circuiti specifici).

**Partizioni variabili:** Nei sistemi multiutente normalmente la memoria è insufficiente per tutti i processi degli utenti attivi, ed è pertanto necessario trasferire l'immagine dei processi in eccesso su un disco, tramite un processo detto **swapping**. Il problema con le partizioni fisse è lo spreco di memoria per programmi più piccoli delle partizioni che le contengono, e pertanto si adottano partizioni variabili. Con

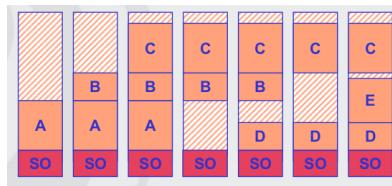


Figura 74: Esempio schematico di partizioni variabili

le partizioni variabili il numero, la dimensione e la locazione delle partizioni varia dinamicamente mentre con le partizioni fisse questi parametri sono stabiliti a priori. Inoltre la flessibilità delle partizioni variabili migliora l'utilizzo della memoria, poiché con le fisse vi è uno spreco maggiore di risorse a causa di partizioni troppo grandi o troppo piccole in relazione ai job. Il difetto delle partizioni variabili è il complicare, rispetto alle fisse, la gestione delle operazioni di allocazione e deallocazione della memoria.

Oltre tutto quando si formano buchi eccessivi in memoria è sufficiente *compattare* la memoria, dove si combinano gli spazi liberi in memoria in un unico grande spazio muovendo tutti i processi in memoria verso il basso.

### Allocazione di memoria:

Quanta memoria dovrebbe essere allocata per un processo quando viene creato o viene portato in memoria tramite swapping? Si utilizzano processi a dimensione fissa in cui viene assegnata al processo esattamente la memoria che necessita. Se si sa invece che i processi tendono a crescere conviene lasciare spazio a disposizione del processo.

Quando un processo cerca di crescere:

- se il processo è adiacente ad uno spazio libero questo può essere allocato
- Se il processo è adiacente ad un altro processo :
  1. può essere spostato in uno spazio di memoria libero sufficientemente grande da contenerlo
  2. uno o più processi dovranno essere trasferiti su disco per creare uno spazio libero abbastanza grande da contenerlo
- se il processo non può crescere in memoria e l'area di swapping su disco è pieno il processo deve aspettare o essere ucciso.

### Gestione della memoria:

Quando la memoria è assegnata dinamicamente il sistema operativo deve gestirla, ed esistono diverse tecniche per farlo.

### Gestione con bitmap:

Attraverso questa tecnica la memoria è divisa in unità di allocazione dove la più piccola è poche words, la più grande è di diversi KB. In corrispondenza ad ogni allocazione c'è un bit nella bitmap settato a 0 se è libera, 1 se è occupata (o viceversa, dipendentemente dalla convenzione) E' importante scegliere nel

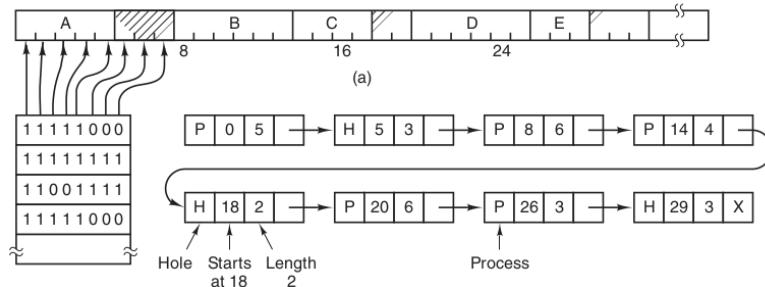


Figura 75: Esempio di bitmap su varie sezioni in corrispondenza con la stessa informazione immagazzinata in una lista

modo migliore la dimensione dell'unità d'allocazione poiché si riflette in modo diretto sull'ottimizzazione dell'occupazione dello spazio e sulla dimensione stessa della bitmap.

La gestione tramite bitmap ha il vantaggio della semplicità per tenere traccia delle parole di memoria in una quantità fissa, ma per poter eseguire un processo con k unità il gestore deve ricercare k zeri consecutivi nella bitmap ed tale ricerca è un'operazione lenta (pertanto le bitmap sono poco usate).

### Gestione con le liste:

Tecnica con la quale si tiene traccia dei segmenti di memoria allocati e liberi; per **segmento** si intende una zona di memoria assegnata ad un processo oppure una zona liberata tra le due assegnate I processi

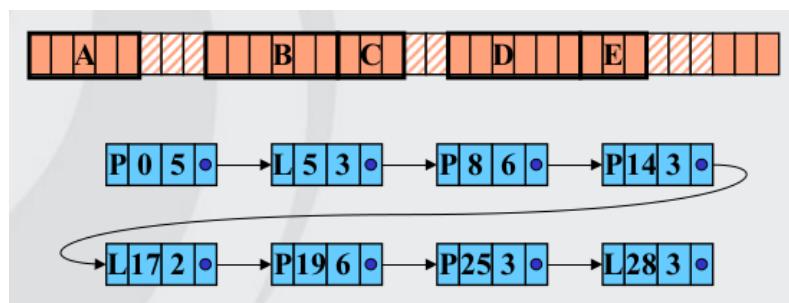


Figura 76: Esempio di inormazione sulla memoria salvata in lista

e le zone libere sono tenuti in una lista ordinata per indirizzo, in questo modo le operazioni di aggiornamento risultano semplificate. La liberazione di una zona di memoria si risolve in quattro casi possibili Mediamente vi è un numero di segmenti liberi pari a metà del numero dei processi e per semplificare le

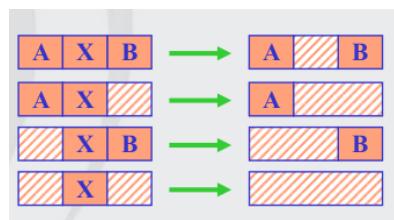


Figura 77: I 4 casi possibili durante liberazione dello spazio

operazioni può essere conveniente utilizzare una lista a doppia concatenazione

**Buddy system:** divide la memoria in partizioni per soddisfare una richiesta di memoria nel miglior modo possibile. Questo sistema divide ricorsivamente la memoria in due metà finchè il blocco ottenuto è grande appena a sufficienza per l'uso, ovvero quando un'ulteriore divisione lo renderebbe più piccolo della dimensione richiesta. E' semplice da implementare soprattutto perchè non richiede alcun hw apposito per la gestione della memoria (MMU).

Tutti gli elementi di memoria della stessa dimensione (quello che viene chiamato buddy in inglese) sono conservati in una lista concatenata ordinata oppure in un albero. Quando un blocco viene rilasciato viene confrontato con il suo vicino: se entrambi sono della stessa dimensione, allora sono combinati ed inseriti nella lista di buddy di dimensione appena più grande. Quando viene richiesto un blocco, l'allocatore comincerà la sua ricerca tra i blocchi di dimensione appena sufficiente, evitando divisioni non necessarie. Rispetto ad altre semplici tecniche di allocazione dinamica della memoria, il buddy system genera alcune frammentazioni quando prova a compattare la memoria. Inoltre il buddy system è soggetto anche a frammentazione interna quando viene assegnata una quantità di memoria maggiore a quella richiesta, ad esempio, se un processo richiede 66K di memoria gliene saranno assegnati 128K, il che si traduce in uno spreco di memoria pari a 62K. Frammentazione esterna quando viene richiesta memoria maggiore alla dimensione del blocco disponibile più grande. In questo caso vi è la suddivisione dello spazio richiesto in due o più pezzi, nessuno dei quali grande abbastanza per soddisfare da solo la richiesta.

### Algoritmi di allocazione:

L'algoritmo più semplice è il **first fit**, per cui viene scansionata la lista finchè non viene trovato uno spazio sufficientemente grande, che verrà poi diviso in due (uno per il processo e uno per la memoria inutilizzata). Una piccola variazione è il **next fit**, che funziona nello stesso modo senonchè tiene traccia di dove si trova quando trova uno spazio libero, in modo da non dover percorrere l'intera coda ogni volta.

Un altro algoritmo di allocazione è il **best fit**, che cerca sull'intera lista e prende il più piccolo spazio adeguato al compito. E' stato dimostrato lasciare più spazi rispetto al first fit. Si può pensare ad una soluzione per questo problema usando il **worst fit**, che prende sempre il più largo disponibile. Ma non è funzionale in ogni caso. Un altro algoritmo è detto **quick fit**, che mantiene liste separate per alcune dimensioni più comuni richieste con puntatori a ciascuna lista, velocizzando notevolmente il processo di ricerca ma rallentando la gestione dello swap.

Una possibile soluzione è il mantenimento di due liste separate per processi(spazio) e spazi liberi, che tuttavia comporta un rallentamento e una maggior ccomplessità quando vengono deallocati spazi di memoria; così facendo gli spazi possono essere ordinati per spazio, per rendere il best fit più veloce.

### Memoria virtuale:

In qualsiasi computer ogni programma può produrre un insieme di indirizzi di memoria; questi indirizzi generati dal programma sono detti **indirizzi virtuali** e formano lo **Spazio di indirizzamento virtuale**. Nei computer senza memoria virtuale l'indirizzo virtuale viene emesso direttamente sul bus di memoria e pertanto la parola di memoria fisica con lo stesso indirizzo viene letta o scritta. Quando viene utilizzata la memoria virtuale gli indirizzi virtuali non vengono mandati direttamente sul bus di memoria ma passano dal chip che mappa gli indirizzi virtuali sugli indirizzi della memoria fisica, il **MMU** (Memory Management Unit).

## Paginazione:

Lo spazio di indirizzamento è diviso in **pagine** mentre le unità nella memoria fisica corrispondenti alle pagine sono detti **frame di pagina**. Pagine e frame hanno sempre la stessa dimensione, normalmente da 512Byte a 16Mbyte, e i trasferimenti tra memoria e disco avvengono sempre per unità di pagina.

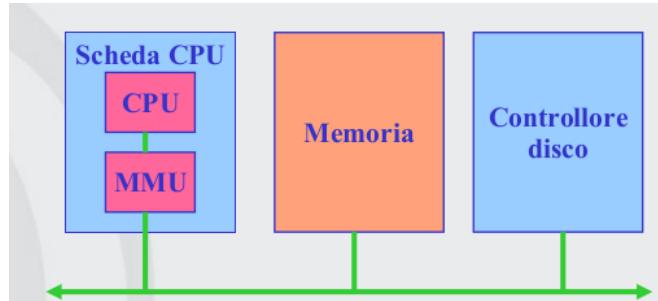


Figura 78: Schema di funzionamento della paginazione

**Page fault:** alcune pagine hw non possono avere corrispondenza in memoria fisica negli attuali circuiti HW un bit presente/assente è utilizzato per tenere traccia se la pagina mappata è presente sulla memoria fisica oppure no. In caso negativo l'MMU genera un'eccezione alla CPU detta **page fault**, a cui seguono alcune azioni:

- Il SO sceglie un frame poco utilizzato
- Salva il suo contenuto su disco (se necessario)
- Recupera la pagina referenziata e la alloca nel frame appena liberato (*fetching* della pagina)
- Aggiorna la mappa
- Riparte con l'istruzione bloccata

**N.B.** non è assolutamente necessario che vi sia corrispondenza tra l'ordinamento della memoria fisica e quella virtuale.

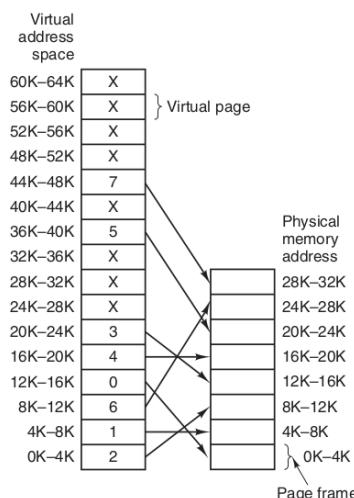


Figura 79: Esempio di non corrispondenza

**Funzionamento MMU:** prendendo un esempio pratico e ipotizzando un indirizzo virtuale di 16 bit, esso viene diviso in:

- un numero di 4 bit
- un offset di 12 bit

Il numero di pagina viene utilizzato come indice nella tabella delle pagine, così da ottenere l'indirizzo fisico. Se il bit *presente/assente* è 0 viene causata un'eccezione. Se il bit è a 1 il numero del frame trovato nella tabella delle pagine viene copiato nei 3 bit di ordine alto del registro di output con i 12 bit di offset (copiati senza modifiche dall'indirizzo virtuale) e il contenuto del registro di output viene messo sul bus di memoria fisica Lo scopo della tabella delle pagine è mappare le pagine virtuali sui frame; vi

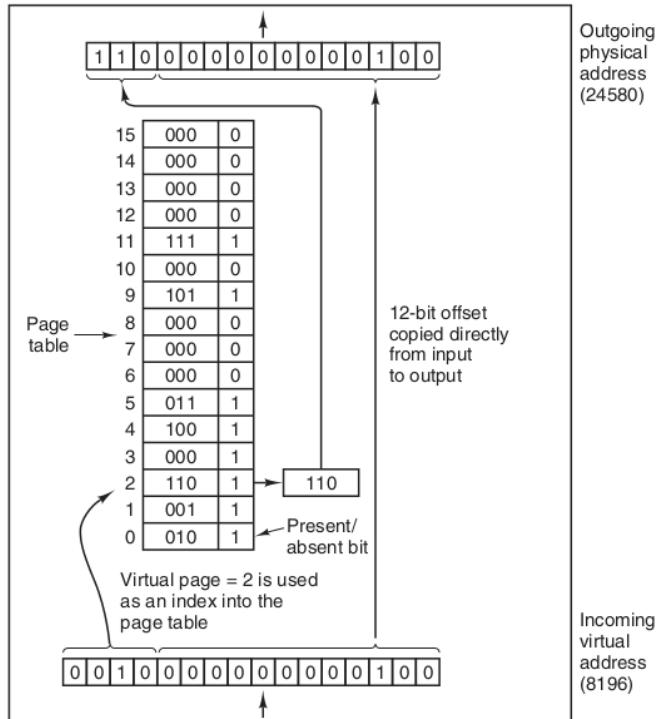


Figura 80: Gestione da parte dell'MMU di un indirizzo

sono tuttavia dei problemi:

- La tabella delle pagine può essere molto grande
- Il mapping da virtuale a fisico deve essere veloce

### Tabella delle pagine:

Il modello più semplice consiste in una **singola tabella delle pagine** costituita da un array di registri hw con un elemento per ogni pagina virtuale indicizzato dal numero di pagina virtuale. Quando viene iniziato un processo il SO carica la sua tabella delle pagine nei registri. Il *vantaggio* è principalmente il mapping immediato senza riferimento in memoria, ma ha *svantaggi* soprattutto per quanto riguarda il costo potenziale del mapping; se la tabella delle pagine è grande il caricamento della tabella delle pagine nei registri ad ogni context switch può alterare le prestazioni (deve esistere una tabella diversa per ogni processo)

All'estremo opposto c'è il mantenere **l'intera tabella delle pagine** in memoria e l'hw necessario è un singolo registro che punta all'inizio della tabella delle pagine. Il *vantaggio* principale è che consente di modificare la mappa di memoria ad un context switch modificando solo un registro, ma ha lo *svantaggio* che durante l'esecuzione di un'istruzione richiede uno o più riferimenti in memoria per leggere gli elementi della tabella delle pagine.

Esiste anche la possibilità di utilizzare una **tabella su più livelli**, evitando così di tenere tutte le tabelle delle pagine in memoria per tutto il tempo. Ogni livello della tabella mappa elementi che possono

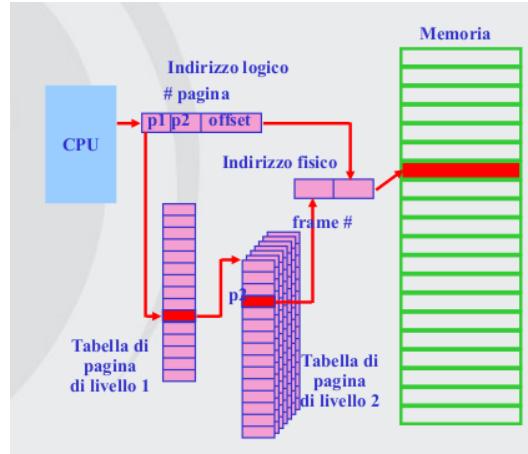


Figura 81: Esempio di una gestione multilivello

trovarsi su più di una tabella del livello successivo.

**Struttura elemento:** Il campo più importante è il numero di frame di pagina, poi c'è il bit presente/assente, i bit di protezione con e informazioni su quali tipi di accesso sono consentiti, i bit di pagina modificata e referenziata per tenere traccia dell'utilizzo e il bit di caching disabilitato è importante per le pagine che mappano su registri di periferiche invece che in memoria

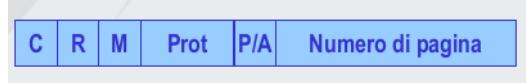


Figura 82: Struttura di ogni elemento della tabella

### Memoria associativa:

Le tabelle delle pagine vengono tenute in memoria per le loro grandi dimensioni, ma ciò può penalizzare le prestazioni (una singola istruzione può fare riferimento a più indirizzi). I programmi tendono a fare la maggior parte dei riferimenti ad un piccolo numero di pagine, quindi solo una piccola parte degli elementi della tabella delle pagine vengono effettivamente letti. Si propone quindi una soluzione attraverso la **memoria condivisa**, o **TLB** (*Translation Lookaside Buffer*) ovvero un dispositivo hw per mappare gli indirizzi virtuali su fisici senza utilizzare la tabella di paginazione. Ogni elemento della memoria associativa contiene informazioni su una pagina, in particolare:

- numero di pagina virtuale
- bit di modifica della pagina
- codice di prenotazione (permessi)
- numero del frame fisico in cui la pagina è situata
- bit che indica se l'elemento è valido o meno

Questi campi hanno corrispondenza 1:1 con i campi nella tabella delle pagine.

**Tabelle delle pagine invertite:**

le tabelle descritte richiedono un elemento per ogni pagina virtuale e con uno spazio di indirizzamento di  $2^{32}$  e pagine di 4K sono necessari  $2^{20}$  elementi (almeno 4M di dati per ogni processo) con 64 bit di indirizzamento la tabella raggiunge la dimensione di milioni di GB. La soluzione è la **tabella delle pagine inverse**; in questo caso la tabella contiene un elemento solo per ogni pagina effettivamente in memoria, con ogni elemento che contiene la coppia (Processo/Pagina virtuale). Il problema è la traduzione degli indirizzi virtuali, occorre infatti consultare l'intera tabella, non un singolo elemento: la soluzione è data dall'uso della memoria associativa, ed eventualmente sfruttare metodi di codifica hash nel caso in cui la pagina cercata non sia nella memoria associativa

# Gestione della memoria\_2:

## Algoritmi di rimpiazzamento:

Per ogni *page fault* il SO deve:

- scegliere una pagina da rimuovere dalla memoria per creare spazio per la nuova
  - se la pagina da rimuovere è stata modificata, deve aggiornare la copia sul disco
  - se non c'è stata alcuna modifica la nuova pagina sovrascrive quella da rimuovere
- scegliendo la pagina da rimpiazzare con un algoritmo per sostituire pagine poco utilizzate (e non a caso) può portare ad un miglioramento delle prestazioni

**Algoritmo ottimo:** è l'algoritmo più semplice concettualmente ma non è implementabile fisicamente. Quando avviene una page fault vi sono alcune pagine in memoria; a queste sono assegnate etichette che identificano *il numero di istruzioni che dovranno essere eseguite prima che la pagina venga referenziata*. Viene poi rimossa la pagina con l'etichetta maggiore. Non è realizzabile in quanto non è possibile cono-

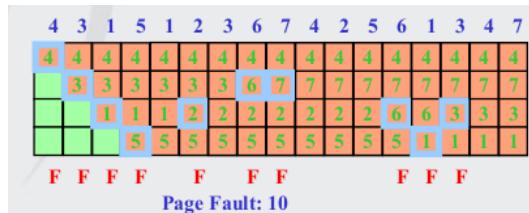


Figura 83: Schema dell'algoritmo ottimo

scere a priori al momento del page fault quali pagine verranno referenziate (sarebbe come prevedere il futuro). Potrebbe essere utile eseguire il programma con un simulatore e tenere traccia dei riferimenti alle pagine (è pertanto usabile per una seconda esecuzione utilizzando le informazioni sui riferimenti alle pagine della prima esecuzione ed è utile come confronto per la valutazione degli algoritmi di rimpiazzamento delle pagine)

**Algoritmo FIFO:** il SO mantiene una lista concatenata di tutte le pagine in memoria con:

- la pagina più vecchia in testa
- la pagina più recente in coda

Quando avviene un page fault viene rimossa la pagina in testa alla lista e la nuova pagina viene aggiunta in coda. In questa forma è usato raramente poiché non è detto che una più vecchia sia effettivamente meno referenziata.

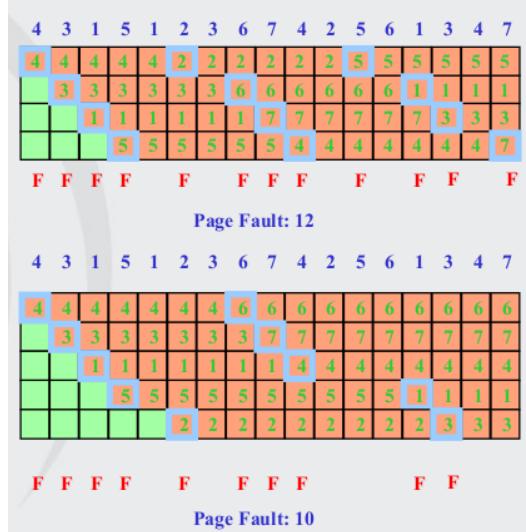


Figura 84: Esempio di gestione FIFO

**Anomalia di Belady:** è un'anomalia che si verifica negli algoritmi di gestione della paginazione FIFO per cui la frequenza di page fault aumenta all'aumentare del numero di frame dedicati ai processi

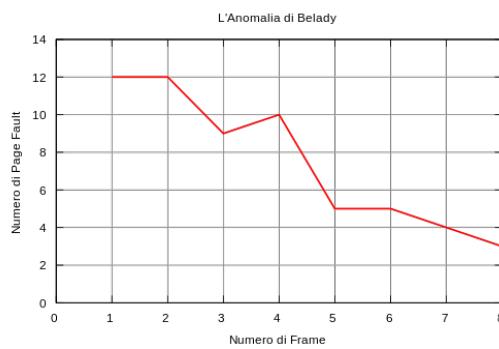


Figura 85: Esempio di grafico d'andamento dell'anomalia di belady

**Algoritmo NotRecentlyUsed (NRU):** La maggior parte dei computer con memoria virtuale prevede 2 bit per la raccolta di informazioni sull'utilizzo delle pagine:

- Il bit R, indica che la pagina è stata referenziata
- Il bit M indica che la pagina è stata modificata

Entrambi sono posti a 0 inizialmente dal SO. Periodicamente il bit R viene azzerato per distinguere le pagine non referenziate recentemente dalle altre. Quando avviene un page fault il SO controlla tutte le pagine e le divide in quattro classi in base al valore dei bit R e M:

- **Classe 0:** non referenziate, non modificate
- **Classe 1:** non referenziate, modificate
- **Classe 0:** referenziate, non modificate
- **Classe 0:** referenziate, modificate

L'algoritmo NRU rimuove una pagina qualsiasi dalla classe di numero inferiore che non sia vuota, ottenendo così prestazioni adeguate (anche se non sempre ottimali, ma mediamente più che accettabili) mantenendo una discreta facilità d'implementazione

**Algoritmo Second Chance:** ottenibile con una semplice modifica dell'algoritmo FIFO, permette di evitare il problema della rimozione delle pagine molto utilizzate; viene controllato il bit R e:

- Se è a 0, la pagina è sia vecchia che non utilizzata e viene rimpiazzata immediatamente
- Se è a 1 il bit viene azzerato e la pagina messa in coda alla lista come se fosse appena stata caricata in memoria

L'algoritmo si basa sulla ricerca di una pagina che non sia stata referenziata nel precedente intervallo di clock. Se tutte le pagine sono state referenziate degenera in un algoritmo FIFO puro

**Algoritmo clock:** l'algoritmo second chance è inefficiente poiché sposta continuamente le pagine sulla lista; un approccio migliore Quando avviene un page fault:

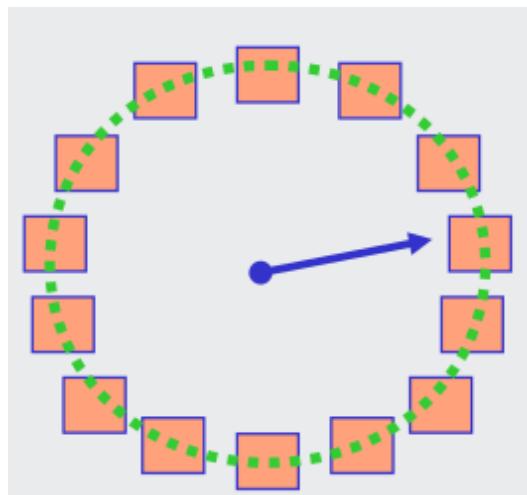


Figura 86: Rappresentazione schematica dell'algoritmo clock

- se il bit R vale 0, la pagina puntata dalla lancetta viene rimossa, la nuova pagina viene inserita nell'orologio al suo posto e la lancetta spostata avanti di una posizione.
- se il bit vale 1 viene azzerato e la lancetta spostata avanti di una posizione

La differenza con l'algoritmo second chance sta nell'implementazione che permette di non rimuovere costantemente pagine.

**Algoritmo LRU(Least Frequently Used):** un modo per provare a rendere realizzabile l'algoritmo ottimo consiste nel ragionare osservando quanto una pagina sia stata utilizzata nel recente passato: se è stata usata molto è probabile verrà usata ancora, se non è stata usata per molto è probabile verrà ignorata ancora. Quando avviene un page fault pertanto si elimina la pagina che non è stata usata per il tempo più lungo. E' tuttavia un algoritmo non banale nell'implementazione e soprattutto trovare la pagina meno usata, eliminarla, rimpiazzarla e muoversi in testa alla lista è un job dispendioso. La soluzione al problema è utilizzare hardware dedicato per migliorare l'efficienza dell'LRU. Come si vede dalla

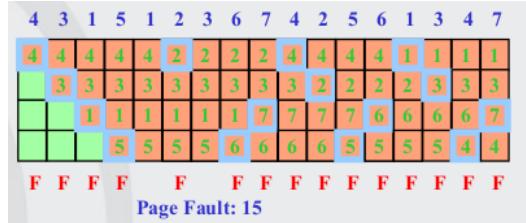


Figura 87: Andamento nel tempo dell'algoritmo LRU

0	0111	0011	0001	0000	0000
1	0000	1011	1001	1000	1000
2	0000	0000	1101	1100	1101
3	0000	0000	0000	1110	1100
0	0000	0111	0110	0100	0100
1	1011	0011	0010	0000	0000
2	1001	0001	0000	1101	1100
3	1000	0000	1110	1100	1110

Figura 88: Sequenza di pagine 0 1 2 3 2 1 0 3 2 3

precedente immagine ad ogni accesso viene messa ad UNO la riga corrispondente alla pagina; viene poi messa a ZERO la colonna corrispondente. All'occorrenza di una page fault valuto gli indici man mano (prima quello precedente, se ce ne sono due a ZERO guardo quelli ancora precedenti fino a trovare la pagina meno utilizzata)

**Algoritmo NFU(Not Frequently Used):** ad ogni pagina viene associato un contatore inizialmente posto a 0; ad ogni clock viene sommato al contatore il bit R e al momento del pagefault viene rimpiazzata la pagina con contatore minimo. Il problema è che NFU non dimentica nulla e se una pagina è stata molto utilizzata non verrà più rimossa anche se non più utile.

E' stata proposta una versione migliorata con introduzione dell'**invecchiamento**: ad ogni clock il contatore scorre a destra introducendo R come bit più significativo

Bit R	101011	110010	110101	100010	011000
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00010000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	t=1	t=2	t=3	t=4	t=5

Figura 89: Schema di andamento dell'algoritmo NFU

**Modello a Working Set:** nel modello più semplice di paginazione le pagine vengono caricate in memoria solo al momento del page fault, ovvero si parla di **paginazione on demand**. La maggior parte dei processi esibisce la località di riferimenti, ovvero durante qualsiasi fase d'esecuzione il processo fa riferimenti ad un piccolo insieme di pagine. L'insieme di pagine che un processo sta correttamente utilizzando viene detto *working set*. Più alto è il numero pagine in memoria più è probabile che il working set sia completamente in memoria (e quindi non avverranno page fault). Un programma che causa page fault per ogni piccolo insieme di istruzioni viene detto **in thrashing**.

Il WS con parametro  $\delta$  all'istante  $t$ ,  $W(\delta, t)$  è l'insieme delle pagine a cui ci si è riferiti nelle ultime  $\delta$  unità di tempo. Possiede alcune proprietà:

$$W(\delta, t) \subseteq W(\delta + 1, t)$$

$$1 \leq |W(\delta, t)| \leq \min(\delta, N)$$

Con  $N$  numero di pagine necessarie al processo Poichè il WS varia molto lentamente nel tempo è pos-

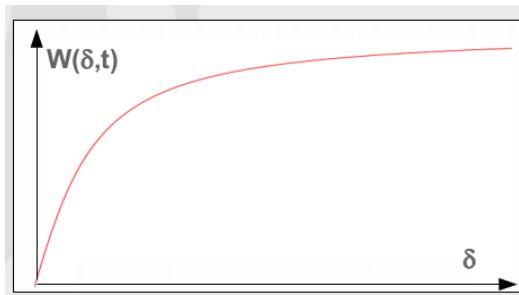


Figura 90: Grafico di andamento della dimensione dell'WS

sibile cercare di indovinare ragionevolmente quali pagine saranno necessarie quando il programma verrà richiamato; sarà così possibile caricare tali pagine prima di riprendere il processo. Sarà inoltre possibile implementare un algoritmo per cui all'avvenire di un page fault si andrà alla ricerca innanzitutto di pagine non appartenti all'WS per eliminarle per prime. (**obs:** ogni volta che avviene una variazione di dimensione nell'WS è perchè avviene un page fault)

Sequenza di riferimenti	Dimensione della finestra (2+5)
24	24
15	24 15
18	24 15 24
23	24 15 18
24	24 15 18 23
17	24 15 18 23 24
18	24 15 18 23 24 17
24	24 15 18 23 24 17 15
18	24 15 18 23 24 17 18
17	24 15 18 23 24 17 18 -
17	24 15 18 23 24 17 18 -
15	24 15 18 23 24 17 18 -
24	24 15 18 23 24 17 18 -
17	24 15 18 23 24 17 18 -
24	24 15 18 23 24 17 18 -
18	24 15 18 23 24 17 18 -

Figura 91: Andamento del WS in un esempio pratico

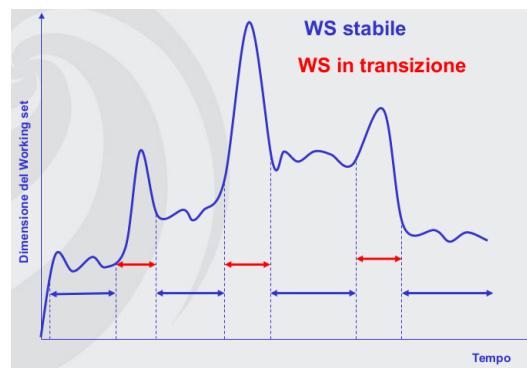


Figura 92: Grafico di andamento dell'WS

### Prestazioni & page fault:

Qual è la perdita di prestazioni per un tasso di page fault 1/100000? Access time senza page fault è di circa 100ns. Il tempo di accesso alla memoria è, nel caso di rimpiazzamento del 50%, pari a  $0.5 \cdot 20\text{ms} + 0.5 \cdot 40\text{ms} = 30\text{ms}$ .

Si parla di **EAT** (effective Access Time) data una probabilità di fault "p"

$$EAT = p * \text{accesso\_con\_page\_fault} + (1 - p) * \text{accesso\_senza\_page\_fault}$$

Prendendo un caso possibile:  $0.00001 * 30 \text{ ms} + (1 - 0.00001) * 100 \text{ ns} = 300 \text{ ns} + 99.999 \text{ ns} = 400 \text{ ns}$  con una perdita di prestazioni del 300%

Qual è la percentuale di fault per un 10% di prestazioni ( $EAT = 110\text{ns}$ )?

$110\text{ns} > p * 30\text{ms} + (1-p) * 100\text{ns}$  dove contando  $1-p$  approssimabile a 1 ottengo:  
 $10\text{ns} > 10 / 30'000'000$

### Località:

Per località si indica che il prossimo accesso di memoria sarà nelle vicinanze di quello corrente, secondo:

- *località spaziale*: il prossimo indirizzo differirà di poco
- *località temporale*: la stessa cella (o pagina) sarà molto probabilmente riutilizzata nel prossimo futuro

La località è la ragione che giustifica il basso numero di page fault.

```
int mat[1024][1024];
max=mat[0][0];
for (i=0; i<1024; i++)
    for (j=0; j<1024; j++)
        if (max<mat[i][j])
            max = mat[i][j];
```

Si nota località spaziale poichè dopo `mat[10][101]` utilizzo `mat[10][102]`.

```
int mat[1024][1024];
max=mat[0][0];
for (i=0; i<1024; i++)
    for (j=0; j<1024; j++)
        if (max<mat[j][i])
            max = mat[j][i];
```

Questa soluzione potrebbe causare numerosi pagefault poichè non rispetta il principio di località poichè dopo l'elemento [10][101] esamino [11][101]. NOTA: non possiamo dare per scontato che la scansione sia effettivamente fatte riga per riga, dipende dal linguaggio(e.g. FORTRAN memorizzava colonna per colonna).

### Strategia di fetch:

Stabilisce quando una pagina deve essere trasferita in memoria:

- La paginazione su richiesta (**on demand**) porta una pagina in memoria solo quando si ha un riferimento a quella e si hanno molti page fault quando un processo parte
- **prepaging** porta in memoria più pagine del necessario (prima che la pagina serva effettivamente, sfruttando WS e località) ed è più efficiente se le pagine su disco sono contigue

**Prepaging:** per risolvere il problema del thrashing molti sistemi a paginazione utilizzano il prepaging prima di eseguire un processo, caricando in memoria le pagine relative al WS. E' necessario, per poter implementare il modello WS, è necessario che il SO tenga traccia di quali pagine sono nel WS stesso e per controllare questa informazione può essere implementato *l'algoritmo di invecchiamento*. Le informazioni dell'WorkingSet inoltre possono essere utilizzate per migliorare le prestazioni dell'algoritmo clock (questo nuovo algoritmo viene detto WSClock): la pagina viene sostituita **solo** se non appartiene all'WS (di fatto combina algoritmo clock con il WS)

**Algoritmi Locali e Globali:** gli algoritmi **locali** assegnano quantità fisse di pagine ai processi e sono algoritmi in cui ogni processo è associato ad un certo numero di pagine per cui ad un page fault sostituisco una pagina con un'altra che mi serve. Gli algoritmi visti tuttavia possono essere applicati a livello **globale**, ovvero considerando tutti i processi gestiti in quel momento dal sistema operativo (dal momento in cui ho bisogno di una nuova pagina non è detto che io sostituisca pagina di un processo corrente ma può capitare di sostituire pagina processi non in esecuzione. → numero di pagine associate ad un processo varia dinamicamente.

Un approccio di questo tipo interagisce con il thrashing dipendentemente dal grado di multiprogrammazione che ho deciso di mantenere sul processo, considerando ad esempio il tempo di utilizzo CPU

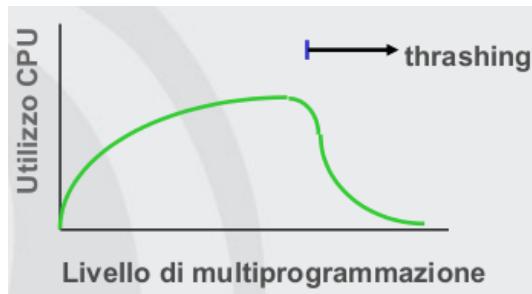


Figura 93: Aumentando multiprogrammazione fino ad un certo livello è lineare, dopo un certo valore tuttavia si ha un crollo delle prestazioni perchè aumentare il numero di processi vuol dire diminuire memoria per ogni processo ed aumentare la possibilità di page fault

**Frequenza page fault:** utilizzando la gran parte degli algoritmi la frequenza di page fault decresce all'aumentare del numero di pagine assegnate Se il numero di processi è troppo grande per mantenerli tutti

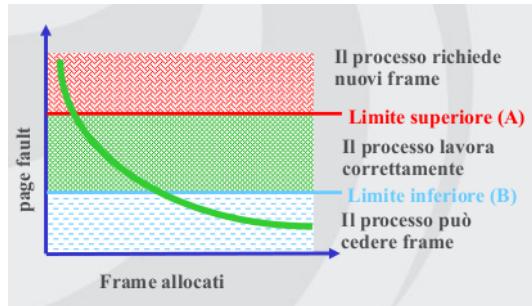


Figura 94: L'algoritmo page fault frequency cerca di mantenere gli errori in un intervallo ragionevole

sotto **A** qualche processo viene rimosso; se il numero di processi è sotto **B** allora ha troppa memoria e parte delle sue pagine possono essere utilizzate da altri processi, pertanto alcune pagine vengono riallocate.

Si può utilizzare un processo in background detto **paging deamon** che viene risvegliato periodicamente per ispezionare lo stato della memoria. Se il numero di frame liberi in memoria non è sufficiente si occupa di selezionare tramite l'algoritmo di rimpiazzamento pagina scelto, le pagine da eliminare riscrivendole su disco se sono state modificate. Il vantaggio è che questo processo ottimizza la ricerca e sostituzione rispetto alla ricerca di un frame solo quando necessario. Se inoltre utilizza tecniche di prepagina è anche probabile che la pagina sia precaricata.

**Migliore dimensione pagina:** la dimensione delle pagine è un parametro del sistema e le motivazioni hanno diversi vantaggi in base alla scelta:

- **piccole:** un blocco di memoria non riempirà esattamente un numero intero di pagine, mediamente quindi metà dell'ultima pagina viene sprecata (è più ottimizzata l'occupazione in memoria)

- **grandi:** con pagine piccole è necessaria una tabella delle pagine grande e il trasferimento di una pagina piccola da disco richiede circa lo stesso tempo di una grande

**E.g.**  $s$  = dimensione processo (128KB);  $p$  = dimensione pagine;  
 $e$  = dimensione elemento nella tabella delle pagine (8Byte)

$$s/p = \text{numerodipagineinmemoria}$$

e quindi

$$s * e/p = \text{dimensionetabella}$$

$$p/2 = \text{memoriasprecata}$$

Il costo è dato da

$$\text{costo} = s * e/p + p/2$$

Derivando rispetto a  $p$  ottengo

$$-s * e/p^2 + 1/2 = 0$$

Da cui  $P = \sqrt{2 * s * e}$

La maggior parte dei sistemi operativi usa dimensioni di pagina da 512 byte a 64KByte.

**Interfaccia della memoria virtuale:** in alcuni sistemi avanzati i programmatore hanno un certo controllo sulla mappa di memoria: questo permette a due o più programmi di condividere la stessa memoria: se è possibile dare un nome ad una zona di memoria un processo può darlo ad un altro processo in modo che quest'ultimo possa inserire la pagina nella sua tabella. La condivisione delle pagine può implementare un sistema a scambio di messaggi ad elevate prestazioni.

**Bloccare le pagine in memoria:** la memoria virtuale e le periferiche di I/O interagiscono in diversi modi ed è necessario implementare una gestione delle periferiche: immaginando ad esempio un processo che ha lanciato una system call per leggere da qualche file o device in un buffer entro il suo indirizzo; aspettando che finisce l'operazione di I/O il processo è sospeso e un altro processo può runnare; quest'ultimo otterrebbe un page fault. Se l'algoritmo è globale c'è una possibilità piccola (ma non nulla) che la pagina contenente il buffer I/O sia scelta per essere rimossa dalla memoria.

Se un I/O device sta eseguendo un trasferimento DMA a quella pagina, rimuoverla causerebbe una perdita di dati (scritti invece che nel buffer nella pagina che ha preso il suo posto). Una soluzione è bloccare le pagine di memoria di processi I/O affinché non sia rimossa. Il bloccaggio di una pagina in memoria è detto **pinning**. Un'altra soluzione è l'eseguire tutto l'I/O all'interno del buffer del kernel ed eseguire successivamente la copia alle pagine utente.

**Memoria su disco:** esistono anche dei problemi legati all'area su disco in cui salvare le pagine e alla sua gestione. L'algoritmo più semplice consiste nell'avere spazio per allocare le pagine soffotorma di una partizione specifica all'interno del disco (o ancora meglio su un disco a parte) separata dalla partizione del file system (la maggior parte dei sistemi Unix lavora in questo modo). Questa partizione non ha un normale file system al suo interno ma essa viene interamente usata per blocchi relativi alle pagine. Quando il sistema è avviato questa partizione è vouta e rappresentata in memoria come una singola entry della dimensione del suo spazio originale. Nella forma più semplice quando un processo è avviato uno spazio della partizione grande come il processo viene allocato e l'area rimanente ridotta di quella dimensione; quando il processo termina viene liberata tale area. Associata ad ogni processo è l'indirizzo del disco della sua area di swap. Questa informazione è mantenuta nella tabella del processo.

A causa della possibilità dei programmi di crescere nello stack è sempre meglio implementare un'area di swap in cui vengono separati gli elementi di text, dati e stack e permettere a ciascuna di queste aree di occupare più di uno spazio su disco.

L'estremo opposto consiste nel non allocare nulla in anticipo e allocare lo spazio su disco per ogni pagina quando viene swappata e deallocarlo quando viene terminata;

Il problema dell'allocazione dinamica è la necessità di riscrivere continuamente la tabella di corrispondenza tra indirizzi; tuttavia avere una dimensione fissata dell'area di swap non sempre è possibile (in questo caso, come fa Windows, si utilizzano file preallocati all'interno del file system)

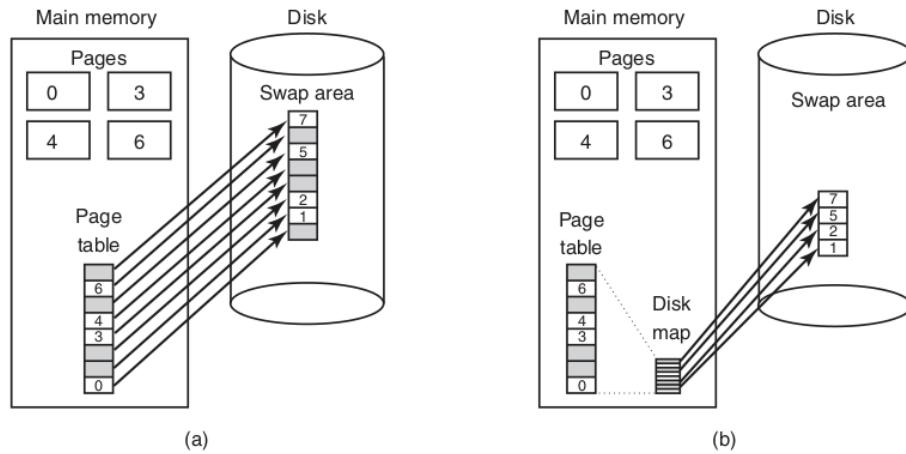


Figura 95: Due algoritmi possibili: (a) con area su disco di dimensione fissata e (b) con dimensione variabile allocata dinamicamente dove si nota la necessità del disk map

**Memory mapped files:** è un segmento di memoria virtuale che è stato assegnato ad una correlazione byte-to-byte come alcune porzioni di risorse file (che può essere anche una risorsa identificata come pseudofile. Una volta presente questa correlazione tra file e memoria permette alle applicazioni di trattare la porzione mappata come fosse memoria primaria. Di fatto operazioni come ad esempio l'I/O possono essere trattate come un normale accesso in memoria. Quando si richiede l'uso di un blocco su disco questo viene caricato in memoria e l'accesso risulta semplificato in quanto non si usano più le chiamate di sistema `read()` e `write()`, pertanto più processi possono condividere in memoria gli stessi file con un netto miglioramento delle performance.

Esempio di codice:

```
#include <sys/mman.h> // Memory Management
int main( int argc , char *argv [] )
{
    char *pt , pt1[1024];
    int fd , len=sizeof(pt1) , offset=0;
    fd = open( _FILE_ , O_RDONLY );
    pt = mmap(NULL, len , PROT_READ,MAP_SHARED, fd , offset );
    memcpy(pt1, pt, len );
    return 0;
}
```

## Segmentazione:

Con l'approccio precedente la memoria virtuale è unidimensionale (indirizzi sempre da 0 all'indirizzo massimo) e può essere utile mantenere due o più spazi di indirizzamento separati. *E.g. → un compilatore ha molte tabelle che vengono costruite durante la compilazione e che crescono durante la compilazione; in uno spazio di indirizzamento unidimensionale ci possono essere tabelle con molto spazio libero e altre completamente occupate o addirittura che necessitano di altro spazio. Ci sono alcune possibili soluzioni ma ciò che effettivamente serve è liberare l'utente dal compito di gestire tabelle che si espandono e contraggono.* La soluzione a questo tipo di problemi è quella di fornire alla macchina spazi degli indirizzi

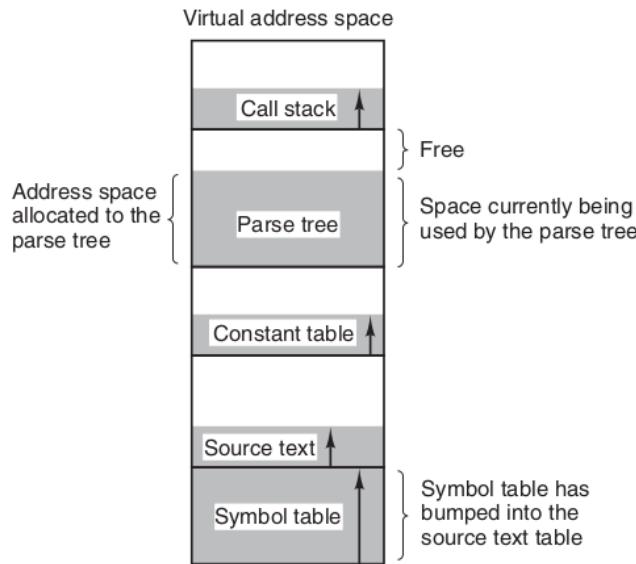


Figura 96: Esempio di come spazio degli indirizzi unidimensionale non sia efficiente

indipendenti definiti **segmenti**. Ogni segmento consiste di una sequenza lineare di indirizzi che iniziano da 0 e salgono fino ad un valore massimo. La lunghezza di ogni segmento può essere ogni valore tra 0 e il massimo consentito. E inoltre segmenti differenti possono (e spesso hanno) lunghezze differenti.

Poiché ogni segmento rappresenta di fatto porzioni differenti di memoria, ognuno può crescere o decrescere indipendentemente dagli altri. Per specificare un indirizzo è necessario specificare un indirizzo in due parti: una per identificare il segmento, l'altra per identificare la porzione al suo interno.

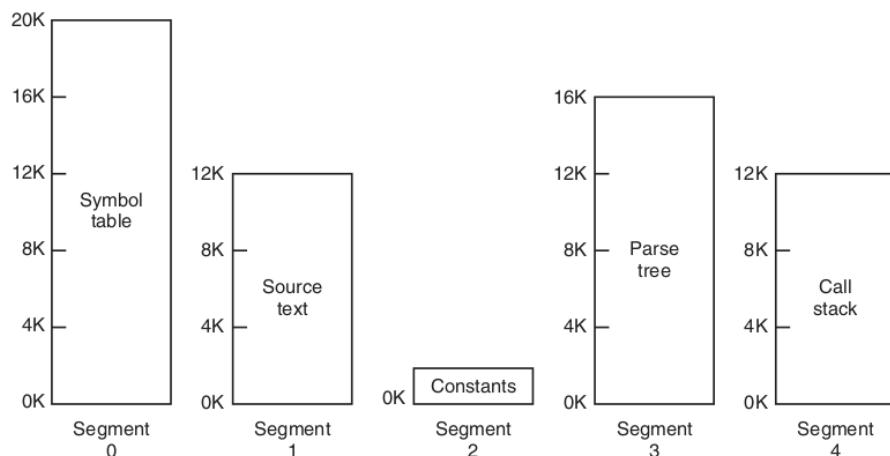


Figura 97: Esempi di segmenti (*nota: solitamente i segmenti contengono elementi omogenei*)

## Vantaggi:

semplifica la gestione di strutture dati che crescono o diminuiscono; se ogni procedura occupa un segmento separato con indirizzo di inizio all'interno di segmento pari a 0, la segmentazione semplifica il linking di procedura compilate separatamente. Facilita inoltre la condivisione di procedure o dati tra processi e dato che ogni segmento è un'entità logica nota al programmatore (procedura, array, stack...) segmenti differenti possono avere differenti tipi di protezione.

Considerazioni	Paginazione	Segmentazione
Il programma è a conoscenza del metodo?	No	Sì
Quanti spazi di indirizzamento ci sono?	1	Molti
È possibile che lo spazio di indirizzamento superi la dimensione fisica della memoria?	Sì	Sì
È possibile distinguere e proteggere separatamente procedure e dati?	No	Sì
È facile gestire tabelle di dimensioni variabili?	No	Sì
Facilita la condivisione di procedure fra gli utenti?	No	Sì
Perché è stato proposto questo metodo?	Per avere uno spazio di indirizzamento di grandi dimensioni	Per poter distinguere codice e dati in spazi di indirizzamento logicamente separati e per facilitare la condivisione e la protezione

Figura 98: Confronto tra paginazione e segmentazione

**Implementazione segmentazione pura:** L'implementazione della paginazione e segmentazione ha una differenza fondamentale: le pagine hanno dimensione fissa, i segmenti variabile. Se i segmenti vengono continuamente caricati e sostituiti in memoria dopo un certo lasso di tempo verranno a formarsi zone inutilizzate di memoria (con il fenomeno noto come **checkboarding**, ovvero la frammentazione esterna). Una possibile soluzione è la compattazione.

**Segmentazione con paginazione:** se i segmenti sono molto grandi è difficile tenerli tutti in memoria, per questo si utilizza una tecnica mista che pagina i segmenti. Nell'implementazione MULTICS ogni programma ha una tabella dei segmenti con un descrittore per segmento (che contiene un indicazione riguardo alla presenza o meno del segmento in memoria). Se il segmento è in memoria il descrittore contiene un puntatore a 18 bit alla tabella delle pagine. Poiché gli indirizzi fisici erano di 24 bits e le pagine erano allineate con confini di 64 byte, solo 18 bits erano necessari al descrittore. Il descrittore contiene anche informazioni riguardo la dimensione della pagina, i bit di protezione e altre info. Quando avviene una referenza in memoria viene eseguito il seguente algoritmo:

1. Il numero del segmento è usato per trovare il descrittore del segmento
2. Viene fatto un controllo per vedere se la pagina del segmento è in memoria. Se non è presente avviene un segfault, se è presente viene localizzata.
3. Viene esaminata l'entry della tabella per la pagina virtuale richiesta
4. Viene aggiunto l'offset alla pagina origine per trovare la word
5. Viene eseguita la read()/store()

Un'altra implementazione è quella dei moderni Intel x86, in cui il cuore è costituito da due tavole chiamate **LDT** (Local Descriptor Table) e **GDT** (Global Descriptor Table). Ogni programma ha la

propria LDT, ma c'è una singola GDT. La prima descrive i segmenti locali di ogni programma, condivisi da ogni programma. La seconda descrive i segmenti di sistema, compreso il s.o. stesso. Per accedere un segmento un programma x86 prima carica un settore per quel dato segmento in uno dei sei registri di segmenti. Durante l'esecuzione il CS register mantiene il selettore sul codice segmento e il DS register mantiene il selettore sul segmento dati.

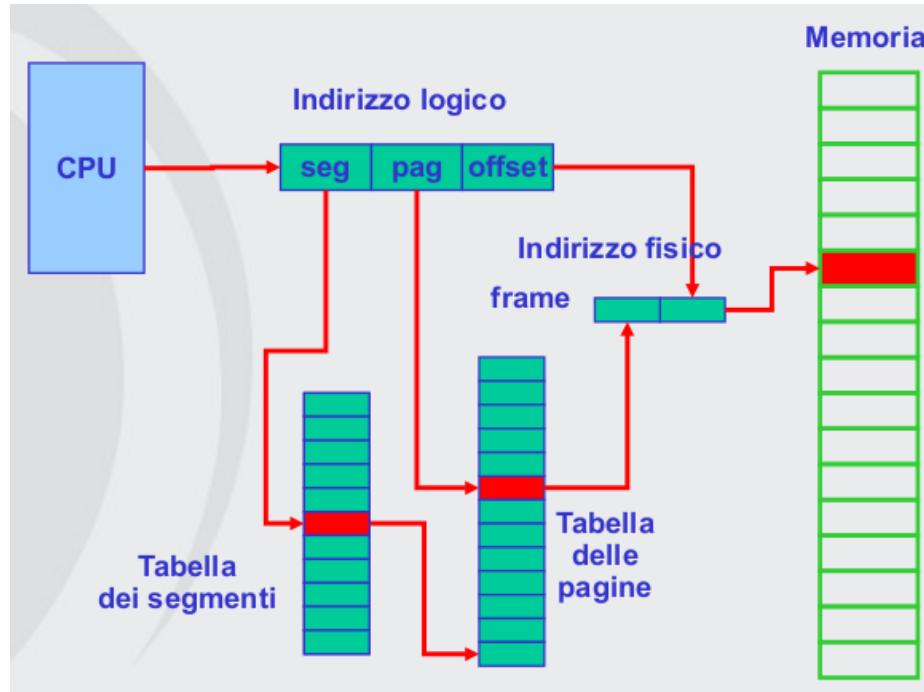


Figura 99: Segmentazione con paginazione

# File System:

Talvolta applicazioni e processi necessitano di grandi quantità di dati, si vuole che questi siano consistenti e duraturi ed accessibili solo da processi voluti. I requisiti fondamentali risultano quindi essere 3:

- Deve essere possibile salvare una grande quantità di informazioni
- L'informazione deve sopravvivere alla terminazione del processo
- Processi multipli devono poter accedere all'informazione contemporaneamente.

I **files** sono unità logiche di informazione create dai processi e l'informazione in essi contenuta deve essere persistente. Come i files con cui il sistema operativo interagisce siano disegnati, salvati, implementati e gestiti fa parte del design del s.o.

Dal punto di vista del s.o. i dati sono il **file system**, che ha due scopi principali:

- Garantire un accesso permanente conveniente e consistente
- Garantire un uso efficiente delle risorse di memorizzazione

**File system:** un file system è l'insieme di algoritmi e strutture dati che realizzano la traduzione tra operazioni logiche sui file e le informazioni memorizzate sui dispositivi fisici (dischi, nastri...). Rappresenta una astrazione unificata dei dispositivi fisici effettivi.

Gli elementi di un FS si dividono in:

- LOGICI:
  - file
  - struttura di directory
- SW:
  - chiamate di sistema
  - routine di gestione
  - algoritmi scheduling
  - drivers

**File:** dal punto di vista dell'utente un file è un insieme di dati correlati e associato ad un nome. Dal punto di vista del s.o. un file è un insieme di byte eventualmente strutturato.

il *nome* è una sequenza limitata di caratteri (le citi dipendentemente dall'implementazione del s.o.); e.g. nel caso dei sistemi MS-DOS ad esempio sono nomi di 8+3 caratteri, case insensitive con estensione usata dal sistema per il trattamento dei file; problemi che questo comporta:

- più programmi possono usare la stessa estensione
- lo stesso file può essere elaborato da più programmi
- l'estensione può essere gestita in modo non corretto dagli utenti

Nel caso di UNIX invece i nomi sono lunghi e solo "/" non è utilizzabile. Le estensioni non sono gestite. In Mac OS i nomi sono lunghi senza estensione, ma associa ad ogni file un eseguibile.

## Struttura di un file:

Si distinguono tre tipi differenti di struttura di un file:

- **Sequenza di byte:** la struttura interna del file è gestita dai programmi applicativi



Figura 100: Sequenza di byte, nessun ordinamento

- **Sequenza di record:** di dimensione fissa, le operazioni di lettura restituiscono un record, le operazioni di scrittura sovrascrivono o appendono un record

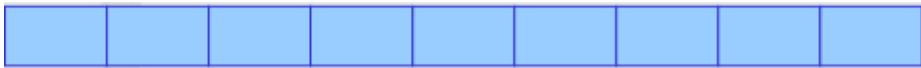


Figura 101: Struttura a record

- **Albero di record:** di lunghezza differente caratterizzati da una *chiave* in base alla quale si ordina l'albero. L'operazione base non è ottenere il record successivo ma un record particolare individuato tramite chiave. Con una struttura ad albero riesco ad accedere alle informazioni che mi servono in un tempo logaritmico invece che lineare.

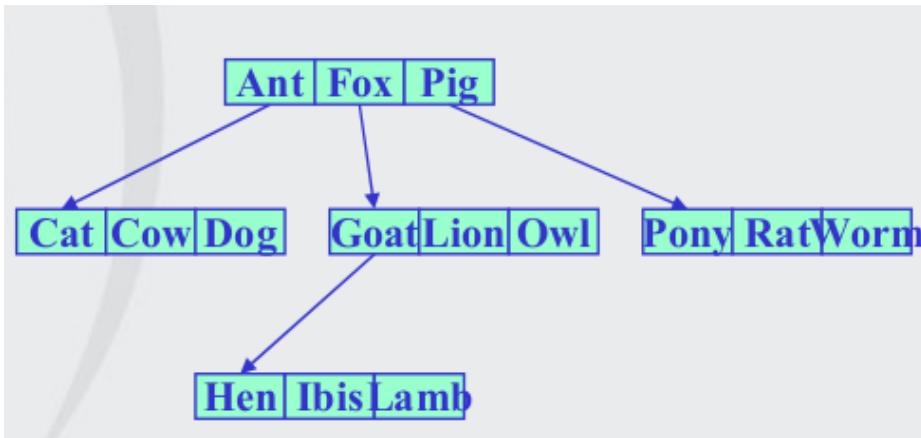


Figura 102: Struttura ad albero

Ogni file ha diversi attributi (definiti anche **metadata**), che possono indicare informazioni al suo riguardo e definirne permessi e utilizzi, ad esempio Nome, Tipo, Creazione, Utente\_Creator, se sono read-only, la lunghezza, la dimensione, l'ultima modifica, l'ultimo accesso...

```
-rw-r--r-- 1 luca 2531 Apr 25 10:01 000-home.htm
drwxr-xr-x 2 luca 512 May 12 03:09 File-System
```

**Operazioni sui file:** sistemi differenti consentono differenti operazioni sui file:

- **Create:** permette la creazione del file senza dati ma permette di settare gli attributi
- **Delete:** eliminazione del file quando non più necessario.
- **Open:** apertura di un file da parte di un processo, necessaria prima del suo utilizzo
- **Close:** quando un processo non necessita più di un file è necessario che venga chiuso.
- **Write:** operazione di scrittura su file alla posizione corrente; se la posizione è la fine del file il file viene incrementato di dimensione.
- **Append:** forma restrittiva di write che permette di scrivere sempre in coda al file
- **Seek:** per i file ad accesso random questo metodo specifica da dove iniziare a prendere i dati.
- **Get Attributes:** metodo per leggere gli attributi del file (e.g. chiamato dalla `make` di Unix per ottimizzare i cicli di compilazione quando deve unire più sorgenti)
- **Rename:** metodo per la rinomina di un file.
- **Set attribute:** talvolta alcuni attributi sono modificabili dall'utente

**Tipi di file:** esistono almeno due tipi di file: **file ordinari** e la **directory**. Queste ultime permettono di rappresentare la struttura logica interna di un file system; non sono altro che file particolari che contengono una lista di file o altre directory. Alcune caratteristiche sono dipendenti dal sistema operativo (e.g. Unix consente l'esistenza dei *file speciali*, in cui posso andare a leggere o scrivere file come sequenza di caratteri (e.g. mouse e tastiera vengono di fatto letti come file di caratteri))

Esistono anche *file speciali a blocchi* utilizzati per trattare i dati salvati su disco, in cui la lettura/scrittura è fatta attraverso blocchi e non caratteri (non posso scrivere un singolo carattere ma devo leggere/scrivere un blocco alla volta). altre tipologie di file sono i *file binari*, che sono quei file che hanno una struttura interna gestita dai programmi, in contrapposizione ai *file ASCII*.

Tipo di file	Usuale estensione
Eseguibile	exe, com, bin, nessuna
Oggetto	obj, o
Codice sorgente	c, cc, java, f, asm
Batch	bat, sh
Testo	txt
Elaboratore di testi	tex, doc, rtf
Libreria	lib, a, so, dll
Stampa o visualizzazione	ps, pdf, dvi
Archivio	zip, tar, tgz
Immagini, audio, multimedia	gif, tiff, jpg, au, wav, mpeg, mov

Figura 103: Riassunto dei tipi di file

### Accesso ai file:

In origine l'accesso ai file era soltanto **sequenziale**, in cui un processo poteva leggere tutti i byte di un file in ordine iniziando dall'inizio, senza potere saltare e leggerli in ordine sparso. Con l'avvento dei dischi venne introdotta la lettura ad **accesso casuale**. Essi sono essenziali per diverse applicazioni (e.g. database). Per la lettura ad accesso casuale si può utilizzare la read specificando la posizione oppure utilizzare la seek per trovare una posizione e poi leggere sequenzialmente da quella posizione.

Un altro tipo di accesso è l'**accesso indicizzato**, in cui da un file indice si risale alla posizione dei dati relativi nel file relativo:

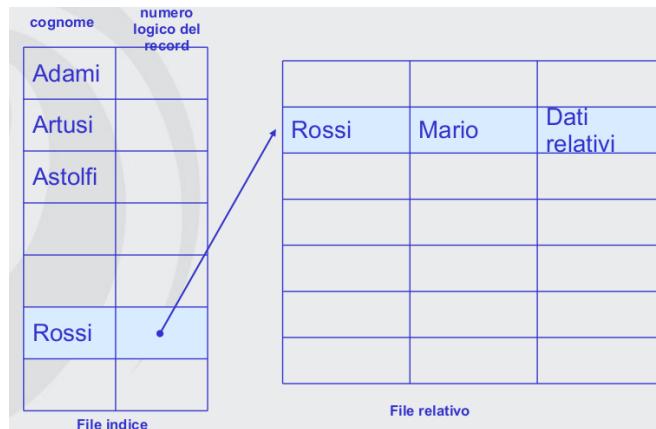


Figura 104: Accesso indicizzato ad un livello

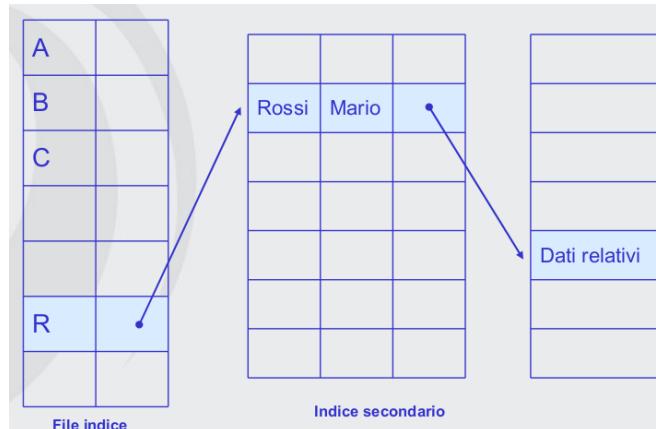


Figura 105: Struttura ad albero a due livelli

## Directory:

Una directory è spesso un file che contiene altri file e/o altre directory e solitamente sono organizzate secondo due strutture principali:

- Ogni voce contiene il nome e gli attributi dei file

<b>Ant</b>	<b>attributi</b>
<b>Fox</b>	<b>attributi</b>
<b>Pig</b>	<b>attributi</b>
<b>Worm</b>	<b>attributi</b>

Figura 106: Directory con nome e attributi

- Ogni voce contiene il nome e un puntatore ad una struttura separata che contiene gli attributi ai file

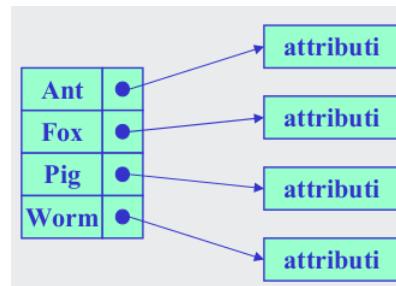


Figura 107: Directory con puntatori

Esistono 3 organizzazioni principali per le directory: Una directory unica, una directory per ogni utente o, la più complessa, un albero di directory arbitrario.

**Directory unica:** particolarmente diffusa nei primi pc basati su floppy disk con memoria ridotta (in cui una struttura ad albero non avrebbe avuto molto senso) e l'organizzazione logica era tale per cui su ogni floppy erano associate le informazioni per un unico programma.

Impensabile in sistemi multiutente a meno di implementazioni estremamente macchinose sulle regole di sequenza di caratteri nei nomi (per introdurre una fittizia organizzazione logica dei processi multiutente)

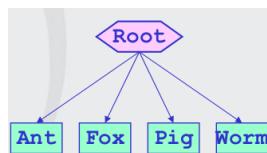


Figura 108: Struttura di una directory unica

**File system a due livelli:** ho una directory root con al di sotto una serie di sottodirectory, spesso associata ognuna ad un particolare programma/utente;

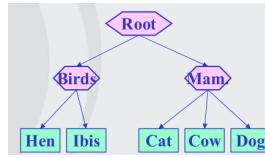


Figura 109: Struttura a due livelli

**Albero:** è il modo più comune di gestione dei file system, in cui ogni directory può contenere un numero non precisato di file o directory; organizzazione flessibile che permette di gestire tutte le situazioni dal punto di vista dei sistemi multiutente. In Unix ho un unico albero che parte da /, in Windows ho una popolazione di alberi (ogni lettera ha un suo albero)

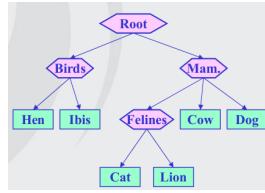


Figura 110: Struttura ad albero di un file system

## Operazioni directory:

anche sulle directory è possibile eseguire diverse operazioni, poichè infatti per accedere ad una directory è necessario rifarsi alle operazioni di sistema, non è a completa libertà dell'utente:

- **Create:** permette la creazione di una dir vuota, è operazione non banale poichè creo dir che contiene già . e ..
- **Delete:** permette di cancellare una directory VUOTA
- **OpenDir:** apre una directory per la consultazione
- **CloseDir:** chiude una dir aperta
- **ReadDir:** restituisce la voce successiva all'interno della directory
- **Rename:** permette di rinominare la directory
- **Link:** aggiunge una voce alla directory (legame alla struttura)
- **Unlink:** elimina una voce

**Link:** esistono anche in ambiente Windows, sono scorciatoie per accedere a file o directory usati frequentemente, condivisi o usati attraverso nomi differenti. Permettono inoltre di avere più di un punto di accesso per lo stesso file/directory benché le informazioni rimangano in un'unica copia. Così facendo memorizzo una particolare informazione in più directory differenti. **N.B.** link != copia, nel link tutte le modifiche ad una directory linkata modifica anche il contenuto in tutte le altre directory, con le copie modificherei solo la copia.

In **Windows** viene creato un file che contiene un certo numero di informazioni tra cui il nome del file a cui si riferisce. Sostanzialmente creo una nuova modalità di accesso alle informazioni contenute in un file/directory. L'utente non lo vede ma il nuovo file ha estensione .lnk e il suo contenuto è il contenuto a file originale da cui è partito il link e quando viene aperto con un qualsiasi programma questi risale il link fino a giungere al file originale da aprire (viene letto prima il contenuto del link). In Windows è

possibile solo un livello di collegamento.

In **Unix** sono possibili più livelli di collegamento, con il file system che non è più un albero ma un grafo che può contenere dei cicli.

Per evitare problemi viene imposto un numero massimo di link attraversabili prima di raggiungere l'informazione.

In unix esistono due tipologie di link:

- **SOFT LINK:** simile ai collegamenti di windows, il file creato contiene solo il nome di riferimento; crea accesso ad un vecchio file con un nuovo nome, creati con parametro -s. Creo così un file che contiene di fatto il nome del file originale attraverso cui accedere ai dati. Eseguibile per dir e file, linkabile su partizioni differenti e con differente inode id. Se la copia reale è eliminata il link NON funziona
- **HARD LINK:** eseguibile solo per i file e non eseguibile per partizioni differenti, ha inoltre lo stesso inode id dell'originale. Se la copia reale viene eliminata continua a funzionare, agendo come fosse il originale file (per eliminare un file devo eliminare tutti gli hard link)

### Allocazione file:

E' necessario avere un accesso veloce ai dati ed utilizzare in contemporanea il disco in modo efficiente; esistono 3 allocazioni: contigua, a liste e indicizzata;

**Contigua:** ogni file occupa un insieme contiguo di blocchi su disco allocati al momento della creazione. L'implementazione è molto semplice ed è sufficiente una tabella con le seguenti informazioni: Ha lo

Nome file	Blocco di partenza	Lunghezza
-----------	--------------------	-----------

Figura 111: Allocazione contigua

svantaggio che è necessario conoscere subito la dimensione del file e per l'allocazione si utilizzano algoritmi simili a quelli per la gestione della memoria (e.g. first fit, best fit etc). Le prestazioni sono ottime per la lettura, ma ho problemi di espansione dei file e della frammentazione della memoria con conseguente utilizzo non efficiente del disco. Poco utilizzata, se non per situazioni come dischi ottici o DVD, in cui si crea un FS che non sarà mai modificato e pertanto creo la struttura che devo gestire, sapendo la dimensione del file da scrivere ed ottimizzando le prestazioni di lettura successive senza problemi di frammentazione ne espansione dei file.

**Allocazione con liste:** evita problemi di espansione e allocazione dei file, ognuno di essi è gestito tramite una lista; da un punto di vista logico per ogni file ho un blocco e ogni blocco ha al suo interno un'informazione che è il puntatore al blocco successivo. Per aggiungere un blocco basta trovare un blocco libero ed aggiungere in coda alla lista il puntatore al file. Il problema principale è che l'accesso casuale non è vantaggioso, poiché di fatto per accedere all' n-esimo blocco devo leggere tutti i precedenti (n-1) blocchi; inoltre la dimensione logica dei blocchi non è più potenza di 2

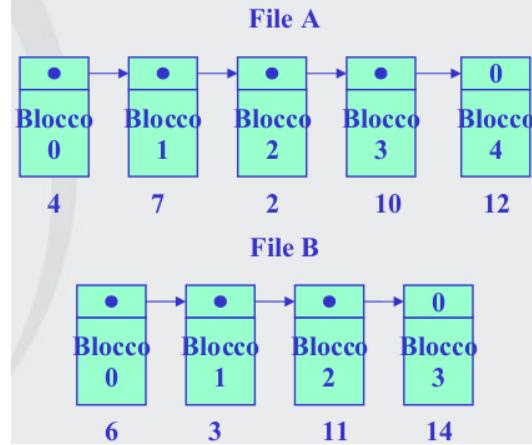


Figura 112: Schema a blocchi

**Allocazione indicizzata:** risolve i problemi precedenti poiché sposta tutti i puntatori dell'allocazione precedente in un solo blocco che è un blocco indice tenuto sempre in memoria; percorrendo la lista per raggiungere una certa posizione logica non devo percorrere anche tutti i blocchi; il problema è che il blocco indice può raggiungere dimensioni notevoli (e.g. se devo indicizzare tutto il disco diventa veramente di dimensioni notevoli)

**FAT:** MS-DOS utilizza l'allocazione indicizzata in cui ogni partizione ha una sua **FAT** (File Allocation Table) che viene caricata in memoria ogni volta che utilizzo quel determinato disco/partizione. In genere vengono memorizzate due versioni della FAT per poter recuperare informazioni in caso di errori. Ogni elemento della tabella è un puntatore che punta all'elemento logico immediatamente successivo dei blocchi; i blocchi non utilizzati hanno puntatore nullo.

La tabella ha una voce per ogni blocco contenente il numero del blocco successivo.

**I-Node:** il file system di Unix conserva gli attributi dei file separatamente dalle directory attraverso una struttura chiamata **i-node**. Ogni i-node ha anche i puntatori ai primi blocchi del file. Se non sono sufficienti, uno dei blocchi è utilizzato per contenere altri indirizzi di blocchi (**blocco a indirezione semplice**). Se nemmeno questo è sufficiente si utilizza un secondo livello (**blocco a indirezione doppia**). In casi estremi si può arrivare a **blocchi a indirezione tripla** in cui servono 3 accessi per giungere all'informazione che mi serve.

Ogni inode ha associato un numero univoco all'interno del dispositivo e ogni file presente è identificato come un collegamento fisico all'inode tramite il suo numero. Quando un programma cerca di accedere ad un file tramite un nome (es. documento.txt), il sistema operativo cerca l'inode corrispondente e recupera tutte le informazioni sopra descritte per operare correttamente con il file.

Un inode occupa un blocco, ma solo una piccola parte contiene gli indirizzi ai blocchi del file. Si usa un approccio multilivello: i primi 12 puntano direttamente ai blocchi, il tredicesimo punta ad un blocco riempito di riferimenti ai blocchi, un blocco indiretto singolo, il quattordicesimo ad un blocco indiretto doppio, il quindicesimo ad un blocco indiretto triplo. Così un file può avere una dimensione massima di  $(12 + 256 + 256^2 + 256^3) * \text{dimensione blocco}$ .

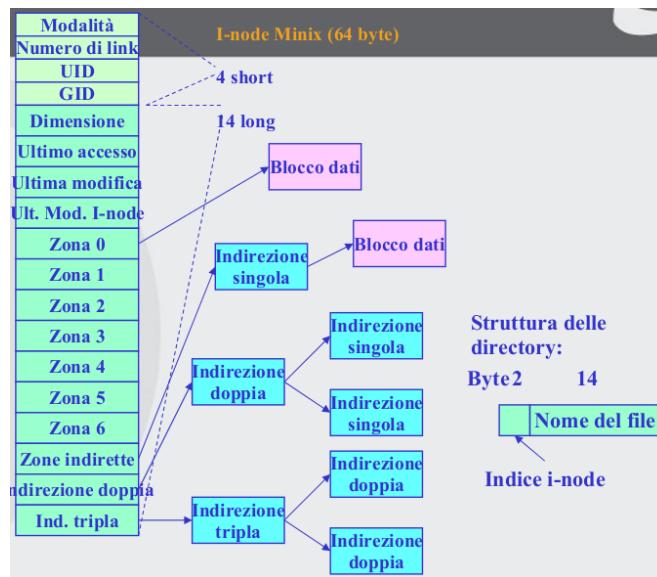


Figura 113: Struttura degli inode di minix

### Utilizzo di un file system:

In ambiente windows il sistema operativo riconosce la presenza di un dispositivo e mette a disposizione il contenuto attraverso il meccanismo della lettera (D:, C:, etc); in sistemi Unix è necessario montare manualmente i dispositivi attraverso il comando `mount <dispositivo> <dir>` t.c. la directory sia vuota. Solo l'amministratore può eseguire tale comando. Posso anche montare un file system su una macchina differente attraverso comando in rete, ad esempio: `mount -t nfs pippo.unipv.it:/users /users` o accedere ad un "finto dispositivo", montando un file come dispositivo (e.g. `mount -t iso9660 /tmp/cdrom-image /cdrom -o loop` che monta un file come se fosse un dispositivo)

In ambiente Windows è tutto più automatizzato con tutti i dispositivi che sono caricati automaticamente dal sistema e visti come *drive* (lettera:). I FileSystem remoti possono essere montati dall'utente e con l'introduzione di NTFS esiste la possibilità di montare un FS in una directory.

Attraverso mount posso montare un file system di Windows all'interno di un file system Unix.

Sotto Unix esiste il file `/etc/fstab` che permette di definire i file system utilizzati e le loro caratteristiche e il comando `umount <dispositivo> o <directory>` che permette di smontare un file system. Il file system è semplicemente come viene organizzata la memoria all'interno del s.o.

Come vengono utilizzati i filesystem dipende dal sistema operativo (Unix ha un'unica radice, Windows ha più radici a seconda del disco).

Ogni dispositivo contiene la cosiddetta descrizione del disco partizione in uso; tipicamente i dischi sono strutturati in più parti. Esiste solitamente un **disco di sistema** che contiene tutte le informazioni

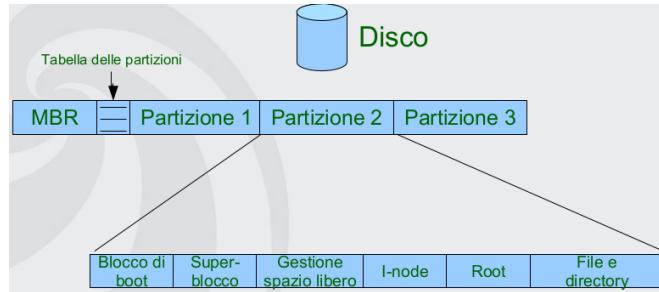


Figura 114: Struttura di un possibile fs di un disco

necessarie per avviare il sistema in se. Ogni disco poi ha un primo blocco detto **MBR** (Master Boot Record) che contiene informazioni su come utilizzare il disco, codice per avviare il pc se è disco di boot e informazioni per accedere alla tabella delle partizioni (che possono essere strutturati anche con partizioni differenti per utilizzi differenti). Il vantaggio di avere, come si vede nell'immagine, più gruppi di inode diminuisce la distanza tra l'informazione contenuta negli inode e i blocchi effettivi.

Il fatto di avere tanti file system differenti ha portato ad avere una gestione del filesystem virtuale: Non ho conoscenza effettiva della struttura del FS ma eseguo chiamate standard che vengono tradotte

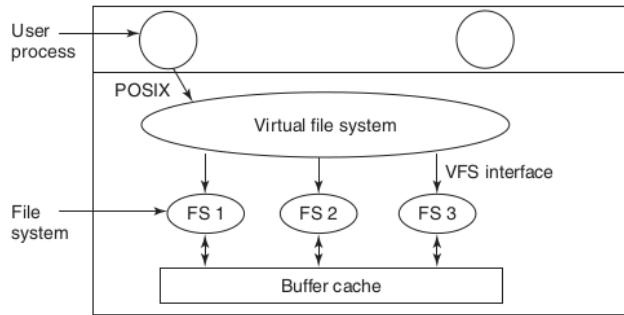


Figura 115: Struttura FS virtuale

da un'interfaccia con il FS in chiamate effettive per il FS (**VFS**, virtual file system). Il tutto viene interfacciato attraverso il **VFS Interface** (Virtual File System).

Nel momento in cui utilizzo funzioni di sistema in realtà sto usando un indirizzo all'interno della tabella

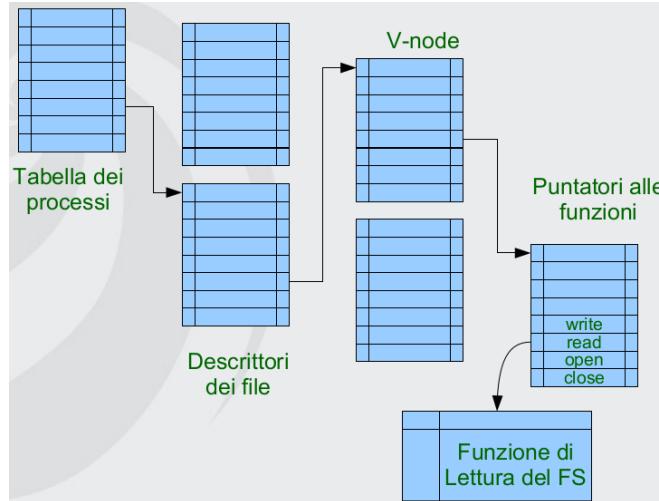


Figura 116: Tabella di azione all'interno di un VFS

che contiene tale operazione. Così facendo posso avere gestione specifica di ogni operazione senza che ad

alto livello occorra conoscere distinzione tra FS e VFS.

I blocchi devono essere dimensionati a dovere per evitare perdita significativa di spazio sul disco. Se ad esempio la dimensione media dei file è minore della dimensione del blocco si ha uno spreco di spazio che può essere notevole. La corretta dimensione dei blocchi migliora anche l'ottimizzazione del tempo di accesso ai dati, infatti supponendo 128KB per tracci e un T di rotazione di 8.33ms:

$$T\_Lettura\_Blocco = 10 + (B/128K + 0.5) * 8.33$$

$$Vel = Dim\_Blocco / Tempo\_Lettura$$

Soltamente si può considerare la dimensione media dei file ed utilizzare diverse partizioni, una con

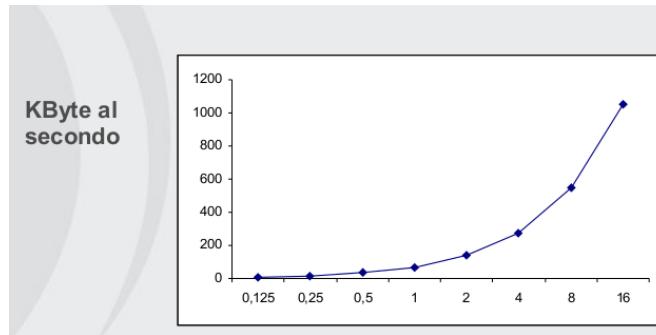


Figura 117: Grafico di andamento del tempo in relazione dei blocchi.

blocchi più grandi per file di grandi dimensioni, un'altra con blocchi più piccoli per file di dimensioni minori (e.g. sorgenti).

L'inconveniente maggiore nell'utilizzare soluzioni come i file compressi è che il danneggiamento di un solo byte può portare all'impossibilità di accesso all'intero file. Per gestire i *blocchi liberi* possono riservarsi alcuni blocchi per la gestione di una lista dei blocchi liberi o utilizzare una bitmap

**Affidabilità:** un filesystem deve essere protetto da danneggiamento, siano essi hw o sw. Generalmente i dischi contengono settori di riserva che sostituiscono settori che nel tempo si danneggiano (soluzioni a posteriori), potendo di fatto memorizzare più dati di quella che è la loro capacità nominale. Di fatto i settori danneggiati vengono segnati per non essere più utilizzati in modo completamente trasparente ad utente e s.o.

La soluzione più comune per l'integrità dei dati rimane il backup.

**Consistenza:** per i file system è importante anche mantenere consistenza nelle informazioni riguardo alla propria gestione:

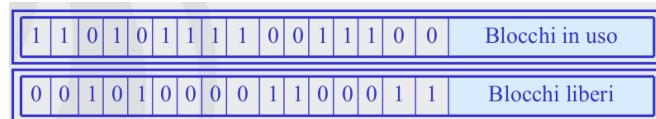


Figura 118: 1): Nessun errore

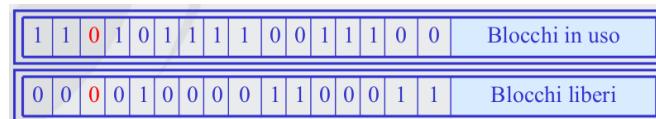


Figura 119: 2): Blocco mancante

Possono avvenire altre due situazioni: E' necessario controllare la consistenza della **struttura delle directory**, attraverso il confronto dei file esistenti con il contenuto delle directory

1   1   0   1   0   1   1   1   0   0   1   1   1   0   0	Blocchi in uso
0   0   2   0   1   0   0   0   0   1   1   0   0   0   1   1	Blocchi liberi

Figura 120: 3): Blocco libero duplicato (non possibile con le bitmap)

1   1   0   1   0   2   1   1   1   0   0   1   1   1   0   0	Blocchi in uso
0   0   1   0   1   0   0   0   0   1   1   0   0   0   1   1	Blocchi liberi

Figura 121: 4): Blocco utilizzato duplicato

I problemi di consistenza possono essere causati da crash di sistema per cui talvolta può capitare che alcuni blocchi non vengano riscritti e pertanto il filesystem divenga inconsistente.

**Prestazioni:** per migliorare le prestazioni parte dei blocchi su disco sono tenuti in buffer in memoria (**cache**) attraverso un'implementazione simile alla paginazione. L'aggiornamento di un blocco avviene periodicamente (UNIX) o ad ogni richiesta di scrittura (Windows); per rimuovere un disco su Windows è consigliabile la rimozione sicura, mentre su UNIX è **sempre** necessario smontare il dispositivo Blocchi inutilizzati in sequenza dovrebbero essere memorizzati vicino per minimizzare i tempi di posizionamento delle testine (defrag del disco, operazione che è tuttavia laboriosa)

# I/O:

Ha caratteristiche molto differenti a seconda dell'hw di cui si sta parlando, con velocità e tipi di dispositivi differenti. Idealmente il s.o. deve essere in grado di gestire tutte queste richieste. Si dividono due tipi di I/O

- A BLOCCHI: blocchi da 512B a 64K, con ogni blocco che può essere letto e scritto indipendentemente dagli altri ed è identificato da un indirizzo. I comandi comprendono rad, write e seek e l'accesso ai file viene fatto a basso livello o con un sistema di FS.
- A CARATTERI: non è indirizzabile e non ha alcuna primitiva di posizionamento (seek). I comandi comprendono get e put e l'editing di linea è possibile mediante librerie ad hoc.

E' fondamentale che il s.o. gestisca l'I/O indipendentemente dal dispositivo.

## Controllori:

tutti i dispositivi hanno un controllore, ovvero una componente elettronica per la gestione del dispositivo. Esempi di controllori sono IDE (Integrated Drive Electronics) o SCSI (Small Computer System Interface) e spesso i controllori sono integrati sulle schede madri. Generalmente ogni controllore utilizza una serie

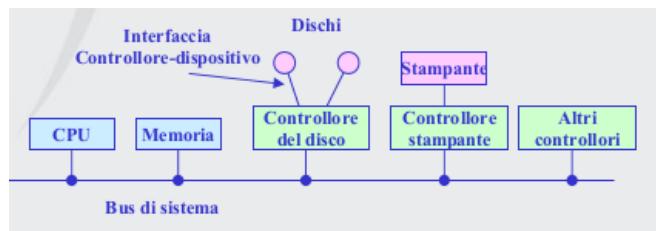


Figura 122: Bus di sistema e controllori

di registri che servono al trasferimento di informazioni dalla CPU al controllore e viceversa. In altri casi si utilizza uno spazio di indirizzamento separato in cui il s.o. effettua l'I/O scrivendo direttamente nel registro del controllore. Possono anche esistere approcci misti con parti direttamente nello spazio della memoria del sistema messe ad un proprio spazio di buffering.

## DMA:

svincola operazioni di I/O dall'utilizzo della CPU dando accesso diretto alla memoria (Direct Access Memory) e richiede un controller DMA che si trova concettualmente tra la CPU e la memoria ed è fisicamente collegato a memoria e CPU tramite BUS

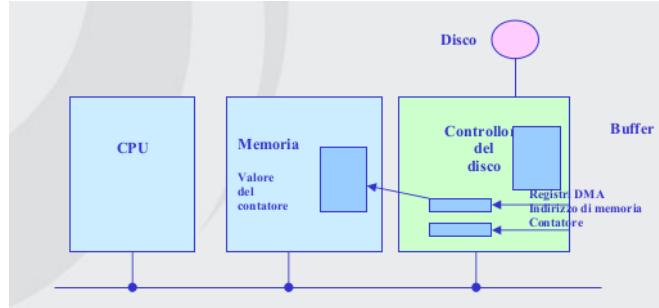


Figura 123: Struttura di un DMA access

## Interrupt:

è possibile anche una gestione tramite interrupt in cui al termine di ogni operazione di I/O corrisponde un segnale rilevato dal controllore di interrupt. Se non vi sono altre istruzioni in corso la richiesta viene gestita immediatamente, altrimenti viene ignorata

L'I/O può essere **programmato**, **guidato dalle interruzioni** o gestito tramite **DMA**

## Device Driver:

contiene tutto il sw dipendente dal dispositivo e impedisce comandi al controllore verificandone intanto il corretto funzionamento. Accetta richieste astratte indipendenti dal dispositivo e le traduce in istruzioni dipendenti. Scrive nei registri del controllore e può essere bloccato oppure no. Nel primo caso viene risvegliato da un interrupt; dopo il completamento dell'operazione di I/O il gestore deve controllare la correttezza dell'operazione.

Esiste sw di I/O che è indipendente dal dispositivo e che si interfaccia in modo uniforme con i device driver; vengono assegnati nomi ai dispositivi e le dimensioni del blocco è indipendente dal dispositivo. L'allocazione è dipendente dal dispositivo e informa su eventuali errori occorsi; spesso nella gestione dell'I/O si fa uso del cosiddetto buffering. Si può creare un buffer

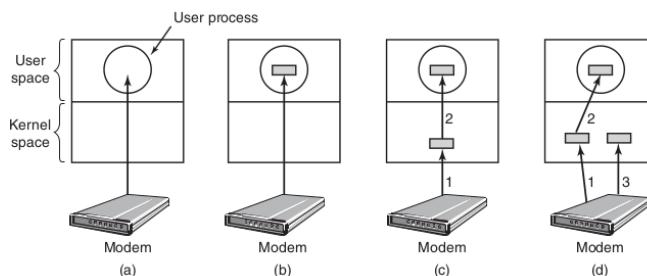


Figura 124: (a)- input senza buffer (b)-buffer in user space (c)-buffer in kernel seguito da una copia in user space (d)-buffer doppio in kernel

## Livelli del SW I/O:

esiste una pila decrescente ordinata che descrive i livelli dell'I/O a partire dall'hw giungendo fino ai processi utente:



Figura 125: Livelli di sw I/O

## Clock:

il clock serve per gestire il timesharing per la CPU e per tutta la gestione dei segnali e lo stesso sw per la gestione del clock è considerato device driver. E' gestita anche l'**ora di sistema** tramite un contatore che misura i tick del clock a partire dal 1-1-70 (generalmente si contano i secondi per avere un'estensione di 136 anni, a differenza del contare i tick effettivi che porterebbe ad overflow dopo 2 anni)

## Dischi magnetici:

è il dispositivo principale di memorizzazione dei dati ed è un tipo di memoria non volatile, con ottima capacità a basso prezzo; E' garantito l'accesso casuale ma con tempi di accesso abbastanza lunghi. Un

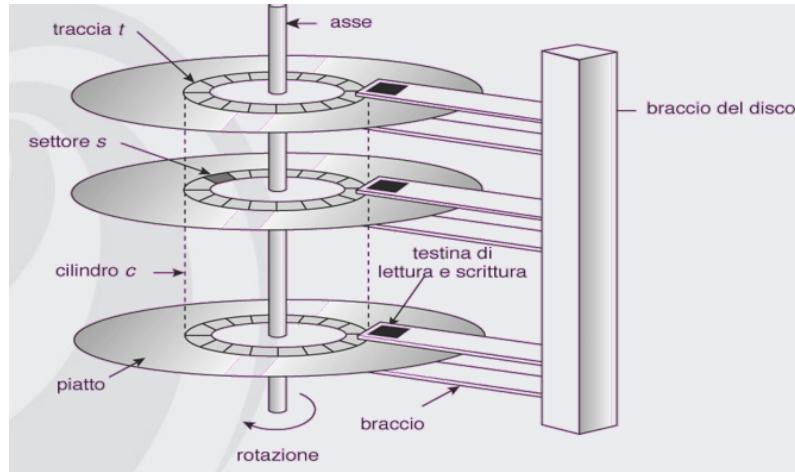


Figura 126: Struttura di un HDD

disco a stato solido, invece, ha tempi di accesso molto più rapidi benché tuttavia abbia un ciclo di vita nettamente inferiore. Non necessitano inoltre di deframmentazione (poiché se anche un file è disperso nel disco grazie ai tempi uniformi di accesso non è un problema) ed hanno una rumorosità bassa. Hanno inoltre un accesso casuale uniforme (accesso non dipende da componenti meccaniche, pertanto non conta come nel disco magnetico la posizione iniziale della testina)

**Struttura:** un disco ha diverse strutture logiche:

- tracce
- cilindri
- settori (si trovano all'interno di una traccia e sono di dimensione fissa o variabile)
- blocchi (unità di trasferimento tra disco e memoria, deve essere compatibile con le dimensione dei settori)

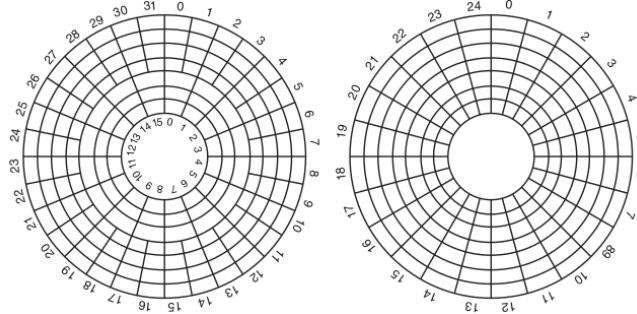


Figura 127: Geometria di un HDD

#### Tempi di accesso:

- **Tempo di seek:** è il tempo necessario per muovere la testina sul cilindro e dipende dalla posizione relativa della testina rispetto all'informazione da raggiungere (la traccia); è un tempo variabile tra i 2 e 10 ms; motivo per cui gli i-node vengono memorizzati vicino alle informazioni da memorizzare. Se gli i-node sono sulla traccia più vicina all'asse e i dati sono sulla traccia più lontana, ad esempio, avrò necessità di tempi molto lunghi.
- **Tempo di latenza:** è il tempo necessario affinchè il settore della traccia sia sotto la testina; dipende dalla velocità di rotazione e se parliamo di un hdd da 7200rpm ho 120 rps, con un totale di 8.3 ms al giro
- **Tempo di trasferimento:** a differenza dei precedenti è proporzionale all'ammontare dell'informazione trasferita e se ad esempio la traccia ha 100 blocchi significa che per ottenere l'informazione rispetto ad un blocco il disco deve ruotare per un centesimo rispetto al suo tempo di rotazione.
- **Tempo medio di accesso:** tempo medio di seek + tempo medio latenza + tempo medio trasferimento

Un controllore che supporta l'accesso DMA deve avere al suo interno i registri DMA, un buffer di memoria in cui scrivere temporaneamente. I controllori più semplici non riescono a gestire I/O simultanei poiché riescono a leggere solo un blocco ogni due, pertanto per leggere due settori consecutivi sono necessarie due rotazioni e per ovviare a questo problema si utilizza la tecnica dell' **interleaving**.

**Interleaving:** la numerazione logica dei settori è differente dalla mappatura fisica dei blocchi stessi. Così facendo saltando un settore si lascia il tempo per il trasferimento del dato precedente allo stesso

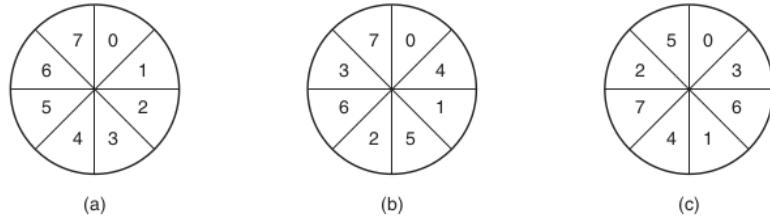


Figura 128: (a) nessun interleaving, (b) interleaving singolo, (c) interleaving doppio

modo si parla di *pendenza di cilindro* per cui la numerazione logica delle tracce è sfasata rispetto a quella reale

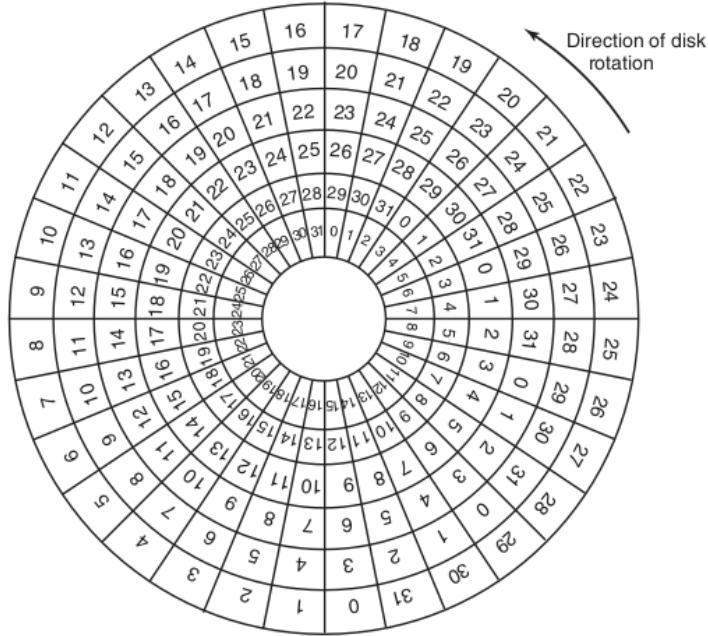


Figura 129: Pendenza di cilindro

Anche per il disco esistono algoritmi di scheduling; ad esso infatti arrivano richieste di lettura/scrittura generate dai processi e tali operazioni possono essere concorrenti o meno ed è necessario stabilire l'ordine di evasione delle richieste. Esistono diversi algoritmi possibili:

- **FCFS:** First Come First Served, le richieste sono evase a seconda del loro tempo di arrivo; è particolarmente semplice da implementare ed è modellizzabile da code FIFO. Non introduce starvation ma ha basse prestazioni per accessi frequenti (sufficiente per sistemi mono utente)
- **SSTF:** cerca la richiesta che è posizionata più vicino rispetto alla posizione corrente della testina. Solitamente si comporta meglio rispetto all'FCFS ma si può verificare la starvation pichè richieste lontane dal centro ottengono un pessimo servizio
- **SCAN:** una testina si muove da un'estremità all'altra del disco e se esiste una richiesta che può essere soddisfatta lungo uno spostamento da uno spostamento all'altro questa viene soddisfatta, altrimenti viene dimandata allo spostamento successivo. Esiste una variante detta LOOK che non sposta la testina fino alla fine ma solo se necessario per soddisfare le richieste pendenti, altrimenti inverte subito il movimento della testina. Il vantaggio maggiore è l'eliminazione dello starvation precedentemente osservato e ha performance elevate per carichi di grandi dimensioni
- **C-SCAN:** la scansione è sempre nella stessa direzione e i tempi di attesa sono più uniformi rispetto allo scan.

La valutazione degli algoritmi si basa su un modello di richieste, come distanza totale o tempo richiesto, ma tempi di latenza e di trasferimento vengono invece ignorati (perchè non possono essere sotto il controllo del s.o. e dipendono dall'hw e dall'ammontare dei dati trasferiti). Le condizioni iniziali da conoscere sono: posizione iniziale e direzione di spostamento.

**Come scelgo un algoritmo?** le prestazioni dipendono dal numero e tipo di richieste e la posizione dei blocchi indice è importante. Anche l'allocazione di file influenza le prestazioni. Inoltre alcuni algoritmi di basso livello sono implementati dal controller quindi sfuggono dal controllo del SO. La scelta degli algoritmi può anche dipendere dalla dimensione della cache del disco (e.g. mettere in cache una traccia intera) o dall'utilizzo di SSD o HDD

**Errori su disco:** devono essere gestiti anche i possibili errori su disco e settori danneggiati. Generalmente possono avvenire errori già in costruzione, oppure per errori di programmazione, di checksum (transitori o permanenti) e di posizionamento. Può essere utile mettere in cache una traccia intera, così facendo si ottimizzano le prestazioni riducendo i trasferimenti.

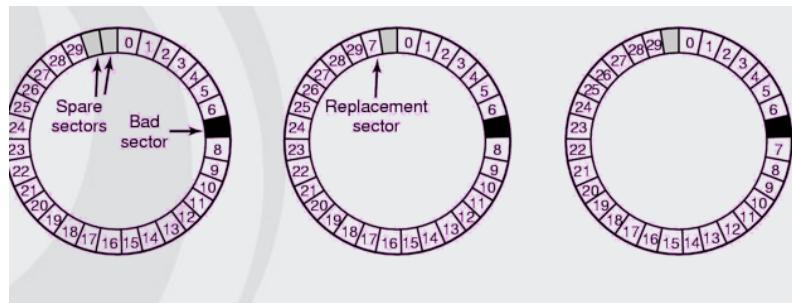


Figura 130: (a) un disco con settori danneggiati, (b) sostituzione di un settore danneggiato, (c) shift dei settori per bypassare il settore danneggiato

**RAID:** Redundant Array of Independent/Inexpensive Disk. Più dischi sono combinati in un unico disco logico dal sistema operativo. Ogni disco contiene una fetta (strip) dei dati per garantire la loro integrità.

- RAID0: è non ridondante e permette alte capacità di trasferimento dati se i trasferimenti riguardano grandi quantità di dati logici contigui; ha un basso tempo di risposta se il numero di richieste è per piccole quantità di dati

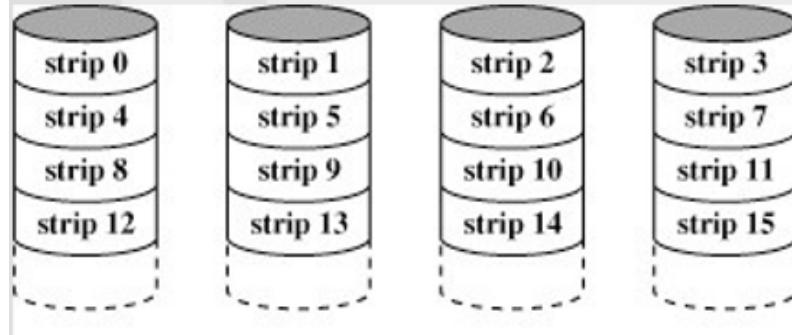


Figura 131: Schema del RAID0

- RAID1: è uno schema mirrored, per cui tutti i dati sono duplicati con la garanzia che se si dovesse rompere un disco le informazioni non andrebbero perse.

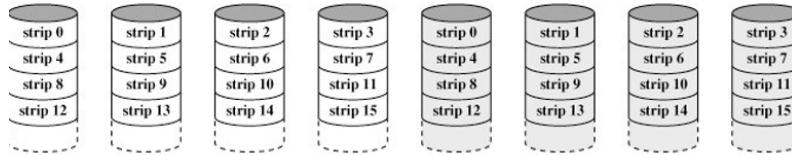


Figura 132: Schema di un raid1

- RAID2: memorizzazione ridondante tramite un codice di Hamming. Ogni disco memorizza bit particolare dell'informazione e ho una serie di dischi che permettono di recuperare l'informazione in caso di errore. I dischi sono sincronizzati tra loro e necessitano dell'aggiornamento ad ogni scrittura (buon trasferimento dati con scarsa frequenza di accesso)

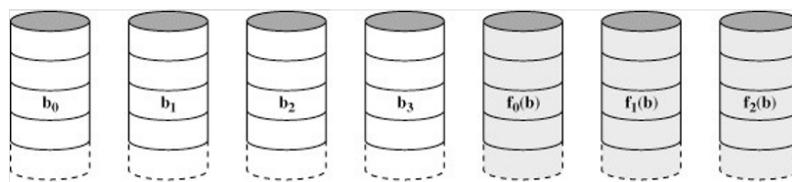


Figura 133: Schema di raid2

- RAID3: simile a RAID 2 ma con un solo bit di parità e con divisione in byte invece che in bit.
- RAID4: possiede una parità di blocco (come raid0 ma con aggiunta della parità) avendo un trasferimento dati non ottimo dovuto a collo di bottiglia sul disco di parità.

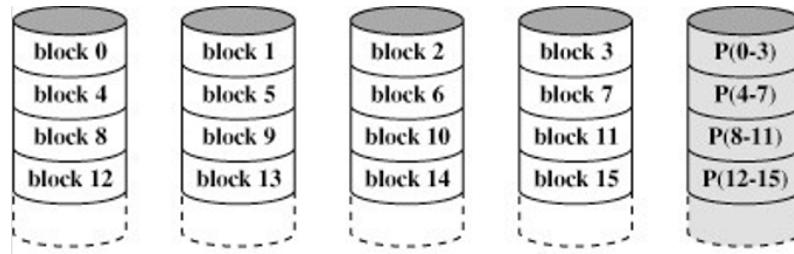


Figura 134: Schema di raid4

- RAID5: come raid 4 ma con la parità distribuita equamente sui dischi, eliminando il collo di bottiglia riguardo all'aggiornamento di questa.

E' possibile combinare a più livelli RAID0 e RAID1 (intercambiabili a seconda delle necessità primarie, se prestazioni o affidabilità)

# Sicurezza:

## Crittografia:

Le ipotesi fondamentali della crittoanalisi sono due, sostanzialmente:

- Eventuali attaccanti hanno una perfetta conoscenza dell'algoritmo utilizzato per cifrare il messaggio e di tutti i suoi dettagli
- Eventuali attaccanti hanno completo accesso al canale di comunicazione e possono pertanto intercettare o modificare il flusso di dati

Si vuole garantire il trittico base della sicurezza (CIA, Confidentiality, Integrity, Availability) più la *non ripudialità* (ovvero nessuno deve poter negare di essere autore di un messaggio, derivabile dall'integrità). E' fondamentale che l'algoritmo di cifratura/decifratura sia pubblico (non funziona l'idea della **security by obscurity**). Infatti secondo il principio di **Kerckhoffs** il "segreto" non deve risiedere nell'algoritmo ma soltanto nella **chiave**.

La sicurezza in ambito crittografico può seguire due direzioni:

- Sistemi *teoricamente* sicuri (non realizzabili fisicamente)
- Sistemi *computazionalmente* sicuri, che rendono inefficiente il tentativo di aggirare l'algoritmo (e.g. un tentativo di bruteforce di un algoritmo computazionalmente sicuro come Argon2). Idealmente il periodo temporale di mantenimento delle informazioni cifrate deve essere  $\leq$  del tempo ideale per violare l'algoritmo con lo stato dell'arte del calcolo

## Steganografia:

Un esempio di metodologia per nascondere il messaggio e trasmetterlo idealmente in modo sicuro può essere la steganografia, il cui scopo è nascondere l'esistenza del messaggio, ad esempio modificando a piacere il bit meno significativo di un'immagine nascondendo così alcuni MB all'interno di una sola immagine. Ovviamente la dimensione di quest'ultima crescerà notevolmente e ad un'analisi attenta il messaggio viene scoperto. Non è un sistema sicuro né conveniente, infatti richiede molti dati per nascondere pochi bit di informazione.

## Crittografia:

I sistemi crittografici sono generalmente classificati in base a 3 criteri:

- Il tipo di operazioni per passare da testo in chiaro a testo cifrato (ad esempio se si tratta di algoritmi unidirezionali come le funzioni di hashing o funzioni bidirezionali come può essere il Rijndael, di cui AES è un'implementazione)
- Il numero di chiavi usate e la loro condivisione/generazione (ovvero se si tratta di algoritmi a chiave privata o chiave pubblica, simmetrica o asimmetrica)
- Il modo in cui viene elaborato l'algoritmo (sw, hw, real-time...)

Definendo **P** il file plaintext,  $K_E$  la chiave di crittazione, **C** il testo cifrato e **E** l'algoritmo allora:

$$C = E(P, K_E)$$

Ovvero il testo cifrato è ottenuto usando l'algoritmo di crittazione noto, E, con il plaintext, P, e la chiave segreta  $K_E$  come parametri. Similmente

$$P = D(C, K_D)$$

E' il testo in chiaro dove D è l'algoritmo di decrittazione,  $K_D$  è la chiave di decrittazione. Il plaintext quindi arriva dal testo cifrato C, dall'algoritmo  $K_D$  attraverso D.

Esistono due tipologie di crittografia basata su chiavi:

- Crittografia a chiave privata (o crittografia simmetrica)
- Crittografia a chiave pubblica (o crittografia asimmetrica)

**Crittografia simmetrica:** è un metodo di cifratura semplice in cui la chiave è unica per entrambe le procedure (cifratura e decifratura) rendendo l'algoritmo estremamente performante e semplice da implementare. Presuppone però che le due parti siano in possesso a priori della chiave di cifratura, e pertanto che in qualche modo se la siano comunicata attraverso un canale che non è garantito essere sicuro). Per ovviare a questo problema può avvenire uno scambio attraverso algoritmi a chiave pubblica della chiave privata, come suggerito ad esempio dallo scambio di chiavi di Diffie-Hellman (protocollo che consente a due entità di stabilire una chiave condivisa e segreta utilizzando un canale di comunicazione insicuro senza la necessità che esse si siano scambiate informazioni in precedenza). Dal punto di vista dell'algoritmo è esattamente quanto descritto sopra quando si parla di plaintext, testo cifrato etc, schematicamente riassumibile in

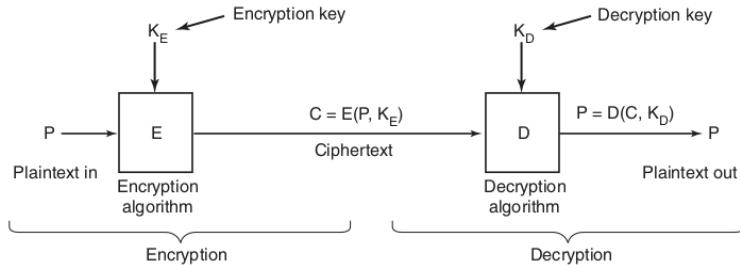


Figura 135: Schema di un funzionamento a chiave privata, inverso in caso di decifratura

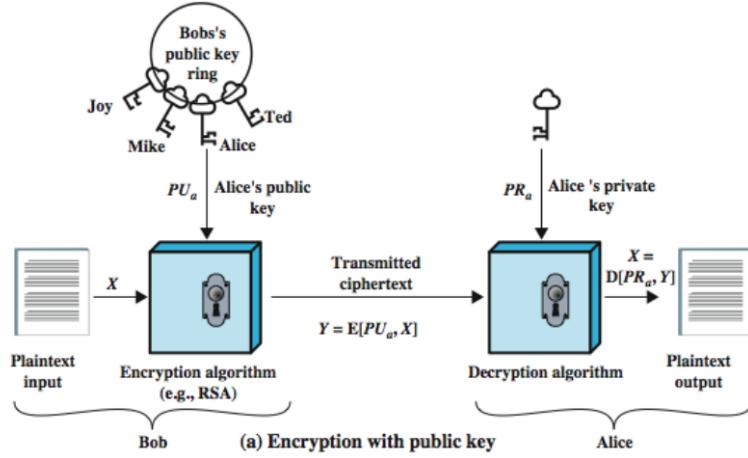


Figura 136: Schema della cifratura a chiave pubblica

**Crittografia asimmetrica:** In questa modalità di scambio di informazioni ogni utente ha una coppia di chiavi, una privata e una pubblica, matematicamente correlate ma t.c. sia impossibile dalla pubblica risalire alla privata (e viceversa). Quando due interlocutori vogliono scambiarsi un messaggio Alice cifra il messaggio attraverso la chiave pubblica di Bob, il quale alla ricezione del messaggio utilizza la sua chiave privata per decifrarlo. In pratica dal punto di vista della cifratura chiave privata e pubblica sono intecambiabili ma non ricostruibili a partire da una delle due. Uno dei sistemi a chiave pubblica è RSA. Il problema della chiave pubblica è che è più lento dell'algoritmo a chiave privata in quanto le chiavi devono essere sensibilmente più complicate.

**Firme dei messaggi:** la crittografia (e in particolare l'utilizzo della chiave pubblica) viene in aiuto anche per l'autenticazione dell'identità dei mittenti. Se infatti l'idea della chiave pubblica viene applicata anche dal mittente questo può garantire la sua identità: Alice quando vuole mandare un messaggio a Bob utilizza in primis la propria chiave privata per cifrare il messaggio, e successivamente la chiave pubblica di Bob. Così facendo solo Bob può leggere il messaggio e questi per farlo deve utilizzare la chiave pubblica di Alice (oltre alla propria privata) essendo così certo che il messaggio sia stato mandato da Alice (altrimenti la sua chiave pubblica non funzionerebbe).

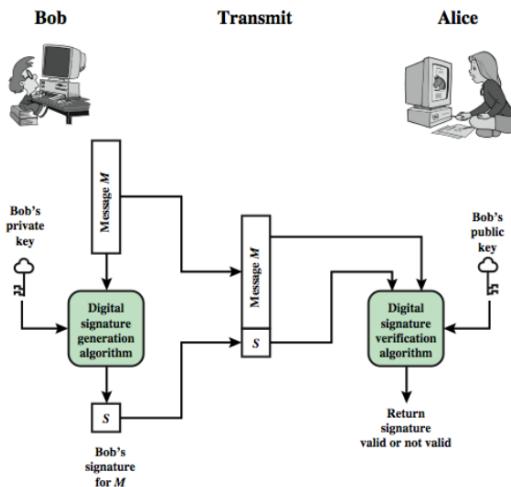


Figura 137: Schema della firma dei messaggi

**Funzioni di hash:** è una funzione non invertibile che mappa una stringa di lunghezza arbitraria in una stringa di lunghezza minore fissata. Deve avere le seguenti proprietà:

- Resistenza alla preimmagine → deve essere computazionalmente intrattabile la ricerca di una stringa in input che dia un hash uguale ad uno dato
- Resistenza alla seconda preimmagine → sia computazionalmente intrattabile la ricerca di una stringa in input che dia un hash uguale a quello di una data stringa
- Resistenza alle collisioni → sia computazionalmente intrattabile la ricerca di una coppia di stringhe in input che diano lo stesso hash

Le funzioni più note sono, ad esempio, MD5, SHA1 (deprecata), SHA2 (SHA224, SHA256, SHA384 etc etc)

**DES:** adottato nel 1977 come standard dal NIST utilizza una chiave simmetrica a 56 bit codificando blocchi di 64 bit, considerato oggi obsoleto. Sostituito dal **triple-DES**, che utilizza 3 chiavi e tre esecuzioni del DES standard seguendo un ordine cifratura-decifratura-cifratura

$$C = E_{K_3}(D_{K_2}(E_{K_1}(M)))$$

con una chiave di lunghezza 168 bit Si tratta a tutti gli effetti di un algoritmo a chiave pubblica in cui

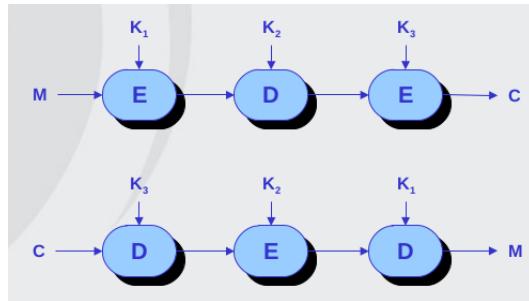


Figura 138: Schema del triplo DES

le chiavi sono due coppie (D,N) e (E,N) dove N è il prodotto di due numeri primi  $p$  e  $q$  con:  
 $ED \text{ mod } (p-1)(q-1) = 1$ ,  $C = m^E \text{ mod } N$ ,  $D = C^D \text{ mod } N = m = m^{ED} \text{ mod } N$ .

La conoscenza dell'algoritmo, di una delle chiavi e di esempi di testi cifrati non è sufficiente per determinare l'altra chiave.

## Sicurezza dei sistemi informatici:

problemi di sicurezza non riguardano solo la protezione dei dati da un utilizzo non corretto, ma anche problemi relativi all'hw. Devono essere garantite 3 proprietà fondamentali:

- **Confidentiality:** riguarda il fatto che un'informazione segreta debba rimanere segreta (se il proprietario dell'informazione decide che non deve essere accessibile a determinate persone deve essere garantita questa proprietà)
- **Integrity:** ovvero utenti non autorizzati non devono poter modificare l'informazione senza il permesso del proprietario della stessa
- **Aviability:** riguarda il fatto che nessuno possa disturbare il sistema e il suo funzionamento (e.g. DOS/DDOS)

**Minacce:** si ditingue in due tipologie di attacchi, gli attacchi **passivi** (accesso a informazioni riservate, analisi del traffico...) e **attivi** (ddos, mascheramento, mitm...)

Esistono diverse tipologie di minacce, tra cui ad esempio una minaccia di **interruzione di servizio**, in cui viene attaccata l'Aviability rendendo inutilizzabile un sistema. Un altro tipo di attacco, questa volta passivo, è l'**intercettazione**, in cui un attaccante ottiene informazioni senza intervenire o modificare le suddette andando a minacciare la Confidentiality. Esistono poi attacchi di **modifica** dei dati, in cui viene alterata l'Integrity (un esempio di attacco di questo tipo sono i ManIntheMiddle); possibile anche, ad esempio, generare informazioni ad hoc andando a modificare l'Autenticità della comunicazione (un esempio di attacco è l'attacco di phishing).

Le tipologie classiche di attacco prevedono l'intervento su dati non cancellati su pagine di memoria, spazi su disco o nastri, usare chiamate a sistema non previste o con parametri errati (e.g. *privilege escalation*), modificare strutture di sistema accessibili agli utenti, falsificare programmi di login...

**Malware:** a livello di attacchi con malware si distinguono due tipologie di programmi:

- **Dipendenti da programma ospite:** porzioni di programma che non possono esistere senza altri programmi o utility
- **Indipendenti:** programmi indipendenti che possono essere eseguiti dal sistema operativo

Alcune tipologie di malware, inoltre, hanno capacità di replicarsi come i Virus o i Worm.

**Trapdoor:** si tratta di punti di accesso nascosto al sistema create con buoni intenti per fare interventi e testing sul sistema che se dimenticate in fase di rilascio possono portare ad accessi indesiderati da parte di attaccanti. Talvolta invece è un attaccante a creare un accesso remoto al sistema attraverso programmi malevoli, creando le cosiddette **backdoor**

**Logic bomb:** porzione di codice che al raggiungimento di una particolare condizione esegue funzioni specifiche (e.g. eliminare contenuto del disco dopo una determinata data); talvolta vengono chiamate **time bombs**.

**Trojan horse:** programma che maschera funzioni malevole (o in generale funzioni secondarie) dietro a funzioni di un altro programma (e.g. software crackati che contengono software malevolo oltre al sw in se) ed è spesso nascosto in programmi innocui.

**Virus:** prendono il nome dal meccanismo con cui si propagano, emulando l'idea di virus dell'organismo umano. Sfrutta anch'esso altri sw di appoggio per nascondersi.

**Macro virus:** in genere programmi come Office permettono la definizione di macro (operazioni riconosciuta che permette di modificare codice o pagine in modo automatizzato); il problema è che le macro vengono memorizzate insieme all'informazione trattata, pertanto è possibile inviare un allegato che contenga, all'interno del codice macro, del codice malevolo eseguito all'apertura dell'allegato stesso in modo totalmente trasparente alla vittima.

**E-mail:** discorso simile può essere fatto per la posta elettronica, che spesso è vettore di trasferimenti malevoli (per rendere credibile il messaggio spesso un malware diffonde attraverso l'account dell'infetto ai suoi contatti)

*vedere worm Morris*

**Password:** è il metodo più classico per controllare gli accessi ad un sistema attraverso la coppia UID/-PWD. La parola crittografata viene salvata in un file. E' necessario non utilizzare informazioni comuni e ottenibili, non ripetere le password. Le password vengono memorizzate dopo essere criptate e vengono confrontati gli hash. Gli UID sono salvati in /etc/passwd su Unix, con le relative ppssword hashate salvate all'interno di /etc/shadow. Si aggiunge inoltre una componente di "salt" alla pwd per evitare che due pwd uguali abbiano lo stesso hash.