Corso Luigi Einaudi, 55/B - Torino

Appunti universitari
Tesi di laurea
Cartoleria e cancelleria
Stampa file e fotocopie
Print on demand
Rilegature

NUMERO: 2372A ANNO: 2018

A P P U N T I

STUDENTE: Chiapello Nicolò

MATERIA: Programmazione di Sistema - (Teoria) - Prof. Cabodi

Il presente lavoro nasce dall'impegno dell'autore ed è distribuito in accordo con il Centro Appunti.
Tutti i diritti sono riservati. È vietata qualsiasi riproduzione, copia totale o parziale, dei contenuti inseriti nel presente volume, ivi inclusa la memorizzazione, rielaborazione, diffusione o distribuzione dei contenuti stessi mediante qualunque supporto magnetico o cartaceo, piattaforma tecnologica o rete telematica, senza previa autorizzazione scritta dell'autore.

ATTENZIONE: QUESTI APPUNTI SONO FATTI DA STUDENTI E NON SONO STATI VISIONATI DAL DOCENTE.
IL NOME DEL PROFESSORE, SERVE SOLO PER IDENTIFICARE IL CORSO.

PROGRAMMAZIONE DI SISTEMA

TEORIA

Politecnico di Torino

Laurea Magistrale in Ingegneria Informatica

Chiapello Nicolò

Contenuto:

Informazioni sul corso

Nome:	Programmazione di Sistema
Settore scientifico-disciplinare:	ING-INF/05
Tipo di attività formativa:	caratterizzante
Crediti:	10 CFU
Docente:	Gianpiero Cabodi
Livello accademico:	laurea magistrale in ingegneria informatica
Periodo:	primo anno, secondo semestre
Anno accademico:	2017 / 2018
Autore:	Chiapello Nicolò
Versione:	1.0.1

Informazioni su quest'opera

Questo lavoro vuole essere la trasposizione della spiegazione fornita in aula, da parte del docente. Esso è frutto di una stesura effettuata direttamente in aula e di successivi eventuali aggiustamenti (specialmente durante il periodo di preparazione all'esame). Non si tratta quindi di sbobinature di lezioni, ma di appunti *real-time*, con tutte le imprecisioni che questo può comportare.
Quest'opera non è stata controllata in alcun modo dai professori e quindi potrebbe contenere degli errori. Se ne trovi uno sei invitato a contattare l'autore, affinché esso venga corretto (nicochina.notes@gmail.com).
Questi appunti sono quindi forniti *as is* : costituiscono uno strumento ottimale per l'accompagnamento allo studio, ma non costituiscono una fonte accademica di nozioni.

Legenda

In questa opera è stata adottata la seguente convenzione:

- *corsivo*: concetto/entità precedentemente già definita
- **grassetto**: concetto di rilievo, in evidenza
- sottolineato: mini-titolo a cui si riferisce il testo successivo
- **giallo grassetto**: definizione di un concetto (riportato nell'indice analitico finale)
- azzurro: argomento di cui si occupa un paragrafo

2. Hashed page table 32
3. Inverted page table 32
Effective address space dimension 33
Example: the Intel 32 and 64-bit architectures 33
1. IA-32: 34
2. Intel x86-64: 36
Example: ARM architecture 36

Virtual memory 37
Background 37
Virtual address space 37
Shared library 39
Demand paging 39
Validity bit 39
Page Fault 40
Aspects of Demand Paging 41
Performance of Demand Paging 42
Demand Paging optimization 42
Copy-on-Write 43
Page replacement 43
Basic Page Replacement 44
Page replacement algorithms 45
1. FIFO algorithm 46
2. Optimal algorithm 47
3. LRU algorithm 48
a. Reference bit 50
b. Second-chance algorithm 50
c. Enhanced second-chance algorithm 51
d. Counting algorithms 51
4. Page-buffering algorithms 51
5. Applications and Page Replacement 52
Allocation of frames 52
1. Fixed allocation 53
2. Priority allocation 53
Global vs. local allocation 53
NUMA 54
Thrashing 54
Località 55
Working-set model 55
Keeping track of the working set 56
Page-Fault frequency 56
Memory-mapped files 57
Memory-mapped file technique for all I/O 58
Memory mapping and copy on write 58
Inter process communication 58
Shared memory 59

2. SSTF 86
3. SCAN 86
a. C-SCAN 87
b. C-LOOK 87
Comparison: selecting a disk-scheduling algorithm 88
Disk Management 88
Swap-Space Management 88
RAID Structure 89

I/O systems 90
Overview 90
I/O Hardware 90
1. Polling 91
2. Interrupt 91
3. DMA 92
Application I/O Interface 93
Kernel I/O structure 93
Characteristics of I/O devices 94
Vectored I/O 95
Kernel I/O Subsystem 95
Error handling 96
I/O protection 96
System Call for I/O 96
Kernel data structure 97
Transforming I/O Requests to Hardware Operations 98
Life cycle of an I/O request 98
STREAMS 98
Performance 99

OS161 - INTRODUCTION 100

Introduction 100
Contenuto di OS161 101

Thread 101
Thread context 102
Structure thread 103
Kernel thread 104
Thread interface 105
MIPS registers 108
System Calls, Exceptions and Interrupts 110
1. System Call 111
2. Exception 111
Two processes in OS161 111
System calls for Process Management 112

Memory Management 112

Background 144
Critical section 145
1. Peterson's algorithm 145
2. Lock 146
Lock implementation 147
3. Spinlock 147

OS161 synchronization 149
1. Disabling interrupts 149
2. Semafori (interrupt based) 149
3. Spinlocks 151
4. Thread blocking 151
5. Locks 152
6. Condition variables 152
Semaphore limitation 152
Condition variables 155
7. Wait channels 157
8. Semafori (multicore) 158

UNIX FILE MANAGEMENT 160

Virtual File System 160
Supporting multiple File System 161
VFS interface 162
VFS di OS161 163
Strutture dati principali 163
Operazioni sui vnode 164

File Descriptor and Open File tables 166
Interfaces 167
File descriptor 167
File pointer 168

Buffer cache 169

modulare, ma più efficiente del passaggio parametri tramite *stack*), OS/161 è scritto con lo scopo principale di essere comprensibile senza alcuna ambizione di essere fluido. Alcune funzioni non sono state implementate, lasciate appositamente allo studente.
È fortemente utilizzato il linguaggio C, ma con alcune *features* non pienamente esplorate nei corsi introduttivi della triennale (APA), come gli *operatori logici bit-wise*, *puntatori void* (`void*`), ecc…

Testi di riferimento

Il corso di Cabodi fa riferimento ai seguenti testi:

- Silbershatz A., Galvin. P. "**Sistemi Operativi**", Addison-Wesley Publishing Company, Nona Edizione

Esame

L'esame di ciascuna parte del corso può essere data separatamente, anche in anni accademici distinti.
Ad ogni appello vengono proposte entrambi le parti, spesso prima Cabodi e poi Malnati. È possibile sostenerne anche una sola alla volta. Ripresentarsi ad un appello significa sostituire automaticamente il voto precedente acquisito in quella parte di esame (anche se ci si ritira).
L'esame è composto da 3 parti aventi il seguente peso:

- teoria (Cabodi): 14 punti
- programmazione (Malnati): 8 punti
- progetto (Malnati): 9 punti

1. Teoria (Cabodi):

Consiste in uno scritto con risposte aperte, problemi a cui si richiede una soluzione o la scrittura di codice. Sono importantissimi i laboratori, dei quali NON è richiesta la consegna settimanale.
Il punteggio ha soglia minima di 8 punti. Durata 70min, 4 domande (2 domande da 3pti e 2 domande da 4pti) di cui solitamente una su OS161 (codice), una di teoria e due esercizi.

2. Programmazione (Malnati):

Scritto di 50min composto da 2 domande di C++ (1 punto ciascuna) e 1 solitamente di C# (2 punti) più la stesura di un programma C++ concorrente (4 punti). Il voto finale ha soglia minima di 4 punti.

3. Progetto (Malnati):

Progetto di gruppo, completo e complesso; l'argomento varia di anno in anno e ha lo scopo di mostrare le difficoltà del processo di produzione di un software e di *team working*. Il progetto è opzionale, ma solo se la somma delle altre due parti supera il 18.
I voti raggiungono spesso 8 o 9 punti, raramente vengono assegnati solo 6 punti.

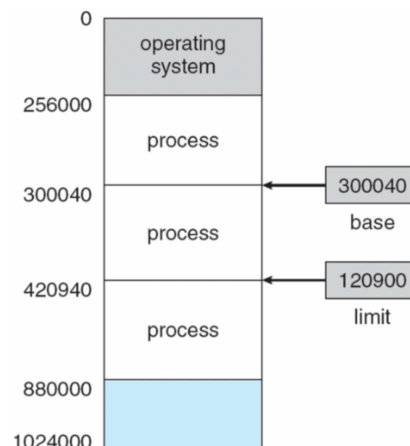
Le **cache** sono un sistema a più livelli di memoria che si frappongono nel mezzo della differenza prestazionale tra disco e CPU. Ciascuna replica il contenuto del disco per "avvicinarlo" alla CPU con componentistica hardware più piccola e più veloce.
Algoritmi di caching devono prevedere i dati più utilizzati e caricarli in cache.

Base and Limit registers

Un altro aspetto critico, specialmente nell'ultimo periodo, è la protezione della memoria a seconda dei permessi accordati a ciascuna tipologia di utente.
Questa protezione permette di proteggere sia da errori casuali nella programmazione (puntatore C alla cella errata) oppure per scenari di *cyber security* (protezione di dati sensibili).

Nei sistemi tradizionali (computer) si cerca di migliorare gli aspetti della gestione di memoria, mentre nei sistemi *mobile* si evitano particolari migliorie e si ha una spinta commerciale a comprare semplicemente una memoria di capacità maggiore.

La rappresentazione grafica della RAM è analoga a quella di un vettore: un vettore indicizzato di celle contigue.



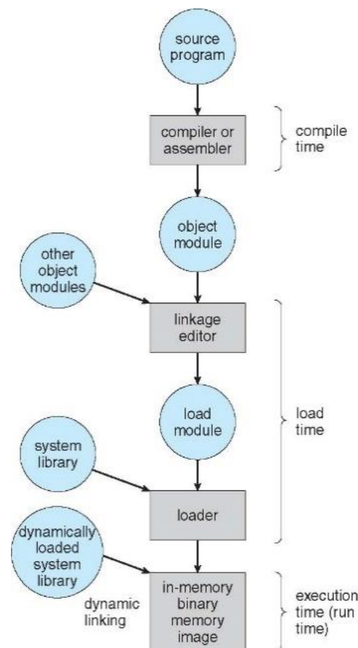
L'aspetto fondamentale è la capacità di mantenere in RAM più programmi contemporaneamente. Questo permette il *multitasking*: si partiziona la RAM in intervalli contigui e ciascuna porzione è assegnata a un processo che è virtualmente in concorrenza (la CPU gestisce l'istruzione di un solo processo alla volta). Una porzione di memoria centrale è delegata anche al sistema operativo e questa non sarà mai *swapped* finché il sistema è attivo.
Ciascuna porzione memoria centrale può essere destinata a:

- sistema operativo
- processo utente

Ciascuna partizione di memoria è dotata di due registri specifici che permettono di gestire il suo inizio e la sua fine:

- **base register**: offset di inizio della partizione
- **limit register**: dimensione della partizione

La **toolchain** è la successione di strumenti che permettono di passare dal codice sorgente all'eseguibile. Essa prevede la compilazione, il linking, il caricamento in memoria e la successiva esecuzione di un programma:



Logical and Physical Address Spaces

Esistono due tipi di indirizzi:

- **logical address (virtual address)**: generato dalla CPU
- **physical address**: generato dalla *Memory Management Unit* e utilizzato per accedere effettivamente alla RAM

Esiste una leggera differenza tra *indirizzi logici* e *indirizzi virtuali*, ma sono minimi e specificati successivamente nel capitolo relativo alla virtualizzazione della memoria. Possiamo dire che il capitolo "Main Memory" di occupa di indirizzi logici, mentre il capitolo "Virtual Memory" è legato agli indirizzi virtuali.

Quando il programma non può essere spostato in RAM (*bindig* effettuato a *compile time* o *load time*), allora indirizzi logici e fisici possono coincidere. Quando invece il programma può essere spostato (*bindig* effettuato a *execution time*), allora i due indirizzi non coincidono più.

Esistono due **spazi di indirizzamento**:

- *logical address space*: *range* degli indirizzi prodotti dalla CPU
- *physical address space*: *range* degli indirizzi realmente assegnati ad un processo, nella RAM

MMU

La **Memory Management Unit (MMU)** è un modulo hardware che mappa indirizzi logici in indirizzi fisici. Effettua anche la paginazione (per ora ignorata).
Il **relocation register** è nome che assume il *base register* in questo contesto. È sommato all'indirizzo logico per ottenere l'indirizzo fisico.

Viene spesso eliminato il programma più vecchio (da più tempo presente in RAM), ma non si vuole perderne le informazioni, quindi lo si memorizza temporaneamente nel disco.

Tale processo permette di avere virtualmente programmi in esecuzione di dimensione maggiore rispetto a quella fisicamente disponibile in RAM.

Il **backing store** è la porzione di spazio nel disco, delegato a contenere i programmi su cui è stato effettuato lo *swap out*.

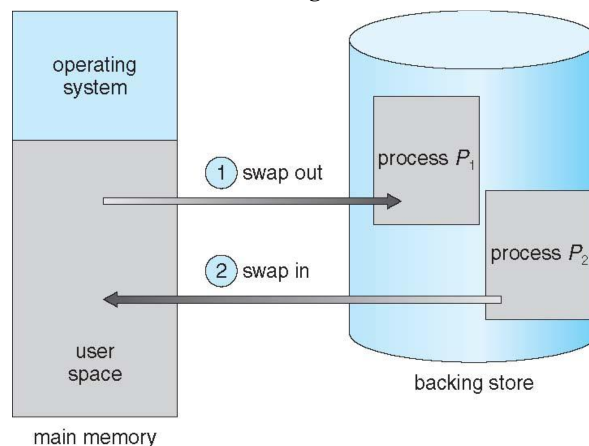
Le azioni compiute per portare sul disco/RAM sono chiamate

- **swap out (roll out)**: spostare frammenti dalla RAM al disco
- **swap in (roll in)**: spostare frammenti dal disco alla RAM

Il sistema mantiene una **ready queue** di tutti i processi che sono stati *swapped out* sul disco.

Per effettuare lo *swapping* è necessario il tempo di accesso a disco, quindi le prestazioni sono povere. È una gestione pensata per quelle occasioni in cui il sistema è in situazioni critiche, quindi le prestazioni sono già povere di suo.
In sistemi *mobile*, piuttosto che lo *swapping*, è preferita la *kill* di processi vecchi. Non si utilizza quindi la memoria, in favore di una politica più aggressiva.

Il processo di *swapping* è schematizzabile come segue:



Context Switch

Il **context switch** è il cambiamento dell'esecuzione in CPU da un programma ad un altro.
In condizioni normali è sufficiente cambiare i registri, ma se è stata effettuata una *swap*, il processo di *context switching* diventa particolarmente lungo. Richiede sicuramente uno *swap in* (programa richiesto) ed eventualmente anche lo *swap out* di un altro programma (per liberare spazio richiesto). Il tempo richiesto diventa quindi considerevole.

Quando il processo 8 termina, esso libera dimensione in memoria. Sono soddisfabili quindi processo che richiedono dimensione minore dello spazio disponibile. In caso contrario è necessario effettuare una *swap out*.
La multiprogrammazione dipende dal numero di partizioni che è possibile far stare contemporaneamente in RAM. Lo spazio però è differente per ogni processo.

Si apre il problema della **frammentazione** della memoria. Si rischia di giungere alla situazione per cui si ha lo spazio disponibile per un nuovo processo, ma suddiviso in tanti piccole porzioni sollo spazio totale degli indirizzi.

In generale è necessaria una struttura dati che contenga le informazioni riguardo alle partizioni libere in memoria. Potrebbe essere una **Free List**, oppure una **Bit Map**.

Allocation algoritm

La scelta di quale partizione utilizzare per inserire un nuovo processo può seguire diverse filosofie:

- **first-fit**: allocare nella prima partizione libera, compatibile con le dimensioni richieste
- **best-fit**: allocare nella più piccola partizione libera, compatibile con le dimensioni richieste
- **worst-fit**: allocare nella più grande dimensione libera, compatibile con le dimensioni richieste

La first-fit ottimizza il tempo necessario per compiere la scelta, di complessità ragionevole se la frammentazione è equilibrata.
La best-fit ottimizza lo spazio utilizzato, sperando di trovare un buco di dimensione esattamente pari alla dimensione del processo. Si avrà facilmente uno scarto, ma questo scarto è il minimo possibile. Tale politica richiede però la scansione totale della memoria per trovare il sito idoneo.
La worst-fit ottimizza la possibilità di riutilizzare il buco lasciato dopo l'allocazione precedente. Tende a privilegiare l'allocazione su porzioni grandi al fine di riempirlo completamente.
Tale politica richiede però la scansione completa della memoria.

L'efficacia di queste politiche dipende dalla varianza delle dimensioni.
Per dimensioni omogenee, è meglio *first-fit* (che è molto simile alla *best-fit*). Nella realtà delle cose, le scelta sono scelte in base a delle euristiche di utilizzo.
In ogni caso non è possibile evitare la frammentazione.

Fragmentation

La frammentazione può essere di più tipi:

- **frammentazione esterna**: deriva dallo spazio lasciato tra diverse allocazioni
- **frammentazione interna**: deriva dalla sovra-allocazione delle partizioni, spesso dovuta all'utilizzo di multipli di una dimensione base (pagina)

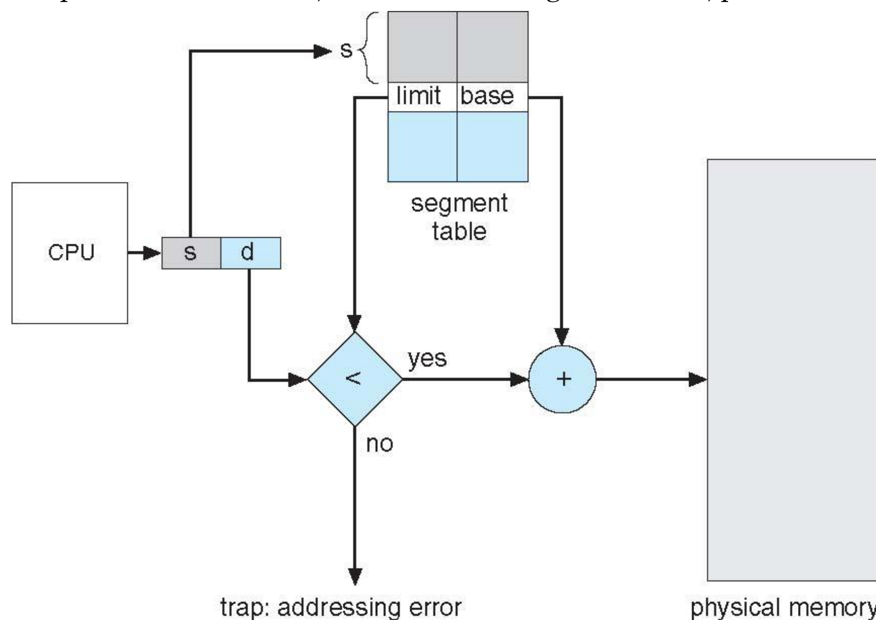
In questo contesto la situazione più critica è la frammentazione esterna. Quella interna non è eccessivamente influente.
La frammentazione è critica perché si può arrivare a sprecare fino a circa la metà dello spazio disponibile.

Il registro **STBR (Segment-Table Base Register)** punta alla locazione della *Segment Table* in memoria centrale, mentre il registro **STLR (Segmant-Table Length Register)** indica il numero di segmenti usati da uno specifico programma.
La protezione dei segmenti è effettuata associando ad ogni entry della *Segment Table*:

- *validity bit*
- *read/write/execute privileges*

Segmentation hardware

Lo schema della *Memory Management Unit*, con la segmentazione è il seguente.
L'indirizzo logico, emesso dalla CPU, è suddiviso in due parti. La parte alta *s* è il **selector** che viene sommata al *STBR* ed è usata per accedere alla *Segment Table*, mentre la low part *d* è il **displacement** (offset) all'interno del segmento.
Il *displacemnt* sarà poi sommato al *base*, ottenuto dalla *Segment Table*, per ottenere l'indirizzo fisico.



3. Paging

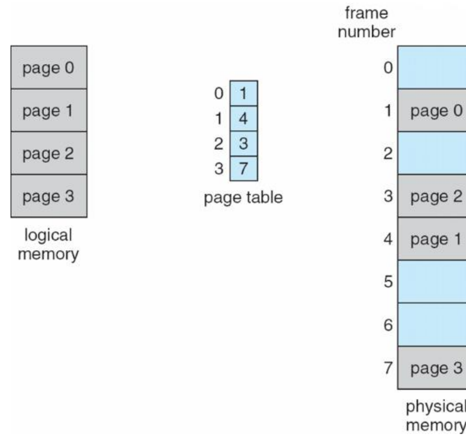
La paginazione è la reale soluzione della frammentazione esterna, ma ha anche altri benefici.
La **paginazione** prevede che ogni processo abbia uno spazio di indirizzamento contiguo a livello di pagine, ma non obbligatoriamente da uno spazio di indirizzamento fisico contiguo.
La RAM è quindi utilizzata a livello di pagine. Non si ha più frammentazione esterna. Allocando solo a multipli di pagina, si ha però un problema di frammentazione interna.
Non si ha più dimensione variabile del blocco di allocazione, dove l'unità fondamentale di allocazione è la pagina.

A rigore si ha la differenza tra i seguenti termini:

- **pagina**: blocco di indirizzamento logico
- **frame**: blocco di indirizzamento fisico

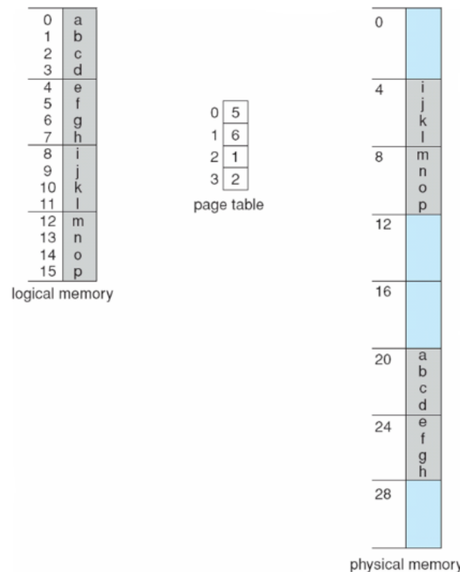
Esempio:

Dato un processo con 4 pagine in memoria logica. Ad ognuna di essa corrisponde una riga nella tabella delle pagine (*Page Table*) e nella RAM (*physical memory*) sono allocati 4 frame corrispondenti. I *frame*, all'interno della memoria centrale, NON sono né ordinati, né contigui.
Il modello di paginazione di memoria logica e fisica è il seguente:



Esempio:

Rispetto all'esempio precedente, si tenta anche di visualizzare il contenuto delle pagine (rappresentato mediante generici simboli alfabetici). Si dimensiona la pagina a $4 byte$ e si accosta la visione logica con la visione fisica della memoria. Il *mapping* tra queste rappresentazioni è effettuato dalla *Page Table* che associa ad ogni numero di pagina un numero (progressivo) di frame.



Esempio:

Calcolare la frammentazione interna, date le seguenti informazioni:

$page_size = 2048 \; byte$

$process_size = 72766 \; byte$

È necessario allocare 36 pagine per coprire anche il resto della seguente frazione:

$$\frac{process_size}{page_size} = \frac{72766 \; byte}{2048 \; byte} = 35 \; pages + 1086 \; byte$$

Quando dev'essere istanziato un nuovo processo, viene consultata la **free frame list**: una struttura dati che contiene i riferimenti (puntatori o indici) a tutti i frame liberi in memoria.
Tale struttura dati viene spesso implementata come:

- lista linkata
- lista realizzata con un vettore di indici

All'interno della lista non si ha un ordine della disposizione dei frame, ma l'ordine dipende dal processo di creazione della lista. Per selezionare le pagine da occupare, viene prelevata l'informazione in testa e usata per creare la *page table*.

$$(free\ frame\ list) \xRightarrow{allocazione} (page\ table)$$

Page Table implementation

La *Page Table* è mantenuta in memoria centrale, è impossibile portarla in CPU a causa delle sue dimensioni.
Per ora assumiamo che ci sia una *Page Table* per ogni processo (è anche possibile creare *Page Table* comuni a più processi).
La *Page Table*, in quanto vettore, dev'essere individuata in memoria mediante due informazioni:

- punto di inizio
- dimensione

In CPU ci sono due registri che contengono queste informazioni riferite alla Page Table del processo attualmente in esecuzione:

- **PTBR (Page-Table Base Register)**: puntatore alla *Page Table*
- **PRLR (Page-Table Length Register)**: dimensione della *Page Table*

Durante un *Context Switch* verranno modificate le informazioni all'interno di questi due registri.

Analogamente alla segmentazione, anche la paginazione deve pagare il costo di un duplice accesso in memoria per leggere/scrivere un dato. Questo perché la *Page Table*, necessaria per accedere in memoria, è anch'essa in memoria.

$$indirizzo\ logico \xRightarrow{accesso\ memoria} indirizzo\ fisico \xRightarrow{accesso\ memoria} lettura/scrittura\ dato$$

Nei processori moderni tale problema del duplice accesso è stato gestito inserendo una sorta di cache, per la precisione una *associative memory*, chiamata **Translation Look-aside Buffer (TLB)**. Si tratta di registri (hardware) contenuti in CPU che cercano di tradurre indirizzo logico in indirizzo fisico cercando di evitare l'accesso alla *Page Table*.
Il principio base è molto simile a quello delle *cache*: si tenta di "avvicinare" le informazioni utili e utilizzate più spesso. Se la *TLB* non contiene l'informazione desiderata, è comunque necessario accedere alla *Page Table* in memoria.

La *TLB* è una tabella nella quale ogni TLB entry contiene diverse informazioni:

- TLB per un solo processo: $entry_{TLB} = (IDpage; IDframe)$
- TLB condivisa tra più processi: $entry_{TLB} = (IDprocess;\ IDpage; IDframe)$

Effective Access Time

Posto ε il **tempo di *associative lookup*** (tempo necessario alla *TLB* per rispondere), esso è molto piccolo. Essendo inferiore ad un colpo di clock, è possibile assimilarlo a zero, cioè considerarlo "gratuito".
Il tempo ε può essere molto inferiore del tempo di accesso in memoria, non dev'essere quindi il tempo dominante all'interno del processo.
La **hit ratio** α è la percentuale di volte in cui il numero di pagina è trovato all'interno della *TLB*. Si tratta di percentuali estremamente elevate, spesso superiori al 99%.
Considerando come unità di misura temporale (*time units*) il tempo di un accesso in memoria RAM, il **Effective Access Time (EAT)** è il tempo medio pesato necessario ad accedere ad una cella in memoria. Sono possibili 1 accesso (il α delle volte) oppure $(n+1)$ accessi (il $(1-\alpha)$ delle volte) , con n il numero di livelli della *page table*. Il valore quantitativo di EAT risulta:

$$\begin{aligned} EAT &= (T_{acc} + \varepsilon) \cdot \alpha + [(n+1) \cdot T_{acc} + \varepsilon] \cdot (1-\alpha) \\ &= (n+1) \cdot T_{acc} + \varepsilon - \alpha \cdot T_{acc} \end{aligned}$$

Nei casi estremi, α assume valore 1 oppure 0:

$$EAT = \begin{cases} T_{acc} + \varepsilon & se\ \alpha = 1 \Longrightarrow TLB\ hit \\ (n+1) \cdot T_{acc} + \varepsilon & se\ \alpha = 0 \Longrightarrow TLB\ miss \end{cases}$$

Esempio:

> Considerando un sistema con una pessima *hit ratio*:
>
> $\alpha = 80\%$
>
> $\varepsilon = 20ns$ per ricerca TLB
>
> $T_{acc} = 100ns$
>
> il Effective Access Time risulta (con $\varepsilon \cong 0$):
>
> $EAT = 0.8 \cdot 100ns + 0.2 \cdot 200ns = 120ns$
>
> che è superiore al tempo di accesso in RAM.

Esempio:

> Considerando invece un sistema più realistico:
>
> $\alpha = 99\%$
>
> $\varepsilon = 20ns$ per ricerca TLB
>
> $T_{acc} = 100ns$
>
> il Effective Access Time risulta (con $\varepsilon \cong 0$):
>
> $EAT = 0.99 \cdot 100ns + 0.01 \cdot 200ns = 101ns$
>
> che è appena maggiore del tempo di accesso in RAM.

Memory protection

La **protezione della memoria** è il processo che garantisce che un processo non possa convertire in indirizzo fisico un indirizzo logico che ricade al di fuori degli intervalli previsti.

Nel caso di allocazione contigua l'insieme di indirizzi ammessi è un range continuo, la protezione quindi è fornita per due sole situazioni critiche:

- sotto al limite fisico inferiore: semplice somma dell'indirizzo con il *base register*
- sopra al limite fisico superiore: utilizzo di un comparatore

Il controllo del *validity bit* è delegato all'hardware. Non è quindi necessario alcun intervento software.

Shared pages

È possibile definire delle **pagine condivise** tra più processi. Il codice, spesso librerie, potrebbe essere condiviso (es. funzioni di libreria di standard I/O). In generale dovrebbe trattarsi di materiale in sola lettura.

È possibile condividere del codice solo se questo è *reentrant*. Un codice è definito **reentrant code** se possiede solo delle istruzioni in sola lettura e nessuna variabile globale. Le funzioni NON *rientranti* sono dette "con memoria" perché memorizzano lo stato precedente e questo ne condiziona l'esecuzione alla loro chiamata.
Il codice può essere condiviso solo se *rientrante*, altrimenti i due processi avrebbero una variabile in comune non protetta (non gestita in mutua esclusione) e ciascuno potrebbe modificare lo stato settato dall'altro.
Le variabili locali sono invece memorizzate nello *stack* e ciascun processo possiede il proprio. Esse non possono quindi essere condivise.

Del codice è definito *rientrante* se soddisfa entrambe le condizioni:

- presenta solo istruzioni e nessuna variabile globale
- istruzioni sono tutte in sola lettura

Le funzioni di libreria di tipo rientrante possono essere condivise:

$$\left.\begin{array}{r}\textit{nessuna variabile globale}\\ \textit{tutte istruzioni in sola lettura}\end{array}\right\} \Longrightarrow \textit{codice rientrante} \Longrightarrow \textit{condivisibile}$$

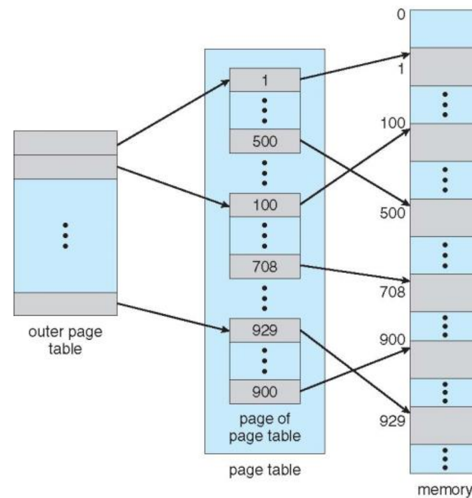
La condivisione permette di risparmiare memoria e viene implementata facendo puntare gli indirizzi (per un stessa funzione) di entrambi i processi, ad una stessa locazione di memoria.

È possibile condividere dei dati per effettuare una programmazione concorrente nella quale processi diversi condividono consapevolmente delle variabili globali. Esse sono accessibili da entrambi e devono essere gestite adeguatamente (mediante meccanismi di sincronizzazione).

Le **pagine private** sono quelle pagine specificatamente assegnate ad un processo. Si tratta di dati eventualmente condivisibili, ma per i quali ogni processo decise di creare una copia privata e accessibile solo a lui.

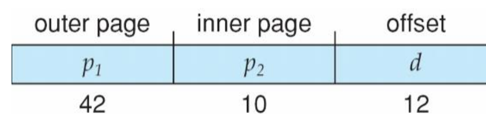
Esempio:

> Consideriamo che il codice condiviso sia di un programma di editor (elaborazione testuale). I tre processi (P_1, P_2, P_3) condividono il codice delle pagine $(ed1, ed2, ed3)$ e ciascuno possiede un'ulteriore pagina di dati personali.

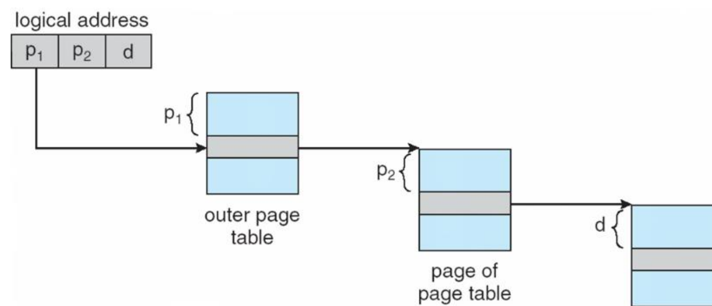


L’indirizzo dovrà quindi essere partizionato in più parti: non solo *p* e *d*, ma il *page number* dovrà essere suddiviso in due sottoparti:

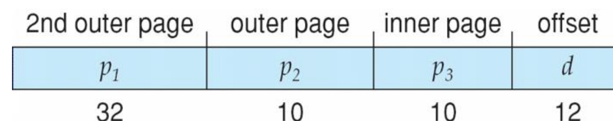
- <u>page number di primo livello (p_1)</u>: utilizzata per accedere alla *outer page table*
- <u>page number di secondo livello (p_2)</u>: utilizzata per accedere alla *page of page table*



Il meccanismo di traduzione avviene quindi in più step successivi: p_1 è sommato all’indirizzo iniziale della outer *page table* per recuperare l’indirizzo iniziale della page of *page table*; esso è sommato a p_2 per ottenere l’indirizzo iniziale della pagina in memoria; e ancora questo è sommato a *d* per ottenere il dato puntato dall’indirizzo originale.

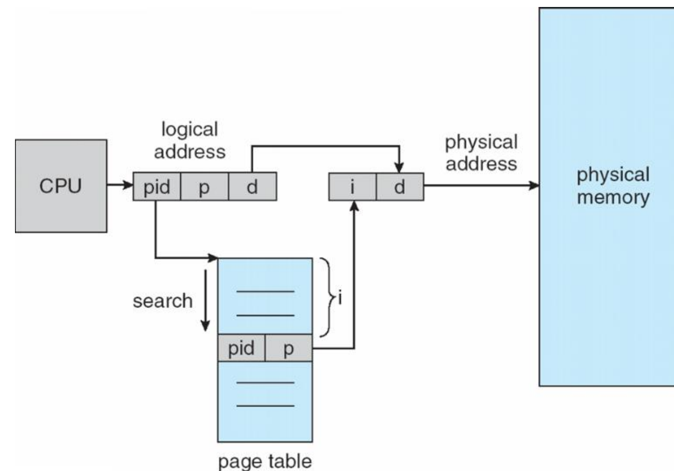


Per sistemi a 64-bit, due soli livelli di tabelle non sono sufficienti, perché si avrebbero pagine di dimensioni eccessive. Si è scelto quindi di adottare uno schema di paginazione a 3 livelli aggiungendo una ulteriore *outer page table*.



Aggiungere un livello non è gratis. Per ogni livello è necessario un ulteriore accesso in memoria per recuperare le informazioni necessarie per accedere alla memoria desiderata.

Per tradurre logico→fisico è necessario fare una scansione della *Page Table* cercando p e, una volta trovato, ritornare l'indice i della riga che lo contiene.



Invece di avere una *Page Table* per ogni processo, in questa tecnica si implementa una solo *inverted Page Table* per tutti i processi.
Inoltre la sua scansione, per trovare p, sarebbe lineare (molto costosa). Si preferisce quindi frapporre una tabella di hash per ottenere un accesso quasi diretto.
Utilizzando la *inverted Page Table* bisogna anche gestire le problematiche legate alla memoria condivisa: è necessaria una lista per memorizzare tutti gli indirizzi virtuali che sono mappati sullo stesso frame.

Effective address space dimension

La dimensione dello spazio di indirizzamento mostrato in una *Page Table* è minore della somma delle pagine allocabili in una memoria. Questo perché qualunque *Page Table* è memorizzata in RAM, quindi le pagine da lei occupate (poche rispetto allo spazio totale) non sono allocabili ai processi. Una *Page Table* deve quindi mappare uno spazio di indirizzamento di dimensione leggermente minore al totale della RAM:

$$\dim(AddressSpace) = \dim(RAM) - \dim(PageTable)$$

Example: the Intel 32 and 64-bit architectures

Le principali famiglie di processori attualmente sul mercato sono le seguenti:

- Intel: applicazioni general purpose
- ARM: applicazioni embedded

La famiglia Intel implementa due tipologie di parallelismo:

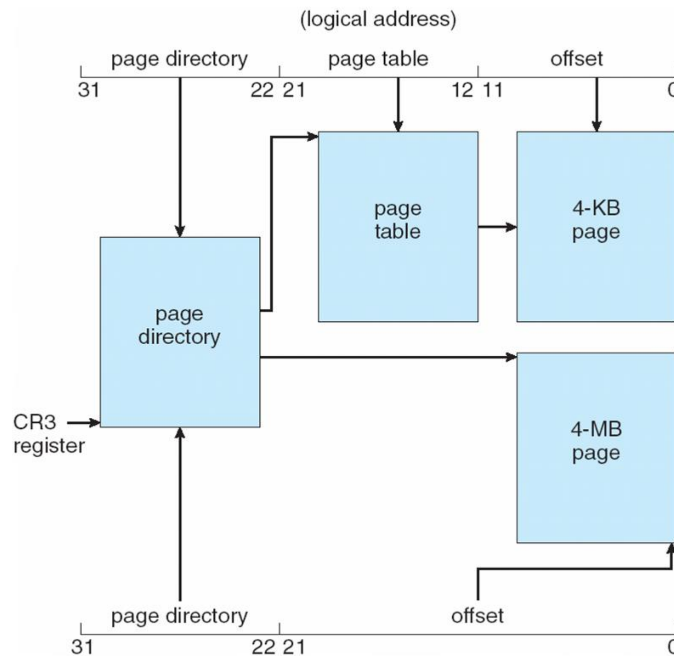
- IA-32 architecture: Pentium CPU a $32bit$
- IA-64 architecture: attuali Intel CPU a $64bit$

Il processo di paginazione permette la seguente traduzione:

$$linear\ address\ (32bit) \overset{paging}{\Longrightarrow} physical\ address\ (32bit)$$

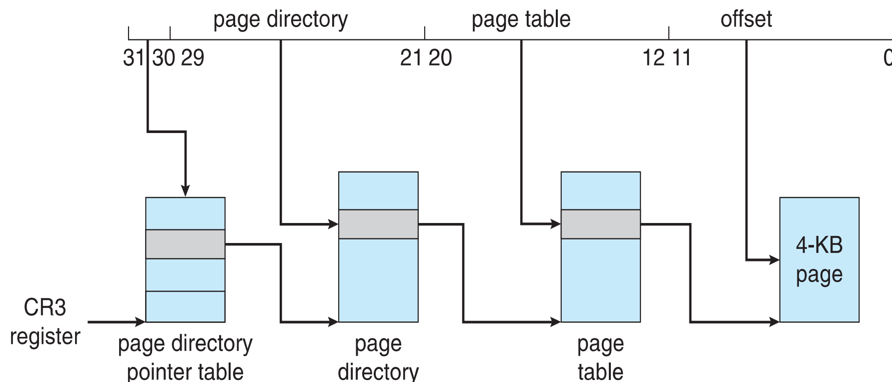
Esso può essere eseguito in uno o due livelli:

- un livello: la *Page Table* è detta *Page Directory* e le pagine hanno dimensione $4MB$
- due livelli: si ha una *Outer Page Table* + una *Inner Page Table* e le pagine hanno dimensione $4KB$



L'unica particolarità è che i processi di segmentazione e paginazione sono effettuati in cascata frapponendo tra essi un indirizzo logico sempre su $32bit$. La scelta del segmento specifico dovrebbe diminuire la lunghezza dell'indirizzo risultante, ma è stato scelto di riportarlo a $32bit$ (rendendo la segmentazione un passaggio da un indirizzo ad un altro). Questo per permettere di combinare in vari modi tali operazioni.

La Intel definì il **PAE (Page Address Extension)** per permettere ad app su $32bit$ di accedere a più di $4GB$ di spazio in memoria. Si tratta di un sistema su più livelli.



Virtual memory

La *virtualizzazione* è una tecnica avanzata di gestione della memoria centrale.
La *Demand Paging* è la paginazione a richiesta, è il principale argomento di questo capitolo. Si basa sull'idea di stabilire una corrispondenza tra pagine logiche e frame fisici solo per le pagine che sono realmente necessarie e solo quando sono richieste.
Gli altri argomenti sono aspetti particolari che caratterizzano la paginazione a richiesta.

Il termine *virtuale* indica qualcosa di ancora più evoluto, rispetto a ciò che è identificato come *logico*.
La differenza è che:

- **indirizzo logico**: corrisponde sempre un indirizzo fisico
- **indirizzo virtuale**: corrisponde ad un indirizzo fisico solo se la pagina è effettivamente presente in memoria

Il termine *paging* intende il rimpiazzamento di pagine per caricare in memoria una pagina richiesta.

Background

Con il termine *background* si indicano le motivazioni che portano alla definizione del concetto di *paginazione a richiesta*.
Il codice dev'essere in memoria per essere eseguito, ma spesso un programma NON serve sempre nella sua interezza. Ci sono parti del programma che non verranno mai attivate (*routine* di gestione dell'errore o di casi particolari, strutture dati grandi di cui serve solo una porzione). Inoltre è facile l'intero programma sia utilizzato un po' alla volta.
La soluzione è quindi di lavorare su un programma parzialmente caricato in memoria, in modo:

- permanente
- definitivo

La principale conseguenza è che un programma, per andare in esecuzione, non è più il vincolo di dover essere tutto contenibile in memoria. La virtualizzazione permette di ignorare i vincoli di memoria fisica: un programma di dimensione N può girare su una memoria fisica di dimensione M, con $N > M$.
Questo può aumentare il grado di parallelismo e migliorare molte caratteristiche, senza riempire inutilmente la RAM ed evitando il tempo di I/O per portare in memoria delle pagine inutili.

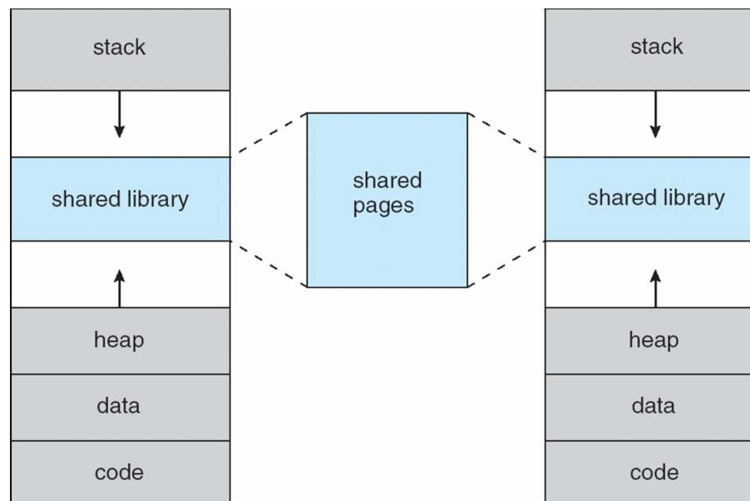
La **virtual memory** è una separazione più netta tra spazio di indirizzamento logico e fisico. Solo una parte del programma dev'essere in memoria per l'esecuzione, quindi il *logical address space* può essere molto maggiore del *physical address space*.

Virtual address space

Il **virtual address space** è la visione logica di come i processi sono salvati in memoria. Ogni processo vede il proprio spazio di indirizzamento partire da 0 ed essere contiguo.
Sono ancora presenti i frame e la MMU; l'unica innovazione è che una pagina può essere solo opzionalmente in RAM.

Shared library

È possibile utilizzare i buchi lasciati nello spazio di indirizzamento per inserire pagine condivise:



Demand paging

Una pagina viene portata in memoria solo quando questa è necessaria. Si hanno i seguenti vantaggi:

- meno I/O richiesto
- meno memoria richiesta
- risposta più rapida

Si ha una similitudine con il processo di *swapping*, solo che invece di operare su interi processi, il **demand paging** opera solo sulle pagine.

Una pagina è richiesta quando si ha un riferimento ad essa. Il riferimento *invalid* può indicare due situazioni distinte:

- la pagina non esiste
- la pagina è ancora in *backing store*, bisogna quindi portarla in memoria

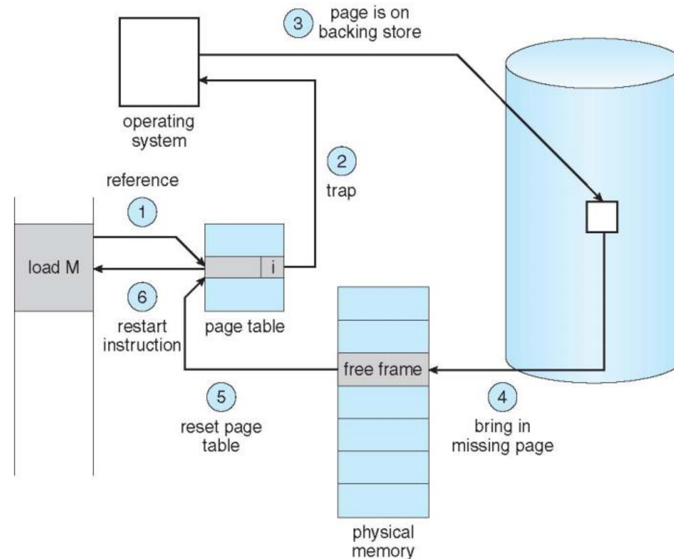
Questo è l'utilizzo del *validity bit* nella tabella delle pagine. La sua funzionalità, con l'utilizzo della memoria virtuale, è quindi estesa anche ad indicare pagine esistenti, ma in quel momento non accessibili (perché memorizzate in *backing store*).
Il **lazy swapper** applica la politica che nessuna pagina è caricata in memoria finché questa non è richiesta.
Il **pager** è il modulo che si occupa di gestire la singola pagina. Il *pager* effettua anche una predizione delle pagine che saranno richieste nel futuro prossimo, ed è quindi opportuno portarle in memoria.

Validity bit

Il **validity bit** può indicare pagina valida, oppure non valida. Si hanno però dei significati distinti per *invalid*, riassumibili come segue:

- *invalid* (I)
    - non presente

Il processo completo per gestire un *Page Fault* è schematizzato come segue:



Aspects of Demand Paging

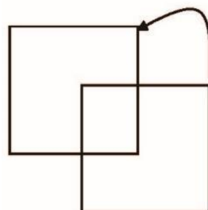
Nel caso estremo, detto **pure Demanding Paging**, un processo può iniziare senza alcuna pagina in memoria. Sarà la prima istruzione eseguita a sollevare il *Page Fault* e a richiedere quindi di caricare una pagina in memoria.

Un'istruzione data può eccedere a più pagine (lettura/scrittura a cavallo di due pagine), quindi generare potenzialmente *Page Fault* multipli.
La **località dei riferimenti** intende che molto probabilmente una porzione di un programma acceda alla medesima porzione di memoria.

È richiesto dell'hardware specifico per gestire la paginazione a richiesta. La *MMU* dev'essere modificata appropriatamente:

- la *Page Table* deve contenere il *validity bit*
- gestione della memoria secondaria (disco)
- gestire la *instruction restart*

La **instruction restar** si ha quando un *page swapping* ha parziale *overlapping* (sovrapposizione) tra spazio sorgente e spazio destinazione. Il sistema operativo deve gestire appropriatamente situazioni simili.



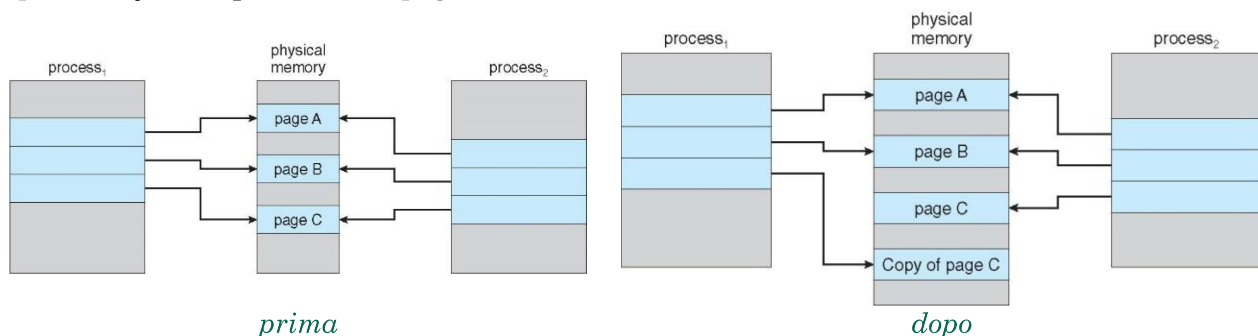
- tutte le pagine non modificabili (*read-only*) non sono riportate in *backing store* (annullato lo *swap out*)
- sui sistemi *mobile* non si effettua *swapping*, piuttosto si accede direttamente al disco (no *backing store*)

Le principali tecniche applicano più/meno o in modalità diverse il *backing store*, anche con tecniche molto differenti tra di loro.

Copy-on-Write

In caso di `fork()`, la **Copy-on-Write (CoW)** permette a entrambi processo padre e figlio di condividere inizialmente lo stesso spazio in memoria. Saranno poi duplicate le pagine solamente nell'istante in cui uno dei due esegue una modifica ad essa.

Il processo $process_1$ modifica la pagina C:



prima *dopo*

Le pagine libere devono essere riempite di zeri. Il *pool di free-frame*, detto **zero-fill-on-demand** *pages pool*, deve sempre possedere almeno una *free-frame* libera, disponibile per essere utilizzata.

Page replacement

Quando non sono disponibili *free frame*, bisogna liberare un *frame* già allocato, eliminando la pagina che questo contiene.

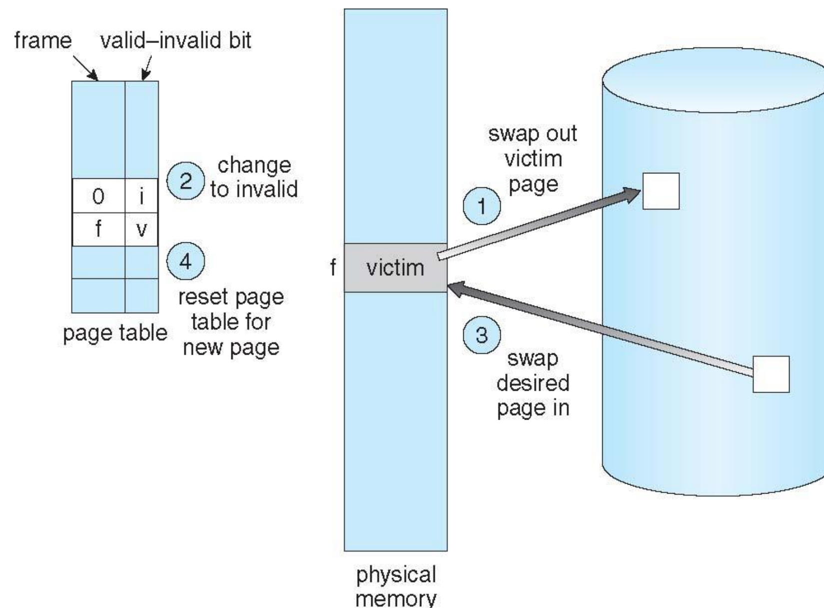
Un'analisi importante è analizzare quanto spazio allocare per ogni processo. Una prima alternativa serebbe quella di assegnargli preventivamente un *set di frames* e sarà il processo stesso a gestirsi il suo spazio. La seconda alternativa è permettere ai processi di prelevare frame ad altri processi.

Il **Page Replacement** è il processo di ricerca di una pagina in memoria, ma non realmente in uso, per selezionarla come bersaglio dello *swap out* per creare una *free frame* necessaria allo *swap in* di un'altra pagina.

L'*algoritmo di Page Replacement* deve prevenire la over-allocation di memoria, introducendo il *Page Replacement* nella routine di servizio del *Page Fault*.

Usando il *Basic Page Replacement* si hanno potenzialmente 2 trasferimenti di pagina per ogni *Page Fault*: uno sicuro (da disco a pagina) mentre l'altro dipende (da *victim frame* a disco, solo se *dirty*). Fondamentale sarà il costo computazionale del *page replacement algorithm*.

Il seguente schema riporta graficamente i passaggi (numerati sequenzialmente) necessari per effettuare un *Page Replacement*.



Il frame *f* è scelto come vittima (*victim frame*) ed è quindi effettuato uno *swap out* su di lui (scrivendolo su disco se necessario). Il frame *f* sarà rimpiazzato, in memoria, dalla nuova pagina che si desidera portare in RAM, ma non prima di aver aggiornato opportunamente la *Page Table* settando I (*invalid*) la entry relativa alla vecchia pagina *0* mappata su *f*.

Page replacement algorithms

Sono possibili diversi Page and Frame Replacement algorithms, ognuno gestisce in modo diverso le criticità legate al processo di sostituzione pagine.
Gli aspetti da tenere in considerazione sono i seguenti:

- **frame-allocation algorithm**: determina
  - quanti *frames* assegnare ad ogni processo (statico o dinamico)
  - quale *frame* rimpiazzare
- **page-replacement algorithm**: ottimizzato per minimizzare il rate di *Page Fault*, sia sul primo accesso che su quelli consecutivi

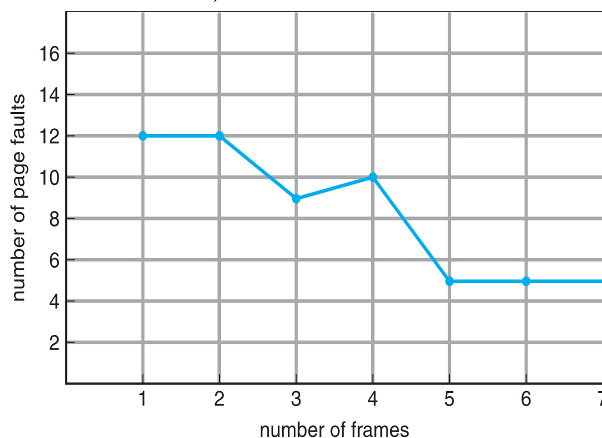
Alcuni *algoritmi di frame allocation* prevedono un'assegnazione statica dei *frames* ad un processo (numero fisso e costante), mentre altri comportano un'assegnazione dinamica (il numero di *frames* per ogni processo è variabile). Essi definiscono inoltre quale *frame* selezionare per essere rimpiazzato.

Per **testare** la bontà di un algoritmo è possibile effettuare una prova sperimentale, simulandolo su una specifica sequenza di riferimenti a pagine, detta **reference string**. Essa non è una stringa

> La chiamata dell'ultimo 7 comporta l'eliminazione della pagina 0 del *resident set*, salvo poi richiederlo immediatamente dopo, pagando un costoso *Page Fault*. In questo caso la predizione di futuro utilizzo è stata pessima.
> Ogni vettore azzurro rappresenta il *resident set* in caso di un *Page Fault*, per un totale di 15 *Page Faults* avvenuti.

L'algoritmo FIFO funziona bene per accessi dotati da **elevata sequenzialità**, cioè nei quali la pagina più vecchia è veramente quella più improbabile da riutilizzare.
Nella maggior parte dei casi, però, quello cronologico è un criterio di selezione troppo grossolano e approssimativo.
Risulta però di facile implementazione: si tratta di una struttura dati con estrazione in testa ed inserimento in coda. Il tutto con un solo puntatore e complessità $O(1)$.

La **anomalia di Belady** contraddice l'inversa probabilità tra $\#frames$ e $\#PageFaults$ affermando che non è sempre garantita la diminuzione dei *Page Faults* all'aumentare dei *frames* disponibili.
Aumentare il numero di *frames* è una soluzione **greedy**: sensata, ma non ottima (perché dipende esclusivamente dalla predizione sul futuro).



2. Optimal algorithm

L'**algoritmo ottimo** è tale perché, pur essendo *greedy*, è dimostrabile che non è possibile fare di meglio. Tale algoritmo sceglie ogni volta come vittima la pagina in base ad un'osservazione sul futuro. Questo è il motivo per cui tale algoritmo è solo ipotetico e non realizzabile: è impossibile conoscere il futuro.
In una macchina reale non si può prevedere il futuro a meno di programmi predicibili, cioè soddisfacenti entrambe le condizioni:

- strettamente sequenziali: no cicli o costrutti condizionali dipendenti dai dati
- noto l'input: noti a priori i dati immessi:

Esempio:

> Data la stessa *reference string* dell'esempio precedente:
> 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

L'implementazione della LRU è molto meno immediata rispetto a quella della FIFO. Sono possibili due approcci:

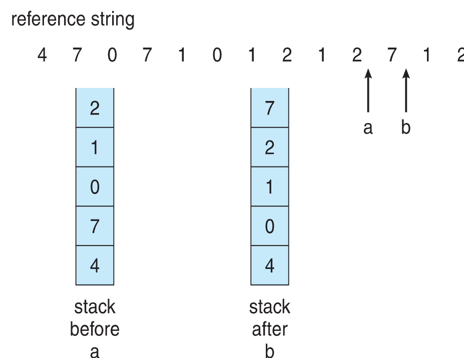
- **counter implementation**: ad ogni pagina è associato un tempo di ultimo accesso, memorizzato in un apposito contatore. Il clock avanza ad ogni accesso in memoria e l'accesso ad una pagina comporta l'aggiornamento, al valore attuale del clock, del suo contatore. Per selezionare la vittima è necessario scandire la tabella per trovare il contatore di valore minore (richiesta più tempo addietro).
- **stack implementation**: non memorizza i tempi, ma crea uno stack già ordinato secondo i tempi di ultimo accesso. Esso è fatto da una lista doppio linkata per rendere $O(1)$ l'estrazione di un elemento. Ad ogni accesso ad una pagina, questa viene portata in testa modificando 6 puntatori (2+2 per elemento precedente/successivo + 1 per head + 1 per tail).

La counter implementation è infattibile perché richiede una scrittura ad ogni accesso in memoria (aggiornamento del valore del contatore), anche in assenza di Page Fault. Inoltre l'algoritmo di ricerca rischia di essere lineare ($O(n)$).
La stack implementation ha un costo di aggiornamento maggiore della semplice scrittura di un contatore, ma portare in testa l'elemento designato come vittima permette di annullare completamente i tempi di ricerca.

Gli algoritmo LRU e OPT sono casi di **stack algorithms**, quindi non sono affetti dalla *Belady's anomaly* ed è quindi aumentare il numero di *frames* disponibili da garanzia di aumento delle prestazioni.

Esempio:

> Il seguente esempio (non mostrati i puntatori) mostra due istantanee (ai tempi a e b) di una *stack implementation*. È possibile notare come, al tempo a, la pagina più recente sia la 2, mentre al tempo b la pagina 7 è estratta dal penultimo posto e portata in testa facendo slittare tutto in basso.



Entrambi gli approcci proposti per il *LRU algorithm* sono lenti e necessitano di un hardware dedicato. Non vengono quindi utilizzati e si preferiscono algoritmi, detti **LRU approximation algorithms**, che cercano di approssimare tali situazioni.
Tali algoritmi, invece di identificare in modo puntuale la singola pagina non usata da più tempo, si limitano a distinguere due gruppi di pagine: utilizzate di recente e non utilizzate di recente.

> Se tutte le pagine avessero $reference\ bit = 1$ sarebbe necessario compiere un intero giro prima di trovare la vittima (caso peggiore). La complessità è quindi compresa tra $O(1)$ e $O(n)$.

c. Enhanced second-chance algorithm

L'**enhanced second-chance algorithm** è un'ulteriore miglioria: si introduce il **modify bit** che notifica il fatto che la pagina sia stata modificata o meno rispetto all'originale (funzione analoga al *dirty bit*). Una pagina non modificata è una vittima migliore perché non dev'essere ricopiata sul disco.
Prendendo la coppia ordinata di bit $(reference; modify)$, si hanno le seguenti situazioni:

- (0,0): pagina né modificata e né utilizzata ⟹ miglior vittima
- (0,1): non usata di recente, ma modificata ⟹ non così buona perché va aggiornato il disco
- (1,0): usata di recente, ma intonsa ⟹ sarà probabilmente riutilizzata a breve
- (1,1): usata di recente e modificata ⟹ sarà probabilmente riutilizzata e va aggiornato il disco

La preferenza per la scelta della vittima segue esattamente l'ordine mostrato sopra. Si utilizza uno schema a *clock* (*buffer circolare*) per iterare più volte alla ricerca di uno (0,0) prima, di un (0,1) in sua assenza e così via. Lo svantaggio di tale approccio è che potrebbero essere necessari fino a quattro giri completi del *buffer*, prima di individuare la vittima sacrificabile.

d. Counting algorithms

Anche i **counting algorithms** tentano di approssimare il *LRU algorithm*, ma con una variante rispetto agli approcci precedenti. Invece di memorizzare il tempo dell'ultimo accesso, misurano la frequenza degli accessi. Si mantiene un contatore di riferimenti ad una certa pagina e la vittima è quella a cui si è fatto accesso più/meno frequentemente:

- **Least Frequently Used (LFU) algorithm**: viene rimpiazzata la pagina col contatore più piccolo perché considerata poco interessante e con i riferimenti più dilazionati nel tempo
- **Most Frequently Used (MFU) algorithm**: viene rimpiazzata la pagina col contatore più grande, per preservare quella appena arrivata (e quindi molto utile)

Entrambi i metodi sono utilizzabili, ma condividono gli stessi svantaggi del *LRU algorithm* classico: necessità di un contatore per ogni *frame*, necessità di un accesso in memoria ad ogni riferimento e ricerca del minimo/massimo ad ogni *Page Fault*.

4. Page-buffering algorithms

I **page-buffering algorithms** si basano sull'idea di mantenere:

- un pool di *free frames*: in caso di *Page Fault* si ha una parte del lavoro già compiuto
- una lista di pagine modificate: salvare tali pagine sul *backing store* in momenti opportuni (stato di *idle*)
- il vecchio contenuto dei *free frames*: quando una vittima viene sacrificata per creare posto, in realtà conserva il suo contenuto, finché non è veramente riutilizzata, per permettere di ripristinarne istantaneamente il contenuto al bisogno

Consideriamo sempre (come abbiamo fatto finora) che il numero di pagine assegnate a ciascun processo non vari durante la sua vita: una volta definito (secondo i criteri *fixed/priority*) rimarrà lo stesso fino alla sua terminazione.

1. Fixed allocation

L'**allocazione fissa** prevede di assegnare un numero fisso e regolare di *frames* a ciascun processo. Esistono due possibilità si assegnazione:

- **equal allocation**: dividere il numero di frame disponibili per il numero di processi attivi
- **proportional allocation**: allocare a seconda della dimensione del processo

Per la equal allocation è necessario escludere dal conteggio i *frames* già assegnati al SO e una porzione da tenere libera per costituire la *free frame buffer pool*. I restanti vengono divisi equamente a seconda del numero di processi.

Per la proportional allocation sono necessari alcuni parametri importanti del processo p_i:

- s_i: dimensione del processo p_i
- $S = \sum s_i$: somma degli spazi virtuali di tutti i processi
- m: numero di frames totali disponibili nel sistema

da cui è possibile definire lo spazio a_i assegnato al processo p_i:

$$a_i = \frac{s_i}{S} \cdot m$$

Esempio:

> Dati due processi (p_1 e p_2) con i seguenti valori:
>
> $m = 64$
>
> $s_1 = 10$
>
> $s_2 = 127$
>
> lo spazio dedicato a ciascun processo è il seguente:
>
> $a_1 = \frac{10}{127+10} \cdot 62 \approx 4$
>
> $a_2 = \frac{127}{127+10} \cdot 62 \approx 57$
>
> Si è utilizzato $(m - 2) = 62$ invece che m per tenere libero un *pool di free frames*.

2. Priority allocation

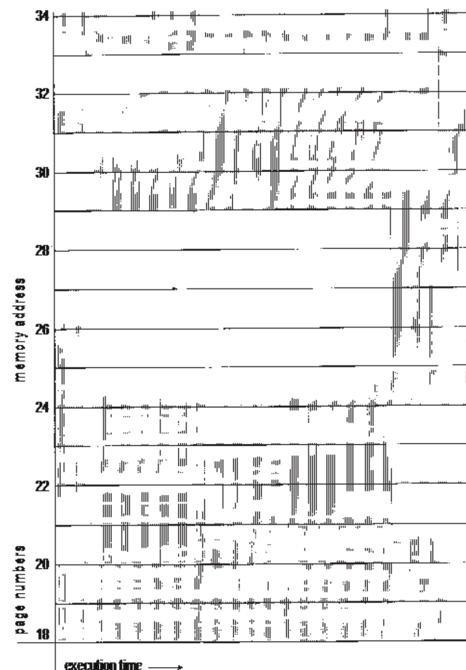
L'**allocazione prioritaria** è analoga ad uno schema di allocazione proporzionale, con la differenza che essa basa la scelta sulla priorità invece che sulla dimensione.
Tale priorità potrebbe essere definita internamente al SO oppure perché è importante che un determinato processo sia portato a compimento prima di altri.

Global vs. local allocation

Finora abbiamo considerato un *Page Replacemnt* locale, cioè scegliendo la vittima all'interno dell'elenco dei suoi frames.

Località

La memoria è utilizzata tanto meglio quanto aumenta la località, nel tempo, degli accessi in memoria di uno specifico processo.
È necessario definire un **locality model** per descrivere la località di accesso di ciascun processo. I processi possono migrare da una località ad un'altra (le località possono sovrapporsi).
La seguente rappresentazione riporta sulle ordinate l'indice di pagina e sulle ascisse l'asse temporale. Una linea marcata indica un accesso ripetuto in un piccolo intervallo di tempo.



In ogni istante la **località** di un processo è l'insieme di indirizzi a cui sta facendo accesso nel dato intervallo temporale. Ogni processo, spesso, accede ripetutamente alle stesse pagine per poi spostarsi su un'altra località.
Mantenere la località significa accedere alle pagine già in memoria, mentre migrare ad un'altra località significa la ricerca di una vittima.

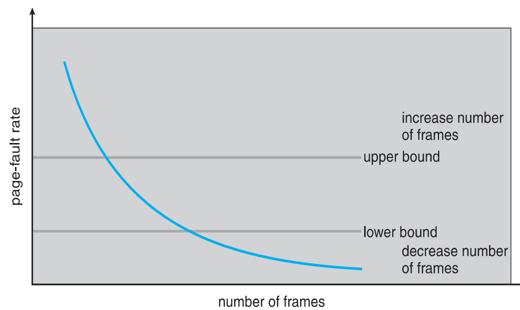
Working-set model

Dal concetto di *località* si sviluppò un modello che permette di fornire ad ogni processo le pagine che in quel momento sono la sua *località*. Il **working-set model** è una strategia di allocazione di pagine che si occupa di caricare dinamicamente in memoria la località di ogni processo.

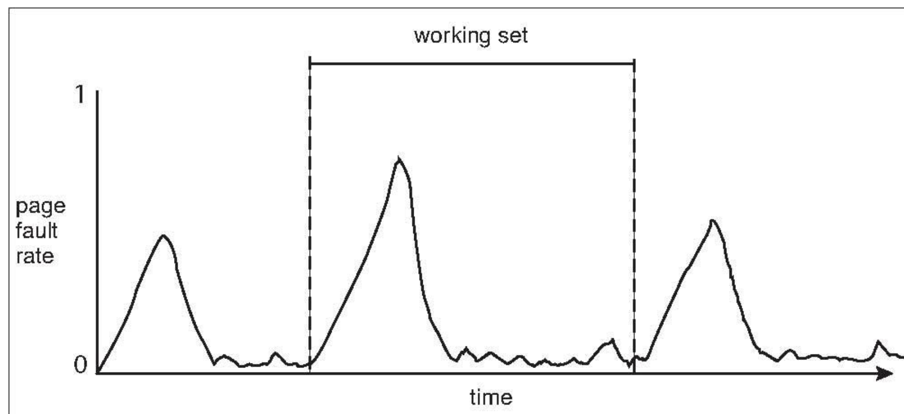
Si definiscono le seguenti quantità:

- Δ: *working-set window*
- WSS_i: *working set size* del processo P_i

Si definisce la località di un processo per un intervallo temporale Δ. Il **working set** è l'insieme di pagine referenziate dal processo P_i negli ultimi Δ istanti più recenti.
Il **WSS (Working Set Size)** rappresenta la cardinalità di un dato *working set*.
Il concetto base è di assegnare meno *frame* a processo più virtuosi (molto locali) e più *frames* ai processi più dispendiosi (meno locali).



Si avranno naturalmente delle variazioni tra *working set* e *reference set* allo scorrere del tempo. Si avranno delle situazioni più stazionarie e altre di variazione.



Nel capitolo "thrashing" si è parlato di politiche di allocazione dinamiche.

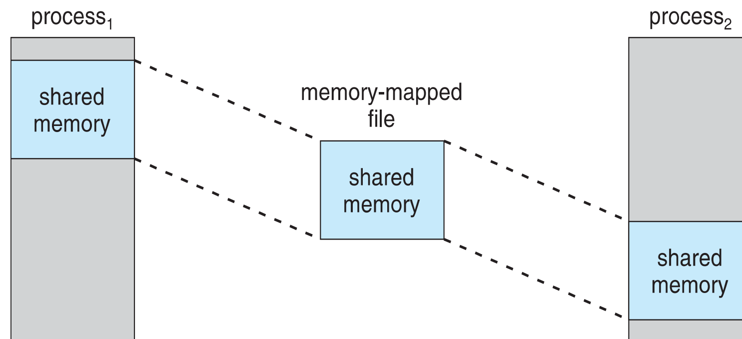
Memory-mapped files

Aspetto collaterale rispetto alla gestione della *memoria virtuale*, è la gestione delle *pagine virtuali* che in quel momento non sono in memoria, ma memorizzate sul disco. Leggere da file è estremamente dispendioso in termini temporali.
Spesso i dati di un file sono letti all'inizio di un programma, caricati in memoria, manipolati e infine salvati in memoria. Ma ci sono altri contesiti (es. database) in cui il lavoro è pesantemente *file-oriented*: il dato è salvato sul file in metà dell'esecuzione.
Per questo il SO fornisce strumenti di virtualizzazione di una parte di un file. Il **memory-mapped file** prevede che il file sia gestito come una parte della memoria, invece che letto ogni volta dal disco.

Viene mappato un `filepointer` come un vettore, non si utilizzano più quindi le funzioni `fread()`, `fscanf()` o `fgets()`, ma piuttosto un semplice accesso ad un vettore (`vett[i]`). Modificare il vettore implica automaticamente trasformare il file nel *file system*.
Il *memory mapping* delega al SO il compito di aggiornare il file ed è molto performante quando si aggiorna ripetutamente la stessa pagina di un file. Il SO implementa delle politiche per aggiornare il file sul disco solamente quando conveniente. Si hanno quindi tante modifiche e pochi *copy back*.

Shared memory

Il *memory mappeed I/O* è quindi utilizzabile per creare memoria condivisa tra due processi.



Per creare un *file mapping* si invocano le seguenti funzioni:

- Unix: `mmap()`
- Windows: `CreateFile()`, `CreateFileMapping()` e `MapViewOfFile()`

Allocating Kernel memory

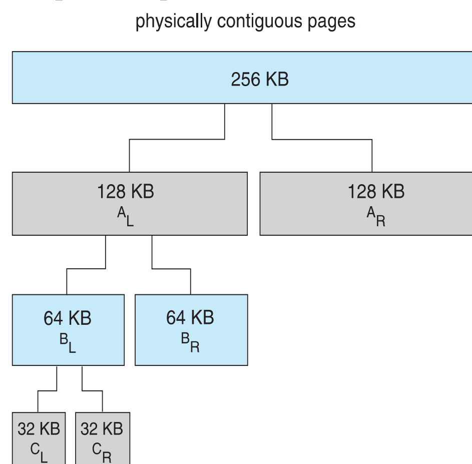
Gli argomenti precedenti erano riferiti all'allocazione per processi utente. Il kernel è gestito in modo differente: cerca di allocare spazio contiguo (non pagina) e necessita di strutture dati di dimensione variabile, anch'esse contigue.

Per il kernel esistono due principali strategie di allocazione:

- **buddy system**
- **slab allocator**

1. Buddy system

Si alloca memoria contigua per potenze di due (2^n). Si approssima ogni dimensione alla potenza di due immediatamente superiore. Si rischia di avere una frammentazione interna fino alla metà della dimensione allocata (caso peggiore).
Si spezzano i blocchi costituendo una gerarchia di allocazione consecutiva. Una `kmalloc()` di questo tipo implementa una gerarchia di questo tipo.



STORAGE MANAGEMENT

File-system interface

Il **file system** è la prima interfaccia in un Sistema Operativo: permette di individuare, aprire e spostare files.

Questo capitolo (*file-system interface*) si occupa di elencare le funzionalità offerte dal file system, mentre il capitolo successivo (*file system implementation*) si focalizza sull'implementazione interna di tale componente.
Al fine di questo corso, il capitolo attuale è semplice ripasso, mentre saranno preponderanti gli argomenti del prossimo.

L'implementazione del *file system* (idealmente semplice) è fortemente complicata dalla volontà di vedere, oltre ai dischi, anche i dispositivi di I/O. Il *file system* è quindi organizzato in diversi *layers* che permettono all'utente di vedere un unico schema (al più alto livello di astrazione) benché questo celi degli strati aventi schemi molto differenti (es. dischi ottici e magnetici hanno metodi di memorizzazione molto diversi).

File Concept

La struttura di un file è descrivibile secondo due punti di vista:

- colui che crea e usa un file
- colui che supporta il file su un disco

Il primo definisce la struttura e il contenuto di un file, il secondo si occupa invece di immagazzinarlo su un supporto elettronico. Sono due prospettive molto diverse.
Un file è sempre se stesso, qualunque tipo di *file system* sia utilizzato per immagazzinarlo (es. FAT32, EXT, ISO).

Dal punto di vista dell'utilizzatore, un file è uno spazio di indirizzamento contiguo, cioè una sequenza ordinata di dati.
Tali informazioni possono essere di diversi tipi:

- dati: file di dati (es. numerici, caratteri, binari)
- progrmmi: file eseguibile

Il contenuto del file è determinato da colui che lo crea secondo alcune convenzioni che determinano i tipi di file:

- *text file*
- *source file*
- *executable file*

Ne risulta che un **file** è una sequenza di dati di un certo tipi, codificata in un certo modo.

- `close()`: termina l'accesso ad un file

Tali operazioni sono spesso implementate tramite *system calls*.

Per effettuare l'operazione di file open, il sistema operativo memorizza le seguenti strutture dati:

- *open-file table*: tabella riportante gli identificatori dei file attualmente aperti
- *file pointer*: puntatore all'ultima locazione di lettura/scrittura del file (viene incrementato per accessi sequenziali)
- *file-open count*: conta il numero di thread che che simultaneamente hanno aperto tale file
- disk allocation of the file
- access rights

Il **file pointer** è un puntatore interno al file (riporta l'ultima posizione), mentre la struttura `FILE*` del C, punta ad una struttura dati contenente tutte le informazioni sul file (compreso il puntatore interno). Non bisogna quindi confondere le due accezioni di *file pointer*:

- puntatore interno: rappresenta una posizione del file e ha un corrispettivo fisico
- puntatore esterno: rappresenta il file intero e non si sposta durante lettura/scrittura

Il *file-open count* non conta tutte le operazioni di read/write, ma solo le `open()`. Le `close()` decrementano tale puntatore e questo meccanismo permette di rilasciare la struttura dati solo quando il file è realmente non più necessario.

Il file locking è un'operazione che permette di avere accesso multiplo al file (in concorrenza). Nel *file system* è possibile assegnare un lock specifico del file: per lavorare su tale file è necessario prima otteneme la *ownership*. Sono simili ai locks di read/write:

- **shared lock**: simile al read lock
- **exclusive lock**: simile al write lock

Gli *shared locks* non sono bloccanti, sono quindi "finti" locks che servono solo per dichiarare la volontà di lettura di un file. Questo permette di avere un programma più leggibile ed eventualemnte avere dei controlli aggiuntivi.

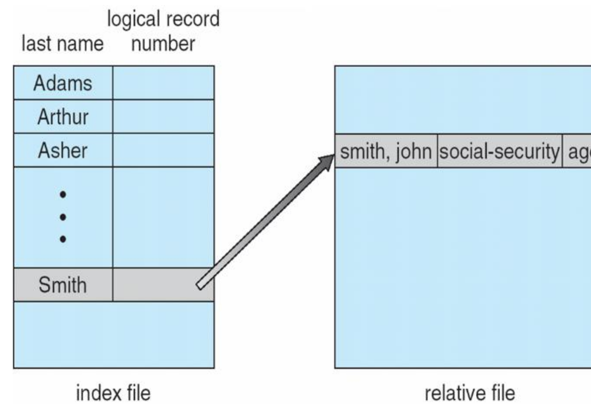
Example:

> Il seguente codice riporta un esempio di *file locking* sfruttando le API Java.
> Definite le seguenti variabili globali (per gestire lock esclusivo e condiviso):
>
> ```
> public static final boolean EXCLUSIVE = false;
> public static final boolean SHARED = true;
> ```
>
> è possibile sfruttarle nel seguente frammento di codice:
>
> ```
> FileLock sharedLock = null;
> FileLock exclusiveLock = null;
> try {
>     RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
>     // get the channel for the file
>     FileChannel ch = raf.getChannel();
>     // this locks the first half of the file - exclusive
>     exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
>     /** Now modify the data . . . */
>     // release the lock
>     exclusiveLock.release();
>     // this locks the second half of the file - shared
> ```

- *index header* interno: l'indice è contenuto nella porzione iniziale del file stesso

Per tutti i problemi di ricerca, l'utilizzo di un indice migliora notevolmente le prestazioni: si cerca l'informazione desiderata sull'indice e si ottiene direttamente il puntatore al blocco del file che la contiene.

Esempio:

> Il seguente schema rappresenta un'organizzazione ad indice, dove questo è memorizzato in un file esterno.
>
> 
>

Disk and Directory Structure

Una volta definiti i file e la loro organizzazione, analizziamo adesso come organizzarli su un disco/*file system*.

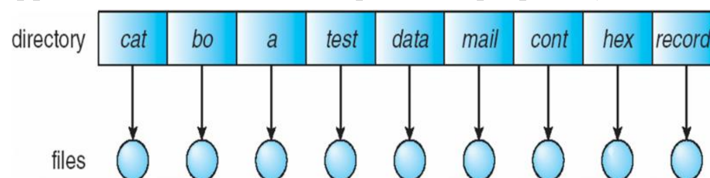
Directory structure

Le **directory** (direttorio o cartella) è un insieme di nodi contenenti le informazioni riguardanti tutti i files. Le *directories* sono quindi i modi di organizzare i file.
A seconda del SO in analisi esistono più strutture di *directories*:

- single-level directory
- two-level directory
- tree-structured directories
- acyclic-graph directories

Sia la struttura del direttorio, che i files, risiedono sul disco.

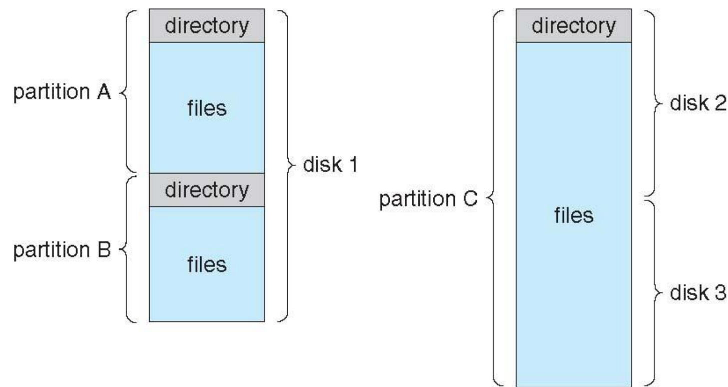
L'organizzazione single-level directory utilizza una sola directory per tutti gli utenti. Questa semplice organizzazione porta però problemi di **naming** (gestire la stringa identificativa di ogni file) e di **grouping** (raggruppamento di file secondo specifiche proprietà).



Disk structure

I files risiedono normalmente su dischi. Indipendentemente dalla tecnologia con cui è realizzato (es. magnetico, a stato solido, ottico) un disco è suddiviso in **partizioni**.

Creare due partizioni sullo stesso disco è come avere due dischi distinti, ma è anche possibile virtualizzare la stessa partizione su più dischi.



Introdurre questo *layer* (le partizioni) tra i *file system* e i dischi permette maggiore flessibilità, ma introduce in problema di organizzazione.

Operations performed on directory

Una directory dev'essere organizzata in modo tale da supportare le seguenti operazioni:

- search for a file
- create a file
- delete a file
- list a directory
- rename a file
- traverse the file system

L'operazione di ricerca maggiormente utilizzata è la ricerca.

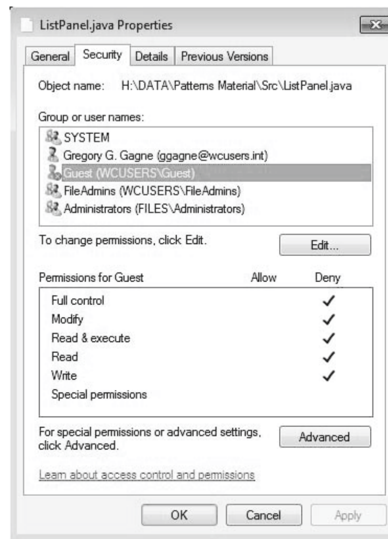
Directory organization

I direttori sono una tabella di ricerca gerarchica e permettono di reperire un file, a partire da un nome *human-readable*. Se si utilizzassero solo numeri per identificare i files, non sarebbero necessari i direttori. Noi umani necessitiamo invece di un'organizzazione gerarchica nella quale i file (identificati da stringhe) sono contenuti in cartelle (anch'esse identificate da stringhe).

L'organizzazione di directory è la l'implementazione della loro struttra, cioè la scelta della struttura dati necessaria e il conseguenete algoritmo di scorrimento.
Realizzare una struttura a direttori non è banale: per un'organizzazione ad albero è necessario implementare un albero n-ario. I SO moderni, inoltre, organizzano delle sovrastrutture per accedere più velocemente ai files (*cacheing*).

Esempio:

I permessi sono visibili, in Windows, graficamente tramite una *access-control list*:



e in Linux, da terminale tramite una *directory listing*:

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

In UNIX si hanno 10 carattei: il primo dice se si tratta di un file o di un direttorio, mente le altre nove sono i diritti di lettura (r), scrittura (w) ed esecuzione (x) rispettivamente per le tre entità (owner, group, public).

limitano a indicare i parametri (*drive*, *cylinder*, *track*, *sector*) per accedere, in lettura/scrittura, ad una specifica locazione del disco.

Il basic file system è abbastanza vicino al dispositivo e ha il compito di comunicare al *device driver*. Lo fa con un maggiore livello di astrazione, limitandosi a specificare il numero del blocco a cui vuole accedere. Sarà poi il *device driver* a tradurlo nelle rispettive coordinate fisiche.
Il controllo dell'I/O vede la geometria del disco, mentre il livello più basso del *file system* (*basic file system*), astrae da questo. In realtà, oggigiorno, tale *file system* potrebbe essere realizzato dal dispositivo stesso perché le geometrie sono più complesse e non standard.
Il *basic file system* si occupa anche di gestire i buffer (dati in transito) e le cache (dati usati di frequente).

Il file organization module conosce e comprende l'esistenza dei file e dei loro nomi. Esso effettua la traduzione tra indirizzi logico/fisici e gestice l'allocazione del disco.

Il logical file system gestisce metadati (`struct` e contenuti) che astraggono da come viene effettivamete realizzato il *file system*. I **file control blocks** (`inodes` in Unix) permettono di tradurre il file name nei corrispondenti *file number*, *file handler* e *location*.
Il *logical file system* gestisce anche i direttori e le informazioni di protezione.

File-System Implementation

Nell'implementare il *file system* occorre tener conto che le strutture dati che fanno riferimento ad un file devono essere consistenti quando viene spenta la macchina e anche che ci sono delle strutture dati aggiuntive quando il sistema è acceso.
I file e i direttori esistono anche quando il disco non è più alimentato. Al *mounting* di un *file system* su un sistema, a tali strutture dati si aggiungono delle strutture dati, in parte ridondanti, in RAM.

Additional structures on disk

Un disco contiene i seguenti elementi:

- *boot control block*
- *volume control block*
- *directory structure*
- *file control block*

Sul disco sono presenti dati effettivi (file), ma anche informazioni di organizzazione (elenco precedente).

Effettuare il **bootstrap** di un sistema significa caricare il sistema operativo. Questa operazione parte da informazioni prese dal disco, in particolare dal **boot control block**.
Esso è contenuto solitamente nel primo blocco del volume e sarà cercato dal SO al suo avvio (nell'ordine specificato nel BIOS).

tabella, assegnando un identificativo univoco generale (valido nell'intero sistema) ai file aperti da ogni processo. L'uso di due tabelle evita la ridondanza delle informazioni comuni in caso di accesso concorrente allo stesso file.
Per la figura a, si accede alla struttura dati della directory (piccola) che contiene la locazione del *file-control block* che a loro volta contengono l'inicazione e tutti i frammenti di dati che compongono il file desiderato.

La ricerca viene effettuata dal nome (*human readable*) del file al suo *file control block*. Questa operazione è effettuata nella `open()`. Le operazioni di `read()`/`write()` agiscono direttamente sul *file control block* che è stato precedentametne recuperato dalla `open()` e memorizzato in un'apposita tabella.

Directory Implementation

I direttori sono delle tabelle di ricerca che possono essere implementate nei seguenti modi:

- lista lineare: accesso lineare
- tabella di hash: accesso diretto

È necessaria però una tabella (lista/hash) per ogni direttorio. In totale si ha quindi un albero di tabelle di simboli.
In generale, per implementare i direttori, è necessario essere esperti di *tabella di simboli*: esse sono un metodo per cercare velocemente informazioni.

Allocation Methods

I **metodi di allocazione** riportano come i blocchi di dischi sono allocati per contenere i files. Essi descrivono quindi come i *data blocks* sono gestiti e raggiunti.

1. Contiguous allocation

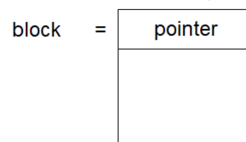
Il metodo più semplice per allocare files è l'**allocazione contigua**: ciascu file occupa un insieme contiguo di blocchi.
Tale soluzione offre le migliori *performance* in molti casi e sono necessari solo due parametri (blocco di partenza e lunghezza) per indirizzare un file. Presenta però problemi di frammentazione esterna e richiede compattazione (*off-line* o *in-line*).

Esempio:

> Il seguente esempio mostra un'allocazione contigua di files su disco e propone la *directory table* (anch'essa memorizzata sul disco) come contenente solo l'identificativo del file e le due informazioni per accedervi (*starting location* e *length*).

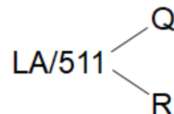
La variante **FAT (File Allocation Table)** utilizza sempre una lista linkata, ma scorpora i puntatori rispetto ai dati. Invece di utilizzare una `struct` contenente dati+puntatori, si mettono i dati da una parte e i puntatori da un'altra parte.
Per ogni file la directory punta al primo elemento della FAT, il cui indice corrisponde all'indice del *disk block* contenente i dati del file. Il blocco della FAT contiene il puntatore al successivo *FAT block* relativo a quel file (il cui indice corrisponde all'indice del *disk block*) e così via. All'interno della FAT si hanno quindi solo puntatori interni alla FAT e la relazione con i dati veri è propri è dato dall'indice dei blocchi della FAT.

L'implementazione di una lista linkata presuppone che non tutto lo spazio di indirizzamento sia occupato da dati: è necessario dello spazio per i puntatori.
Ogni file è una lista linkata di *disk blocks* che possono essere distribuiti ovunque all'interno del disco. Si paga però un po' di *overhead*: un blocco non è completamente utilizzato per i dati, ma sono necessari dei *metadati* (in questo caso il puntatore) per gestire la struttura dati.

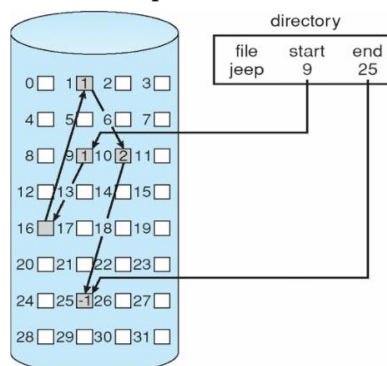


Il *mapping* può essere effettuato dividendo il *Logical Address* (*LA*) per la dimensione del blocco diminuita di un byte (*overhead* necessario per il puntatore al successivo blocco).

Esempio:

> Supponendo che i blocchi del disco abbiano dimensione $512byte$ e che un puntatore occupi $1byte$, dev'essere considerato solo lo spazio effettivo dedicato ai dati (la loro differenza). Il calcolo da effettuare risulta quindi il seguente:
>
> 
>
>
> Il blocco a cui fare accesso è il blocco Q-esimo nella catena linkata di blocchi rappresentanti il file. Il *displacement* interno al blocco, invece, è pari a $(R + 1)$.

Per la linked allocation, nella tabella di direttori, invece di riportare indirizzo di partenza e dimensione (allocazione contigua), vengono riportati il puntatore al primo blocco e all'ultimo. Per passare dalla head alla tail si scorrono tutti i puntatori.



Tale schema di allocazione, simile a quello utilizzato in memoria principale, permette accesso diretto. È necessaria una *index table* ed è permesso un accesso dinamico senza frammentazione esterna, ma si ha *overhead* per i blocchi indici.

Il *mapping* può essere effettuato dividendo il *Logical Address* (LA) per la dimensione del blocco. Si utilizzano alcuni blocchi per contenere interamente la *index table*.

Esempio:

> Supponendo $256KB$ la dimensione massima del file e i blocchi di dimensione $512byte$, è necessario un solo blocco per contenere la *index table*. Il calcolo da effettuare è quindi il seguente:
>
> $$LA/512 \rightarrow Q, R$$
>
> Il valore Q indica il *displacement* all'interno della *index table* necessario per ottenere l'informazione del blocco desiderato. Invece il valore R rappresenta il *displacement* interno al blocco necessario per individuare l'inizio del file.

Esempio:

> Per mappare (da logico a fisico) gli indirizzi di un file avente una lunghezza non nota a priori è necessario utilizzare uno schema linkato (*index table* composto da blocchi linkati) perché non ci sono limiti di dimensione (impossibile conoscere a priori il numero di blocchi da dedicare alla *index table*).
> Supponendo i blocchi di dimensione $512byte$, i calcoli da eseguire sono i seguenti:
>
> $$LA / (512 \times 511) \rightarrow Q_1, R_1$$
>
> Il valore Q indica il blocco della index table contenente le informazioni di interesse. Il valore R_1 è invece utilizzato per eseguire il seguente ulteriore calcolo:
>
> $$R_1 / 512 \rightarrow Q_2, R_2$$
>
> Il valore Q_2 indica il *displacement* dentro il blocco della *index table* (precedentemente individuato), mentre il valore R_2 indica il *displacement* all'interno del blocco del disco, per identificare l'inizio del file di interesse.

Se non è sufficiente un *blocco di indici* per indirizzare tutti i *data block*, allora è possibile:

- linked scheme: tabella indici che punta ad una lista di blocchi indici
- two-level index: tabella a due livelli

L'utilizzo degli *inode* è adatto, sotto il profilo delle performance, per gestire tutti i tipi di files, da quelli più piccoli a quelli più grandi, senza appesantire eccessivamente la struttura. Questo perché file di piccole dimensioni utilizzano direttamente i *direct blocks* (annullando l'*overhead*) mentre i puntatori indiretti sono utilizzati solo quando strettamente necessario.

Performance

Il miglior tipo di accesso dipende dal tipo di accesso che è necessario effettuare. Si hanno le seguenti caratteristiche:

- allocazione contigua: ottima sia per accesso sequenziale che diretto
- allocazione linkata: buona per accesso sequenziale, ma non supporta quello diretto
- allocazione a index: più complicata e va bene per situazioni in cui c'è una predominanza di accesso diretto.

Le performance dell'accesso a file sono fortemente dipendenti dagli accessi al disco, perché questi sono molto lunghi. Mentre l'accesso alla RAM ha una frequenza vicina a quella del clock GHz, l'accesso al disco ha ordini di grandezza minori (quindi più lenti). Gli SSD hanno prestazioni nettamente differenti (migliori).

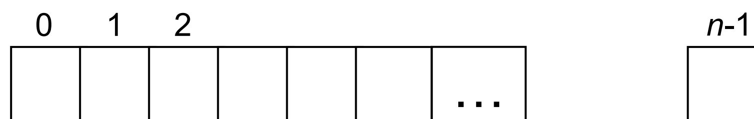
Free-Space Management

Lo spazio libero si può gestire:

- bitmap: vettore di bit utilizzati come flag (blocco libero/occupato)
- free list: lista linkata di blocchi liberi

1. Bitmap

In RAM non esiste accesso diretto al bit, quindi le **bitmap** sono in realtà emulate tramite intere word. La bitmap è più compatta, di 8 volte, rispetto l'utilizzo di un vettore di byte.
Supponiamo, in questa sede, che la bitmap sia implementata come un vettore di words. Il valore 1 indica che il blocco è libero, mentre lo 0 indica un blocco occupato.



$$bit[i] = \begin{cases} 1 \Rightarrow block[i]\ free \\ 0 \Rightarrow block[i]\ occupied \end{cases}$$

Per reperire il numero del blocco libero (`1`) che è possibile utilizzare, è necessario applicare il seguente calcolo:

$$(number\ of\ bits\ per\ word) \cdot (number\ of\ 0\text{-}value\ words) + (offset\ of\ first\ 1\ bit)$$

Le CPU possiedono delle istruzioni interne per trovare word completamente piene di `0`, quindi è facile individuare word che abbiano anche solo un `1`. È però necessario effettuare una scansione lineare per trovare blocchi contigui liberi. Per fortuna, in questo, la bitmap è molto rapida.

Il **grouping** significa allocare blocchi contenenti puntatori ai prossimi blocchi liberi. Prevede di modificare la lista linkata in modo da avere allocazione a blocchi. Vengono memorizzati gli indirizzi ai prossimi $(n-1)$ free blocks, nel primo free block, insieme al puntatore ai prossimi blocchi che contengono puntatori ad ulteriori free blocks.

Il **counting** mantiene il puntatore al primo dei blocchi liberi e il contatore dei blocchi consecutivi liberi. Tale soluzione è quindi più orientata al contiguo perché si ha una lista di intervalli contigui di blocchi liberi.

Dagli schemi di gestione precedenti (bitmap e free list), sono stati proposti schemi ancora più complicati, come gli **space maps**. Esso è utilizzato nel file system ZFS e si basa su metaslab (slab variati).

Efficiency and Performance

L'efficienza dipende dai seguenti fattori:

- algoritmi di allocazione nel disco e ricerca nei direttori
- organizzazione delle entry dei singoli direttori
- organizzazione dell'allocazione dei blocchi per i file
- dimensione fissa o variabile delle strutture dati

I moderni sistemi operativi implementano le seguenti metodologie per migliorare le prestazioni:

- dati e metadati vicini: mantenere dati e metadati (FCB e inode) in blocchi adiacenti del disco per effettuare accesso rapido
- *buffer cache*: inserire un buffer in RAM, contenente copie dei blocchi di disco utilizzati frequentemente
- accessi sincroni vs asincroni: si tratta di due gestioni opposte: la lettura sincrona attende che la read sia completata mentre quella asincrona esegue immediatamente l'istruzione successiva (viceversa per la scrittura)
- *read-ahead*: tecnica che prevede la lettura anche dei blocchi successivi a quello richiesto, per ottimizzare accessi sequenziali

Page cache

La **page cache** è il modulo del sistema operativo che gestisce pagine libere, corrispondenti a blocchi su disco, per fare *memory-mapped I/O* o simili.
Il **buffer cache** è invece un modulo che tiene in memoria copie di blocchi di memoria.

Il SO utilizza quindi il seguente schema per effettuare I/O: prima di scrivere su disco, controllare se questo blocco non sia presente in *buffer cache*.

NFS

Il **NFS (Network File System)** è l'implementazione e specificazione di un sistema software per accedere a file remoti attraverso LAN (Local Area Network) o WAN (Wide Area Network).

- traccia periferica: più larga e può contenere più bits
- traccia interna: più corta e può contenere meno bits

Questa politica impone ai produttori hardware di dischi di complicare la mappatura tra settori/cilindri e byte, per fornire all'esterno un'interfaccia più comoda, attraverso la quale è sufficiente richiedere la posizione (indice) del singolo byte da ottenere.
Il disco possiede una logica interna e contiene al suo interno una serie di blocchi numerati.

Disk Scheduling

Un SO può accedere ad un disco o fornire funzionalità di accesso al disco. Le operazioni sul disco costano in termini prestazionali di tempistiche a causa di:

- velocità di trasferimento dei dati: tempo di lettura/scrittura dei dati, dovuto a ritardi meccanici (dischi magnetici) o elettromagnetici (stato solido)
- latenza rotazionale: tempo necessario perché, a causa della rotazione del disco, l'inizio del settore desiderato arrivi a trovarsi sotto la testina
- tempo di seek: tempo necessario a spostare radialmente la testina fino alla traccia
- tempo di posizionamento: somma di latenza rotazionale e tempo di seek

Essendo i tempi fisici interni al *device*, sono dipendenti dalla tecnologia implementata. L'unico punto che è possibile migliorare, è schedulare opportunamente più accessi consecutivi, per evitare che le testine percorrano troppa strada inutile.
Il **disk scheduling** è un processo di serializzazione degli accessi al disco, per minimizzare gli spostamenti tra blocchi. Vengono applicate le stesse politiche utilizzate per gli ascensori.

Analogamente allo scheduling della CPU, anche nel disk scheduling si ha un'unica risorsa condivisa tra più richieste. È quindi plausibile che in ogni istante ci siano molte richieste in attesa (stato di waiting).
Esistono più algoritmi per gestire richieste molteplici e concorrenti di richiesta di dischi I/O. Un sistema reale è dinamico: non è possibile prevedere le richieste future. In questo capitolo analizzeremo una situazione semplificata, operando su una stringa di richieste, note a priori.

Le principali politiche di *disk-scheduling* sono le seguenti:

- FCFS: first come, first served
- SSTF: shortest seek time, first
- SCAN: politica dell'ascensore

1. FCFS

L'**algoritmo FCFS (First Come, First Served)** è una semplice politica FIFO. Le richieste sono servite nellostesso ordine con cui giungono.
Tale politica è semplice da implementare, ma non ha grandi performance.

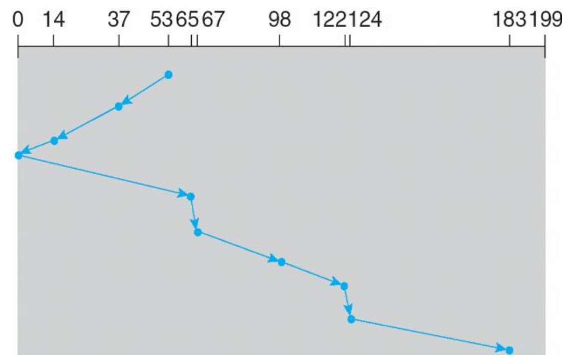
Esempio:

Data le seguenti posizione iniziale e elenchi di richieste:

$head_starting_position = 53$

$queue = 98, 183, 37, 122, 14, 124, 65, 67$

la rappresentazione grafica dei movimenti della testina è la seguente:



Varianti della politica dell'ascensore sono le seguenti:

- C-SCAN
- C-LOOK

a. C-SCAN

L'**algoritmo C-SCAN**, una volta servite tutte le richieste "in salita", prevede di riportare rapidamente la testina alla posizione iniziale (blocco zero) e riportire nuovamente a salire.
Rispetta alla SCAN, si evita di servire le richieste "in discess", si hanno così dei vantaggi a livello tecnologico.

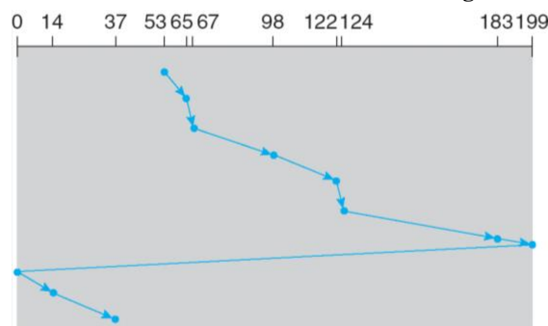
Esempio:

Data le seguenti posizione iniziale e elenchi di richieste:

$head_starting_position = 53$

$queue = 98, 183, 37, 122, 14, 124, 65, 67$

la rappresentazione grafica dei movimenti della testina è la seguente:

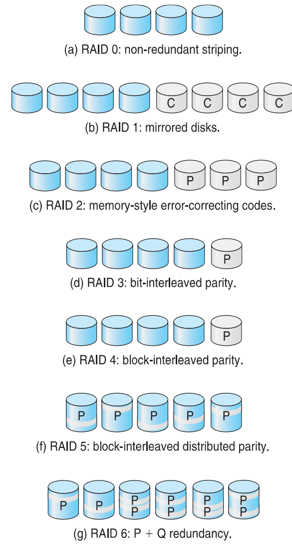


b. C-LOOK

L'**algoritmo C-LOOK** migliora ancora leggermente la C-SCAN perché, nel riportare la testina a condizioni iniziali, non raggiunge necessariamente lo zero. Si ferma invece alla richiesta avente indice più basso, dopo il completamente del movimento "in salita".

RAID Structure

La struttura **RAID (Redundant Array of Inexpensive Disks)** è una forma di ridondanza utilizzando un dispositivo virtualizzato da più dischi paralleli. Viene duplicata l'informazione su dischi diversi per garantire robustezza e affidabilità.



I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

La gestione per l'interazione con un periferico può essere effettuata nelle seguetni modalità:

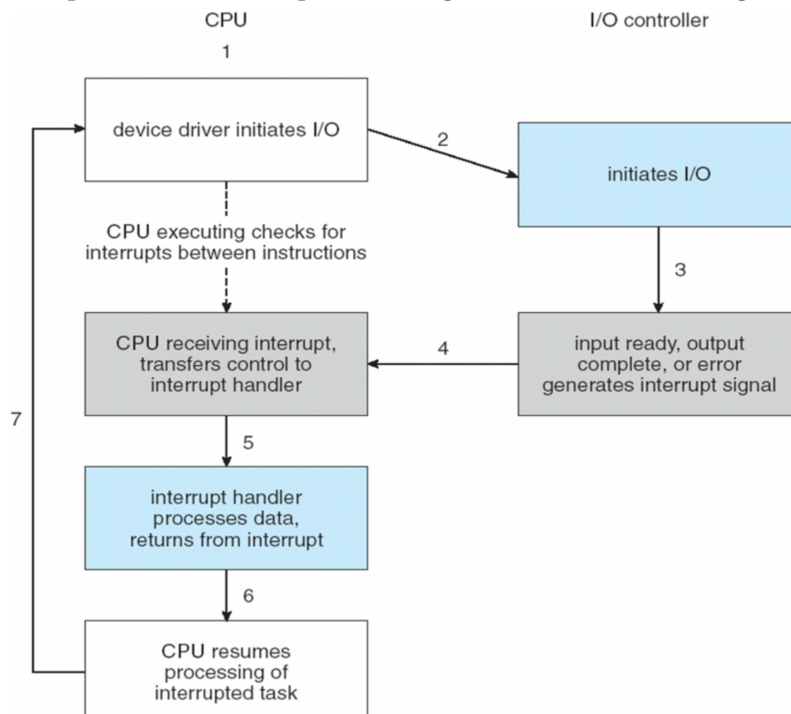
- polling
- interrupt
- DMA

1. Polling

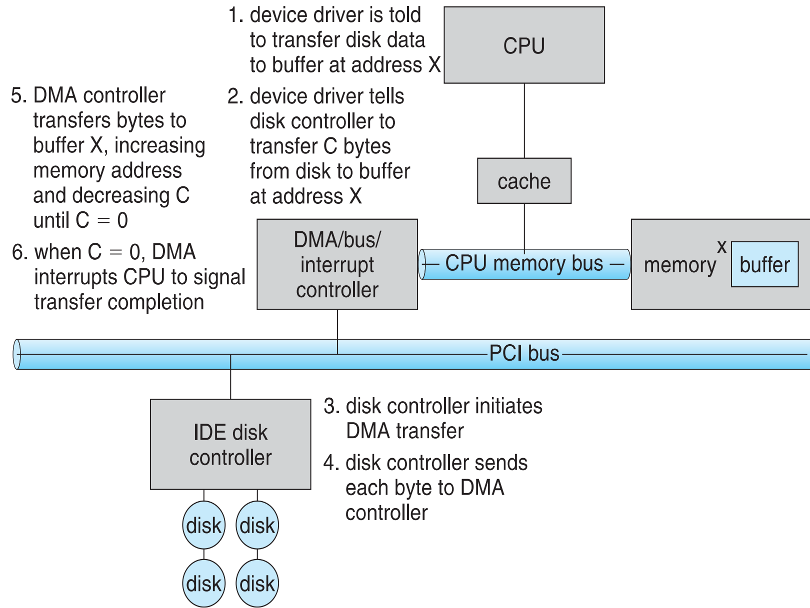
Il **polling** prevede la lettura costante e continua della *parola di stato* finché non è settato in modalità di avere dei dati pronti. Si consuma CPU time e impedisce al processore di eseguire altre operazioni perché la sua attenzione è completamente catalizzata dal controllo dello stato della periferica.

2. Interrupt

La modalità **interrupt** delega al priferico stesso di notificare alla CPU che il dato è pronto. Ad ogni periferica è collegato un numero di interrupt (problematica hardware).
Il **device-driver** è la parte si SO di più basso livello (più vicino all'hardware) che viene spesso scorporato e scritto dal produttore del dispositivo, seguendo determinate regole.



Gli step successivi per la gestione in DMA di un trasferimento è schematizzato come segue:

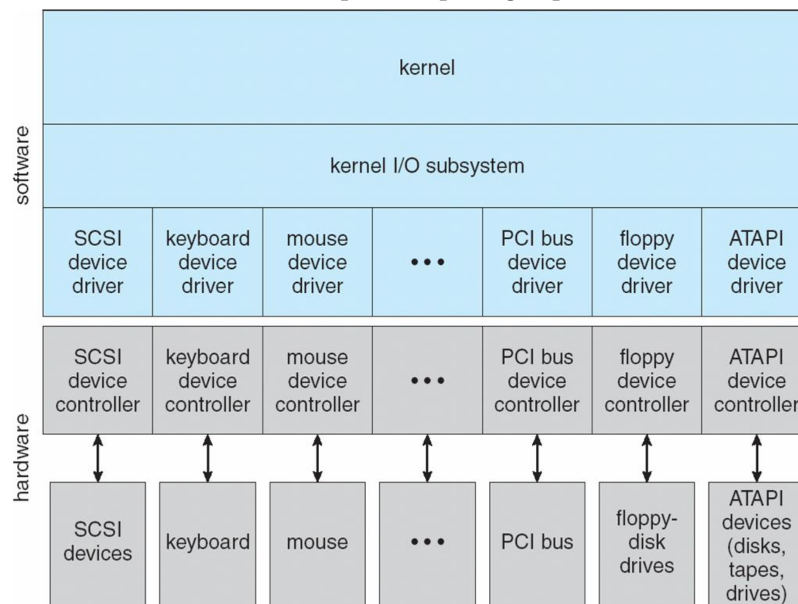


Application I/O Interface

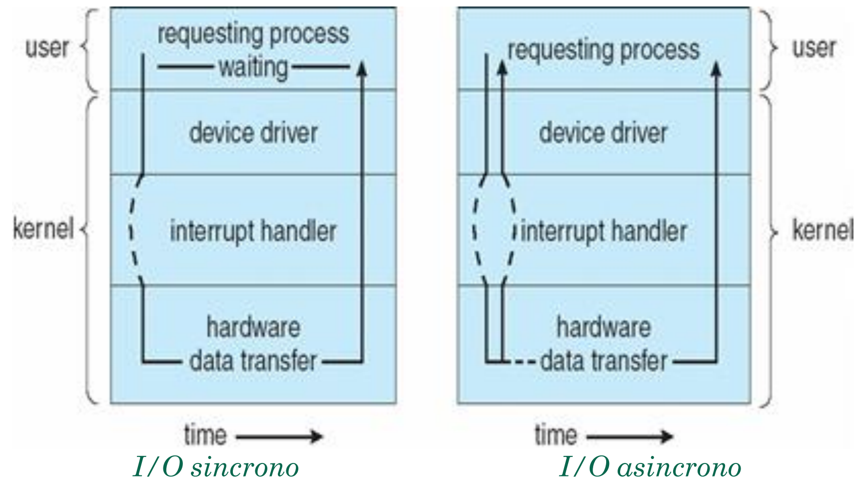
I sistemi operativi moderni mascherano i dispositivi di I/O quasi come se fossero dei file. I livelli alti del kernel, che gestiscono l'input/output, mostrano ai programmi utente delle `read()` e delle `write()`. Mentre a basso livello si distingue tra le varie periferiche, salendo di livello si introducono dei layer che uniformano i tipi di accesso e mostrano all'utente un'interfaccia di accesso unica (mischiando sia file che periferiche di I/O). Di conseguenza i file descriptor possono "nascondere" dei dispositivi periferici.

Kernel I/O structure

Lo schema sottostante mostra come il sottosistema del kernel per I/O gestisce tutti i dispositivi allo stesso modo. Sono i *device driver* ad essere specifici per ogni periferica a cui sono riferiti.



Dal punto di vista del flusso di esecuzione c'è differenza tra i due metodi di I/O:



I/O sincrono *I/O asincrono*

La differenza sta tra la discesa e la salita. Nella modalità sincrona il processo richiedente rimane in uno stato di *waiting*. In modalità asincrona, invece, il processo richiedente prosegue ad operare finché il kernel non gli notifica la terminazione della comunicazione richiesta.

Vectored I/O

L'**interrupt vettorizzato** è una modalità che permette di raccogliere insieme più operazioni di I/O ed effettuare una sola volta la richiesta di interruzione.

Kernel I/O Subsystem

Il sottosistema che gestisce l'I/O è caratterizzato sia dall'arrivo di richieste di I/O, ma anche dell'ottimizzazione di esse, tramite politiche di *scheduling*:

- coda *per-device*
- *fairness*
- *Quality Of Service*

Per migliorare le prestazioni si può utilizzare un buffer: una zona di memoria per disaccoppiare le tempistiche di chi fornisce un servizio e chi lo richiede. Questa tecnica richiede due operazioni di lettura: una sul buffer e l'altra dal buffer.
Le problematiche di bufferizzazione sono da gestire a livello software e spesso includono attese e locking.

Dal punto di vista software, il sottosistema I/O del kernel deve gestire delle tabelle dello stato di dispositivi. Esse sono analoghe alle tabelle utilizzate per i files.