

# Sistemi Operativi

Capitoli:

1. Introduzione
  - a. Che cos'è un sistema operativo?
  - b. Storia dei sistemi operativi
  - c. Concetti di base dei SO
  - d. Le chiamate di sistema
  - e. Struttura di un sistema operativo
  - f. Introduzione al linguaggio C
2. Processi e thread
  - a. Processi
  - b. Thread
  - c. Comunicazione tra processi
  - d. Problema di comunicazione tra processi
  - e. Scheduling
3. Gestione della memoria
  - a. Sistemi di base per la gestione della memoria
  - b. Swapping
  - c. Memoria virtuale
  - d. Algoritmi di sostituzione delle pagine
  - e. Problematiche di progettazione dei sistemi di paging
  - f. Aspetti realizzativi
  - g. Segmentazione
4. File system

- a. File
  - b. Directory
  - c. Realizzazione del file system
5. Input Output
- a. Principi dell'hardware
  - b. Principi del software
  - c. Livelli software
  - d. Dischi
  - e. Clock

## 1. Introduzione

I calcolatori vengono dotati di software detto sistema operativo, il cui compito è gestire processori, dispositivi di io, e fornire ai programmi utente un'interfaccia semplificata con l'hardware. Un errore comune è considerare la GUI (Graphical User Interface) come parte del SO, in realtà è solo un'altra applicazione che lo utilizza.

Al livello inferiore abbiamo i hardware che in molti casi è composto da più livelli.

Poi abbiamo il livello di microarchitettura nel quale i dispositivi fisici sono raggruppati per fornire unità funzionali, contiene CPU, ALU, datapath, registri ecc..

Successivamente abbiamo il livello del linguaggio macchina (anche detto ISA) dove è presente l'insieme di hardware e istruzioni a disposizione del programmatore in linguaggio assembly.

La maggior parte dei computer ha due modalità operative: kernel (o supervisor) e user. L'interprete dei comandi (shell), i sistemi a finestre, i compilatori, gli editor e gli altri programmi indipendenti dalle applicazioni vengono eseguite in modalità user. Il sistema operativo è quella porzione di software che viene eseguito in modalità kernel, ed è protetto dalle manomissioni degli utenti dell'hardware. I compilatori ed editor vengono eseguiti in modalità utente.

Infine al di sopra dei programmi di sistema abbiamo le applicazioni: programmi che vengono acquistati o scritti dagli utenti.

### 1.a Che cos è un sistema operativo?

Il sistema operativo realizza due funzionalità:  
estendono la macchina, fornendo ai programmati ed alle applicazioni software astrazioni;  
gestiscono le risorse hardware.

## **Il sistema operativo come macchina estesa.**

L'architettura a livello linguaggio macchina è primitiva e risulta difficile da programmare.  
Al programmatore non interessa di come è composto un disco di memoria o che modulazione di frequenza viene usata per la scrittura dati, il sistema operativo nasconde tutte queste cose dell'hardware e presenta al programmatore una semplice interfaccia a file, l'astrazione fornita dal SO è più semplice e facile rispetto a quella fornita dall'hardware.

L'astrazione è un concetto chiave per gestire la complessità: in un sistema complesso si identificano delle caratteristiche salienti rispetto ad altre considerate trascurabili e lo si rappresenta attraverso un modello che contiene solo gli aspetti principali.

Per esempio il file è un concetto astratto che ogni utente conosce e utilizza: mentre l'utente "vede" il nome, la data, il tipo, il percorso astratto ove è memorizzato, il SO conosce tutti i dettagli che servono per la sua memorizzazione sul dispositivo I/O, come ad esempio la sua posizione fisica.

## **Il sistema operativo come gestore delle risorse**

Il SO ha come compito principale quello di mettere a disposizione dell'utente una comoda interfaccia. I calcolatori moderni hanno più processori, memorie ecc. il compito del SO è quello di gestire un'allocazione ordinata di risorse ai vari programmi che competono tra loro per usarle.

Tra i compiti del SO troviamo:

- gestire i conflitti derivanti dall'accesso alle risorse condivise (es. stampanti, memoria,...);
- garantire che ogni processo abbia accesso alle risorse in coerenza con le autorizzazioni di cui dispone;
- tenere traccia di quali processi stanno utilizzando quale risorsa.

La gestione delle risorse in termini di condivisione può avvenire sotto due aspetti:

- rispetto al tempo: la risorsa è utilizzata a turno dai processi;
- rispetto allo spazio: la risorsa è partizionata e più processi accedono concorrentemente alla risorsa (es. memoria principale).

## **1.b Storia dei sistemi operativi**

### **La prima generazione (1945-1955): valvole e schede a spinotti**

Erano macchine enormi, avevano migliaia di valvole, milioni di volte dell'attuale personal computer.

Tutta la programmazione veniva effettuata in linguaggio macchina, predisponendo una serie di spinotti su schede particolari che servivano per controllare le funzioni elementari della macchina. La quasi totalità delle applicazioni erano semplici calcoli numerici.

Negli anni '50 vennero introdotte le schede perforate, che rendevano possibile scrivere programmi sulle schede e leggerli tramite il calcolatore, invece di usare le schede a spinotti.

### **La seconda generazione (1955-1965): transistor e sistemi batch**

L'introduzione dei transistor durante la metà degli anni '50 cambiò radicalmente la situazione. Questi calcolatori, ora chiamati mainframe, erano molto costosi e solo grosse compagnie, agenzie governative o università potevano spendere milioni di dollari.

Per far girare un job (cioè un programma o un insieme di programmi), un programmatore doveva prima scrivere il programma su carta (in FORTRAN o in assembler), poi doveva copiarlo su schede perforate (successivamente sui nastri), infine portare il pacchetto da un operatore e aspettare i dati in uscita.

Successivamente, per velocizzare le operazioni, vennero inventati i sistemi batch (sistemi di elaborazione a lotti). L'idea era quella di trasferire i job su nastro magnetico usando un calcolatore poco costoso, ma non adatto ad eseguire i calcoli che venivano eseguiti su calcolatori molto più costosi.

## **La terza generazione (1965-1980): circuiti integrati e multiprogrammazione**

Nasce l'IBM System/360, che utilizza i circuiti integrati, ed il sistema operativo OS/360 che permette la multiprogrammazione (la zSeries è un suo discendente).

Questi sistemi possono leggere i programmi dal disco attraverso una tecnica denominata di spooling (poi utilizzata per l'output).

Nasce il primo sistema timesharing, CTSS (Compatible Time Sharing System) sviluppato al M.I.T. e, successivamente, il MULTICS. Da una versione derivata e ridotta, utilizzata su un minicomputer PDP-7, si gettano le basi per la progettazione del sistema UNIX.

## **La quarta generazione (dal 1980 a oggi): i Personal Computer**

Digital Research riscrive il CP/M (Control Program for Microcomputers) per utilizzare il processore Zilog Z80.

IBM progetta il primo PC con il DOS (Disk Operating System) e il linguaggio Basic.

Apple progetta Lisa (troppo costoso) e Macintosh, il primo ambiente user-friendly grazie all'implementazione della GUI.

Microsoft, influenzata dal successo di Macintosh, realizza Windows e Apple MacOS.

Iniziano ad essere utilizzate alcune versioni di UNIX sui PC: Linux, FreeBSD, ....

### **1.c Concetti di base sui SO**

I SO forniscono dei concetti di base e delle astrazioni: processi, file, protezione, spazio indirizzi, IO e Shell sono i principali concetti.

#### **I processi**

Il processo è l'esecuzione di un programma, ad ogni processo vengono associati:

- un suo spazio di indirizzamento dove sono memorizzati il programma eseguibile, i dati e lo stack;
- un insieme di risorse, tra cui registri (inclusi PC e SP), file aperti, elenco di processi correlati, allarmi in sospeso, ....;

- una riga all'interno della tabella dei processi.

I processi possono creare uno o più altri processi, detti processi figli e questi possono ripetere nuovamente l'operazione. I processi collegati possono cooperare per raggiungere un obiettivo comune ed hanno bisogno di sincronizzarsi attraverso la comunicazione. Questa comunicazione è detta InterProcess Communication (IPC).

Ai processi possono essere inviati dei segnali in analogia agli interrupt hardware. Ad ogni persona autorizzata all'uso di un sistema (utente) l'amministratore del sistema assegna un UID (User IDentification). Ogni processo che viene eseguito ha l'UID della persona che l'ha lanciato (un processo figlio ha lo stesso UID del processo che l'ha generato). Gli utenti possono essere membri di gruppi, identificato da un GID (Group Identification).

## **Lo spazio degli indirizzi**

Ogni computer ha una memoria principale per gestire l'esecuzione dei programmi.

Nei sistemi operativi più semplici c'è sempre solo un programma alla volta in memoria. In quelli più moderni possono risiedere in memoria allo stesso tempo più programmi, ma sono necessari meccanismi di protezione per evitare interferenze.

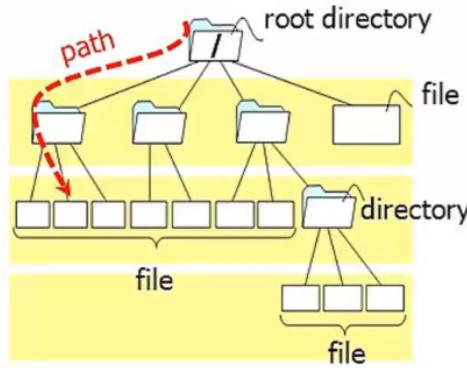
Nei primi sistemi operativi c'era il problema del processo a cui non bastava il suo spazio di indirizzamento. Ora, la virtualizzazione della memoria consente al processo di vedere uno spazio indirizzo unico anche se parte dei dati sono davvero in memoria principali ed altri sul disco (richiamandoli solo all'occorrenza).

## **File**

I file sono il modello astratto per raggruppare insiemi di dati nascondendo le peculiarità dell'hardware che li memorizza o utilizza (nastri, dischi, dispositivi di I/O,...).

Il file system è l'ambiente che gestisce i file ed è organizzato in modo gerarchico: ciascun nodo intermedio è un contenitore di altri file (directory) o altri contenitori, in modo ricorsivo.

L'elenco delle directory che occorre attraversare dalla radice (root) per raggiungere il file è detto percorso o path del file (il simbolo "/" o "\" è utilizzato come separatore dei nomi).



Prima che un file possa essere letto/scritto è necessario aprirlo e in questo momento sono controllati i permessi, se l'accesso è consentito è restituito un numero intero detto descrittore del file; se l'accesso è proibito, viene restituito un codice di errore.

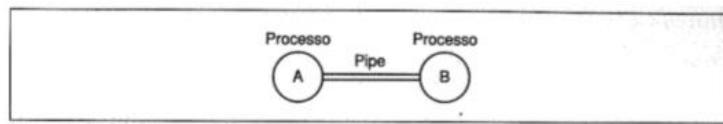
In UNIX è importante notare il concetto di file system montato: cioè il file system del dispositivo di I/O viene attaccato all'albero principale.

In UNIX i file speciali permettono di considerare i dispositivi di I/O come file e sono di due tipi:

- a blocchi: sono utilizzati per modellare dispositivi in grado di indirizzare casualmente blocchi di dati (es. dischi);
- a caratteri: sono utilizzati per modellare dispositivi che utilizzano le sequenze (o stream) di caratteri (es. stampanti, tastiere, ...).

Si può utilizzare uno pseudofile, chiamato pipe, per far comunicare due processi. Se i processi A e B vogliono comunicare utilizzando una pipe devono:

- inizializzarla in anticipo;
- il processo A, che vuole spedire dati al processo B, scrive sulla pipe come se fosse un file in uscita;
- il processo B può leggere i dati leggendo dalla pipe, come se fosse un file d'ingresso.



**Figura 1.16** Due processi connessi da una pipe.

## **Input/output**

Tutti i computer hanno dispositivi fisici per acquisire input e produrre output.

Esistono molti tipi di input e di output, incluse tastiere, monitor, stampanti e così via. Sta al sistema operativo gestire questi dispositivi. Di conseguenza, ogni sistema operativo ha un sottosistema di I/O per gestire i suoi dispositivi di I/O. Parte del software di I/O è indipendente dai dispositivi, ossia si applica a molti o a tutti i dispositivi allo stesso modo. Altre parti, come i driver dei dispositivi, sono specifici per particolari dispositivi di I/O.

## **Protezione**

La protezione dei dati è una caratteristica importante dei SO.

Considerando come esempio UNIX, i file sono protetti mediante l'assegnazione di un codice di protezione a 9 bit, costituito da tre campi di 3 bit, uno per il proprietario del file, uno per gli altri membri del medesimo gruppo del proprietario e uno per tutti gli altri; ogni campo ha un bit per l'accesso in lettura, un bit per l'accesso in scrittura, un bit per l'accesso in esecuzione (bit rwx).

Esempio:

owner other everyone

r w x    r - x    - - x

## **Shell**

Il SO è il codice che esegue le chiamate di sistema; editor, i compilatori, gli assemblatori, i linker e l'interprete dei comandi (la shell in UNIX) non fanno parte del SO.

La shell (interprete dei comandi) è l'interfaccia principale tra utente e SO.

La GUI (Graphic User Interface) è un'applicazione che sta sopra al SO proprio come l'interprete dei comandi, ma non fa parte del SO.

## **Deadlock**

Anche se non è stato menzionato è importante sapere in che cosa consiste: Quando due processi interagiscono possono mettersi in una situazione di stallo (deadlock). Basta pensare a 4 autobus che arrivano contemporaneamente ad un incrocio e si bloccano perché non possono passare.

## **1.d Le chiamate di sistema**

Qualsiasi computer monoprocessoress può eseguire un'istruzione alla volta, se un processo utente ha bisogno di leggere un file deve chiamare un servizio di sistema, che restituirà poi il controllo al chiamante una volta eseguito il servizio.

Una **chiamata di sistema** o system call è una procedura speciale che viene eseguita in modalità kernel, ad esempio:

```
count = read(file, buffer, nbyte);
```

La chiamata di sistema read() restituisce il numero di byte realmente letti nel buffer: non è detto che si riesca a leggere nbytes. Se la chiamata non può essere completata, a causa di un parametro non corretto o di un errore sul disco, count viene messo uguale a -1.

#### Gestione dei processi

Chiamata	Descrizione
pid = fork()	Crea un processo figlio identico al padre
pid = waitpid(pid, &stalloc, options)	Attende che un figlio termini
s = execve(name, argv, environp)	Sostituisce l'immagine core di un processo
exit(status)	Termina l'esecuzione del processo e restituisce lo stato

#### Gestione dei file

Chiamata	Descrizione
fd = open(file, how, ...)	Apre un file in lettura, scrittura o entrambi
s = close(fd)	Chiude un file aperto
N = read(fd, buffer, nbytes)	Legge dati da un file in un buffer
n = write(fd, buffer, nbytes)	Scrive i dati da un buffer in un file
position = lseek(fd, offset, whence)	Sposta il puntatore del file
s = stat(name, &buf)	Ottiene informazioni sullo stato del file

#### Gestione delle directory e del file system

Chiamata	Descrizione
s = mkdir(name, mode)	Crea una nuova directory
s = rmdir(name)	Rimuove una directory vuota
s = link(name1, name2)	Crea un nuovo elemento, name2, che punta a name1
s = unlink(name)	Rimuove un elemento dalla directory
s = mount(special, name, flag)	Monta un file system
s = umount(special)	Smonta un file system

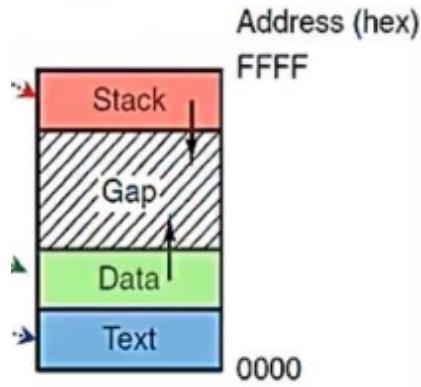
#### Varie

Chiamata	Descrizione
S = chdir(dirname)	Cambia la directory di lavoro
S = chmod(name, mode)	Cambia i bit di protezione di un file
S = kill(pid, signal)	Manda un segnale al processo
Seconds = time(&seconds)	Calcola il tempo trascorso a partire dal 1 Gennaio 1970

## Chiamate di sistema per la gestione di processi

In POSIX (lo standard per UNIX) la chiamata fork() permette di creare una replica del processo (chiamato processo figlio), a questo punto i due processi avranno vite separate

La memoria dei processi in UNIX è suddivisa in tre segmenti: il segmento testo (cioè il codice programma), il segmento dati (cioè le variabili) e il segmento stack (utilizzato per gestire le attivazioni delle procedure). Il segmento dati cresce verso l'alto e lo stack cresce verso il basso. In mezzo a loro c'è un intervallo di spazio inutilizzato.



## Chiamate di sistema per la gestione dei file

Molte chiamate di sistema fanno riferimento al file system. Prima di poter leggere o scrivere un file, è necessario aprirlo con la chiamata di sistema open, read e write servono per leggere e scrivere il file, con close è possibile chiudere il file.

## Chiamate di sistema per la gestione delle directory

Le chiamate mkdir() e rmdir() rispettivamente creano directory o rimuovono directory vuote.

La chiamata link() permette di creare un riferimento ad un file o una directory.

La chiamata mount() permette di fondere insieme due file system, è utilizzata quando si vuole montare il file system di un dispositivo (es. CD-ROM) all'interno del file system del sistema:

`mount("/dev/fd0", "/mnt", 0);`

il primo parametro è il nome di un file speciale a blocchi, il secondo specifica dove montarlo e il terzo la modalità (lettura o scrittura).

## Le API Win32 di Windows

Windows e UNIX differiscono per il modello di programmazione: un programma UNIX richiede dei servizi attraverso le chiamate di sistema mentre un programma Windows è guidato dagli eventi. Anche Windows ha le chiamate di sistema.

Microsoft ha definito un insieme di procedure chiamate le API Win32 (Application Program Interface) che i programmatore possono utilizzare per ottenere i servizi del SO.

## Confronto tra system call

UNIX	Win32	Descrizione
fork	CreateProcess	Crea un nuovo processo
waitpid	WaitForSingleObject	Aspetta la terminazione di un processo
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Termina esecuzione
open	CreateFile	Crea un file o apre un file esistente
close	CloseHandle	Chiude il file
read	ReadFile	Legge dati da un file
write	WriteFile	Scrivi dati in un file
lseek	SetFilePointer	Sposta il puntatore nel file
stat	GetFileAttributesEx	Recupera gli attributi del file
mkdir	Create Directory C	Crea una nuova directory
rmdir	Remove Directory	Rimuove una directory vuota
link	(none)	Win32 non supporta i collegamenti
unlink	DeleteFile	Cancella un file esistente
mount	(none)	In Win32 non esiste
umount	(none)	In Win32 non esiste
chdir	SetCurrentDirectory	Cambia la directory corrente di lavoro
chmod	(none)	Win32 non supporta la protezione (NT sì)
kill	(none)	Win32 non supporta questi segnali
time	GetLocalTime	Restituisce l'orario corrente

## 1.e Struttura di un SO

Esistono sei differenti strutture: monolitici, a livelli, microkernel, client-server, virtuali macchine e exokernels.

### Sistemi monolitici

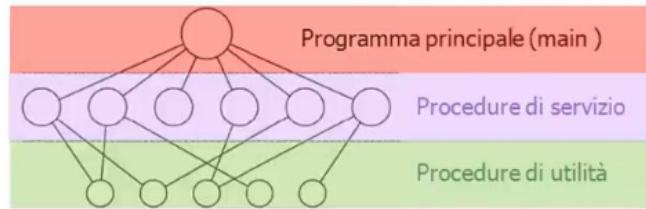
I sistemi monolitici sono l'organizzazione più comune; Il SO viene eseguito come un unico programma in modalità kernel.

Il SO è un unico grande programma binario eseguibile che contiene all'interno un insieme di procedure collegate tra loro. Ogni procedura nel sistema è libera di richiamarne un'altra, in base ai propri compiti e del risultato che possono offrire. La presenza di migliaia di procedure che si possono chiamare reciprocamente senza limiti rende il sistema di difficile comprensione.



Questa organizzazione suggerisce la seguente struttura base per il SO:

- un programma principale che richiama la procedura di servizio richiesta;
- un insieme di procedure di servizio che realizzano le chiamate di sistema;
- un insieme di procedure di supporto che aiutano le procedure di servizio.



## Sistemi a livelli

Una generalizzazione del modello strutturale dei sistemi monolitici è il SO a livelli, ognuno costruito sopra quello sottostante.

Il primo sistema di questo tipo fu il sistema THE (Technische Hogeschool Eindhoven) sviluppato da Dijkstra nel 1968, ma era ancora un unico eseguibile.

Livello	Funzione
5	L'operatore
4	Programmi utente
3	Gestione dell'I/O
2	Comunicazione operatore-processo
1	Gestione della memoria
0	Allocazione del processore

Un altro sistema era il MULTICS ed era descritto come una serie di anelli concentrici, dove quelli più interni erano più privilegiati di quelli esterni.

## Microkernel

Con l'approccio a strati, i progettisti potevano scegliere dove definire il confine tra kernel e utente.

Inizialmente tutti i livelli erano nel kernel. Ma, in realtà, bisogna collocare il meno possibile in modalità kernel, dato che errori lì possono far cadere immediatamente il sistema.

La difettosità di un codice dipende dalla dimensione del modulo, dalla sua età (più è vecchio più è stato testato e quindi più sicuro) e altri fattori. Statisticamente sono presenti 10 bug in 1000 righe di codice.

L'idea di base della struttura microkernel è di incrementare l'affidabilità suddividendo il sistema operativo in piccoli moduli collaudati e di avere un solo modulo eseguito in modalità kernel (il microkernel).

I SO a microkernel sono utilizzati in quelle applicazioni che richiedono alta affidabilità (come gli ambienti real-time, industriali, militari).

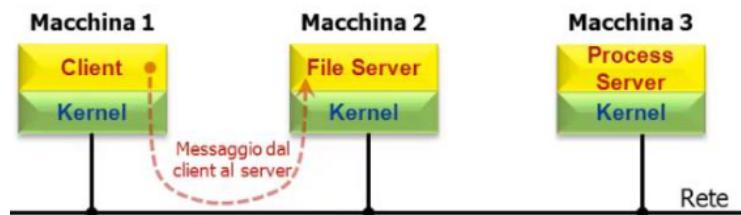
## Modello client-server

Una leggera variazione rispetto all'idea del microkernel è distinguere due classi di processi:

- i server, ognuno dei quali mette a disposizione alcuni servizi;
- i client, che li utilizzano.

Spesso il livello più basso è un microkernel, ma non è necessario.

La comunicazione tra client e server avviene spesso attraverso lo scambio di messaggi. Per ottenere un servizio, un processo client costruisce un messaggio indicando la propria richiesta e lo invia al servizio appropriato. Il servizio esegue il lavoro e rimanda indietro la risposta.



## Macchine virtuali

Il sistema di timesharing TSS/360 di IBM (1967) è stato il primo tentativo di virtualizzazione: più utenti potevano lavorare sui terminali collegati alla stessa macchina. Il sistema z/VM, utilizzato sugli attuali mainframe IBM zSeries, è il discendente diretto del TSS.

Mentre IBM disponeva già da quarant'anni di un prodotto di macchina virtuale, l'idea della virtualizzazione è stata ignorata nel mondo dei PC/server fino a pochi anni fa. La virtualizzazione si è diffusa anche nel mondo del web hosting.

Oggi anche gli utenti finali possono eseguire due o più sistemi operativi contemporaneamente sulla stessa macchina (VirtualBox, VMware, ...).

Quando Sun Microsystem (oggi Oracle) inventò il linguaggio di programmazione Java, creò anche una macchina virtuale (cioè un'architettura di computer) chiamata JVM (Java Virtual Machine).

## Exokernel

Piuttosto che clonare la macchina reale, come avviene con le macchine virtuali, un'altra strategia è partizionare, assegnando quindi a ogni utente un sottoinsieme delle risorse. Una macchina virtuale potrebbe così prendersi i blocchi del disco da 0 a 1023, la successiva i blocchi da 1024 a 2047 e così via.

Il livello base, che gira in modalità kernel, è un programma chiamato **exokernel**. Il suo compito è quello di allocare le risorse alle macchine virtuali e di controllare i tentativi di impiego, in modo che ciascuna VM utilizzi esclusivamente le proprie risorse.

Ogni macchina virtuale a livello utente esegue il proprio SO ed è limitata ad utilizzare le uniche risorse che ha chiesto e che le sono state assegnate.

Il vantaggio dello schema dell'exokernel è che risparmia uno strato di corrispondenza: Negli altri progetti ogni macchina virtuale pensa di avere il proprio disco, con blocchi che vanno da 0 a un massimo, così il monitor della macchina virtuale deve mantenere delle tabelle per rimappare gli indirizzi del disco (e di tutte le altre risorse). Con l'exokernel, questo rimappaggio non è più necessario, poiché esso necessita solo di tenere traccia di quale sia la macchina virtuale a cui è stata assegnata una certa risorsa.

Inoltre questo metodo tiene separata la multiprogrammazione (che è nell'exokernel) dal codice del sistema operativo utente (che è nello spazio utente).

## 1.f (NP) Il linguaggio C

I sistemi operativi sono normalmente scritti in C.

Un puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile o l'inizio di una struttura dati.

## Il linguaggio C

Java è basato su C quindi presenta molte somiglianze ad esso. Però, una caratteristica che C possiede e che Java non ha sono i puntatori esplicativi.

Si considerino le seguenti istruzioni:

```
char c1, c2, *p;  
c1 = 'x';  
p = &c1;  
c2 = *p;
```

in cui si dichiarano c1 e c2 come variabili carattere e p come una variabile che punta un carattere. La prima assegnazione salva il codice ASCII per il carattere 'x' nella variabile c1. La seconda assegna l'indirizzo della c1 alla variabile puntatore p. La terza assegna il contenuto della variabile puntata da p alla variabile c2. Così facendo, alla fine anche c2 contiene il codice ASCII di 'x'.

I puntatori sono costrutti estremamente potenti ma allo stesso tempo molto delicati: non avendo controlli è possibile indirizzare la memoria in modo improprio e commettere errori (motivo per il quale un programmatore Java non può utilizzarli, se non implicitamente).

La gestione della memoria del C è statica e allocata in modo esplicito (con la funzione malloc()) e rilasciata dal programmatore al termine del lavoro svolto (con la funzione free()).

I sistemi operativi sono dei veri e propri sistemi real-time utilizzati per scopi generali, quindi l'esistenza di un garbage collector che entra in azione in modo arbitrario non è ammissibile.

Inoltre il C è un linguaggio che ha un livello concettuale non troppo distante dalla macchina fisica. Tutte queste considerazioni, unite al controllo totale della memoria da parte dei programmatori, rendono il C il linguaggio ideale per scrivere i sistemi operativi.

## File di intestazione

I file di intestazione (o header) hanno estensione .h e contengono dichiarazioni e definizioni usate dichiarazioni, definizione e macro.

Le macro sono dichiarate attraverso la parola chiave `#define`. La macro è un pezzo di codice a cui è assegnato un nome: in fase di pre-processamento è sostituito dal frammento di codice corrispondente.

Un file sorgente C ha estensione `.c` e può includere uno o più file intestazione che utilizzano la direttiva `#include`

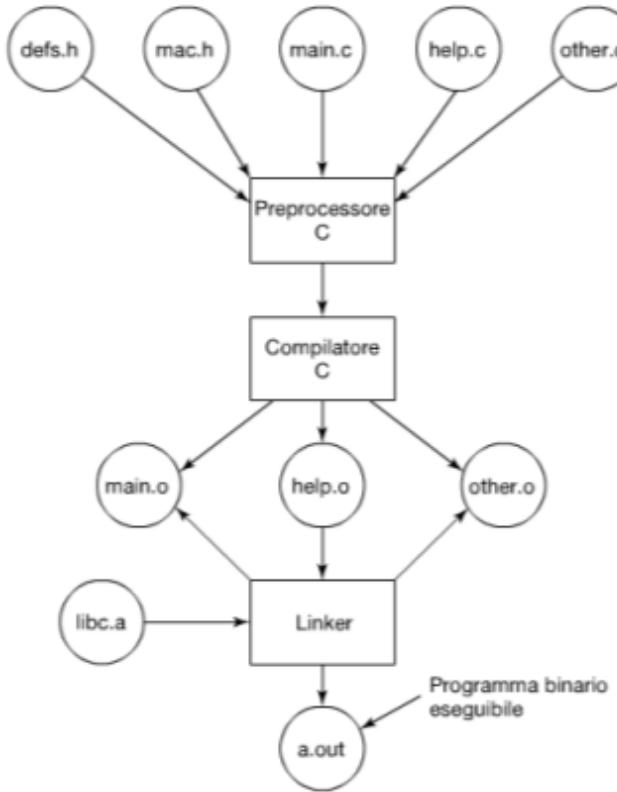
I file di intestazione possono anche contenere compilazioni condizionate, per esempio:

```
#ifdef X86  
intel_int_ack();  
#endif
```

che viene compilata in una chiamata verso la funzione `intel_int_ack` se la macro `X86` è definita, altrimenti nulla.

## Il modello di run-time

1. Il preprocessore assembla insieme i file sorgenti (inclusi quelli di intestazione).
2. Il compilatore produce i file oggetto (estensione `.o`);
3. Il linker collega tutti i file oggetto e le altre librerie creando un file eseguibile binario dipendente dalla macchina e ottimizzato per quella architettura.



## 2. Processi e thread

Il concetto fondamentale di ogni sistema operativo è il processo: un'astrazione di un programma in esecuzione.

### 2.a Processi

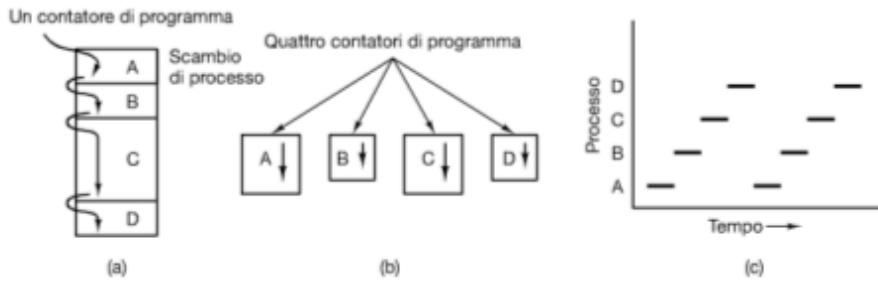
Tutti i computer moderni svolgono spesso molti compiti contemporaneamente. Per gestire tali attività è necessario un sistema di multiprogrammazione con più processi.

In ogni sistema multiprogrammato la CPU passa da processo a processo rapidamente (10:100ms), dando l'illusione di eseguire più processi contemporaneamente (pseudoparallelismo).

### Il modello di processo

I processi sono il software che è in esecuzione o “gira” sul computer (compreso il SO). Il processo è un’istanza dell’esecuzione di un programma, inclusi i valori attuali del program counter, dello stack, dei registri e delle variabili.

Concettualmente ogni processo ha la sua CPU virtuale. In realtà la CPU passa avanti e indietro da processo a processo (multiprogrammazione).



In (a) è mostrato un computer che fa multiprogrammazione di quattro programmi in memoria. In (b) sono mostrati quattro processi, ognuno con il suo flusso di controllo e ognuno che gira indipendentemente dagli altri. In (c) vediamo che, osservati su un intervallo abbastanza lungo, tutti i processi hanno avuto un avanzamento, ma in un preciso particolare istante solo un processo è realmente in esecuzione.

Mentre un programma ha una natura statica, la sua esecuzione (cioè il processo) ha una natura dinamica legata al valore del suo stato (PC, stack, ....). Un programma, inoltre, può originare più processi.

In UNIX il comando ps permette di conoscere i processi attivi; in Windows si utilizza il task manager.

## Creazione del processo

I sistemi operativi hanno bisogno di un modo per creare processi.

Un processo viene creato durante uno di questi eventi:

- inizializzazione del sistema;
- esecuzione di una chiamata di sistema di creazione di un processo;
- richiesta dell'utente di creare un processo;
- inizio di un job in modalità batch.

All'avvio vengono in genere creati vari processi, tra cui:

- i processi attivi che interagiscono con gli utenti e svolgono un compito per loro;

- i processi in background, non associati ad un utente in particolare, che svolgono funzioni specifiche all'arrivo di una richiesta (vengono anche chiamati demoni).

In UNIX, per creare un processo nuovo esiste una sola chiamata di sistema: fork(). Questa chiamata crea un clone esatto (figlio) del processo chiamante (padre). Dopo la fork i due processi hanno la stessa immagine della memoria, le stesse stringhe di ambiente e i medesimi file aperti.

Soltamente, il processo figlio poi esegue una execve() o una chiamata di sistema simile per cambiare la propria immagine di memoria ed eseguire un nuovo programma.

In Windows viene utilizzato CreateProcess() sia per creare un nuovo processo sia per il caricamento del programma corretto nel nuovo processo.

Sia in UNIX che in Windows, il genitore e il figlio hanno spazi di indirizzi distinti: ciò significa che una modifica ad una variabile dell'uno non influenza l'altro. Invece, sono condivisi i riferimenti a medesime risorse (es file, ...).

## Chiusura di un processo

Una volta creato, un processo comincia a girare fino a quando non ha svolto il proprio compito.

La chiusura di un processo si verifica a seguito di una delle seguenti condizioni:

- uscita normale (condizione volontaria): exit() in UNIX e ExitProcess() in Windows;
- uscita su errore (condizione volontaria): esempio bug nel programma;
- errore critico (condizione involontaria): esempio run-time error;
- terminato da un altro processo (condizione involontaria): kill() in UNIX e TerminateProcess() in Windows.

## Gerarchie di processi

Generalmente, quando un processo ne crea un altro, il processo genitore continua ad essere associato al processo figlio.

In UNIX, il processo figlio può a sua volta creare altri processi, formando una gerarchia di processi. Un processo, tutti i suoi figli e gli ulteriori discendenti costituiscono un gruppo di processi.

Windows, invece, non ha il concetto di gerarchia di processi. Tutti i processi sono uguali. Quando un processo crea un processo figlio ha un token speciale (chiamato handle) che può utilizzare per controllarlo ma anche cederlo ad altri invalidando il concetto di gerarchia.

## Stati di un processo

Sebbene ogni processo sia un'entità indipendente con il proprio program counter e il proprio stato interno, i processi hanno bisogno di interagire con altri processi. Un processo può generare alcuni output che un altro processo utilizza come input.

Un processo si blocca quando non è più nelle condizioni di svolgere il proprio compito: potrebbe essere in attesa di un risultato che ancora non è disponibile o il sistema operativo ha deciso di allocare la CPU ad un altro processo.

Queste due condizioni sono completamente differenti: nel primo caso la sospensione è inerente al problema, nel secondo è tecnicismo del sistema.

Nella Figura vediamo un diagramma di stato che mostra i tre stati in cui può trovarsi un processo:



- In esecuzione (CPU effettivamente in uso in quel momento);
- Pronto (può essere eseguito, è temporaneamente sospeso per consentire a un altro processo di essere eseguito);
- Bloccato (incapace di proseguire finché non avviene un qualche evento esterno).

Fra i tre stati sono possibili quattro transizioni, come mostrato nella figura. La transizione 1 accade quando il processo si blocca in attesa di output. La transizione 2 accade quando lo scheduler blocca il processo per metterne in esecuzione un altro. La transizione 3 si verifica quando lo scheduler mette in esecuzione l'altro processo. La transizione 4 accade quando l'input è disponibile.

Un processo in esecuzione può ricevere dei segnali di allarme che, analogamente agli interrupt nel caso hardware, ne bloccano l'esecuzione, causano il salvataggio dello stato e fanno sì che il processo esegua una procedura di gestione dei segnali speciali. Al termine della procedura il processo potrà riprendere l'esecuzione dal punto dove si era interrotto.

Molte trap rilevati dall'hardware, come l'esecuzione di una istruzione illegale o di un indirizzo errato, sono convertiti in segnali per il processo responsabile.

I segnali possono essere attivati anche in modo temporizzato, come nel caso dell'invio di un messaggio per eseguire un controllo sulla ricezione entro un certo timeout.

## Implementazione dei processi

Per implementare il modello di processo, il SO mantiene una tabella chiamata tabella dei processi, con una riga per processo, chiamata process control block. Questa voce contiene il program counter, lo stack pointer, l'allocazione della memoria, lo stato dei file aperti, le informazioni relative alla gestione e allo scheduling e qualunque altra cosa che serve salvare nello stato bloccato e pronto affinché sia possibile riavviare il processo come se non si fosse mai fermato.

## Modellazione della multiprogrammazione

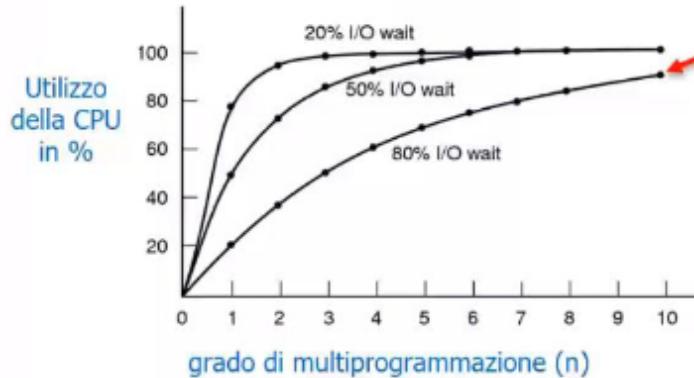
L'utilizzo della CPU può essere migliorato per mezzo della multiprogrammazione.

Se un processo in media esegue calcoli solo per il 20% del tempo in cui risiede in memoria, con 5 processi contemporaneamente in memoria la CPU dovrebbe essere occupata al 100%.

In realtà i processi attendono anche l'I/O.

Da un punto di vista probabilistico, se un processo spende una frazione  $p$  del suo tempo in attesa dell'I/O, con  $n$  processi in memoria la probabilità che stiano tutti aspettando l'I/O è  $p^n$ .

L'utilizzo della CPU è la probabilità complementare:  $1 - p^n$ .



Se i processi impiegano l'80% del loro tempo in attesa dell'I/O, devono come minimo esserci contemporaneamente 10 processi in memoria per portare la CPU ad un utilizzo del 90%. Questo modello semplificato permette di effettuare previsioni sull'utilizzo della CPU.

Esempio: un computer ha 512 MB di memoria, se il SO occupa 128 MB e un processo 128 MB, può contenere in memoria fino a 3 processi. Se l'attesa media dell'I/O è dell'80%, l'utilizzo della CPU è pari a  $1 - 0,8^3 = 49\%$ . Aggiungendo 512MB miglioriamo del 79% =  $1 - 0,8^7$ .

## 2.b Thread

Nei sistemi operativi tradizionali, ogni processo dispone di uno spazio degli indirizzi e di un singolo thread di controllo. Tuttavia ci sono frequenti situazioni in cui è desiderabile avere molteplici thread di controllo in esecuzione nello stesso spazio degli indirizzi, in quasi parallelo, come se fossero (quasi) processi separati.

### Uso dei thread

I thread sono dei processi leggeri o miniprocessi che, a differenza dei processi, sono tra loro fortemente correlati poiché condividono spazio degli indirizzi e dati.

Esistono varie ragioni per avere questi miniprocessi, esaminiamone alcune.

Il modello di programmazione diventa più semplice (l'applicazione si può decomporre in thread sequenziali che possono essere eseguiti in modalità quasi-parallela).

Essendo più leggeri sono più veloci da creare e cancellare.

Hanno un migliore utilizzo della CPU quando le attività sono I/O bound.

Un ambiente che consente a più thread di girare nello stesso processo è chiamato multithreading. Le CPU moderne che supportano il multithreading riescono a passare da un thread all'altro nell'ordine dei nanosecondi.

## Il modello a thread classico

Il modello di processo si basa su due concetti indipendenti:  
Raggruppamento di risorse;  
Esecuzione.

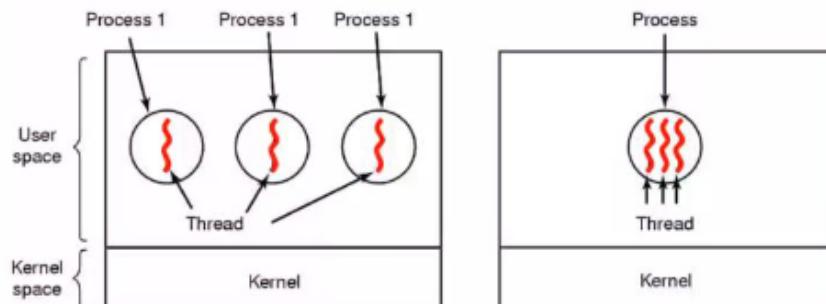
Un processo può essere visto come un modo per raggruppare risorse:  
Uno spazio degli indirizzi (programma e dati);  
File aperti, processi figli, situazioni d'allarme in sospeso, gestori di segnale, informazioni sugli account e altro.

I thread invece sono entità schedulate per l'esecuzione. Più thread possono essere eseguiti nello stesso ambiente di processo.

Il thread ha un program counter (che tiene traccia di quale istruzione eseguire come successiva), alcuni registri (che contengono le sue variabili di lavoro attuali), uno stack (contenente la storia della sua esecuzione) e uno stato.

I thread hanno i medesimi stati dei processi e la stessa dinamica pronto-esecuzione-bloccato.

Più processi che girano in parallelo su un computer non è la stessa cosa di più thread in parallelo. Nel primo caso, i processi condividono solo la memoria fisica, i dischi e le altre risorse; nel secondo i thread condividono anche lo spazio di indirizzamento, lo stack, i registri e lo stato.



Nella Figura (a) vediamo tre processi tradizionali. Ogni processo ha il suo spazio degli indirizzi e un singolo thread di controllo. Nella Figura (b) vediamo invece un singolo

processo con tre thread di controllo. Sebbene in entrambi i casi abbiamo tre thread, nella Figura (a) ognuno di loro opera in un diverso spazio degli indirizzi, mentre nella Figura (b) tutti e tre condividono lo stesso spazio degli indirizzi.

I thread in un processo non sono tanto indipendenti quanto i processi concorrenti.

Poiché tutti i thread hanno lo stesso spazio di indirizzamento (le stesse variabili globali), possono leggere, scrivere o cancellare lo stack di un altro thread. Tra i thread, infatti, non esiste protezione perché è impossibile realizzarla e non dovrebbe essere necessaria.

Mentre i processi possono essere di proprietà di diversi utenti e competere per ottenere risorse comuni, ogni processo (di proprietà di un singolo utente) può creare più thread che dovrebbero cooperare non entrando mai in conflitto tra loro.

L'organizzazione a processi dovrebbe essere utilizzata quando le attività dei processi non sono tra loro correlate. Mentre un modello a thread si dovrebbe utilizzare quando i compiti di ciascuno sono parte dello stesso lavoro e, quindi, è necessaria una stretta collaborazione.

Elementi per processo	Elementi per thread
Spazio degli indirizzi	Contatore di programma
Variabili globali	Registri
File aperti	Stack
Processi figli	Stato
Allarmi in sospeso	
Segnali e gestori dei segnali	
Informazioni relative agli account	

Ogni thread ha un proprio stack.

In un ambiente multithreading i processi partono con un singolo thread, quest'ultimo ha la capacità di crearne altri chiamando la procedura `thread_create()`. All'atto della creazione non è necessario specificare nulla poiché il nuovo thread girerà nel medesimo spazio di indirizzamento del thread che l'ha creato. Talvolta i thread possono avere relazioni gerarchiche (padre-figlio) oppure essere allo stesso livello (più comune). Il thread che crea un altro thread riceve in uscita l'identificatore del thread creato.

Quando un thread ha terminato il suo lavoro, esso può uscire chiamando la procedura `thread_exit()`.

In alcuni sistemi, un thread può rimanere in attesa di un altro thread attraverso la procedura `thread_join()`.

Un'altra chiamata comune è la `thread_yield()`, che permette di rilasciare la CPU ad un altro thread. Questa chiamata è importante perché permette di realizzare la multiprogrammazione senza che ci sia interruzione, così come accade per i processi.

## Alcuni problemi presenti nei thread

Se un processo che ha più thread crea un processo figlio, quest'ultimo avrà gli stessi thread del padre? In caso contrario, il processo potrebbe non funzionare correttamente, dato che ognuno di loro potrebbe essere essenziale. In caso affermativo, che cosa accadrebbe se un thread nel genitore fosse bloccato su una chiamata di `read`, ad esempio, della tastiera? Ora ci sarebbero due thread bloccati sulla tastiera, uno nel genitore e uno nel figlio. Quando viene digitata una riga, entrambi i thread ne riceveranno una copia? Solo il genitore? Solo il figlio?

Un'altra complicazione è relativa al fatto che i thread condividono molte strutture di dati. Che cosa accade se un thread chiude un file mentre un altro lo sta ancora leggendo?

Anche se questi problemi possono essere risolti con un po' di attenzione, occorre avere le idee chiare su come dovrebbero andare le cose nei casi descritti.

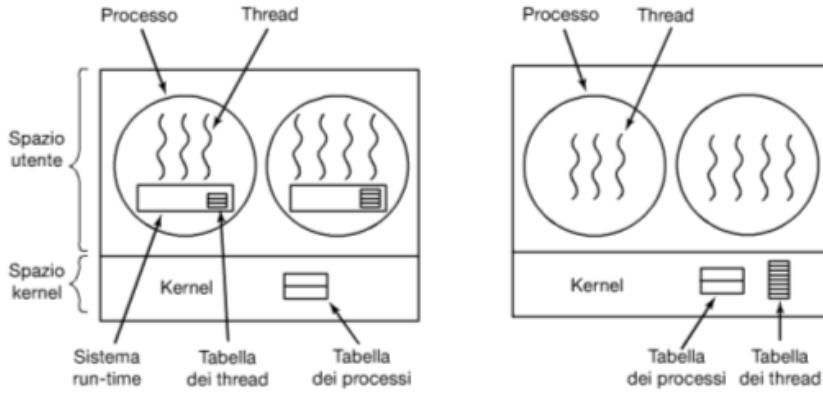
## Thread POSIX

Per permettere la scrittura di programmi portabili che usino i thread, IEEE ha definito i thread nello standard IEEE 1003.1c. I thread definiti nello standard sono chiamati Pthread e sono supportati dai principali sistemi UNIX. Questo standard definisce più di 60 chiamate di sistema specifiche per i Pthread.

Ciascun pthread ha un identificatore, un insieme di registri (compreso il program counter) e un insieme di attributi (dimensione dello stack, parametri per lo scheduling, ...) memorizzati in una struttura.

## Realizzazione dei thread

I thread possono essere gestiti nello spazio utente (a), nel kernel (b) o in entrambi.



## Implementazione dei thread nello spazio utente

Il pacchetto di thread viene messo interamente nello spazio utente. Il kernel non ne è a conoscenza poiché, per quanto lo riguarda, gestisce processi ordinari, a singolo thread.

Così facendo, è possibile realizzare thread anche su sistemi operativi che non li supportano. Normalmente sono realizzati in questo modo attraverso una libreria.

Questa realizzazione ha una struttura generale illustrata nella (a). I thread sono eseguiti sopra un sistema run-time, che è una raccolta di procedure che gestiscono i thread.

In questa realizzazione ogni processo ha bisogno della sua tabella dei thread personale, per tener traccia dei thread nel processo. Essa è simile alla tabella dei processi del kernel, ad eccezione del fatto che mantiene traccia solo delle proprietà dei thread (PC, SP, registri, stato, ...). La tabella dei thread è gestita nell'ambiente di run-time.

## Vantaggi dei thread nello spazio utente

Quando un thread termina l'esecuzione, la procedura che salva lo stato del thread e lo scheduler sono locali, in questo modo la loro invocazione è molto più efficiente rispetto ad una system call del kernel. Questo rende lo scheduling dei thread molto veloce.

Questo tipo di implementazione permette a ogni processo di avere un proprio algoritmo personalizzato di schedulazione. I thread possono crescere e replicarsi numerosi poiché non esistono dei limiti (se non quelli della disponibilità della memoria) nello spazio utente. Nel kernel invece non può crescere troppo la tabella dei processi o la dimensione dello stack.

## Svantaggi dei thread nello spazio utente

Se un thread si blocca (per una chiamata bloccante o page fault) il kernel blocca l'intero processo in cui esso è inserito. Bisogna permettere a ognuno di usare chiamate bloccanti, ma di impedire che un thread bloccato possa interferire con gli altri. Tutte le chiamate di sistema potrebbero essere cambiate con nuove chiamate non bloccanti oppure anticiparle con istruzioni specifiche tipo select().

Un altro problema è che nessun altro thread nel processo potrà essere eseguito fino a che un thread in esecuzione non rilascia volontariamente la CPU.

Il sistema di run-time può utilizzare una modalità timesharing e round-robin utilizzando i segnali di allarme, ma oltre a creare sovraccarico al crescere della frequenza ( $>1$  volta/sec), si corre il rischio di non poterli più utilizzare per gestire le altre situazioni dove sono comunque utili.

I programmati usano i thread proprio nelle applicazioni che si bloccano spesso (come i Web Server che fanno molte system call), in questo caso è difficile per il kernel passare da un thread all'altro.

## **Implementazione dei thread nel kernel**

Supponiamo ora che il kernel sia a conoscenza dei processi e li gestisca. La tabella dei thread è nel kernel, non nel singolo processo. Quando un thread vuole creare uno nuovo o distruggerne uno esistente, esso esegue una chiamata al kernel, che esegue la creazione o la distruzione, aggiornando la tabella dei thread del kernel.

La tabella dei thread contiene sempre i registri dei thread, lo stato e altre informazioni, ma ora stanno nel kernel invece che nello spazio utente.

## **Vantaggi dei thread nel kernel**

Tutte le chiamate che potrebbero bloccare un thread sono realizzate come chiamate di sistema. Quindi il blocco dei thread non è più un problema in quanto il kernel può passare dall'uno all'altro senza problemi. I thread nel kernel non richiedono quindi la riscrittura di chiamate di sistema non bloccanti.

## **Svantaggi dei thread nel kernel**

Tutte le chiamate sono realizzate con system call, con maggiori costi rispetto ad una chiamata ad un sistema di procedure run-time.

A causa di questo elevato costo di creazione e distruzione dei thread alcuni sistemi riciclano le strutture dati utilizzate per i thread distrutti.

Che succede in un ambiente multithread quando un processo esegue una fork? Il nuovo processo ha lo stesso numero di thread del padre o solo uno?

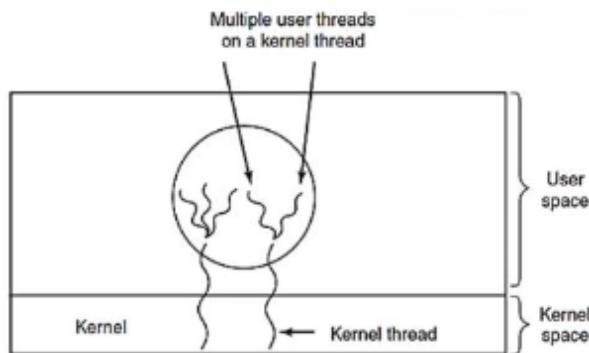
Nel modello classico, i segnali possono essere spediti ai processi e non ai thread.

Quando arriva un segnale ad un processo che contiene più thread quale di essi dovrà gestirlo?

## Implementazioni ibride

Sono stati studiati vari modi per cercare di combinare i vantaggi delle due soluzioni.

Una possibilità è usare i thread del kernel e fare in modo che ciascuno di essi utilizzi a turno un thread di livello utente, come mostrato nella Figura.



Con tale approccio il kernel è in grado di schedulare i soli thread a livello kernel.

## Attivazioni dello scheduler

In certe situazioni i thread nel kernel sono migliori di quelli a livello utente, ma sono anche più lenti. Le attivazioni dello scheduler simulano la funzionalità dei thread nel kernel, ma con migliori prestazioni e maggiore flessibilità.

L'efficienza è ottenuta evitando inutili transizioni tra lo spazio utente e quello del kernel quando un processo si blocca. Il kernel assegna un certo numero di processori virtuali ad ogni processo e fa in modo che il sistema di run-time assegna i thread ai processori.

Quando il kernel riconosce che un thread è bloccato, lo comunica al sistema di run-time (upcall): a questo punto il sistema di run-time può rischedulare i thread.

Le attivazioni dello scheduler dipendono dalle upcall che violano il concetto che un livello inferiore possa utilizzare i servizi di uno superiore tipico delle architetture a strati.

## Thread pop-up

I thread sono spesso utili negli ambienti distribuiti. Un esempio è la gestione dei messaggi in ingresso per richieste di un servizio. Quando arriva un nuovo messaggio il sistema crea un nuovo thread (chiamato pop-up) per gestire il messaggio stesso. Questo thread è detto thread pop-up.

Dato che nascono nel momento in cui arriva il messaggio non hanno alcuna storia da ripristinare (registri, stack o altro). Ognuno viene avviato da zero e ognuno è identico agli altri, quindi la creazione è velocissima: quello che serve quando arriva un messaggio.

Il thread pop-up può essere eseguito sia nello spazio utente che nel kernel. Di solito, si preferisce questa seconda strada per la semplicità di accesso alle tabelle del kernel e ai dispositivi di I/O, anche se è più pericolosa.

## 2.c Comunicazioni tra processi

I processi necessitano di comunicare con altri processi, preferibilmente in un modo ben strutturato, non usando gli interrupt. Questa comunicazione tra processi si chiama IPC (InterProcess Communication).

I problemi da affrontare sono tre:

- Come un processo possa passare informazioni a un altro;
- Come gestire le situazioni in cui due o più processi competono per accedere ad una risorsa comune evitando che si sovrappongano;
- Definire regole di sincronizzazione tra processi dipendenti (es. produttore/consumatore).

È anche importante menzionare che due di questi problemi si possono applicare ugualmente bene anche i thread. Il primo invece è semplice per i thread, dato che condividono uno spazio degli indirizzi comune.

## Corse critiche

I processi che stanno lavorando insieme possono condividere una parte di memoria comune che ciascuno può leggere e scrivere.

Le corse critiche nascono quando due o più processi stanno leggendo o scrivendo dati condivisi e il risultato finale dipende dalla sequenza di esecuzione.

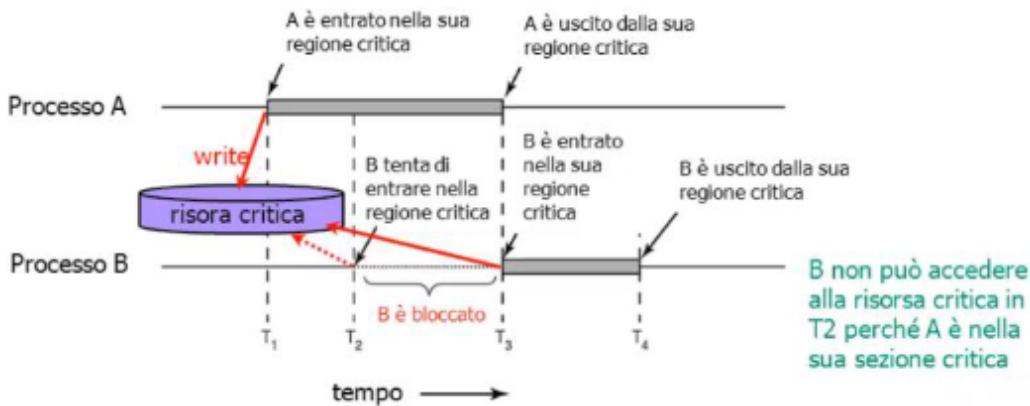
## Sezioni critiche

La chiave per evitare le corse critiche è vietare che più processi possono leggere e scrivere le risorse condivise contemporaneamente (mutua esclusione). Cioè un qualche sistema per essere certi che, se un processo sta usando una risorsa, l'altro processo venga escluso dal fare la stessa cosa.

La scelta delle operazioni primitive appropriate per il raggiungimento della mutua esclusione è uno dei maggiori problemi progettuali di ogni sistema operativo.

La questione delle race condition può anche essere formulata nel seguente modo: Un processo A fa calcoli e poi deve accedere alla memoria condivisa (sezione o regione critica). Arriva B ma è bloccato fino a che A non esce dalla regione critica.

Se possiamo fare in modo che due processi non si trovino mai nelle loro regioni critiche allo stesso momento, avremmo trovato la soluzione alle race condition.



Per ottenere una buona soluzione servono quattro condizioni:

- due processi non devono mai trovarsi contemporaneamente nelle loro regioni critiche;
- non può essere fatta alcuna ipotesi sulle velocità o sul numero delle CPU;
- nessun processo in esecuzione al di fuori della sua regione critica può bloccare altri processi;

- nessun processo dovrebbe restare in attesa infinita di entrare nella sua regione critica.

## Mutua esclusione con busy waiting

La mutua esclusione è possibile utilizzando il busy waiting: mentre un processo è nella sua regione critica gli altri devono attendere. Esistono varie soluzioni.

### Disabilitare gli interrupt

In un sistema a singolo processore la soluzione più semplice è quella di disabilitare tutti gli interrupt appena entrato nella sua regione critica e li riabiliti prima di lasciarla. In questo modo la CPU non può passare da un processo all'altro.

Ogni processo può aggiornare le risorse condivise senza timore che qualsiasi altro processo possa interferire.

Un simile approccio non è generalmente molto invitante perché non è saggio dare ai processi utente la possibilità di disabilitare gli interrupt. Di solito è il kernel che decide quando disabilitare gli interrupt se sta eseguendo istruzioni critiche (in questo caso un processo utente che si intromette può portare a degli errori).

In un sistema multiprocessore disabilitare gli interrupt di una CPU non impedisce alle altre CPU di averli attivi e quindi di interferire.

Questa non rappresenta una buona soluzione, sono necessari sistemi più sofisticati.

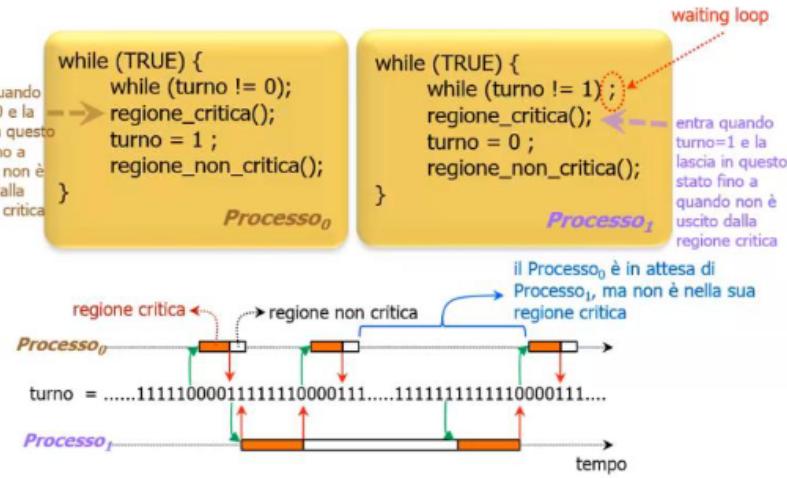
### Variabili lock

Si tratta di una soluzione software semplice: si utilizza una variabile condivisa (chiamata lock) per bloccare la risorsa, inizialmente posta a 0. Quando un processo vuole entrare nella sua regione critica, prima controlla il lock. Se esso è 0, il processo lo imposta a 1 ed entra nella regione critica. Se invece il lock è posto a 1, il processo aspetta finché non vale 0.

Questa soluzione presenta però il seguente difetto: supponiamo che un processo legga il lock e veda che è 0. Prima che riesca a metterlo a 1, viene eseguito un altro processo e, anch'esso leggendo il lock a 0, lo imposta ad 1. A questo punto sono entrambi nella sezione critica.

Non vi è alcuna garanzia che una risorsa ritenuta “libera” perché ha lock posto a 0 non diventi occupata ancora prima che il processo le assegni il valore di “occupata”.

## Alternanza stretta



La variabile interna *turn*, inizialmente a 0, tiene traccia di chi tocca entrare nella regione critica. In principio, il processo 0 controlla *turn*, la trova a 0 ed entra nella sua regione critica. Anche il processo 1 crede che sia 0 e quindi rimane in attesa, in un rapido ciclo, testando continuamente *turn* per vedere quando diventa 1.

Quando il processo 0 lascia la regione critica, imposta *turn* a 1, per permettere al processo 1 di entrare nella sua regione critica.

Questa tecnica è chiamata busy waiting e andrebbe evitata perché spreca tempo di CPU. Si dovrebbe utilizzare quando l'attesa è breve. Un lock che utilizza il busy waiting è chiamato spin lock.

Questo algoritmo evita le corse critiche ma viola la condizione 3, cioè che un processo possa rimanere bloccato se un altro non è nella regione critica.

## Soluzione di Peterson (1981)

```

#define FALSE 0
#define TRUE 1
#define N 2
int turno;
int interessato[N];
void entra_in_regione (int processo) { /* numero dei processi */
    /* Turno stabilisce a chi tocca */
    /* inizializzato con tutti false (0) */
    int altro;
    altro = 1 - processo; /* il processo complementare */
    interessato[processo] = TRUE; /* il processo è interessato! */
    turno = processo; /* è il turno del processo */
    while (turno == processo && interessato[altro] == TRUE); /* il processo vale 0 o 1 */
}
void lascia_la_regione (int processo){ /* il processo che lascia la */
    interessato[processo] = FALSE; /* regione non è più interessato */
}

```

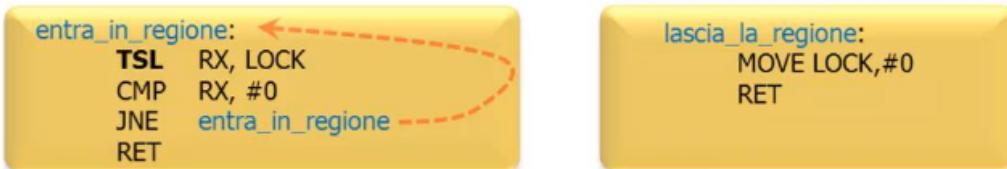
Prima di entrare nella propria regione critica, ogni processo chiama enter\_region con il suo numero di processo personale come parametro. Questa chiamata farà sì che si metta in attesa, se necessario, finché non è sicuro entrare. Dopo che ha finito con la regione critica, il processo chiama leave\_region per indicare che ha terminato e per consentire eventualmente l'ingresso dell'altro processo.

Anche se i due processi invocassero quasi contemporaneamente la procedura, solo l'ultimo scriverebbe la variabile turno. In questo modo solo uno dei due avrebbe la possibilità di non rimanere bloccato dal ciclo while.

## Istruzione TSL

Alcune CPU possiedono nativamente l'istruzione TSL REGISTER, LOCK (Test and Set Lock). Tale istruzione assembly legge il contenuto della variabile lock nel registro RX e poi salva un valore non zero all'indirizzo di memoria lock. Le operazioni di lettura della parola e del suo salvataggio sono garantite come indivisibili, cioè nessun altro processore può accedere alla parola di memoria finché l'istruzione non è terminata. La CPU che esegue l'istruzione TSL chiude il bus di memoria e vieta ad altre CPU di accedere alla memoria finché non ha finito.

Per realizzare la mutua esclusione con questo metodo si può usare il codice illustrato nella Figura.



Prima di entrare nella regione critica, un processo chiama `enter_region` e rimane in `busy waiting` finché il lock non è libero. All'uscita invoca `leave_region`.

Un'istruzione alternativa a TSL è XCHG (exchange), che scambia il contenuto di due posizioni atomicamente (es. tra registro e memoria).

## Sleep e wakeup

Sia la soluzione di Peterson che l'istruzione TSL (o XCHG) sono corrette ma hanno entrambe due problemi:

Fanno sprecare tempo di CPU perché richiedono `busy waiting`.

Un processo con bassa priorità che è sospeso e mantiene la regione critica non riesce a liberare la risorsa a causa della presenza di un processo con più alta priorità che è in esecuzione ma in `busy waiting` (problema dell'inversione delle priorità)

Per superare queste problematiche esistono due primitive:

- `sleep()`: blocca il processo chiamante fino a che un altro processo non lo risveglia;
- `wakeup(p)`: sveglia il processo `p`.

## Il problema del produttore-consumatore

Due processi condividono un buffer comune di dimensione fissata. Uno di loro, il produttore, mette informazioni nel buffer e l'altro, il consumatore, le preleva.

Esistono due problemi:

- quando il produttore vuole inserire un nuovo elemento ma il buffer è pieno;
- quando il consumatore vuole leggere un elemento ma il buffer è vuoto.

La soluzione prevede che un processo (produttore/consumatore) rimanga dormiente finché l'altro (produttore/consumatore) non inserisce/rimuove elementi.

Per tenere traccia del numero di elementi nel buffer si utilizza la variabile `count`.

```

#define DIM 100           /* dimensione del buffer */
int contatore = 0;      /* numero di elementi presenti nel buffer */
void produttore(void){   /* ciclo infinito
    int elemento;        /* produce un elemento
    while (TRUE) {        /* se il buffer è pieno
        elemento = produce(); /* addormentati
        if (contatore==DIM) /* (altrimenti) inserisce l'elemento nel buffer
            sleep();     /* incrementa il numero di elementi presenti
        inserisce(elemento); /* se il buffer era vuoto,
        contatore++;        /* occorre ri-svegliare il consumatore
        if (contatore==1)    /* addormentati
            wakeup(consumatore); /* utilizza l'elemento
    } }                   */
void consumatore(void){  /* ciclo infinito
    int elemento;        /* se il buffer è vuoto,
    while (TRUE) {        /* addormentati
        if (contatore==0) /* (altrimenti) estraie un elemento dal buffer
            sleep();     /* decrementa il numero di elementi presenti
        elemento = estrai (); /* se il buffer era pieno,
        contatore--;       /* occorre svegliare il produttore
        if (contatore==DIM-1) /* utilizza l'elemento
            wakeup(produttore);
        usa(elemento);    /* addormentati
    } }                   */
}

```

Supponiamo che il buffer inizialmente sia vuoto e sia avviato il consumatore che trova count a 0. A questo punto lo scheduler decide di interromperlo (prima della sleep()) e avviare il produttore. Il produttore inserisce un elemento nel buffer e incrementa count. Ora count è posto a 1, quindi cerca di svegliare il consumatore (che non dorme ancora). A questo punto lo scheduler attiva il consumatore che va subito in sleep(). Quando il produttore riempirà il buffer dormiranno entrambi.

La sostanza del problema è che **wakeup(p)** inviato ad un processo sveglio va perso. Per risolvere tale problematica si può aggiungere un bit di attesa **wakeup**. Se si invia un **wakeup(p)** ad un processo sveglio, si imposta questo bit a 1; quando il processo tenterà di addormentarsi, verificherà lo stato del bit. Se è 1 allora lo azzera e rimane sveglio; se è 0 si addormenta.

Questa soluzione termina la sua efficacia nel momento in cui aumentano i processi ed un bit non è più sufficiente a descriverne gli stati. Si possono aumentare i bit di **wakeup**, ma il principio del problema rimane sempre.

## Semafori

L'idea alla base di un semaforo è quella di contare il numero di risvegli. Un semaforo può essere 0 (nessun risveglio) o assumere un valore positivo quando uno o più risvegli sono

in attesa.

Su un semaforo sono possibili due operazioni: down() e up() (rispettivamente generalizzazioni di sleep() e wakeup()).

L'operazione down() decrementa il valore del semaforo se semaforo > 0, altrimenti (semaforo = 0) va in sleep senza completare il down().

L'operazione up() aumenta il valore del semaforo.

Più processi possono essere regolati da un semaforo.

Dopo un'operazione up() su un semaforo con più processi dormienti, il semaforo sarà ancora 0, ma un processo (scelto a caso dal sistema) potrà completare il suo down() risvegliandosi.

È essenziale sottolineare che tutte queste operazioni vengono fatte come indivisibili azioni atomiche.

I semafori risolvono il problema della perdita di wakeup, come illustrato nella Figura.

```
#define N 100          /* dimensione del buffer */
typedef int semaforo; /* i semafori sono interi >0 */
semaforo mutua_esclusione = 1; /* controlla l'accesso alla regione critica */
semaforo vuoto = N;           /* conta il numero di posizioni vuote */
semaforo pieno = 0;           /* conta il numero di posizioni riepite */
void produttore(void){
    int elemento;
    while (TRUE) {
        elemento = produce(); /* genera un element da inserire nel buffer */
        down(&vuoto);       /* decremente vuoto */
        down(&mutua_esclusione); /* entra nella regione critica */
        inserisce(elemento); /* inserisce l'elemento nel buffer */
        up(&mutua_esclusione); /* abbandano la regione critica */
        up(&pieno);         /* incrementa il numero di posizioni riempite */
    }
}
void consumatore(void){
    int elemento;
    while (TRUE) {
        down(&pieno);      /* decremeta il numero di posizioni riempite */
        down(&mutua_esclusione); /* entra nella regione critica */
        elemento = estrai(); /* prende un element dal buffer */
        up(&mutua_esclusione); /* abbandano la regione critica */
        up(&vuoto);         /* incrementa il numero di posizioni vuote */
        consume(elemento);   /* utilizza l'elemento estratto */
    }
}
```

I semafori risolvono il problema della perdita dei risvegli che abbiamo visto con sleep() e wakeup(p), le primitive down() e up() sono realizzate come system call.

Il sistema operativo prima di controllare il semaforo, disabilita brevemente tutti gli interrupt in modo da rendere le primitive atomiche. Nel caso ci siano più CPU occorre

proteggere il semaforo con variabili di lock o istruzioni TSL (o XCHG) in modo da essere sicuri che solo una CPU alla volta possa esaminare il semaforo.

Il semaforo è binario poiché assume due valori e fa in modo che un solo processo per volta possa entrare nella sezione critica.

## Mutex

I mutex sono una versione semplificata del semaforo. Essi sono utili solo per gestire la mutua esclusione di risorse condivise o pezzi di codice.

Sono facili da implementare ed efficienti, ciò li rende particolarmente utili nei pacchetti di thread. Un mutex è una variabile binaria che può trovarsi in soli due stati: locked o unlocked.

Anche se basta un bit per rappresentare un mutex, spesso si usa un numero intero: 0 significa unlocked, tutti gli altri valori significano locked.

Con i mutex si usano due procedure:

- `mutex_lock()`: quando un processo ha bisogno di entrare nella regione critica;
- `mutex_unlock()`: quando ha terminato l'accesso alla regione critica.

I mutex sono così semplici che se l'istruzione TSL è disponibile, possono essere realizzati nello spazio utente. Il codice di `mutex_lock` e `mutex_unlock` è mostrato nella Figura.

```
mutex_lock:  
    TSL REGISTER, MUTEX | copia mutex sul registro e pone mutex=1  
    CMP REGISTER,#0    | mutex=0?  
    JZE ok             | se mutex=0, mutex è sbloccato ed esce  
    CALL thread_yield  | #0 è occupato, schedulare un altro thread  
    JMP mutex_lock     | prova ancora  
ok: RET              | ritorna al chiamante è nella regione critica  
  
mutex_unlock:  
    MOVE MUTEX, #0     | memorizza 0 nel mutex  
    RET                 | ritorna al chiamante
```

Esiste una differenza cruciale tra `enter_region` e `mutex_lock`. La prima crea busy waiting, la seconda invece, quando un thread fallisce nell'acquisire un lock, chiama `thread_yield` per cedere la CPU a un altro thread, evitando il busy waiting.

I pthread forniscono un certo numero di funzioni utilizzabili per sincronizzare i thread. Il meccanismo di base utilizza una variabile mutex, che può essere locked o unlocked, a guardia di ogni regione critica.

Un thread che vuole entrare nella regione critica prima prova a bloccare il mutex associato. Se il mutex è unlocked, il thread può entrare immediatamente e il lock è impostato atomicamente impedendo l'ingresso ad altri thread. Se il mutex è già locked, il thread chiamante è bloccato finché non diviene unlocked.

Se molteplici thread sono in attesa dello stesso mutex, quando passa in stato unlocked, solo a uno è consentito di continuare e di metterlo così a sua volta in locked. Questi lock non sono obbligatori, sta al programmatore essere certo che i thread li utilizzino correttamente.

Thread call	descrizione
Pthread_mutex_init	crea un mutex
Pthread_mutex_destroy	elimina un mutex
Pthread_mutex_lock	tenta il lock e si blocca
Pthread_mutex_trylock	tenta il lock o fallisce
Pthread_mutex_unlock	rilascia un lock

I pthread offrono un secondo meccanismo di sincronizzazione: le variabili condizione. I mutex sono buoni per permettere o meno l'accesso ad una regione critica. Le variabili condizione permettono di bloccare i processi se non si verificano alcune condizioni.

Thread call	descrizione
Pthread_cond_init	crea una variabile condizione
Pthread_cond_destroy	elimina una variabile condizione
Pthread_cond_wait	si blocca in attesa di un segnale
Pthread_cond_signal	segnale di risveglio per un thread
Pthread_cond_broadcast	risveglia tutti i thread addormentati

## Monitor

Con i semafori e mutex la comunicazione tra processi sembra apparentemente facile, invece è difficile scrivere programmi corretti.

Un monitor è un insieme di procedure, variabili e dati strutturati, raggruppate in un particolare tipo di modulo o pacchetto. I processi possono richiedere le procedure in un monitor ogni volta che vogliono, ma non possono accedere direttamente alle strutture dati interne del monitor.

I monitor hanno una proprietà importante che li rende utili per realizzare la mutua esclusione: in un dato istante, un solo processo per volta può essere attivo in un monitor.

La soluzione è l'uso delle variabili condizione abbinate alle due operazioni: wait() e signal(). Quando una procedura dentro il monitor scopre che non può continuare (es.

buffer pieno o vuoto) esegue una `wait()` su una variabile condizione e blocca il processo. Un altro processo partner può risvegliare il processo inviando una `signal()` sulla variabile condizione.

Le variabili condizione non sono contatori. Non accumulano segnali per usarli in seguito come fanno i semafori.

Anche `sleep()` e `wakeup()` sembrano simili alle operazioni `wait()` e `signal()`, ma le prime avevano il difetto che mentre un processo stava tentando di entrare in sleep non si accorgeva del `wakeup(p)` dell'altro se non era già entrato in sleep. Con i monitor ciò non accade.

## Scambio di messaggi

Questo metodo usa due primitive, `send()` e `receive()` che, come i semafori e diversamente dai monitor, sono chiamate di sistema piuttosto che costrutti del linguaggio.

- `send(destination, &message)`: invia un messaggio ad un destinatario;
- `receive(source, &message)`: riceve un messaggio da una fonte.

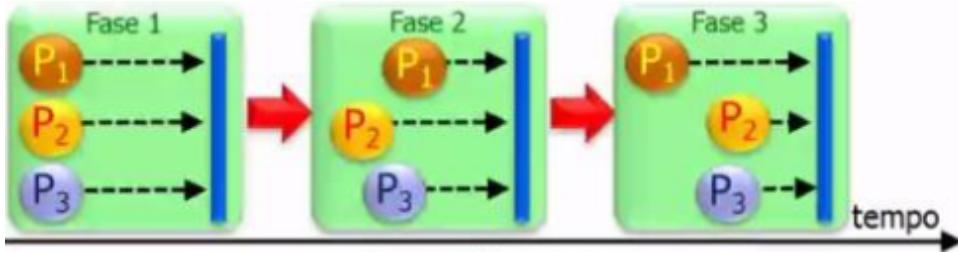
Se il messaggio non è disponibile, il ricevente può rimanere bloccato finché non ne riceve uno, oppure può restituire immediatamente un codice di errore.

## Barriere

Il meccanismo di sincronizzazione delle barriere è destinato a gruppi di processi, piuttosto che a comunicazioni tra due processi.

Alcune applicazioni sono suddivise in fasi e hanno la regola che nessun processo può proseguire nella fase successiva finché tutti i processi non sono pronti a procedere alla fase successiva.

Questa condotta può essere ottenuta mettendo una barriera al termine di ogni fase. Quando un processo raggiunge la barriera, viene bloccato finché tutti gli altri processi non raggiungono la barriera.



Dopo che il primo processo termina esegue la primitiva barrier() e viene sospeso. Solo quando l'ultimo processo arriva alla barriera potranno essere tutti rilasciati.

## **2.d Problemi classici di comunicazione tra processi**

La letteratura dei sistemi operativi è piena di problemi interessanti che sono stati discussi utilizzando una varietà di metodi di sincronizzazione.

### **Problema dei filosofi a cena**

Cinque filosofi sono seduti a un tavolo circolare con un piatto di spaghetti davanti. Per mangiare gli spaghetti sono necessarie due forchette e sul tavolo ci sono solo 5 forchette, quindi solo due filosofi possono mangiare contemporaneamente. Quando non mangia, il filosofo pensa.

Siamo in grado di scrivere un programma tale che per ciascun filosofo gli permetta di pensare per il tempo a lui necessario e mangiare senza che il sistema si blocchi?

La procedura take\_fork(i) aspetta finché la i-esima forchetta non è disponibile, quando è libera la solleva.

Sfortunatamente questa è una soluzione sbagliata: se tutti i filosofi prendono la forchetta di sinistra contemporaneamente nessuno sarà in grado di trovare l'altra e ci sarà uno stallo (deadlock).

### **Starvation**

Potremmo modificare il programma in modo che un filosofo dopo aver preso la forchetta sinistra controlli quella di destra. Se disponibile, la prende; altrimenti, ripone la sinistra sul tavolo, attende per un certo tempo e ripete ciclicamente il processo.

Tuttavia questa soluzione non funziona per un altro motivo: se tutti i filosofi iniziano l'algoritmo contemporaneamente prendono la forchetta di sinistra e guardano che manca

quella alla loro destra, allora posano quella alla loro sinistra e ripetono questa sequenza di passaggi all'infinito.

Una situazione come questa, in cui tutti i programmi continuano ad essere eseguiti indefinitivamente, senza riuscire a fare alcun progresso, si chiama Starvation.

## Alcune soluzioni al problema

Una possibile soluzione sfrutta un'attesa casuale tra il momento in cui viene rilasciata la forchetta alla sinistra, perché quella alla destra non è disponibile, e il nuovo tentativo. In molti campi di applicazioni non è un problema riprovare finché non si estrae un numero casuale vincente, tuttavia non è accettabile.

Un'altra soluzione sfrutta il miglioramento di quella con deadlock: si proteggono le istruzioni che seguono think() con un semaforo binario. Prima di iniziare ad acquisire le forchette, un filosofo esegue una down() su un mutex e dopo aver riposto la forchetta un up() sullo stesso mutex. Questa soluzione è idonea ma non efficiente: mangia un solo filosofo per volta.

## 2.e Scheduling

Quando un computer è multiprogrammato, ha più processi o thread che competono per ottenere la CPU nel medesimo istante. Questa situazione si verifica quando due o più di loro sono in stato pronto. Con una CPU disponibile un solo processo/thread può essere selezionato per entrare in esecuzione. La parte del sistema operativo che effettua questa scelta è chiamato scheduler e l'algoritmo utilizzato è detto algoritmo di scheduling.

Molti dei problemi che si applicano allo scheduling dei processi si applicano anche ai thread. Quando il kernel gestisce i thread, lo scheduling è fatto per thread indipendentemente dal processo di appartenenza.

## Introduzione allo scheduling

Ai tempi dei sistemi batch l'algoritmo di scheduling era semplice: bastava eseguire il prossimo job sul nastro.

Con i sistemi multiprogrammati, l'algoritmo di scheduling è diventato più complesso in quanto ci sono più utenti in attesa di servizio.

Con l'avvento del PC, la situazione si è evoluta in due modi.

Primo, per la maggior parte del tempo è attivo un solo processo. Un utente che sta

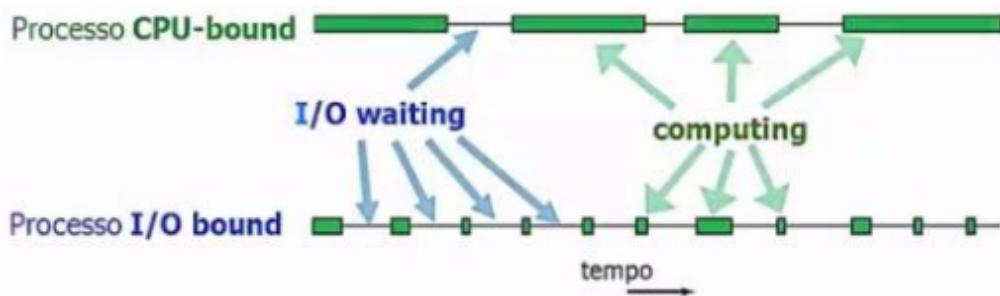
utilizzando un word processor molto probabilmente non sta utilizzando concorrentemente un'altra applicazione. Quindi quando un utente digita un comando nel programma, per lo scheduler è semplice individuare il processo da eseguire, il programma di word processing è l'unico candidato.

Secondo, oggi i computer sono diventati così veloci che raramente la CPU è una risorsa scarsa. La maggior parte dei programmi per PC è limitata dalla velocità di inserimento dati dell'utente.

La situazione è completamente diversa se si considerano i server collegati in rete.

## Comportamento dei processi

Quasi tutti i processi alternano passi di elaborazione a richieste di I/O, come illustrato nella Figura.



Alcuni processi spendono la maggior parte del loro tempo in elaborazione, mentre altri spendono la maggior parte del loro tempo in attesa dell'I/O. La prima situazione è chiamata CPU-bound, la seconda I/O-bound.

I processi limitati nel calcolo (CPU-bound) utilizzano la CPU per lunghi periodi ed hanno attese di I/O poco frequenti, al contrario gli altri usano la CPU (I/O-bound) per brevi periodi ma hanno attese di I/O frequenti.

Con l'incremento delle performance delle CPU i processi tendono ad essere più I/O-bound che CPU-bound.

## Quando effettuare lo scheduling

Un aspetto cruciale dello scheduling è stabilire quando è il momento opportuno per eseguirlo. Le situazioni in cui è necessario schedulare sono varie:

- Quando è creato un nuovo processo si deve decidere se deve essere eseguito il processo genitore o il processo figlio;

- Quando un processo termina, qualche altro processo deve essere scelto nell'insieme dei processi pronti;
- Quando un processo si blocca sull'I/O o su un semaforo bisogna selezionare un altro processo dall'elenco dei processi pronti;
- Quando si verifica un interrupt di I/O, può essere presa una decisione di programmazione (che dipende dal tipo di interrupt).

## Tipi di scheduling

Gli algoritmi di scheduling possono essere suddivisi in due categorie rispetto alla capacità dello scheduling di interrompere l'esecuzione dei processi:

- non preemptive: lo scheduler una volta mandato in esecuzione un processo lo lascia andare finché non si blocca (attesa di I/O o di un altro processo) o rilascia spontaneamente la CPU;
- preemptive: lo scheduler sceglie un processo da eseguire per un tempo massimo stabilito. Se il processo è ancora in esecuzione al termine dell'intervallo di tempo, lo scheduler lo sospende e sceglie un altro processo da eseguire (se disponibile).

In diverse aree applicative sono necessari differenti sistemi operativi e, conseguentemente, specifici algoritmi di scheduling in quanto quello che lo scheduler dovrebbe ottimizzare non è medesima cosa in tutti i sistemi.

Saranno trattati tre ambienti differenti:

- Sistemi batch;
- Sistemi interattivi;
- Sistemi real-time.

I sistemi batch sono ancora utilizzati nel mondo degli affari per l'elaborazione di paghe, esecuzione di accrediti e addebiti su conti corrente ed altre attività periodiche.

Nei sistemi batch non ci sono utenti impazienti in attesa al loro terminale di una risposta rapida a una piccola richiesta. Di conseguenza sono spesso accettabili algoritmi non preemptive.

In un ambiente con utenti interattivi, l'uso della preemptive è essenziale per evitare che un processo monopolizzi la CPU e neghi il servizio agli altri. Anche i server ricadono in

questa categoria, dato che normalmente servono molteplici utenti (remoti), e tutti sempre di fretta.

In sistemi con vincoli real-time, la preemption potrebbe non essere necessaria poiché i processi sanno che devono dare dei risultati in tempi brevi, generalmente fanno il loro lavoro e si bloccano rapidamente.

## Obiettivi degli algoritmi di scheduling

Per progettare un algoritmo di scheduling ci sono obiettivi comuni e altri che dipendono dal tipo di ambiente.

### Tutti i sistemi

- Equità – dare a ogni processo una equa condivisione della CPU
- Applicazione della policy – assicurarsi che la policy dichiarata sia attuata
- Bilanciamento – tenere impegnate tutte le parti del sistema

### Sistemi batch

- Throughput – massimizzare il numero di job per ora
- Tempo di turnaround – ridurre al minimo il tempo da quando un lavoro è sottoposto alla sua fine
- Utilizzo della CPU – mantenere la CPU sempre impegnata

### Sistemi interattivi

- Tempo di risposta – rispondere alle richieste rapidamente
- Adeguatezza – far fronte alle aspettative dell'utente

### Sistemi real-time

- Rispetto delle scadenze – evitare la perdita di dati
- Prevedibilità – evitare il degrado della qualità nei sistemi multimediali

In ogni situazione, l'equità è importante.

Forzare le politiche del sistema è scorretto. Se la policy locale è che i processi di controllo della sicurezza devono essere eseguiti quando lo richiedono, anche se ciò implica che le paghe siano in ritardo di 30 secondi, lo scheduler deve essere certo che questa politica venga sempre rispettata.

Tutte le parti del sistema devono essere impegnate in modo bilanciato. Se la CPU e i dispositivi di I/O sono tenuti sempre impegnati vengono eseguiti più lavori al secondo rispetto ad avere componenti inattivi.

Nei sistemi batch sono molto rilevanti due metriche:

- il throughput, cioè il numero di job completati nell'unità di tempo;

- il tempo di turnaround, cioè il tempo medio di esecuzione dei processi batch.

Anche se è una metrica non appropriata per i sistemi batch, viene spesso utilizzata la percentuale di utilizzo della CPU.

Nei sistemi interattivi è fondamentale il tempo di risposta: il tempo che trascorre dall'invio del comando a quando si ottiene il risultato.

L'adeguatezza della risposta è, invece, la percezione degli utenti rispetto al tempo impiegato per svolgere quel compito (lo scheduler potrebbe assegnare male le priorità di esecuzione e causa un degrado delle risposte).

Per i sistemi in tempo reale è essenziale soddisfare le scadenze ed evitare errori durante l'esecuzione del processo.

## Scheduling nei sistemi batch

Gli algoritmi di scheduling utilizzati nei sistemi batch sono:

- First-come first-served;
- Shortest job first;
- Shortest remaining time next.

Tra questi alcuni sono validi anche nei sistemi interattivi.

### First-come first-served

L'algoritmo first-come first-served è il più semplice algoritmo di scheduling non preemptive.

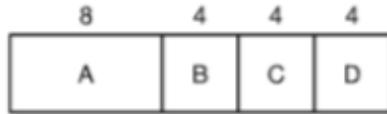
La CPU è assegnata ai processi in ordine di arrivo. C'è un'unica coda dei processi pronti. Questo algoritmo è facile da capire e da programmare.

D'altro canto un sistema con processi eterogenei per tempo di esecuzione potrebbe rallentare i processi veloci in assenza di preemption.

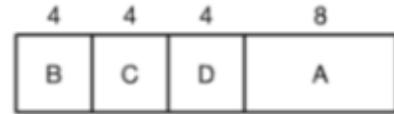
### Shortest job first

Si tratta di un algoritmo senza preemption in cui i tempi di esecuzione dei processi sono noti in anticipo (la predizione dei tempi di esecuzione per job ripetitivi non è difficile).

I job entrano nella coda in ordine, ma lo scheduler sceglie quello che termina prima.



(a)



(b)

Vi troviamo i lavori A, B, C e D con un tempo di esecuzione di 8, 4, 4 e 4 minuti, rispettivamente. Eseguendoli in ordine il tempo di turnaround sarebbe di 8 minuti per A, 12 minuti per B, 16 per C e 20 per D, con una media di 14 minuti.

Consideriamo questi lavori usando l'algoritmo shortest job first, come illustrato nella Figura (b). I tempi di turnaround diventano 4, 8, 12 e 20 minuti, con una media di 11 minuti.

La formula di turnaround medio è  $(4A+3B+2C+D)/4$ .

Si noti che l'algoritmo in questione è ottimale solo nel caso in cui tutti i lavori siano disponibili contemporaneamente.

## **Shortest remaining time next**

Questo algoritmo è una versione preemptive dell'algoritmo shortest job first.

Lo scheduler in questo caso sceglie sempre il processo il cui tempo che manca alla fine dell'esecuzione è il più breve. Anche in questo caso il tempo di esecuzione deve essere conosciuto in anticipo. All'arrivo di un nuovo lavoro il suo tempo totale è confrontato al tempo restante dei processi attuali. Se il nuovo lavoro ha bisogno di minor tempo del processo attuale per terminare, il processo attuale viene sospeso ed è avviato il nuovo lavoro.

Questo schema permette ai nuovi lavori brevi di ottenere un buon servizio.

## **Scheduling nei sistemi interattivi**

Nei sistemi interattivi possono essere utilizzati i seguenti algoritmi:

### **Scheduling round-robin**

Il round-robin è uno degli algoritmi più vecchi, semplice, equilibrato e diffuso.

Ad ogni processo è assegnato un <quantum> di tempo di esecuzione in CPU, allo scadere del quale viene interrotto e si passa, con modalità circolare e parietaria, al successivo processo. La scelta del <quantum> è un fattore critico: un valore troppo

breve provoca troppe interruzioni (overhead), mentre uno troppo alto provoca una coda di attesa eccessiva

## Scheduling a priorità

Nello scheduling round-robin c'è l'assunzione implicita che tutti i processi sono ugualmente importanti. In molti contesti reali questa è un'ipotesi troppo restrittiva.

Ciò porta allo scheduling a priorità. L'idea di base è assegnare a ciascun processo una priorità e il processo eseguibile con la priorità più alta è quello cui è consentita l'esecuzione.

Per evitare che i processi ad alta priorità siano girino per tempi indeterminati, lo scheduler può abbassare la priorità del processo attualmente in esecuzione a ogni scatto del clock o assegnare un <quantum> di tempo massimo di CPU.

Le priorità possono essere assegnate ai processi staticamente o dinamicamente.

È conveniente dividere i processi per classi di priorità utilizzando lo scheduling con priorità per le classi e quello round-robin all'interno di ciascuna classe.

## Shortest process next

Poiché l'algoritmo shortest job first produce sempre il minor tempo medio di risposta sui sistemi batch, sarebbe apprezzabile che si potesse usare anche sui sistemi interattivi.

Il problema sta nel calcolare il tempo di esecuzione che non è sempre lo stesso.

Un possibile approccio è quello di misurare i tempi di esecuzione e di utilizzarli per le stime successive attraverso una media pesata tenendo conto che le misure più recenti sono più attendibili (aging).

iterazione	0	1	2	3
Tempo di esecuzione	$T_0$	$T_1$	$T_2$	$T_3$
Calcolo	$T_0$	$T_0/2+T_1/2$	$T_0/4+T_1/4+T_2/2$	$T_0/8+T_1/8+T_2/4+T_3/2$
stima di	$T_1$	$T_2$	$T_3$	$T_4$

## Scheduling garantito

Un approccio completamente diverso per lo scheduling è quello di fare promesse reali agli utenti riguardo alle prestazioni e poi mantenerle.

“Se ci sono  $n$  utenti collegati mentre state lavorando, avrete all’incirca  $1/n$  del tempo della CPU”.

Per mantenere questa promessa, il sistema deve tener traccia di quanta CPU ogni processo ha avuto dal momento della sua creazione e calcola il rapporto *avuto/promessa*. Se un qualsiasi processo ha un valore più basso di tale rapporto è il successivo candidato ad essere messo in esecuzione, finché il suo rapporto è il più basso tra tutti quelli pronti.

## Scheduling a lotteria

Fare promesse agli utenti e poi mantenerle è una bella idea, ma difficile da realizzare.

Si può assegnare ai processi un biglietto della lotteria per le diverse risorse del sistema, come il tempo di CPU. Ogni volta che deve essere presa una decisione di scheduling si pesca un biglietto della lotteria e il processo che ha quel biglietto si aggiudica la risorsa.

Differentemente dallo scheduling con le priorità in cui è difficile dire cosa significa avere una certa priorità, in questo algoritmo se un processo possiede 20 biglietti su 100 ha il 20% delle probabilità di ottenere la risorsa (che al crescere del tempo approssima la frequenza).

Lo scheduling a lotteria ha due proprietà interessanti:

è reattivo: anche i processi neonati possono vincere la lotteria, fin dalle prime scelte dello scheduler. I processi che cooperano, se lo desiderano, possono scambiarsi i biglietti che detengono per alterare le priorità di esecuzione.

Può essere utilizzato per risolvere problemi difficili da gestire con altri metodi come ad esempio, si consideri lo streaming video, dove sono necessarie differenti velocità di trasferimento (esprese in frame al secondo) a seconda del processo. Assegnando tanti biglietti quanti sono i frame automaticamente la CPU approssima le proporzioni desiderate.

## Scheduling fair-share

Finora abbiamo supposto che ogni processo fosse schedulato per proprio conto, senza considerare a chi appartenesse.

Di conseguenza, se un utente X avvia 4 processi (A, B, C e D), e l’utente Y ne avvia uno solo (E), l’utente X si prenderà l’80% della CPU e l’utente solo il 20%.

Per evitare questo, prima di schedularlo, alcuni sistemi prendono in considerazione chi possiede un processo e assegnano una porzione di CPU in base agli utenti collegati (quindi non in base ai processi).

50% della CPU a ogni utente avremo: A E B E C E D E A E B E C E D E ...

75% della CPU all'utente X e il 25% all'utente Y avremo: A B E C D E A B E C D E ...

## Scheduling nei sistemi real-time

In un sistema real-time il tempo gioca un ruolo essenziale. I sistemi real-time sono suddivisi in due categorie:

- Hard real-time: le scadenze devono essere sempre rispettate;
- Soft real-time: il mancato rispetto di una scadenza non è auspicabile, ma comunque tollerabile.

Gli eventi in un sistema real-time possono essere classificati come:

- Periodici, quando si verificano a intervalli di tempo regolari;
- Non periodici, quando si verificano in modo imprevedibile.

Un sistema soft real-time con eventi periodici è sostenibile se riesce a far fronte agli eventi stessi, ovvero se riesce a trattare un evento prima che ne arrivi un altro.

Se ci sono  $m$  eventi periodici e ogni evento avviene con frequenza pari a  $1/P_i$ , supponendo che ciascun evento richieda  $C_i$  secondi di tempo CPU per gestirlo, allora il sistema è in grado di reggere il carico se e solo se:  $\sum_{i=0}^m C_i / P_i < 1$ .

Un sistema real-time che soddisfa questo criterio è detto schedulabile.

## Scheduling dinamico e statico

Gli algoritmi di scheduling per i sistemi real-time possono essere statici o dinamici.

Gli algoritmi statici prendono la decisione di scheduling prima che il sistema inizi a funzionare. Questa tecnica è applicabile solo quando ci sono informazioni disponibili in anticipo riguardo il lavoro da svolgere e le scadenze da rispettare.

Gli algoritmi dinamici prendono le decisioni di scheduling in fase di esecuzione. Questa tecnica non necessita di conoscere in anticipo alcuna informazione sul compito da svolgere e sui tempi.

## La policy in contrapposizione al meccanismo

Finora abbiamo presupposto che tutti i processi del sistema appartengano a utenti differenti e che siano di conseguenza in competizione per la CPU. Talvolta, accade che un processo abbia molti figli eseguiti sotto il suo controllo (es. un dbms può avere molti processi figli).

Il processo genitore conosce la priorità da assegnare ai propri figli. Sfortunatamente nessuno degli scheduler finora presentati accetta alcun input dai processi utente riguardo alle decisioni di scheduling. Di conseguenza, lo scheduler raramente prende la decisione migliore.

La soluzione a questo problema sta nel separare il meccanismo di scheduling dalla politica di scheduling. L'algoritmo di scheduling ha parametri che possono essere riempiti dai processi utente.

## Scheduling a thread

Quando i processi hanno più thread, possiamo definire due livelli di parallelismo: sui processi e sui thread. Lo scheduling in questi ambienti si differenzia a seconda che siano supportati thread utente o a livello kernel (o entrambi).

Consideriamo i thread utente. Dato che il kernel non è a conoscenza dell'esistenza dei thread perché conosce solo i processi che li contengono. Supponiamo che:

Il processo A ha tre thread: A1, A2, A3.

Il processo B ha tre thread: B1, B2, B3.

Lo scheduler sceglie il processo A e dà al processo il controllo per il suo quantum di tempo. Ora lo scheduler di thread interno ad A decide quale thread eseguire, diciamo A1. Poiché per questi thread non ci sono interrupt esso può continuare a funzionare quanto vuole (max quantum di A). Se utilizza l'intero quantum del processo, il kernel seleziona un altro processo da eseguire.

Quando verrà rieseguito il processo A, il thread A1 riprenderà l'esecuzione. Continuerà a consumare tutto il tempo di A finché non è finito. Il suo comportamento non influisce su altri processi.

La sequenza di attivazione A1 B1 A2 B2 A3 B3 non è possibile con questi parametri e i thread utente.

Nel caso dei thread a livello kernel. Il kernel conosce i thread e ne prende uno per l'esecuzione. Al thread è assegnato un quantum di tempo e viene forzatamente sospeso

se supera quel valore di tempo di esecuzione.

La sequenza di attivazione A1 B1 A2 B2 A3 B3 è ora possibile.

Vantaggi e svantaggi dei thread a livello utente:

- Lo scambio di esecuzione tra thread è veloce e sono sufficienti poche istruzioni macchina;
- Se un thread va in blocco su una operazione di I/O si deve sospendere l'intero processo;
- Si può utilizzare lo scheduler di thread specifico dell'applicazione (es. web server) e che abiliti una strategia di attivazione migliore del kernel, poiché conosce ciò che fanno i vari thread.

Vantaggi e svantaggi dei thread a livello kernel:

- Lo scambio di contesto è oneroso di molti ordini di grandezza rispetto all'altro caso, poiché occorre invalidare la mappa di memoria e la cache;
- Il kernel può decidere quale thread mandare in esecuzione tenendo conto anche dell'overhead causato da un eventuale cambio di contesto;
- Se un thread va in blocco non sospende l'intero processo.

## 3. Gestione della memoria

La memoria principale RAM è una risorsa che va gestita attentamente. Il concetto di gerarchia di memoria:

I computer attuali possono avere pochi MB di cache volatile, costosa e molto veloce, qualche GB di memoria principale, di media velocità e prezzo e qualche TB di storage su disco non volatile, economico e lento. La parte del SO che gestisce la memoria è chiamata gestore della memoria. Il suo compito è gestire con efficienza la memoria: tener traccia di quali parti di memoria sono in uso, allocare memoria ai processi secondo le loro necessità e liberarla a lavoro terminato.

### 3.a Sistemi di base per la gestione della memoria

Il modo più semplice per gestire la memoria è senza forme di astrazione. Ogni programma vede semplicemente la memoria fisica, così com'è (numero di celle e dimensione del contenuto).

I primi mainframe (prima del 1960), minicomputer (prima del 1970) e PC (prima del 1980) non avevano astrazione della memoria.

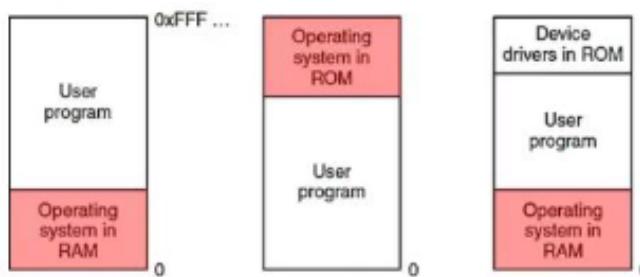
Per eseguire un programma è necessario caricarlo nella memoria principale. Più programmi possono stare in memoria ma è necessario che non si sovrappongono.

Anche con il solo modello di memoria fisica esistono diverse possibilità.

Il SO in fondo della memoria nella RAM (mainframe e minicomputer)

SO in testa alla memoria nella ROM (palmare e sistemi embedded)

SO in fondo alla memoria nella RAM, driver delle periferiche in testa alla memoria nella ROM (PC con MS-DOS).



Il primo e il terzo modello hanno lo svantaggio che un errore nel programma utente può eliminare il SO. La parte del sistema operativo nella ROM è chiamata BIOS (Basic Input Output System).

## Monoprogrammazione senza swapping

Anche senza l'astrazione di memoria, è possibile fornire all'utente l'idea di avere in esecuzione più programmi contemporaneamente.

Il sistema operativo salva l'intero contenuto della memoria in un file sul disco, carica ed esegue il programma successivo. Finché in memoria vi è un solo programma per volta, non vi sono conflitti. Questo concetto è chiamato swapping.

Con l'aggiunta di hardware speciale, è possibile eseguire più programmi contemporaneamente, anche senza swapping.

La mancanza di una astrazione della memoria è tuttora comune nei dispositivi quali radio, lavatrici, ecc... Il software indirizza la memoria in modo assoluto e funziona solo perché i programmi sono noti in anticipo e gli utenti non sono liberi di eseguire il proprio software su questi.

## Lo spazio di indirizzamento

Esporre la memoria fisica ai processi presenta delle criticità.

Se i programmi utente possono indirizzare ogni byte della memoria, allora possono facilmente sovrascrivere il sistema operativo;

Con questo modello è difficile avere un'organizzazione rigida della memoria che consenta a più programmi di essere in esecuzione contemporaneamente.

Per permettere a molteplici applicazioni di risiedere in memoria contemporaneamente senza interferire l'un l'altra devono essere risolti due problemi: la protezione e il riposizionamento.

Una soluzione primitiva è stata usata sull'IBM 360: etichettare grossi pezzi di memoria con una chiave di protezione e confrontare la chiave del processo in esecuzione con quella di ogni parola di memoria prelevata. Questo approccio tuttavia non risolve il secondo problema, sebbene si possano riposizionare i programmi quando vengono caricati, ma si tratta di una soluzione lenta e complicata.

Una soluzione migliore è inventare una nuova astrazione per la memoria: lo spazio degli indirizzi. Così come il concetto di processo crea una specie di CPU astratta per eseguire i programmi, lo spazio degli indirizzi crea una specie di memoria astratta per farci vivere i programmi.

Uno spazio degli indirizzi è l'insieme degli indirizzi che un processo può usare per accedere alla memoria. Ogni processo ha il suo spazio di indirizzamento, indipendente da quello appartenente ad altri processi.

Il concetto di spazio degli indirizzi è molto generale ed è appropriato in molti contesti (numeri di telefono, porte I/O, indirizzi IPv4, domini internet).

Più difficile è come dare a ogni programma il suo spazio degli indirizzi: l'indirizzo 28 in un programma significa una locazione fisica diversa dall'indirizzo 28 in un altro programma.

## Registri base e registri limite

Questa soluzione si basa su una versione particolarmente semplice della rilocazione dinamica. Quello che fa è mappare lo spazio degli indirizzi di ogni processo su di una parte diversa di memoria fisica in un modo semplice.

La soluzione classica è di dotare ogni CPU con due registri hardware speciali, solitamente chiamati registro base e registro limite. I programmi sono caricati in posizioni

di memoria consecutive dovunque vi sia spazio e senza riposizionamento durante il caricamento.

Al momento dell'esecuzione di un processo, il registro base è caricato con l'indirizzo fisico dove comincia il suo programma in memoria e il registro limite è caricato con la lunghezza del programma.

Ogni volta che un processo consulta la memoria prima di inviare l'indirizzo sul bus di memoria l'hardware della CPU aggiunge automaticamente il valore di base all'indirizzo verificando se l'indirizzo non eccede il limite di memoria riservato al processo, nel caso di errore l'accesso viene vietato.

In molte implementazioni i registri limite e base sono protetti (solo il sistema operativo può modificarli).

Uno svantaggio della rilocazione per mezzo dei registri limite e base è la necessità di eseguire una somma e un confronto con ogni riferimento della memoria.

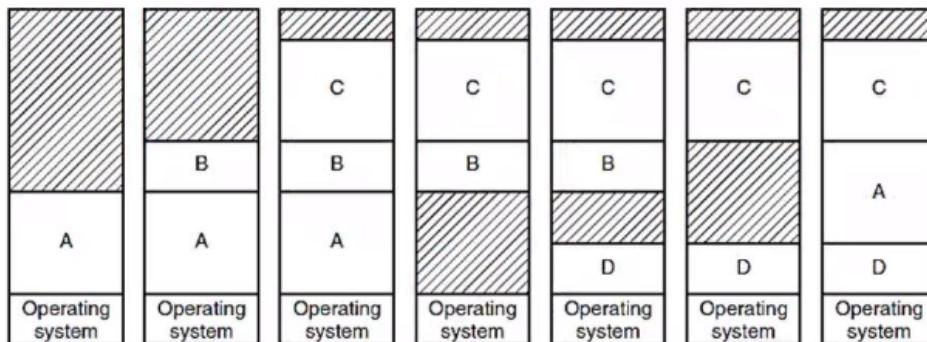
I confronti possono avvenire velocemente, ma le somme sono lente a causa del tempo di propagazione, a meno che non siano usati circuiti sommatori speciali.

### **3.b Swapping**

Se la memoria fisica del computer è abbastanza ampia da contenere tutti i processi, gli schemi descritti finora più o meno funzioneranno. Nella realtà tuttavia la RAM non riesce a contenere tutti i processi e per gestire la memoria sono possibili due approcci:

Swapping: si prende un processo e lo si esegue per un certo tempo, quando ha terminato si salva su disco. I processi inattivi sono memorizzati su disco, così non occupano memoria mentre non sono in esecuzione.

Memoria virtuale: consente ai programmi di essere eseguiti anche quando sono solo parzialmente nella memoria principale.



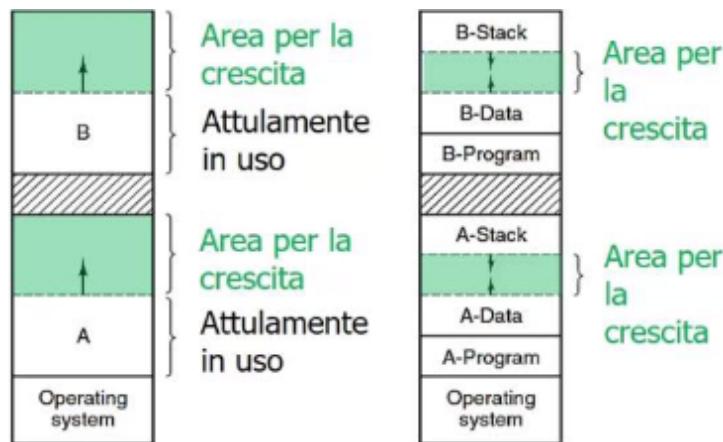
Inizialmente il processo A è in memoria, vengono poi creati o viene fatto lo swapping dal disco dei processi B e C. Nella Figura (d) viene fatto lo swapping (salvato) su disco di A. Quindi arriva D ed esce B. Alla fine A viene ricaricato. Poiché ora A è in una posizione diversa, i suoi indirizzi devono essere rilocati dal software o dall'hardware durante l'esecuzione del programma.

Lo swapping crea molteplici spazi vuoti nella memoria, è possibile combinarli tutti in un unico spazio vuoto spostando tutti i processi il più in basso possibile. Questa tecnica è conosciuta come memory compaction. Di solito non viene fatta perché richiede molto tempo di CPU.

Un punto da evidenziare è quanta memoria dovrebbe essere allocata per un processo quando viene creato o quando ne viene fatto lo swapping dal disco:

Se i processi hanno una dimensione fissa, l'allocazione è semplice: il sistema operativo conosce esattamente di quanto ha bisogno, alloca ciò che è necessario.

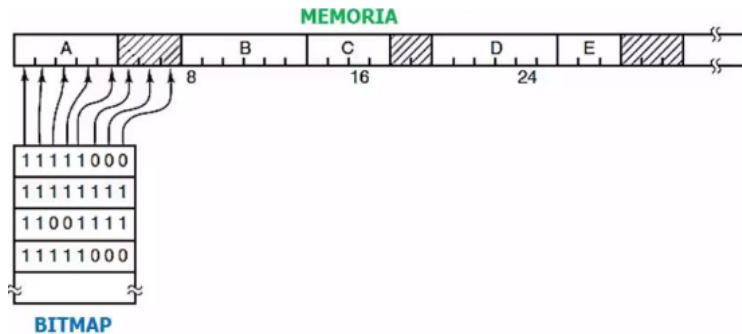
Se i processi crescono di dimensione (come normalmente fanno), allocando dinamicamente dalla memoria heap o lo stack, se vi è spazio adiacente al processo, esso può essere allocato e il processo può crescere in tale spazio. D'altra parte, se il processo ha un altro processo adiacente, bisognerà trovare uno spazio di memoria sufficientemente largo ad ospitarlo. Una soluzione è allocare un po' di memoria extra per prevenire la sua crescita.



Quando la memoria è assegnata dinamicamente, il sistema operativo deve gestirla. Ci sono due modalità per tener traccia dell'utilizzo della memoria: bitmap e liste.

## Gestione della memoria con bitmap

La memoria, con una bitmap, è divisa in unità di allocazione piccole come qualche parola fino a diversi KB. La bitmap ha bit quante le unità di allocazione della memoria, è 0 se l'unità è libera e 1 se è utilizzata.



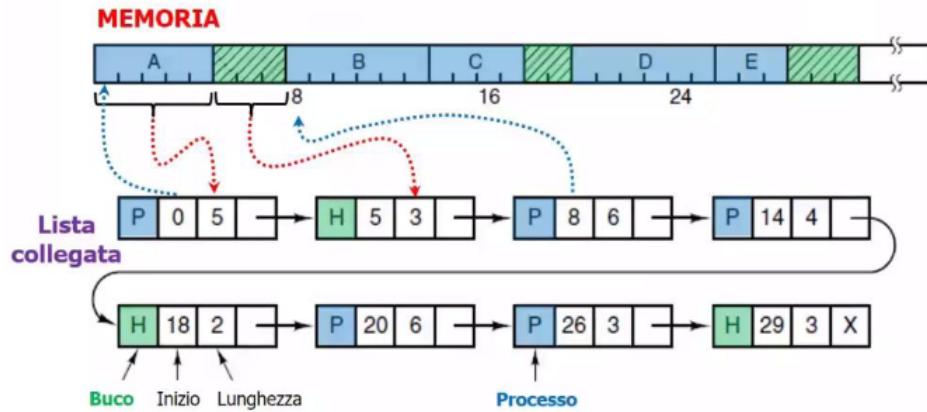
In questo sistema è cruciale la scelta della dimensione dell'unità di allocazione: più è piccola la dimensione dell'unità di allocazione, maggiore è la dimensione della bitmap; più è grande la dimensione dell'unità di allocazione, minore è la dimensione della bitmap ma aumenta di sprecare memoria nell'unità se la dimensione del processo non è esattamente multiplo di quella dell'unità di allocazione.

Il problema principale di questa soluzione è che quando dobbiamo portare in memoria un processo di  $k$  unità, il gestore deve ricavare nella bitmap  $k$  bit liberi consecutivi (la ricerca sequenziale è un'operazione lenta).

Un'unità di allocazione di 4 byte richiede solo 1 bit della mappa, invece una memoria di  $32^n$  bit che utilizza 32 bit per unità di allocazione richiederà  $32^{n-1}$  bit per la mappa.

## Gestione della memoria con liste collegate

Un altro sistema per gestire la memoria è attraverso una lista collegata che memorizza le posizioni dei segmenti occupati dai processi (P) e quelle dei segmenti liberi (H).



In questo esempio la lista dei segmenti è in ordine di indirizzo. Fare questo ordinamento ha il vantaggio che quando il processo finisce o ne viene fatto lo swapping, l'aggiornamento della lista è molto semplice.

Un processo che finisce ha normalmente due vicini. Questi possono essere sia processi sia spazi vuoti, il che porta alle quattro combinazioni:



Quando viene creato un nuovo processo o ne viene fatto lo swapping sul disco possono essere utilizzati vari algoritmi per trovargli una posizione in memoria. Si assume che il gestore della memoria conosca quanta memoria occorre allocare. Troviamo i seguenti algoritmi:

- First fit (il primo va bene): è il più semplice, il gestore della memoria scorre la lista finché non trova uno spazio vuoto abbastanza grande per il processo, se lo spazio trovato è più grande viene diviso in spazio per processo e spazio libero, è veloce;
- Next fit (il prossimo va bene): lavora allo stesso modo del precedente, ma tiene traccia di ogni posto dove ha trovato uno spazio adatto. La volta dopo che è interpellato, cerca nella lista a partire dal punto dove era rimasta l'ultima volta, non riparte da capo, ha performance leggermente peggiori rispetto al first fit.

- Best fit: cerca all'interno della lista, dall'inizio alla fine, prendendosi lo spazio più piccolo ma che sia comunque sufficiente per il processo, così da minimizzare lo spreco di spazio, è più lento degli altri due perché deve scorrere tutta la lista ogni volta che viene chiamato, tende a riempire la memoria con piccoli spazi inutilizzabili e quindi spreca più memoria;
- Worst fit: come il precedente però si prende lo spazio disponibile più grande in modo da avere un nuovo spazio vuoto sarà il più grande possibile, non è un buon algoritmo;
- Quick fit: mantiene liste divise per alcune delle più comuni dimensioni richieste, una per gli spazi da 4KB, una per quelli da 8KB etc... Ha lo svantaggio di tutti gli schemi ordinati per dimensione, quando un processo termina o viene inviato su disco, trovare i suoi vicini per unire gli spazio vuoti è un processo dispendioso, se non sono fusi insieme gli spazi liberi la memoria si frammenta rapidamente in un gran numero di piccoli buchi in cui non entra nessun processo;

I primi 4 algoritmi possono essere accelerati mantenendo elenchi separati per i processi e spazi vuoti. In questo modo, tutti dedicano la loro energia nella ricerca di spazi vuoti, ma il prezzo da pagare è in termini di complessità: quando si deve deallocare la memoria, il segmento del processo deve essere rimosso dall'elenco dei processi e deve essere inserito nella lista degli spazi vuoti.

### **3.c Memoria virtuale**

Mentre da un lato i registri base e limite possono essere utilizzati per creare l'astrazione dello spazio di indirizzamento, dall'altro sorge un nuovo problema: la gestione del software bloat (bloatware).

La dimensione del software aumenta molto più velocemente della dimensione della memoria.

Sorge la necessità di eseguire programmi che eccedono in dimensioni rispetto alla memoria, oltre al conseguente bisogno di disporre di sistemi in grado di supportare molteplici programmi eseguiti contemporaneamente, che stiano in memoria, ma che complessivamente siano più grandi della stessa. Il problema dei programmi più grandi della memoria è presente fin dalle origini dell'informatica.

Lo swapping non è un'opzione interessante, poiché i dispositivi di I/O sono troppo lenti rispetto alla velocità della memoria.

Negli anni '60 fu adottata una soluzione che divideva i programmi in pezzi piccoli, chiamati Overlay. All'avvio di un programma veniva caricato in memoria il gestore degli overlay, che caricava e avviava subito l'overlay 0. Al termine, veniva indicato al gestore degli overlay di caricare l'overlay 1 sopra l'overlay 0 in memoria (se c'era spazio per farlo) oppure sovrascrivendo l'overlay 0 (in mancanza di spazio). Gli overlay erano tenuti sul disco e portati dentro e fuori la memoria all'occasione dal gestore degli overlay.

Il lavoro di swapping era fatto dal sistema operativo, ma la suddivisione del programma in pezzi era fatto manualmente dal programmatore. Suddividere grandi programmi in piccoli pezzi modulari richiedeva molto tempo, era noioso e suscettibile di errori. Pochi programmatori erano in grado di farlo.

Per risolvere, si è cercato di delegare questo compito al computer. Il metodo che fu escogitato è noto come memoria virtuale. L'idea alla base della memoria virtuale è che ogni programma ha il proprio spazio degli indirizzi personale, suddiviso in pezzi chiamati pagine. Ogni pagina è un intervallo di indirizzi contigui ed è mappato sulla memoria fisica, ma non tutte le pagine devono stare nella memoria fisica per eseguire il programma.

Quando il programma fa riferimento a una parte del suo spazio degli indirizzi che è: Nella memoria fisica, l'hardware esegue il necessario mappaggio diretto (on-the-fly); Non è in memoria fisica, il sistema operativo avvisa che manca il pezzo, va a prendere il pezzo mancante e rieseguire l'istruzione fallita.

La memoria virtuale è una generalizzazione dell'idea del registro base e limite. Con la memoria virtuale, invece di avere rilocazione separata per i segmenti di testo e dati, l'intero spazio di indirizzamento può essere rimappato sulla memoria fisica in pagine di stessa dimensione.

La memoria virtuale funziona bene in sistemi con multiprogrammazione, con bit e pezzi di molti programmi contemporaneamente in memoria. Mentre un programma è in attesa che venga caricata una pagina richiesta, la CPU può essere assegnata a un altro processo.

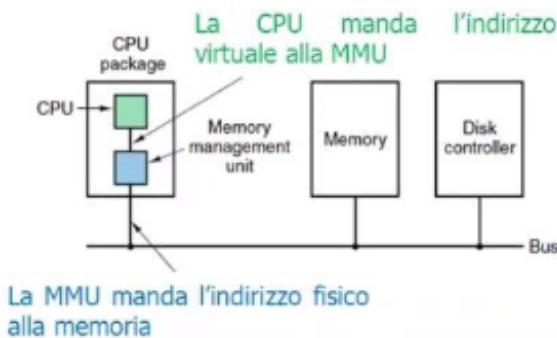
## Paginazione

La maggior parte dei sistemi di memoria virtuale usa una tecnica chiamata paginazione o paging. Su qualsiasi computer i programmi referenziano un insieme di indirizzi di memoria.

Gli indirizzi possono essere generati usando l'indicizzazione, i registri base, i registri segmento o in altri modi.

Questi indirizzi generati dal programma sono chiamati indirizzi virtuali e formano lo spazio virtuale degli indirizzi. Sui computer senza memoria virtuale, l'indirizzo virtuale è messo direttamente sul bus indirizzi e provoca la lettura o la scrittura della parola della memoria fisica con lo stesso indirizzo (la parola di memoria fisica coincide con quella virtuale).

Quando è usata la memoria virtuale, gli indirizzi virtuali non vanno direttamente al bus indirizzi, ma a una MMU (Memory Management Unit), un bit di presenza/assenza tiene traccia di quali pagine sono presenti fisicamente in memoria.



Lo spazio degli indirizzi virtuali è suddiviso in unità di dimensione fissa, chiamate pagine. Le unità corrispondenti nella memoria fisica sono chiamate frame o page frame. Le pagine e i frame sono generalmente della stessa dimensione. I trasferimenti tra la RAM e il disco sono sempre di pagine intere.

Nella figura il frame segnato 0K-4K significa che gli indirizzi fisici o virtuali in quella pagina vanno da 0 a 4095. L'intervallo 4K-8K si riferisce agli indirizzi da 4096 al 8191 e così via.



Quando una pagina virtuale non è nella memoria, la MMU si accorge che il riferimento nell'istruzione non è in memoria e causa un errore di pagina (page fault) al sistema operativo.

Il sistema operativo preleva il frame meno utilizzato e ne scrive il suo contenuto sul disco (se è stato modificato). Prende poi la pagina appena referenziata e la mette nel frame appena liberato. Cambia la mappa all'interno della MMU e riavvia l'istruzione che era in trap.

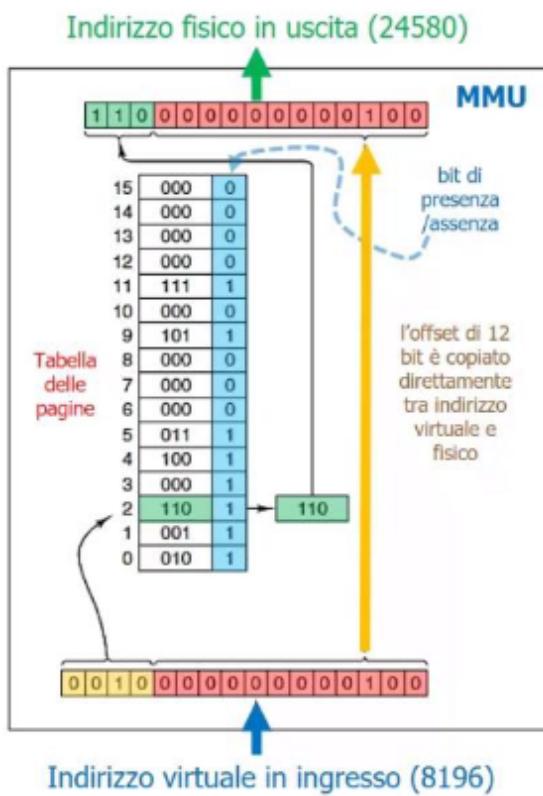
All'interno della MMU c'è una tabella delle pagine che permette di traslare un indirizzo virtuale in uno fisico.

Il numero di pagina è usato come indice della tabella che porta al numero di frame corrispondente alla pagina virtuale. Se il bit presente/assente è 0, avviene un trap al sistema operativo; se il bit è 1, il numero di frame trovato nella tabella delle pagine viene copiato nei tre bit più significativi del registro di output, insieme all'offset di 12 bit che è copiato senza modifiche dall'indirizzo virtuale in arrivo. Insieme formano un indirizzo fisico di 15 bit. Il registro di output è poi messo sul bus di memoria come indirizzo fisico di memoria.

## **Tabelle delle pagine**

L'indirizzo virtuale è diviso in due parti, una parte è utilizzata come indice nella tabella delle pagine (i bit più significativi) e una come offset (i bit meno significativi).

Lo scopo della tabella delle pagine è di mappare le pagine virtuali sui frame delle pagine.



## Struttura di una voce della tabella delle pagine

Anche se la struttura fisica del record dipende dalla macchina, il tipo di informazioni presenti è generalmente lo stesso.

N° Frame	Presenza / Assenza	Protezione	Pagina Modificata	Pagina Referenziata	Disabilità caching
----------	--------------------	------------	-------------------	---------------------	--------------------

La lunghezza della voce usuale è di 32 bit e contiene:

- Il numero del frame (page frame);
- Bit presente/assente: se questo bit è 1, la voce è valida e può essere utilizzata; se è 0, la pagina virtuale è in memoria (page fault) e va caricata;
- Bit di protezione: specificano quali tipi di accesso sono consentiti (lettura, scrittura, esecuzione);
- Bit pagina modificata: tiene traccia della modifica della pagina; è asserito se la pagina è stata modificata, prima di sovrascriverla in memoria va scritta sul disco, in caso contrario quella sul disco è ancora valida;
- Bit pagina referenziata: tiene traccia dell'uso della pagina (lettura o scrittura); serve per

aiutare il sistema operativo a scegliere quale pagina scaricare quando si verifica un page fault;

- Bit caching disabilitata: consente di disabilitare la cache per la pagina.

In qualsiasi sistema di paging, devono essere analizzati due aspetti importanti:

La conversione dell'indirizzo virtuale in quello fisica deve essere veloce.

Più è ampio lo spazio di indirizzamento virtuale maggiore è la dimensione della tabella delle pagine.

Il più semplice approccio è disporre di una tabella delle pagine che contiene una serie di registri hardware indicizzati per pagina:

è un approccio semplice che non richiede riferimenti di memoria durante la mappatura; è costoso se la tabella delle pagine è grande e può avere basse prestazioni poiché richiede il caricamento della intera tabella delle pagine ad ogni cambio di contesto.

Se invece la tabelle delle pagine è interamente in memoria principale:

L'hardware ha bisogni di un registro (che punta alla tabella delle pagine) e quindi è molto flessibile.

L'accesso alla memoria rallenta le prestazioni.

## Translation lookaside buffer

La tabella delle pagine è normalmente in memoria. Osservando che la maggior parte dei programmi tende a fare un gran numero di riferimenti a un piccolo numero di pagine e non il contrario, solo una piccola parte delle righe della tabella delle pagine viene letta frequentemente; il resto è poco utilizzato.

La soluzione escogitata è equipaggiare i computer di un piccolo dispositivo hardware per mappare gli indirizzi virtuali sugli indirizzi fisici senza passare dalla tabella delle pagine. Questo dispositivo si chiama TLB (translation lookaside buffer) o qualche volta anche memoria associativa.

Si trova abitualmente all'interno della MMU e consiste di un numero ridotto di righe (8 o al massimo 64). Ogni riga contiene informazioni riguardo una pagina, tra cui il numero di pagina virtuale, un bit impostato quando la pagina viene modificata, i permessi e il frame fisico in cui si trova la pagina. Questi campi hanno una corrispondenza uno-a-uno con i campi nella tabella delle pagine, eccetto che per il numero di pagina virtuale, che non è necessario nella tabella delle pagine. Un altro bit indica se la riga è uso o no.

Analizziamo adesso come funziona il TLB. Quando un indirizzo virtuale viene presentato alla MMU per la traduzione, l'hardware prima guarda se il suo numero di pagina virtuale è presente nel TLB confrontandolo simultaneamente (cioè in parallelo) con tutte le righe. Se trova un riscontro valido il frame è preso direttamente dal TLB, senza andare alla tabella delle pagine; altrimenti la MMU rileva la TLB miss (fallimento di TLB) e fa una normale ricerca sulla tabella delle pagine. Quindi sfratta una delle righe dal TLB e la rimpiazza con la riga della tabella delle pagine appena cercata. In questo modo se quella pagina è riusata a breve, la seconda volta si avrà una TLB hit (successo di TLB) invece che una TLB miss.

Quando una riga è eliminata dal TLB, il bit modificato è copiato di nuovo nella riga della tabella delle pagine nella memoria.

## Gestione software del TLB

Le tabelle delle pagine e la TLB possono essere gestiti dall'hardware (le uniche trap al SO avvengono quando una pagina non è in memoria) oppure dal software.

Molte macchine moderne eseguono quasi tutta questa gestione delle pagine tramite software. Le righe del TLB sono caricate dal sistema operativo. Quando si verifica una TLB miss, l'MMU passa il problema al sistema operativo con un errore TLB. Il SO deve trovare la pagina nella tabella delle pagine, sostituire la riga nella TLB, e rieseguire l'istruzione che ha generato la trap. Tutto deve avvenire in poche istruzioni poiché le TLB miss sono molto più frequenti che le page fault.

Se la TBL è grande (64 righe) e il software efficiente (pochi TBL miss) questo riduce la complessità della MMU e libera lo spazio sulla CPU. Sono state ideate varie soluzioni per ridurre le TBL miss a volte il SO può caricare in anticipo la TLB con delle pagine che è probabile che nel breve possano essere utilizzate.

Il modo consueto di trattare una TLB miss, sia hardware sia software, è andare alla tabella delle pagine e localizzare la pagina referenziata. Se le pagine che contengono la tabella delle pagine è sempre nella TLB questa ricerca è più veloce. Un modo è di riservare uno spazio di posizioni fisse nella TBL in modo che queste righe non possano essere rimosse quando accade un TBL miss.

Nella gestione TLB via software è fondamentale capire la differenza fra due tipi di miss:

Soft miss avviene quando la pagina di riferimento non è nel TLB ma è nella memoria. Tutto ciò di cui si ha bisogno in questo caso è che il TLB sia aggiornato. Nessun accesso al disco è necessario, e quindi può essere completata in pochi nanosecondi.

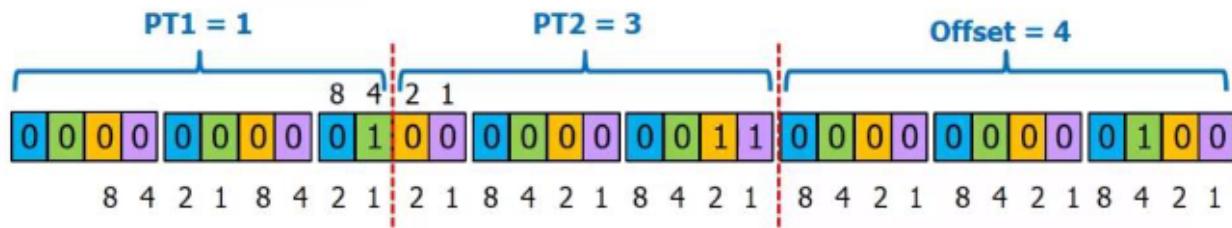
Hard miss avviene quando la pagina stessa non è in memoria (e ovviamente nemmeno nel TLB). Per prendere la pagina serve un accesso al disco, il che comporta parecchi millisecondi. Una hard miss è milioni di volte più lenta di una soft miss.

## Tabelle delle pagine

I TLB sono usati per velocizzare la traduzione dell'indirizzo virtuale nell'indirizzo fisico. L'altro problema da affrontare è come gestire spazi degli indirizzi virtuali molto grandi. Esistono due approcci.

### Tabelle delle pagine multilivello

Consideriamo un indirizzo virtuale a 32 bit partizionato in un campo PT1 a 10 bit, un campo PT2 a 10 bit e un campo Offset a 12 bit. Poiché gli offset sono 12 bit, le pagine sono 4 KB e ve n'è un totale di  $2^{20}$ .

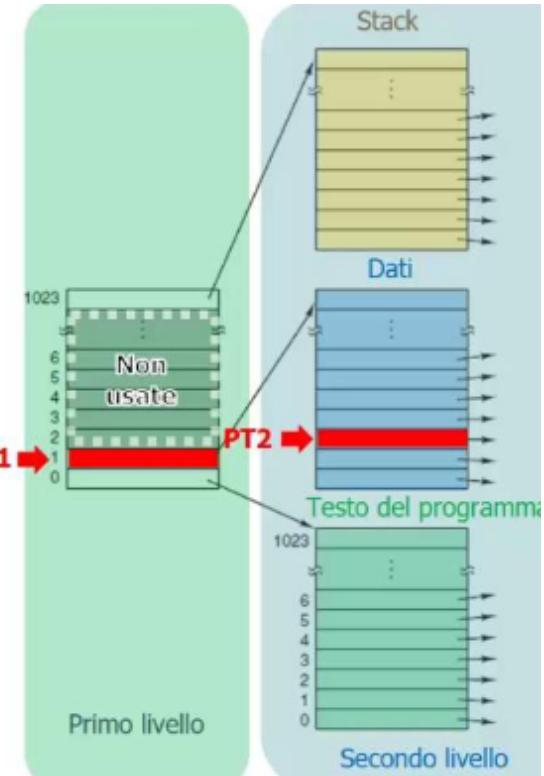


La tabella delle pagine è organizzata in due livelli: PT1 e PT2. Quando la MMU deve tradurre un indirizzo virtuale, utilizza i primi bit più significativi. Nella figura è riportato l'indirizzo virtuale 0x00403004.

Il segreto del metodo della tabella delle pagine multilivello sta nell'evitare di mantenere tutte le tabelle delle pagine in memoria per tutto il tempo. In particolare quelle non necessarie dovrebbero essere messe da parte.

La MMU utilizza prima PT1 per indirizzare la tabella delle pagine di primo livello e ottenere la riga delle pagine di secondo livello. La riga localizzata tramite l'indicizzazione nella tabella delle pagine di livello superiore produce l'indirizzo o il numero di frame di una tabella delle pagine di secondo livello.

La MMU utilizza poi PT2 come indice nella tabella delle pagine di secondo livello per individuare ed estrarre la riga che contiene il numero di frame.



Se quella pagina non è in memoria, il bit Presente/Assente nella voce della tabella delle pagine sarà zero, provocando un page fault. Se la pagina è nella memoria, il numero di frame preso dalla tabella delle pagine di secondo livello è combinato con l'offset per costruire l'indirizzo fisico. Questo indirizzo è messo nel bus e inviato alla memoria.

La cosa interessante da notare è che, sebbene lo spazio degli indirizzi contenga più di un milione di pagine, sono effettivamente necessarie solo quattro tabelle delle pagine: la tabella di livello più alto e le tabelle di secondo livello da 0 a 4 MB (per il testo del programma), da 4 MB a 8 MB (per i dati) e i 4 MB in alto (per lo stack).

I bit Presente/Assente in 1021 voci della tabella delle pagine di livello più alto sono impostati a 0, forzando un page fault nel caso qualche processo provasse ad accedervi. Se accadesse, il sistema operativo noterebbe che il processo starebbe provando a far riferimento a memoria cui non dovrebbe e intraprenderebbe un'azione adeguata, come mandargli un segnale o terminarlo.

Il sistema della tabella delle pagine a due livelli può essere esteso a più livelli favorendo maggior flessibilità.

## Tabelle delle pagine invertite

Le tabelle delle pagine multilivello lavorano bene fino a 32 bit di spazio di indirizzi virtuali. Con un computer da 64 bit la tabella delle pagine diventerebbe enorme (più di 30 milioni di GB).

Poiché lo spazio di indirizzamento è  $2^{64}$  byte, se si utilizzano pagine da 4 KB, la tabella delle pagine ha 252 righe. Se ogni riga occupa 8 Byte, la tabella è di oltre 30 PB.

Una soluzione differente è quella delle tabelle delle pagine invertite. Si utilizza una sola riga per frame nella memoria reale, piuttosto che una riga per pagina dello spazio virtuale degli indirizzi.

Per esempio, con indirizzi virtuali con 64 bit, una pagina da 4 KB e 1 GB di RAM, una tabella delle pagine invertire richiede solo 262.144 voci. La voce tiene traccia di ciò che è memorizzato nel frame (processo, pagina virtuale).

Sebbene le tabelle delle pagine invertire risparmiano una gran quantità di spazio, quando lo spazio virtuale degli indirizzi è molto superiore rispetto alla memoria fisica, esse presentano un grosso svantaggio: la traduzione da virtuale a fisica diventa molto difficile.

Quando il processo n riferenzia la pagina virtuale p, l'hardware non può utilizzare p come indice nella tabella delle pagine ma deve cercare invece la coppia (n, p) nell'intera tabella delle pagine invertite. Questa ricerca deve essere eseguita per ogni riferimento alla memoria e non solo in caso di page fault.

La soluzione è data dall'impiego del TLB che memorizza le pagine più utilizzate. Nel caso di una TLB miss deve essere comunque fatta una ricerca sulla tabella delle pagine invertite. In questo caso si può utilizzare una tabella hash sull'indirizzo virtuale.

### **3.d Algoritmi di sostituzione delle pagine**

Quando si verifica un page fault, per far spazio alla pagina entrante il sistema operativo deve scegliere una pagina da rimuovere dalla memoria per far spazio alla pagina che manca.

Se la pagina da rimuovere è stata modificata mentre era in memoria, deve essere riscritta sul disco, prima di essere riscritta. Se tuttavia la pagina non è stata cambiata la copia su disco è già aggiornata e non c'è bisogno di riscrittura.

Ci sono molti metodi per la scelta della miglior pagina candidata per al rimozione, l'idea di base è minimizzare il numero di page fault futuri selezionando quelle pagine che sono usate raramente.

Vale la pena sottolineare che il problema della “sostituzione delle pagine” si verifica anche quando occorre sostituire i dati nella cache, quindi i procedimenti che verranno mostrati hanno una valenza generale.

## L'algoritmo ottimo di sostituzione delle pagine

Il miglior algoritmo di sostituzione delle pagine è semplice da descrivere ma impossibile da implementare.

Al momento del page fault, nella memoria si trova un insieme di pagine, una di loro sarà utilizzata dall'istruzione successiva.

Se ogni pagina fosse etichettata con il numero di istruzioni da eseguire prima che quella pagina sia referenziata per la prima volta, l'algoritmo di sostituzione ottimale indica che deve essere rimossa la pagina con il più alto numero di etichetta. Se una pagina non sarà usata per 8 milioni di istruzioni e un'altra pagina per 6 milioni di istruzioni, rimuovere la prima allontana il page fault che ricaricherebbe la pagina il più in là possibile.

Al momento del page fault, il sistema operativo non ha modo di sapere quando ciascuna delle pagine sarà referenziata la volta successiva.

## Not recently used (NRU)

Al fine di consentire al sistema operativo la raccolta di statistiche utili sull'uso delle pagine, molti computer con memoria virtuale hanno due bit di stato, R e M, associati a ciascuna pagina.

R viene impostato quando si fa riferimento alla pagina (in lettura o scrittura).

M è impostato quando la pagina viene scritta.

I bit sono contenuti in ciascuna riga della tabella delle pagine.

All'avvio di un processo, entrambi i bit di pagina per tutte le pagine sono impostati a 0 dal sistema operativo. Ad ogni impulso del clock, il bit R viene azzerato, per distinguere le pagine che non sono state recentemente referenziate da quelle che lo sono state.

Quando avviene un page fault, il sistema operativo ispeziona tutte le pagine e le divide in 4 categorie basate sui valori attuali dei loro bit R e M:

- classe 0: non referenziato, non modificato ( $R=0, M=0$ );
- classe 1 : non referenziato, modificato ( $R=0, M=1$ );
- classe 2: referenziato, non modificato ( $R=1, M=0$ );
- classe 3: referenziato, modificato ( $R=1, M=1$ ).

Sebbene le pagine di classe 1 sembrino, a prima vista, impossibili, si verificano quando un interrupt del clock azzera il bit R di una pagina di classe 3. Gli interrupt del clock non azzerano il bit M perché questa informazione è necessaria per conoscere se la pagina deve essere riscritta su disco o meno. Azzerare R ma non M produce una pagina di classe 1.

L'algoritmo NRU (not recently used) rimuove una pagina a caso dalla classe non vuota con il numero più basso. In questo algoritmo è chiara l'idea che è meglio rimuovere una pagina modificata che non è stata referenziata nemmeno una volta nell'ultimo intervallo del clock piuttosto che una pagina pulita usata frequentemente.

NRU è facilmente comprensibile, discretamente efficiente da implementare e fornisce prestazioni che, per quanto non ottimali, possono risultare adeguate.

## First-in, first-out (FIFO)

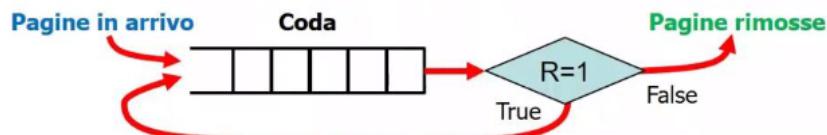
Il sistema operativo tiene una lista delle pagine attualmente in memoria, con l'arrivo più recente in coda e il più vecchio in testa.

A un page fault, la pagina di testa è rimossa e la nuova aggiunta in coda all'elenco.

Poiché la gestione FIFO non tiene conto dell'utilizzo della pagina ma solo dell'ordine di caricamento in memoria è raramente utilizzata.

## Seconda chance

Una semplice modifica a FIFO che evita il problema di gettare una pagina usata di frequente è controllando il bit R della pagina più vecchia. Se è 0, la pagina è vecchia e inutilizzata e viene così sostituita immediatamente. Se R è 1, il bit viene azzerato, alla pagina viene data una seconda chance e messa in fondo alla lista. Poi la ricerca continua.



Sebbene questo algoritmo sia ragionevole, è inefficiente poiché fa scorrere di continuo le pagine lungo la sua lista.

## Clock

Un approccio migliore è di tenere tutti i frame su una lista circolare a forma di orologio.

La lancetta indica la pagina più vecchia. Quando avviene un page fault, la pagina indicata dalla lancetta viene controllata. Se il suo bit R è 0, allora viene sostituita con la nuova pagina inserita al suo posto nell'orologio e la lancetta spostata in avanti di una posizione. Se R è 1, viene azzerato e la lancetta avanzata alla pagina successiva. Questo processo è ripetuto finché non viene trovata una pagina con R = 0.



## Least recently used (LRU)

Una buona approssimazione dell'algoritmo ottimale si basa sull'osservazione che le pagine usate più frequentemente nelle ultime istruzioni lo saranno anche nelle successive. Al contrario quelle inutilizzate da molto tempo lo resteranno ancora per molto.

Questa idea suggerisce un metodo: quando si verifica un errore di pagina, meglio buttare via la pagina che è rimasta inutilizzata per più tempo. Questa strategia si chiama paginazione LRU (least recently used).

Sebbene questo algoritmo sia teoricamente fattibile, non è economico. Per implementare pienamente l'LRU è necessario mantenere una lista collegata di tutte le pagine in memoria, con davanti quelle più usate e dietro quelle meno usate. La difficoltà sta nel fatto che l'elenco deve essere aggiornato a ogni riferimento alla memoria. Trovare una pagina in memoria e cambiargli la posizione è un'operazione che costa del tempo.

Tuttavia sono possibili due realizzazioni hardware:

Contatore a 64 bit;

Matrice binaria.

## LRU con contatore a 64 bit

Il contatore viene incrementato automaticamente dopo ogni istruzione, il contatore è memorizzato in ogni riga nella tabella delle pagine. Ad ogni riferimento in memoria, il contatore di ciascuna pagina referenziata è aggiornato nella tabella delle pagine.

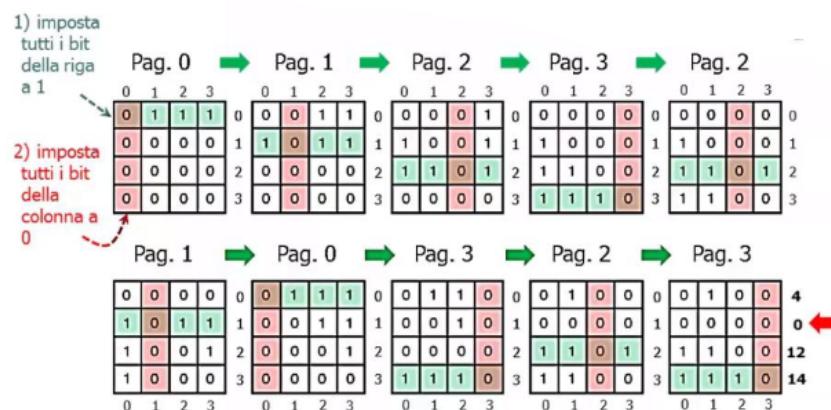
Quando si verifica un page fault, il SO cerca nella tabella delle pagine quella con il minimo valore del contatore ce corrisponde alla pagina utilizzata meno recentemente.

## LRU con matrice binaria

Per un macchina con n pagine fisiche (frame), l'hardware LRU può mantenere una matrice di  $n \times n$  bit, inizialmente tutti a zero.

Ogni volta che la pagina k viene referenziata, l'hardware prima imposta tutti i bit della riga k a 1, poi imposta tutti i bit della colonna k a 0.

La riga il cui valore binario è più basso indica il frame meno recentemente utilizzato.



## Not Frequently Used (NFU)

Sebbene entrambi i precedenti algoritmi siano realizzabili, non esistono hardware che li realizzino. Occorre invece una soluzione che si possa implementare via software.

Una possibilità è l'algoritmo NFU (not frequently used).

Richiede un contatore software associato a ogni pagina e inizialmente impostato a zero. Il sistema operativo fa la scansione di tutte le pagine in memoria, a ogni interrupt del clock.

Per ciascuna pagina viene sommato il bit R (che è 0 o 1) al contatore.

I contatori tengono traccia di quante volte ciascuna pagina è stata referenziata.

Quando avviene un page fault la scelta di quale pagina sostituire cade su quella con il contatore più basso.

Il problema principale dell'NFU è che non si dimentica mai nulla.

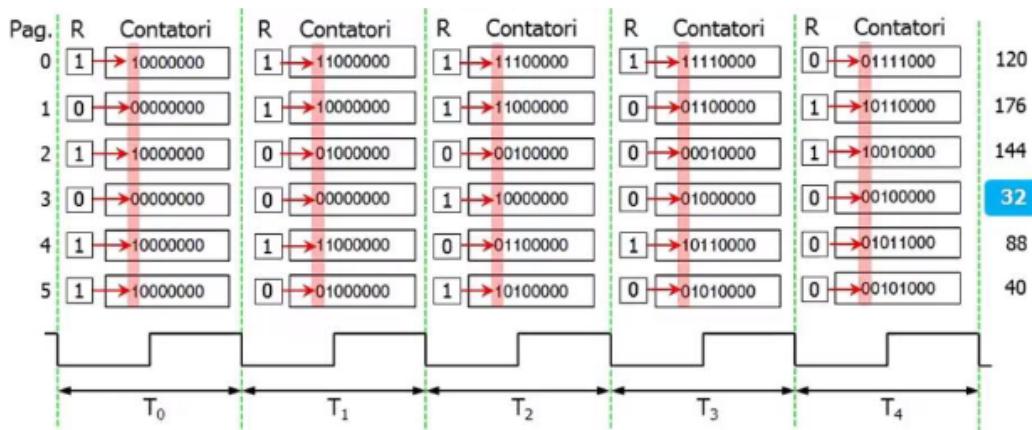
Per esempio, nel caso di un compilatore a molteplici passaggi, le pagine usate frequentemente durante il passaggio 1 avrebbero ancora un contatore alto pure nei passaggi successivi. Spesso questa circostanza si verifica quando il primo ha un tempo di esecuzione maggiore degli altri passaggi.

Di conseguenza il sistema operativo rimuoverebbe pagine utili invece di pagine non più utilizzate.

## Aging

Consideriamo una memoria con 6 pagine. Inizialmente i contatori sono tutti azzerati. Ad ogni ciclo di clock si esegue uno shift a destra e si aggiunge il bit R nella posizione più significativa.

Quando si verifica un page fault, viene rimossa la pagina che ha il minimo valore del contatore (pag 3 nell'esempio).



Questo algoritmo si differenzia dall'LRU in due modi:

- utilizzando un solo bit per intervallo di clock sarà possibile registrare al massimo un riferimento per pagina anche se ce ne fossero di più nel medesimo intervallo di tempo;
- nell'algoritmo di Aging i contatori sono codificati con un numero finito di bit, questo limita l'orizzonte di analisi a quel numero di cicli di clock.

## Working set

La gestione delle pagine più semplice si ha quando i processi sono avviati senza pagine in memoria: appena la CPU prova a prelevare la prima istruzione, genera un page fault, facendo sì che il SO prelevi la pagina che la contiene.

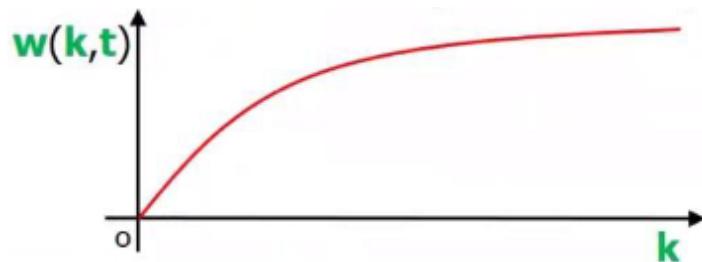
Questa strategia è detta demand paging poiché le pagine sono caricate a richiesta (on demand) e non in anticipo.

Nella maggior parte dei casi vale un principio di località di riferimento, che significa che durante qualunque fase dell'esecuzione il processo utilizza solo una frazione piccola delle sue pagine: il Working Set. Se l'intero working set è in memoria, il processo sarà eseguito senza causare page fault. Se la memoria disponibile è troppo piccola per contenere l'intero working set, il processo provocherà molti page fault e sarà lento. Se un processo causa molti page fault ogni poche istruzioni si dice processo di thrashing.

Molti sistemi di paging cercano di assicurarsi che il working set del processo sia in memoria prima di eseguirlo (Working Set Model). È stato progettato per ridurre notevolmente il tasso di errore di pagina.

Caricare le pagine prima di eseguire i processi è anche detto prepaginazione o prepaging. Notate che il working set cambia nel tempo. Molti programmi non referenziano lo spazio degli indirizzi uniformemente, ma l'insieme di riferimenti tendono a raggrupparsi su un numero ristretto di pagine.

A ogni istante di tempo  $t$  esiste un insieme che consiste di tutte le pagine usate dai  $k$  riferimenti alla memoria più recenti. Questo insieme  $w(k, t)$  è il working set.



$w(k, t)$  è una funzione monotona non decrescente di  $k$  che tende al numero delle pagine del programma. Il limite di  $w(k, t)$  è asintotico in  $k$  e cresce rapidamente con  $k$  piccolo, mentre al crescere di  $k$  cresce più lentamente, poiché un programma non può referenziare più pagine di quante ne contiene il suo spazio degli indirizzi e pochi programmi utilizzeranno ogni singola pagina.

La prepaginazione consiste nel caricamento di queste pagine prima che il processo sia riavviato.

L'algoritmo utilizza un registro a scorrimento con  $k$  posizioni in cui finisce ogni riferimento delle pagine in memoria. Ad ogni page fault è necessario ricaricare il registro e riordinarlo.

Per implementare il modello a working set serve che il sistema operativo tenga traccia di quali pagine sono nel working set. Avere questa informazione porta inoltre immediatamente a un possibile algoritmo di sostituzione delle pagine: quando accade un page fault, cerca una pagina al di fuori del working set e la rimuove.

Invece di considerare  $k$  riferimenti a ritroso, si può utilizzare l'approssimazione di considerare l'insieme delle pagine utilizzate negli ultimi  $T$  ms del tempo di esecuzione. Con questa definizione è più facile lavorare.

Ogni processo conta solo il suo tempo di esecuzione (tempo virtuale attuale o current virtual time). Con questa approssimazione il working set di un processo è l'insieme delle pagine referenziate negli ultimi  $T$  secondi del suo tempo virtuale.

Andiamo adesso ad analizzare un algoritmo di sostituzione delle pagine che si basa sul working set. L'idea base è quella di trovare una pagina che non sia nel working set e rimuoverla.

Poiché solo le pagine in memoria sono da considerare candidate alla rimozione, le pagine assenti dalla memoria sono ignorate da questo algoritmo. Ogni riga della tabella delle pagine contiene:

Il tempo ( $h$ ) dell'ultima volta che la pagina è stata usata;

il bit  $R$  se è stata referenziata nell'ultimo ciclo di clock;

Il bit  $M$  se la pagina è stata modificata.

A ogni page fault, si cerca nella tabella delle pagine la pagina da rimuovere, se il bit  $R$  è: 1, il tempo virtuale attuale è scritto nella riga della pagina ed essa non è una candidata alla rimozione.

0, si calcola l'età della pagina. Se l'età è maggiore di  $T$ , la pagina non è più nel working set e la nuova pagina la sostituisce. Se età è minore o uguale a  $T$  allora la pagina è ancora nel Working Set ed è temporaneamente risparmiata, ma la pagina con l'età maggiore (il valore inferiore di tempo di ultimo utilizzo) viene contrassegnata.

Se la scansione dell'intera tabella non trova alcuna pagina da rimuovere significa che tutte appartengono al working set viene quindi rimossa quella contrassegnata con l'età

maggiori.

Nel peggio dei casi, tutte le pagine sono state referenziate nell'ultimo ciclo del clock (da cui  $R = 1$ ), per cui ne viene presa una a caso per la rimozione tra quelle pulite ( $M=0$ ).

## WSClock

L'algoritmo base del working set è piuttosto lento perché ad ogni page fault occorre scorrere l'intera tabella delle pagine al fine di trovare una pagina adatta alla rimozione, .

Un algoritmo migliore, basato sull'algoritmo Clock, ma che usa anche le informazioni del working set, è chiamato WSClock. Grazie alla sua semplicità di implementazione e alle buone prestazioni, nella pratica è quello più usato.

La struttura dati necessaria è una lista circolare di frame, come nell'algoritmo Clock. Inizialmente questo elenco è vuoto. Quando è caricata la prima pagina, essa viene aggiunta all'elenco.

Ogni riga contiene il campo tempo di ultimo utilizzo (T time of last use), il bit R e il bit M. Come nell'algoritmo clock, a ogni page fault quella indicata dalla lancetta dell'orologio è esaminata per prima. Se il bit R è 1, la pagina è stata usata nel ciclo del clock, quindi non è la candidata ideale alla rimozione e il bit R è impostato a 0. La lancetta avanza alla pagina successiva e l'algoritmo rieseguito per la nuova pagina.

Se l'età  $> T$  AND  $R = 0$  AND  $M = 0$ , non è nel working set e esiste una copia valida su disco. La pagina può essere sovrascritta.

Se l'età  $> T$  AND  $R = 0$  AND  $M = 1$ , non è nel working set e non esiste una copia valida su disco. Per evitare un cambiamento di processo, viene schedulata la scrittura sul disco e la lancetta passa alla pagina successiva.

In caso la lancetta fa un giro dell'orologio e torna al punto di partenza sono da considerare due situazioni:

- 1) è stata programmata almeno una scrittura;
- 2) non ci sono scritture programmata.

Nel primo caso, la lancetta continua a scorrere, alla ricerca di una pagina pulita. La prima pagina pulita incontrata viene rimossa.

Nel secondo caso, tutte le pagine sono nel working set, la cosa più semplice da fare è richiamare una pagina pulita qualunque e usarla. In mancanza di pagine pulite viene scelta come vittima la pagina attuale e scritta sul disco.

## Riepilogo degli algoritmi di sostituzione delle pagine

Non c'è modo di prevedere i riferimenti futuri ad una pagina, quindi l'algoritmo ottimo è utile solo come un punto di riferimento.

L'algoritmo NRU divide le pagine in 4 classi a seconda dei bit R e M: è scelta una pagina a caso nella classe 0. NRU è facile da realizzare ma ne esistono di migliori.

FIFO mantiene l'ordine in cui sono caricate le pagine in memoria, tenendole in una lista collegata. la rimozione della pagina più vecchia è semplice, ma quella pagina potrebbe essere ancora in uso, quindi il FIFO non è utilizzabile..

L'algoritmo Seconda Chance è una modifica del FIFO che controlla se la pagina è in uso prima di rimuoverla. Se lo è, la pagina è risparmiata. Questa modifica incrementa enormemente le prestazioni.

L'algoritmo Clock è semplicemente una diversa implementazione del Seconda Chance. Ha le stesse proprietà prestazionali, ma impiega leggermente meno tempo a eseguire l'algoritmo.

L'LRU è un algoritmo eccellente, ma non può essere implementato senza un hardware speciale.

L'NFU è un tentativo grezzo di avvicinarsi all'LRU.

L'Aging è un'approssimazione dell'NFU decisamente migliore e può essere efficacemente implementata.

L'algoritmo del Working Set fornisce prestazioni ragionevoli, ma richiede lunghi tempi di elaborazione.

Il WSClock è una variante che fornisce buoni risultati ed è anche efficiente.

## 3.e Problemi di progettazione dei sistemi di paginazione

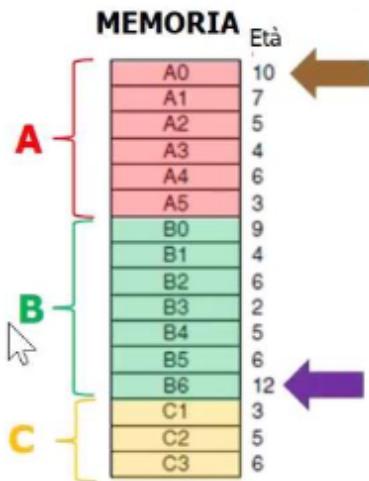
I progettisti di sistemi operativi devono considerare altri aspetti al fine di ottenere buone prestazioni dai sistemi di paging:

- Politiche di allocazione globale e locale;
- Controllo del carico;
- Dimensioni della pagina;
- Dati e istruzioni su spazi separati;

- Pagine condivise.

## Politiche di allocazione globali e locali a confronto

Il problema principale associato a questa scelta è la quantità di memoria che dovrebbe essere ripartita tra processi eseguibili.



I processi A, B e C sono processi in esecuzione. Se A da un page fault, l'algoritmo di sostituzione delle pagine dovrebbe cercare di trovare la pagina meno usata in tutte le pagine in memoria o solo quelle di A?

Se guardasse solo alle pagine di A, la pagina con l'età inferiore sarebbe A5, in questo caso è detto “algoritmo di sostituzione delle pagine locali”.

Se la pagina da rimuovere fosse fra quelle con il valore minore senza guardare a quale appartenesse sarebbe scelta B3 e in questo caso è detto “algoritmo di sostituzione delle pagine globali”.

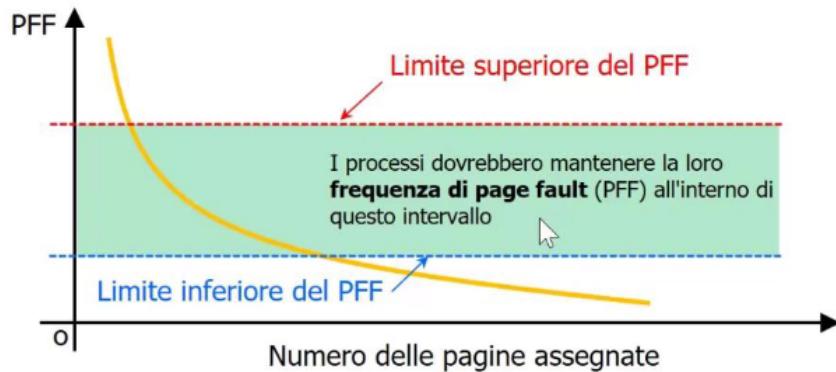
Gli algoritmi globali funzionano meglio quando la dimensione del Working Set può variare durante la vita di un processo.

Se è usato un algoritmo locale e il working set aumenta, ne risulta del thrashing, anche in presenza di frame liberi. Se il working set si restringe, gli algoritmi locali sprecano memoria.

Se è utilizzato un algoritmo globale, il sistema deve continuamente decidere quanti frame assegnare a ciascun processo.

Un metodo è di allocare dinamicamente le pagine di un processo (partendo da un valore iniziale proporzionale alla sua dimensione) facendo in modo che la PFF (page fault

frequency) sia contenuta in un certo intervallo.



Per misurare la frequenza dei page fault basta contare il numero di page fault per secondo. La linea tratteggiata marcata A corrisponde a una frequenza di page fault troppo alta, quindi per diminuire la frequenza dei page fault al processo in errore sono dati più frame. La linea tratteggiata marcata B corrisponde a una percentuale di page fault così bassa da supporre che il processo abbia troppa memoria. In questo caso possono essergli tolti dei frame. Il PFF cerca così di tenere la frequenza di paginazione per ciascun processo all'interno di limiti accettabili.

FIFO, LRU e tutte le approssimazioni dell'LRU possono lavorare sia con politiche di sostituzione delle pagine locali sia globali. Working set e WSClock possono lavorare esclusivamente con politiche di sostituzione delle pagine locali.

## Controllo del carico

Anche se si utilizza il miglior algoritmo di sostituzione delle pagine e l'assegnazione ottimale dei frame il sistema può andare in sovraccarico. Se il Working Set di tutti i processi superano la capacità di memoria si verifica del thrashing. L'unica soluzione reale è di mandare dei processi su disco (swapping).

Nel decidere quale processo mantenere in memoria e quale mandare su disco occorre considerare:

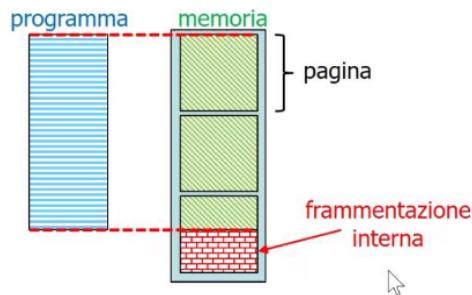
- Il grado di multiprogrammazione,
- La dimensione del processo,
- La frequenza di paginazione (PFF)
- Il tipo di processo (I/O bound o CPU bound).

## Dimensione delle pagine

La dimensione delle pagine è un parametro scelto dal sistema operativo.

Anche se l'hardware è stato progettato per lavorare con frame di k byte, il sistema operativo può scegliere di lavorare con multipli di k. Determinare la migliore dimensione delle pagine dipende da vari aspetti tra loro correlati.

La dimensione dei programmi, dei dati o del segmento di stack non sarà mai un multiplo esatto della dimensione della pagina e quindi ci sarà sempre dello spreco di memoria (frammentazione interna).



Per minimizzare lo spazio inutilizzato all'interno di pagine, esse dovrebbero essere più piccole possibili. Pagine piccole permettono maggiore efficienza per l'esecuzione dei processi: si caricano in memoria solo le componenti che servono. Pagine piccole però richiedono tabelle delle pagine enormi.

I trasferimenti da e verso il disco sono generalmente di una pagina per volta, con la maggior parte del tempo impiegato per la ricerca e il ritardo della rotazione è meglio trasferire pagine grandi.

Caricare 64 pagine di 512 byte (32 KB) richiede  $64 \times 10 = 640$  ms, mentre per caricare 4 pagine da 8 KB servono solo  $4 \times 12 = 48$  ms.

Su alcune macchine la tabella delle pagine deve essere caricata nei registri hardware ogni volta che la CPU passa da un processo a un altro. In questo caso è meglio avere pagine grandi poiché si riduce il tempo necessario per caricare i registri. Gli attuali computer utilizzano pagine da 1, 4 o 8 KB.

Con il crescere delle dimensioni delle memorie anche la dimensione delle pagine tende a crescere anche se in modo non lineare. Quadruplicando la memoria RAM raramente si raddoppia la dimensione della pagina.

## Istruzioni separate e spazi dei dati

La maggior parte dei computer ha un solo spazio degli indirizzi contenente sia i programmi sia i dati.



Se questo spazio degli indirizzi è abbastanza grande, tutto funziona bene. Quando è troppo piccolo costringe i programmi ad adattarsi nell'intero spazio di indirizzi.

Una soluzione è di avere spazi degli indirizzi separati per le istruzioni e i dati: I-space e D-space. In questo modo entrambi gli spazi possono essere paginati indipendentemente l'uno dall'altro. Ognuno ha la sua tabella delle pagine, con il suo mappaggio delle pagine virtuali nei frame fisici.



## Pagine condivise

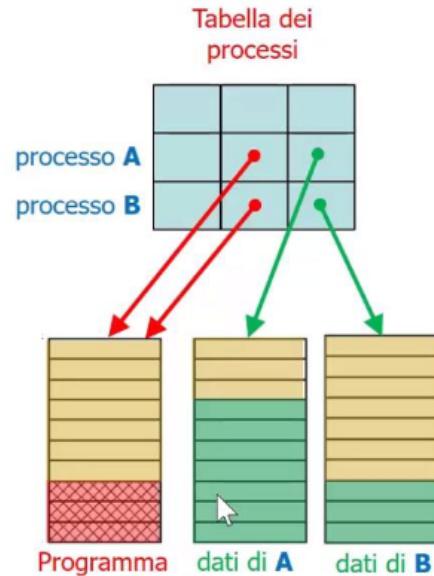
Un altro problema che si presenta durante la progettazione è la condivisione.

In un grande sistema a multiprogrammazione è normale che molti utenti eseguano lo stesso programma contemporaneamente. Per evitare due copie della stessa pagina in memoria contemporaneamente è ovviamente più efficiente condividere le pagine. Il problema è che non tutte le pagine sono condivisibili. In particolare le pagine di sola lettura (read-only), come il testo dei programmi, possono essere condivise, ma le pagine dei dati devono essere gestite con attenzione.

Supponiamo che i processi A e B stiano entrambi eseguendo un editor e condividendo le sue pagine. Se lo

scheduler decide di rimuovere A dalla memoria, eliminando tutte le sue pagine e riempiendo i frame vuoti con qualche altro programma si avrà che, per riportarle di nuovo in memoria, B darà luogo a un gran numero di page fault.

Analogamente, quando A termina, è fondamentale che sia in grado di scoprire se le pagine sono ancora in uso, in modo che il loro spazio su disco non sia liberato accidentalmente.



La condivisione dei dati è più difficile di quella del codice, anche se non è impossibile. In un sistema di paging, ognuno di questi processi ha una riga nella tabella delle pagine ed entrambe puntano allo stesso insieme di pagine (non è presente nessuna copia e tutte le pagine dati sono contrassegnate read-only). Finché i processi leggono i dati tutto funziona bene, non appena un processo tenta di scrivere la violazione del read-only provoca una trap al SO e viene fatta la copia della pagina che ha generato l'errore, ora le due copie hanno vita indipendente.

Così facendo le copie sono ridotte al minimo indispensabile (copy on write)

Cercare tutte le tabelle delle pagine per vedere se una pagina è condivisa è troppo dispendioso; così, per tenere traccia delle pagine condivise, sono necessarie delle strutture dati speciali. Condividere dati invece che codice è più complesso, ma non impossibile.

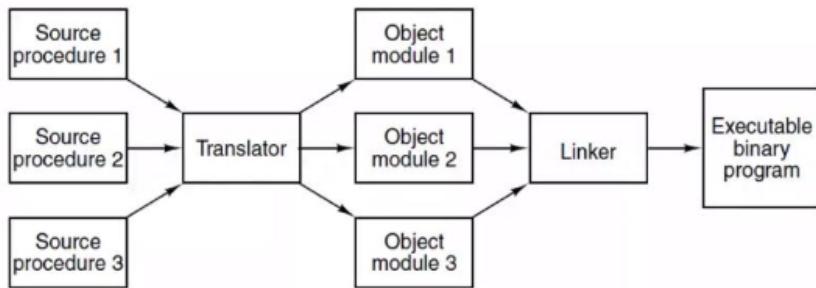
## Librerie condivise

La condivisione può essere fatta con una granularità diversa dalle singole pagine.

Se un programma è avviato due volte, molti sistemi operativi condivideranno automaticamente tutte le pagine testo in modo che ve ne sia in memoria una sola copia.

Le pagine di testo sono sempre in sola lettura, per cui in questo caso non c'è alcun problema. A seconda del sistema operativo, ciascun processo può avere la sua copia privata delle pagine dei dati o essere condivise contrassegnandole come read-only. Se

un qualche processo modifica una pagina di dati, ne viene fatta una copia privata per lui, ossia è applicato il “copy on write”.



Dopo la fase di traduzione, il linker esegue il collegamento dei programmi oggetto. In essi qualsiasi riferimento esterno non definito (chiamata a funzione di sistema come `printf()`) è ricercato e incluso nelle librerie.

Al termine del collegamento il linker scrive sul disco un programma binario eseguibile che contiene tutte le funzioni (dalle librerie sono incluse solo quelle necessarie).

Se un programma utilizza pesantemente i moduli dell'interfaccia grafica potrebbe arrivare ad includere anche 50MB di funzioni.

Collegare centinaia di programmi con tutte queste librerie sprecherebbe una quantità enorme di spazio quando fossero caricate. Si utilizzano librerie condivise, dette librerie a collegamento (DLL, “dynamic link library” in Windows).

Quando un programma è collegato con librerie condivise, invece di includere la funzione richiamata, il linker include una piccola routine di stub che permette di attivare la funzione in fase di esecuzione (run-time). Così facendo, se un altro programma ha già caricato la libreria condivisa non serve ricaricarla. Si noti che quando una libreria condivisa viene utilizzata non viene letta in memoria l'intera libreria.

Oltre a rendere file eseguibili più piccoli e risparmiare spazio nella memoria, le librerie condivise hanno un altro vantaggio: se una di esse viene aggiornata, non serve ricompilare i programmi che la richiamano. I vecchi eseguibili binari continuano a funzionare.

Le librerie condivise tuttavia non consentono la rilocazione del codice al volo, necessaria quando diversi processi accedono alla stessa libreria. Esistono due possibili soluzioni:

- copy-on-write delle pagine per ciascun processo (ma così si perdono i vantaggi della condivisione);

- compilare le librerie con uno speciale flag che permette al compilatore di non produrre le istruzioni che utilizzino indirizzi assoluti (codice indipendente dalla posizione).

Le librerie condivise sono un caso particolare di una struttura più generale chiamata file mappati in memoria. Un processo può inviare una chiamata di sistema per mappare un file in una porzione del suo spazio indirizzi virtuali.

I file mappati in memoria forniscono un modello alternativo per l'IO: invece di fare letture e scritture, si accede al file come un array di caratteri in memoria. Se due o più processi mappano lo stesso file contemporaneamente essi possono comunicare attraverso la memoria condivisa. Quando il SO dispone dei file mappati in memoria, si possono utilizzare per la gestione delle librerie condivise.

## Politica di ripulitura

La paginazione lavora al meglio quando c'è abbondanza di frame liberi che possono essere richiamati se si verificano dei page fault. Se tutti i frame sono allocati con pagine modificate, prima di poter caricare in memoria una nuova pagina è necessario scrivere su disco quella sostituita.

Per garantire l'esistenza di frame liberi, molti sistemi di paging hanno un processo background chiamato paging daemon (demone della paginazione), che ispeziona periodicamente lo stato della memoria

Mantenere a portata di mano una scorta di frame garantisce prestazioni migliori rispetto a usare tutta la memoria e poi cercare di ritrovare il frame al momento opportuno. Il paging daemon assicura perlomeno che tutti i frame liberi siano puliti, così da non doverli scrivere in tutta fretta sul disco quando sono necessari.

Se ci sono pochi frame liberi, inizia a selezionare pagine da eliminare usando un algoritmo di sostituzione delle pagine. La politica di ripulitura può utilizzare l'algoritmo dell'orologio a due passate:

- Primo giro controllato dal paging deamon, se la pagina è modificata è scritta sul disco, altrimenti va avanti.
- Secondo giro si usa l'algoritmo standard di sostituzione delle pagina aumentando la probabilità di trovare una pagina pulita.

## Interfaccia di memoria virtuale

La memoria virtuale è trasparente ai processi e ai programmatori: essi vedono un grande spazio di indirizzamento virtuale su un computer con memoria fisica inferiore.

In alcuni sistemi avanzati, i programmatori hanno un certo controllo sulla mappa di memoria e la possono utilizzare in modi non tradizionali per migliorare il comportamento del programma.

Uno dei motivi per dare il controllo ai programmatori sulla loro mappa della memoria è quello di permettere due o più processi di condividere la stessa memoria.

Un'altra tecnica di gestione avanzata della memoria è chiamata distributed shared memory (memoria condivisa distributiva). L'idea di base è quella di consentire a processi in rete di condividere un insieme di pagine come uno spazio di indirizzamento lineare e condiviso.

Quando un processo fa riferimento a una pagina che non possiede genera un page-fault: Il gestore dell'errore di pagina localizza la macchina che possiede la pagina e le invia un messaggio di richiesta di rimozione della pagina alla mappa e inviarla nella rete.

Quando la pagina arriva, essa è mappata e l'istruzione che ha provocato l'errore viene riavviata.

### **3.f Aspetti realizzativi**

#### **Implicazioni delle paginazione sul SO**

Ci sono quattro momenti in cui il sistema operativo è coinvolto con paginazione:

##### **Durante la creazione del processo.**

Quando viene creato un nuovo processo, il SO deve:

Determinare quante pagine assegna al programma e ai dati e creare le corrispondenti tabelle delle pagine.

Allocare e inizializzare in memoria lo spazio per la tabella pagine.

Riservare uno spazio per garantire lo swap su disco delle pagine.

Inizializzare l'area di swap su disco con programma e dati così che quando il nuovo processo parte i page fault causeranno il caricamento in memoria delle pagine.

Registrare nella tabella dei processi tutte le informazioni della tabella pagine e dell'area di swap su disco.

##### **Durante l'esecuzione del processo.**

Quando un processo è schedulato per l'esecuzione, la MMU deve essere ripristinata per il nuovo processo e la TLB deve essere svuotata (nessuna informazione del processo precedente deve essere mantenuta).

La nuova tabella delle pagine deve diventare quella corrente, di solito copiandola o copiando un puntatore ad essa in un dato registro hardware.

## Durante un page-fault.

Quando si verifica un errore di pagina il SO deve:

Leggere i registri hardware per determinare quale indirizzo virtuale ha causato il page fault.

Calcolare la pagina mancante localizzandola sul disco.

Trovare un frame disponibile per collocare la nuova pagina, rimuovendo la vecchia.

Leggere la pagina dal disco.

Ripristinare il Program Counter per far in modo che punti all'istruzione che ha provocato l'errore in modo da ri-respingerla.

## Durante la chiusura del processo

Quando un processo termina, il SO deve rilasciare la tabella delle pagine, le pagine e lo spazio su disco che le pagine occupano. Se alcune delle pagine sono condivise con altri processi, le pagine in memoria e su disco possono essere rilasciate solo quando l'ultimo processo che le utilizza è terminato.

## Gestione dei page fault

Quando accade un page fault si verificano i seguenti eventi:

L'hardware esegue una trap nel kernel, salvando il PC nello stack.

Viene avviata una routine in codice assembly per salvare i registri generali e le altre informazioni volatili. Il SO cerca di scoprire quale pagina virtuale è richiesta (potrebbe essere scritto in un registro hardware).

Una volta che l'indirizzo virtuale che ha causato l'errore è noto, il SO verifica se l'indirizzo è valido e la protezione coerente con l'accesso e se è presente un frame libero lo usa altrimenti attiva l'algoritmo di sostituzione delle pagine che trova una pagina disponibile.

Se la pagina selezionata è sporca, viene schedulata per il trasferimento sul disco e ha luogo un cambio di contesto: viene sospeso il processo in page fault ed eseguito un altro processo fino a quando il trasferimento sul disco non è completato.

Non appena il frame è stato scritto sul disco (ed è pulito), il sistema operativo schedula un'operazione per portare la pagina in memoria. Mentre la pagina viene caricate, il processo che ha provocato l'errore è ancora sospeso e un altro processo utente è eseguito.

Quando l'interrupt del disco indica che la pagina è arrivata, le tabelle delle pagine vengono aggiornate (il frame è contrassegnato con lo stato normale).

L'istruzione che ha provocato l'errore viene ripristinata e il PC è riportato all'istruzione che ha generato l'errore. Il processo in errore è schedulato e il SO ritorna alla routine che lo ha chiamato.

Questa routine ricarica i registri e altre informazioni di stato, quindi ritorna allo spazio utente per proseguire l'esecuzione, come se non ci fosse stato il page fault.

## Backup delle istruzioni

Quando un programma fa riferimento a una pagina che non è nella memoria, l'istruzione che ha causato il page fault viene interrotta e avviene una trap per il SO. Dopo che il SO ha recuperato la pagina mancante è necessario riavviare l'istruzione che causa la trap.

Un registro interno nascosto in alcune CPU salva il PC prima di eseguire ogni istruzione: queste macchine possono anche avere un secondo registro che indica quali registri sono già stati autoincrementati o autodecrementati e di quanto.

Il SO utilizza queste informazioni al fine di ripristinare lo stato precedente prima dell'istruzione che ha provocato l'errore e poterla riavviare come se la pagina fosse in memoria.

Se l'hardware non dispone di questi registri è il SO che deve capire cosa è successo e come ripristinare l'istruzione in errore.

## Blocco delle pagine in memoria

Memoria virtuale e IO interagiscono in modo sottile.

Si consideri un processo che vuole leggere un file all'interno del proprio spazio di indirizzamento, mentre è in attesa dell'IO viene sospeso e un altro processo va in esecuzione. Quest'ultimo ottiene un errore di pagina.

Se l'algoritmo di paginazione è globale, c'è una piccola probabilità che la pagina contenente il buffer di IO venga rimossa dalla memoria. Se il dispositivo di IO è pronto

per eseguire il trasferimento DMA a quella pagina, rimuovendo i dati finora scritti finirebbero di nuovo sul disco.

Esistono 2 soluzioni:

Bloccare le pagine in memoria che stanno effettuando un IO in modo che non vengano rimosse (pinning della memoria).

Eseguire tutte le operazioni di IO all'interno dei buffer del kernel e successivamente copiare i dati su pagine utente.

## Memoria secondaria

Quando una pagina viene rimossa dalla memoria non sappiamo dove è memorizzata sul disco.

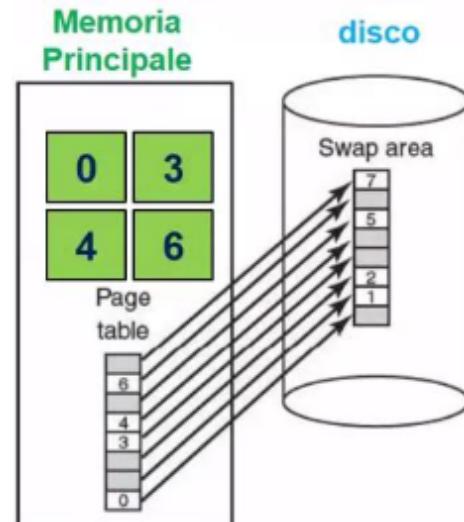
L'algoritmo più semplice (adottato da molti sistemi UNIX) per allocare su disco lo spazio delle pagine è avere una partizione speciale per lo swapping (area di swap) o ancora meglio su un disco separato (per bilanciare il carico di I/O).

All'avvio del sistema questa partizione di swap è vuota ed è rappresentata in memoria come una singola voce, con il suo inizio e la sua dimensione.

Nello swapping statico ad ogni processo corrisponde sul disco uno spazio identico a quello occupato in memoria. Quando i processi terminano, il loro spazio su disco viene liberato.

Nella tabella dei processi è memorizzato soltanto l'indirizzo di inizio sul disco della sua area di swap.

Le pagine 0, 3, 4 e 6 in memoria, mentre le pagine 1, 2, 5 e 7 sono sul disco. L'area di swap sul disco è grande quanto il processo (otto pagine), ogni pagina ha una posizione fissa che viene scritta quando la pagina è rimossa dalla memoria. Le pagine vengono memorizzate in modo continuo in funzione del loro numero di pagina virtuale. Tutte le pagine in memoria hanno una copia ombra (o shadow) su disco,



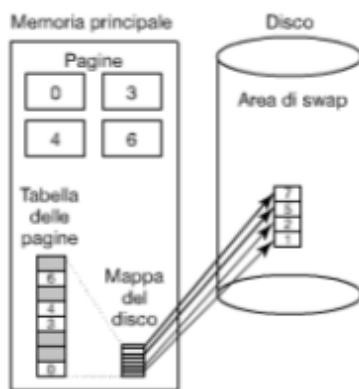
che potrebbe non essere aggiornata se la pagina in memoria è stata modificata.

Questo semplice modello presenta però un problema: i processi possono aumentare di dimensione dopo essere stati avviati. Sebbene il testo programma sia generalmente terminato, l'area dei dati può talvolta crescere, così come lo stack.

Possiamo risolvere riservando aree di swap separate per il testo, i dati e lo stack e sovradimensionando tale spazio per permettere la crescita dei dati e dello stack.

Oppure possiamo non allocare niente in anticipo e allocare lo spazio su disco per ciascuna pagina quando ne venga fatto lo swapping su disco, deallocandola quando viene riportata in memoria.

In questo modo i processi in memoria non sono vincolati a un dato spazio di swap. Questa tecnica è nota come swapping dinamico. Lo svantaggio è che per tener traccia di ogni pagina su disco è necessario avere un indirizzo del disco in memoria. In altre parole, deve esserci una tabella per ogni processo che indica per ogni pagina su disco dove essa sia.



Le pagine non hanno un indirizzo su disco fisso. Quando viene fatto lo swapping su disco di una pagina viene scelta “al volo” una pagina libera sul disco e viene aggiornata la mappa del disco.

## Separazione delle politiche dal meccanismo

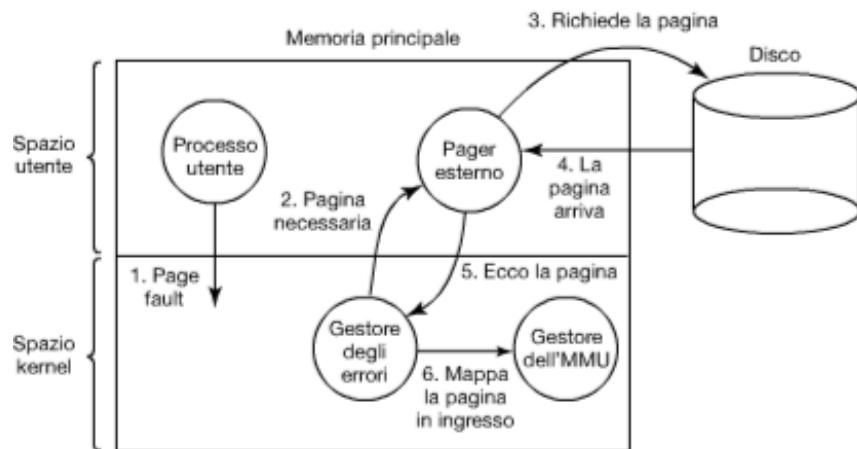
Uno strumento importante per gestire la complessità di qualunque sistema è di separare la politica dal meccanismo.

Questa teoria può essere applicata alla gestione della memoria, poiché nella maggior parte dei computer il gestore della memoria è eseguito come un processo a livello utente.

Il sistema della gestione della memoria è suddiviso in tre parti:

1. Un gestore a basso livello dell'MMU;
2. Un gestore dei page fault è parte del kernel;
3. Un Pager esterno eseguito nello spazio utente.

Una volta che il processo inizia a essere eseguito può generare un page fault. Il gestore degli errori capisce di quale pagina virtuale c'è bisogno e attiva il Pager esterno. Il Pager esterno la richiede al disco, ottiene la pagina e la passa al gestore. Il gestore della memoria chiede al gestore del MMU di mappare la pagina. A questo punto il processo utente può essere riavviato.



Questa implementazione lascia aperta la questione su dove è messo l'algoritmo di sostituzione delle pagine.

Se sta nel Pager, esso non ha accesso ai bit M e R di tutte le pagine fondamentali per qualsiasi algoritmo. Quindi occorre passargli queste informazioni.

Se sta nel kernel si ottiene modularità del codice e maggiore flessibilità. Lo svantaggio principale è il sovraccarico del passaggio dalla modalità kernel a quella utente.

### **3.g Segmentazione**

La memoria virtuale illustrata finora è monodimensionale poiché gli indirizzi virtuali vanno da 0 a un certo indirizzo massimo. Per molti problemi, avere più spazi degli indirizzi virtuali può essere utile. Per esempio, un compilatore utilizza tante tabelle man mano che procede nella compilazione, includendo eventualmente:

1. Il testo sorgente;
2. La tabella dei simboli, contenente i nomi e gli attributi delle variabili;
3. La tabella contenente tutte le costanti intere e a virgola mobile utilizzate;
4. L'albero di parsing, contenente l'analisi statica del programma;
5. Lo stack usato per le chiamate di procedura nel compilatore.

Con il procedere della compilazione ognuna delle prime quattro tabelle cresce continuamente. Durante la compilazione l'ultima cresce e si comprime in modi imprevedibili.



Consideriamo che cosa accade se un programma ha un numero di variabili molto più grande del solito e una normale quantità di tutto il resto. Il pezzo dello spazio degli indirizzi allocato per la tabella dei simboli può riempirsi, ma può esservi molto spazio nelle altre tabelle. Quello che occorre è un modo per liberare il programmatore dal dover gestire l'espansione e la contrazione delle tabelle, nello stesso modo in cui la memoria virtuale elimina il fastidio di organizzare i programmi in overlay.

Una soluzione diretta molto generica è fornire la macchina di molti spazi degli indirizzi indipendenti, chiamati segmenti.

Ogni segmento consiste di una sequenza lineare di indirizzi, da 0 a un certo massimo. La lunghezza di ciascun segmento può variare durante l'esecuzione. Poiché ogni segmento costituisce uno spazio degli indirizzi separato, segmenti diversi possono crescere o compattarsi indipendentemente, senza influenzarsi l'un l'altro.

Per specificare un indirizzo in questa memoria segmentata o bidimensionale, il programma deve fornire un indirizzo a due parti: un numero di segmento e un indirizzo all'interno del segmento.

È bene sottolineare che il segmento è una entità logica, di cui il programmatore è a conoscenza e che utilizza come accade per gli array, le procedure, ...

Oltre a semplificare il trattamento delle strutture di dati che crescono o si riducono di dimensione, una memoria segmentata ha anche altri vantaggi.

Se ogni procedura occupa un segmento separato (indirizzo iniziale 0), il linking di procedure compilate separatamente è facile.

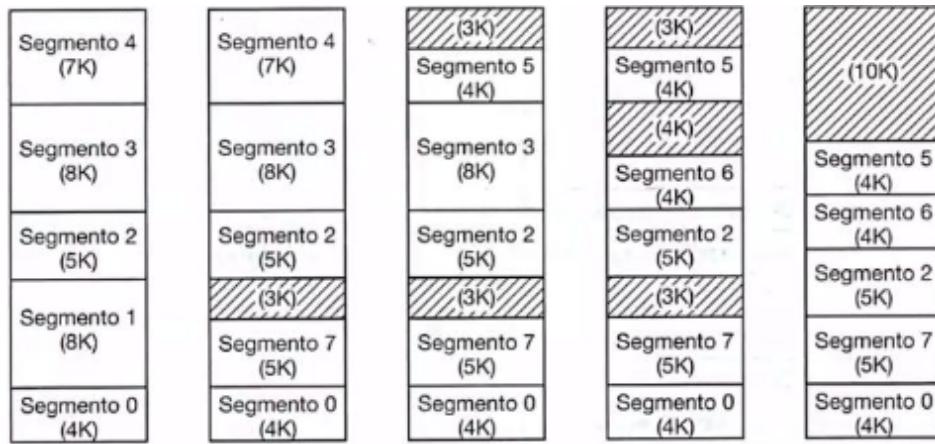
La segmentazione facilita inoltre la condivisione di procedure o di dati tra diversi processi: le librerie condivise possono essere messe in un segmento e condivise da molteplici processi.

I segmenti possono avere diversi tipi di protezione a seconda del loro contenuto (una procedura può essere eseguita e non scritta, un array utilizzato ma non eseguito). Per questo è importante inserire in un segmento un solo tipo di oggetto.

Mentre il programmatore sa cosa c'è dentro un segmento, il contenuto delle pagine e la loro gestione è totalmente invisibile al programmatore.

## **Implementazione della segmentazione pura**

L'implementazione della segmentazione differisce dalla paginazione perché la lunghezza delle pagine è fissa, nei segmenti varia.



Esempio di memoria fisica contenente in principio cinque segmenti ma man mano vengono tolti e aggiunti segmenti.

Dopo che il sistema è stato in esecuzione per un po', la memoria sarà fatta a pezzetti, la memoria verrà suddivisa tra segmenti contenti dati e zone di memoria libera (checkerboarding o frammentazione esterna). Gli spazi vuoti saranno poi uniti per ottenere un unico grande spazio vuoto.

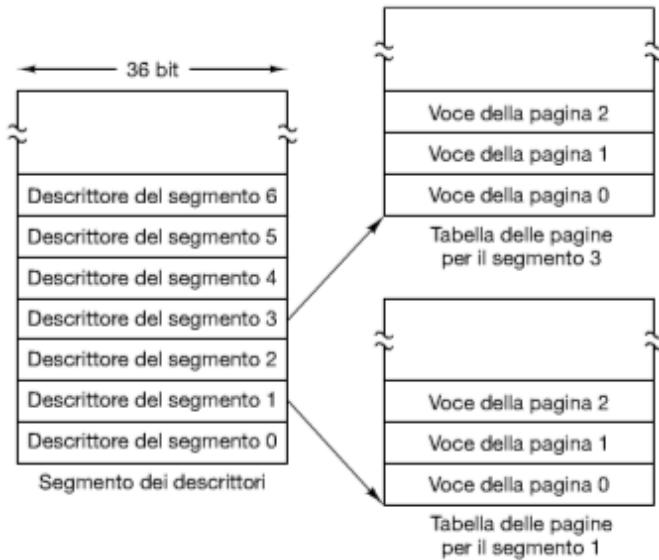
## Segmentazione con la paginazione: MULTICS

Se i segmenti sono grandi non è conveniente mantenerli nella memoria principale. Questo porta all'idea di paginarli, in modo che siano in giro solo quelle pagine effettivamente necessarie. Molti sistemi hanno supportato la paginazione dei segmenti.

Il SO MULTICS forniva ciascun programma di una memoria virtuale sino a  $2^{18}$  segmenti (più di 250.000), ognuno dei quali poteva essere lungo fino a 65.536 parole.

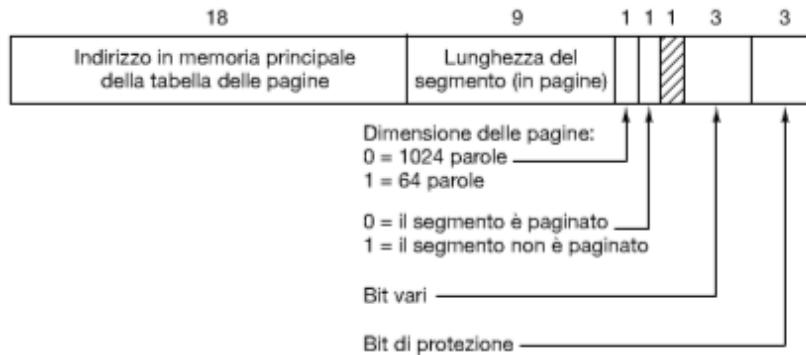
Ciascun segmento è paginato, combinando i vantaggi della paginazione (la dimensione delle pagine uniforme senza tenere l'intero segmento in memoria, ma solo la parte che deve essere usata) con i vantaggi della segmentazione (facilità di programmazione, modularità, protezione e condivisione).

Ogni programma MULTICS ha una tabella dei segmenti, con un descrittore per segmento. La stessa tabella dei segmenti è segmentata e paginata.



Un descrittore di segmento contiene un'indicazione se il segmento sia nella memoria principale o meno. Se una qualunque parte del segmento è in memoria, il segmento è considerato in memoria e la sua tabella delle pagine.

Nel MULTICS un indirizzo consiste di due parti: il numero di segmento e l'indirizzo nel segmento. L'indirizzo nel segmento è ulteriormente suddiviso in un numero di pagina e un offset.



Quando avviene un riferimento alla memoria:

1. Il numero del segmento è usato per trovare il descrittore del segmento.
2. Si controlla se la tabella delle pagine del segmento è in memoria. Se lo è, viene localizzata. Se non lo è avviene un segment fault.
3. Viene esaminata la riga della tabella delle pagine per la pagina virtuale richiesta. Se la pagina stessa non è in memoria viene generato un page fault. Se lo è, viene

estratto l'indirizzo della pagina in memoria.

4. All'inizio della pagina è aggiunto l'offset.
5. Legge o scrive la pagina.

Ripetere tutti questi passaggi ad ogni riferimento di memoria può rallentare l'intero sistema.

MULTICS contiene un TLB ad alta velocità a 16 parole che può ricercare in un sol colpo una determinata chiave in tutte le sue voci.

Ad ogni riferimento di memoria prima si controlla dentro la TLB. Se la coppa (segmento, indirizzo) è presente troviamo direttamente il frame senza dover guardare nel segmento descrittore o nella tabella delle pagine.

## **Segmentazione con la paginazione: l'Intel x86 (Pentium)**

La memoria virtuale sul Pentium è simile a quella sul MULTICS.

Mentre il MULTICS aveva 256.000 segmenti indipendenti, ciascuno con massimo 64.000 parole a 36 bit, l'x86 ha 16.000 segmenti indipendenti, ognuno contenente fino a un miliardo di parole a 32 bit. Sebbene vi siano meno segmenti, la dimensione dello stesso è più grande.

La memoria virtuale del Pentium è fondata su due tavole:

- LDT (local descriptor table, tabella dei descrittori locali);
- GDT (global descriptor table, tabella dei descrittori globali).

Ciascun programma ha la propria LDT, ma vi è una sola GDT, condivisa da tutti i programmi sul computer.

La LDT descrive i segmenti locali a ciascun programma, inclusi il suo codice, i dati, lo stack e così via, mentre la GDT descrive i segmenti di sistema, compreso il sistema operativo stesso.

Per accedere a un segmento, un programma x86 carica prima un selettore per quel segmento stesso all'interno di uno dei sei registri segmento della CPU. Durante l'esecuzione, il registro CS contiene il selettore per il segmento del codice e il registro DS contiene il selettore per il segmento dei dati.

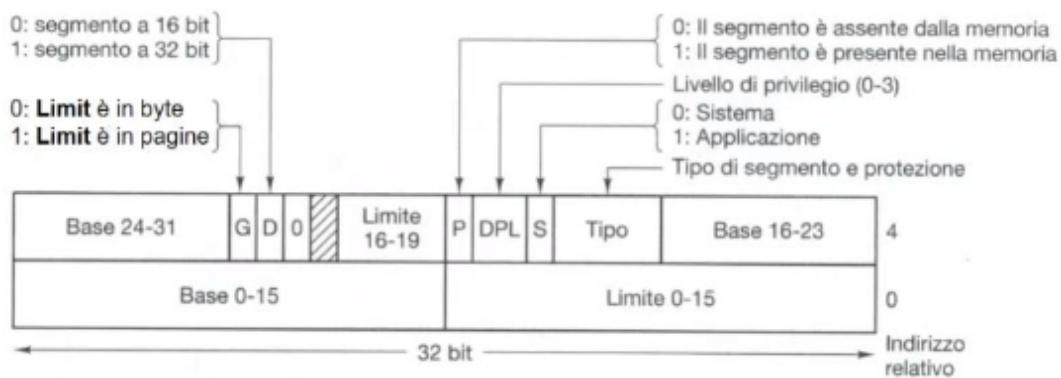
Ogni selettore è un numero a 16 bit:

- 1 bit indica se il segmento è locale o globale, cioè se sta nella LDT (0) o nella GDT (1);

- 13 bit specificano la riga nella tabella LDT o nella GDT;
- 2 bit sono di protezione.

Al momento in cui un selettor è caricato in un registro segmento, il suo descrittore corrispondente è richiamato dalla LDT o dalla GDT e memorizzato nei registri dei microprogrammi.

Un descrittore consiste di 8 byte, incluso l'indirizzo di base del segmento, la dimensione e altre informazioni.



Andiamo ora a tracciare i passi tramite cui una coppia <selettor, offset> è convertita in un indirizzo fisico.

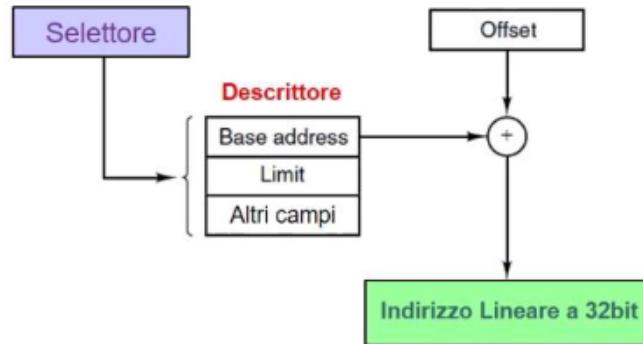
Appena il microprogramma sa quale registro segmento è stato usato, può trovare il descrittore completo corrispondente a quel selettor nei suoi registri interni.

Si verifica una trap se il segmento non esiste o è salvato su disco per effetto del paging oppure se l'offset è dopo la fine del segmento (Limit).

Nel descrittore ci sono solo 20 bit disponibili (invece che 32). Se il campo Gbit (Granularity bit) è 0, il campo Limite è in byte, fino a 1 MB. Se è 1, il campo Limite fornisce la dimensione del segmento in pagine invece che byte.

Poiché la dimensione della pagina nel Pentium è 4 KB, 20 bit sono sufficienti per segmenti sino a  $2^{32}$  byte.

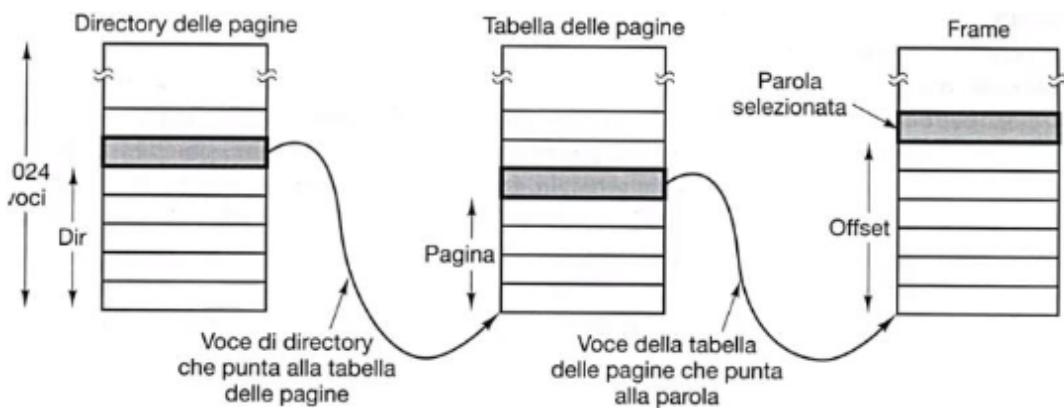
Dando per assunto che il segmento sia in memoria e l'offset all'interno dell'intervallo, al campo Base (32 bit) del descrittore è aggiunto l'offset per formare un indirizzo lineare.



Se la paginazione è disabilitata (tramite un bit nel registro di controllo globale), l'indirizzo lineare è interpretato come un indirizzo fisico e inviato alla memoria per la lettura o la scrittura. D'altra parte, se la paginazione è attivata, l'indirizzo lineare è interpretato come un indirizzo virtuale e mappato sull'indirizzo fisico usando le tabelle delle pagine.

Ogni programma in esecuzione ha una directory delle pagine composta di 1024 voci a 32 bit.

Un registro globale indica la riga di questa directory che punta a una tabella delle pagine contenente 1024 voci a 32 bit. Le voci della tabella delle pagine puntano ai frame delle pagine.



L'indirizzo lineare è diviso in tre campi:

Il campo Dir è l'indice nella directory delle pagine: individua un puntatore alla corretta tabella delle pagine.

Il campo Pagina viene utilizzato come indice nella tabella delle pagine per trovare l'indirizzo fisico del frame della pagina.

L'Offset viene aggiunto l'indirizzo del frame della pagine per ottenere l'indirizzo fisico del byte o della parola necessaria.

Le righe della tabella delle pagine sono lunghe 32 bit, 20 dei quali contengono il numero di frame di pagina. I restanti bit, impostati dall'hardware a beneficio del sistema operativo sono:

- Bit di accesso;
- Dirty bit,
- Bit di protezione;
- Altri bit di utilità.

Ogni tabella delle pagine ha 1024 righe per frame di pagine a 4KB in modo che una singola tabella delle pagine gestisce 4MB di memoria. Per evitare di fare accessi ripetuti alla memoria, l'x86 ha una piccola TLB che mappa direttamente le combinazioni <Dir, Pagina> utilizzate più recentemente.

## 4. File system

Tutte le applicazioni per computer devono memorizzare e recuperare informazioni.

Durante l'esecuzione un processo può memorizzare una quantità limitata di informazioni all'interno del proprio spazio di indirizzamento. Riscontriamo diversi problemi:

Primo problema: Per alcune applicazioni una certa dimensione può essere sufficiente, ma può rivelarsi troppo piccola per altri.

Secondo problema: quando il processo termina, le informazioni scritte all'interno del suo spazio di indirizzamento sono perdute. Per molte applicazioni le informazioni devono essere conservate per settimane, mesi o anche indefinitamente, anche quando un crash del computer uccide il processo.

Terzo problema: frequentemente più processi devono poter accedere contemporaneamente ad una parte delle informazioni quindi l'informazione non deve essere disponibile solo per un processo.

Una possibile soluzione è rendere questa stessa informazione indipendente da qualunque processo. I requisiti essenziali per il salvataggio delle informazioni a lungo termine sono quindi tre:

1. Deve essere possibile salvare una grandissima quantità di informazioni;

2. Le informazioni devono permanere oltre la fine del processo che le usa;
3. Le informazioni devono poter essere acquisite contemporaneamente da molteplici processi.

Per questo tipo di memorizzazione a lungo termine sono stati usati per anni i dischi magnetici, oltre ai nastri e ai dischi ottici, che però hanno prestazioni molto inferiori.

Un disco contiene una sequenza lineare di blocchi dalle dimensioni fisse e supporta due operazioni: la lettura del blocco k e la scrittura del blocco k.

Si tratta di operazioni molto scomode, specialmente su grandi sistemi usati da molte applicazioni e utenti concorrenti.

Alcune delle tante domande che possono sorgere sono le seguenti.

1. Come trovare le informazioni?
2. Come evitare che un utente legga le informazioni di un altro?
3. Come sapere quali blocchi sono liberi?

Il SO può risolvere questa complessità con una nuova astrazione: il file.

In un SO le principali tre astrazioni sono quindi: Processi e thread, Spazio di indirizzamento e File.

I file sono unità logiche di informazioni create dai processi. I processi possono leggere file esistenti e crearne di nuovi se necessario. Le informazioni salvate nei file sono persistenti, cioè non influenzate dalla creazione e dalla fine del processo. Un file dovrebbe scomparire solo quando il suo proprietario lo rimuove esplicitamente.

I file sono gestiti dal sistema operativo. I principali argomenti nella progettazione di un SO sono le caratteristiche dei file: struttura, denominazione, accesso, utilizzo, protezione, realizzazione e gestione. La parte del sistema operativo che nella sua interezza ha a che fare con i file è conosciuta come file system.

Dal punto di vista dell'utente, l'aspetto principale di un file system è come esso appare, ossia, che cosa costituisca un file, come siano denominati e protetti i file, quali operazioni siano consentite sui file e così via. Il fatto che per tenere traccia della memoria libera siano usati elenchi collegati piuttosto che bitmap e quanti siano i settori all'interno di un blocco logico non sono aspetti importanti per l'utente, ma lo sono molto per i progettisti di file system. Analizzeremo entrambe le parti.

## **4.a I File**

### **Nomi dei file**

La principale caratteristica di qualsiasi meccanismo di astrazione è il modo in cui gli oggetti sono denominati.

Le regole esatte per la denominazione dei file possono in qualche modo variare da sistema a sistema, ma tutti i sistemi operativi attuali consentono, come nomi validi stringhe che vanno da uno a otto caratteri (ad esclusione dei meta caratteri usati dal file system: “< > ecc...”).

Molti file system supportano nomi lunghi fino a 255 caratteri e distinguono fra lettere maiuscole e minuscole, mentre altri no. UNIX ricade nella prima categoria, MS-DOS nella seconda.

Molti sistemi operativi adottano la convenzione di separare il nome del file dal uso tipo (estensione del file).

In MS-DOS, per esempio, i nomi dei file vanno da 1 a 8 caratteri, più un'estensione opzionale da 1 a 3 caratteri.

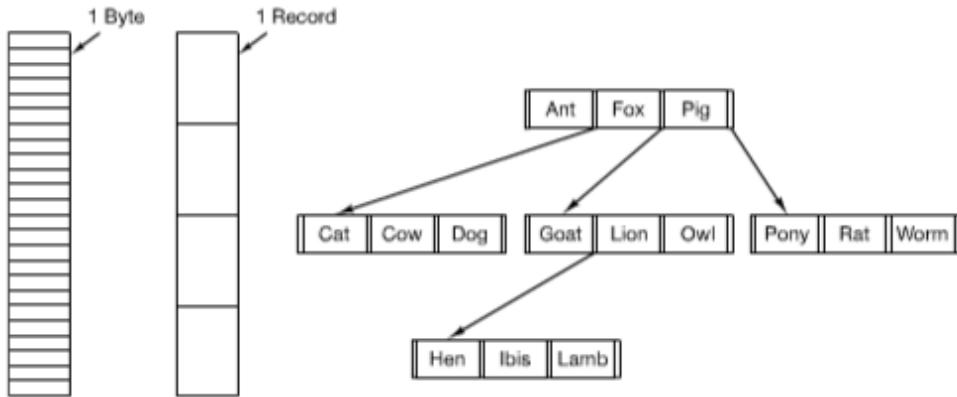
### **Struttura dei file**

I file possono essere strutturati in tanti modi diversi quelle più comuni sono.

Sequenza di byte (senza struttura). Il sistema operativo non sa cos'è contenuto nel file, vede solo una sequenza di byte. Ciò da massima flessibilità: i programmi utente possono metter tutto ciò che vogliono nei loro file. Tutte le versioni di UNIX, MS-DOS e Windows usano questo modello.

Sequenza di record. Il sistema operativo vede una sequenza di record a lunghezza fissa e con una struttura ben definita. Le operazioni di scrittura e lettura avvengono sui record. I primi mainframe utilizzavano record da 80 caratteri così come le 80 colonne di una scheda perforata.

Albero di record. Non necessariamente della stessa lunghezza, ognuno contenente un campo chiave in una posizione fissa nel record. L'albero è ordinato sul campo chiave così da velocizzare le ricerche. L'operazione base non è prendere il record "successivo" ma prendere il record con una chiave ben precisa.



## Tipi dei file

Molti sistemi operativi supportano diversi tipi di file. UNIX e Windows per esempio hanno file e directory normali. UNIX ha anche file speciali a caratteri o a blocchi.

I file normali sono quelli contenenti informazioni utente. Le directory sono file di sistema usati per mantenere la struttura del file system.

I file speciali a caratteri sono legati all'input/output e usati per modellare i dispositivi seriali di I/O, come terminali, stampanti e network.

I file speciali a blocchi sono usati per modellare i dischi.

I file normali sono generalmente sia file ASCII sia file binari.

I file ASCII sono composti da righe di testo, non necessariamente della stessa lunghezza. In alcuni sistemi ogni riga termina con un carattere speciale di "a capo" (carriage return) o "nuova riga" (line feed).

I file ASCII sono facili da visualizzare e stampare e possono essere corretti con un qualunque editor di testo. Inoltre, i programmi che utilizzano file ASCII per l'input/output possono essere interfacciati con facilità (es. pipe).

Altri file sono binari (cioè non sono file ASCII). Sono sequenze di bit che non vengono interpretati come caratteri e la loro struttura interna è conosciuta solo dai programmi che li utilizzano. Non possono essere stampati e maneggiati da editor di testi.

In figura vediamo un semplice file binario eseguibile preso nelle prime versioni di UNIX. Il sistema operativo esegue il file solo se ha un formato corretto:  

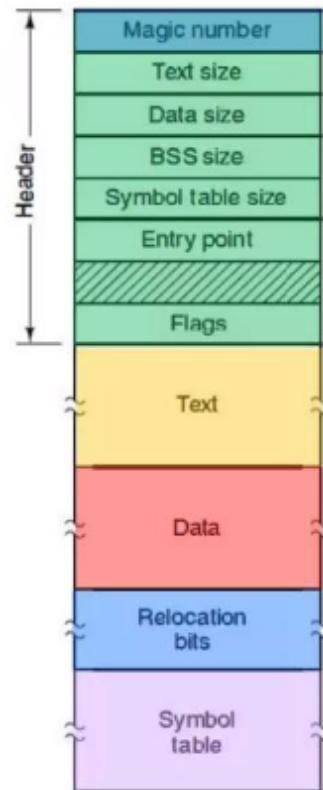
- intestazione (header);

- testo;
- dati;
- bit di rilocazione (relocation bit);
- tabella dei simboli.

L'intestazione inizia con un identificatore per il file eseguibile (numero magico).

Questo numero è seguito da:

- le dimensioni dei segmenti che compongono il file (testo, dati, BSS e tabella dei simboli);
- l'indirizzo di inizio del programma (entry point);
- alcuni flag.



## Accesso ai file

I primi sistemi operativi avevano un solo tipo di accesso ai file: l'accesso sequenziale. I byte vengono letti in ordine, a partire dall'inizio. I file sequenziali erano perfetti quando l'archiviazione era su nastro magnetico piuttosto che su disco, visto che potevano essere riavvolti.

Con la comparsa dei dischi è possibile leggere i byte o i record in qualsiasi ordine (file ad accesso casuale) o accedere ai record secondo una chiave. Per specificare dove cominciare a leggere possono essere usati due metodi.

L'operazione `read()` permette di leggere i dati a partire da una posizione specifica nel file.

L'operazione `seek()` permette di impostare la posizione corrente nel file in modo da poterlo poi leggere sequenzialmente da quella posizione.

## Attributi dei file

Ogni file ha un nome, un contenuto e alcuni attributi (meta-dati): timestamp di creazione, dimensione, utenti autorizzati, ecc...

L'elenco degli attributi cambia da sistema a sistema. La tabella mostra le più comuni.

Attributo	Significato
Protezione	Chi può leggere, eseguire e scrivere il file.
Password	Password necessaria per accedere al file.
Creator ID	IDentificatore dell'utente che ha creato il file.
Owner	Proprietario attuale.
Read-only flag	0 = Read/Write, 1 = Read-Only.
Hidden flag	0 = normale, 1 = non visualizzabile in elenco.
System flag	0 = file normale, 1 = file di sistema.
Archive flag	0 = è stato effettuato il backup, 1 = deve essere effettuato il backup.
ASCII/binary flag	0 = file ASCII file, 1 = file binario.
Random access flag	0 = solo accesso sequenziale, 1 = accesso casuale.
Temporary flag	0 = normale, 1 = file rimosso con la terminazione del processo.
Lock flag	0 = unlocked, 1 = locked.
Lunghezza del record	Dimensione del record in byte.
Posizione della chiave	Offset della chiave all'interno di ciascun record.
Lunghezza della chiave	Dimensione della chiave in byte.
Tempo di creazione	Timestamp (data e orario) quando il file è stato creato.
Ultimo accesso	Timestamp (data e orario) relativo all'ultimo accesso al file.
Ultima modifica	Timestamp (data e orario) relativo all'ultima modifica del file.
Dimensione corrente	Dimensione del file espresso in byte.
Dimensione massima	Massimo numero di byte che il file può contenere.

## Operazioni sui file

I file nascono per memorizzare informazioni e per poterle rileggere in un secondo momento. Esistono varie operazioni disponibili nei SO, le chiamate di sistema più comuni sono:

- `create()`: crea un file vuoto con alcuni attributi impostati.
- `delete()`: rimuove il file e libera lo spazio sul disco.
- `open()`: prima di poter usare un file, un processo deve aprirlo. Memorizza in memoria gli attributi e l'elenco degli indirizzi del disco per un accesso rapido nelle chiamate a seguire.
- `close()`: chiude il file liberando la memoria dai riferimenti al file perché non verrà più utilizzato. Molti sistemi incoraggiano la cosa imponendo ai processi un numero

massimo di file aperti. Un disco è scritto a blocchi e la chiusura di un file forza la scrittura dell'ultimo blocco del file.

- `read()`: legge i dati dal file. Generalmente i byte vengono letti dalla posizione attuale, occorre specificare quanti dati bisogna leggere e il buffer in memoria ove inserirli.
- `write()`: scrive i dati nel file, dalla posizione corrente.
- `append()`: aggiunge dati alla fine del file.
- `seek()`: cambia la posizione corrente nei file ad accesso casuale. Dopo questa chiamata i dati possono essere letti o scritti a partire da quella posizione.
- `get attributes()`: permette di leggere gli attributi dei file.
- `set attributes()`: permette all'utente di impostare alcuni degli attributi.
- `rename()`: permette di rinominare il file.

## **Un esempio di programma che usa le chiamate del file system**

In questo paragrafo prenderemo in esame un semplice programma UNIX che copia un file da uno sorgente a uno di destinazione.

Il programma, `copyfile`, può essere per esempio chiamato dalla linea di comando `copiaFile sorgente destinazione`.

Se destinazione esiste già, sarà sovrascritto, altrimenti sarà creato. Il programma deve essere chiamato esattamente con due parametri, entrambi due nomi di file consentiti. Il primo è il di input; il secondo il file di output.

Le quattro dichiarazioni di `#include` vicine all'inizio del programma causano l'inclusione nel programma di una gran quantità di definizioni e prototipi di funzioni. Queste sono necessarie per rendere il programma conforme agli standard internazionali pertinenti, ma non ci interessano ulteriormente.

La riga successiva è un prototipo di funzione per `main`, richiesto dall'ANSI C, ma anch'esso non significativo per i nostri scopi.

La prima istruzione `#define` è una definizione che dichiara la stringa di caratteri `BUF_SIZE` come una macro che si traduce nel numero 4096. Il programma leggerà e scriverà in pezzi di 4096 byte. La seconda dichiarazione `#define` determina chi può accedere al file di output.

Il programma principale è chiamato main e ha due parametri, argc e argv. Il primo indica quante stringhe sono presenti nella riga di comando che invoca il programma, incluso il nome programma. Dovrebbero essere 3. Il secondo è un array di puntatori ai parametri.

Il programma utilizzerà: una matrice di 4096 caratteri per leggere e scrivere un blocco di 4KB del file, due riferimenti ai file (descrittori dei file) in\_fd, per il file di input e out\_fd per quello di output, byte\_letti e byte\_scritti memorizzando il numero di byte letti e scritti.

```
#include <sys/types.h>          /* Include i file di intestazione necessari */
#include <fcntl.h>                /* */
#include <stdlib.h>               /* */
#include <unistd.h>               /* */
int main (int argc, char *argv[]);  /* prototipo ANSI del main() */
#define BUF_SIZE 4096              /* Utilizza un buffer di 4096 byte */
#define OUTPUT_MODE 0700            /* protezione per il file di output 1110000000 */

int main (int argc, char *argv[]) {
    int in_fd, out_fd, byte_letti, byte_scritti;
    char buffer[BUF_SIZE];
    if (argc != 3) exit(1);
    in_fd = open (argv[1], O_RDONLY);
    if (in_fd < 0) exit(2);
    ou_fd = create (argv[2], OUTPUT_MODE);
    if (ou_fd < 0) exit(3);
    while (1) {
        byte_letti = read(in_fd, buffer, BUF_SIZE);
        if (byte_letti <= 0) break;
        byte_scritti = write (out_fd, buffer, byte_letti);
        if (byte_scritti <= 0) exit(4);
    }
    close(in_fd);
    close(ou_fd);
    if (byte_letti == 0)  exit(0);
    else                  exit(5);
}
```

/\* Errori di sintassi se non argv è diverso da 3 \*/  
/\* Apre il file sorgente \*/  
/\* Se non può essere aperto, uscita con errore \*/  
/\* Crea il file di destinazione \*/  
/\* Se non può essere creato, uscita con errore \*/  
/\* Ciclo per la copia \*/  
/\* Prova a leggere un blocco di 4KB \*/  
/\* Non è riuscito a leggere byte, esce dal ciclo \*/  
/\* Ora prova a scrivere i byte letti sul file \*/  
/\* Non ha scritto alcun byte, errore \*/  
/\* Chiude i due file \*/  
/\* L'ultima lettura è ok, uscita senza errori \*/  
/\* Errore nell'ultima lettura, uscita con errore \*/

## 4.b Le Directory

I file system organizzano file in contenitori denominati directory o cartelle.

### Sistemi di directory a livello singolo

La forma più semplice di sistema di directory è di avere una sola directory contenente tutti i file, talvolta chiamata directory principale (root directory). Il vantaggio di questo schema è la semplicità e la capacità di localizzare i file rapidamente, c'è solo un posto dove guardare.

### Sistemi di directory gerarchici

L'adozione di una definizione ricorsiva in cui ogni directory può contenere altre directory (e/o file), con il vincolo che per ciascuna di esse esista sempre un solo contenitore, conduce ad una struttura gerarchica

## Path name

Quando il file system è organizzato secondo un albero per identificare in modo univoco ogni oggetto (dir o file) della struttura sono utilizzati due metodi:

Path name assoluto: ogni file (o dir) viene identificato dal percorso necessario per raggiungerlo a partire dalla directory principale. Il percorso è composto dalla sequenza ordinata delle directory, separate da un carattere speciale “/” in UNIX, “\” in Windows e “>” in MULTICS. Se il primo carattere del nome percorso è il separatore, allora il riferimento è assoluto.

Path name relativo: il file o la directory viene individuato utilizzando un punto relativo sull'albero (la directory corrente, la home dell'utente, ...).

La maggior parte dei sistemi operativi hanno in ciascuna directory due file speciali:

- “.” è la directory corrente;
- “..” è il suo genitore (tranne per root che fa riferimento a se stesso).

## Operazioni sulle directory

Le chiamate di sistema per la gestione delle directory hanno qualche variante da sistema a sistema ma in genere:

`mkdir()`: crea una directory vuota (ad eccezione di “.” e “..” messi in automatico).

`rmdir()`: rimuove una directory vuota (cioè contenere esclusivamente “.” e “..”).

`opendir()`: carica in memoria tutti i riferimenti di localizzazione sul disco prima della lettura (così come avviene per i file).

`closedir()`: chiude la directory al termine della lettura e libera il corrispondente spazio di memoria.

`readdir()`: restituisce la successiva voce di una directory aperta.

`rename()`: rinomina una directory esistente.

`link()`: crea un link hard (collegamento fisico) tra un file esistente e il pathname indicato. È una tecnica che permette a un file di apparire in più di una directory. Si tratta di un link

hard poiché il file system incrementa il contatore nell'i-node file (in questo modo si tiene traccia del numero di directory che contengono il file).

unlink(): rimuove il collegamento nella directory, se il file è unico è cancellato dal file system altrimenti è rimosso solo un collegamento.

## **4.c Implementazione del file system**

Agli utenti interessa come sono chiamati i file, che operazioni si possono fare su di loro, com'è fatto un albero di directory e simili questioni di interfaccia. Agli sviluppatori interessa il modo in cui sono memorizzati file e directory e com'è gestito lo spazio su disco.

### **Layout del file system**

I file system sono memorizzati sul disco. La maggior parte dei dischi è suddivisibile in una o più partizioni, con file system indipendenti su ciascuna.

Il Settore 0 del disco è chiamato MBR (master boot record) ed è usato per avviare il computer. La fine dell'MBR contiene la tabella delle partizioni dove sono indicati l'inizio e di fine di ogni partizione. Una della partizioni della tabella è indicata come attiva.

Quando il computer è avviato, il BIOS legge ed esegue il MBR. MBR localizza la partizione attiva, legge ed esegue il primo blocco (blocco di boot o boot block) sulla partizione. Il programma nel blocco di boot carica il sistema operativo contenuto nella partizione. Ogni partizione inizia con un blocco di boot anche se non contiene un sistema operativo avviabile perché ne potrebbe contenere uno in futuro.

Anche se in generale il contenuto della partizione cambia da file system a file system uno schema generale è: Il blocco di boot contiene tutti i parametri fondamentali relativi al file system ed è letto in memoria quando il computer è avviato o il file system viene usato per la prima volta. Le informazioni tipiche nel blocco di boot includono un numero magico che identifica il tipo di file system, il numero di blocchi nel file system e altre informazioni amministrative chiave.

### **Implementazione dei file**

Una questione importante nella memorizzazione di un file è tenere traccia di quali blocchi del disco siano associati a ogni file. A seconda dei diversi sistemi operativi sono usati metodi diversi.

## **Allocazione contigua**

Lo schema di allocazione più semplice è quello di memorizzare ciascun file come blocchi adiacenti del disco. Poiché la dimensione del file sarà difficilmente multiplo della dimensione del blocco, si crea un piccolo spreco di spazio interno all'ultimo blocco (così come accadeva per le pagine in memoria).

Vantaggi:

Semplice da realizzare: è sufficiente conoscere l'indirizzo del disco del primo blocco e il numero dei blocchi del file.

Alte prestazioni: l'intero file può esser letto dal disco con una singola operazione (il tempo di seek e la latenza domina il tempo di accesso al disco).

Svantaggi:

Piccolo spreco di spazio nell'ultimo blocco.

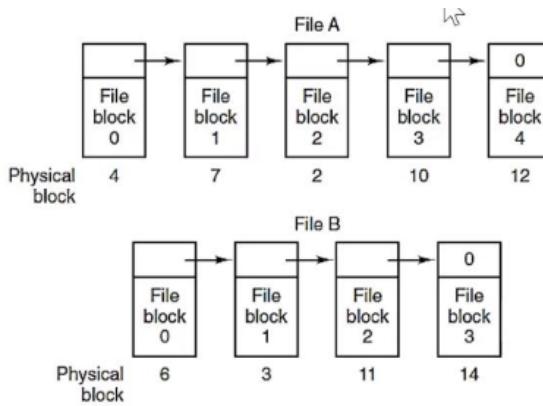
Frammentazione a causa della rimozione dei file (gli spazi vuoti tra file non riescono ad essere impiegati per i nuovi).

L'allocazione contigua è stata utilizzata inizialmente con i dischi magnetici in ragione alle loro caratteristiche. Il sistema è stato abbandonato perché all'atto della creazione del file era necessario specificarne la dimensione massima (non sempre nota in anticipo).

Con l'avvento dei CD-ROM, dei DVD e altri supporti ottici l'allocazione contigua venne nuovamente usata. In questi casi si parla di "cicli e ricicli della storia dell'informatica" per indicare che anche con l'incremento tecnologico, in modo bizzarro e inaspettato, algoritmi o metodi divenuti ormai obsoleti ritornino ad essere attuali.

## **Allocazione a liste concatenate**

Il secondo metodo per memorizzare i file è utilizzare una lista concatenata di blocchi del disco (non necessariamente adiacenti). La prima parola di ogni blocco è usata come puntatore al successivo. Il resto del blocco è per i dati.



### Vantaggi:

Possono essere utilizzati tutti i blocchi del disco (non c'è spazio libero inutilizzato tra i file).

Per ciascuna entry nelle directory è sufficiente memorizzare solo l'indirizzo del primo blocco del disco.

La lettura dei file in modo sequenziale è facile ma non può essere svolta in un sol colpo come prima.

### Svantaggi:

Piccolo spreco dell'ultimo blocco.

L'accesso casuale è estremamente lento: prima di arrivare ad un blocco occorre accedere ai puntatori dei blocchi che lo precedono.

Poiché la dimensione del blocco dati non è una potenza di due, a causa dello spazio riservato al puntatore (pochi byte), ci potrebbe essere una inefficienza dovuta ai programmi che leggono e scrivono quantità di dati espressi in  $2^n$ .

## Allocazione con tabella in memoria

Entrambi gli svantaggi dell'allocazione a liste collegate possono essere eliminati una tabella in memoria dei puntatori di ogni blocco del disco (FAT File Allocation Table).

In entrambe le figure abbiamo due file. Il file A usa i blocchi 4, 7, 2, 10 e 12, in quest'ordine e il file B usa i blocchi 6, 3, 11 e 14, in quest'ordine.

Le catene dei dati sono terminati con uno speciale terminatore (-1). L'intero blocco è disponibile per i dati e l'accesso casuale è semplice.

Le entry nella directory mantiene il numero di blocco iniziale (un intero). L'intera tabella deve essere in memoria. L'idea della FAT non scala bene con dischi di grandi dimensioni.

Physical block	
0	blocco libero
1	blocco libero
2	10
3	11
4	7
5	blocco libero
6	3
7	2
8	blocco libero
9	blocco libero
10	12
11	14
12	-1
13	blocco libero
14	-1
15	blocco libero

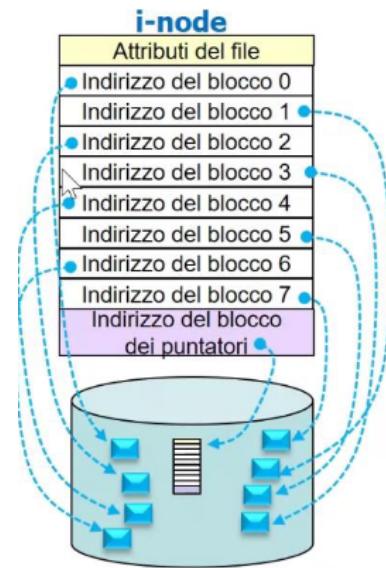
File A: blocks 2, 3, 4, 7, 10, 11  
File B: blocks 5, 6, 8, 9, 12, 14

## I-node

Per ogni file viene creata una struttura dati chiamata i-node (index-node), che contiene gli attributi e gli indirizzi dei blocchi del file.

Poiché l'i-node ha un numero fisso di posizioni, per consentire l'incremento dei blocchi del file, l'ultima cella è riservata ad un blocco simile.

L'i-node è caricato in memoria solo quando il file viene aperto (la sua dimensione è quindi più piccola di una FAT). Mentre la FAT è proporzionale alla dimensione del disco, la dimensione dell'i-node dipende solo dal numero massimo di file che possono essere aperti.



## Implementazione delle directory

Al momento dell'apertura di un file, il sistema operativo usa il path name per localizzare la voce della directory. Questa fornisce: l'indirizzo del primo blocco del disco (allocazione contigua e gli schemi di lista collegata) o il numero dell'i-node.

La directory associa il nome del file, gli attributi e il puntatore ai dati. Nel caso dell'i-node gli attributi possono essere memorizzati nell'i-node stesso.

Tutti i sistemi operativi moderni supportano nomi di file di lunghezza variabile. Si può definire una lunghezza massima (255 caratteri) all'interno di ciascuna voce nello spazio delle directory.

Questa soluzione è semplice da realizzare ma spreca spazio nella directory poiché solo alcuni file hanno nomi molto lunghi.

Le voci della tabella delle directory può contenere due parti:

Una di lunghezza fissa per gli attributi del file.  
Una di lunghezza variabile per il nome del file

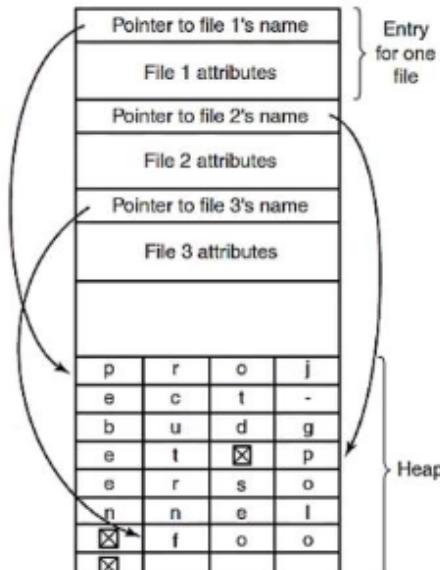
Lo svantaggio di questo metodo è che quando un file viene cancellato, si crea un vuoto di dimensioni variabile (frammentazione), lo spazio si può compattare perché è internamente in memoria.

Se una voce di directory è nella congiunzione di due pagine, la lettura del nome del file può causare un page fault.

Entry for one file	File 1 entry length															
	File 1 attributes															
	p	r	o	j												
	e	c	t	-												
	b	u	d	g												
	e	t	<input checked="" type="checkbox"/>													
File 2 entry length																
File 2 attributes																
<table border="1"> <tbody> <tr><td>p</td><td>e</td><td>r</td><td>s</td></tr> <tr><td>o</td><td>n</td><td>n</td><td>e</td></tr> <tr><td>l</td><td><input checked="" type="checkbox"/></td><td></td><td></td></tr> </tbody> </table>					p	e	r	s	o	n	n	e	l	<input checked="" type="checkbox"/>		
p	e	r	s													
o	n	n	e													
l	<input checked="" type="checkbox"/>															
File 3 entry length																
File 3 attributes																
<table border="1"> <tbody> <tr><td>f</td><td>o</td><td>o</td><td><input checked="" type="checkbox"/></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </tbody> </table>					f	o	o	<input checked="" type="checkbox"/>								
f	o	o	<input checked="" type="checkbox"/>													

Un'altra soluzione è di fare tutte le voci di directory di lunghezza fissa e tenere i nomi dei file in memoria heap alla fine dello spazio della directory. L'eliminazione di un file, crea uno spazio utile all'inserimento di un altro poiché le voci hanno tutte la stessa dimensione. Possono ancora verificarsi page fault durante l'accesso ai nomi dei file.

Per velocizzare la ricerca all'interno di directory estese (più di 100 file) si può utilizzare una tabella hash per ogni directory. Le tabelle hash hanno tempo di accesso velocissimi, ma lo svantaggio di una maggiore complessità nella gestione. Un differente approccio per



accelerare le ricerche è di utilizzare una memoria cache.

## File condivisi

Quando parecchi utenti lavorano insieme a un progetto spesso hanno bisogno di condividere i file. Risulta comodo per un file condiviso apparire nelle diverse directory degli utenti.

Il file system è ora un grafo aciclico orientato o DAG (directed acyclic graph), piuttosto che un albero.

La condivisione dei file è comoda, ma introduce anche dei problemi.

Se le directory contengono realmente gli indirizzi del disco, quando il file viene collegato deve essere fatta una copia.

Se un utente accoda qualcosa al file, i nuovi blocchi saranno visibili solo nella directory dell'utente che ha fatto l'append. I cambiamenti non saranno visibili all'altro utente, rendendo inutile così lo scopo della condivisione. Questo problema è risolvibile in due modi.

Unix: I blocchi del disco non sono elencati nelle directory, ma in una piccola struttura dati associata al file (l'i-node).

Link simbolico: il collegamento crea un nuovo file di tipo link che permette al sistema operativo di accedere al file sorgente.

Ciascuno di questi metodi ha i suoi svantaggi.

Nel primo metodo, quando B collega il file di C, l'i-node registra C come proprietario del file e il contatore dei riferimenti al file a 2 (Count). Se ora C rimuove il file, se si cancella l'i-node, B avrà una voce che punta ad un i-node non valido oppure, se l'i-node è stato riassegnato ad un altro archivio, ad un file errato. L'unica cosa possibile è cancellare la voce dalla directory di C e mantenere l'i-node in C (con Count = 1 e Owner = C). B è l'unico utente che ha una voce di directory per un file di proprietà di C.

Il secondo metodo, non si presenta questo problema perché solo il vero proprietario ha un puntatore per l'i-node. Quando il proprietario rimuove il file, viene distrutto. I collegamenti simbolici richiedono più tempo di gestione. Il file contenente il path deve essere letto, quindi il percorso analizzato fino a raggiungere l'i-node. Tutte queste attività possono richiedere accessi supplementari al disco. Inoltre è richiesto un i-node

aggiuntivo per ogni link simbolico. I link simbolici hanno il vantaggio che possono essere utilizzati per collegare facilmente file che risiedono su macchine dislocate ovunque nel mondo. I link (simbolici o hard), a causa della presenza di più voci nelle directory, favoriscono l'eccesso di copia durante le operazioni di duplicazione o di backup.

## File system basati su log strutturati

I cambiamenti nella tecnologia hanno spinto l'evoluzione dei file system: le CPU diventano sempre più veloci, i dischi, le memorie e le cache sempre più grandi ed economici. L'unico parametro a non migliorare considerevolmente è il tempo di ricerca dei dischi (per quelli meccanici).

Per questa ragione è stato progettato un nuovo tipo di file system: l'LFS (log-structured file system).

L'idea è di soddisfare la maggior parte delle lettura direttamente dalla cache del file system (senza avere accesso al disco). La maggior parte degli accessi al disco sarà in scrittura. Alcuni sistemi effettuano piccole scritture che sono altamente inefficienti: una scrittura da 50 µs impiega 10 ms per il tempo di seek e 5 ms per il ritardo di rotazione.

I progettisti dell'LFS decisero di reimplementare il file system UNIX a partire da queste considerazioni, in modo da raggiungere la larghezza di banda del disco.

L'idea di base è di strutturare l'intero disco come un file di log, tutte le scritture sono registrate nella memoria e raccolte in un unico segmento che verrà poi appeso al log sul disco.

Un segmento singolo può contenere i-node, blocchi di directory e blocchi dati, tutti mischiati insieme. All'inizio di ciascun segmento si trova un segmento di sommario, che indica che cosa si trova nel segmento. Se si riesce a far stare il segmento medio intorno al MB, può essere sfruttata l'intera larghezza di banda del disco.

Per facilitare la ricerca degli i-node, divenuto non più accessibile con l'i-number, si tiene una loro mappa, indicizzata per i-number. La mappa è sia sul disco sia nella cache, così gli da mantenere in memoria gli i-node più usati.

Se il sistema continua a scrivere i segmenti nel file di log, si satura il disco. I segmenti possono contenere blocchi che sono cambiati, per esempio per effetto di una modifica del contenuto di un file o di una sua cancellazione.

Così l'LFS ha un thread cleaner (pulitore) che esegue una scansione circolare del log per compattarlo.

Le misurazioni mostrano che l'LFS sorpassa un classi file system di UNIX di un ordine di grandezza sulle piccole scritture, mentre ha prestazioni uguali o migliori sulle operazioni di lettura e sulle operazioni di scrittura estese.

## File system basati su journaling

I file system basati su log strutturato sono un'idea interessante, ma non sono diffusi a causa della loro incompatibilità con i file system esistenti. L'idea di base è quella di tenere un diario di ciò che il file system sta per fare prima che lo faccia, in modo che in caso di crash si possa recuperare il lavoro. Tali file system sono chiamati file system journaling.

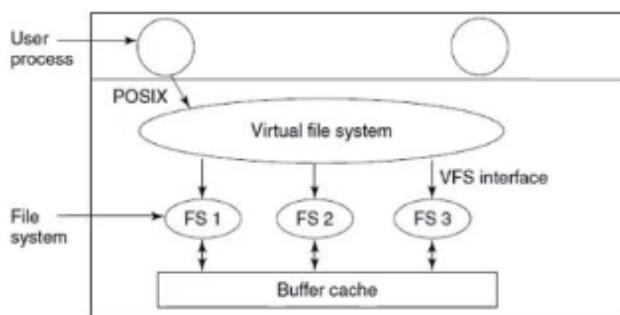
Affinché il journaling funzioni occorre che le operazioni nel log devono essere idempotenti: possono essere ripetute tutte le volte che serve senza produrre effetti collaterali. Per migliorare l'affidabilità, un file system può introdurre il concetto di transazione atomica, più operazioni possono essere raggruppate in un'unità unica e svolta in modo atomico.

## File system virtuali

I sistemi operativi sono in grado di gestire diversi file system.

Un sistema Windows potrebbe avere un file system principale NTFS ma anche diversi drive FAT 32 o FAT 64. Windows gestisce questi file system identificando ciascuno con una differente lettera di unità (C:, D:). Non c'è alcun tentativo di integrare file system eterogenei in un solo unificato.

Al contrario, tutti i moderni sistemi UNIX cercano di integrare più file system in una sola gerarchia sotto root. I sistemi UNIX usano il Virtual File System (VFS) la parte comune dei file system è posta su un livello astratto più alto, mentre il codice specifico su un livello più basso.



Con questo approccio è semplice aggiungere nuovi file system:

I progettisti prima analizzano le tipologie di funzioni che il VFS può ricevere e poi scrivono il layer sottostante che permette al VFS di soddisfarle.

Se il file system esiste già, occorre realizzare delle funzioni di wrapping verso il VFS.

## 5. Input Output

Tra i compiti del sistema operativo troviamo anche la gestione dei dispositivi di I/O:

- Controllare i dispositivi
- Inviare comandi ai dispositivi
- Catturare gli interrupt dei dispositivi
- Gestire gli errori provenienti dai dispositivi
- Fornire un'interfaccia semplice e device-independent per la gestione dei dispositivi

La parte di software responsabile della gestione dell'I/O rappresenta una frazione significativa dell'intero sistema operativo.

### 5.a Principi hardware

#### Dispositivi di I/O

I dispositivi di I/O possono essere suddivisi in due categorie:

Dispositivi a blocchi: Dischi fissi, CD-ROM, penne USB, ecc... Gestiscono informazioni in blocchi di dimensioni fisse, ognuno con il proprio indirizzo. I trasferimenti avvengono con uno o più blocchi (consecutivi).

Dispositivo a caratteri: Stampanti, interfacce di rete, mouse, ecc... Gestiscono flussi di caratteri, a prescindere da qualsiasi struttura. Non sono indirizzabili e non hanno alcuna operazione di ricerca.

Ci sono altri dispositivi che non sono indirizzabili a blocchi e non generano flussi di caratteri, troviamo:

I clock, producono interrupt a intervalli ben definiti. La RAM della scheda video, è una memoria dentro il controller video che rappresenta con una bitmap l'immagine sullo schermo.

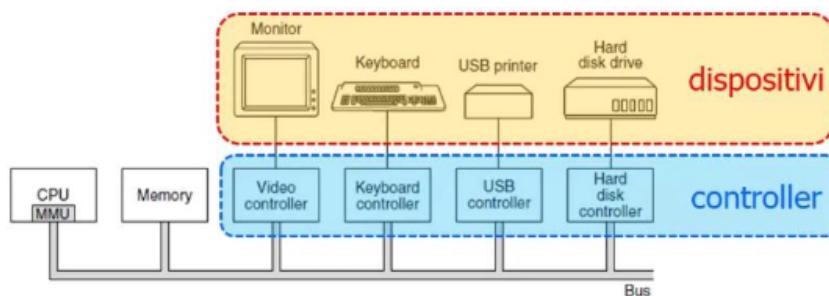
I dispositivi di I/O possono avere velocità molto diverse tra loro, quindi è il software che deve cercare di sfruttare al meglio le potenzialità del dispositivo durante il trasferimento

dati. La maggior parte di questi tende a divenire sempre più veloce con il passare degli anni.

## Controller dei dispositivi

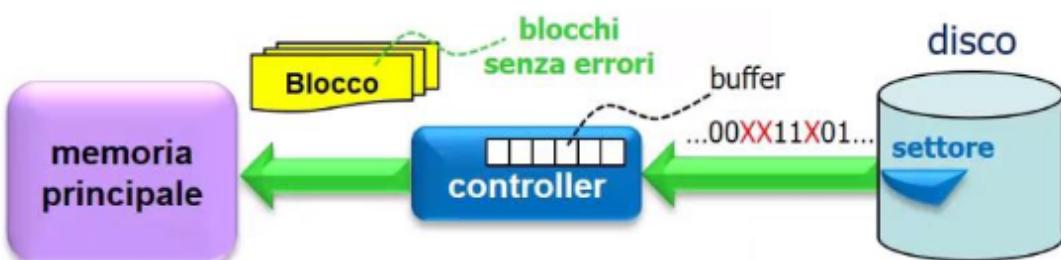
I dispositivi di I/O consistono tipicamente di una componente meccanica e di una elettronica.

La componente elettronica è detta controller del dispositivo (device controller) o adattatore (adapter). Sui personal computer è un chip sulla scheda madre o una scheda inseribile in un alloggiamento di espansione (PCI). La parte meccanica è il dispositivo stesso.



Molti controller possono gestire più dispositivi identici.

Se l'interfaccia fra il controller e il dispositivo è standard, possono esistere due mercati separati di produttori. Per esempio, un disco fornisce un flusso di bit (preamble, dati del settore e ECC). Il controllore accoda in un buffer lo stream di bit, esegue l'eventuale correzione agli errori e spedisce i blocchi nella memoria principale.



Quando la CPU vuole leggere una parola (dalla memoria o da una porta di I/O):

- Pone l'indirizzo sull'address bus;
- Asserisce il segnale READ sul bus di controllo;

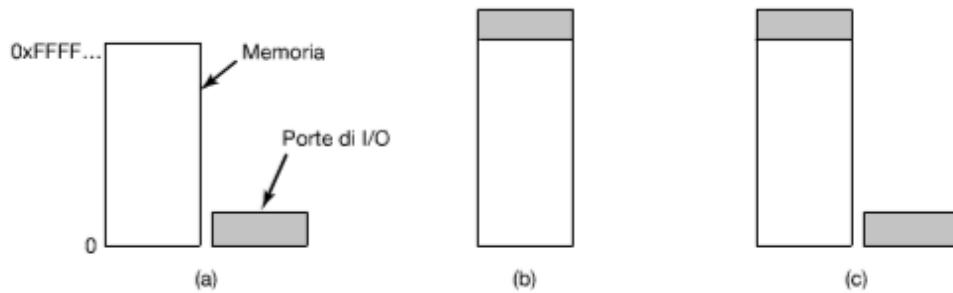
- Asserisce un segnale per dire da dove si svolge la lettura (I/O o memoria);
- Attende la risposta della memoria o del dispositivo.

## Input/output mappato in memoria

Ogni controller ha alcuni registri di controllo usati per le comunicazioni con la CPU. Scrivendo questi registri il sistema operativo comanda al dispositivo di inviare dati, accettarli, accendersi o spegnersi, ... Leggendo questi registri, il sistema operativo capisce quale sia lo stato del dispositivo o se è disposto ad accettare un nuovo comando. Oltre ai registri di controllo, molti dispositivi hanno un buffer di dati su cui il sistema operativo può scrivere e leggere.

La CPU comunica con il dispositivo in tre diverse modalità:

- Come un vero e proprio dispositivo (port-mapped I/O o I/O isolato);
- Come parte della memoria (memory-mapped I/O);
- Entrambi (hybrid-mapped I/O).



### Port-mapped I/O

Comunica come un vero e proprio dispositivo di I/O (8 o 16 bit). L'insieme di tutte le porte di I/O forma lo spazio delle porte di I/O, che è protetto in modo che i programmi utente non possano accedervi (lo può fare solo il sistema operativo).

Le istruzioni utilizzate dalla CPU per leggere/scrivere i registri di controllo all'indirizzo PORT sono:

- IN REG,PORT: la CPU legge il registro di controllo PORT e salva il risultato nel registro REG;
- OUT PORT,REG: la CPU scrive il contenuto di REG in un registro di controllo del dispositivo.

Poiché le informazioni sono mappate su indirizzi separati dalla memoria:

- La cache non viene influenzata dai dati che transitano verso/dal dispositivo

(contrariamente l'effetto sarebbe disastroso).

- I dispositivi non devono esaminare i riferimenti di memoria per capire quando rispondere.

D'altro canto questo schema presenta i seguenti svantaggi:

- Per leggere e scrivere i registri di controllo del dispositivo sono necessarie istruzioni dedicate assembly IN e OUT (non esistono istruzioni C o C++ che possano realizzarle) . L'utilizzo di queste istruzioni aggiunge overhead al controllo dell'I/O.
- È necessario un meccanismo di protezione speciale per controllare lo svolgimento delle operazioni di I/O da parte dei processi utente.
- A differenza dell'I/O mappato in memoria che vede i registri di controllo del device come locazioni di memoria (e quindi variabili di un qualsiasi programma C/C++), occorre un passaggio ulteriore per spostare il contenuto dal registro della CPU alla memoria.
- I driver di controllo dei dispositivi non possono essere scritti utilizzando esclusivamente il linguaggio C (o C++) .
- Il sistema operativo non riesce ad assegnare in modo semplice e dinamico i dispositivi ai processi utente.

## **Memory-mapped I/O**

Il secondo approccio, consiste nel mappare tutti i registri di controllo nella memoria. A ciascun registro di controllo viene assegnato un indirizzo di memoria univoco non più utilizzabile come spazio di memoria principale.

Questo schema presenta i seguenti vantaggi:

- I registri di controllo dei dispositivi sono delle variabili in memoria e possono essere gestiti da un programma C/C++.
- Il sistema operativo deve semplicemente evitare di mettere lo spazio di indirizzo contenente i registri di controllo nello spazio di indirizzamento virtuale di qualsiasi utente.
- Ogni dispositivo ha i suoi registri di controllo su una pagina diversa dello spazio di indirizzamento, in questo modo il sistema operativo può assegnare dinamicamente i dispositivi includendo le pagine desiderate.
- È sufficiente una sola istruzione di test in memoria per controllare lo stato del dispositivo.
- I driver di controllo possono essere scritti in C/C++.

D'altro canto però ha anche i seguenti svantaggi:

- Il caching di un registro di controllo dispositivo produce effetti disastrosi. L'hardware deve avere la capacità di disabilitare la cache sulle pagine di IO.

- Se c'è un solo spazio di indirizzamento la memoria e tutti i dispositivi di IO devono esaminare gli indirizzi che passano sull'address bus per vedere a quali rispondere. Semplice su un'architettura a singolo bus.
- Se esiste un collegamento dedicato CPU-Memoria, i dispositivi di IO non vedono gli indirizzi di memoria sul collegamento e quindi non hanno modo di rispondergli.

## Hybrid

Il terzo approccio è uno schema ibrido con accesso indipendente per:

- Buffer dati (memory-mapped IO)
- Registri di controllo (port-mapped IO).

Questa architettura viene usata anche dal Pentium.

## Direct memory access (DMA)

La CPU accede ai controller dei dispositivi per lo scambio dei dati indipendentemente dal tipo di mappatura dell'IO. La CPU può richiedere dati al controller di I/O un byte alla volta, sprecando il tempo della CPU. Si usa una diversa tecnica chiamata DMA (direct memory access - accesso diretto alla memoria) che permette alla CPU di disinteressarsi del trasferimento dati e di svolgere altre attività in parallelo.

Il sistema operativo può usare il DMA solo se l'hardware ha un controller DMA. I controller possono avere un proprio DMA oppure si può utilizzare un controller DMA (scheda madre) per più dispositivi. Il controller DMA è un master del bus, quindi ha accesso al bus di sistema indipendentemente dalla CPU.

Esso contiene molti registri che possono essere scritti e letti dalla CPU, tra cui un registro degli indirizzi di memoria, un registro dei conteggi dei byte e uno o più registri di controllo. I registri di controllo specificano le porte di I/O da usare, la direzione del trasferimento (lettura dal dispositivo di I/O o scrittura verso il dispositivo di I/O), l'unità di trasferimento (un byte o una parola alla volta) e il numero di byte da trasferire alla volta.

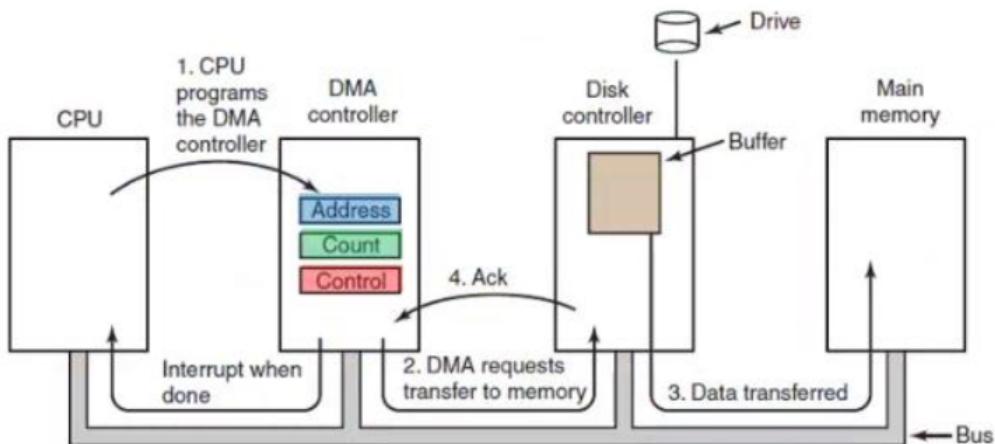
Quando si usa il DMA la procedura di lettura dal disco è la seguente.

Il controller del disco legge l'intero blocco (uno o più settori) dal drive in modalità seriale: bit a bit. Quando il blocco è stato caricato tutto nel buffer interno al controllore, si controlla che non si siano verificati errori. Il controllore genera poi un interrupt. Il sistema operativo legge con una ISR il blocco dal buffer byte/parola alla volta e attraverso una MOV lo sposta in memoria.

1. La CPU programma il controller DMA impostando i suoi registri (Address, Count e Control) in modo che sappia che cosa trasferire e dove. Poi dice al controller di leggere i dati nel suo buffer interno.
2. Una volta terminato il caricamento del buffer, il controller DMA, preso il controllo del bus come master, chiede al controller del disco la lettura del buffer per il trasferimento in memoria.
3. La scrittura in memoria avviene con un ciclo di bus standard. Quando l'operazione di scrittura è completata, il controller del disco manda un segnale di conferma sul controller DMA, sempre tramite il bus.
4. Il controller DMA incrementa poi l'indirizzo di memoria da usare e diminuisce il conteggio dei byte.

Se questo conteggio è ancora maggiore di 0, allora i passi da 2 a 4 vengono ripetuti, finché il conteggio arriva a 0.

A quel punto il controller DMA manda un interrupt alla CPU per avvisarla che il trasferimento è completato. Quando l'esecuzione passa al sistema operativo, non deve copiare il blocco del disco in memoria, dato che è già lì.



Molti bus (e anche controller DMA) possono funzionare in due modalità:

- Una parola alla volta: il controller DMA richiede il trasferimento di una parola. Se la CPU vuole anch'essa il bus, il controller DMA deve aspettare. Il meccanismo è chiamato cycle stealing perché il controller DMA spia e ruba un ciclo di bus occasionale dalla CPU rallentandola leggermente.

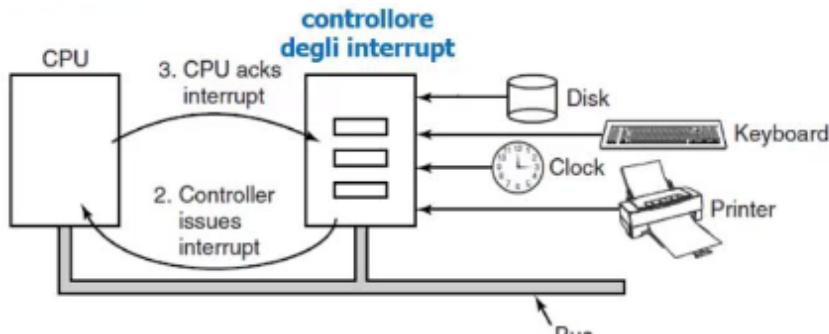
- Un blocco alla volta (modalità burst): il controllore DMA dice al dispositivo di acquisire il bus, emettere una serie di trasferimenti, quindi rilascia il bus. È più efficiente della precedente per quanto riguarda la durata del trasferimento ma corre il rischio di bloccare la CPU e gli altri dispositivi per un periodo di tempo dipendente dalla dimensione del trasferimento.

## Interrupt rivisitati

Quando un dispositivo di I/O finisce il lavoro che gli è assegnato, causa un interrupt, che invia al controller degli interrupt (o arbitro) della scheda madre, che decide poi che cosa fare:

Se nessun altro interrupt è in sospeso, viene elaborata immediatamente l'interruzione.

Se ce n'è un altro in corso o un altro dispositivo ha inviato un segnale di richiesta si decide in base alla priorità assegnata ai dispositivi.



Il controller segnala alla CPU l'interruzione, attende il riconoscimento e pone un numero sulle linee di indirizzo per specificare quale dispositivo ha inviato l'interruzione.

Il numero è usato come indice in una tabella (vettore degli interrupt) che restituisce il nuovo valore per il program counter: che punterà l'inizio della routine di servizio dell'interrupt (ISR).

L'ISR per comunicare al controller degli interrupt che può accettare altre istruzioni, scrive un valore in una delle porte di I/O del controller stesso.

Prima di avvisare la ISR l'hardware salva lo stato della CPU (Pc, registri, stack, ...) Uno dei problemi è dove salvare le informazioni.

Nei registri interni: Il controller degli interrupt non può essere avvisato finché tutte le informazioni rilevanti non sono state lette (una seconda interruzione potrebbe

sovrascrivere i registri interni). Questa strategia porta a lunghi tempi morti quando gli interrupt sono disabilitati e ad una possibile perdita di interrupt e di dati.

Nello stack: Innanzitutto: in quale stack? Quello del processo utente o uno nel kernel? L'uso dello stack nel kernel è migliore di quello dei processi utente dal punto di vista dell'affidabilità dei riferimenti alle pagine (SP). Tuttavia, il passaggio alla modalità kernel può richiedere tempo di CPU a causa del cambio di contesto nella MMU.

## Interrupt precisi e imprecisi

La maggior parte delle moderne CPU usano le pipeline e sono spesso superscalari (unità funzionali parallele). Nei sistemi più vecchi, al termine dell'esecuzione di ogni istruzione l'hardware controllava se ci fosse un interrupt in sospeso. Se era così, il Program Counter e la parola di stato del programma (PSW) venivano inseriti nello stack ed eseguita la ISR. Al termine della ISR venivano ripristinati dallo stack PC e PSW. Tutte le istruzioni eseguite prima dell'interrupt erano state eseguite completamente, ciò non è vero nelle attuali architetture.

Che cosa accade se si verifica un interrupt mentre la pipeline è piena (caso frequente)? Molte istruzioni si troverebbero in un diverso stato di esecuzione. Quando si verifica l'interrupt, il valore del PC non determina il confine netto tra le istruzioni eseguite e quelle non eseguite. Infatti, molte istruzioni potrebbero essere state eseguite parzialmente.

Su una macchina superscalare la situazione è ben più complessa, le istruzioni possono essere suddivise in micro-operazioni eseguite in un ordine che dipende dalla disponibilità delle unità funzionali e dei registri. Un interrupt che lascia la macchina in uno stato ben definito è detto interrupt preciso e ha quattro proprietà:

1. Il PC (program counter) è salvato in un posto conosciuto;
2. Tutte le istruzioni eseguite prima di quella puntata dal PC sono state completamente eseguite;
3. Nessuna istruzione oltre a quella puntata dal PC è stata eseguita;
4. Lo stato di esecuzione dell'istruzione puntata dal PC è conosciuto.

Un interrupt che non sottostà a queste specifiche è detto interrupt impreciso.

Macchine con interrupt imprecisi di solito delegano il sistema operativo per capire come gestire la situazione e riversano sullo stack grandi quantità di dati sullo stato interno. Il

codice necessario a riavviare la macchina è estremamente complicato. Inoltre gli interrupt e il ripristino dello stato da un interrupt sono operazioni lente.

Questo genera una situazione paradossale in cui talvolta le veloci CPU superscalari non sono adatte per gestire sistemi real-time a causa di interrupt lenti.

## **5.b Principi del software**

Prima analizzeremo gli obiettivi di un software di I/O. Poi ci dedicheremo ai diversi metodi per eseguire un I/O.

### **Obiettivi del software di I/O**

Indipendenza dal dispositivo: i programmi che gestiscono l'I/O devono poterlo fare indipendentemente dal tipo di dispositivo.

Denominazione uniforme: l'identificatore di un file o di un dispositivo non deve dipendere dal tipo di dispositivo.

Gestione degli errori: gli errori andrebbero gestiti il più possibile a livello hardware (controller, driver, ...). Molti errori (come quelli di lettura dal disco) sono transitori e scompaiono se si ripete l'operazione.

Trasferimenti sincroni/asincroni: i trasferimenti possono essere bloccanti (sincroni) o gestiti con interrupt (non bloccanti, asincroni). A causa della differente velocità, i dispositivi di I/O sono tipicamente asincroni. Tuttavia è molto più facile scrivere programmi utenti con primitive bloccanti (es. `read()`), quindi sta al SO dare al programmatore l'illusione di utilizzare chiamate sincrone quando in realtà i dispositivi sono trattati con primitive asincrone.

Bufferizzazione: i dati che vengono da un dispositivo non possono essere memorizzati direttamente nella loro destinazione finale e dovrebbero essere bufferizzati. Per esempio, il sistema operativo non sa a chi è destinato un pacchetto che arriva dalla rete finché non è assemblato ed esaminato.

Condivisione: alcuni dispositivi di I/O (es. disco) sono risorse condivise e possono essere utilizzati da molti utenti contemporaneamente. Altri dispositivi, come ad esempio le unità a nastro, sono dedicate ad un singolo utente: nessuno può utilizzare il dispositivo finché l'utente non ha finito.

I dispositivi possono essere gestiti con tre diverse tecniche:

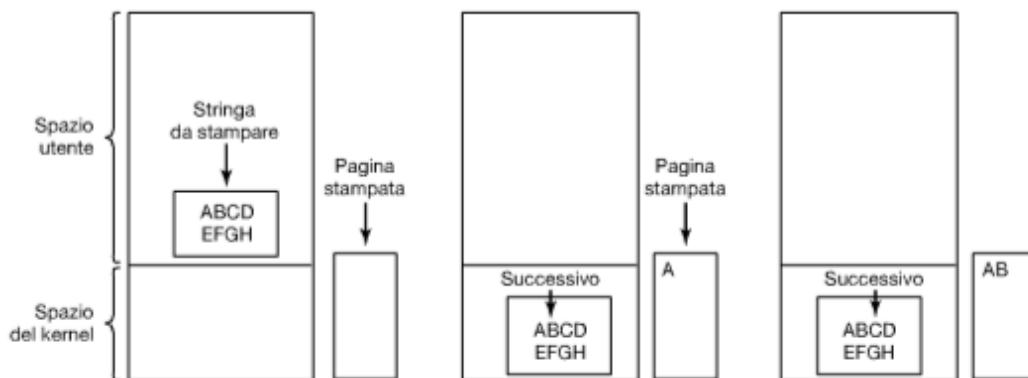
## I/O programmato

L'I/O programmato è la tecnica più semplice perché la CPU svolge tutto il lavoro. Si consideri un processo utente che voglia stampare un certa stringa dalla stampante.

Il processo tenta di aprire una connessione alla stampante facendo una chiamata di sistema, se la stampante è occupata o fuori linea, questa chiamata fallirà, restituendo un codice di errore o bloccandosi finché la stampante non sarà disponibile, a seconda del sistema operativo e dei parametri della chiamata.

Quando la stampante torna disponibile informa il SO.

Il sistema operativo copia la stringa nello spazio kernel (array p), e controlla se la stampante è ancora disponibile, altrimenti aspetta. A quel punto il sistema operativo invia un carattere alla volta al registro dati della stampante.



La CPU esegue il polling della stampante per controllare se è pronta ad accettare un altro carattere (polling o busy waiting).

L'I/O programmato è semplice ma ha lo svantaggio di utilizzare il tempo della CPU finché tutte le operazioni di I/O non sono finite.

Generalmente, il busy waiting è una prassi che andrebbe limitata ai soli casi di I/O veloce.

Esistono metodi migliori.

## I/O guidato dagli interrupt

Supponiamo di stampare su una stampante senza buffer: stampa un carattere alla volta. Se la stampante può stampare per esempio 100 caratteri/s, ogni carattere impiega 10

ms per essere stampato. Ciò significa che la CPU si fermerà in un ciclo di inattività di 10 ms.

Per permettere alla CPU di fare qualcos'altro nell'attesa che la stampante sia pronta è possibile usare gli interrupt. Avvenuta la chiamata di sistema per la stampa della stringa, il buffer è copiato nello spazio del kernel e il primo carattere inviato alla stampante. A questo punto la CPU richiama lo scheduler e viene eseguito un altro processo. Il processo che ha richiesto la stampa della stringa è bloccato finché non è stampata l'intera stringa.

Quando la stampante ha stampato il carattere ed è pronta ad accettare il successivo, genera un interrupt. Questo interrupt ferma il processo attuale e salva il suo stato. Poi è eseguita l'ISR della stampante.

Se non ci sono più caratteri da stampare, il gestore degli interrupt sblocca l'utente. Altrimenti, esegue l'output del carattere successivo, avvisa l'interrupt e ritorna al processo che stava eseguendo appena prima dell'interrupt, continuando da dove era stato lasciato.

## I/O con DMA

Uno svantaggio ovvio dell'I/O guidato dagli interrupt è che avviene un interrupt a ogni carattere. Gli interrupt richiedono tempo, sprecando del tempo di CPU. Una soluzione è l'uso del DMA che invia i caratteri della stampante uno alla volta, senza l'intervento della CPU. Ottima soluzione se si devono stampare molti caratteri e gli interrupt sono lenti.

Questa strategia richiede sì un hardware speciale (il controller DMA), ma consente alla CPU di svolgere altre attività durante l'I/O.

Poiché il controller DMA è generalmente molto più lento della CPU principale potrebbero esserci delle situazioni (rare) in cui conviene utilizzare le altre due tecniche.

## 5.c Livello software

Il software di I/O è generalmente organizzato in quattro livelli.

Ogni strato ha una funzione ben definita e definisce dei servizi agli strati adiacenti attraverso la sua interfaccia. La funzionalità e le interfacce sono diverse da sistema a sistema.



## Driver delle interruzioni

Anche se l'I/O programmato è talvolta utile, si utilizzano diffusamente gli interrupt per gestire i dispositivi di I/O.

Gli interrupt dovrebbero rimanere così nascosti in profondità che il sistema operativo potrebbe quasi ignorarne l'esistenza. Il miglior modo per nasconderli è fare in modo che il driver che avvia un'operazione di I/O si blocchi finché l'I/O non è terminato e si verifica l'interrupt.

Il driver può bloccarsi facendo per esempio una down su un semaforo, una wait su una variabile condizione o qualcosa di simile.

Quando si verifica l'interrupt la procedura fa il necessario per gestirlo, e sbloccherà il processo che l'ha fatta partire, mandando un signal su una variabile o up su di un semaforo. In tutti i casi, l'effetto di interruzione sarà un processo che era stato precedentemente bloccato ora sarà in grado di riprendere l'esecuzione.

Questo modello funziona meglio se i driver sono strutturati come processi kernel, con i loro stati, stack e contatori di programma personali.

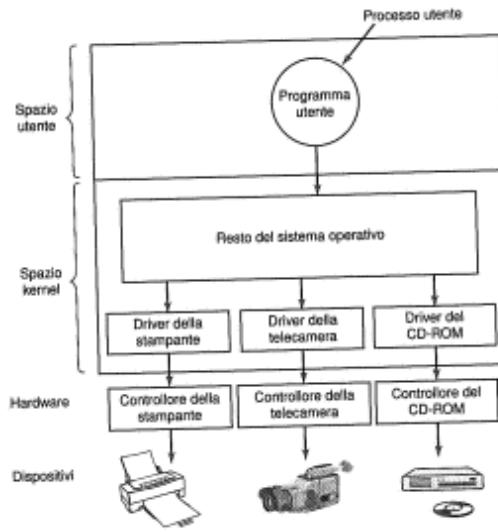
## Driver dei dispositivi

Per controllare un dispositivo di IO è necessario un codice (driver di periferica) specifico del dispositivo, scritto dal produttore del dispositivo. I produttori di dispositivi generalmente forniscono driver per la gran parte dei sistemi operativi.

Per accedere ai registri del controller, il driver di periferica deve essere parte del kernel nel sistema operativo. È possibile costruire driver eseguiti nello spazio utente, utilizzando le chiamate di sistema. Così avremo un isolamento del kernel e il sistema diventerà altamente affidabile.

I progettisti dei sistemi operativi sanno che pezzi di codice (driver) scritto da terzi saranno installati all'interno della loro architettura. Esiste un modello ben definito di ciò che un driver fa e come deve interagire con il resto del sistema.

I driver di periferica sono normalmente posizionati al di sotto del resto del sistema operativo.



I sistemi operativi di solito classificano driver come:

- Dispositivi a blocchi (dischi) che indirizzano uno o più blocchi di dati in modo indipendente.
- Dispositivi a caratteri (tastiere, stampanti, ...) che gestiscono i flussi di caratteri.

Le caratteristiche più importanti per i driver sono la capacità di:

- essere rientranti (o reentrant): durante l'esecuzione devono aspettarsi di essere chiamati più volte ancor prima che sia completata la precedente chiamata.
- garantire sistemi pluggable a caldo, i dispositivi possono essere aggiunti o rimossi mentre il computer è in funzione.

Anche se non è concesso ai driver di effettuare chiamate di sistema, essi devono interagire con il kernel per allocare o deallocare pagine di memoria fisica, per gestire la MMU, i timer, il controller DMA, quello degli interrupt e molte altre cose.

## Software per l'I/O indipendente dal dispositivo

Il software per la gestione dei dispositivi I/O si compone di una parte specifica e un'altra parte indipendente al dispositivo. Il confine esatto tra i driver e il software indipendente

dal dispositivo dipende dal sistema e dal dispositivo.



La funzione base del software indipendente dal dispositivo è quella di eseguire tutte quelle funzioni di I/O trasversali a tutti i dispositivi e di fornire un'interfaccia uniforme al software a livello utente.

## Interfacciamento uniforme dei driver dei dispositivi

Una questione fondamentale di un sistema operativo è come rendere tutti i dispositivi e i driver più o meno simili tra loro. La modalità di interfacciamento tra driver e sistema operativo deve essere standard per ogni dispositivo. Senza standard l'interfacciamento di un nuovo driver richiede uno sforzo di programmazione enorme.

Il software indipendente dal dispositivo si prende cura di mappare i nomi simbolici dei dispositivi nel driver adatto.

In UNIX tutti i dispositivi hanno:

- Il numero del dispositivo primario (major device number), utilizzato per individuare il driver appropriato.
- Il numero del dispositivo secondario (minor device number), utilizzato come parametro per il driver per specificare l'unità.

In UNIX il nome di dispositivo, come /dev/disk0, specifica l'i-node di un file speciale che contiene i riferimenti al driver del disco (major device number) e all'unità 0 (minor device number).

## Buffering

Il buffering costituisce un problema, sia per i dispositivi a blocchi sia per quelli a caratteri.

Consideriamo un processo che voglia leggere dei dati da un modem. Il processo utente esegue una read() e si blocca in attesa di un carattere. Ogni carattere genera un

interrupt. L'ISR da il carattere al processo utente e si blocca. Questa soluzione è inefficiente perché il processo viene eseguito molte volte per brevi periodi di tempo.

L'aggiunta di un buffer di n caratteri nello spazio utente permette alla ISR di scrivere più caratteri nel buffer finché non è pieno. Solo a quel punto risveglia il processo utente.

Cosa accade se il buffer viene paginato su disco all'arrivo di un carattere?

L'ISR mette i caratteri in un buffer nel kernel, quando è pieno, la pagina che contiene il buffer nello spazio utente viene portata in memoria (se era su disco) e poi è copiato in una sola operazione dalla pagina kernel a quella utente. Cosa succede ai caratteri che arrivano quando il buffer è pieno?

Si usano quindi due buffer (schema a doppio buffer) nel kernel che si scambiano di ruolo: uno per leggere dal modem e l'altro per copiare nello spazio utente. I caratteri che arrivano mentre la pagina non è pronta o il buffer è pieno vengono memorizzati nell'altro. Il primo viene copiato e, poi, svuotato e cambia ruolo: ora può memorizzare i caratteri quando il secondo sarà pieno.

Un'altra soluzione adotta il buffer circolare: un puntatore indica la prossima locazione dove inserire i nuovi dati  $P_{FINE}$ , un altro la prima locazione riempita nel buffer  $P_{INIZIO}$ .

$P_{FINE}$  avanza quando arrivano nuovi dati dal modem;

$P_{INIZIO}$  avanza quando il sistema operativo elimina ed elabora i dati.

Quando il buffer è pieno  $P_{FINE} = P_{INIZIO}$

Il buffering è una tecnica largamente utilizzata, ma occorre far attenzione all'eccesso di copia, da uno spazio all'altro, che rallenta le prestazioni.

## Segnalazione degli errori

Quando si tratta con i dispositivi di IO gli errori sono più frequenti di altri ambiti.

Le classi più comuni di errori sono:

- Errori di programmazione: un processo chiede qualcosa di impossibile.
- Errori fisici del dispositivo (settore rovinato).

Quando si verifica un errore, un software <chiamato> ha varie possibilità:

- Delegare il chiamante (nel caso del driver: il software indipendente dal dispositivo) al trattamento dell'errore.
- Tentare di risolvere l'errore localmente o reiterando più volte l'operazione o ignorando l'errore o terminando il processo chiamante.

## Allocazione/rilascio dei dispositivi dedicati

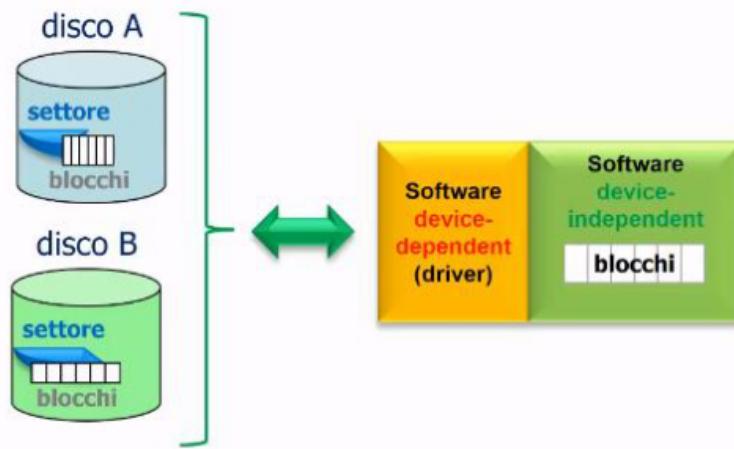
Alcuni dispositivi (es. masterizzatori) possono essere utilizzati da un solo processo per volta, è il SO che gestisce le richieste per l'uso di tali dispositivi.

Possono essere utilizzati due possibili approcci:

- Il processo che vuole utilizzare il dispositivo tenta una open() su un file speciale al dispositivo stesso. Se è libero riesce nell'operazione e blocca l'utilizzo esclusivo del device.
- Si utilizzano i classici meccanismi di sincronizzazione tra processi: il processo che chiede un device occupato va in sleep finché il device non è libero.

## Dimensione dei blocchi device-independent

Dischi differenti possono avere settori di diverse dimensioni, sta al software indipendente dal dispositivo nascondere questo aspetto e fornire una dimensione di blocco uniforme (astrazione).



## Software di I/O del livello utente

La maggior parte del software per la gestione dell'IO è all'interno del sistema operativo, tuttavia una piccola parte consiste di librerie che verranno poi collegate ai programmi utente.

Quando un programma contiene per esempio la chiamata: count = write(fd, buffer, nbytes)

La procedura write() che appartiene ad una libreria nello spazio utente sarà collegata con il programma e contenuta nel programma binario presente in memoria in fase di

esecuzione.

Tutte le procedure e librerie sono chiaramente parte del sistema di gestione dell'IO.

Non tutto il software a livello utente per la gestione dell'IO è costituito dalle procedure nelle librerie, esiste anche il sistema di spooling. Lo spooling è il modo per interagire con i dispositivi di IO dedicati in un sistema multiprogrammato.

Nel caso della stampante esiste un processo demone e una directory speciale (directory di spooling).

- Per stampare un file, un processo scrive il file da stampare nella directory di spooling.
- Successivamente il demone legge i file memorizzati nella directory e li stampa, uno alla volta.

## **5.d Dischi**

### **Hardware dei dischi.**

Tra i principali dispositivi di IO troviamo i dischi:

1. Magnetici (fissi). Le operazioni di lettura o scrittura hanno la stessa velocità
2. Raid. Utilizzati per le configurazioni di alta affidabilità.
3. A stato solido (SSD). Prestazioni eccellenti ed assenza di componenti meccaniche.
4. Ottici (DVD, CD-ROM) Normalmente utilizzati per la distribuzione di programmi.

### **Dischi Magnetici**

La superficie dei dischi è divisa in zone: Quelle esterne hanno più settori rispetto a quelle interne.

La geometria mostrata al SO è diversa da quella fisica, sarà il controller a rimappare le richieste del SO verso il numero reale di settori. Prima di poter utilizzare un disco occorre definirgli un formato: attraverso la formattazione a basso livello. Una serie di tracce concentriche, ciascuna contenente un numero di settori, con brevi spazi tra i settori (intersector gap).

Ogni settore contiene:

Un preambolo, ovvero una sequenza di bit che consente di riconoscere l'inizio del settore che può contenere anche il numero dei cilindri/settori e altre informazioni.

Dati (payload), la cui dimensione viene determinata dal programma di formattazione di

basso livello.

Codice di Correzione dell'errore (ECC), un codice di Hamming oppure un altro codice che permette di correggere errori multipli (codice Reed-Solomon).

## Formattazione del disco

La formattazione del disco ne riduce la dimensione di circa il 15-20% e incide anche sulle prestazioni. Durante la formattazione a basso livello la posizione del settore 0 su ogni traccia è spostato di un certo offset rispetto alla traccia precedente. Questo offset, chiamato pendenza del cilindro, permette di migliorare le prestazioni riducendo gli spostamenti della testina nelle operazioni che coinvolgono più tracce.

La pendenza del cilindro dipende dalla geometria del disco:

- Un disco con RPM da 10k compie un giro in 6ms.
- Se la traccia ha 300 settori, la testina incontra il settore ogni 20 nanosecondi.
- Se il tempo di seek è di 800 nanosecondi, significa che in una seek transitano 40 settori.

Si consideri un controller con un buffer della capacità di un settore che deve leggere due settori consecutivi. Dopo aver letto il primo settore del disco ed eseguito il calcolo dell'ECC, i dati devono essere trasferiti alla memoria principale. Quando la copia della memoria sarà completata, il controllore dovrà aspettare almeno un intero periodo di rotazione prima che arrivi il secondo settore. Questo problema può essere eliminato numerando i settori in modalità interleaved durante la formattazione del disco.

Se il processo di copia è molto lento potrebbe essere necessario il doppio interleaving. Dopo la formattazione a basso livello, il disco viene partizionato: ogni partizione è come un disco separato.

Le partizioni permettono l'esistenza di più sistemi operativi. Nella maggior parte dei computer, il settore 0 contiene il Master Boot Record (MBR) che contiene la tabella delle partizioni ove è scritto il settore di avvio e la dimensione di ciascuna partizione. Questa partizione deve essere considerata come attiva nella tabella delle partizioni.

Il passo finale nella preparazione di un disco è di formattare ad alto livello ciascuna partizione, questa operazione crea:

1. un blocco di avvio
2. una free list o una bitmap per la gestione dello spazio libero
3. una directory principale

#### 4. un file system vuoto

A questo punto il sistema è pronto per essere avviato.

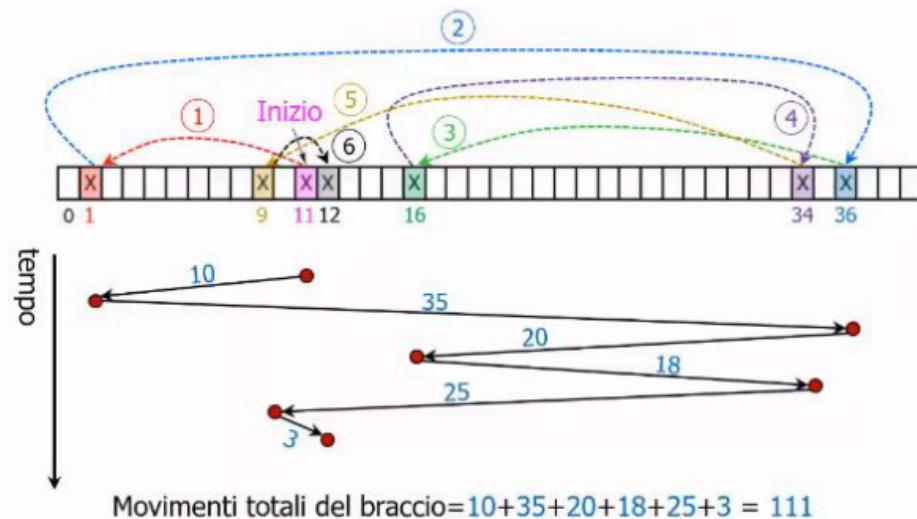
## Algoritmi di scheduling del braccio del disco

Il tempo richiesto per la lettura o la scrittura di un blocco del disco è determinato da tre fattori:

1. Il tempo di ricerca o seek (il tempo per muovere la testina al giusto cilindro);
2. Il ritardo rotazionale (il tempo affinché il settore giusto ruoti sotto la testina);
3. Tempo di trasferimento dei dati.

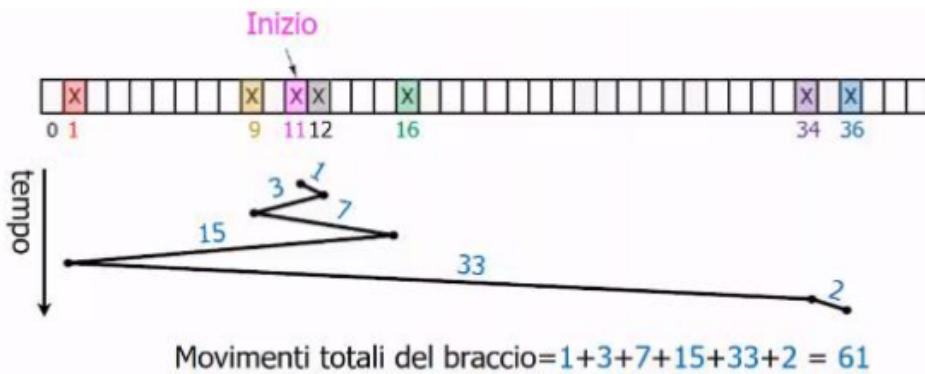
Per la maggior parte dei dischi, il tempo di seek è decisamente maggiore degli altri due. Se il driver del disco accetta una richiesta per volta nel ordine in cui arrivano (first-come, first-served FCFS), poco da fare per ottimizzare il tempo di ricerca.

FCFS: Arriva una richiesta di lettura di un blocco sul cilindro 11. Mentre è in corso la ricerca del cilindro 11, arrivano nuove richieste dei cilindri 1, 36, 16, 34, 9 e 12, in quest'ordine.



SSF (Shortest Seek First): Arriva una richiesta di lettura sul blocco del cilindro 11.

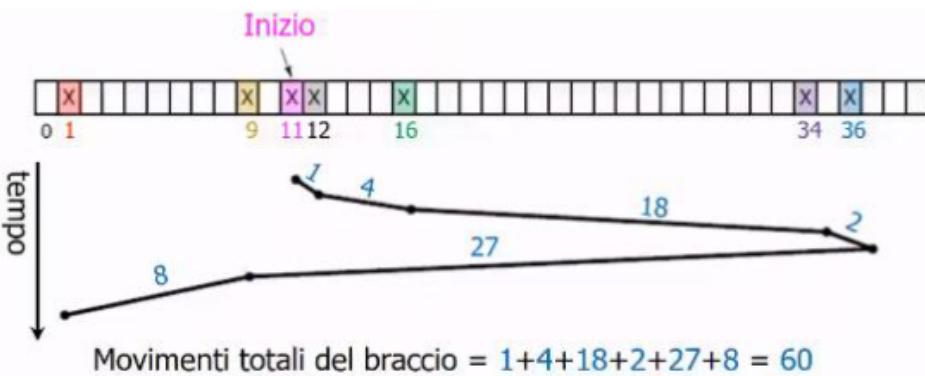
Arrivano poi le richieste: 1, 36, 16, 34, 9 e 12. L'algoritmo riordina le richieste al fine di minimizzare il tempo di ricerca.



SSF ha un problema simili a quello degli ascensori degli edifici alti: i cilindri (piani) centrali sono serviti meglio (perché hanno maggiore priorità) rispetto a quelli posti alle due estremità.

L'algoritmo dell'ascensore: utilizza un bit che rappresenta la direzione attuale dell'ascensore: UP o DOWN. Quando ha completato una richiesta, il driver del disco controlla il bit di direzione: UP se esiste una richiesta pendente verso l'alto il movimento del braccio è in alto altrimenti il bit di direzione è invertito o DOWN se esiste una richiesta pendente verso il basso il movimento del braccio è in basso altrimenti il bit di direzione è invertito.

La posizione iniziale è sempre il cilindro 11 poi arrivano le altre richieste.



In generale è peggio del SSF. Indipendentemente dalle richieste, nell'algoritmo dell'ascensore il limite superiore di movimenti è pari a due volte il numero di cilindri.

## Gestione degli errori

I produttori di dischi si spingono sempre ai limiti della tecnologia, aumentando le densità lineari dei bit.

I difetti di fabbricazione producono settori difettosi, cioè settori che non rileggono correttamente il valore che vi è appena stato scritto. Se il difetto è di pochi bit, è possibile correggerlo ogni volta grazie all'ECC. Altrimenti non può essere mascherato.

Nel caso di blocchi danneggiati possono essere adottati due approcci: trattarli a livello di controller oppure a livello di sistema operativo.

## Memoria stabile

I dischi molte volte commettono degli errori: un settore buono può deteriorare e divenire difettoso. Interi dischi possono smettere di funzionare inaspettatamente.

I RAID proteggono i dati contro queste difettosità, ma non proteggono errori di scrittura che alterano i dati originali senza sostituirli dai nuovi.

L'ideale sarebbe un disco che lavorasse tutto il tempo senza errori. Sfortunatamente, è irrealizzabile.

Quello che si può avere è un sistema stabile che quando gli arriva un comando di scrittura, ha due possibilità: scrivere i dati correttamente o non fare nulla. L'obiettivo è la coerenza del disco a ogni costo.

La memoria stabile utilizza una coppia di dischi identici con blocchi corrispondenti che lavorano insieme a formare un blocco esente da errori. Per ottenere questo risultato sono definite le tre operazioni:

1. Scritture stabili: viene scritto sul disco 1, e successivamente riletto per verificare che sia stato scritto correttamente. Se ciò non accade, le operazioni di scrittura e rilettura sono eseguite n volte finché non funzionano. Al termine di n tentativi si utilizza un blocco di riserva. Una volta scritto il disco 1, si ripetono le stesse operazioni sul 2 finché anch'esso avrà il medesimo contenuto.
2. Letture stabili: viene letto il blocco dal disco 1. Se questo ha un ECC errato, la lettura si ritenta di nuovo, per n volte. Se risulta ancora un ECC errato, viene letto il disco 2 (è altamente improbabile che siano entrambi guasti).
3. Crash recovery (ripristino da un crash): Dopo un crash, un programma di ripristino fa la scansione di entrambi i dischi per fare il confronto dei blocchi corrispondenti. Se uno di loro presenta un errore di ECC, il blocco difettoso è sovrascritto con quello dell'altro disco. Se sono entrambi validi ma con diverso contenuto, il blocco sul disco 1 sovrascrive il blocco del disco 2.

## **5.e Clock**

I clock (chiamati anche timer) sono fondamentali per il funzionamento di un sistema multiprogrammato per molte ragioni.

Segnano il tempo e permettono il controllo del tempo di esecuzione reale dei processi evitando così il monopolio della CPU.

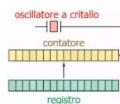
Il software del clock può presentarsi sotto forma di driver di dispositivo, sebbene il clock non sia né un dispositivo a blocchi né un dispositivo a caratteri.

### **Hardware del clock**

Nei computer sono utilizzati comunemente due tipi di clock, entrambi abbastanza diversi dagli orologi a cui siamo abituati.

I clock più semplici bastava collegarli alla tensione di alimentazione (110/220 volt) per avere un interrupt ogni ciclo di tensione (50/60 Hz). Oggi non esistono più.

L'altro tipo di clock è composto da tre componenti: un oscillatore a cristalli, un contatore (decrementato ad ogni impulso) e un registro (utilizzato per caricare il contatore).



Quando un pezzo di cristallo di quarzo è sottoposto ad una tensione, per la sua proprietà piezo-elettrica, genera un segnale periodico (centinaia di MHz a seconda del cristallo prescelto).

Il segnale base può essere moltiplicato per un numero intero per arrivare a frequenze dell'ordine dei GHz. Questi circuiti sono presenti in ogni computer.

Il segnale è inviato ad un contatore che decrementa il suo valore fino a zero generando l'interrupt e per poi ricominciare.

I clock programmabili hanno due modalità di funzionamento:

- One-shot: quando il clock parte, copia il valore del registro nel contatore e quindi decrementa il contatore ad ogni impulso del cristallo. Quando il contatore arriva a zero, provoca un interrupt e si arresta finché non viene esplicitamente riavviato dal software.
- Onda quadra: dopo aver raggiunto lo zero e causato l'interruzione, il registro è copiato nel contatore e il processo si ripete all'infinito. Gli interrupt periodici sono detti clock ticks.

Il vantaggio dei clock programmabili è che la frequenza di interrupt può essere controllata dal software.

## Software del clock

L'hardware clock genera interrupt a intervalli ben precisi. Tutto il che coinvolge il tempo deve essere fatto dal software: il driver del clock. Tra i compiti più comuni in vari SO:

1. Mantenere l'ora del giorno;
2. Evitare che i processi siano eseguiti più tempo di quanto consentito;
3. Contabilizzare l'uso della CPU;
4. Gestire la system call alarm() invocata dai processi utente;
5. Fornire il timer watchdog per le componenti del SO che ne fanno richiesta;
6. Eseguire il profiling, il monitoraggio ed elaborazioni statistiche.

## Soft timer

Quasi tutti i computer hanno un secondo orologio programmabile che può essere impostato per causare le interruzioni a qualsiasi frequenza.

I timer soft evitano interruzioni: è il kernel che, quando è in esecuzione per qualche ragione, prima di rientrare nella modalità utente controlla che non sia scaduto un timer soft.

I timer soft sono controllati con la stessa velocità con cui il kernel entra in azione per svolgere altri compiti, quindi quando:

- viene attivata una system call;
- accade una page miss della TLB;
- accade un page fault;
- accade un interrupt di I/O;
- la CPU diventa inattiva (idle).