

# SISTEMI OPERATIVI

## GESTIONE DEI PROCESSI

### DEFINIZIONI:

DEADLOCK: Ogni processo, in un insieme di processi, si trova in attesa di un evento che può essere provocato solo da uno degli altri processi dell'insieme.

Il deadlock **provoca l'attesa infinita** di tutti i processi.

STARVATION: **L'impossibilità** perpetua, da parte di un processo pronto all'esecuzione, **di ottenere le risorse** sia hardware sia software di cui necessita per essere eseguito.

### SCHEDULING DELLA CPU:

TURNAROUND TIME: **Tempo medio di completamento** di un processo da quando entra in coda di ready per la prima volta a quando termina

WAITING TIME: La **somma del tempo** passato da un processo P in coda di ready.

### RELAZIONE TRA TURNAROUND TIME E WAITING TIME DI UN PROCESSO:

TURNAROUND TIME = TEMPO DI ESECUZIONE + WAITING TIME

WAITING TIME = TURNAROUND TIME – TEMPO DI ESECUZIONE

### ALGORITMI DI SCHEDULING DELLA CPU:

FCFS (First Come First Served)

- gestito come coda FIFO
- Non preemptive
- Non va bene per time-sharing perchè non garantisce tempi di risposta brevi
- Non va bene per sistemi Real-Time perchè non è preemptive
- **Non soffre di starvation**

SJF (Shortest Job First)

- 2 versioni: preemptive oppure non preemptive. Nella versione preemptive se arriva in coda di ready un processo che ha un burst di CPU **minore della rimanenza** del processo in esecuzione **il processo running viene interrotto** e la CPU passa al nuovo processo.
- **Soffre di starvation** sia nella versione preemptive che in quella non-preemptive.

## Round Robin (RR)

- Ogni processo dispone di un **quanto di tempo**, se entro il quanto di tempo il processo non lascia la CPU viene rimesso in coda di ready e la CPU viene assegnata al prossimo processo in coda. **La coda è gestita secondo il criterio FCFS.**
- **Implementa il TIME-SHARING**, particolarmente adatto per sistemi interattivi.
- Se ci sono **n processi in coda di ready** e il quanto di tempo è  $q$ , allora ogni processo riceve  $1/n$  del tempo della CPU e nessun processo aspetta per più di  $(1/n)q$  unità di tempo.
- Le prestazioni di RR **dipendono dal valore di  $q$** :
  - $q$  che tende a infinito rende RR uguale a FCFS
  - $q$  tendente a 0 aumenta il parallelismo (virtuale) ma aumenta il numero di context switch e di conseguenza l'overhead

## SCHEDULING CON PRIORITA'

- la CPU è assegnata al processo con la **priorità maggiore** tra quelli presenti in coda di ready
- Calcolo della priorità:
  - interna al sistema: calcolata dal SO
  - esterna al sistema: assegnata con criteri esterni al SO
- 2 versioni: preemptive oppure non preemptive.
- Usano un **meccanismo di aging per prevenire lo starvation** (la priorità di un processo aumenta man mano che rimane in coda di ready)

**NOTA: FCFS e SJF sono algoritmi a priorità** dove nel primo caso la priorità è determinata dal tempo di arrivo in coda di ready, mentre nel secondo caso dalla durata del burst di CPU dei processi

## CODE MULTIPLE CON RETROAZIONE

- **Più code di ready gestite da algoritmi diversi**
- Processi divisi per classi:
  - FOREGROUND: Processi interattivi
  - BACKGROUND: Processi che non interagiscono con l'utente
  - BATCH: Processi la cui esecuzione può essere differita
- **Il processo può essere spostato dal sistema operativo da una coda all'altra per:**
  - Adattarsi alla lunghezza del burst di CPU del processo.
  - Gestire ogni coda con lo scheduling più adatto
  - Viene usato RR per i processi con priorità più alta e FCFS per i processi che hanno bassa priorità

## **/\* METODI PER LA GESTIONE DEI DEADLOCK:**

1. **PREVENIRE O EVITARE** i deadlock, usando un protocollo di richiesta e assegnamento delle risorse
2. lasciare che il deadlock si verifichi, ma **FORNIRE STRUMENTI PER LA SCOPERTA E IL RECUPERO**, utilizzando il grafo di assegnazione.
3. **LASCIARE AGLI UTENTI LA PREVENZIONE/GESTIONE (soluzione adottata nei moderni sistemi operativi) \*/**

## **PROBLEMA DELLA SEZIONE CRITICA:**

1. MUTUA ESCLUSIONE: Se un processo è entrato nella propria sezione critica(e non ne è ancora uscito), nessun altro processo può accedere alla propria sezione critica.
2. PROGRESSO: Se un processo lascia la propria sezione critica deve permettere ad un altro processo di entrare nella propria sezione critica.  
Se la sezione critica è vuota e più processi vogliono entrare uno di questi dev'essere scelto in un tempo finito.
3. ATTESA LIMITATA: Se un processo ha richiesto l'accesso alla propria sezione critica , esiste un limite al numero di volte in cui viene consentito ad altri processi di passargli davanti. (prima o poi il processo deve accedere alla propria sezione critica).

**NOTA: L'attesa limitata garantisce che non ci sia starvation.**

## SOLUZIONI AL PROBLEMA DELLA SEZIONE CRITICA:

### 1) SINCRONIZZAZIONE HARDWARE

Utilizzo di istruzioni macchina privilegiate e ATOMICHE:

- TestAndSet (var1)            *// testa e modifica il valore di una cella di memoria*
- Swap (var1, var2)           *// scambia il valore di due celle di memoria*

/\* Il codice della TestAndSet (visto a lezione) **non garantisce l'attesa limitata**, perchè uno stesso processo potrebbe ipoteticamente uscire e rientrare nella propria sezione critica nello stesso quanto di tempo.

NOTA: il meccanismo di aging non funziona perchè il problema non riguarda lo scheduling ma il codice.

INOLTRE utilizza il meccanismo di BUSY-WAITING e quindi spreca CPU inutilmente. \*/

**NOTA:** La **disabilitazione degli interrupt da parte del processo è vietata**, perchè equivale a **togliere il controllo della macchina** al sistema operativo; ad esempio non viene più ricevuto l'interrupt che segnala lo scadere del timer hardware.

Le istruzioni che disabilitano gli interrupt sono **istruzioni macchina privilegiate che possono essere usate solo dal sistema operativo**.

### 2) SEMAFORI

Un semaforo è una struttura dati usata per **implementare meccanismi di sincronizzazione**.

La struct del semaforo ha:

- un **campo value**, che conta il valore del semaforo
- un **puntatore a struct** che punta alla **lista di PCB in attesa** sul semaforo.  
Struct PCB \*waiting\_list;

## WAIT E SIGNAL:

Le operazioni di *wait* e *signal* possono essere implementate attraverso le syscall *sleep()* e *wakeup(P)*.

- *sleep()*: **toglie la CPU al processo** che la invoca e **manda in esecuzione** uno dei processi in ready queue.  
**NOTA:** Il processo sospeso viene messo nella **lista di attesa del semaforo** e non in coda di ready.
- *wakeup(P)*: rimette il processo P in coda di ready.

```
"wait (sem *S)" { // codice di wait: prende come parametro un puntatore a struct sem  
    S->value -- ;  
  
    if S->value < 0 { // se il valore è < 0 ...  
  
        sleep(); } // ... il processo viene messo in waiting_list  
  
}
```

**NOTA:** La chiamata di *sleep()* provoca un context switch e il processo che si addormenta sul semaforo non consuma CPU inutilmente.

```
"signal (sem *S)" { // codice di signal: prende come parametro un puntatore a struct sem  
    S->value ++ ;  
  
    if S->value <= 0 { // se dopo l'incremento il valore è <= 0 c'è qualcuno in wait  
  
        wakeup(P); } // toglie P da waiting_list e lo rimette in coda di ready  
  
}
```

**PROBLEMA:** wait e signal sono **esse stesse sezioni critiche** perchè accedono a una struttura dati condivisa (il semaforo).

**SOLUZIONE:** dato che wait e signal sono porzioni di codice brevi è possibile usare uno **spinlock(busy-waiting)** oppure la **disabilitazione degli interrupt** (che avviene sotto il controllo del SO dato che sono codice del SO stesso).

**NOTA:** In ambiente Unix wait e signal sono implementate da semop.

## ALGORITMI:

### LETTORI-SCRITTORI

- 2 SEMAFORI + COUNTER:

- sem mutex = 1;
- sem scrivi = 1;
- int numLettori = 0;

“Scrittore” {            // codice scrittore

wait(scrivi);            // accesso in scrittura in mutua esclusione

// ESEGUI LA SCRITTURA DEL FILE

signal(scrivi)

}

“Lettore” {            // codice lettore

wait(mutex);

numLettori ++;        // conta i lettori

if numLettori == 1    wait(scrivi);        // il primo lettore blocca la scrittura del file

signal(mutex);        // una volta fatto l'accesso in lettura tutti i lettori possono accedere

// ...LEGGI IL FILE...

wait(mutex)

numLettori --;        // decrementa i lettori

if numLettori == 0    signal(scrivi);        // l'ultimo lettore rilascia il semaforo degli scrittori

signal(mutex);        // ... e quello dei lettori

}

**NOTA:** L'algoritmo “Reader-First” soffre di Starvation per gli scrittori, nel caso in cui arrivino sempre nuovi lettori che non rilascino mai il semaforo (“scrivi”) degli scrittori.

## PRODUTTORI-CONSUMATORI

- BUFFER CIRCOLARE (array di struct) CONDIVISO DI DIMENSIONE "SIZE":
  - `typedef struct { ... } item;`
  - `item buffer [SIZE];`
- DATI, INDICI E SEMAFORI:
  - `sem full = 0, empty = SIZE, mutex = 1;` // semafori
  - `item nextp, nextc;` // dati
  - `int in = 0, out = 0;` // indici
- full: conta il numero di posizioni piene nel buffer (all'inizio 0);
- empty: conta il numero di posizioni vuote nel buffer (all'inizio SIZE);
- mutex (semaforo binario): regola l'accesso in mutua esclusione al buffer;

```
"Produttore" {           // codice produttore

    while(true){

        // PRODUCI UN ITEM NEXTP

        wait(empty);       // attende che ci siano posizioni vuote
        wait(mutex);       // acquisisce il lock (accesso in mutua esclusione)

        buffer[in] = nextp; // inserisce nextp nel buffer
        in = (in + 1) mod SIZE; // aggiorna in (il modulo garantisce la circolarità)

        signal(mutex);
        signal(full);       // NOTA: il produttore fa la signal su "full"

    }
}

"Consumatore" { // codice consumatore

    while(true){

        wait(full);         // attende che ci siano posizioni piene
        wait(mutex);       // acquisisce il lock (accesso in mutua esclusione)

        nextc = buffer[out] // rimuove un elemento dal buffer
        out = (out + 1) mod SIZE; // aggiorna out (il modulo garantisce la circolarità)

        signal(mutex);
        signal(empty);      // NOTA: il consumatore fa la signal su "empty"

        // CONSUMA UN ITEM NEXTC

    }
}
```

## 5 FILOSOFI

Problema: 5 Filosofi ad un tavolo rotondo con 5 bacchette condivise, una a destra e una a sinistra. Ogni filosofo deve usare due bacchette per mangiare.

Problema: se ogni filosofo prende una bacchetta nessun filosofo può mangiare: starvation.

- DATI CONDIVISI (BACCHETTE) RAPPRESENTATI COME SEMAFORI:
  - sem chopstick [5];           // tutti i semafori inizializzati a 1

```
"Filosofo i" {           // codice 5 filosofi

while(true) {

    wait(chopstick [ i ]);           // bacchetta 1
    wait(chopstick [ i + 1 mod 5]);   // bacchetta 2

    // MANGIA

    signal(chopstick [ i ]);
    signal(chopstick [ i + 1 mod 5]);

    // PENSA

}
}
```

**NOTA:** L'algoritmo soffre di deadlock.

POSSIBILI SOLUZIONI:

1. SOLO 4 FILOSOFI A TAVOLA CONTEMPORANEAMENTE;
2. UN FILOSOFO PUO' PRENDERE LE SUE BACCHETTE SOLO SE SONO ENTRAMBE DISPONIBILI: RICHIEDE IMPLEMENTAZIONE CON SEZIONE CRITICA: sem *mutex*;
3. SOLUZIONE ASIMMETRICA: FILOSOFO DISPARI PRENDE PRIMA LA BACCHETTA DI SINISTRA, MENTRE UN FILOSOFO PARI PRENDE PRIMA LA BACCHETTA DI DESTRA (un filosofo rimane bloccato, simile a soluzione 1.).

## GESTIONE DELLA MEMORIA

### BINDING (associazione degli indirizzi):

PUO' AVVENIRE IN 3 FASI:

1. **COMPILAZIONE:** genera **codice STATICO** (o assoluto): Il programma **deve essere ricompilato** ogni volta che viene caricato in RAM.
2. **CARICAMENTO:** genera **codice STATICAMENTE RILOCABILE**.  
Il **compilatore associa degli indirizzi relativi** rispetto all'inizio del programma (indirizzo virtuale 0). **Gli indirizzi assoluti vengono associati in fase di caricamento** dal Loader: **non è necessario ricompilare**.
3. **A RUN TIME:** genera **codice DINAMICAMENTE RILOCABILE**.  
Il codice in esecuzione usa **sempre e solo indirizzi relativi(logici)**: La trasformazione di un indirizzo relativo in assoluto avviene a run time.

### HARDWARE A SUPPORTO DEL BINDING DINAMICO:

REGISTRO DI RILOCAZIONE: Contiene l'indirizzo di partenza(base address) **dell'area di RAM** in cui è stato caricato il programma in esecuzione.

MEMORY MENAGEMENT UNIT (MMU): Somma l'indirizzo relativo(logico) al base address contenuto nel registro di rilocazione ottenendo l'indirizzo fisico.

**NOTA:** L'indirizzo logico è l'**offset dall'indirizzo base**.

### CALCOLO DELLO SPAZIO DI INDIRIZZAMENTO:

SPAZIO DI INDIRIZZAMENTO LOGICO: Insieme degli indirizzi logici: **da 0 a max**

SPAZIO DI INDIRIZZAMENTO FISICO: Insieme degli indirizzi fisici: **da  $r + 0$  a  $r + \text{max}$** ,  $r$  = base-address.

Gli indirizzi fisici vengono caricati nel MAR(Memory Address Register) per indirizzare una cella della memoria centrale.

**NOTA:** Lo spazio di indirizzamento fisico e logico è sempre una potenza di 2 in quanto gli indirizzi sono scritti in bit, quindi avremo che una certa architettura usa  $n$  bit per scrivere un indirizzo, questo significa che lo spazio di indirizzamento di quella macchina è pari a  **$2^n$  byte (la memoria è indirizzata al byte)**.

### RICORDARE LE POTENZE DI 2

$2^{10}$	=	1 Kib
$2^{20}$	=	1 Mib
$2^{30}$	=	1 Gib
$2^{40}$	=	1 Tib

Quindi:  $2^{35} = 2^{30} * 2^5 = 32 * 1\text{Gib} = 32\text{Gib}$

## LIBRERIE:

### 2 TIPI DI LIBRERIE:

1. LIBRERIE STATICHE: collegate al codice sorgente dal compilatore o dal loader, **diventano parte dell'eseguibile**.  
NOTA: il codice della libreria viene caricato in RAM anche se non viene utilizzato: **grande spreco di memoria**.
2. LIBRERIE DINAMICHE: **caricate a run time** quando il programma in esecuzione invoca una subroutine della libreria.  
**NOTA**: Le librerie dinamiche possono essere **aggiornate senza bisogno di ricompilare**, inoltre il **codice può essere condiviso tra processi**.

## DEFINIZIONI:

FRAMMENTAZIONE INTERNA: La porzione di spazio concessa a **un processo** non viene completamente utilizzata

FRAMMENTAZIONE ESTERNA: La somma delle differenze tra la dimensioni delle partizioni e quella dei processi, cioè lo **spreco globale** (somma delle frammentazioni interne)

## TECNICHE DI GESTIONE DELLA MEMORIA:

### SWAPPING

- Tecnica per permettere l'avvicendamento dei processi in memoria centrale.
- Viene **copiato il PCB di un processo non in esecuzione** (sta effettuando operazioni di I/O) in un area di **memoria secondaria** chiamata **area di swap** (swap-out) e viene ricaricato in memoria prima di assegnargli la CPU (swap-in).
- **TECNICA OBSOLETA a causa dell'overhead** prodotto: tempi di accesso alla memoria secondaria molto lenti.

### ALLOCAZIONE CONTIGUA DELLA MEMORIA PRIMARIA

- Il sistema operativo inizializza un **REGISTRO LIMITE, corrispondente alla dimensione della partizione**. Ogni indirizzo logico usato dal processo utente deve essere inferiore al valore scritto nel registro limite.  
**NOTA**: La protezione della memoria è garantita dal registro di rilocalizzazione, e dal **registro limite** che pongono i limiti, rispettivamente inferiore e superiore oltre i quali non possono essere generati indirizzi logici.
- La porzione di memoria dedicata ai processi utente è suddivisa in **partizioni di dimensione fissa**
- Ogni partizione **contiene un unico processo** dall'inizio alla fine della sua esecuzione, quando un processo termina lascia spazio ad un altro processo.

- **Limita il grado di multiprogrammazione** perché questo **dipende dal numero di partizioni**.
- **Soffre di frammentazione interna ed esterna**

#### ALLOCAZIONE A PARTIZIONI MULTIPLE VARIABILI

- Un processo riceve una **quantità di memoria pari** alla sua dimensione.
- Tecniche di assegnazione della memoria:
  - First Fit: si sceglie la **prima partizione disponibile** ad ospitare il processo
  - Best Fit: si sceglie la **più piccola partizione sufficientemente grande**
  - Worst Fit: si sceglie la partizione più grande
- L'avvicendamento di processi con dimensioni differenti crea buchi in RAM non contigui e sempre più piccoli: **frammentazione esterna e interna** (dovuta all'eccessivo costo computazionale per tenere traccia dei buchi molto piccoli)
- E' possibile risolvere il problema della frammentazione usando la **compattazione della memoria**, tecnica che prevede la **rilocazione dei processi** in modo da rendere contigue tutte le aree libere di RAM creando un unico spazio libero: **richiede codice dinamicamente rilocabile**
- La compactazione è **molto pesante** e durante questa operazione nessun processo utente può essere eseguito.

#### PAGINAZIONE DELLA MEMORIA

**IDEA:** I processi sono scomposti in segmenti della **stessa dimensione** che possono essere allocati in RAM in modo **non contiguo**

- La memoria primaria (spazio di indirizzamento fisico) è **divisa in frame** tutti **della stessa dimensione**
- Lo spazio di indirizzamento logico viene sempre visto dal processo come uno spazio contiguo, ma è **diviso in pagine**

PAGINA: riferita allo spazio di **indirizzamento logico**

FRAME: riferito allo spazio di **indirizzamento fisico**

**NOTA:** Frame e pagine hanno sempre la **stessa dimensione**

## PAGINAZIONE METODO BASE:

- Il sistema operativo cerca x frame liberi in cui caricare le pagine che contengono le istruzioni da eseguire
- Ad **ogni processo** è associata una **page table (PT)**, cioè un **array** le cui entry contengono i **numeri dei frame** associati alle pagine che contengono le istruzioni e i dati del processo.

**NOTA:** Gli indici dell'array **corrispondono alle pagine** del processo.

- Utilizzo di **indirizzi come coppie di valori**, per poter specificare l'offset all'interno del frame:
  - il **primo elemento specifica il numero del frame**, cioè l'indirizzo base in memoria fisica (RAM) a partire dal quale è contenuta la pagina.
  - Il **secondo elemento specifica l'offset** all'interno della pagina.

## TRADUZIONE DEGLI INDIRIZZI (da logici a fisici)

- Il numero della pagina viene usato come **indice nella page table** per sapere in quale frame è contenuta la pagina.
- L'offset viene applicato a partire **dall'indirizzo base del frame**

## OSSERVAZIONI SULLA PAGINAZIONE:

- La paginazione **implementa automaticamente una forma di protezione** della memoria: La corrispondenza di frame e pagine garantisce l'univocità di indirizzamento.
- **Evita la frammentazione esterna**
- Soffre di **frammentazione interna (mediamente ½ pagina a processo)**
- La paginazione **è una forma di rilocalizzazione dinamica**: ad ogni pagina corrisponde un valore del registro di rilocalizzazione.
- Non è necessario avere un registro limite perché l'offset specificato nell'indirizzo logico raggiunge al massimo la dimensione del frame.
- Permette la **condivisione di codice**, ad esempio pagine condivise possono essere usate per contenere una libreria dinamica

## MEMORIA ASSOCIATIVA: TLB (Translation Look-aside Buffer):

**PROBLEMA:** Nei moderni computer il numero di pagine per processo è molto elevato (nell'ordine delle migliaia) di conseguenza non è possibile gestire la page table a livello hardware.

**NOTA:** Non è possibile mantenere la page table in RAM in quanto per ogni indirizzo sarebbe necessario effettuare due accessi in RAM: il primo per accedere alla page table e il secondo per accedere all'istruzione/dato.

**Il costo di accesso in RAM è nell'ordine di 100 cicli di clock** (contro 3/5 della cache di primo livello) quindi ovviamente questo è inaccettabile.

**SOLUZIONE:** Utilizzo di **hardware dedicato**:

- **memoria associativa: TLB** (memoria a semiconduttore)
- **page table base register (PTBR):** Registro che contiene l'indirizzo base della page table associata al processo in esecuzione (ad uso esclusivo del sistema operativo)

## FUNZIONAMENTO TLB: COPPIE DI CELLE CHIAVE-VALORE:

- Ogni entry si riferisce ad una **coppia di celle**, in cui la prima contiene la chiave e la seconda il valore associato alla chiave.
- Specificando una chiave in input al TLB questa viene **confrontata in parallelo** con tutte le chiavi presenti
- Fornisce in output il **valore associato alla chiave** specificata in input

## TLB E PAGE TABLE:

- Nel TLB viene caricata la page table (tutta se ci sta o una parte)
- Il campo **chiave** contiene il **numero di pagina**
- Il campo **valore** contiene il **numero di frame**

## CALCOLO HIT E MISS

$n$  = costo dell'accesso in RAM (in nsec)

$m$  = costo dell'accesso al TLB (in nsec)

HIT RATIO =  $k\%$  =  $0,k$  con hit ratio dato.

Degrado delle prestazioni rispetto a un sistema che non utilizza la paginazione:

$$(n + m) * 0,k + (n * 2) * (1 - 0,k)$$

**NOTA:** Il TLB utilizza un **meccanismo di caching**: se si verifica un miss la coppia chiave-valore viene copiata in una entry. Se il TLB è pieno viene utilizzato il criterio *Last Recently Used*. Ad ogni context switch il TLB viene svuotato.

## CALCOLO DELLO SPAZIO DI INDIRIZZAMENTO LOGICO:

- Un indirizzo logico è una sequenza di bit.
- Se dividiamo questa sequenza per rappresentare la pagina (prima parte) e l'offset (seconda parte) dobbiamo decidere **quanti bit dedicare** ad entrambe. (La scelta dipende dall'hardware su cui deve girare il sistema operativo)
- Se la dimensione di un frame (e quindi di una pagina) è  $2^n$  allora significa che usiamo  $n$  bit per l'offset.
- Se usiamo  $n$  bit per l'offset allora abbiamo a disposizione  $m - n$  bit per rappresentare le pagine, con  $m$  = numero di bit usati per l'indirizzo.
- Quindi la dimensione dello spazio di indirizzamento logico è:  $2^{(m-n)} * 2^n$  byte (l'indirizzamento in memoria è sempre al byte).

Esempio:

dimensione frame =  $2^{12}$  byte = 4096 byte,  $n = 12$ ;  
dimensione indirizzo = 22 bit,  $m = 22$ ;

spazio di indirizzamento logico =  $2^{(22-12)} * 2^{12} = 2^{10} * 2^{12} = 2^{22} = 2^{20} * 2^2 = 4\text{Mib}$

**NOTA:** Lo spazio di indirizzamento logico rappresenta il **numero di frame indirizzabili**

## CALCOLO DELLO SPAZIO DI INDIRIZZAMENTO FISICO:

Partendo dallo spazio di indirizzamento logico di dimensione  $2^k$ , possiamo ricavare la **dimensione massima** dello spazio di indirizzamento fisico semplicemente **moltiplicando la dimensione della pagina/frame alla dimensione dello spazio di indirizzamento logico**:

spazio di indirizzamento fisico =  $2^k * 2^n = 2^{(k+n)}$

**NOTA:** Lo spazio di indirizzamento logico rappresenta il **numero di frame indirizzabili**, quindi per ottenere la dimensione massima dello spazio fisico basta moltiplicare la dimensione del frame per il numero di frame indirizzabili.

## ALTRI ESEMPI:

Calcolo dimensione PT da spazio di indirizzamento fisico:

spazio di indirizzamento fisico =  $2^n$  =  $n$  bit per scrivere l'indirizzo fisico  
dimensione del frame =  $2^m$   
numero di frame =  $2^{(m-n)} = 2^k$

$2^k$  = dimensione minima delle entry della PT

**NOTA:** In sistemi reali la dimensione della entry della PT è sempre un multiplo di byte ( $2^3$ )

## PAGINAZIONE A PIU' LIVELLI (PAGINAZIONE GERARCHICA):

La PT di un processo (ogni processo è associato ad una PT), nei sistemi moderni, può essere molto grande e occupare molto spazio in memoria, nello specifico deve essere **allocata in più frame**.

**PROBLEMA:** La paginazione è stata introdotta per evitare di avere aree contigue molto grandi in memoria, per poter tenere una PT in un unico frame avremmo bisogno di pagine estremamente grandi (nell'ordine dei TB!!!).

**SOLUZIONE:** Paginare la PT:

- **TABELLA DELLE PAGINE INTERNA:** LA PT VERA E PROPRIA
- **TABELLA DELLE PAGINE ESTERNA:** LA PT DELLA PT (ci dice in quali frame sono memorizzate le pagine della PT in memoria)

## CALCOLO DELLA DIMENSIONE DI UNA PT A DUE LIVELLI:

Esempio 1:

- Macchina con 32 bit di spazio logico e fisico e frame/pagina da 4 Kbyte ( $2^{12}$ )
- Indirizzo composto da: 20 bit pagine + 12 bit offset
- Entry PT =  $2^{20}$
- NOTA: il numero delle entry e la dimensione dovrebbero essere uguali dal momento che per indirizzare  $2^{20}$  pagine servono 20 bit. In realtà nei sistemi reali le entry della page table sono sempre multipli di byte ( $2^4$ ), assumiamo che le entry siano della grandezza di un int, cioè 4 byte.
- Grandezza massima della PT =  $4 \text{ byte} * 2^{20} = 4\text{Mb} = 2^{22}$
- La grandezza del frame (data) è  $4\text{Kb} = 2^{12}$
- La PT occupa  $2^{22} / 2^{12} \text{ frame} = 2^{10} = 1024 \text{ frame}$  in memoria primaria.
- Per tenere traccia dei 1024 frame è della PT interna (in RAM) è necessario usare una **PT esterna con 1024 entry**
- Ogni entry della PT è grande 4 byte (vedi sopra) quindi la PT esterna occupa:  
 $1024 \text{ entry} = 2^{10} * 4 \text{ byte} = 4\text{Kb} = 2^{12}$ , cioè, nel caso in esempio, la grandezza di un frame
- Quindi l'indirizzo logico sarà composto da:
  - 10 bit più significativi indirizzano le  $2^{10} = 1024$  entry della PT interna (in RAM)
  - 10 bit intermedi indirizzano i  $2^{10} = 1024$  **frame** (il base address del frame)
  - Sommando l'offset si ottiene l'indirizzo corretto all'interno del frame

### Esempio 2:

- Spazio logico su 64 bit, frame/pagina da  $4\text{Kb} = 2^{12}$  bit e 4 byte per ogni entry della PT
- Indirizzo composto da  $64 - 12 = 52$  bit per le pagine + 12 bit per l'offset
- **dimensione PT interna:**  $2^{52} = \text{numero entry della PT} * 4 \text{ byte} = 2^{52} * 2^2 = 2^{54} \text{ byte}$
- PT interna è contenuta in  $2^{54} / 2^{12} = 2^{42}$  frame
- **PT esterna** ha un **numero di entry pari al numero di frame** necessari a contenere la PT interna =  $2^{42}$  entry
- **dimensione della PT esterna:**  $2^{42} * 2^2 \text{ byte} = 2^{44} \text{ byte}$

**NOTA:**  $2^{44} \text{ byte} = 16 \text{ Tb}$  non esiste un sistema dotato di una quantità simile di RAM perciò è necessario paginare anche la PT esterna seguendo lo stesso principio, per questo si parla di paginazione a più livelli.

### **TABELLA DELLE PAGINE INVERTITA (IPT):**

- Usata nelle architetture a 64 bit
- Descrive **l'occupazione dei frame della memoria fisica da parte di tutti i processi**
- C'è **una sola IPT** per tutto il sistema
- La dimensione della IPT dipende strettamente dalla dimensione della RAM, in quanto:
- **L'indice di ogni entry** della IPT corrisponde al **numero di un frame in RAM**
- Ogni entry della IPT contiene una coppia:
  - **<process-ID, page number>** :
  - **process-ID:** ID del processo che possiede la pagina
  - **page-number:** numero di pagina associata al **frame corrispondente alla entry** (l'indice dell'array: la PT è un array)
- Indirizzo logico è **una tripla:** **<process-ID, page number, offset>**
- Per generare l'indirizzo fisico si cerca nella IPT la coppia: **<process-ID, page number>**. Se la si trova **all'i-esimo elemento**, si genera l'indirizzo fisico: **<i, offset>**

Esempio: All'indice  $pt[0]$  della page table, corrispondente al base address del frame 0, avremo una coppia <process-ID, page number> che ci dice quale pagina è associata a quel frame e a quale processo appartiene la pagina.

Quindi si cerca la coppia <process-ID, numero pagina> una volta trovata la coppia ad un indice  $i$  la MMU **genera l'indirizzo fisico associando l'indice  $i$  e l'offset**

### MEMORIA VIRTUALE:

- La memoria virtuale è un insieme di tecniche per permettere l'esecuzione di processi in cui **codice e/o dati non sono stati caricati completamente** in RAM
- Viene **caricato** in memoria primaria **una parte di programma solo se** deve essere eseguito e **solo quando** deve essere eseguito
- Si cerca in memoria **solo la porzione di strutture dati o codice** che vengono usate in una certa fase di esecuzione del programma

### VANTAGGI:

- **E' possibile eseguire processi che sfruttano uno spazio di indirizzamento logico maggiore di quello fisico**
- **E' possibile avere contemporaneamente in esecuzione processi** che, insieme, **occupano più spazio dello spazio di indirizzamento fisico** (quindi della memoria primaria)
- **Aumenta il grado di multiprogrammazione e quindi il throughput della CPU**
- **I programmi partono più velocemente perché non è necessario che vengano caricati completamente in memoria primaria**

### SVANTAGGI:

- Aumento di **traffico tra la RAM e l'Hard Disk**
- L'esecuzione del singolo programma può richiedere più tempo che se la memoria virtuale non fosse implementata
- In situazioni particolari può verificarsi il **thrashing: degradazione drastica delle prestazioni complessive del sistema**

## PAGINAZIONE SU RICHIESTA (DEMAND PAGING):

### IDEA:

La pagina viene **caricata in RAM solo al momento del primo indirizzamento** di una locazione appartenente alla pagina.

### PAGE FAULT:

L'istruzione eseguita dalla CPU indirizza una **pagina** che **non viene trovata in memoria**, genera una trap gestita dal gestore di interrupt

### CONSEGUENZA DEL PAGE FAULT:

- Il processo viene tolto dalla CPU e messo in uno stato di **waiting for page**
- Il **pager** (modulo del sistema operativo) **inizia il caricamento** della pagina mancante
- in attesa di completare il caricamento la CPU viene assegnata a un altro processo
- Quando la pagina è in memoria primaria il processo viene rimesso in coda di ready

### BIT DI VALIDITA':

- Per sapere se una pagina è stata caricata in RAM il sistema operativo usa un **bit di validità associato ad ogni entry della page table**
- 0 se la pagina non è presente, 1 se è presente
- Se si tenta di fare riferimento a una pagina con bit di validità a 0 viene generato page fault

**NOTA:** La paginazione su richiesta e, in generale, **la memoria virtuale ha bisogno di hardware specifico** per essere implementata, in particolare un **bit di validità** testabile dall'hardware in grado di generare page fault

### PRESTAZIONI DELLA PAGINAZIONE SU RICHIESTA:

- **ma (medium access time)** = tempo di accesso in RAM se il dato è presente
- **p** = probabilità di un page fault
- **eat (effective access time)** =  $(1 - p) * ma + [p * \text{tempo di gestione del page fault}]$

### Esempio 1:

- $m_a = 200 \text{ nsec}$
- $p = 0.001$  (1 page fault ogni 1000 accessi)
- $e_{at} = (1 - 0,001) * 200 + 0,001 * 8.000.000 \text{ (8 millisec)} = 0,999 * 200 + 8000 = 8.199,8$
- (circa 8 microsecondi)
- L'esecuzione rallenta di più di 40 volte

### Esempio 2:

- Se vogliamo avere un degrado massimo del 10% allora  $p$  dev'essere tale che
- $e_{at} = 220 \geq 200 + 8 * 10^6 * p$
- $20 \geq 8 * 10^6 * p$
- $p \leq 2,5 * 10^{-6}$ ,

cioè non più di 1 page fault ogni 400.000 accessi.

### **AREA DI SWAP:**

E' la porzione del dispositivo di memoria secondaria **dedicata al sistema operativo**. Viene riservata in fase di installazione (del dispositivo).

**IDEA:** Si utilizza un'area del dispositivo di memoria secondaria come estensione della memoria principale

### FUNZIONE DELL'AREA DI SWAP

- All'avvio gli eseguibili vengono copiati nell'area di swap, in modo da diminuire il tempo di gestione del page fault.
- L'area di swap è usata per **liberare spazio in memoria primaria** in modo da poter ospitare pagine mancanti che devono essere caricate in RAM a seguito di page fault.
- Se si verifica un page fault e **tutti i frame in RAM sono occupati** occorre liberarne uno.
- **VICTIM PAGE:** Pagina rimossa.
- Se la victim page **contiene dati modificati** e/o fa **parte dello stack o della heap** di un processo in esecuzione, deve essere **salvata nell'area di swap**

**NOTA:** Se la victim page è una pagina di codice semplice non deve essere salvata

**DIRTY BIT:** E' un bit associato ad ogni entry della page table (come anche il bit di validità) che viene impostato a 1 se la pagina viene modificata e quindi va salvata in area di swap.

**NOTA:** Conviene scegliere pagine con Dirty Bit a 0 per risparmiare tempo.

## ALGORITMI DI SOSTITUZIONE DELLE PAGINE:

Un algoritmo di sostituzione delle pagine deve **minimizzare** il numero di page fault. Per fare questo deve essere in grado di **prevedere** quale pagina non genererà in futuro ulteriori page fault e sceglierla come victim page.

## TESTING DELL'ALGORITMO DI SOSTITUZIONE DELLE PAGINE:

Si usano **sequenze di riferimenti in memoria principale**.

### FIFO: SOSTITUZIONE SECONDO L'ORDINE DI ARRIVO

- Viene **rimossa** la pagina che si trova da più tempo in memoria
- **facile** implementazione, ma **cattive prestazioni**, perché non tiene conto del contenuto della pagina per valutarne la rimozione:
  - Se, ad esempio, la pagina contiene una variabile o il codice di una procedura che viene utilizzata spesso, non conviene rimuoverla, preferendo una pagina che invece contiene solo codice che viene usato poco, o solo all'inizio, magari per l'inizializzazione.
- Soffre dell'**anomalia di Belady**: Usando un **numero maggiore di frame**, con particolari stringhe di riferimento, il **numero di page fault aumenta** anziché diminuire.

Es.

reference string: 7 0 1 | 2 0 3 | 0 4 2 | 3 0 3 | 2 1 2 | 0 1 7 | 0 1

7	7	7	2	2	2	4	4	4	0	0	0	7	7	7
	0	0	0	3	3	3	2	2	2	1	1	1	0	0
		1	1	1	0	0	0	3	3	3	2	2	2	1

L'algoritmo produce 15 page fault.

### ALGORITMO DI SOSTITUZIONE OTTIMALE (OPTIMAL)

- Sceglie come vittima, la **pagina che sarà usata più tardi**
- Non è implementabile, è usato solo come termine di paragone per misurare le prestazioni di altri algoritmi.
- Non soffre dell'anomalia di Belady

Es.

reference string: 7 0 1 | 2 0 3 | 0 4 2 | 3 0 3 | 2 1 2 | 0 1 7 | 0 1

7	7	7	2	2	2	2	2	7
	0	0	0	0	4	0	0	0
		1	1	3	3	3	1	1

L'algoritmo produce 9 page fault.

### ALGORITMO LRU (Last Recently Used)

- Cerca di approssimare OPT rimuovendo la pagina che **non viene usata da più tempo**.
- Invece di guardare in avanti **guarda indietro**
- Non soffre dell'anomalia di Belady
- **Non implementabile nelle CPU moderne** perché richiede troppo supporto hardware, ad esempio, una soluzione sarebbe associare un TIMER ad ogni entry della page table, ma è difficilmente realizzabile.

Es.

reference string: 7 0 1 | 2 0 3 | 0 4 2 | 3 0 3 | 2 1 2 | 0 1 7 | 0 1

7	7	7	2	2	4	4	4	0	1	1	1
	0	0	0	0	0	0	3	3	3	0	0
		1	1	3	3	2	2	2	2	2	7

L'algoritmo produce 12 page fault

## APPROSSIMAZIONI DI LRU:

- Utilizzano un **Reference bit**: Un bit associato ad ogni entry della Page Table. Si tratta di un semplice supporto HW messo a disposizione di molti processori moderni.
- All'avvio di un processo i reference bit delle sue pagine sono inizializzati a 0.
- Quando viene **indirizzata** una pagina il suo reference bit viene **messo a 1**.

**NOTA: Non è possibile sapere l'ordine di utilizzo.** Le pagine usate più di recente sono quelle con reference bit a 1 (sono state usate più di recente rispetto a quelle che non sono state utilizzate per niente).

## ALGORITMO DELLA SECONDA CHANCE

- Utilizza **FIFO + LRU con reference bit**
- **FIFO**: In caso di page fault il sistema operativo esamina la pagina entrata in RAM per prima (si trova da più tempo in RAM)
- **LRU**: Il sistema operativo controlla il reference bit della pagina:
  - **Ref. bit a 0: Elimina la pagina**
  - **Ref. bit a 1: Seconda chance**: Il reference bit viene **messo a 0** e il sistema operativo passa a esaminare la pagina successiva della coda FIFO.
- Implementato come **coda circolare**
- **NOTA**: Nel caso peggiore, in cui tutte le pagine con Ref. Bit a 1, l'algoritmo **deve scorrere tutta la coda** fino a tornare alla prima pagina che verrà rimossa. In questo caso l'algoritmo si comporta come FIFO.
- Nel caso peggiore **soffre dell'anomalia di Belady** (perché si comporta come FIFO)

## ALGORITMO DELLA SECONDA CHANCE MIGLIORATO

- Utilizza **sia il reference bit che il dirty bit**
- Le pagine sono raggruppate in 4 classi (Reference Bit, Dirty bit):
  - (0, 0) **non usata** di recente, **non modificata** (scelta migliore per l'eliminazione)
  - (1, 0) **usata** di recente ma **non modificata** (meno buona per l'eliminazione)
  - (0, 1) **non usata** di recente ma **modificata** (meno buona per l'eliminazione)
  - (1, 1) **usata** di recente e **modificata** (scelta peggiore)

- L'algoritmo è lo stesso della seconda chance ma **c'è un ordinamento migliore** delle pagine.
- Nel caso peggiore **soffre dell'anomalia di Belady**

### TECNICHE AGGIUNTIVE:

- Il sistema operativo si **riserva un pool di frame liberi da quelli della RAM** associati ai processi (Sistema usato praticamente da tutti i sistemi operativi)
- Se si verifica un page fault e se la pagina deve essere salvata (dirty bit a 1), la pagina viene trasferita in uno dei frame del pool, anziché venire salvata nell'area di swap (accesso alla RAM molto più rapido rispetto alla mem. secondaria)
- Il processo che ha generato page fault può ripartire prima perché non deve attendere il salvataggio della pagina in memoria secondaria.
- Se la pagina vittima viene riferita subito dopo si trova ancora in RAM.

### ALLOCAZIONE DEI FRAME:

#### SCELTA DEL NUMERO DI FRAME PER PROCESSO

ALLOCAZIONE UNIFORME: **Stesso numero di frame** per tutti i processi. Se abbiamo  $n$  frame e  $p$  processi allora ogni processo ha a disposizione  $n/p$  frame.

ALLOCAZIONE PROPORZIONALE: Il numero di frame per processo varia **in base alla dimensione dei processi** per distribuire i frame.

ALLOCAZIONE IN BASE ALLA PRIORITA': Il numero di frame assegnati a ogni processo **dipende dalla priorità del processo.**

### ALLOCAZIONE LOCALE E GLOBALE

#### SCELTA DELLA PAGINA VITTIMA ALL'OCCORRENZA DI UN PAGE FAULT

ALLOCAZIONE LOCALE: La **vittima è scelta tra le pagine del processo** che ha generato page fault.

**NOTA:** Il numero di frame per processo rimane invariato

ALLOCAZIONE GLOBALE: La **vittima è scelta tra tutte le pagine** in RAM (di solito escluse quelle del sistema operativo).

**NOTA:** E' molto probabile che venga **eliminata una pagina di un processo diverso** rispetto a quello che ha generato page fault.

## PROBLEMI DELL'ALLOCAZIONE GLOBALE

- Il **turnaround** di ogni processo è **fortemente influenzato** dal comportamento degli altri processi e potrebbe variare moltissimo da un'esecuzione all'altra.
- Se, per esempio, preso un processo  $P_1$ , gli altri processi generano molti page fault e sottraggono pagine a  $P_1$ , anche  $P_1$  genera molti page fault.

## PROBLEMI DELL'ALLOCAZIONE LOCALE

- Se si danno troppe pagine a un processo (scelta del numero di frame per processo) **può peggiorare il throughput del sistema** perché gli altri processi generano più page fault.

## **OSSERVAZIONI:**

- L'**allocazione globale** fornisce un **throughput maggiore** e riesce a gestire la **multiprogrammazione** in modo più flessibile (è stato verificato empiricamente), quindi:
- L'allocazione globale è **preferita per i sistemi multitasking e time sharing**

## **THRASHING**

Il thrashing è la condizione in cui si innesca un **circolo vizioso, legato alla scarsa disponibilità di frame per processo**, in cui ad ogni page fault i processi si "rubano" i frame l'un l'altro.

**NOTA:** Il thrashing si verifica quando si tenta di **aumentare troppo il grado di multiprogrammazione**.

## **SOLUZIONE AL THRASHING: GESTIRE LA FREQUENZA DEI PAGE FAULT:**

- Si stabilisce una **soglia di frequenza accettabile** di page fault rispetto alle prestazioni che si vogliono ottenere dal sistema.
- **Al di sotto** della soglia: Si può **aumentare** il grado di multiprogrammazione
- **Al di sopra** della soglia: necessario **diminuire** il grado di multiprogrammazione

**NOTA:** L'**allocazione locale** limita il problema del thrashing, ma se vengono dati troppi pochi frame per processo, per **aumentare il grado di multiprogrammazione**, c'è comunque rischio di thrashing.

**SOLUZIONE DEFINITIVA:** Dotare il sistema di **RAM sufficientemente grande**

## OSSERVAZIONI SULLA DIMENSIONE DELLE PAGINE:

La tendenza generale è quella di utilizzare **pagine grandi**

- **meno page fault**
- **Page table più piccole**
- **migliori prestazioni d'uso dell'HD**

**NOTA:** L'unico svantaggio rispetto all'utilizzo di pagine piccole è quello della frammentazione interna, ma la diminuzione dei costi della RAM e il conseguente aumento delle dimensioni rendono questo "problema" insignificante.

**Oggi**, grazie alla diminuzione dei costi della RAM e grazie alle innovazioni tecnologiche **l'utilizzo della memoria virtuale è molto meno frequente**. E' addirittura possibile, nei sistemi moderni **disabilitare la memoria virtuale** o **ridurre tantissimo l'area di swap**

## STRUTTURA DEI PROGRAMMI

- Il modo di utilizzare i dati **influisce enormemente** sul numero di page fault
- **Strutture dati diverse possono generare un numero di page fault molto alto oppure molto basso**
- Le matrici (array bidimensionali) sono **allocati per riga**. Se si scrive un programma che accede agli elementi per colonna si aumenta il rischio di page fault.

## MEMORIA VIRTUALE IN WINDOWS 10:

### DEMAND PAGING WITH CLUSTERING

- Quando viene caricata una pagina in memoria **vengono caricate anche alcune pagine adiacenti**, che potrebbero essere utilizzate a breve
- alla creazione di un processo gli vengono **assegnati due numeri**:
  - **insieme di lavoro minimo: minimo numero di pagine** che il sistema operativo garantisce di allocare in RAM per quel processo (di solito 50)
  - **insieme di lavoro massimo**: l'opposto (di solito 345)
- Il sistema operativo mantiene una **lista di frame liberi**, con associato un numero minimo di frame che devono essere presenti nella lista.
- Se un processo genera page fault e **non ha raggiunto il suo insieme di lavoro massimo**, la pagina mancante viene **caricata in uno dei frame liberi**

- Se il processo **ha raggiunto l'insieme di lavoro massimo** viene scelta come vittima una delle sue pagine: **criterio di sostituzione locale**
- Se si raggiunge il **limite minimo** di frame liberi in RAM viene innescata una procedura per liberare spazio:
  - Vengono rimosse tutte le pagine di tutti i processi in RAM **superiori all'insieme di lavoro minimo**.
  - Nei sistemi Intel viene usato l'**algoritmo della seconda chance**

#### MEMORIA VIRTUALE IN SOLARIS:

- Utilizza una normale paginazione su richiesta
- Utilizza il parametro **lostfree**: associato all'elenco di frame liberi che **indica l'eventuale mancanza di frame liberi**.
- Ogni  $\frac{1}{4}$  di secondo il sistema operativo controlla se il numero di frame liberi è inferiore a lostfree. In tal caso viene attivato il processo **pageout**.
- Pageout funziona in **due fasi**, applicando una **variante dell'algoritmo della seconda chance**
- Se pageout non riesce a mantenere il numero minimo di frame liberi, è possibile che si stia verificando il fenomeno del thrashing
- In questo caso il sistema operativo può decidere di rimuovere tutte le pagine di un processo, scegliendo tra i processi che sono inattivi da più tempo.

#### DOMANDE ESAME:

1) Domande del tipo: "Date certe caratteristiche..."

- ... il sistema deve prevedere un meccanismo di prevenzione del thrashing?"
- ... la tabelle delle pagine deve avere il bit di validità?"
- ...questo sistema deve prevedere un algoritmo di sostituzione delle pagine?"

**SOLUZIONE:** Per rispondere a questo tipo di domande bisogna **calcolare lo spazio di indirizzamento** logico e fisico. **Se lo spazio di indirizzamento logico è maggiore di quello fisico** allora significa che è necessario implementare la memoria virtuale e quindi la risposta è affermativa, in caso contrario la risposta è negativa.

2) Domande del tipo: "Date certe caratteristiche..."

- ... il sistema deve usare una paginazione a più livelli?"
- ... è sufficiente una paginazione a 2 livelli?"

**SOLUZIONE:** Per rispondere dobbiamo calcolare la dimensione massima della tabella delle pagine di un processo. Se la **dimensione della PT è maggiore della dimensione del frame** è necessario usare una paginazione a più livelli, altrimenti no.

Nel caso di più livelli di paginazione applichiamo lo stesso ragionamento alla PT interna.

**ATTENZIONE:** Nel calcolare la dimensione della PT ricordare che la dimensione delle entry è **arrotondata al byte!!**

3) Domande del tipo: "Date certe caratteristiche..."

- ...quale dimensione minima dovrebbero avere le pagine per non dover usare una paginazione a più livelli?"

**NOTA:** La dimensione minima della pagina deve essere uguale alla dimensione della PT più grande.

**SOLUZIONE:** Per rispondere bisogna impostare un **sistema di disequazioni** di primo grado (La dimensione delle entry della PT deve essere nota):

$k$  = numero di bit usati per l'indirizzo logico (spazio logico)

$m$  = numero di bit usati per il numero di pagina

$n$  = numero di bit usati per l'offset

$2^x$  = dimensione delle entry della PT

La relazione  **$k = m + n$**  è sempre valida (\*)

Per poter essere contenuta in un frame **la dimensione massima della PT dev'essere  $\leq$  alla dimensione della pagina**, quindi:

$2^m$  (numero di entry) \*  $2^x$  (dimensione delle entry)  $\leq 2^n$  (dimensione della pagina), quindi:  **$m + x \leq n$**

Impostiamo il sistema:

$$k = m + n$$

$$m + x \leq n$$

Es. per  $x = 2$  (dimensione di 4 byte, cioè  $2^2$ ) e  $k = 30$  otteniamo:

$$30 = m + n$$

$$m + 2 \leq n$$

Si ricava che  $n \geq 16$ , cioè le pagine devono avere una **dimensione minima** di  $2^{16} = 64\text{KB}$

(\*)fissato lo spazio logico se aumentiamo la dimensione della pagina,  $n$  aumenta e  $m$  diminuisce e viceversa

# GESTIONE DELLA MEMORIA DI MASSA

## HARD DISK:

- Composto da **piatti**: una serie di dischi sovrapposti di diametro da 4,5 a 9 cm.
- Ogni piatto è suddiviso in una serie di **tracce** circolari concentriche
- Ogni traccia è suddivisa in **settori**
- L'insieme delle tracce nella stessa posizione su dischi diversi si chiama **cilindro**
- I settori sono l'**unità minima di memorizzazione**. Ogni settore memorizza un **blocco** di dati.
- L'HD viene visto dal sistema operativo come un **array unidimensionale di blocchi** logici. Oggi **la dimensione del blocco è di 4Kb (4096 byte)**
- Il blocco è la **più piccola unità di trasferimento** dei dati.
- Ogni settore contiene un blocco.
- **L'array** di blocchi logici (viene visto così dal sistema operativo) **viene tradotto in settori** del disco **in modo sequenziale**: Il settore 0 è il primo della traccia più esterna del primo piatto e poi a seguire.

**NOTA:** La realizzazione fisica del piatto (è circolare) fa sì che non sia possibile avere tracce della stessa grandezza: quelle verso il centro sono più piccole di quelle verso l'esterno, quindi le tracce esterne contengono più settori di quelle interne.

- **Seek time:** Il tempo che impiega la testina a **raggiungere la traccia** che contiene il settore contenente il blocco da leggere o scrivere.
- Per **minimizzare il seek time medio** il sistema operativo **riorganizza la sequenza di richieste dei processi** in modo che le testine debbano muoversi il meno possibile.

**NOTA:** La **latenza rotazionale** non può essere migliorata (non tramite software almeno) e quindi viene stimata al tempo di  $\frac{1}{2}$  rotazione completa.

## ALGORITMI DI SCHEDULING DEI DISCHI RIGIDI:

I processi effettuano delle richieste di accesso creando una sequenza di richieste che dev'essere gestita dal sistema operativo.

Per analizzare gli algoritmi usiamo la sequenza: 98, 183, 37, 122, 14, 124, 65, 67

Dove i singoli valori corrispondono al numero della traccia.

**NOTA:** L'ottimizzazione può interessare solo il movimento perpendicolare (dall'interno all'esterno e viceversa) del braccio, perciò ci interessa solo trovare la traccia e non il settore.

Un'altra considerazione da fare è che dato che **i piatti del disco ruotano tutti insieme e le testine si muovono tutte insieme**, quindi per semplicità si può supporre nell'analisi che esista un solo piatto.

### FCFS:

Le richieste sono gestite nell'ordine di arrivo (quindi nell'ordine in cui sono state generate dai processi)

### ANALISI DELL'ALGORITMO

sequenza: 98, 183, 37, 122, 14, 124, 65, 67

Assumendo di partire dalla traccia 53 (un numero a caso) e analizzando la sequenza di richieste la testina deve attraversare:

$$98 - 53 + 183 - 98 + 183 - 37 + \dots + 67 - 65 = 640 \text{ tracce}$$

### C-SCAN (Circular SCAN):

- Fornisce un tempo di attesa medio **più uniforme**.
- Tratta i settori /cilindri come una **lista circolare**:
  - La testina si muove **da un'estremo all'altro** servendo le richieste
  - Quando raggiunge l'estremità del disco torna indietro immediatamente **senza servire alcuna richiesta**

## ANALISI DELL'ALGORITMO

sequenza: 98, 183, 37, 122, 14, 124, 65, 67

Assumendo di partire dalla traccia 53 e un massimo di 200 tracce (da 0 a 199):

- Attraversa le tracce da 53 a 199 servendo le richieste: 37, 65, 67, 98, 122, 124, 183.
- torna a 0
- Attraversa le tracce da 0 a 37 per servire le richieste: 14, 37

La testina ha attraversato in tutto  $199 - 53 + 37 = 183$  tracce + 200 tracce per tornare indietro.

**NOTA:** Il numero totale di tracce attraversate sarebbe in effetti di 383 tracce ma bisogna considerare che il tempo impiegato dalla testina per tornare indietro è minimo perché non deve mai fermarsi e ripartire.

## **FORMATTAZIONE DEL DISCO:**

2 tipi di formattazione:

### FORMATTAZIONE FISICA (A BASSO LIVELLO)

- **Eseguita dal costruttore:** Associa un numero (il numero di blocco logico) ad ogni settore e assegna uno spazio (a ogni settore) dedicato al codice di correzione degli errori (in aggiunta ai 512/4096 byte del settore)
- Viene scelta la dimensione dei blocchi fisici (512/4096 byte per settore)

### FORMATTAZIONE LOGICA

- **Eseguita dal sistema operativo:** Necessaria per **creare e gestire il File System** del sistema operativo **sull'HD**
- Il sistema operativo crea la **lista di blocchi liberi**, e una directory iniziale da cui partono tutte le altre directory
- Sull'HD vengono riservate le aree gestite dal sistema operativo:
  - Il **boot block**: corrisponde in genere a più blocchi logici consecutivi sull'HD: **contiene il codice di avvio del sistema operativo**
  - L'area che contiene gli **attributi** dei file
- Il sistema operativo si riserva una porzione dedicata all'**area di swap**

## SISTEMI RAID:

- Un sistema RAID è composto da un **insieme di hard disk** (disk array) che **viene visto** dal sistema operativo **come un disco singolo**
- Il **controller del RAID** gestisce i dischi (fa in modo che il sistema operativo li veda come un disco singolo)

## 2 IDEE DI FONDO DEL SISTEMA RAID

- **Distribuire l'informazione su più dischi:** In modo da parallelizzare operazioni di accesso ai dati
- **Duplicare l'informazione su più dischi:** In modo da avere più backup

## RAID LIVELLO 0 (Block Striping)

- **Non c'è duplicazione** dei dati (di fatto non è un vero RAID)
- Il **disco virtuale** (ciò che viene visto dal sistema operativo: insieme di blocchi numerati consecutivamente) viene **diviso in strip**: Ogni strip contiene un certo numero di blocchi
- I dischi sono **suddivisi in strip** secondo la **tecnica dello striping**:
  - $\text{NUMERO\_STRIP} \% \text{DISCHI\_NEL\_SISTEMA}$
- **Maggiore velocità:** Le operazioni vengono eseguite in parallelo su tutti i dischi
- **Minore affidabilità:** Più dischi che possono rompersi.

## RAID LIVELLO 1 (Mirroring)

- Per ogni disco dati viene usato un **secondo disco di mirroring** che contiene una copia del disco

## RAID DI LIVELLO 01

- Combinazione dei livelli 0 e 1
- Fornisce i **massimi livelli di prestazione**
- Utilizza le **tecniche di striping e di mirroring**: Per avere le stesse prestazioni di un RAID 0 e la stessa affidabilità di un RAID 1 richiede di avere il doppio dei dischi.
- Accesso in parallelo anche nel caso di strip consecutivi perché si può accedere allo strip dal disco che ne contiene la copia  
**NOTA:** Alcuni sistemi prevedono un disco di spare (nel caso un disco si rompa)

## STRINGA DI PARITA'

E' possibile **ricostruire una stringa persa** se abbiamo calcolato (e salvato) la **parità bit a bit** delle stringhe

**PARITA' con n stringhe = String 0 XOR String 1 XOR ... XOR String n**

Es.

3 stringhe di 8 bit: a, b, c

a)	01100011	----->	parità:	00000011
b)	10101010		b)	10101010
c)	11001010		c)	11001010
parità = 00000011			a)	01100011

**NOTA:** Per calcolare la parità le stringhe devono avere la stessa lunghezza, ma la lunghezza può essere arbitraria, quindi possiamo considerare stringhe formate da **tutti i bit che compongono uno strip** del sistema RAID

## RAID DI LIVELLO 4 (Striping con parità)

- Usa lo striping (RAID 0) a livello di blocchi
- Calcola uno **strip di parità** per poter implementare operazioni di recovery:
  - Un disco memorizza gli strip di parità **nella stessa posizione** degli strip corrispondenti
- E' necessario ricalcolare la parità dello strip ad ogni modifica

## CONFRONTO RAID 4 E RAID 01

- RAID 4 Risparmia spazio perché non c'è bisogno di raddoppiare ogni disco ma ne basta uno in più
- Nel RAID 4 il **recupero dei dati** in caso di guasto è **più lento** perché bisogna calcolare lo strip di parità. Nel RAID 01 è sufficiente recuperare i dati nel disco di backup.
- Il disco di parità del RAID 4 è un collo di bottiglia e rischia di danneggiarsi a causa della sollecitazione continua.

## RAID DI LIVELLO 5 (Striping con parità distribuita)

- Stesso funzionamento del 4 ma distribuisce gli strip di parità su tutti i dischi

## RAID DI LIVELLO 6 (Striping con doppia parità)

- Resiste anche al guasto di 2 dischi contemporaneamente

## MEMORIE A STATO SOLIDO:

- SSD(Solid State Disk): Memorie a stato solido montate su schede
- PEN DRIVE(Pen flash): Memorie a stato solido incapsulate in un contenitore rimovibile tramite usb
- Le memorie a stato solido sono **organizzate in pagine** da 2 a 16 Kbyte e **ogni operazione coinvolge l'intera pagina**.
- Un SSD può complementare un HD (se non sostituirlo del tutto) in 2 modi
  - Come **livello di memoria cache** permanente collocata tra l'HD e la RAM
  - Come secondo HD (più veloce) su cui memorizzare sistema operativo, applicativi, file o l'area di swap.
- Solitamente gli SSD usano uno scheduling FCFS per servire le richieste provenienti dai processi.

## INTERFACCIA DEL FILE SYSTEM:

- **Fornisce i meccanismi per la memorizzazione e l'accesso ai dati e agli applicativi**
- Il File system è composto da 2 parti:
  - Un **insieme di file**
  - Una struttura di **directory** che permette di organizzare i files

## IL FILE

- **Unità logica di informazione** memorizzata **permanentemente** su un supporto di **memoria secondaria**.
- Un file è **dotato di**:
  - **Nome**
  - **Posizione logica** all'interno del file system
  - **Attributi** (proprietà): dimensioni, diritti di accesso, data di creazione, accesso, modifica, etc...

## ATTRIBUTI DEI FILE

- **PATHNAME DEL FILE:** E' la **posizione logica** del file **all'interno del File system**

**NOTA:** Questa informazione **non è memorizzata** da nessuna parte (è un'informazione che serve solo all'utente)

- **Dimensione** corrente del file.
- **Permessi di accesso/protezione**
- **Data e ora:** creazione, modifica, ultimo accesso
- **Identificazione del proprietario:** Specifica l'utente proprietario del file.

**NOTA:** Un file può essere visto come un tipo di dato astratto **definito dalle operazioni** che possono essere fatte su quello stesso file.

## **OPERAZIONI SUI FILE:**

- **Creazione:** Il sistema operativo deve trovare lo spazio per il file e **creare un accesso al file** attraverso la directory
- **Lettura/scrittura:** Il sistema operativo mette a disposizione delle system call per accedere al file specificando la modalità. **Per ogni file aperto** Il sistema operativo deve **gestire un puntatore (il file position indicator)** che indica la posizione corrente all'interno del file.

**NOTA:** Il sistema operativo deve gestire la variazione della dimensione (aumento o diminuzione)

- **Rimozione:** Il sistema operativo recupera lo spazio occupato in memoria secondaria
- **Troncamento:** Cancella i dati memorizzati e recupera lo spazio ma **mantiene gli attributi**
- Rinominare, copiare, spostare.

## **METODI DI ACCESSO:**

- **Sequenziale:** I dati vengono acceduti dall'inizio del file
- **Accesso diretto:** Si accede direttamente al byte interessato

**NOTA:** I metodi di allocazione dei file in memoria secondaria vanno valutati anche da come permettono di implementare in modo più o meno efficiente l'accesso diretto.

## LE DIRECTORY:

Le directory **sono file** che **contengono le informazioni relative ai file** a cui si riferiscono.

**NOTA:** Perdere i dati di una directory comporta quasi sempre la perdita di accesso al file.

### STRUTTURA DELLA DIRECTORY

- Ogni file directory contiene **un certo numero di entry**
- Ogni entry contiene il **nome simbolico**(dato dall'utente) più **una o più informazioni aggiuntive:**

- **gli attributi del file**

oppure

- **un puntatore ad una struttura che li contiene**

**NOTA:** Un file directory non può essere modificato **nemmeno dall'utente proprietario**. Il sistema operativo garantisce l'integrità della struttura di ogni directory.

## ESEMPI DI STRUTTURE DEI FILE SYSTEM:

### MS\_DOS:

NOME_FILE_1.ESTENSIONE_1	ATTRIBUTI
NOME_FILE_2.ESTENSIONE_2	ATTRIBUTI
NOME_FILE_3.ESTENSIONE_3	ATTRIBUTI
NOME_FILE_4.ESTENSIONE_4	ATTRIBUTI
...	...

In dettaglio:

Ogni file directory era fatto da una serie di entry di 32 byte suddivise come:

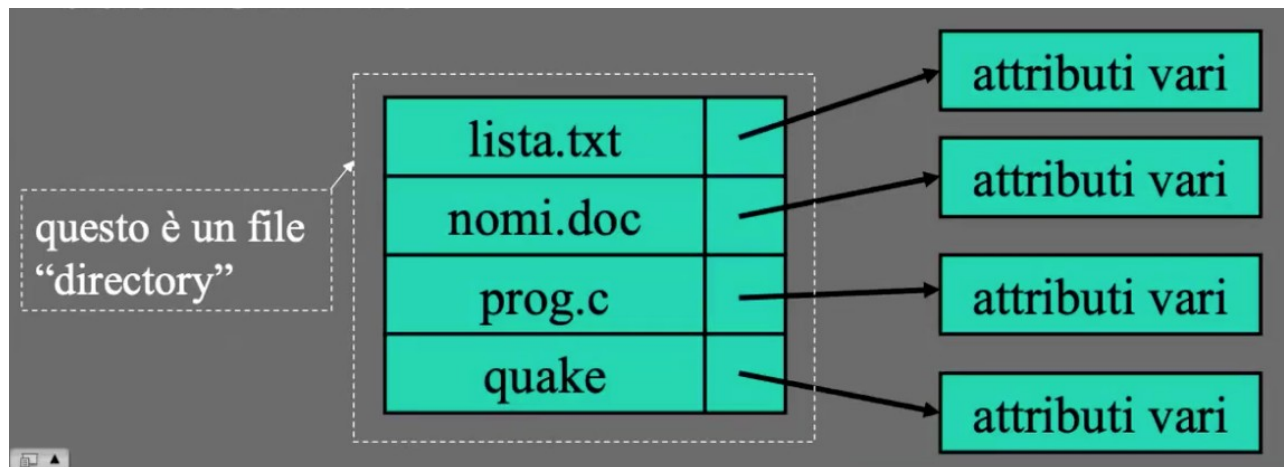
FILE_NAME	.EXT	ATTRIBUTI	RESERVED	TIME	DATE	FIRST BLOCK NUM	SIZE
8	3	1	10	2	2	2	4

**NOTA:** Il First Block Number contiene il numero del primo blocco in memoria che contiene i dati relativi al file.

**NOTA:** Se usiamo 4 byte per indicare la dimensione del file significa che stiamo imponendo che la grandezza massima del file possa essere di  $2^{32} - 1$  byte.

## UNIX:

Unix utilizza un **puntatore a una struttura interna**, memorizzata in memoria secondaria e gestita direttamente dal sistema operativo che contiene le informazioni sul file.



## NTFS:

Nel file system NTFS (New Technology File System) usato in ambiente Windows, da Windows XP, Le directory sono organizzate con un **albero di ricerca bilanciato**, in cui ogni foglia corrisponde a un file.

## ORGANIZZAZIONE DEL FILE SYSTEM:

### /\* FILE SYSTEM A DUE LIVELLI

- Una directory principale che **contiene i riferimenti alle directory secondarie** di ogni utente
- Nasce il concetto di pathname. \*/

### FILE SYSTEM GERARCHICO

- Ogni directory può contenere:
  - riferimento a un **file**
- oppure
  - un riferimento a un'altra **directory**
- **Root:** La radice del file system, cioè la directory dalla quale si diramano tutte le altre.
- **Home:** Il nodo dal quale si dirama la porzione del file system **dedicato all'utente**
- **Working directory (current):** La directory corrente sulla quale l'utente si trova durante l'utilizzo del sistema.

- **Path name assoluto:** Il percorso per raggiungere un file **partendo dalla root**.

**NOTA:** In ambiente Windows la root si indica con \ “backslash”. In ambiente Unix si usa / “slash”

- **Path name relativo:** Il percorso per raggiungere un file **partendo dalla current/working directory**.

**NOTA:** Un pathname relativo inizia sempre con il **nome** della directory, inoltre:

- . (punto) indica la current directory
- .. (punto punto) indica la **parent**
- **NOTA:** E' possibile navigare il file system **dal basso verso l'alto** utilizzando ../../.. etc...
- **La struttura ad albero non permette di condividere file o directory con nomi diversi**

## DIRECTORY CON STRUTTURA A GRAFO ACICLICO:

- **Permette di condividere file con nomi diversi**
- **Link:** I collegamenti a file o directory
- Permette di accedere a un file (o ad una directory) da percorsi diversi

## /\* DIRECTORY A GRAFO GENERALE

- Una directory può “contenere” **il nome di una directory parent**

**NOTA:** Questo può generare cicli infiniti

- Le **strutture a grafo generale sono proibite** perché possono generare cicli infiniti e situazioni difficili da gestire. \*/

## ACCESSO RAPIDO AI FILE:

**L'accesso ai file tramite pathname sarebbe estremamente inefficiente** perché richiederebbe un accesso in memoria secondaria ogni volta che si devono verificare i dati relativi a una directory del path, fino ad arrivare al file.

APERTURA DEI FILE: L'apertura di un file (tramite system call “open” o “fopen”) **copia alcune informazioni fondamentali relative al file nella OPEN FILE TABLE** (struttura del sistema operativo). **Dopo l'apertura del file l'accesso passa dalle informazioni relative, contenute nella open file table del sistema operativo che si trova in RAM.** Quando un programma chiude un file le informazioni relative nella open file table possono essere rimosse.

## INFORMAZIONI COPIATE NELLA OPEN FILE TABLE:

- **Attributi** del file
- Una **porzione del file**: Se viene fatta una modifica, questa viene fatta in RAM sulla copia del file e solo successivamente il sistema operativo copia le modifiche in memoria secondaria.

## REALIZZAZIONE DEL FILE SYSTEM: METODI DI ALLOCAZIONE DEI FILES

- Il sistema operativo vede l'HD come un enorme **array** , in cui **ogni entry è una sequenza di 4096 byte** (4 Kbyte)
- Il sistema operativo accede ogni blocco comunicando il numero di blocco al **controller del disco** (ogni operazione viene fatta su un blocco):
  - Il controller del disco riceve il comando dal sistema operativo, recupera il blocco all'interno del settore corrispondente e **invia i dati in RAM tramite il DMA (Direct Memory Access)**: Canale di comunicazione diretta tra memoria secondaria e RAM (Vedi architetture1)
- Il sistema operativo memorizza ogni file nei blocchi dell'HD
- Per ogni file il sistema operativo mantiene una Open File Table

**NOTA:** Quando il file supera la dimensione di un blocco è necessario allocarlo su più blocchi.

## 3 STRATEGIE DI ALLOCAZIONE PER FILE SU PIU' BLOCCHI:

### ALLOCAZIONE CONTIGUA

- Ogni file è allocato in blocchi contigui dell'HD
- **Per recuperare il file è sufficiente memorizzare negli attributi Il numero del primo blocco.**

**NOTA:** Non è necessario conoscere il numero di blocchi occupati perché negli attributi è sempre memorizzata la dimensione.

### VANTAGGI

- **Accesso semplice e veloce:** la testina viene posizionata solo una volta (all'inizio del primo blocco, poi i dati verranno letti in sequenza)
- **Sono sufficienti poche informazioni**

## SVANTAGGI

- Produce **frammentazione esterna**
- **Se la dimensione del file aumenta bisogna riallocarlo**

## ALLOCAZIONE CONCATENATA

- Catena di blocchi implementata come una **linked list**: Ogni blocco contiene (negli ultimi byte) un **puntatore al blocco in cui prosegue il file**.
- Per risalire al blocco è sufficiente memorizzare tra gli attributi del file il numero del primo blocco da cui parte la catena.
- Nell'ultimo blocco, nello spazio riservato al puntatore viene memorizzato un valore negativo (o 0) (come per le linked list l'ultimo nodo contiene NULL)

## VANTAGGI

- Non soffre di frammentazione esterna

## SVANTAGGI

- Accesso diretto al file **altamente inefficiente**:  
In generale **sono necessari n accessi al disco per accedere all'n-esimo blocco**: bisogna leggere il primo per poter accedere al secondo, leggere il secondo per accedere al terzo, e così via..
- Se si danneggia un blocco **si perde l'accesso** alla catena da quel blocco in avanti

## CLUSTER DI BLOCCHI

- L'HD viene visto come una **successione di cluster di blocchi** (unisce i vantaggi dell'allocazione contigua e concatenata... ..ma anche i problemi)
- I blocchi nel cluster sono **adiacenti**, quindi in pratica è come avere blocchi più grandi

## VANTAGGI

- **Tempi di accesso ai file si riducono**, perché si deve riposizionare meno volte la testina (vantaggio dell'allocazione contigua)
- **Minore spreco di spazio per i puntatori**

## SVANTAGGI

- **Aumenta la frammentazione esterna** (problema dell'allocazione contigua)

## ALLOCAZIONE CONCATENATA CON FAT (File Allocation Table)

- E' un area all'inizio del disco gestita come **un array in cui l'indice di ogni entry corrisponde a un blocco.**
- Ogni entry contiene il **numero di un blocco (o cluster)** in cui è contenuto un file **oppure 0** se il blocco è vuoto, **oppure un valore speciale che segnala la fine del file**
- La FAT **riproduce una lista concatenata utilizzando un array** in cui:
  - **L'indice dell'array si riferisce al blocco "i" in memoria**
  - **Lo slot "i" dell'array (FAT[i]) contiene il numero del blocco successivo** in cui sono memorizzati i dati del file.
  - Se l'ultimo blocco di file è il numero j, allora la j-esima entry della FAT (FAT[j]) contiene il marcatore di fine file.

**NOTA:** La FAT **riproduce una linked list senza usare una linked list:** ha i vantaggi dell'allocazione concatenata senza il problema dei puntatori.

## VANTAGGI

- La FAT è **tenuta in RAM**, in questo modo **l'accesso diretto ai file migliora notevolmente.**
- In caso di perdita (corruzione) di un blocco possiamo risalire alla catena di blocchi riprodotta nella FAT
- Gestione automatica dei blocchi liberi: Sono i blocchi rappresentati dalle entry il cui valore è 0.

## SVANTAGGI

- **Occupazione spazio** in memoria principale: tanto più è grande il disco tanto più è grande la FAT, a meno di utilizzare cluster (che però aumentano la frammentazione)
- Se la FAT viene persa **non c'è più modo di accedere ai dati**, quindi deve essere frequentemente salvata sull'HD

## DOMANDA ESAME:

Calcolare la dimensione della FAT con:

- HD da 20 Gb
- dimensione del blocco da 1Kb

Calcolare la percentuale del disco “sprecata” per la FAT

**SOLUZIONE:** Per risolvere questo tipo di esercizi dobbiamo capire di quanti bit(byte) abbiamo bisogno per scrivere i numeri dei blocchi. Il numero dei bit necessari **determina la dimensione delle entry** della FAT.

1. Tradurre in base 2 la dimensione dell'HD e del blocco

- $HD = 20 \text{ Gb} = 20 * 2^{30}$
- $BLOCCO = 1 \text{ Kb} = 2^{10}$

2. Dividere la dimensione dell'HD per la dimensione del singolo blocco in modo da ottenere il numero di blocchi.

- $20 * 2^{30} / 2^{10} = 20 * 2^{20}$

3. Determinare la dimensione delle entry della FAT calcolando il numero di bit necessari per poter scrivere i numeri di tutti i blocchi.

- $20 * 2^{20} \approx 20.000.000$

**NOTA:** Per calcolare velocemente il numero di bit necessari a scrivere un qualsiasi numero, basta calcolare la **maggiore potenza di 2 minore o uguale** al numero cercato, in quest'esempio:

- $20 = 2^4$  ( $2^4$  è la maggiore potenza di 2 minore di 5)
- $2^{20}$  (è già espresso come potenza di 2)
- $2^4 * 2^{20} = 2^{24}$  è la massima potenza di 2 minore di  $20 * 2^{20}$  quindi ci servono **25 bit (no, non 24)**.

**NOTA:** Se  $2^{24}$  è la potenza massima prima di superare il numero, allora vuol dire che il 25-esimo bit dev'essere a 1 perché i bit si contano **partendo da 0**.

- Dato che, **come nel caso delle entry della PT** la dimensione è sempre **arrotondata al byte** (8 bit) useremo 32 bit = 4 byte per le entry della FAT.

4. Moltiplicare la dimensione della entry per il numero di blocchi.

- $4 * 20 * 2^{20} = 80 \text{ Mb}$
- la percentuale di spazio “sprecato” è lo 0,0004 %

## ALLOCAZIONE INDICIZZATA

- Si tiene traccia di tutti i blocchi **che contengono un file** scrivendone il numero in **un altro blocco: il blocco indice**.
- Ci sono tanti blocchi indice quanti sono i file
- **Ogni entry del blocco indice contiene un numero di blocco** che contiene una porzione del file, le entry vuote sono inizializzate con -1.
- Approccio **simile alla paginazione**

## VANTAGGI

- **Non c'è frammentazione esterna** (non c'è bisogno di avere blocchi contigui)
- **Accesso diretto al file efficiente**, una volta portato il **blocco indice in RAM** basta visitare le entry per conoscere la posizione dei blocchi dov'è contenuto il file.

## SVANTAGGI

- **Frammentazione interna:** per file molto piccoli, ad esempio se un file è contenuto in un unico blocco, **si spreca tutto lo spazio del blocco indice** (non ci sarebbe nemmeno bisogno di averlo).

**REGOLA GENERALE:** In un file system, fino al 90% dei file **non supera la dimensione di un blocco**.

**PROBLEMA:** Come fare se il blocco indice **non è sufficientemente grande** per memorizzare tutti i blocchi in cui è contenuto il file?

**Esempio:** Con blocchi da 1 Kb quanti blocchi possono essere riferiti da un blocco indice?

- $1\text{Kb} = 1024 = 2^{10}$  quindi abbiamo bisogno di 11 bit
- La dimensione delle entry è arrotondata al byte quindi ci servono 16 bit = 2 byte per scrivere il numero di ogni blocco.
- $1024 / 2 = 512$  entry, quindi un blocco indice può riferire 512 blocchi
- Dato che ciascun blocco è della dimensione di 1 Kb possiamo avere file grandi al massimo 512Kb

**SOLUZIONE:**

## SCHEMA CONCATENATO

- L'ultima entry del blocco indice **punta ad un altro blocco indice**, e così via.

## SCHEMA A PIU' LIVELLI

- Il blocco indice contiene solo **puntatori ad altri blocchi indice** (simile alla **paginazione a 2 livelli**).
- **Ogni blocco indice interno riferisce n blocchi** che contengono i dati del file.

## UNIX: GLI I-NODE

- **i-node:** E' la **struttura dati** utilizzata per la gestione dei file nei sistemi operativi Unix-like: **contiene tutti gli attributi e l'elenco dei blocchi dati** di un file
- **Ad ogni file** (cioè ad ogni file e ogni directory-file) **è associato un i-node (index node)**
- Sono gestiti direttamente dal sistema operativo e memorizzati sull'HD in una porzione riservata (al sistema operativo)
- **Struttura dell' i-node:** Per tenere traccia di tutti i blocchi dati del file, ogni i-node contiene lo spazio per memorizzare:
  - **Metadati del file:** I **primi campi** dell'i-node servono per memorizzare gli **attributi del file** (spiegazione in dettaglio nella sezione LINK UNIX)
  - **10 puntatori diretti a blocchi** di dati del file (10 blocchi dati)
  - **1 puntatore single indirect:** Punta a un **blocco indice** che contiene **puntatori a blocchi di dati** del file. ( $\rightarrow \square \rightarrow \square\square\square\square$ ) ( $1 = n$ )

**NOTA:** Il numero di blocchi riferiti dal blocco indice **dipende dalla dimensione delle entry**, cioè dal numero di byte utilizzati per scrivere il numero dei blocchi. Es. Se utilizziamo 4 byte per scrivere il numero del blocco e la dimensione di ogni blocco è 1Kb (1024 byte) allora il blocco indice può contenere  $1024 / 4 = 256$  riferimenti a blocchi.

- [illegible]

## CALCOLARE LA DIMENSIONE MASSIMA DI UN FILE IN UN SISTEMA UNIX:

$\text{num\_pointers} = \text{DIMENSIONE\_BLOCCO} / \text{DIMENSIONE\_ENTRY};$

- puntatori diretti:  $10 * \text{DIMENSIONE\_BLOCCO} +$
- single indirect:  $\text{num\_pointers} * \text{DIMENSIONE\_BLOCCO} +$
- double indirect:  $\text{num\_pointers}^2 * \text{DIMENSIONE\_BLOCCO} +$
- double indirect:  $\text{num\_pointers}^3 * \text{DIMENSIONE\_BLOCCO}$

**Esercizio:** Qual'è la dimensione massima di un file in un sistema Unix che adotta il sistema degli i-node se i blocchi sono grandi 1 Kb e usiamo 4 byte per scrivere il numero di un blocco?

### SOLUZIONE:

$\text{num\_pointers} = 1 \text{ Kb} = 1024 \text{ byte} / 4 \text{ byte} = 256$  **entry del blocco indice**

$10 * 1024 + 256 * 1024 + 256^2 * 1024 + 256^3 * 1024 = \text{TANTO}$

**NOTA:** Tutti i sistemi ambiente Unix implementano il file system utilizzando gli i-node.

## CALCOLARE GLI ACCESSI IN MEMORIA SECONDARIA CON GLI I-NODE:

- PUNTATORI DIRETTI: **2 accessi**
  - 1 accesso all'i-node (che viene poi portato in RAM)
  - 1 accesso al blocco dati
- SINGLE INDIRECT: **3 accessi**
  - 1 accesso all'i-node (che viene poi portato in RAM)
  - 1 accesso **al blocco indice**
  - 1 accesso al blocco dati
- DOUBLE INDIRECT: **4 accessi**
  - 2 accessi ai blocchi indice + gli altri 2
- TRIPLE INDIRECT: **5 accessi**
  - 3 accessi ai blocchi indice + 2

## NTFS(New Tecnology File System)

- **Ogni file è descritto da un elemento**
- Gli elementi hanno **dimensione fissa** (da 1 a 4 Kbyte)
- Gli elementi **sono numerati consecutivamente**
- Tutti gli elementi del file system sono contenuti nella **Master File Table (MFT)**:  
Un file gestito dal sistema operativo memorizzato in uno spazio dedicato sul disco.
- **File reference**: Il numero dell'elemento che riferisce un file: è associato al nome del file nella directory
- Un elemento contiene gli attributi del file corrispondente

### **DIFFERENZA TRA NTFS E I-NODE:**

- Nei file system NTFS se il file è molto piccolo, **i dati del file possono essere contenuti nell'elemento corrispondente** (a differenza degli i-node)  
  
**NOTA:** In questo caso l'**accesso diretto al file risulta molto efficiente**, è sufficiente fare un solo accesso in memoria secondaria per ottenere i dati.
- NTFS implementa una variante dell'allocazione indicizzata a schema concatenato: L'elemento contiene più **puntatori a cluster di blocchi dati**, e uno (o più) **puntatori a blocchi di puntatori**
- Unix implementa una variante dell'allocazione indicizzata con schema a più livelli.

### **GESTIONE DELLO SPAZIO LIBERO SU DISCO:**

- Unix tiene traccia dello spazio libero usando il **superblocco** (in Windows è la Master File Table (MFT)) contenuto nei blocchi iniziali dell'HD **a partire dal blocco numero 1**. (dopo ci sono gli i-node o gli elementi (in Windows))
- **Superblocco**: Struttura dati del sistema operativo che tiene traccia dei blocchi liberi (lista di blocchi liberi)
- Una lista di blocchi può essere implementata usando un **vettore di bit**: Un array di dimensione pari al numero dei blocchi (o dei cluster) che indica se il blocco corrispondente alla entry dell'array è libero:  $\text{bit}[i] = 1$ , oppure occupato:  $\text{bit}[i] = 0$  (sistema usato da MAC/OS) ...
- ... oppure **usando la FAT**

**NOTA:** Per qualunque implementazione del file system è importante che il sistema operativo implementi un **sistema di caching** dei file in RAM (soprattutto dei file aperti)

## LINK UNIX:

### Recap:

- In Unix ogni **file è identificato univocamente dall'i-node** (cioè dalla struttura dati del sistema operativo utilizzata per la gestione dei file)
- Ogni **i-node è identificato** all'interno del file system grazie al **proprio numero** (come i processi dal PID)
- **RICORDA:** La directory è **UN FILE** formato da un insieme di entry: **ogni entry contiene due campi:**
  - **nome del file**
  - **numero dell'i-node** associato al file.
- Quando l'utente specifica il nome di un file in una cartella, il sistema operativo legge il numero dell'i-node corrispondente, recupera l'i-node sul disco (1° accesso in memoria) e può così accedere alle informazioni (contenute nei blocchi) (facendo altri accessi in memoria)

## DUE TIPI DI LINK IN UNIX:

### HARD LINK(Link fisico)

- Il **nome mnemonico** (*stringa*) del file **dato dall'utente**.
- **Regular file** (file regolare): Il tipo di file più comune: Contengono **dati generati dall'utente** (o da programmi o applicazioni).  
Altri tipi di files in Unix sono le Directory, i Symbolic Link, i Device Files, Pipes, etc..
- In Unix un regular file può avere **uno o più link fisici, cioè lo stesso file può avere nomi diversi**.
- In altre parole, una directory può avere più entry che contengono lo **stesso numero di i-node riferito a nomi diversi**

**NOTA:** Una cartella non può contenere due entry uguali (stesso nome\_file e stesso numero di i-node)

## LINK COUNTER:

- E' un **campo di tipo int dell'i-node**
- Il link counter **conta il numero di hard link** che si riferiscono a quel file. Alla creazione **di un regular file** viene **inizializzato a 1**.
- **E' possibile** creare un nuovo hard link da riga di comando: **comando ln** oppure tramite la **system call: link()**

## NOTA:

- Alla creazione di una nuova directory il **link counter della cartella creata è inizializzato a 2**:
- La nuova directory contiene **almeno due entry** che contengono:
  - Il **numero dell'i-node** (una directory è un file) **riferito al nome “.”** : La directory stessa
  - Il **numero dell'i-node riferito al nome “..”** : La directory parent
- Il link counter della **cartella parent viene incrementato di 1**, perché c'è una nuova cartella (quella appena creata) che si riferisce a quel file.

**Domanda:** Leggendo l'i-node di una cartella come facciamo a sapere quante sottocartelle contiene?

**Risposta:**  $n - 2$  (non consideriamo “.” e “..”) , con  $n$  = valore del link counter.

- **Non è possibile** creare hard link che si riferiscono a directory perché questo **porterebbe alla creazione di cicli** (grafo generale proibito)
- **Non è possibile** creare hard link a file **che stanno in volumi o partizioni diverse**

## SIMBOLIC LINK(Soft link)

- Sono **puntatori a file**
- Implementati allocando un **nuovo i-node** che viene **associato al link simbolico**
- L'associazione viene fatta **specificando nel campo File type** dell'i-node il valore corrispondente al link simbolico (il valore è rappresentato dalla costante “S\_IFLINK” che si trova in <sys/stat.h>. In genere 0120000 (in ottale) , 40960 (in decimale))

## ESERICIZI PREPARATORI:

1) Su un HD che usa allocazione concatenata (senza FAT) è memorizzato un file A di dimensione 0x8000 byte, si sa che nell'ultimo blocco di A sono presenti 64 byte del file. Sapendo che per scrivere il numero di un blocco vengono usati 27 bit arrotondati al numero minimo di byte. Quanto è grande l'HD?

**SOLUZIONE:** La chiave per risolvere l'esercizio è ragionare sulla dimensione dei blocchi e su come viene implementata l'allocazione concatenata:

- La grandezza di un blocco è un multiplo di 2
- La dimensione del file è anch'essa un multiplo di 2:  
 $0x8000 \text{ byte} = 8 * 16^3 = 2^3 * 2^4 * 3 = 2^{15} = 32\text{Kb}$
- Di conseguenza il file dovrebbe occupare completamente tutti i blocchi, ma questo non è possibile perché l'allocazione concatenata prevede che in ogni blocco si riservi un po' di spazio (in questo caso 4 byte) per il puntatore al blocco successivo.
- Quindi si può considerare lo spazio occupato dell'ultimo blocco 64 byte come la somma dei puntatori (cioè lo spazio che contiene il numero del blocco successivo) di ogni blocco usato per contenere il file A
- Se usiamo 4 byte per scrivere il numero di un blocco allora se facciamo  $64/4$  otteniamo il numero di blocchi usati per contenere il file A, cioè 16
- Dividendo la dimensione del file A per il numero di blocchi che lo contengono otteniamo la grandezza del blocco:  $32\text{Kb} / 16 = 2\text{Kb}$  (grandezza di un blocco)
- Moltiplichiamo la dimensione del blocco per il numero dei blocchi, cioè  $2^{27}$  (nota che se vengono usati 27 bit per scrivere il numero di ogni blocco allora potremo indirizzare al massimo  $2^{27}$  blocchi):  $2\text{Kb} * 2^{27} = 2^{11} * 2^{27} = 2^{38} = 2^8 * 2^{30} = 256\text{Gb}$
- La dimensione dell'HD è 256 Gb

2) L'HD di un sistema operativo Unix ha dimensione 512 Gb ed è suddiviso in blocchi da 0x1000 byte. Di un file A si sa che, una volta in RAM i suoi attributi, incluso il numero del suo index-node, sono necessarie 4 operazioni di I/O per leggere l'ultimo byte del file. Qual'è la dimensione **minima** di A?

**SOLUZIONE:** Per risolvere l'esercizio è necessario ricordare la formula per calcolare la dimensione di un file nell'allocazione con gli i-node in Unix, cioè:

$\text{num\_pointers} = \text{DIMENSIONE\_BLOCCO} / \text{DIMENSIONE\_ENTRY};$

- puntatori diretti:  $10 * \text{DIMENSIONE\_BLOCCO} +$
- single indirect:  $\text{num\_pointers} * \text{DIMENSIONE\_BLOCCO} +$
- double indirect:  $\text{num\_pointers}^2 * \text{DIMENSIONE\_BLOCCO} +$
- double indirect:  $\text{num\_pointers}^3 * \text{DIMENSIONE\_BLOCCO}$

QUINDI:

- La prima cosa da fare è calcolare il numero di entry del blocco indice: Sappiamo che la dimensione dell'HD è 512Gb =  $2^{39}$  byte e che la dimensione di ogni blocco è  $0x1000 = 1 * 16^3 = 4096 = 4Kb = 2^{12}$  byte
- Quindi abbiamo  $2^{39} / 2^{12} = 2^{27}$  blocchi indirizzabili, quindi abbiamo bisogno di 27 bit per poter scrivere il numero di ogni blocco
- Se usiamo il **minor numero di byte** per indirizzare ogni blocco allora scriveremo il numero su 4 byte, quindi ogni entry avrà la dimensione di 4 byte.
- Il numero di entry del blocco indice sarà, quindi, la dimensione del blocco diviso la dimensione delle entry:  $2^{12} / 2^2 = 2^{10}$
- Ora è possibile calcolare la dimensione del file facendo attenzione che dobbiamo calcolare la dimensione **minima**:
- $10 * 2^{12} + 2^{10} * 2^{12} + 1 = 10 * 2^{12} + 2^{22} + 1$
- **NOTA:** Se sono necessarie 4 operazioni di I/O per leggere l'ultimo byte allora significa che dobbiamo usare il puntatore di doppia indirezione, ma l'esercizio ci chiede di calcolare la dimensione **minima** del file, questo significa che ci basta considerare il **primo byte, rappresentato da + 1 nella soluzione**, successivo ai  $2^{12}$  blocchi riferiti dai puntatori di singola indirezione.
- Se, con gli stessi dati del problema, viene richiesto di calcolare la dimensione massima di un file dobbiamo calcolare:
- $10 * 2^{12} + 2^{10} * 2^{12} + 2^{20} * 2^{12} = 10 * 2^{12} + 2^{22} + 2^{32}$

3) In un sistema operativo un indirizzo fisico è scritto su 28 bit, l'offset più grande in una pagina è 3FFF, lo spazio logico è il doppio di quello fisico e nel sistema possono essere presenti contemporaneamente 1024 processi.

Se il sistema adottasse una IPT (Inverted Page Table) quanto sarebbe grande questa tabella?

**SOLUZIONE:** Per rispondere alla domanda è necessario ricordare la struttura e il funzionamento della IPT:

- La IPT descrive **l'occupazione dei frame della memoria fisica da parte di tutti i processi**
- Ogni entry della IPT ha la forma **<process-ID, page number>**

QUINDI:

- Lo spazio logico ha dimensione  $2^{29}$ , cioè  $2 * 2^{28}$  essendo il doppio dello spazio fisico.
- Usiamo 14 bit per l'offset di una pagina:  $3FFF = 11\ 1111\ 1111\ 1111$
- se abbiamo 1024 processi contemporaneamente attivi significa che dobbiamo poter tenere traccia di  $1024 = 2^{10}$  PID all'interno della IPT, questo significa che ci servono 10 bit per poter scrivere il numero di ogni processo.
- Se l'indirizzo fisico è scritto su 28 bit e usiamo 14 bit per l'offset allora questo significa che possiamo indirizzare  $2^{14}$  frame/pagine, e quindi, dato che **ogni indice della IPT** (che è un array) **corrisponde ad un frame** allora la ipt avrà  $2^{14}$  entry
- Otteniamo il numero delle pagine (in cui è diviso lo spazio logico) dividendo la dimensione dello spazio logico per la dimensione dell'offset, e quindi:  $2^{29} / 2^{14} = 2^{15}$
- Quindi avremo che per ogni entry scriveremo 10 bit (per i PID) + 15 bit per le pagine dei processi, quindi 25 bit per ogni entry.
- La dimensione delle entry della tabella delle pagine è arrotondata al byte, quindi usiamo 4 byte per ogni entry
- Moltiplichiamo il numero di entry per la dimensione e otteniamo:  $4\ \text{byte} * 2^{14} = 4 * 2^4 * 2^{10} = 64\text{Kb}$
- La dimensione della IPT è al massimo 64 Kb

4) Un sistema ha un tempo di accesso in RAM di 70 ns, adotta un TLB con un tempo di accesso di 10 ns e un HIT RATE del 95% e usa un algoritmo di rimpiazzamento delle pagine. Quando si verifica un miss, nel 20% dei casi la pagina indirizzata non è in RAM e il page fault ha un costo di gestione di 1 microsecondo, indipendentemente dal valore del dirty bit. Qual'è l'effective access time (eat) del sistema? (non considerare il costo di accesso al TLB in caso di miss e, in caso di page fault, considerare solo il costo di gestione)

**SOLUZIONE:**

- Per risolvere il problema bisogna dividere i casi:
- $0.95 * (80 + 20)$  rappresenta i casi di HIT (95%), in cui avremo un solo accesso in RAM (per accedere ai dati) del costo di 70 ns + il costo di accesso al TLB 10 ns
- Nel rimanente 5% dei casi avremo che:
  - nel 2% dei casi abbiamo un page fault con un costo di gestione di 1000 nanosec (1 microsec)
  - mentre nel rimanente 8% (in caso di miss senza page fault) avremo 2 accessi in RAM, il primo per accedere alla page table e il secondo per accedere al dato, con un costo del doppio di un accesso in RAM ( $70 \text{ ns} * 2$ )
- **RISULTATO:**  $0.95 * (70+10) + 0.05 * (0.2 * 1000 + 0.8 * (70 * 2)) = 91.6$

5) In un sistema paginato è noto che lo spreco di memoria primaria dovuto alla frammentazione interna è in media di circa 1Kb per processo. Un indirizzo fisico è scritto su 26 bit e lo spazio di indirizzamento logico è 4 volte quello fisico. Qual'è la dimensione della tabella delle pagine più grande di questo sistema?

**SOLUZIONE:** Per rispondere a questa domanda dobbiamo ricordare che la frammentazione in un sistema paginato è in media di mezza pagina per processo quindi le pagine hanno dimensione di  $2\text{Kb} = 2^{11}$ , detto ciò possiamo procedere con la risoluzione:

- 26 bit per scrivere un indirizzo fisico significa avere uno spazio di indirizzamento fisico di  $2^{26}$  (byte)
- Lo spazio logico è 4 volte quello fisico, quindi:  $2^2 * 2^{26} = 2^{28}$ , quindi usiamo 28 bit per un indirizzo logico.
- Sapendo che una pagina è grande  $2\text{Kb} = 2^{11}$  allora il numero di pagine che possiamo indirizzare con indirizzo logico è:  $2^{28} - 2^{11} = 2^{17}$  pagine.
- Ogni entry della PT deve contenere un numero di frame, se usiamo 26 bit per l'indirizzo fisico e un frame è grande  $2^{11}$  byte allora avremo  $2^{26} / 2^{11} = 2^{15}$  frame. Arrotondiamo al byte la dimensione della entry e quindi otteniamo 16 bit = 2 byte per scrivere il numero di ogni frame, quindi ogni entry della PT è grande 2 byte
- Quindi abbiamo che  $2^{17} \text{ entry} * 2 \text{ byte} = 2 * 2^7 * 2^{10} = 2^9 * 2^{10} = 256\text{Kb}$

6) Di un sistema paginato si sa che lo spazio di indirizzamento logico è grande 1 Gb e un frame è grande 0x800 byte. Si sa inoltre che la PT più grande del sistema occupa 1 Mb. Quanto può essere grande, al massimo, lo spazio di indirizzamento fisico del sistema?

**SOLUZIONE:**

- Spazio logico =  $2^{30}$  byte, quindi usiamo 30 bit per l'indirizzo
- Offset =  $0x800 = 8 * 16^2 = 2^3 * 2^{4*2} = 2^{3+8} = 2^{11}$ , 11 bit per l'offset
- Quindi abbiamo  $2^{30} / 2^{11} = 2^{19}$  pagine indirizzabili quindi  $2^{19}$  entry
- La PT più grande occupa 1 Mb =  $2^{20}$  byte
- Se la PT più grande occupa  $2^{20}$  byte e deve memorizzare  $2^{19}$  entry significa che, per essere grande al massimo  $2^{20}$  ogni entry deve essere grande al massimo 2 byte, perché  $2^{20} = 2^{19} * 2$
- Quindi, dato che con 2 byte possiamo numerare al massimo  $2^{16}$  frame grandi  $2^{11}$  byte, la dimensione massima dello spazio fisico è:  $2^{16} * 2^{11} = 2^{27} = 2^7 * 2^{20} = 128\text{Mb}$