



**Università degli Studi di Verona**  
**Dipartimento di Informatica**  
**A.A. 2018-2019**

## **RIASSUNTO DEL CORSO DI “SISTEMI OPERATIVI”**

Created by *Davide Zampieri*  
Based on slides provided by *prof. Graziano Pravadelli*

❖ **Indice degli argomenti**

- *Introduzione (da pagina 3):*
  - Ruolo del sistema operativo e sua evoluzione
  - Elementi architetturali
  - Struttura e funzioni di un sistema operativo
- *Gestione dei processi (da pagina 10):*
  - Stati dei processi, cambiamento di contesto, creazione e terminazione di processi
  - Thread a livello utente e a livello kernel
- *Scheduling (da pagina 16):*
  - Scheduling a lungo, medio e breve termine
  - Scheduling con prelazione
  - Algoritmi di scheduling (FCFS, SJF, a priorità, HRRN, RR)
  - Valutazione degli algoritmi (modelli deterministici e probabilistici, simulazione)
- *Sincronizzazione fra processi (da pagina 23):*
  - Sezione critica (coerenza nei dati condivisi)
  - Soluzioni software (algoritmo del fornaio) e hardware (test and set, swap)
  - Costrutti per la sincronizzazione (semafori, monitor)
  - Problemi classici (produttore/consumatore, lettori/scrittore, filosofi)
- *Deadlock (da pagina 32):*
  - Condizioni per l'innesto di un deadlock
  - Rappresentazione dello stato di un sistema con grafi di allocazione
  - Tecniche di prevenzione, rilevazione e ripristino
  - Algoritmo del banchiere
- *Memoria primaria, virtuale e secondaria (da pagina 38):*
  - Schemi di gestione della memoria primaria (allocazione contigua, paginazione, segmentazione)
  - Algoritmi di sostituzione delle pagine (FIFO, ideale, LRU, approssimazioni di LRU)
  - Struttura logica e fisica dei dischi
  - Strutture RAID
- *File system (da pagina 59):*
  - Concetto di file, attributi e operazioni relative
  - Concetto di directory e relativa struttura
  - Struttura e montaggio di un file system
  - Metodi di allocazione dello spazio su disco (contiguo, concatenato, indicizzato)
  - Gestione dello spazio libero su disco
- *Sistema di I/O (da pagina 67):*
  - Hardware per I/O
  - Tecniche di I/O (polling, interrupt, DMA)
  - Device driver e interfaccia verso le applicazioni
- *Esempi di domande a crocette e domande sulla teoria (da pagina 69)*

# DEFINIZIONE E STORIA DEI SISTEMI OPERATIVI

## Che cos'è un sistema operativo?

Il sistema operativo è un *insieme di programmi*, il cui compito è quello di fare da intermediario tra hardware e utente per *semplificare l'utilizzo del computer e rendere efficiente l'uso dell'hardware stesso*. Il sistema operativo è anche *l'ambiente che coordina l'utilizzo dell'hardware da parte dei programmi*: in questo modo riesce ad *evitare i conflitti di allocazione delle risorse hardware e software*. Riassumendo, il sistema operativo è un *gestore di risorse* e un *programma di controllo*.

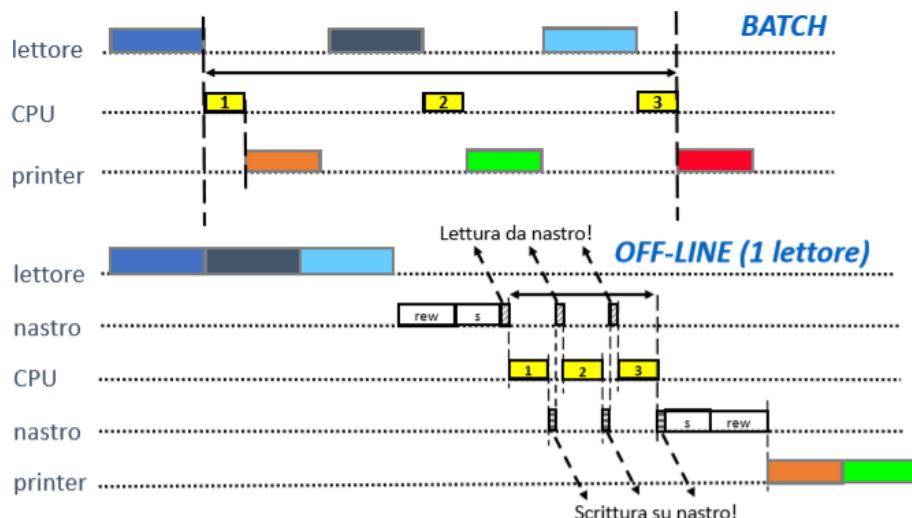
## Obiettivi di un sistema operativo.

Gli obiettivi di un sistema operativo sono *astrazione* (cioè *semplificare l'uso del sistema*) ed *efficienza*, i quali però risultano in contrasto tra di loro.

## Storia dei sistemi operativi.

La storia dei sistemi operativi si può suddividere in *4 generazioni*. Il passaggio da una generazione all'altra si ha quando vi è un sensibile *aumento dell'utilizzo del processore*.

- **1<sup>a</sup> generazione (1946-1955)**: è caratterizzata dall'uso di *device driver*, ossia programmi di interazione con *periferiche* quali lettori e perforatori di schede; i primi software erano semplicemente *library* di funzioni comuni.
- **2<sup>a</sup> generazione (1955-1965)**: si introducono i *transistor*; si separano i ruoli di *operatore e programmatore*; si introduce il *batching*, ossia il *raggruppamento di programmi simili nell'esecuzione*; si ha il primo vero esempio di sistema operativo grazie al *monitor residente*, ossia un gestore perennemente caricato in memoria che contiene i driver per i dispositivi di I/O e che ha il compito di interpretare le schede di controllo, oltre ad occuparsi del *job sequencing* (ovvero passare da un programma all'altro); si introduce l'*off-line processing*, ovvero si usano i *nastri magnetici* per ridurre i tempi morti nell'elaborazione della CPU dovuti alla scarsa velocità delle operazioni di I/O (il vero vantaggio si aveva nel caso di più lettori di nastro).



## Sovrapposizione di CPU e I/O.

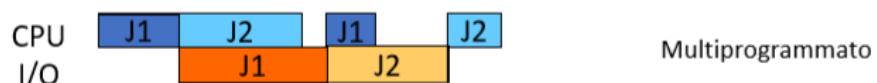
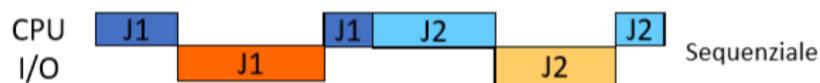
I meccanismi per la sovrapposizione di CPU e I/O sono: il *polling* (che prevede un'*interrogazione continua del dispositivo*); e un *meccanismo asincrono* (che si basa sui concetti di *interrupt* e *DMA*). Quando la CPU riceve un interrupt: interrompe l'istruzione corrente salvandone lo stato, "serve" l'interruzione, riprende

l'istruzione interrotta. Nel caso di dispositivi veloci, però, gli *interrupt* sono molto frequenti e questo causa *inefficienza*, perciò si ricorre al DMA, ossia uno *specifco controllore hardware* che si occupa del trasferimento dei blocchi di dati tra I/O e memoria senza interessare la CPU (ciò comporta *un solo interrupt per ogni blocco di dati*).

### Buffering e Spooling.

Il Buffering prevede la sovrapposizione di CPU e I/O dello stesso job ed è utile quando la velocità dell'I/O e della CPU sono simili. Nella realtà però, i dispositivi di I/O sono più lenti della CPU, perciò si ricorre allo Spooling, che prevede la sovrapposizione di CPU e I/O di job diversi ed è possibile grazie all'*accesso casuale dei dischi* (ossia quando il disco viene utilizzato come un *grande buffer unico per tutti i job*).

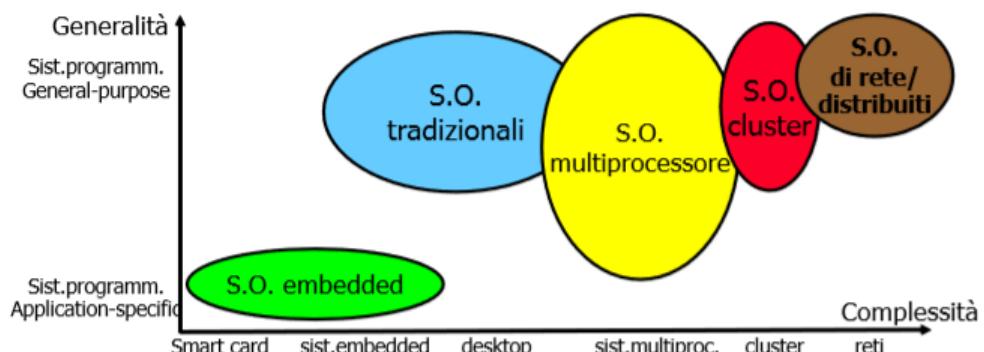
- **3<sup>a</sup> generazione (1965-1980)**: si introduce il *time sharing (multitasking)* che *migliora l'interattività* e sancisce la *nascita dei sistemi moderni* (ossia quelli dotati di tastiera, monitor e accesso a dati e programmi); si introducono la *multiprogrammazione* e i *circuiti integrati* per poter dedicare le fasi di attesa (I/O) all'esecuzione di un nuovo job.



### Protezione.

Nei sistemi originari non si poneva il problema della condivisione, ma ora l'esecuzione di un programma può influenzare l'esecuzione degli altri. Per questo si ricorre a tre tipi di protezione: *protezione dell'I/O*, ossia impedire a programmi diversi di usare I/O contemporaneamente (si realizza con la presenza di *user e supervisor* e rendendo le operazioni di I/O privilegiate, cioè eseguibili solo dalle *system call*); *protezione della memoria*, ossia impedire ad un programma di leggere/scrivere in una zona di memoria che non gli appartiene (si realizza con la protezione dello spazio dei vari processi, cioè associando con istruzioni privilegiate dei *registri limite a ogni processo*); *protezione della CPU*, ossia avere la garanzia che il S.O. mantenga il controllo del sistema (si realizza associando un *timer ad ogni job* alla scadenza del quale il controllo ritorna al S.O.).

- **4<sup>a</sup> generazione (1980-????)**: è caratterizzata dalla nascita di *diverse tipologie di S.O.* (per PC e workstation, di rete, distribuiti, real-time, embedded).



# COMPONENTI DI UN SISTEMA OPERATIVO

## **Gestione dei processi.**

Un processo è un *programma* che *necessita di risorse* e viene eseguito *in modo sequenziale* (cioè un'istruzione alla volta). Il sistema operativo è responsabile della *creazione e distruzione* dei processi, della *sospensione e riesumazione* dei processi e della *sincronizzazione e comunicazione* tra processi.

## **Gestione della memoria primaria.**

La memoria primaria (RAM, cache) *conserva i dati condivisi tra CPU e I/O*. Perciò un *programma*, per poter essere eseguito, *deve essere caricato in memoria*. Il sistema operativo si occupa di: gestire lo spazio di memoria; decidere quale processo caricare in memoria; gestire allocazione e rilascio dello spazio di memoria.

## **Gestione della memoria secondaria.**

La memoria primaria è volatile e piccola, perciò è indispensabile avere una memoria secondaria (HDD, SSD) per poter *Mantenere grandi quantità di dati in modo permanente*. Il sistema operativo si occupa di: gestire lo spazio libero sulla memoria di massa; allocare lo spazio sulla memoria di massa; ordinare gli accessi ai dispositivi.

## **Gestione dell'I/O.**

Il sistema operativo nasconde all'utente le specifiche caratteristiche dei dispositivi di I/O. Il sistema di I/O è una *generica interfaccia verso i device driver* (specifici per ogni dispositivo) che *accumula gli accessi ai dispositivi* (buffering).

## **Gestione dei file.**

I file sono una raccolta di *informazioni correlate memorizzate su supporti fisici* diversi controllati da driver con caratteristiche diverse. Il sistema operativo si occupa di: *creare, cancellare e modificare* file e directory; *fornire primitive* per la gestione di file e directory; *gestire la corrispondenza tra file e spazio fisico* su memoria di massa; salvare informazioni a scopo di *backup*.

## **Protezione.**

La protezione è un meccanismo usato per *controllare l'accesso alle risorse* da parte di utenti e processi. Il sistema operativo si occupa di: *definire accessi autorizzati* e non; *definire i controlli* per gli accessi; fornire strumenti per *verificare le politiche* di accesso.

## **Rete (sistemi distribuiti).**

Un sistema distribuito è una *collezione di risorse di calcolo connesse tramite una rete* che non condividono né la memoria né un clock. Il sistema operativo è responsabile della *gestione in rete delle varie componenti* (processi distribuiti, memoria distribuita, file system distribuito).

## **Interprete dei comandi (Shell).**

La Shell è un *programma che legge e interpreta comandi* (istruzioni di controllo), i quali permettono di creare e gestire processi e di gestire la memoria, l'I/O, le protezioni e la rete.

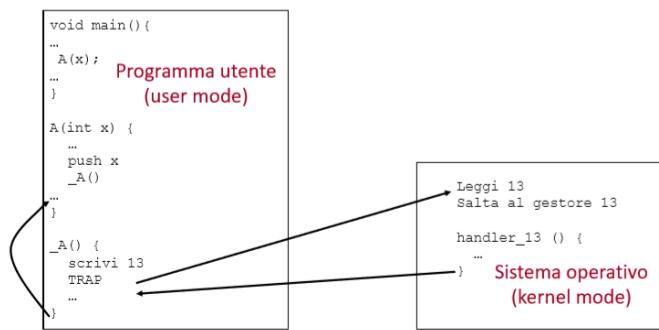
## **System Call.**

Le System Call sono l'*interfaccia tra i processi e il sistema operativo*. Ci sono diverse opzioni per la comunicazione tra sistema operativo e processo:

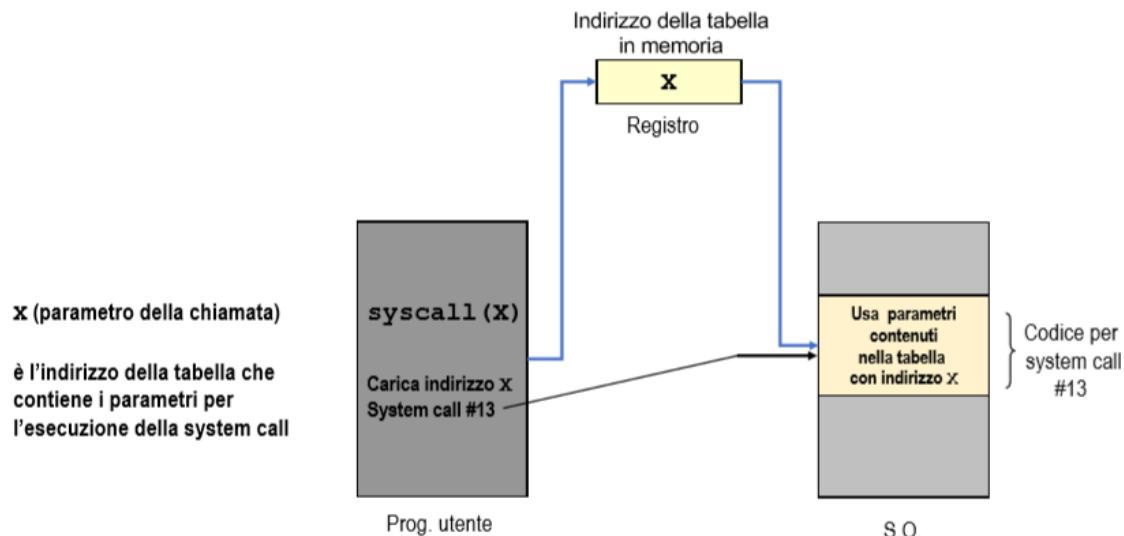
- Passare i parametri della System Call tramite i *registri*.

- Passare i parametri della System Call tramite lo *Stack* del programma.

- Chiamata alla funzione di libreria A (x)
- Parametro x nello stack
- Invocazione della vera system call \_A corrispondente ad A
- \_A mette il numero di system call in un punto noto al S.O.
- \_A esegue una TRAP (interruzione non mascherabile)  
effetto: passaggio da *user mode* a *kernel mode*
- Inizia l'esecuzione ad un indirizzo fisso (gestore interrupt)
- Il S.O., in base al numero di system call, smista la chiamata al corretto gestore che viene eseguito
- Una volta terminato, il controllo viene restituito al programma di partenza (funzione di libreria A ( ) )



- Memorizzare i parametri in una *tavella in memoria* e passare l'indirizzo della tabella in un registro o nello Stack.



## Programmi di sistema.

Se le System Call sono una *vista lato programma* delle operazioni di un sistema, i programmi di sistema sono una *vista lato utente* delle operazioni di un sistema. I programmi di sistema si occupano di:

- Gestione e manipolazione dei file.
- Formattazione di documenti.
- Fornire informazioni sullo stato del sistema (data, memoria libera).
- Fornire strumenti di supporto alla programmazione (compilatori, assemblatori).
- Interpretare i comandi (Shell).

## I servizi di un sistema operativo.

I principali servizi offerti da un sistema operativo sono:

- Esecuzione di programmi.
- Operazioni di I/O.
- Manipolazione del file system.
- Comunicazione (memoria condivisa, scambio di messaggi).
- Rilevamento di errori logici e fisici.
- Allocazione e contabilizzazione delle risorse.
- Protezione e sicurezza.

# ARCHITETTURA DI UN SISTEMA OPERATIVO

## Struttura di un sistema operativo.

Esistono vari tipi di strutture dei sistemi operativi:

- 1) Sistemi monolitici.
- 2) Sistemi a struttura semplice.
- 3) Sistemi a livelli.
- 4) Virtual machine.
- 5) Sistemi basati su kernel.
- 6) Sistemi client/server.

### 1) Sistemi monolitici.

Nei sistemi monolitici non c'è *gerarchia*, ciò implica...

- la presenza di un *unico strato software* tra utente e hardware;
- che le *componenti* si trovano tutte *allo stesso livello*;
- che ci sia un *insieme di procedure* che possono chiamarsi *vicendevolmente*.

Gli svantaggi di questi sistemi sono che il *codice dipendente* dall'architettura hardware viene *distribuito su tutto il sistema operativo*, e che *test e debugging* diventano *difficili* da effettuare.

### 2) Sistemi a struttura semplice.

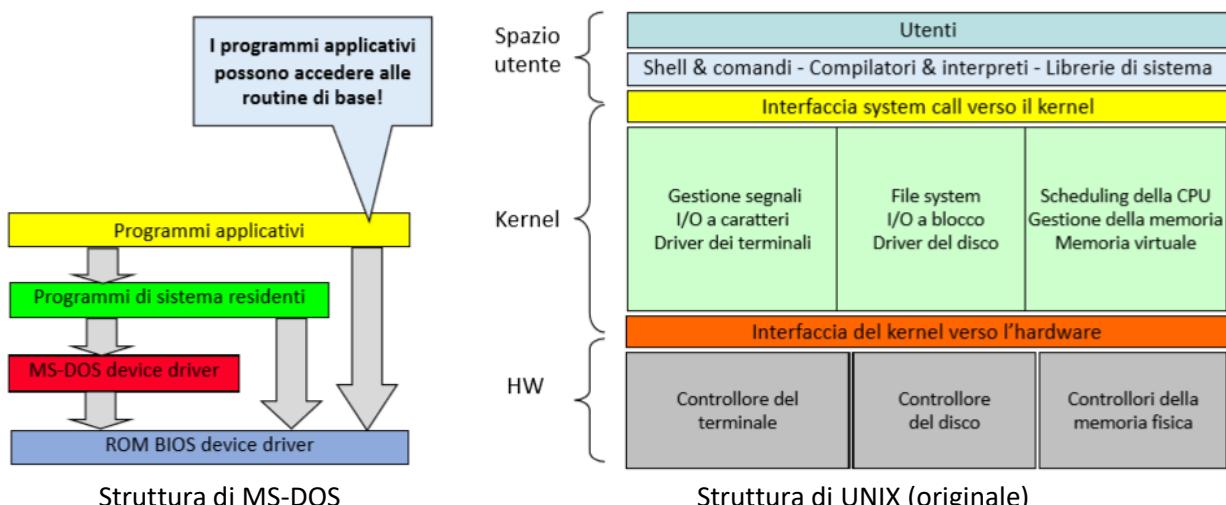
Nei sistemi a struttura semplice c'è una *minima organizzazione gerarchica*. Questo permette di *definire i livelli* della gerarchia *in maniera molto flessibile*. Inoltre, la strutturazione mira a ridurre i costi di sviluppo e manutenzione. Esempi di sistemi semplici: MS-DOS, UNIX originale.

#### MS-DOS.

Era pensato per fornire il *maggior numero di funzionalità nel minimo spazio*. Inoltre, non era suddiviso in moduli e non prevedeva il dual-mode (in quanto il processore Intel 8088 non lo forniva). Aveva una *struttura minima*, ma le interfacce e i livelli di funzionalità non erano ben definiti.

#### UNIX originale.

Aveva una *struttura base limitata* (a causa delle limitate funzionalità hardware) caratterizzata dalla presenza del *kernel*, ovvero tutto ciò che sta tra il livello dell'interfaccia delle system call e l'hardware. Il kernel forniva: file system, scheduling della CPU, gestione della memoria, etc.



### 3) Sistemi a livelli.

Nei sistemi a livelli, i servizi sono *organizzati per livelli gerarchici*: al livello più alto troviamo l'interfaccia utente, mentre a quello più basso troviamo l'hardware. Ciascun livello può usare solo le funzioni fornite dai livelli inferiori, e inoltre definisce precisamente il tipo di servizio e l'interfaccia verso il livello superiore (nascondendone l'implementazione). Esempi di sistemi a livelli: THE, MULTICS, OS/2.

#### Vantaggi e svantaggi.

- + *Modularità*, che facilita la messa a punto e la manutenzione del sistema.
- *Difficoltà nel definire* in maniera appropriata *gli strati*.
- *Minor efficienza*, dovuta al fatto che ogni strato aggiunge overhead alle system call.
- *Minor portabilità*, dovuta al fatto che le funzionalità dipendenti dall'architettura sono sparse sui vari livelli.

#### THE.

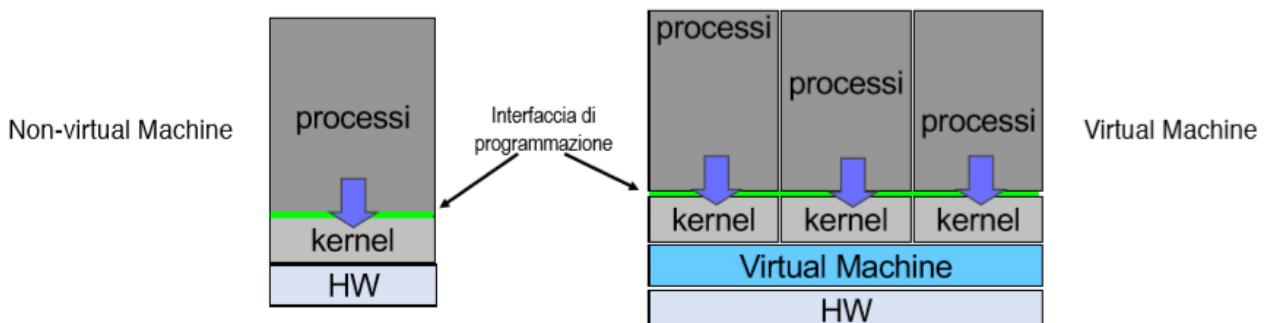
Ideato da Dijkstra nel 1968, era un sistema operativo accademico per sistemi batch. THE è il *primo esempio di sistema a livelli*, in quanto si trattava di un *insieme di processi cooperanti* (sincronizzati tramite semafori).



### 4) Virtual machine.

I sistemi VM sono un'estremizzazione dell'approccio a livelli e sono pensati per offrire un sistema di time-sharing "multiplio". Il concetto chiave dei sistemi VM è la *separazione di multiprogrammazione e presentazione*. Nei sistemi VM anche il sistema operativo viene trattato come hardware, ovvero:

- Una virtual machine fornisce un'interfaccia identica all'HW sottostante.
- Il sistema operativo si esegue sopra la virtual machine.
- La virtual machine dà l'illusione di processi multipli (ciascuno in esecuzione sul proprio hardware).



### **Vantaggi e svantaggi.**

- + *Protezione completa del sistema*, in quanto ogni virtual machine è isolata dalle altre.
- + *Presenza di più sistemi operativi* sulla stessa macchina.
- + *Ottimizzazione delle risorse*, in quanto la stessa macchina può ospitare ciò che senza virtual machine doveva essere eseguito su macchine separate.
- + *Buona portabilità*.
- *Problemi di prestazioni*.
- *Necessità di gestire il dual mode virtuale*, in quanto il sistema di gestione delle virtual machine esegue in kernel mode, mentre la virtual machine esegue in user mode.
- *Mancanza di condivisione*, in quanto ogni virtual machine è isolata dalle altre (risolvibile condividendo un volume del file system e definendo via software una rete virtuale tra le virtual machine).

### **Virtual machine moderne.**

Il concetto di virtual machine, anche se in contesti diversi e con certi vincoli, viene usato ancora oggi per:

- Eseguire programmi MS-DOS in Windows (tramite emulazione di 8086).
- Eseguire in “contemporanea” Linux e Windows (tramite VMware o VirtualBox).
- Eseguire programmi Java usando la JVM (Java Virtual Machine).

### **5) Sistemi basati su kernel.**

I sistemi basati su kernel sono caratterizzati da *due soli livelli*: servizi *kernel* e servizi *non-kernel*. Le implementazioni moderne di UNIX sono sistemi operativi basati su kernel.

### **Vantaggi e svantaggi.**

- + *Gli stessi dei sistemi a livelli*, pur senza avere troppi livelli.
- *Non è così generale* come un sistema a livelli.
- *Mancanza di regole organizzative* per le parti del sistema operativo *fuori dal kernel*.
- *Tendenza a diventare monolitico* a causa della complessità del kernel.

### **6) Sistemi client/server.**

Nei sistemi client/server, *tutti i servizi* del sistema operativo sono realizzati come processi utente (client). Il client chiama un processo servitore (server) per usufruire di un servizio e il server, dopo l'esecuzione, restituisce il risultato al client. In questo caso, il kernel si occupa solo della gestione della comunicazione tra client e server. Il modello client/server si presta bene per i sistemi operativi distribuiti, mentre i sistemi operativi moderni spesso realizzano in questo modo alcuni servizi.

### **Implementazione di un sistema operativo.**

I sistemi operativi sono *tradizionalmente* scritti in *linguaggio assembler*, ma i più *moderni* possono essere scritti anche in *linguaggi ad alto livello* (C/C++). Così facendo si favorisce: la rapidità dell'implementazione, la compattezza, la comprensione, la manutenzione e la portabilità.

# PROCESSI E THREAD

## ❖ Concetto di processo

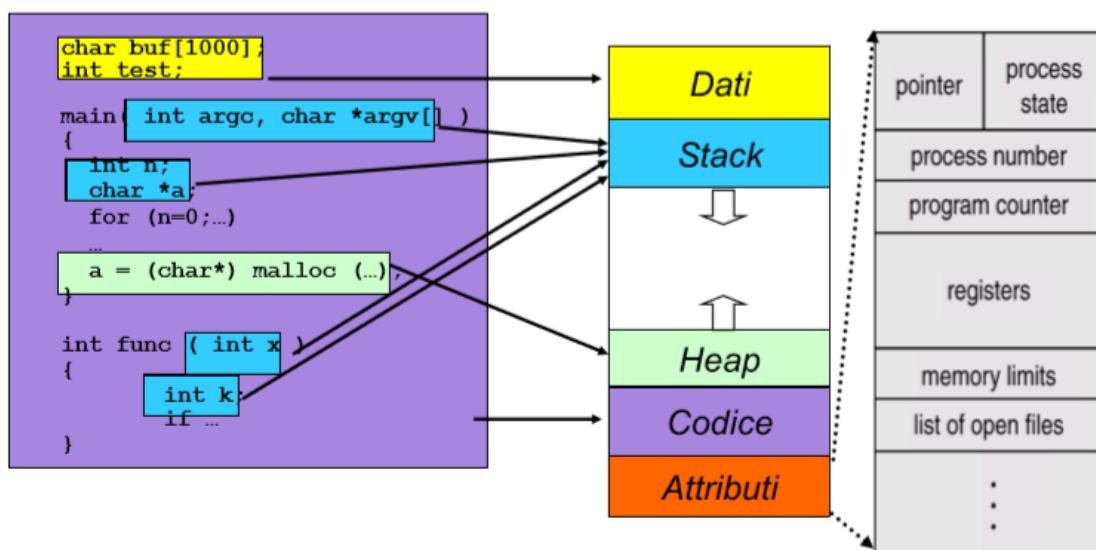
### **Programma e processo.**

Un *processo* (concetto dinamico) è un'*istanza di un programma* (concetto statico) *in esecuzione*. Il processo viene eseguito in maniera sequenziale (un'istruzione alla volta) ma, in un sistema multi-programmato, i processi evolvono in modo concorrente. Inoltre, va ricordato che: le risorse (fisiche e logiche) sono limitate, e il sistema operativo stesso consiste di più processi.

### **Immagine in memoria.**

Un processo consiste di varie parti:

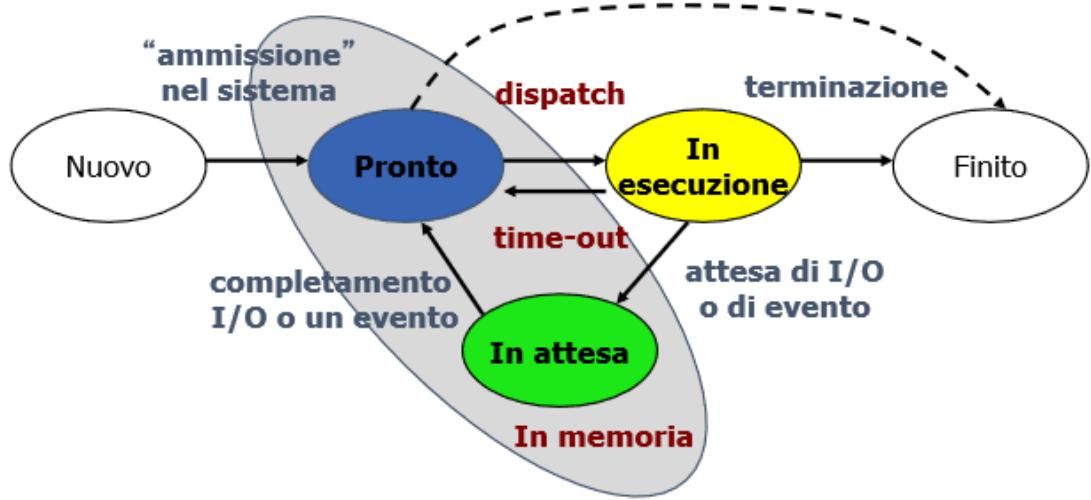
- *Codice*, che contiene le istruzioni del programma.
- *Dati*, che contiene le variabili globali.
- *Stack*, che contiene le chiamate a funzione, i parametri e le variabili locali.
- *Heap*, che contiene le varabili allocate nella memoria dinamica.
- *Attributi* o PCB (Process Control Block), che rappresenta il processo e contiene a sua volta:
  - Stato del processo, program counter e valori dei registri.
  - Informazioni sulla memoria (registri limite, tabella pagine).
  - Informazioni sullo stato dell'I/O (richieste pendenti, file).
  - Informazioni sull'utilizzo del sistema (CPU).
  - Informazioni di scheduling (priorità).



## ❖ Stati di un processo

### **Diagramma degli stati.**

Durante la sua esecuzione, un *processo evolve attraverso diversi stati*. Un possibile diagramma degli stati, che cambia a seconda del sistema operativo, può essere il seguente:



### Scheduling.

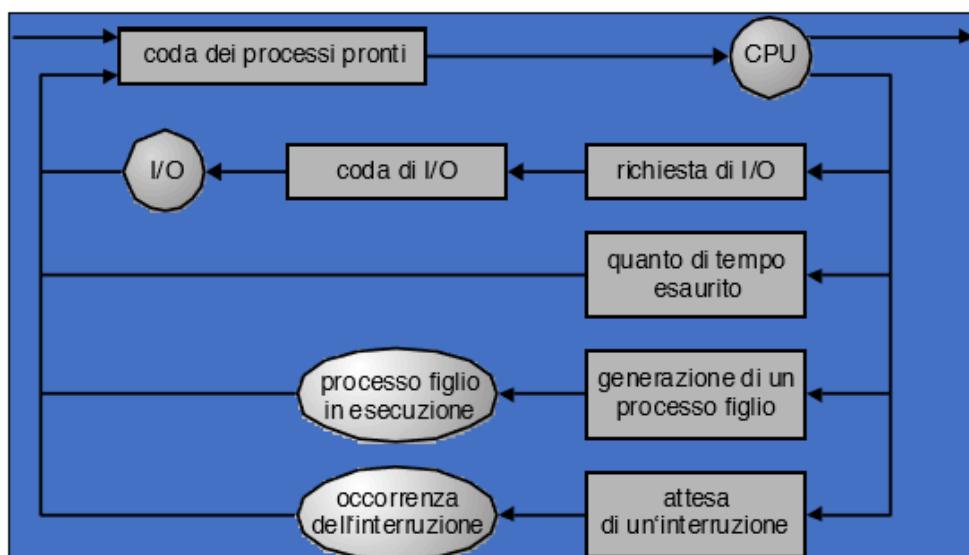
Lo scheduling è il meccanismo di *selezione del processo da eseguire* nella CPU, che deve garantire:

- *Multiprogrammazione*, per massimizzare l'uso della CPU nel caso di più processi in memoria.
- *Time-sharing*, per commutare frequentemente la CPU tra i processi in modo che ognuno creda di avere la CPU tutta per sé.

### Code di scheduling.

Ogni processo è inserito in una serie di code, che possono essere: *ready queue*, ovvero la coda dei processi pronti per l'esecuzione; oppure *coda di un dispositivo*, ovvero la coda dei processi in attesa che il dispositivo si liberi. All'inizio, il processo rimane nella ready queue fino a quando non viene selezionato per essere eseguito (*dispatched*). Poi, durante l'*esecuzione*, può succedere che:

- Il processo *necessita di I/O* e viene inserito in una coda di un dispositivo.
- Il processo *termina il suo quanto di tempo* e viene forzatamente reinserito dalla CPU nella ready queue.
- Il processo crea un figlio e ne attende la terminazione.
- Il processo si mette in attesa di un evento.

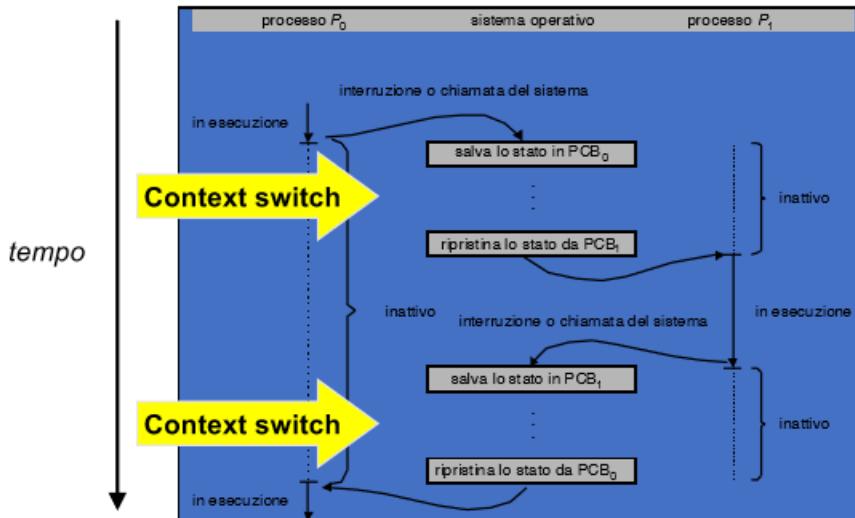


## Dispatch.

Il dispatcher ha il compito di effettuare il *cambio di contesto* (context switch), che consiste nel passaggio della CPU ad un nuovo processo. Le fasi del cambio di contesto sono:

- *Salvataggio dello stato* (PCB) del vecchio processo e *caricamento dello stato* del nuovo processo.
- *Passaggio alla modalità utente*, in quanto all'inizio della fase di dispatch il sistema si trova ancora in modalità kernel.
- *Salto all'istruzione da eseguire* del processo appena arrivato nella CPU.

La *durata* del cambio di contesto (che dipende dall'architettura utilizzata) è *puro sovraccarico*, ovvero il sistema non compie alcun lavoro utile durante la commutazione.



## ❖ Operazioni sui processi

### Creazione di un processo.

Un processo può creare un *figlio*, il quale *ottiene le risorse* dal sistema operativo o dal padre *per spartizione o condivisione*. Il processo figlio può essere eseguito in maniera: *sincrona*, quando il padre attende la terminazione dei figli; o *asincrona*, con l'evoluzione parallela di padre e figlio. Nei sistemi UNIX la creazione di un processo può avvenire tramite tre diverse system call:

- *Fork*, che crea un figlio che è un duplicato esatto del padre.
- *Exec*, che carica sul figlio un programma diverso da quello del padre.
- *Wait*, che si usa per l'esecuzione sincrona tra padre e figlio.

```
/*
 * Creazione di un nuovo processo tramite fork (figlio identico al padre):
 * - il processo padre riceverà il PID del figlio
 * - il processo figlio riceverà 0 (solo per sapere di essere il figlio)
 * - se chiama getpid(), gli verrà restituito ciò che c'è nella variabile pid del padre
 */
int pid = fork();
if (pid < 0) {
    // se il figlio non viene creato, fork() restituisce un numero negativo
    fprintf(stderr, "Errore di creazione");
    exit(-1);
} else if (pid == 0) {
    // il PC del figlio si trova esattamente all'istruzione fork()
    // quindi essendo pid = 0, in questo blocco andrà messo il codice del figlio
    /*
     * Chiamata di exec (il codice del figlio viene cambiato):
     */
    execvp("/bin/ls", "ls", NULL);
} else {
    // anche il PC del padre si trova esattamente all'istruzione fork()
    // quindi essendo pid > 0, in questo blocco andrà messo il codice del padre
    /*
     * Chiamata di wait (il padre attende il figlio):
     */
    wait(NULL);
    printf("Figlio ha terminato\n");
    exit(0);
}
```

## Terminazione di un processo.

La terminazione di un processo può avvenire quando il *processo* stesso:

- *Finisce la sua esecuzione.*
- *Viene terminato forzatamente dal padre* (per eccesso nell'uso delle risorse, quando il compito richiesto al figlio non è più necessario, o quando il padre termina e il sistema operativo non permette ai figli di sopravvivere al padre).
- *Viene terminato forzatamente dal sistema operativo* (quando l'utente chiude un'applicazione, o quando si verificano errori aritmetici, di protezione o di memoria).

## Relazioni tra processi.

I processi possono essere indipendenti o cooperanti. Nei processi *indipendenti*: l'esecuzione è deterministica (dipende solo dal proprio input) e riproducibile; il processo non influenza né viene influenzato da altri processi; non c'è nessuna condivisione dei dati con altri processi. Nei processi *cooperanti*: il processo influenza e può essere influenzato da altri processi; l'esecuzione non è deterministica e non è riproducibile.

Tra i vantaggi dei processi cooperanti si trovano:

- *Condivisione delle informazioni.*
- *Accelerazione del calcolo*, tramite l'esecuzione parallela di sub-task su multiprocessore.
- *Modularità*, con funzioni distinte su vari processi.
- *Convenienza*.

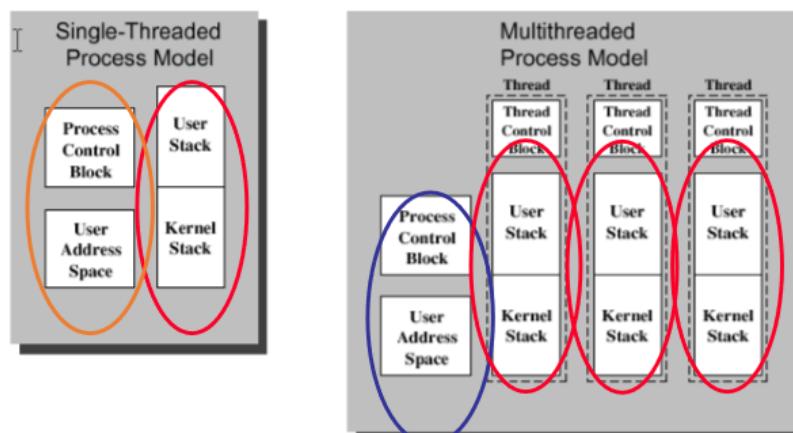
## ❖ Concetto di thread

### Processo e thread.

Un processo unisce due concetti: il *possesso delle risorse* (spazio di memoria, file, I/O), e l'*utilizzo della CPU* (priorità, stato, registri). Queste due caratteristiche sono indipendenti e possono essere considerate separatamente, perciò parleremo di *processo* (come unità minima di possesso delle risorse) e *thread* (come unità minima di utilizzo della CPU). Ad un processo sono associati: *spazio di indirizzamento* e *risorse del sistema*. Ad una singola thread, invece, sono associati: *stato di esecuzione*, *program counter*, insieme di *registri* della CPU e *Stack*. Le *thread* inoltre *condividono* tra loro lo *spazio di indirizzamento*, le *risorse* e lo *stato del processo*.

### Multithreading.

In un sistema operativo classico, ad un processo corrisponde una thread. Nei sistemi multithread invece, si ha la possibilità di supportare più thread per ogni singolo processo. Di conseguenza si avrà la *separazione tra flusso di esecuzione* (thread) e *spazio di indirizzamento* (processo). Ad esempio, in un processo con thread singola ci sarà un flusso associato ad uno spazio di indirizzamento, mentre in un processo con thread multiple ci saranno più flussi associati ad un singolo spazio di indirizzamento.



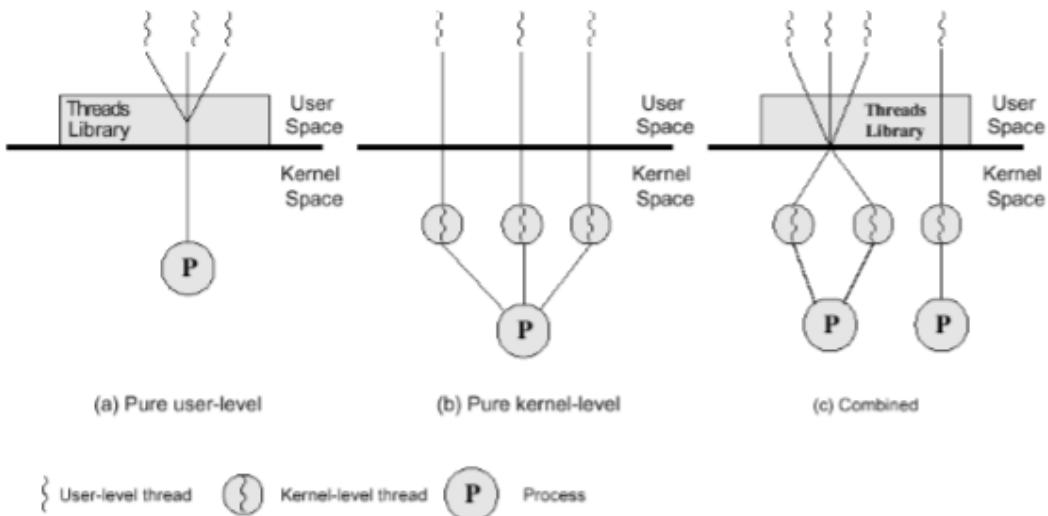
## Vantaggi delle thread.

- *Riduzione del tempo di risposta*: mentre una thread è bloccata (a causa di I/O o di elaborazione lunga), un'altra thread può continuare a interagire con l'utente.
- *Condivisione delle risorse*: le thread di uno stesso processo condividono la memoria senza dover introdurre tecniche esplicite di condivisione (come invece avviene per i processi), e quindi la sincronizzazione e la comunicazione è agevolata.
- *Economia*: la creazione e la terminazione delle thread e il context switch fra thread è più veloce rispetto ai processi.
- *Scalabilità*: il multithreading aumenta il parallelismo se l'esecuzione avviene su multiprocessore (una thread in esecuzione su ogni processore).

## Stati di una thread e implementazioni.

Gli stati di una thread sono *come quelli di un processo* (pronta, in esecuzione, in attesa), ma lo stato del processo può, in generale, non coincidere con lo stato della thread. Una thread in attesa deve bloccare l'intero processo? Dipende dall'*implementazione*, che può essere:

- *User-level*, ossia quando la gestione è affidata alle applicazioni ed il kernel ignora l'esistenza delle thread.
- *Kernel-level*, ossia quando la gestione è affidata al kernel e le applicazioni usano le thread tramite le system call.
- *Combinata*.



## User-level thread: vantaggi e svantaggi.

- + *Efficienza*, in quanto non è necessario passare alla modalità kernel per usare le thread.
- + *Scheduling variabile* da applicazione ad applicazione.
- + *Portabilità*, in quanto le thread possono girare su qualunque sistema operativo senza modificare il kernel.
- *Il blocco di una thread blocca il processo* (superabile con accorgimenti specifici come I/O non bloccante).
- *Non è possibile sfruttare il multiprocessore*, in quanto c'è una sola thread in esecuzione per ogni processo.

## Kernel-level thread: vantaggi e svantaggi.

- + *Il blocco di una thread non blocca il processo*, in quanto lo scheduling è a livello di thread.
- + Si hanno *più thread* dello stesso processo *in esecuzione contemporanea* su CPU diverse.
- + Anche le *funzioni del sistema operativo* stessa possono essere *multithread*.
- *Scarsa efficienza*, in quanto il passaggio fra thread implica un passaggio attraverso il kernel.

## ❖ Gestione dei processi del sistema operativo

### Esecuzione del kernel.

Il *sistema operativo* è un *programma* a tutti gli effetti. Ma il sistema operativo in esecuzione può essere considerato un processo? Ci sono diverse opzioni: kernel eseguito separatamente; kernel eseguito all'interno di un processo utente; kernel eseguito come processo.

### Kernel separato.

È tipico dei primi sistemi operativi, ed è quando il *kernel* esegue al di fuori di ogni processo. Il sistema operativo possiede uno *spazio riservato in memoria*, prende il controllo del sistema ed è *sempre in esecuzione in modo privilegiato*. Il concetto di processo viene applicato solo ai processi utente.

### Kernel in processi utente.

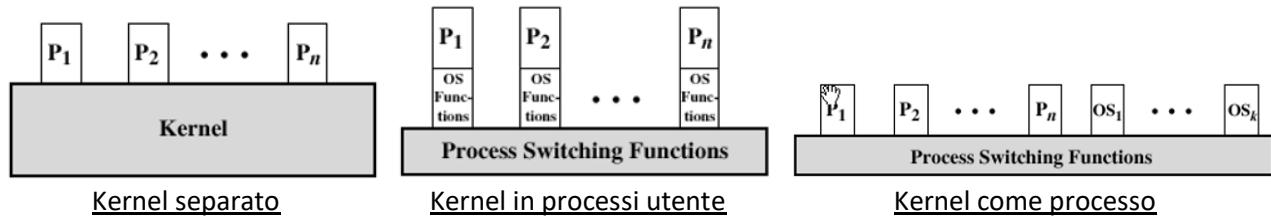
I *servizi* del sistema operativo sono *procedure chiamabili dai programmi* utente e sono accessibili in modalità protetta (kernel mode). L'immagine dei processi prevede:

- *Kernel Stack*, per gestire il funzionamento di un processo in modalità protetta (chiamata a funzione).
- *Codice e dati del sistema operativo condivisi* tra i processi utente.

In caso di interrupt durante l'esecuzione di un processo utente, serve solo un mode switch (cioè quando il sistema passa da user mode a kernel mode e viene eseguita la parte di codice relativa al sistema operativo), che è più leggero rispetto al context switch. Dopo il completamento del suo lavoro, il sistema operativo può decidere di riattivare lo stesso processo utente (mode switch) o un altro (context switch).

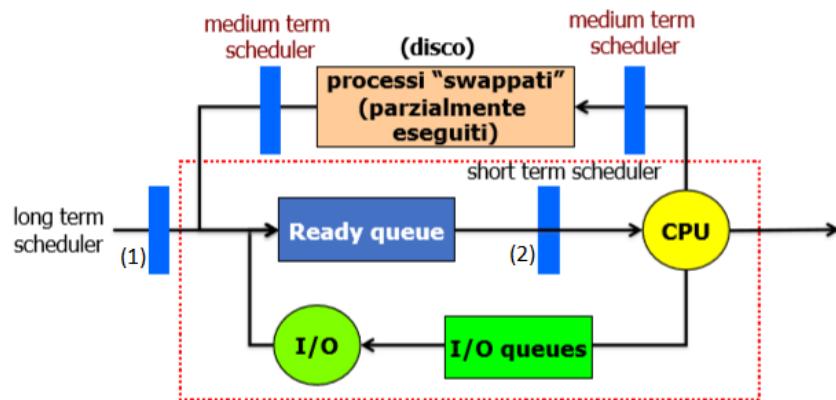
### Kernel come processo.

I *servizi* del sistema operativo sono *processi individuali* eseguiti in modalità protetta. Tuttavia, una minima parte del sistema operativo deve comunque eseguire al di fuori di tutti i processi (scheduler). Questo approccio è *vantaggioso per sistemi multiprocessore*, dove i processi del sistema operativo possono essere eseguiti su un processore ad hoc.



# SCHEDULING

## ❖ Concetto di scheduling



## Scheduling dei processi.

Lo scheduling è l'assegnazione di attività nel tempo. L'utilizzo della multiprogrammazione impone l'esistenza di una strategia per regolamentare l'ammissione dei processi nel sistema (1) e l'ammissione dei processi all'esecuzione (2).

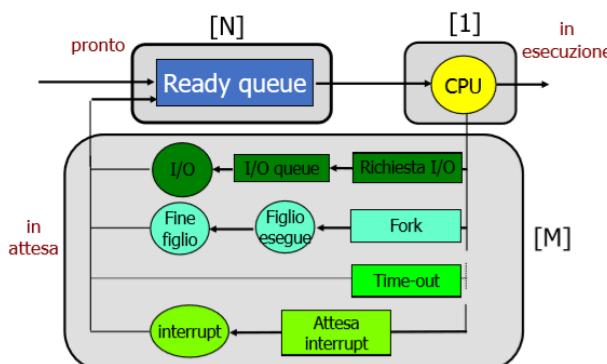
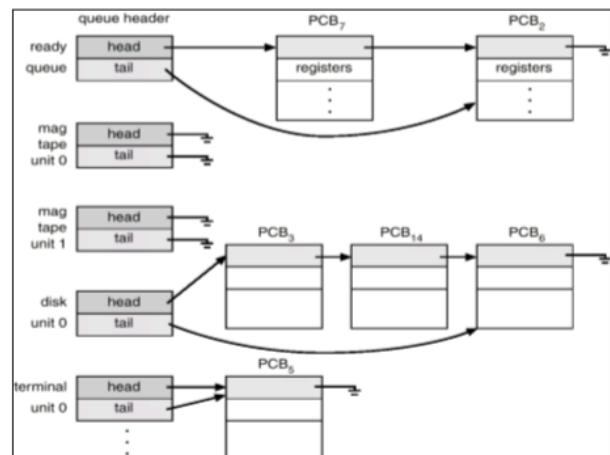


Diagramma di accodamento



Implementazione delle code

## Tipi di scheduler.

La strategia citata precedentemente si chiama *scheduler*. Lo scheduler può essere:

- A lungo termine (job scheduler), che seleziona quali processi devono essere portati dalla memoria alla ready queue.
- A breve termine (CPU scheduler), che seleziona quale processo deve essere eseguito dalla CPU.

## Caratteristiche degli scheduler.

Lo *scheduler a breve termine* è invocato spesso, perciò deve essere *veloce*. Lo *scheduler a lungo termine*, invece, è invocato più raramente, perciò può essere *lento* o addirittura assente (infatti è usato principalmente in sistemi con risorse limitate). Lo *scheduler a lungo termine*, inoltre, *controlla il grado di multiprogrammazione* e il mix di processi, ovvero bilancia I/O-bound e CPU-bound. Infine, i sistemi operativi con memoria virtuale prevedono un livello intermedio di *scheduling (a medio termine)*, il quale ha il compito di effettuare momentanee rimozioni forzate (*swapping*) di processi dalla CPU, per ridurre il grado di multiprogrammazione.

## ❖ Scheduling della CPU

### CPU scheduler.

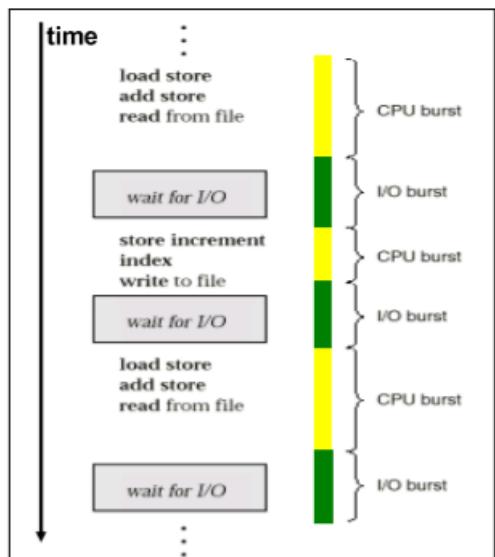
Lo scheduler della CPU è un *modulo del sistema operativo* che *seleziona un processo* tra quelli in memoria pronti per l'esecuzione, e gli alloca la CPU. Data la frequenza di invocazione, lo scheduler è una parte critica del sistema operativo, e quindi c'è la necessità di avere degli algoritmi di scheduling.

### CPU dispatcher.

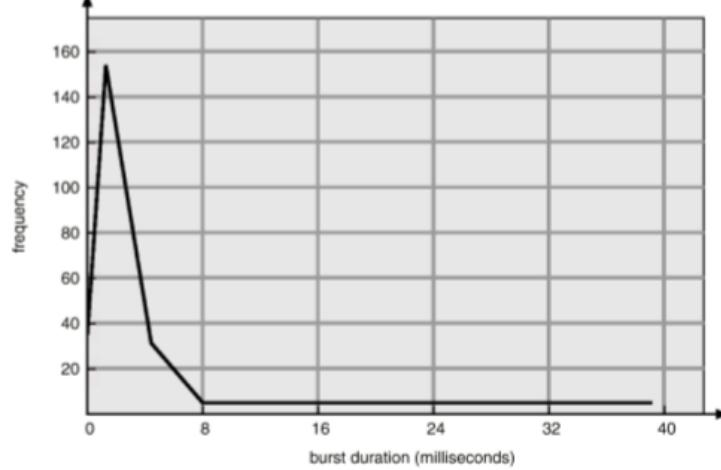
Il dispatcher della CPU è un *modulo del sistema operativo* che *passa il controllo della CPU al processo scelto* dallo scheduler. Le fasi del passaggio sono: switch del contesto, passaggio alla modalità user, salto all'opportuna locazione nel programma per farlo ripartire. La *latenza di dispatch* è il tempo necessario al dispatcher per fermare un processo e farne ripartire un altro. Ovviamente deve essere più bassa possibile.

### Modello astratto del sistema.

L'esecuzione di un processo consiste nell'*alternanza ciclica di un burst* (sequenza) di *CPU* e di uno di *I/O*. I burst si distribuiscono in maniera *esponenziale*, ovvero ci sono *numerosi burst brevi e pochi burst lunghi*.



Modello a cicli di burst CPU-I/O

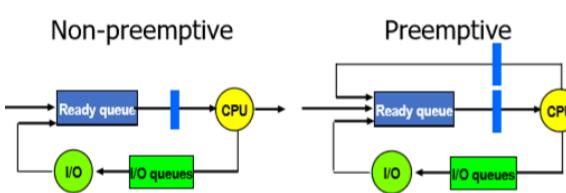


Distribuzione dei CPU burst

### Prelazione.

La prelazione è il *rilascio forzato della CPU*. Quindi lo scheduling può essere:

- Senza prelazione (*non-preemptive*), quando il processo che detiene la CPU non la rilascia fino al termine del burst.
- Con prelazione (*preemptive*), quando il processo che detiene la CPU può essere forzato a rilasciarla prima del termine del burst.



Non-preemptive vs Preemptive



Schema complessivo degli stati di un processo

## ❖ Algoritmi di scheduling

**Metriche di scheduling e criteri di ottimizzazione (+ da massimizzare, – da minimizzare).**

- + Utilizzo della CPU, in quanto l'obiettivo è tenere la CPU occupata il più possibile.
- + Throughput, ovvero il numero di processi completati per unità di tempo.
- Tempo di attesa (waiting time -  $t_w$ ), che è influenzato dall'algoritmo di scheduling e rappresenta la quantità totale di tempo spesa da un processo nella coda di attesa.
- Tempo di completamento (turnaround -  $t_t$ ), ovvero il tempo necessario ad eseguire un particolare processo dal momento della sottomissione al momento del completamento.
- Tempo di risposta (response time -  $t_r$ ), ovvero il tempo trascorso da quando una richiesta è stata sottoposta al sistema fino alla prima risposta del sistema stesso.

### First-Come, First-Served (FCFS).

In questo algoritmo la coda dei processi è una *coda FIFO*, ovvero il primo processo arrivato è il primo ad essere servito. L'implementazione di questo algoritmo è molto *semplice*. Uno svantaggio, invece, è il cosiddetto *effetto convoglio*, che si verifica quando processi brevi si accodano ai processi lunghi precedentemente arrivati.

Esempi			Tempo di attesa medio			
Processo	Tempo di arrivo	CPU burst	Processo	$T_r$	$T_w$	$T_t$
P1	0	24	P1	0	0	24
P2	2	3	P2	22	22	25
P3	4	3	P3	23	23	26

Tempo	0	24	27	30
P1				
P2				
P3				

Processo	Tempo di arrivo	CPU burst	Processo	$T_r$	$T_w$	$T_t$
P1	4	24	P1	2	2	26
P2	0	3	P2	0	0	3
P3	2	3	P3	1	1	4

Tempo	0	3	6	30
P1				
P2				
P3				

$$t_w_{\text{medio}} = (0+22+23)/3 = 15$$

$$t_w_{\text{medio}} = (2+0+1)/3 = 1$$

### Shortest-Job-First (SJF).

Questo algoritmo associa ad ogni processo la lunghezza del prossimo burst di CPU, quindi il processo con il burst di CPU più breve viene selezionato per l'esecuzione. L'algoritmo SJF ha due schemi: *non-preemptive* e *preemptive*. In quest'ultimo caso l'algoritmo si chiama *Shortest-Remaining-Time-First* (SRTF) e prevede che se arriva un nuovo processo con un burst di CPU più breve del tempo che rimane da eseguire al processo in esecuzione, quest'ultimo viene rimosso dalla CPU per fare spazio a quello appena arrivato. SJF è un *ottimo* algoritmo perché offre il *minimo tempo medio di attesa*.

### Calcolo del prossimo burst di CPU.

Si utilizzano le lunghezze dei burst precedenti ( $t_n$ ) come proiezione di quelli futuri ( $\tau_{n+1}$ ). Ma così facendo è possibile fare soltanto una *stima*. La formula utilizzata si chiama *media esponenziale*:  $\tau_{n+1} = \alpha * t_n + (1-\alpha) * \tau_n$  con  $0 < \alpha < 1$ . Se  $\alpha = 0$ , la storia recente non viene usata ( $\tau_{n+1} = \tau_n$ ). Se  $\alpha = 1$ , conta solo l'ultimo burst reale ( $\tau_{n+1} = t_n$ ).

Esempi				Tempo di attesa medio			
Processo	Tempo di arrivo	CPU burst		Processo	T <sub>r</sub>	T <sub>w</sub>	T <sub>t</sub>
P1	0	7		P1	0	0	7
P2	2	4		P2	6	6	10
P3	4	1		P3	3	3	4
P4	5	4		P4	7	7	11

Tempo	0		7	8		12	16
P1							
P2							
P3							
P4							

Processo	Tempo di arrivo	CPU burst		Processo	T <sub>r</sub>	T <sub>w</sub>	T <sub>t</sub>
P1	0	7		P1	0	9	16
P2	2	4		P2	0	1	5
P3	4	1		P3	0	0	1
P4	5	4		P4	2	2	6

Tempo	0	2	4	5	7		11	16
P1								
P2								
P3								
P4								

### Scheduling a priorità.

In questo algoritmo viene associata una priorità ad ogni processo. La *CPU* viene quindi *allocata al processo con la priorità più alta*. Anche questo algoritmo può essere *preemptive* o *non-preemptive*. Inoltre, in Linux esiste un comando (*nice*) che *cambia la priorità*. Anche SJF può essere visto come uno scheduling a priorità (calcolata come 1 / lunghezza del burst successivo). Le *politiche di assegnamento* della priorità possono essere: *interne al S.O.* (limiti di tempo, requisiti di memoria, numero di file aperti); *esterne al S.O.* (importanza del processo, soldi pagati per l'utilizzo del computer). Un *problema* di questo algoritmo è il cosiddetto *starvation*, ovvero il fatto che processi a bassa priorità possano non essere mai eseguiti. Una *soluzione* è rappresentata dall'invecchiamento (*aging*), cioè dall'aumento della priorità col passare del tempo.

Esempio				Tempo di attesa medio			
Proc.	T. di arrivo	Pr.	CPU burst	Processo	T <sub>r</sub>	T <sub>w</sub>	T <sub>t</sub>
P1	1	3	10	P1	5	5	15
P2	0	1	1	P2	0	0	1
P3	2	3	2	P3	14	14	16
P4	0	4	1	P4	18	18	19
P5	1	2	5	P5	0	0	5

Tempo	0	1		6		16	18	19
P1								
P2								
P3								
P4								
P5								

### Highest Response Ratio Next (HRRN).

È un *algoritmo a priorità non-preemptive*. La *priorità* si calcola come  $R = (t_{\text{attesa}} + t_{\text{burst}}) / t_{\text{burst}} = 1 + t_{\text{attesa}} / t_{\text{burst}}$ . La priorità è considerata maggiore per valori di R più alti ed è *dinamica*, cioè dipende anche dal tempo di attesa. Inoltre, la priorità può essere *ricalcolata: al termine di ogni processo; oppure al termine di un processo solo se nel frattempo ne sono arrivati altri*. Questo algoritmo *favorisce* i processi che si

completano in *poco tempo* (come SJF), ma anche quelli che hanno *atteso molto* (superando il “favoritismo” di SJF verso i job corti).

Esempio			Calcolo priorità R (termine processo)				Tempo di attesa medio	
Proc.	T. di arrivo	CPU burst	Proc.	t=0	t=2	t=7	t=8	
P1	1	10	P1	-	1+1/10	1+6/10	1+7/10	
P2	0	2	P2	1	-	-	-	
P3	2	2	P3	-	1+0/2	1+5/2	1+6/2	
P4	2	1	P4	-	1+0/1	1+5/1	-	
P5	1	5	P5	-	1+1/5	-	-	

Processo	T <sub>r</sub>	T <sub>w</sub>	T <sub>t</sub>	Tempo	0	2	7	8	10	19
P1	9	9	19	P1						
P2	0	0	2	P2						
P3	6	6	8	P3						
P4	5	5	6	P4						
P5	1	1	6	P5						

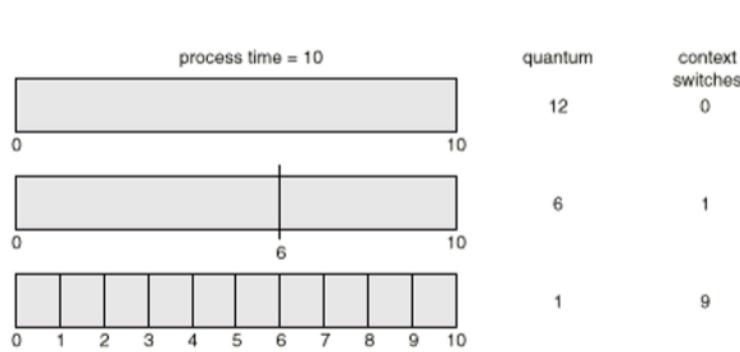
### Round Robin (RR).

È un *algoritmo* di scheduling *basato sul time-out*. Ciò significa che, ad ogni processo, viene assegnata una piccola parte (*quanto*) del tempo di CPU (i valori tipici vanno da 10 a 100 millisecondi). Al termine del quanto, il processo è prelazionato e messo nella ready queue, che in questo caso è una coda circolare. Con  $n$  processi nella coda, ogni processo ottiene  $1/n$  del tempo di CPU in blocchi di  $q$  unità di tempo alla volta, e nessun processo attende più di  $(n-1) * q$  unità di tempo. L’algoritmo RR è intrinsecamente *preemptive* (in pratica è un *FCFS con prelazione*). La scelta del quanto è importante, infatti, con un  $q$  troppo grande si sfocia nell’algoritmo *FCFS*, mentre con un  $q$  troppo piccolo potrebbe esserci *troppo overhead* dovuto ai context switch. Quindi un *valore ragionevole* di  $q$  deve essere molto maggiore del tempo di context switch, per fare in modo che almeno l’80% dei burst di CPU siano minori di  $q$ . Per quanto riguarda le *prestazioni* di RR, il suo tempo di turnaround può essere maggiore o uguale a quello di SJF, mentre il suo tempo di risposta può essere minore o uguale a quello di SJF.

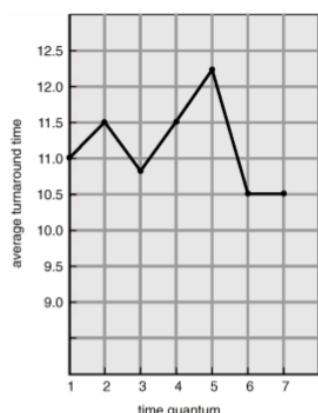
Esempio			Tempo di attesa medio							
Proc. (q=2)	T. di arrivo	CPU burst	Processo	T <sub>r</sub>	T <sub>w</sub>	T <sub>t</sub>				
P1	0	5	P1	0	7	12				
P2	0	1	P2	2	2	3				
P3	0	7	P3	3	8	15				
P4	0	2	P4	5	5	7				

Tempo (q=2)	0	2	3	5	7	9	11	12	14	15
In esecuzione	P1	P2	P3	P4	P1	P3	P1	P3	P3	
	P2	P3	P4	P1	P3	P1	P3			
Nella ready queue	P3	P4	P1	P3						
	P4	P1								



Relazione tra “quanto” e “context switch”



Relazione tra “quanto” e “turnaround”

## Code multilivello.

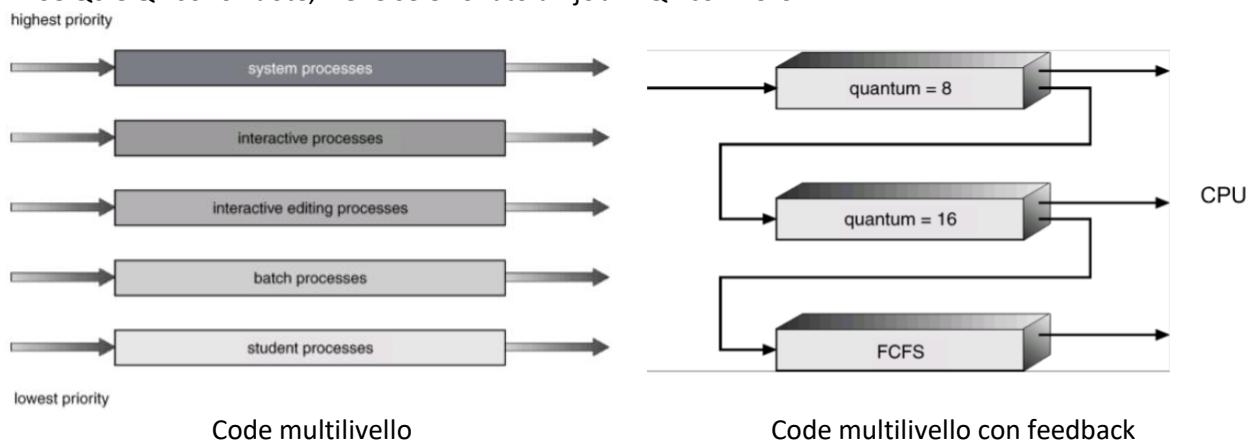
Esiste una classe di algoritmi in cui la *ready queue* è *partizionata in più code*, ad esempio una coda per i job in foreground (interattivi) e una coda per i job in background (batch). Ogni coda avrà quindi il suo algoritmo di scheduling, ad esempio i job in foreground verranno gestiti con RR e i job in background verranno gestiti con FCFS. Quello delle code multilivello è un meccanismo più generale, ma anche più complesso. Per questo è necessario uno *scheduling tra le code*, che può essere:

- *A priorità fissa*, quando si servono prima tutti i job di sistema, poi quelli in foreground, poi quelli in background, ma con la possibilità di starvation per code a priorità bassa.
- *Basato su time slice*, quando ogni coda ottiene un quanto del tempo di CPU che può usare per schedulare i suoi processi (ad esempio 80% per i job in foreground con RR, 20% per i job in background con FCFS).

## Code multilivello con feedback.

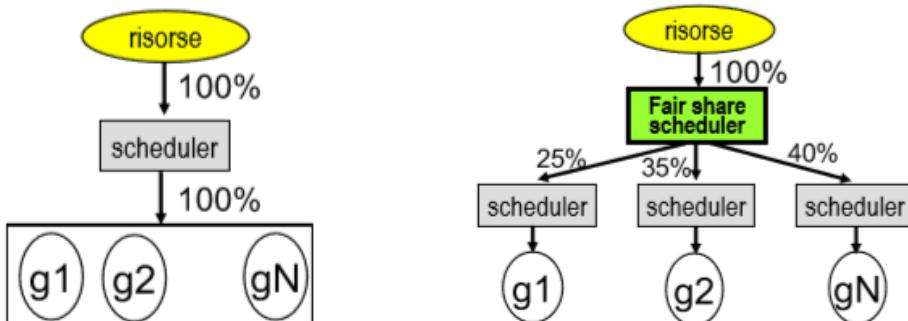
Nelle code multilivello *classiche*, un processo viene *allocato definitivamente ad una coda*. Nelle code multilivello *con feedback* (o adattative), invece, *un processo può spostarsi da una coda all'altra* a seconda delle sue caratteristiche, ed è quindi possibile implementare l'*aging*. Tra i *parametri dello scheduler* avremo quindi: numero delle code, algoritmi per ogni coda, criteri per la promozione o degradazione di un processo, criteri per definire la coda di ingresso di un processo. In un *esempio con 3 code* (Q0: RR con quanto 8 ms, Q1: RR con quanto 16 ms, Q2: FCFS), la CPU serve nell'ordine Q0, Q1, Q2 (i processi in Q1 vengono serviti se Q0 è vuota). Il *funzionamento* delle code multilivello con feedback è il seguente:

- Un nuovo job entra in Q0 e quando ottiene la CPU, riceve 8 ms di quanto; se non finisce entro il quanto, viene prelazionato e degradato alla coda Q1.
- Se Q0 è vuota, si seleziona un job di Q1 che riceve 16 ms di quanto; se non finisce, viene prelazionato e messo in Q2.
- Se Q0 e Q1 sono vuote, viene selezionato un job in Q2 con FCFS.



## Scheduling fair share.

Le politiche di scheduling precedenti sono orientate al processo, ma un'applicazione può essere composta da più processi. *Fair share* cerca di fornire *equità alle applicazioni* (e quindi agli utenti) e *non ai singoli processi*, suddividendo le risorse tra gruppi di processi invece che tra la totalità dei processi.



### **Contesto reale.**

In un contesto reale, l'obiettivo è quello di *minimizzare la complessità*. Gli algoritmi reali *usano la prelazione* e sono spesso *basati su RR*. Un esempio è rappresentato dallo *scheduler di Solaris* (Unix di Sun), che:

- È basato su priorità con *aging*.
- Priorità = priorità base + priorità corrente, dove:
  - Priorità base = [-20, ..., +20] con -20 = max e +20 = min.
  - Priorità corrente =  $0.1 * CPU(5n)$ , dove  $CPU(t)$  è l'utilizzo della CPU negli ultimi  $t$  secondi e  $n$  è il numero medio di processi pronti all'esecuzione nell'ultimo secondo.
- Dimentica il 90% dell'utilizzo di CPU degli ultimi  $5n$  secondi, per favorire i processi che hanno usato poco la CPU.

### ❖ Valutazione degli algoritmi

#### **Modello deterministico (analitico).**

È un metodo di valutazione *basato sull'algoritmo e su un preciso carico di lavoro* (è ciò che abbiamo fatto negli esempi precedenti). Questo metodo di valutazione *definisce le prestazioni di ogni algoritmo per uno specifico carico*, e quindi il risultato è applicabile solo al caso considerato. Di solito viene usato per *illustrare gli algoritmi*, in quanto richiede conoscenze troppo specifiche sulla natura dei processi.

#### **Modello a reti di code.**

Siccome non esiste un preciso gruppo di processi sempre uguali per utilizzare il modello deterministico, ed essendo possibile determinare le distribuzioni di CPU burst e I/O burst, si può usare un *sistema di calcolo descritto come una rete di server ognuno con la propria coda*. Nel modello a reti di code, si usano *formule matematiche* che indicano: la probabilità che si verifichi un determinato CPU burst; la distribuzione dei tempi di arrivo nel sistema dei processi. Da queste formule matematiche è possibile ricavare: utilizzo; throughput medio; tempi di attesa.

#### **Simulazione.**

Per la simulazione è *necessario programmare un modello del sistema*, utilizzando dati statistici o reali. Questo metodo di valutazione è abbastanza *preciso, ma costoso*.

#### **Implementazione.**

L'unico modo assolutamente sicuro per valutare un algoritmo di scheduling è quello di *codificarlo, inserirlo nel sistema operativo e vedere come funziona*.

# SINCRONIZZAZIONE TRA PROCESSI

## Buffer P/C: modello SW.

Il modello Produttore/Consumatore è un *modello astratto* in cui il produttore produce un messaggio, mentre il consumatore consuma un messaggio. L'esecuzione è *concorrente*, in quanto il produttore può aggiungere al buffer e il consumatore può togliere dal buffer. Un vincolo di questo modello è che il *buffer* è *limitato*, quindi non posso aggiungere in buffer pieni e non posso consumare da buffer vuoti.

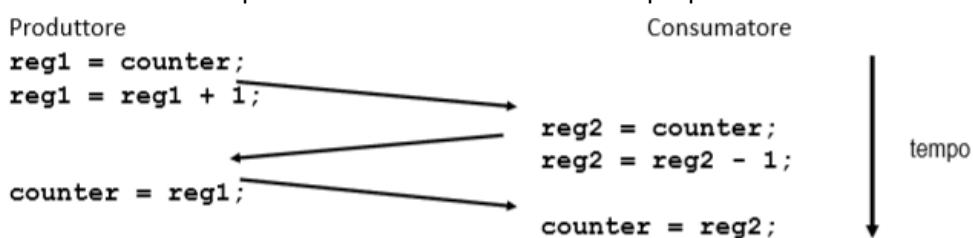
```
/*
 * Buffer circolare di N posizioni:
 * - counter è il numero di elementi nel buffer
 * - in è la prima posizione libera
 * - out è la prima posizione occupata
 * - se in=out, allora il buffer è vuoto
 * - se out=(in+1)%n, allora il buffer è pieno
 */

void deposit(item p) {
    // attende finché counter=N (buffer pieno)
    while(counter == N);
    buffer[in] = p;
    in = (in+1) % N;
    counter++;
}

item remove() {
    // attende finché counter=0 (buffer vuoto)
    while(counter == 0);
    next = buffer[out];
    out = (out+1) % N;
    counter--;
    return next;
}
```

## Buffer P/C: problema.

In Assembly, le istruzioni `counter++` e `counter--` sono *separate in più istruzioni*. Queste ultime vengono sempre *eseguite sequenzialmente*, ma non è noto il loro ordine di *interleaving*. Per esempio, nell'esecuzione alternata mostrata in figura c'è un'inconsistenza: se supponiamo `counter` pari a 5, P produce un item e fa `counter++` (idealmente `counter` diventa 6) e C consuma un item e fa `counter--` (idealmente `counter` ritorna 5), ma `counter` varrà 4 invece di 5 perché c'è stato un *context switch* proprio in mezzo alle istruzioni Assembly.



## Sezione critica.

Il problema che si presenta è che P e C possono modificare `counter` contemporaneamente. Una soluzione consiste nel proteggere l'accesso alla *sezione critica*, ovvero alla porzione di codice in cui si accede ad una risorsa condivisa.

## Criteri per la soluzione al problema della SC.

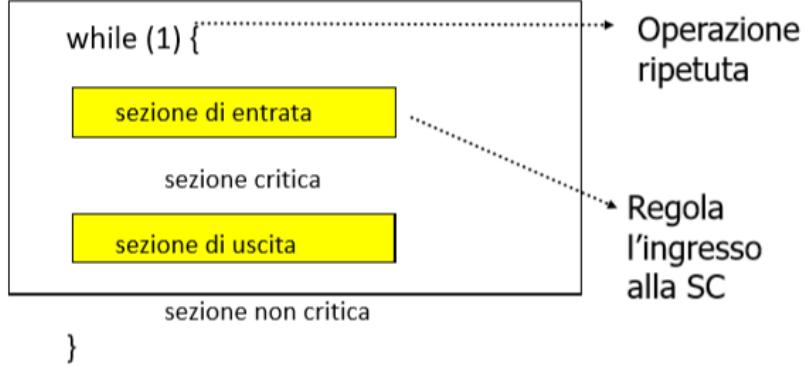
- *Mutua esclusione*: un processo alla volta può accedere alla sezione critica.
- *Progresso (progress)*: solo i processi che stanno per entrare nella sezione critica possono decidere chi entra (la decisione non può essere rimandata all'infinito).
- *Attesa limitata (bounded waiting)*: deve esistere un massimo numero di volte per cui un processo può entrare (di seguito).

## Tipologie di soluzioni al problema della SC.

Assumendo che la *sincronizzazione* si svolge *in ambiente globale* e che è necessaria la *condivisione di celle di memoria* (variabili condivise), ci possono essere due tipologie di soluzioni:

- *Soluzioni software*, che aggiungono codice alle applicazioni e non necessitano di alcun supporto hardware o del sistema operativo.
- *Soluzioni "hardware"*, che aggiungono codice alle applicazioni e necessitano di supporto hardware.

Nella figura è mostrata la *struttura* di un generico processo che accede a una risorsa condivisa (le parti evidenziate in giallo cambiano a seconda della soluzione adottata).



### ❖ Soluzioni software

#### Algoritmo 1 (turno).

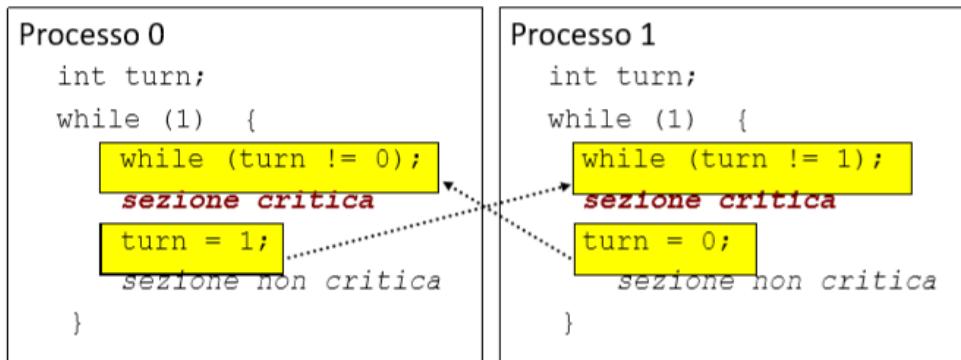
```

PROCESS i
int turn; /* se turn = i allora entra il processo i */
while (1) {
    while (turn != i); /* sezione di entrata */
    sezione critica
    turn = j; /* sezione di uscita */
    sezione non critica
}
  
```

2 soli processi  
(i=0,1); j = 1-i

Questo algoritmo richiede una *stretta alternanza tra i processi*, infatti quando i oppure j non sono interessati a entrare in SC, anche l'altro processo non può più entrare in SC. Questo, unito al fatto che non c'è nessuna nozione di "stato", fa sì che non venga rispettato il criterio del *progresso*.

#### Algoritmo 1: problema.



Se P0 cede il turno a P1 e poi non ha più necessità di entrare nella sezione critica, anche P1 non può più entrare nella sezione critica.

### Algoritmo 2 (stato).

```
PROCESS i
boolean flag[2]; /* inizializzato a FALSE */
while (1) {
    flag[i]=true; /* vuole entrare in SC */
    while (flag[j]==true); /* sezione di entrata */
    sezione critica
    flag[i]=false; /* sezione di uscita */
    sezione non critica
}
```

2 soli processi  
(i=0,1); j = 1-i

### Algoritmo 2: problema.

Questo algoritmo risolve il problema dell'algoritmo 1 ma l'esecuzione in sequenza dell'istruzione flag[] = true da parte dei due processi potrebbe portare a un *deadlock*:

- t0, P0 esegue flag[0]=true;
- t1, P1 esegue flag[1]=true;
- t2, P0 esegue while(flag[1]==true);
- t3, P1 esegue while(flag[0]==true);
- t4, deadlock (P0 e P1 bloccati sulla condizione del while).

### Algoritmo 2 (variante).

```
PROCESS i
boolean flag[2]; /* inizializzato a FALSE */
while (1) {
    while (flag[j]==true); /* sezione di entrata */
    flag[i]=true; /* vuole entrare in SC */
    sezione critica
    flag[i]=false; /* sezione di uscita */
    sezione non critica
}
```

Si invertono le istruzioni della sezione di entrata. Così facendo però violiamo la *mutua esclusione*, in quanto entrambi i processi possono trovarsi in SC se eseguono in sequenza il while prima di impostare flag a true.

### Algoritmo 3 (turno e stato).

```
PROCESS i
int turn; /* di chi è il turno? */
boolean flag[2]; /* iniz. a FALSE */
while (1) {
    flag[i] = TRUE; /* voglio entrare */
    turn = j; /* tocca a te, se vuoi */
    while (flag[j] == TRUE && turn == j);
    sezione critica
    flag[i] = FALSE;
    sezione non critica
}
```

2 soli processi

Questo algoritmo risolve anche il problema dell'algoritmo 2, in quanto entra il primo processo che esegue turn=j oppure turn=i.

### Algoritmo 3 (dimostrazione).

Mutua esclusione:

- $P_i$  entra nella SC se  $\text{flag}[j]=\text{false}$  o  $\text{turn}=i$
- Se  $P_i$  e  $P_j$  sono entrambi in SC allora  $\text{flag}[i]=\text{flag}[j]=\text{true}$
- Ma  $P_i$  e  $P_j$  non possono aver superato entrambi il while, perché  $\text{turn}$  vale  $i$  oppure  $j$
- Quindi solo uno dei due processi è entrato

Progresso e attesa limitata:

- Se  $P_j$  non è pronto per entrare nella SC, allora  $\text{flag}[j]=\text{false}$  e  $P_i$  può entrare
- Se  $P_j$  ha impostato  $\text{flag}[j]=\text{true}$  e si trova nel while allora  $\text{turn}=i$  oppure  $\text{turn}=j$
- Se  $\text{turn}=i$ ,  $P_i$  entra nella SC
- Se  $\text{turn}=j$ ,  $P_j$  entra nella SC
- In ogni caso quando  $P_j$  esce dalla SC imposta  $\text{flag}[j]=\text{false}$  e quindi  $P_i$  può entrare nella SC
- Quindi  $P_i$  entra nella SC al massimo dopo un'entrata di  $P_j$

### Algoritmo del fornaio.

```
N processi
```

```
PROCESS i
Iniz. a false  boolean choosing[N]; /* Pi sceglie un numero */
Iniz. a 0      int number[N];      /* ultimo numero scelto */

Prendo un numero → while(1) {
    choosing[i] = TRUE;
    number[i] = Max(number[0],...,number[N-1])+1;
    choosing[i] = FALSE;
    j sta scegliendo → for(j = 0; j < N; j++) {
        while(choosing[j] == TRUE);
        while(number[j] != 0 && number[j] < number[i]);
    }
    j è in CS e ha numero inferiore → sezione critica
    number[i] = 0;
    sezione non critica
}
```

Questo algoritmo risolve il problema con  $N$  processi. L'idea è che ogni processo sceglie un numero e il numero più basso verrà servito per primo. Per situazioni di numero identico si usa un confronto a due livelli (numero pescato, id processo). L'algoritmo è *corretto* in quanto soddisfa le tre proprietà.

### ❖ Soluzioni hardware

#### Interrupt e istruzioni atomiche.

Una soluzione "hardware" per risolvere il problema della SC è quella di *disabilitare gli interrupt* mentre una variabile condivisa viene modificata. Il problema di questa soluzione è che se il test per l'accesso è lungo, gli interrupt dovranno essere disabilitati per troppo tempo. Una soluzione alternativa consiste nell'usare le *istruzioni atomiche* (TestAndSet, Swap), ovvero nel fare in modo che l'operazione per l'accesso alla risorsa occupi un unico ciclo di istruzione (non interrompibile).

#### Utilizzo di TestAndSet e Swap.

- L'istruzione atomica *TestAndSet* assegna true alla variabile che le viene passata e ritorna il vecchio valore di questa variabile (che aveva salvato precedentemente).
- L'istruzione atomica *Swap* scambia il valore delle variabili che le vengono passate.

```

bool TestAndSet(bool &var) {
    bool temp;
    temp = var;
    var = TRUE;
    return temp;
}

bool lock; /* inizializzato a FALSE */

while(1) {
    while(TestAndSet(lock));
    /* sezione critica:
     * passa solo il primo processo che arriva e trova lock==FALSE
     */
    lock = FALSE;
    /* sezione non critica */
}

```

```

void Swap(bool &a, bool &b) {
    bool temp;
    temp = a;
    a = b;
    b = temp;
}

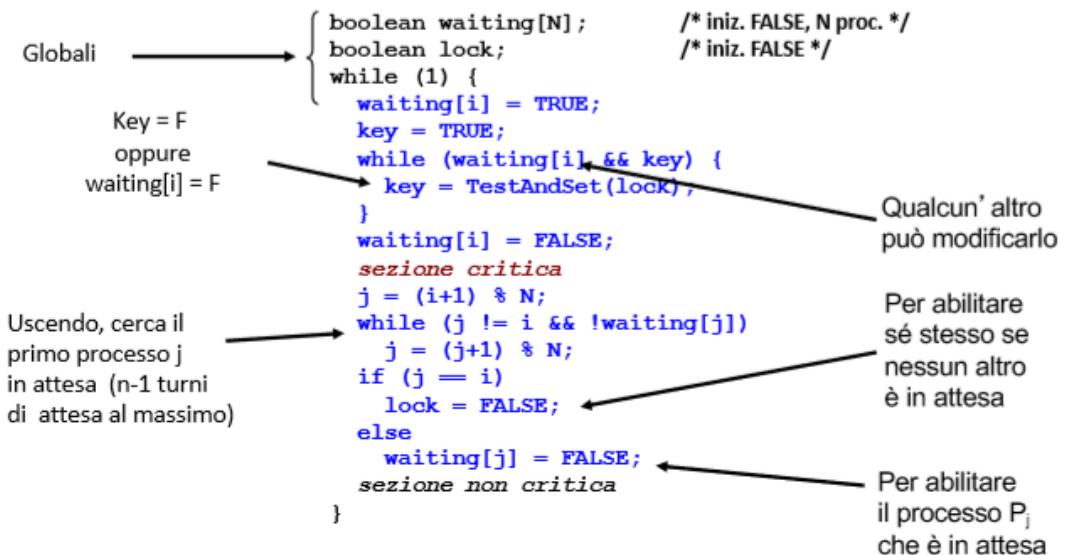
bool lock; /* inizializzato a FALSE */

while(1) {
    dummy = TRUE; /* locale del processo */
    do {
        Swap(dummy, lock);
    } while (dummy == TRUE);
    /* sezione critica:
     * quando dummy==FALSE Pi accede alla SC,
     * gli altri processi continuano a scambiare TRUE con TRUE e
     * non accedono a SC finché Pi non pone lock a FALSE
     */
    lock = FALSE;
    /* sezione non critica */
}

```

### TestAndSet con attesa limitata.

TestAndSet e Swap non rispettano l'*attesa limitata*, cioè manca un equivalente della variabile *turn*. Per realizzare un algoritmo con attesa limitata sono necessarie *variabili addizionali*, come mostrato in figura.



### Vantaggi e svantaggi delle soluzioni hardware.

- + Scalabili, cioè sono indipendenti dal numero di processi coinvolti.
- + L'estensione a  $N$  sezioni critiche è immediata.
- Maggiore complessità per il programmatore rispetto alle soluzioni software (vedi sopra).
- Serve busy waiting e quindi si spreca tempo di CPU (nei cicli while).

### ❖ Semafori

#### Introduzione.

Alcuni dei *problemi delle soluzioni precedenti* sono che: *non sono banali* da aggiungere ai programmi e sono *basate sul busy waiting* (attesa attiva). In alternativa perciò, si ricorre ai *semafori*, ovvero una soluzione generica che funziona sempre. I semafori sono *variabili intere* a cui si accede attraverso *due primitive atomiche*:

- $V(S)$  o Signal, che incrementa il valore del semaforo  $S$  di 1.
- $P(S)$  o Wait, che tenta di decrementare il valore del semaforo  $S$  di 1, in quanto se il valore di  $S$  è 0, non si può decrementare ed è necessario attendere.

## Semafori binari e interi.

I *semafori interi* possono essere inizializzati con valori interi maggiori o uguali a 0. I *semafori binari*, invece, possono essere inizializzati solamente a 0 (FALSE) oppure a 1 (TRUE). I semafori binari, tuttavia, hanno lo stesso potere espressivo di quelli interi. Nelle immagini sottostanti è riportata l'*implementazione concettuale* (non reale) di questi semafori:

- Semaforo binario

```
P(s) :
while (s == FALSE); // attesa
s = FALSE;

V(s) :
s = TRUE;
```

- Semaforo intero

```
P(s) :
while (s==0); // attesa
s--;

V(s) :
s++;
```

In queste altre immagini, invece, è riportata la loro l'implementazione reale, in cui devo garantire l'*atomicità*:

- Semaforo binario (con busy waiting)

```
/* s inizializzato a TRUE */
P(bool &s)
{
    key = FALSE;
    do {
        Swap(s, key);
    } while (key == FALSE);
}
```

```
V(bool &s)
{
    s = TRUE;
}
```

- Semaforo intero (con busy waiting)

```
bool mutex; /* Sem. binario iniz. TRUE */
bool delay; /* Sem. binario iniz. FALSE */

P(int &s)
{
    P(mutex);
    s = s - 1;
    if (s < 0) {
        V(mutex);
        P(delay);
    } else V(mutex);
}

V(int &s)
{
    P(mutex);
    s = s + 1;
    if (s <= 0) {
        V(delay);
    }
    V(mutex);
}
```

**P,V = semaforo intero P,V = semaforo binario**

- Semaforo intero (senza busy waiting)

```
V(Sem &s) {
    P(mutex);
    s.value = s.value + 1;
    if (s.value <= 0) {
        V(mutex);
        PCB *p = remove(s.List);
        wakeup(p);
    } else
        V(mutex);
}
```

```
bool mutex /*sem bin. inizializzato a true*/
```

```
typedef struct {
    int value;
    PCB *List;
} Sem;
```

```
P(Sem &s) {
    P(mutex);
    s.value = s.value - 1;
    if (s.value < 0) {
        V(mutex);
        append (process i, s.List);
        sleep();
    } else
        V(mutex);
}
```

Abbiamo quindi visto che l'implementazione reale è diversa da quella concettuale, in quanto il valore di un semaforo intero può diventare minore di 0 e il semaforo stesso conta quanti processi sono in attesa. Inoltre, per garantire attesa limitata, la lista dei PCB può essere FIFO (strong semaphore).

## Semafori senza busy waiting.

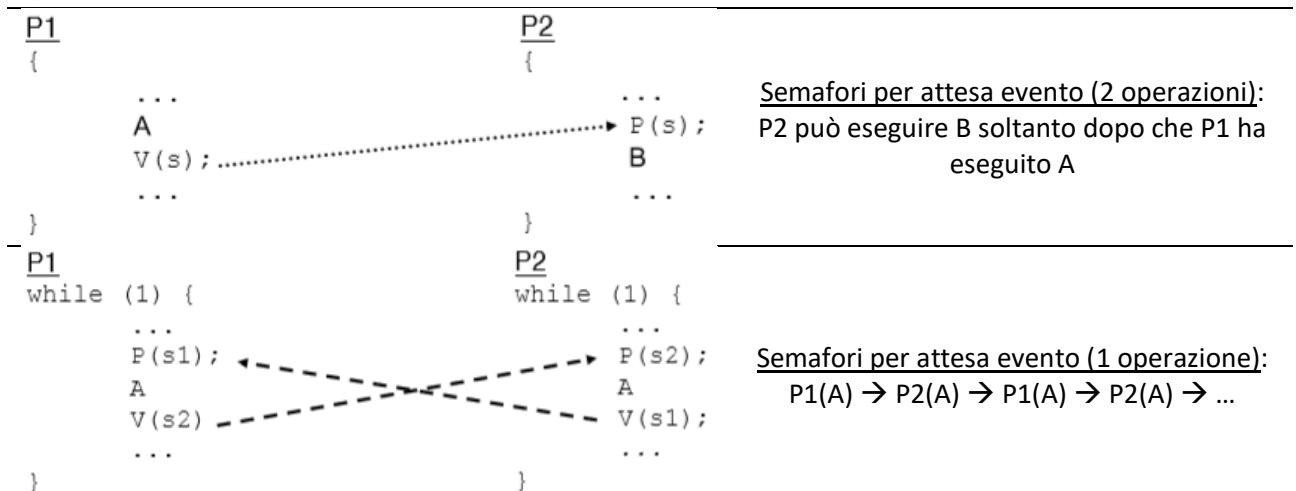
Il busy waiting viene eliminato mettendo il processo nello *stato waiting* al posto di farlo aspettare in un altro semaforo (P di delay). Tuttavia, il busy waiting rimane nella P e nella V del mutex, ma siccome la *modifica* del mutex è *veloce*, verrà sprecato *poco tempo*. In alternativa posso *disabilitare gli interrupt* durante P e V.

### Usi principali dei semafori.

- *Protezione di una sezione critica* per N processi: si usa un semaforo binario di mutua esclusione (mutex) inizializzato a 1.
- *Sincronizzazione* (del tipo attesa di evento) tra processi, che può essere di due tipi:
  - P1 e P2 devono sincronizzarsi rispetto all'esecuzione di *due operazioni* A e B, perciò si usa un semaforo binario inizializzato a 0.
  - P1 e P2 devono sincronizzarsi rispetto all'esecuzione di *un'operazione* A, si usano due semafori binari, uno inizializzato a 1 e l'altro inizializzato a 0.

```
/* valore iniziale di s = 1 (mutex) */
while (1) {
    P(s);
    sezione critica
    V(s);
    sezione non critica
}
```

### Semafori e sezione critica



### Problematiche dei semafori.

- *Deadlock (blocco critico)*: si ha quando un processo rimane bloccato in attesa di un evento che solo lui può generare.
- *Starvation*: si ha quando l'attesa all'interno di un semaforo è indefinita.

### Problemi classici: Produttore/Consumatore.

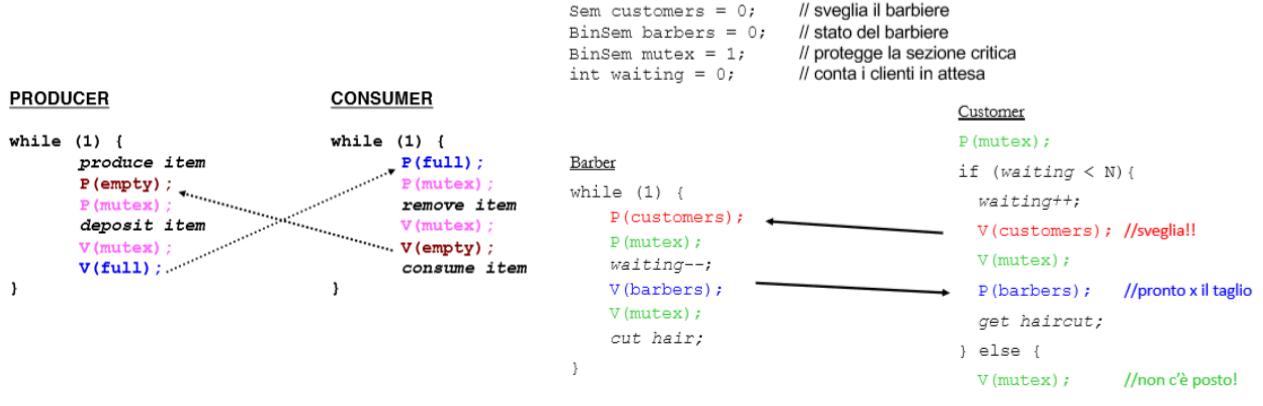
Servono 3 semafori:

- Un semaforo binario inizializzato a TRUE per garantire la mutua esclusione nell'utilizzo del buffer (*mutex*).
- Un semaforo intero inizializzato a N per bloccare il Produttore se il buffer è pieno (*empty*).
- Un semaforo intero inizializzato a 0 per bloccare il Consumatore se il buffer è vuoto (*full*).

### Problemi classici: Sleepy Barber.

Un negozio ha una sala d'attesa con N sedie ed una stanza con la sedia del barbiere. In assenza di clienti, il barbiere si addormenta. Quando entra un cliente:

- Se le sedie sono tutte occupate, il cliente se ne va.
- Se il barbiere è occupato, il cliente si siede.
- Se il barbiere è addormentato, il cliente lo sveglia.



### Produttore/Consumatore

### Sleepy Barber

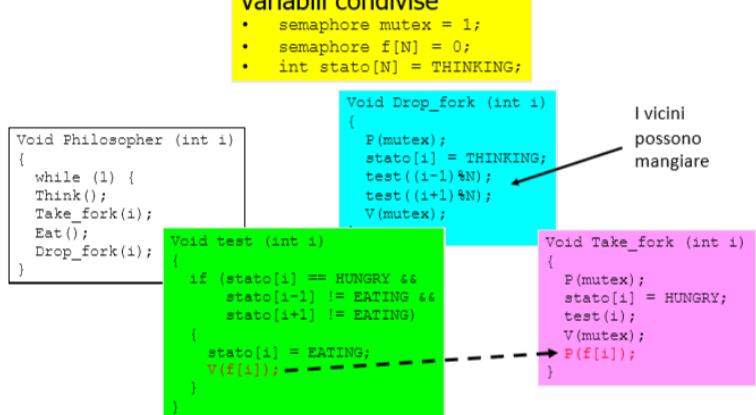
#### Problemi classici: Dining Philosophers.

- *Soluzione intuitiva:*  $N$  semafori inizializzati a 1. Con  $P(s[j])$  cerco di prendere la bacchetta  $j$  e con  $V(s[j])$  rilascio la bacchetta  $j$ . Tuttavia, questa soluzione è incompleta, in quanto è possibile che si verifichi un *deadlock* se tutti i filosofi tentano di prendere la bacchetta alla loro destra (o sinistra) contemporaneamente.
- *Soluzione corretta:* servono  $N$  semafori inizializzati a 0, un semaforo inizializzato a 1 per la *mutua esclusione* e tre stati per ogni filosofo così descritti:
  - *THINKING*, se un filosofo pensa non interagisce con gli altri.
  - *HUNGRY*, se un filosofo ha fame prende 2 bacchette (una alla volta) e inizia a mangiare, ma se non ci sono 2 bacchette libere il filosofo non può mangiare.
  - *EATING*, quando un filosofo termina di mangiare rilascia le bacchette.

```

do {
    P(s[i])
    P(s[(i+1) % N])
    ...
    // mangia
    ...
    V(s[i]);
    V(s[(i+1) % N]);
    ...
    // pensa
    ...
} while (1);

```



### Dining Philosopher (soluzione intuitiva)

### Dining Philosophers (soluzione corretta)

#### Limitazioni dei semafori.

Tra i problemi derivanti dall'utilizzo dei semafori troviamo la *difficoltà nella scrittura dei programmi* e la *scarsa visibilità della correttezza delle soluzioni*. Un'alternativa è utilizzare specifici costrutti forniti dai linguaggi di programmazione ad alto livello, come i *monitor* o le *classi synchronized* di Java.

#### Struttura e caratteristiche del monitor.

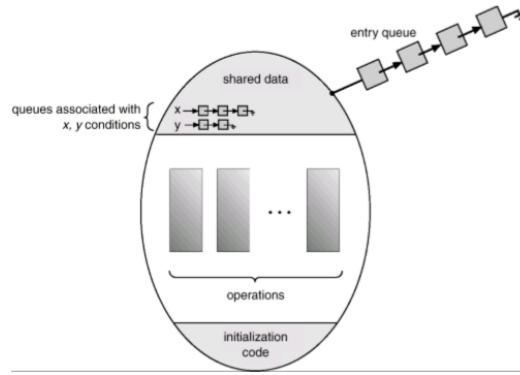
I monitor (simili al concetto di *classe*) sono dei costrutti per la condivisione sicura ed efficiente di dati tra processi. Le *variabili* del monitor sono visibili solo all'interno del monitor stesso e le *procedure* del monitor accedono solo alle variabili definite nel monitor. Siccome è possibile che un solo processo alla volta sia attivo in un monitor, il programmatore non deve codificare esplicitamente la mutua esclusione. Quindi con i monitor si programma rischiando di fare *meno errori* rispetto ai semafori, ma si ha lo svantaggio che *pochi linguaggi* forniscono i monitor e che i monitor richiedono la presenza di *memoria condivisa*.

```

monitor xyz{
    // dichiarazione di variabili (stato del monitor)
    entry P1 (...) {
        ...
    }
    entry Pn (...) {
        ...
    }
    {
        // codice di inizializzazione
    }
}

```

Struttura di un monitor



Rappresentazione di un monitor

### Tipi **condition** e primitive **wait** e **signal**.

Per permettere ad un processo di attendere all'interno del monitor è necessario definire variabili di tipo **condition**. Queste variabili vengono dichiarate all'interno del monitor e sono accessibili solo tramite due **primitive simili (ma non uguali) a quelle dei semafori**:

- **wait**, che fa in modo che il processo che invoca la **wait** venga bloccato fino all'invocazione della sua corrispondente **signal** da parte di un altro processo.
- **signal**, che sveglia esattamente un processo (se più processi sono in attesa, lo scheduler deciderà quale processo può entrare; mentre se nessun processo è in attesa, non ci sarà alcun effetto).

Dopo una **signal**, il processo che l'ha invocata può: bloccarsi e passare l'esecuzione all'eventuale processo sbloccato; oppure uscire dal monitor (in questo caso la **signal** deve essere l'ultima istruzione della procedura).

```

monitor BinSem
{
    boolean busy; /* iniz. FALSE */
    condition idle;

    entry void P( )
    {
        if (busy) idle.wait();
        busy = TRUE;
    }
    entry void V( )
    {
        busy = FALSE;
        idle.signal ();
    }
    busy = FALSE; /* inizializzazione */
}

```

Esempio di monitor

```

Producer()
{
    while (TRUE){
        make_item(); // crea nuovo item
        ProducerConsumer.enter(); // chiamata alla funzione enter
    }
}

Consumer()
{
    while (TRUE){
        ProducerConsumer.remove(); // chiamata alla funzione remove
        consume_item(); // consuma item
    }
}

```

Implementazione di Buffer P/C

### Sincronizzazione in Java.

La sezione critica viene protetta utilizzando i metodi **synchronized**, ovvero metodi che possono essere eseguiti da una sola thread alla volta e che vengono realizzati mantenendo un singolo **lock** per oggetto (se usassimo **synchronized** su metodi statici, avremo un singolo lock per classe). Le primitive **wait()**, **notify()**, **notifyAll()** sono ereditate da tutti gli oggetti. Nell'immagine sottostante c'è l'**implementazione di Buffer P/C**.

<pre> public class BoundedBuffer {     Object [] buffer;     int nextin;     int nextout;     int size;     int count; } // costruttore public BoundedBuffer (int n){     size = n;     buffer = new Object[size];     nextin = 0;     nextout = 0;     count = 0; } </pre>	<pre> public synchronized deposit(Object x) {     while (count == size) wait();     buffer[nextin] = x;     nextin = (nextin+1) mod N;     count = count + 1;     notifyAll(); }  public synchronized Object remove() {     Object x;     while (count == 0) wait();     x = buffer[nextout];     nextout = (nextout+1) mod N;     count = count - 1;     notifyAll();     return x; } </pre>
---	---

# DEADLOCK

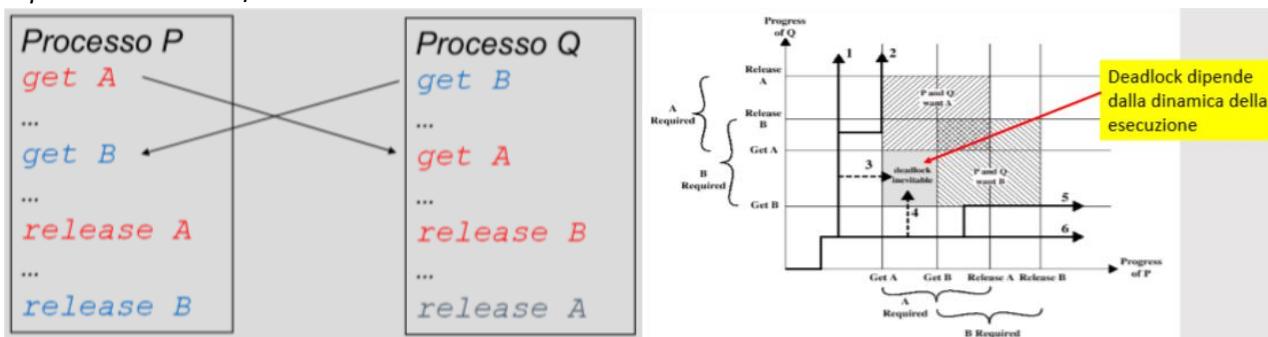
## ❖ Introduzione

### Definizione.

Un insieme di processi è in *deadlock* quando ogni processo è in attesa di un evento (rilascio di una risorsa) che può essere causato da un processo dello stesso insieme. Un esempio di deadlock è rappresentato da un ponte (risorsa condivisa) che si può attraversare in una sola direzione: se c'è deadlock (due macchine hanno imboccato il ponte), si può risolvere mandando indietro una macchina (preemption + rollback). Ma è anche possibile che più macchine debbano essere spostate e che si verifichi la *starvation* (una macchina non passa mai perché passano sempre le macchine dall'altro lato).

### Traiettoria delle risorse.

Se supponiamo che i processi P e Q debbano utilizzare le risorse A e B in modo esclusivo, si avranno 6 possibili sequenze di richiesta/rilascio.



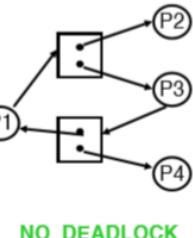
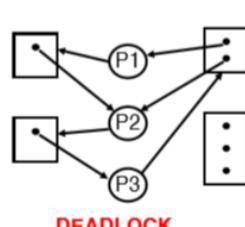
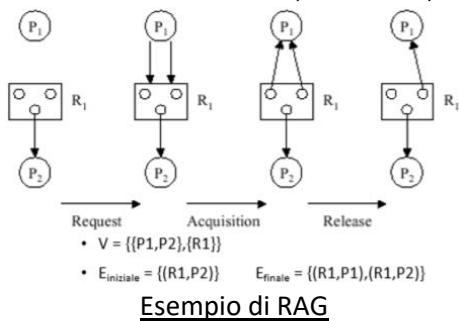
### Condizioni necessarie.

Se le seguenti *condizioni* sono vere contemporaneamente, potrebbe verificarsi un deadlock (se anche solo una non si verifica, non si avrà mai il deadlock):

- *Mutua esclusione*, cioè almeno una risorsa deve essere non condivisibile.
- *Hold and Wait*, cioè deve esistere un processo che detiene una risorsa e attende di acquisirne un'altra.
- *No Preemption*, le risorse non possono essere rilasciate se non volontariamente dal processo che le usa.
- *Attesa circolare*, cioè devono esistere dei processi che attendono ciclicamente il liberarsi di una risorsa.

### Modello astratto (RAG).

Il Resource Allocation Graph è un grafo del tipo  $G(V,E)$ , dove  $V$  indica i nodi (*cerchi* per i processi, *rettangoli* con tanti punti quante sono le relative istanze per le risorse) ed  $E$  indica gli archi (da processi a risorse quando il processo *richiede* una risorsa e da risorse a processi quando il processo *detiene* una risorsa). Se il RAG non contiene cicli, non si avrà mai il deadlock. Se il RAG contiene cicli e si ha una sola istanza per ogni risorsa, potrebbe verificarsi un deadlock; altrimenti, se ci sono più istanze, dipende dallo schema di allocazione.



RAG e deadlock

## Gestione dei deadlock.

Esistono vari meccanismi alternativi per la gestione dei deadlock:

- *Prevenzione statica*, ovvero evitare che si possa verificare una delle quattro condizioni precedenti.
- *Prevenzione dinamica* (avoidance) basata su allocazione delle risorse, che non viene mai usata poiché richiede una conoscenza troppo approfondita delle richieste delle risorse.
- *Rivelazione* (detection) e *ripristino* (recovery), ovvero permettere che si verifichino i deadlock mettendo a punto dei metodi per rilevarli e riportare il sistema al funzionamento normale.
- *Algoritmo dello struzzo*, ovvero non fare nulla in quanto i deadlock sono rari e gestirli costa troppo.

### ❖ Prevenzione statica

#### Obiettivo.

L'obiettivo della prevenzione statica è quello di impedire che si verifichi una delle quattro condizioni che devono essere vere contemporaneamente per portare ad un deadlock. In realtà la condizione di *mutua esclusione* non si può togliere in quanto è irrinunciabile per certi tipi di risorsa.

#### Hold and Wait.

Per impedire che si verifichi la condizione di Hold and Wait si deve fare in modo che un processo allochi all'inizio tutte le risorse che intende utilizzare. Così facendo un processo può ottenere una risorsa solo se non ne ha altre. I *problem*i di questa soluzione sono il basso utilizzo delle risorse e la possibilità di starvation (rappresentata dalla richiesta di molte risorse molto "popolari").

#### No Preemption.

Per impedire che si verifichi la condizione di No Preemption si deve fare in modo che un processo che richiede una risorsa non disponibile debba cedere tutte le altre risorse che detiene. In alternativa, tale processo può cedere le risorse che detiene sulla base delle richieste degli altri processi. Il *problema* di questa soluzione è che è fattibile solo per risorse il cui stato può essere facilmente "ristabilito".

#### Attesa circolare.

Per impedire che si verifichi la condizione di Attesa circolare si deve assegnare una priorità (ordinamento globale) ad ogni risorsa. Così facendo un processo può richiedere risorse solo in ordine crescente di priorità e quindi l'attesa circolare diventa impossibile. La funzione di priorità  $F: R \rightarrow N$  è costruita in modo che  $F(R0) < F(R1) < \dots < F(Rn)$ , quindi se  $P0 \rightarrow R0 \rightarrow P1 \rightarrow \dots \rightarrow Rn-1 \rightarrow Pn \rightarrow Rn \rightarrow P0$  si avrebbe  $F(R0) < F(R1) < \dots < F(Rn) < F(R0)$ , il che è impossibile.

### ❖ Prevenzione dinamica

#### Obiettivo.

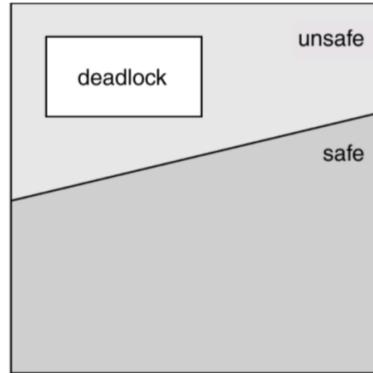
Le tecniche di prevenzione statica possono portare ad un basso utilizzo delle risorse, in quanto mettono dei vincoli sul modo in cui i processi possono accedere a tali risorse. Le tecniche di prevenzione dinamica, invece, si basano sulle richieste, in quanto fanno l'analisi dinamica del grafo delle risorse per evitare situazioni cicliche. Il *requisito* alla base di queste tecniche è la conoscenza del caso peggiore, cioè bisogna conoscere il massimo numero di istanze di ogni risorsa richiesta da ogni processo.

#### Stato safe.

Lo *stato* di una risorsa viene calcolato sulla base del numero di istanze allocate e del numero di istanze disponibili. Il sistema si trova in uno stato sicuro (*safe*) se esiste una sequenza safe ovvero se, usando le risorse disponibili, il sistema può allocare risorse ad ogni processo (in qualche ordine) in modo che ciascuno di essi possa terminare la sua esecuzione.

### Sequenza safe.

Una sequenza di processi ( $P_1, \dots, P_n$ ) è detta safe se, per ogni  $P_i$ , le risorse che  $P_i$  può richiedere possono essere esaudite usando le risorse disponibili e quelle detenute da  $P_j$  (con  $j < i$  e attendendo che  $P_j$  termini). Se tale sequenza non esiste, siamo in uno stato unsafe. Non tutti gli stati unsafe sono stati di deadlock, ma da uno stato unsafe posso andare in deadlock.



### Stato safe e unsafe – esempio.

Supponiamo di avere 3 processi ( $P_0, P_1, P_2$ ) e 12 istanze di 1 risorsa. Al tempo  $T_0$  abbiamo 3 istanze libere e perciò siamo in uno stato safe, in quanto esiste una sequenza safe ( $P_1 \rightarrow P_0 \rightarrow P_2$ ). Se al tempo  $T_1$  il processo  $P_2$  richiede 1 istanza e gli viene assegnata, invece, saremo in uno stato unsafe, in quanto avremo solo 2 istanze libere. A questo punto  $P_1$  può eseguire, ma quando termina rende disponibili solo 4 istanze e siccome  $P_0$  ne richiede 5 e  $P_2$  ne richiede 6 si avrà un deadlock ( $P_2 \leftrightarrow P_0$ ).

<u><math>T_0</math></u>	Richieste	Possedute
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2

<u><math>T_1</math></u>	Richieste	Possedute
$P_0$	10	5
$P_1$	4	2
$P_2$	9	3

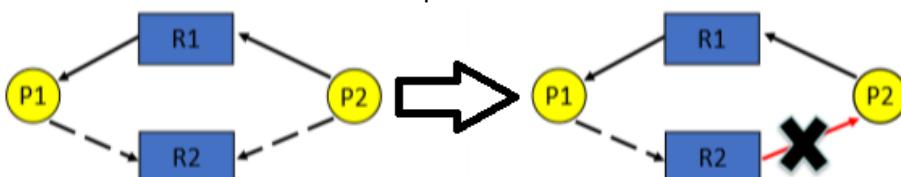
### Algoritmi.

L'idea alla base della prevenzione dinamica è quella di utilizzare algoritmi che lasciano il sistema sempre in uno stato safe. Il funzionamento di tali algoritmi prevede che, all'inizio, il sistema è in uno stato safe e poi, ogni volta che un processo richiede una risorsa, tale risorsa gli viene assegnata se e solo se si rimane in uno stato safe. Lo svantaggio di questa tecnica è che l'utilizzo delle risorse è minore rispetto a quello che si ha usando altre tecniche. Esistono quindi due soluzioni alternative:

- *Algoritmo con RAG*, che funziona solo se c'è una sola istanza per ogni risorsa.
- *Algoritmo del banchiere*, che funziona qualunque sia il numero di istanze.

### Algoritmo con RAG.

Al RAG vengono aggiunti gli *archi di reclamo*, i quali si rappresentano con una freccia tratteggiata e indicano che un processo può richiedere una determinata risorsa in futuro. Per fare ciò, all'inizio, ogni processo deve dire quali risorse vorrebbe usare durante la sua esecuzione. A questo punto, una richiesta viene soddisfatta se e solo se l'allocazione della risorsa non crea un *ciclo* nel RAG. Inoltre, per rilevare tali cicli, serve un algoritmo. Nell'esempio rappresentato in figura, all'inizio siamo in uno stato sicuro ma se, ad un certo punto,  $P_2$  richiede  $R_2$ , la sua richiesta non viene accettata poiché lo stato diventerebbe unsafe.



## Algoritmo del banchiere.

È meno efficiente dell'algoritmo con RAG, ma funziona con più istanze delle risorse. L'idea è che il banchiere non deve mai distribuire tutto il denaro che ha in cassa perché altrimenti non potrebbe più soddisfare i clienti successivi. All'inizio, ogni processo dichiara la sua massima richiesta e poi, ogni volta che un processo richiede una risorsa, si determina se siamo ancora in uno stato safe. Perciò l'algoritmo del banchiere è costituito da un algoritmo di allocazione e da un algoritmo di verifica dello stato, che funzionano in questo modo:

- *Strutture dati*

```
int available[m]; /* n° di istanze di Ri disponibili */
int max[n][m]; /* matrice delle richieste di risorse */
int alloc[n][m]; /* matrice allocazione corrente */
int need[n][m]; /* matrice bisogno rimanente ovvero
    need[i][j] = max[i][j] - alloc[i][j]*/
```

- *Algoritmo di allocazione*

```
void request(int req_vec[]) {
    if (req_vec[] > need[i][])
        error(); /* superato il massimo preventivo */
    if (req_vec[] > available[])
        wait(); /* attendo che si liberino risorse */
    available[] = available[] - req_vec[];
    alloc[i][] = alloc[i][] + req_vec[];
    need[i][] = need[i][] - req_vec[];
    if (!state_safe()) /* se non è safe, ripristino il vecchio stato */
        available[] = available[] + req_vec[];
    alloc[i][] = alloc[i][] - req_vec[];
    need[i][] = need[i][] + req_vec[];
    wait();
}
```

Richieste del processo P<sub>i</sub>

"simulo" l'assegnazione

rollback

- *Algoritmo di verifica dello stato*

```
boolean state_safe() {
    int work[m] = available[];
    boolean finish[n] = {FALSE,...,FALSE};
    int i;
    while (finish != {TRUE,...,TRUE}) {
        /* cerca Pi che NON abbia terminato e che possa
        compilare con le risorse disponibili in work */
        for (i=0; i<n) && (finish[i] || (need[i] > work));
        if (i==n)
            return FALSE; /* non c'è → unsafe */
        else {
            work[] = work[]+alloc[i];
            finish[i] = TRUE;
        }
    }
    return TRUE;
}
```

Ho già sottratto le richieste di P<sub>i</sub>

Sono arrivato in fondo, senza trovare finish[i]=FALSE e need[i] <= work[]

L'ordine non ha importanza. Se + processi possono eseguire, ne posso scegliere uno a caso, gli altri eseguiranno dopo, visto che le risorse possono solo aumentare

## Algoritmo del banchiere – esempio.

Supponiamo di avere 5 processi (P<sub>0</sub>, P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>) e 3 tipi di risorsa (A con 10 istanze, B con 5 istanze, C con 7 istanze). Al tempo T0 siamo in uno stato safe, in quanto esiste una sequenza safe (P<sub>1</sub>→P<sub>3</sub>→P<sub>4</sub>→P<sub>2</sub>→P<sub>0</sub>). Al tempo T1 il processo P<sub>1</sub> richiede [1,0,2] istanze. Siccome  $req\_vec \leq available$ , cioè  $[1,0,2] \leq [3,3,2]$ , le risorse possono essere assegnate a P<sub>1</sub>. A questo punto siamo ancora in uno stato safe, in quanto esiste una sequenza safe (P<sub>1</sub>→P<sub>3</sub>→P<sub>4</sub>→P<sub>0</sub>→P<sub>2</sub>). Al tempo T2 il processo P<sub>0</sub> richiede [0,2,0] istanze. Siccome  $req\_vec \leq available$ , cioè  $[0,2,0] \leq [2,3,0]$ , le risorse possono essere assegnate a P<sub>0</sub>. A questo punto, però, siamo in uno stato unsafe perché le istanze disponibili non soddisfano nessuna delle richieste dei processi.

T0:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	A B C	A B C	A B C	A B C
P <sub>0</sub>	0 1 0	7 5 3	3 3 2	7 4 3
P <sub>1</sub>	2 0 0	3 2 2		1 2 2
P <sub>2</sub>	3 0 2	9 0 2		6 0 0
P <sub>3</sub>	2 1 1	2 2 2		0 1 1
P <sub>4</sub>	0 0 2	4 3 3		4 3 1

T1:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	A B C	A B C	A B C	A B C
P <sub>0</sub>	0 1 0	7 5 3	2 3 0	7 4 3
P <sub>1</sub>	3 0 2	3 2 2		0 2 0
P <sub>2</sub>	3 0 2	9 0 2		6 0 0
P <sub>3</sub>	2 1 1	2 2 2		0 1 1
P <sub>4</sub>	0 0 2	4 3 3		4 3 1

Verifica stato safe T1:

- work = (2,3,0)
- i=0 finish=FALSE, need[0] = (7,4,3) > work
- i=1 finish=FALSE, need[1] = (0,2,0) < work
  - work = work + (3,0,2) = (5,3,2)
  - finish[1] = TRUE
- i=2 finish=FALSE, need[2]=(6,0,0) > work
- i=3 finish=FALSE, need[3]=(0,1,1) < work
  - work = work + (2,1,1) = (7,4,3)
  - finish[3] = TRUE
- ...

T2:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	A B C	A B C	A B C	A B C
P <sub>0</sub>	0 3 0	7 5 3	2 1 0	7 2 3
P <sub>1</sub>	3 0 2	3 2 2		0 2 0
P <sub>2</sub>	3 0 2	9 0 2		6 0 0
P <sub>3</sub>	2 1 1	2 2 2		0 1 1
P <sub>4</sub>	0 0 2	4 3 3		4 3 1

❖ Rilevazione e ripristino

## Approcci.

La prevenzione (statica o dinamica) è conservativa e riduce eccessivamente l'utilizzo delle risorse. Due approcci alternativi sono rappresentati da: *rilevazione e ripristino con RAG* e *algoritmo di rilevazione*.

## Rilevazione e ripristino con RAG.

Funziona solo con una risorsa per tipo e consiste nell'analizzare periodicamente il RAG per verificare se esistono deadlock (*detection*) e, in tal caso, iniziare il ripristino (*recovery*). Il *vantaggio* di questa tecnica è che non richiede la conoscenza anticipata delle richieste, mentre lo svantaggio è il costo del ripristino.

## Algoritmo di rilevazione.

È basato sull'esplorazione di ogni possibile sequenza di allocazione per i processi che non hanno ancora terminato. Se la sequenza va a buon fine (*safe*), non c'è deadlock. L'algoritmo funziona in questo modo:

- *Strutture dati*

```
int available[m]; /* n° di istanze di Ri disponibili */
int alloc[n][m]; /* matrice allocazione corrente */
int req_vec[n][m];/* matrice delle richiesta */
```

- *Algoritmo di detection*

```

int work[m] = available[m];
bool finish[] = (FALSE,...,FALSE), found = TRUE;
while (found) {
    found = FALSE;
    for (i=0; i<n && !found; i++) {
        /* cerca un Pi con richiesta soddisfacibile */
        if (!finish[i] && req_vec[i][] <= work[]) {
            /* assume ottimisticamente che Pi esegua fino al termine
               e che quindi restituiscia le risorse */
            work[] = work[] + alloc[i][];
            finish[i]=TRUE;
            found=TRUE;
        }
    }
} /* se finish[i]=FALSE per un qualsiasi i, Pi è in deadlock */

```

Se non è così  
il possibile deadlock  
verrà evidenziato  
alla prossima  
esecuzione dell'algoritmo

## **Algoritmo di rilevazione – esempio.**

Supponiamo di avere 5 processi ( $P_0, P_1, P_2, P_3, P_4$ ) e 3 tipi di risorsa (A con 7 istanze, B con 2 istanze, C con 6 istanze). Al tempo  $T_0$  siamo in uno stato safe, in quanto esiste una sequenza safe ( $P_0 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1 \rightarrow P_4$ ).

Al tempo  $T1$  il processo P2 richiede [0,0,1] istanze. A questo punto siamo in deadlock perché le istanze disponibili soddisfano solamente la richiesta del processo P0. Perciò il deadlock è formato da P1, P2, P3 e P4.

TO:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P <sub>0</sub>	0 1 0	0 0 0	0 0 0
P <sub>1</sub>	2 0 0	2 0 2	
P <sub>2</sub>	3 0 3	0 0 0	
P <sub>3</sub>	2 1 1	1 0 0	
P <sub>4</sub>	0 0 2	0 0 2	

### Verifica stato safe T0:

- |                  |                                 |    |                       |
|------------------|---------------------------------|----|-----------------------|
| • work = (0,0,0) |                                 |    |                       |
| • i=0            | req[0]=(0,0,0) <= work          | OK |                       |
|                  | work = work + (0,1,0) = (0,1,0) |    | finish[0] = true P0 ✓ |
| • i=1            | req[1]=(2,0,2) <= work          | NO |                       |
| • i=2            | req[2]=(0,0,0) <= work          | OK |                       |
|                  | work = work + (3,0,3) = (3,1,3) |    | finish[2] = true P2 ✓ |
| • i=3            | req[3]=(1,0,0) <= work          | OK |                       |
|                  | work = work + (2,1,1) = (5,2,4) |    | finish[3] = true P3 ✓ |
| • ...            |                                 |    |                       |

T1:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P <sub>0</sub>	0 1 0	0 0 0	0 0 0
P <sub>1</sub>	2 0 0	2 0 2	
P <sub>2</sub>	3 0 3	0 0 1	
P <sub>3</sub>	2 1 1	1 0 0	
P <sub>4</sub>	0 0 2	0 0 2	

Verifica stato safe  $T_1$ :

- work = (0,0,0)
  - i=0            req[0]=(0,0,0) <= work      OK  
                  work = work + (0,1,0) = (0,1,0)      finish[0] = true      P0✓
  - i=1            req[1]=(2,0,2) <= work      NO
  - i=2            req[2]=(0,0,1) <= work      NO
  - i=3            req[3]=(1,0,0) <= work      NO
  - i=4            req[4]=(0,0,2) <= work      NO

## Ripristino.

L'algoritmo di rilevazione può essere chiamato: dopo ogni richiesta, ogni N secondi, quando l'utilizzo della CPU scende sotto una certa soglia. Se l'algoritmo rileva un deadlock ci sono due possibilità: *uccidere i processi coinvolti* oppure *prelazionare le risorse dei processi coinvolti*.

### **Uccisione dei processi coinvolti.**

L'uccisione dei processi coinvolti in un deadlock può essere effettuata in due modi diversi (costosi):

- *Uccisione di tutti i processi*, in cui tutti i processi sono costretti a ripartire perdendo il lavoro svolto.

- *Uccisione selettiva* fino alla scomparsa del deadlock, in cui si invoca l'algoritmo di rilevazione dopo ogni uccisione, la quale può avvenire in base alla priorità, ai tipi di risorse allocate, a quante risorse mancavano, a quanto tempo mancava alla fine.

### **Prelazione delle risorse.**

Per decidere a quale processo togliere le risorse e in quale ordine, si effettua un *rollback* ad uno stato safe e si riparte dal quello stato. Nel peggio dei casi si avrà un *total rollback*, cioè si ripartirà da zero. Il problema di questa tecnica è che c'è possibilità di *starvation* se tolgo le risorse sempre agli stessi processi.

### ❖ Conclusione

### **Soluzione combinata.**

Ognuno dei tre approcci visti ha vantaggi e svantaggi, perciò nessuno è sempre superiore agli altri. Per risolvere questo problema, si può usare una *soluzione combinata* che consiste in:

- Partizionare le risorse in classi.
- Usare una strategia di ordinamento tra classi di risorse (priorità).
- Definire l'algoritmo più appropriato per ogni classe.

### **Classi di risorse.**

- 1) Risorse interne usate dal sistema (PCB, I/O).
- 2) Memoria.
- 3) Risorse di processo (file).
- 4) Spazio di swap (blocchi su disco).

### **Algoritmi specifici.**

- 1) Prevenzione tramite *ordinamento* delle risorse.
- 2) Prevenzione tramite *prelazione* (tipicamente un job può essere “swappato”).
- 3) Prevenzione *dinamica* (tipicamente la richiesta massima di risorse è nota a priori).
- 4) Prevenzione tramite pre-allocazione violando la condizione di *Hold and Wait* (tipicamente la richiesta massima di memoria è nota a priori).

### **Algoritmo dello struzzo.**

È la soluzione più semplice possibile e consiste nel non fare nulla, perché in fin dei conti: i deadlock si verificano poche volte, la prevenzione è costosa, il ripristino è costoso e gli algoritmi spesso sono sbagliati.

# GESTIONE DELLA MEMORIA

## ❖ Introduzione

### Condivisione della memoria.

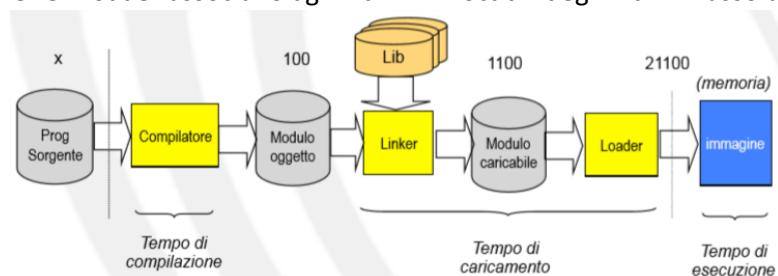
La condivisione della memoria da parte di più processi è essenziale per l'efficienza del sistema. Tuttavia, si hanno *problematiche* relative a: allocazione della memoria ai singoli job; protezione e condivisione dello spazio di indirizzamento; gestione dello swap.

### Trasformazione programma-processo.

Ogni programma deve essere portato in memoria e trasformato in processo per essere eseguito. La trasformazione da programma a processo avviene attraverso varie *fasi*:

- La CPU preleva le istruzioni da eseguire dalla memoria in base al valore del program counter.
- L'istruzione viene codificata e può prevedere il prelievo di operandi dalla memoria.
- Al termine dell'esecuzione dell'istruzione, il risultato può essere scritto in memoria.
- Quando il processo termina, la sua memoria viene rilasciata.

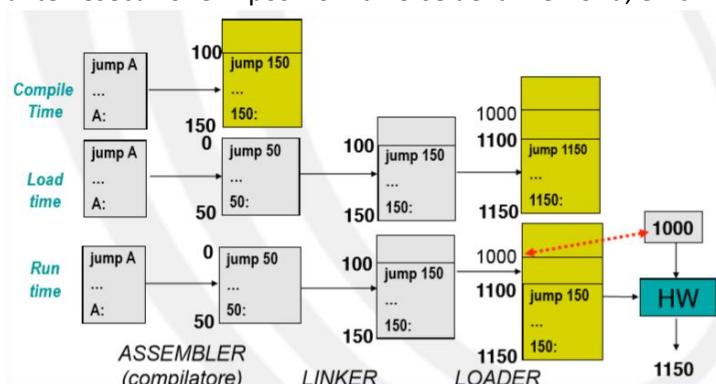
In ogni fase si ha una diversa *semantica degli indirizzi*, infatti, gli indirizzi del programma sorgente sono simbolici e diventano indirizzi fisici grazie al *compilatore*, il quale associa agli indirizzi simbolici degli indirizzi rilocabili. Infine, il *linker* e il *loader* associano agli indirizzi rilocabili degli indirizzi assoluti.



### Binding degli indirizzi.

Il collegamento tra indirizzi simbolici e indirizzi fisici viene detto binding. Il binding di dati e istruzioni in indirizzi di memoria può avvenire in tre momenti distinti:

- Al tempo di compilazione (*compile-time*, statico), quando è possibile generare l'indirizzo assoluto perché è noto a priori in quale parte della memoria risiederà il processo; se la locazione di partenza cambia è necessario ricompilare.
- Al tempo di caricamento (*load-time*, statico), quando è necessario generare codice rilocabile (riposizionabile); siccome si usano gli indirizzi relativi, se cambia l'indirizzo di riferimento, devo ricaricare.
- Al tempo di esecuzione (*run-time*, dinamico), quando il binding viene posticipato perché il processo può essere spostato durante l'esecuzione in posizioni diverse della memoria; è richiesto supporto hardware.



## **Linking.**

È un collegamento che può essere:

- *Statico* (tradizionale), in cui tutti i riferimenti sono definiti prima dell'esecuzione e l'immagine del processo contiene una copia delle librerie usate.
- *Dinamico*, in cui il link è posticipato al tempo di esecuzione e il codice del programma non contiene il codice delle librerie ma solo un riferimento (stub) per poterle recuperare.

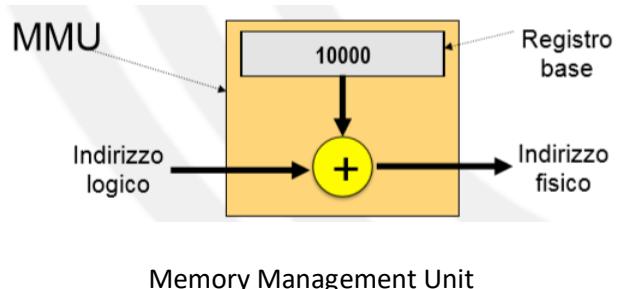
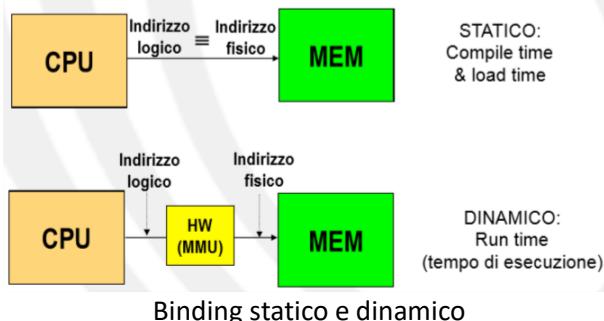
## **Loading.**

È un collegamento che può essere:

- *Statico* (tradizionale), in cui tutto il codice è caricato in memoria al tempo dell'esecuzione.
- *Dinamico*, in cui il caricamento dei moduli è posticipato in corrispondenza del primo utilizzo e il codice non utilizzato non viene caricato (utile per codici con molti casi "speciali").

## **Spazi di indirizzamento.**

Lo spazio di indirizzamento logico è legato ad uno spazio di indirizzamento fisico. L'indirizzo logico (detto anche indirizzo virtuale) è generato dalla CPU, mentre l'indirizzo fisico è quello visto dalla memoria. Nel binding a *compile-time* o a *load-time*, indirizzo fisico e logico coincidono. Nel binding a *run-time*, invece, indirizzo fisico e logico sono generalmente diversi. Infine, in quest'ultimo caso, si ha anche la presenza di un dispositivo hardware chiamato *Memory Management Unit* (MMU), il quale prevede che il valore di un registro base venga aggiunto ad ogni indirizzo logico generato da un processo.



## **Rilocazione.**

In un sistema multi programmato non è possibile conoscere in anticipo dove un processo può essere posizionato in memoria, e per questo il *binding a run-time* non è possibile. Inoltre, l'esigenza di avere lo swap impedisce di poter utilizzare indirizzi rilocati in modo statico, e per questo anche il *binding a load-time* non è possibile. Di conseguenza si deve ricorrere alla rilocazione, che può essere:

- *Dinamica*, in cui la gestione della memoria appartiene totalmente al S.O. (usata in sistemi complessi).
- *Statica*, in cui il S.O. ha una gestione limitata della memoria (usata in sistemi per applicazioni specifiche).

## **Schemi di gestione della memoria.**

Esistono diversi tipi di schemi, ma tutti prevedono che il programma sia interamente caricato in memoria. Nelle soluzioni realistiche, invece, si utilizza la memoria virtuale. Gli schemi sono: **allocazione contigua**; **paginazione**; **segmentazione**; **segmentazione paginata**.

### **❖ Allocazione contigua**

## **Tecniche.**

Nell'allocazione contigua i processi sono allocati in memoria in posizioni contigue all'interno di una partizione. Per cui, la *memoria* deve essere *divisa in partizioni*, che possono essere *fisse o variabili*.

### **Partizioni fisse.**

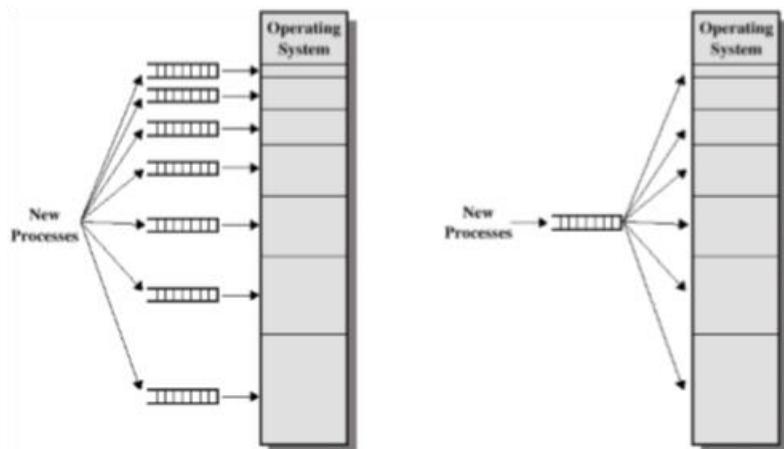
Questa tecnica prevede che la memoria sia un *insieme di partizioni* di dimensioni predefinite (tipicamente diverse). L'assegnazione della memoria è effettuata dallo scheduling a lungo termine e prevede due opzioni: *una coda per partizione; una coda per tutta la memoria*. Nel primo caso, il processo viene assegnato alla partizione più piccola in grado di contenerlo, ma è poco flessibile in quanto possono esserci partizioni vuote e job in attesa in altre code. Nel secondo caso, l'unica coda può essere gestita tramite:

- Politica *FCFS*, che è facile ma causa un basso utilizzo della memoria.
- Scansione della coda *best-fit-only*, che sceglie il job con dimensioni più simili alla partizione.
- Scansione della coda *best-available-fit*, che sceglie il primo job che può stare nella partizione.

Nel caso delle partizioni fisse, la *MMU* contiene dei registri di rilocazione per proteggere lo spazio dei vari processi. Tali registri contengono il valore dell'indirizzo più basso (*registro base*) e il limite superiore dello spazio logico (*registro limite*). Ogni indirizzo logico deve risultare minore del limite.

### **Partizioni fisse – vantaggi e svantaggi.**

- + Relativa semplicità.
- Grado di multiprogrammazione limitato dal numero di partizioni.
- Frammentazione interna, quando la dimensione della partizione è più grande della dimensione del job.
- Frammentazione esterna, quando ci sono partizioni non utilizzate che non soddisfano le esigenze dei processi in attesa.



### **Partizioni variabili.**

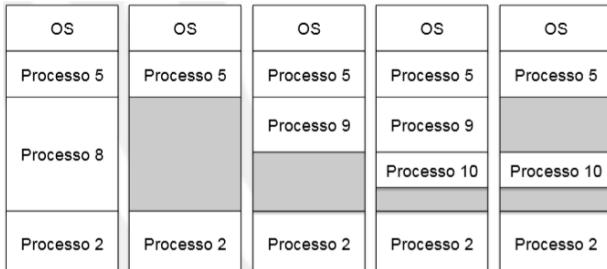
Questa tecnica prevede che la memoria sia divisa in partizioni di dimensioni variabile in base alla *dimensione dei processi*. Si fa ciò per eliminare il problema della frammentazione interna. La memoria, quindi, è vista come un *insieme di buche* (hole), cioè blocchi di memoria disponibili. Quando arriva un processo, si cerca una buca che lo possa contenere e gli viene allocata tale memoria. Per fare questo, il S.O. mantiene informazioni sulle buche e sulle partizioni allocate attraverso una *lista di buche libere*. Inoltre, esistono varie *strategie* per soddisfare le richieste dei processi:

- *First-fit*, che alloca la prima buca grande a sufficienza (è tipicamente la strategia migliore).
- *Best-fit*, che richiede la scansione della lista e alloca la più piccola buca grande a sufficienza.
- *Worst-fit*, che richiede la scansione della lista e alloca la buca più grande.

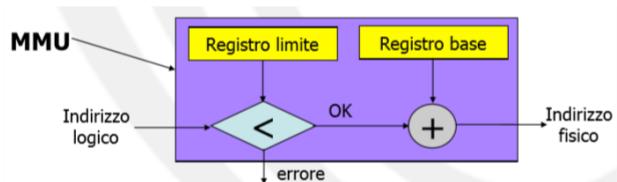
Come per le partizioni fisse, anche nel caso delle partizioni variabili la *MMU* contiene dei registri di rilocazione per proteggere lo spazio dei vari processi.

## Partizioni variabili – vantaggi e svantaggi.

- + Non c'è frammentazione interna per costruzione.
- C'è frammentazione esterna in quanto esiste lo spazio disponibile in memoria, ma non è contiguo.



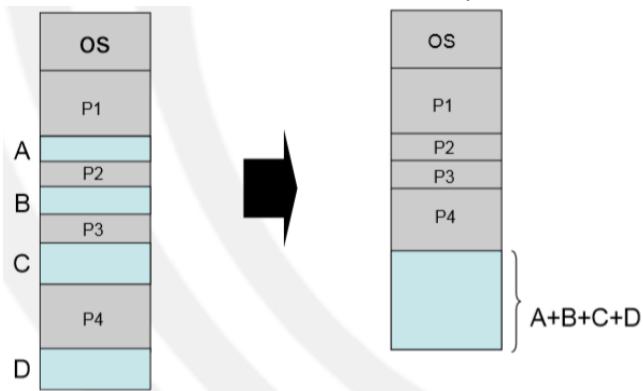
Assegnazione della memoria (partizioni variabili)



MMU (partizioni fisse e variabili)

## Riduzione della frammentazione – compattazione.

La tecnica della *compattazione* è una soluzione intuitiva (ma costosa) per ridurre il problema della frammentazione spostando il contenuto della memoria in modo da rendere contigui i blocchi di un processo. La compattazione è possibile solo se la rilocazione è dinamica, in quanto richiede modifica del registro base.

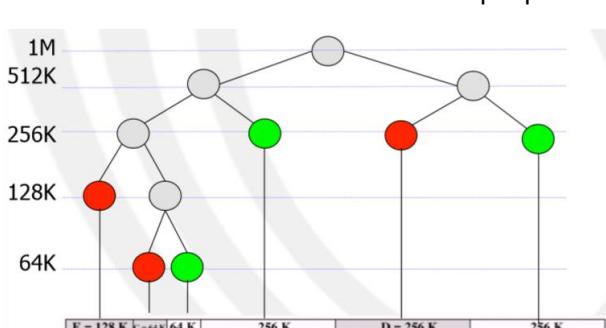


## Riduzione della frammentazione – buddy system.

La tecnica del *buddy system* è un compromesso tra partizioni fisse e variabili, in quanto la memoria è vista come una serie di liste di blocchi di dimensione compresa tra la dimensione del più piccolo blocco allocato e quella del più grande blocco allocato. All'inizio tutta la memoria è disponibile, perciò la lista di blocchi avrà dimensione pari a tutta la memoria. Quando arriva una *richiesta*, si cerca un blocco libero con dimensione adatta (purché sia pari a una potenza di 2): se il blocco esiste, viene allocato; altrimenti, un blocco viene suddiviso in due blocchi di dimensione minore e si ricontrolla. Quando c'è un *rilascio* di memoria, il blocco torna a far parte della lista dei blocchi di dimensione corrispondente e, se si formano due blocchi adiacenti con la stessa dimensione, vengono compattiati per ottenere un unico blocco libero di dimensione maggiore. La *frammentazione interna* esiste ancora, ma è dovuta solo ai blocchi che hanno la dimensione più piccola.

1 Mbyte block				
Request 100 K	A = 128 K	128 K	256 K	512 K
Request 240 K	A = 128 K	128 K	B = 256 K	512 K
Request 64 K	A = 128 K	C = 64 K	B = 256 K	512 K
Request 256 K	A = 128 K	C = 64 K	B = 256 K	D = 256 K
Release B	A = 128 K	E = 64 K	256 K	D = 256 K
Release A	128 K	E = 64 K	256 K	D = 256 K
Request 75 K	E = 128 K	C = 64 K	256 K	D = 256 K
Release C	E = 128 K	128 K	256 K	D = 256 K
Release E		512 K	D = 256 K	256 K
Release D				1 M

Buddy system (esempio)



Buddy system (liste di blocchi)

## ❖ Paginazione

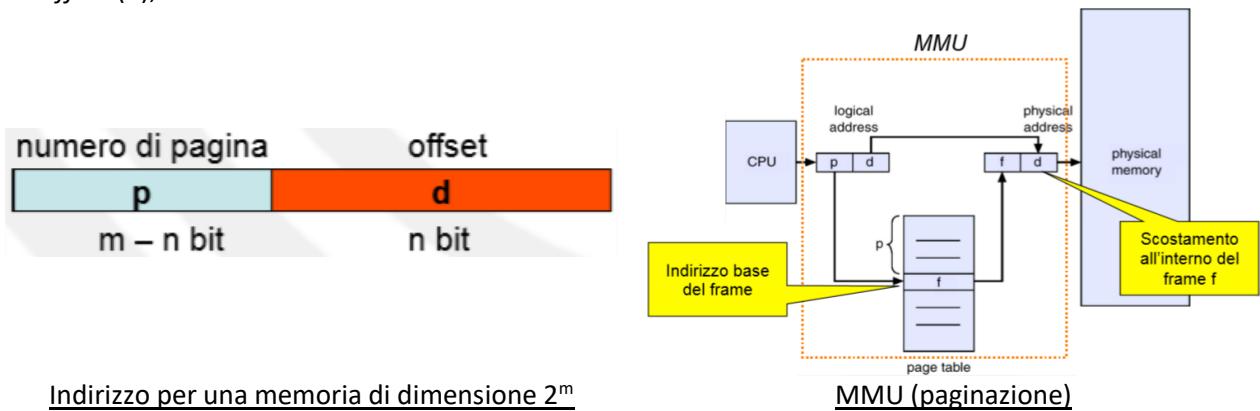
## Frame e pagine.

La paginazione permette di eliminare la frammentazione esterna facendo in modo che lo spazio di indirizzamento fisico di un processo sia non contiguo, e quindi allocando memoria fisica dove essa è disponibile. La memoria fisica è divisa in blocchi di dimensione fissa detti *frame*, mentre la memoria logica è divisa in blocchi della stessa dimensione detti *pagine*. La frammentazione interna esiste ancora, ma solo nell'ultima pagina.

## **Traduzione degli indirizzi.**

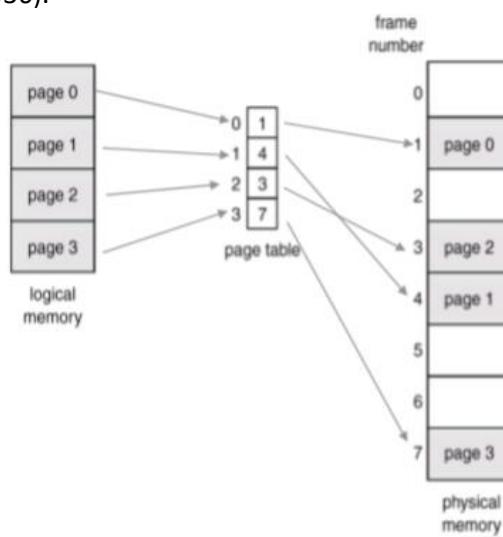
L'indirizzo generato dalla CPU viene diviso in due parti:

- *Numero di pagina* (p), che viene usato come indice nella tabella delle pagine per ottenere l'indirizzo base.
  - *Offset* (d), che combinato con l'indirizzo base definisce l'indirizzo fisico che viene inviato alla memoria.



## **Tabella delle pagine.**

Per mantenere traccia della pagina a cui corrisponde ogni frame si utilizza una tabella delle pagine (*page table*). Quindi, ogni processo avrà la propria tabella delle pagine, la quale verrà usata per tradurre indirizzi logici in indirizzi fisici. Nell'esempio sottostante (memoria da 8KB e pagine da 1KB), l'indirizzo logico 936 diventa l'indirizzo fisico 1960 perché si trova nella *page\_0* che corrisponde al *frame\_1*, e quindi si fa indirizzo base del frame (1024) + offset (936).



## **Implementazione tramite registri.**

In questa implementazione della tabella delle pagine, le entry (righe) sono mantenute nei registri. Questa soluzione è *efficiente* ma è fattibile solo se il *numero di entry* è *limitato* (perché ci sono pochi registri), infatti *allunga i tempi di context switch* poiché richiede il salvataggio dei registri.

### Implementazione in memoria.

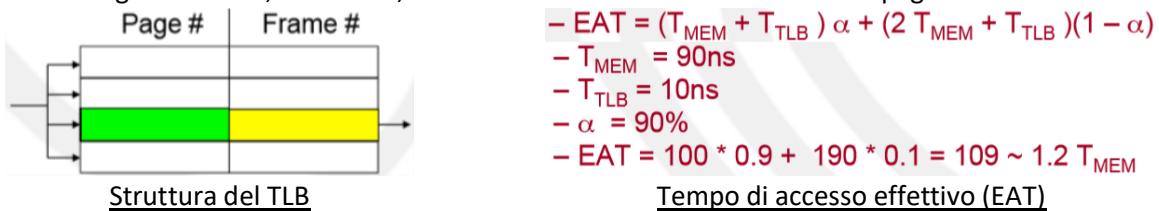
In questa implementazione, la tabella delle pagine risiede in memoria e vengono utilizzati solo due registri:

- *Page-table base register* (PTBR), che punta alla tabella delle pagine.
- *Page-table length register* (PTLR), che contiene la dimensione della tabella delle pagine (opzionale).

Così facendo il *context switch* è più breve perché richiede solo modifica di due registri, ma si ha il problema che ogni accesso ai dati richiede due accessi in memoria, uno per ottenere la tabella delle pagine e uno per ottenere effettivamente il dato.

### Translation Look-aside Buffers.

Il problema del doppio accesso può essere risolto tramite una *cache veloce* detta TLB, la quale confronta l'elemento fornito con il campo chiave (cioè il numero di pagina) di tutte le entry contemporaneamente. Tuttavia, il TLB è molto costoso e quindi all'interno di esso viene memorizzata solo una piccola parte delle entry della tabella delle pagine. Inoltre, ad ogni context switch, il TLB viene ripulito per evitare mapping di indirizzi errati. Quindi, durante un *accesso alla memoria*: se la pagina cercata è nel TLB, esso ritorna il numero di frame con un singolo accesso; altrimenti, è necessario accedere alla tabella delle pagine in memoria.



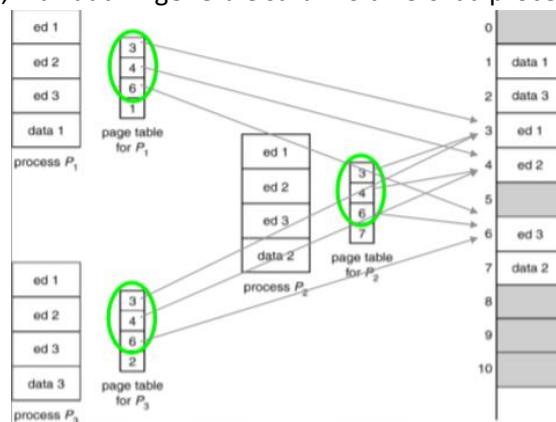
### Protezione.

Nel caso della paginazione, la protezione è realizzata associando dei *bit di protezione ad ogni frame*, come ad esempio:

- *Bit di validità* (valid-invalid bit), che è associato ad ogni entry della tabella delle pagine e indica se la pagina associata è nello spazio di indirizzamento logico del processo o meno.
- *Bit di accesso*, che è usato per dire se una pagina è modificabile/eseguibile o meno.

### Condivisione.

Nel caso della paginazione, se abbiamo un *codice condiviso*, vi sarà un'unica copia fisica e più copie logiche (una per ogni processo). Il *codice read-only*, ovvero quello che non cambia mai durante l'esecuzione, può essere condiviso tra i processi, ma i *dati* in generale saranno diversi da processo a processo.



### Gestione della tabella delle pagine.

Siccome nelle architetture moderne lo spazio di indirizzamento virtuale è molto maggiore dello spazio fisico, sono necessari meccanismi per gestire il problema della dimensione della tabella delle pagine. In particolare, possiamo avere: *paginazione della tabella delle pagine* (multilivello); *tabella delle pagine invertita*.

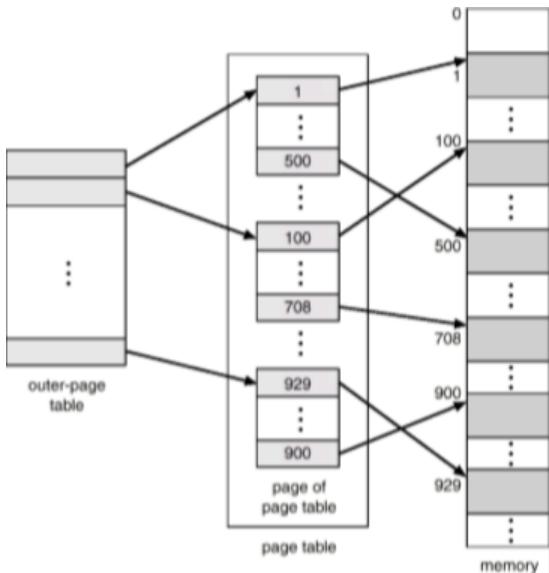
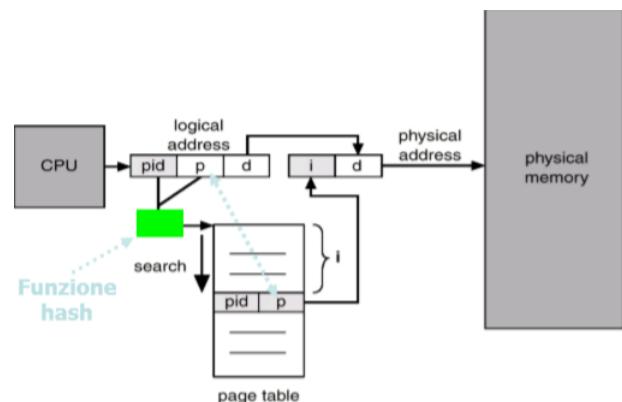


Tabella delle pagine a 2 livelli



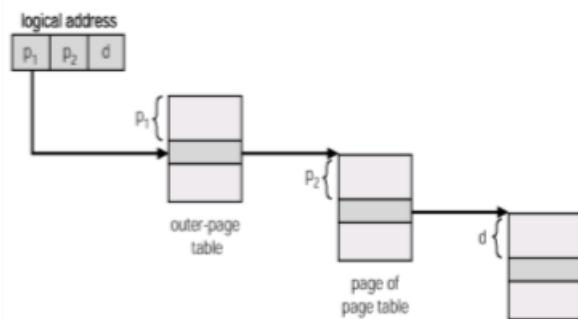
MMU (tabella delle pagine invertita)

### Tabella delle pagine multilivello.

Si tratta di *paginare la tabella delle pagine*. In pratica, solo alcune parti della tabella delle pagine sono memorizzate esplicitamente in memoria, le altre infatti sono su disco. Esistono versioni a 2, 3 o 4 livelli. In un esempio di tabella delle pagine a 2 livelli con indirizzo logico a 32 bit e dimensione della pagina pari a  $2^{12}$ , avremo 12 bit per l'*offset* (*d*) e 20 bit per il *numero di pagina* così suddivisi: 10 bit per l'indice della tabella delle pagine esterna (*p<sub>1</sub>*); 10 bit per l'*offset* all'interno della pagina della tabella delle pagine interna (*p<sub>2</sub>*). Quindi, per *tradurre* un indirizzo logico in un indirizzo fisico bisognerà:

- Recuperare la tabella di secondo livello corrispondente all'indirizzo che si trova alla riga indicata da *p<sub>1</sub>*.
- Recuperare, dalla tabella di secondo livello, l'indirizzo che si trova alla riga indicata da *p<sub>2</sub>*.
- Sommare *d* (offset) al valore fornito dalla tabella di secondo livello.

Siccome ogni livello è memorizzato come una tabella separata in memoria, la traduzione dell'indirizzo logico in quello fisico può richiedere fino ad un massimo di 4 accessi in memoria (nel caso di una tabella delle pagine a 3 livelli). Tuttavia, il TLB mantiene comunque *prestazioni* ragionevoli.



Traduzione degli indirizzi

$$\begin{aligned}
 - \text{EAT} &= (T_{\text{MEM}} + T_{\text{TLB}}) \alpha + (4 T_{\text{MEM}} + T_{\text{TLB}})(1 - \alpha) \\
 - T_{\text{MEM}} &= 90 \text{ ns} \\
 - T_{\text{TLB}} &= 10 \text{ ns} \\
 - \alpha &= 90\%
 \end{aligned}$$

$$\text{EAT} = (10+90)*0,9 + (10+360)*0,1 = 127 \text{ ns} \sim 1.4 \text{ TMEM}$$

Tempo di accesso effettivo (EAT)

### Tabella delle pagine invertita.

Si tratta di avere una *tabella unica nel sistema*, anziché una per ogni processo, avente una entry per ogni frame (pagina fisica). Tale tabella contiene l'indirizzo virtuale della pagina che occupa quel frame e informazioni sul processo che usa quella pagina. Un *problema* di questa soluzione è che più di un indirizzo virtuale può corrispondere ad un certo indirizzo fisico (frame), di conseguenza è necessario cercare il valore desiderato e perciò vi è un aumento del tempo necessario per cercare un riferimento ad una pagina. Tuttavia, tale ricerca non può essere sequenziale per questioni di efficienza. Infatti, si usa l'equivalente di una *tabella hash* per gestire le situazioni in cui diversi indirizzi virtuali corrispondono allo stesso frame.

## ❖ Segmentazione

### Motivazioni e vista logica.

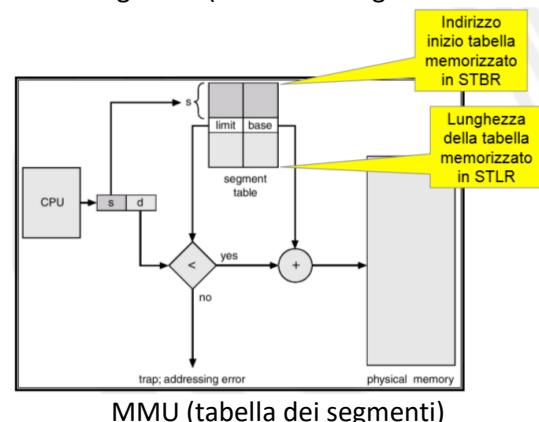
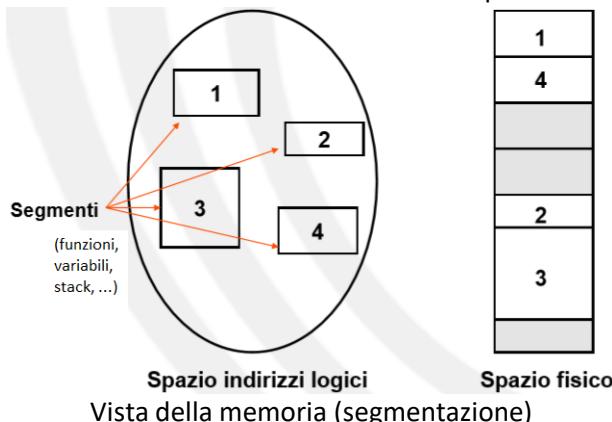
Il motivo alla base dell'introduzione della segmentazione è quello di avere uno schema di gestione della memoria che assomigli alla vista che l'utente ha della memoria stessa. Un programma, quindi, è rappresentato come una collezione di segmenti, cioè di unità logiche (aventi dimensioni differenti tra loro) che rappresentano funzioni, variabili, stack, e così via. Tali segmenti formano lo *spazio degli indirizzi logici*.

### Tabella dei segmenti.

Se si usa la segmentazione, l'*indirizzo logico* è formato da: numero di segmento e offset. Inoltre, esiste una *tabella dei segmenti* (simile alla tabella delle pagine) che mappa gli indirizzi logici bidimensionali in indirizzi fisici unidimensionali. Ogni entry di questa tabella contiene: l'indirizzo fisico di partenza del segmento in memoria (*base*) e la lunghezza del segmento (*limit*). Anche la segment table viene implementata in memoria e prevede che vengano utilizzati due registri:

- *Segment-table base register* (STBR), che punta alla locazione in memoria della tabella dei segmenti.
- *Segment-table length register* (STLR), che indica il numero di segmenti usati da un programma.

Questi registri vengono utilizzati per: controllare se un *indirizzo logico* è *valido* (numero di segmento < STLR); calcolare l'indirizzo dell'elemento da recuperare nella tabella dei segmenti (numero di segmento + STBR).

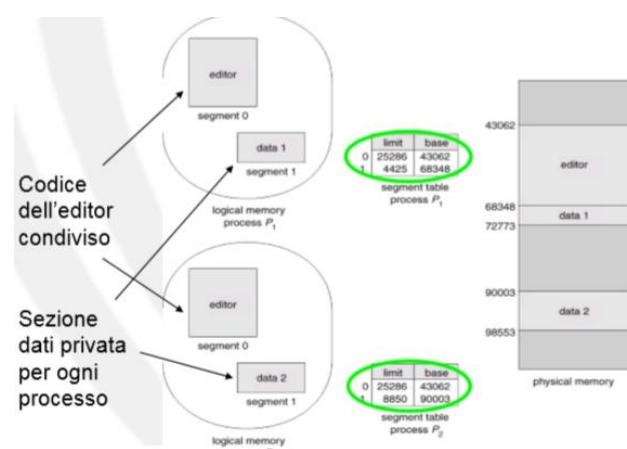


### Protezione e condivisione.

La segmentazione supporta naturalmente la protezione e la condivisione di porzioni di codice, infatti un segmento è un'entità che ha una semantica ben definita. La *protezione* si realizza associando ad ogni segmento: un *bit di modalità* (read/write/execute) e un *bit di validità* (che vale 0 se il segmento è illegale). Per realizzare la *condivisione*, invece, basta inserire le parti del programma che si vogliono condividere (ad esempio le funzioni di libreria) in un segmento.

Tabella dei segmenti		
Segmento	Limite	Base
0	600	219
1	14	2300
2	100	90
3	580	1327

- Indirizzi logici
  - <0, 430>
    - segmento 0, offset 430
      - 430 < 600? OK
    - indirizzo fisico = 430 + base di 0 = 430 + 219 = 649
  - <1, 20>
    - segmento 1, offset 20
      - 20 < 14 NO! indirizzo non valido!



## Frammentazione.

Nel caso della segmentazione, il S.O. deve allocare spazio in memoria per tutti i segmenti di un programma. Siccome tali segmenti hanno *lunghezza variabile*, la loro *allocazione dinamica* avviene tramite algoritmi first-fit o best-fit. Quindi, in particolare nel caso di segmenti con dimensione significativa, può ancora verificarsi il problema della *frammentazione esterna*.

## ❖ Riassunto

### Vantaggi e svantaggi della paginazione.

- + Non esiste frammentazione (a parte una minima frammentazione interna).
- + L'allocazione dei frame non richiede algoritmi specifici.
- C'è una separazione tra la vista utente e la vista fisica della memoria.

### Vantaggi e svantaggi della segmentazione.

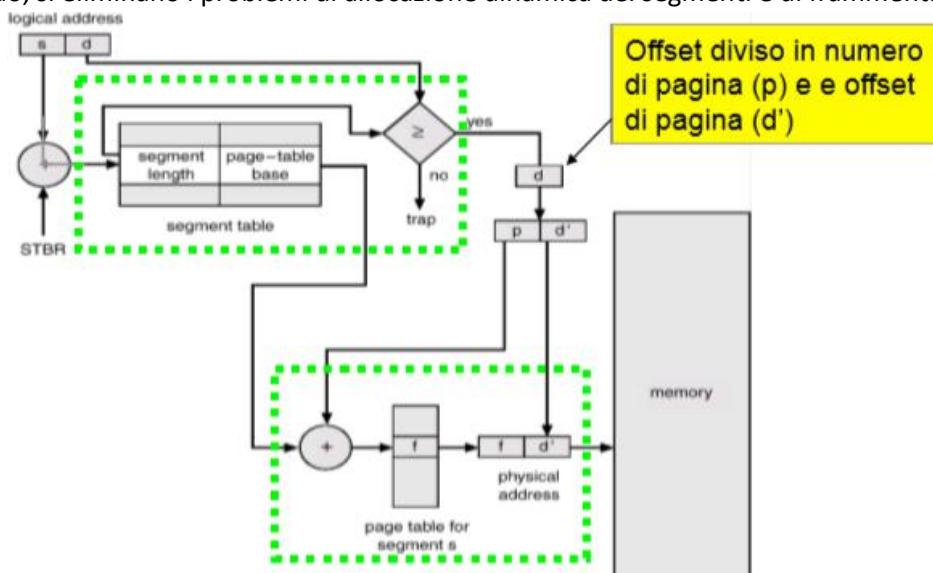
- + C'è consistenza tra vista utente e vista fisica della memoria.
- + Possibilità di proteggere e condividere i segmenti.
- Richiede algoritmi specifici per l'allocazione dinamica dei segmenti.
- Possibilità di frammentazione esterna.

## Segmentazione paginata.

È possibile combinare i due schemi per migliorarli entrambi. La soluzione utilizzata consiste nel *paginare i segmenti* e si realizza in questo modo:

- Ogni segmento è suddiviso in pagine.
- Ogni segmento possiede la sua tabella delle pagine.
- La tabella dei segmenti non contiene l'indirizzo base di ogni segmento, ma l'indirizzo base delle tabelle delle pagine per ogni segmento.

In questo modo, si eliminano i problemi di allocazione dinamica dei segmenti e di frammentazione esterna.



# MEMORIA VIRTUALE

## ❖ Introduzione

### Gestione della memoria.

La caratteristica principale degli schemi per la gestione della memoria visti in precedenza è che l'*intero programma* deve essere caricato in memoria per essere eseguito. In generale, questo non è strettamente necessario, in quanto solo *una parte del programma* può essere caricata in memoria. Di conseguenza lo spazio degli indirizzi logici può essere molto più grande dello spazio degli indirizzi fisici e, così facendo, più processi possono essere mantenuti in memoria.

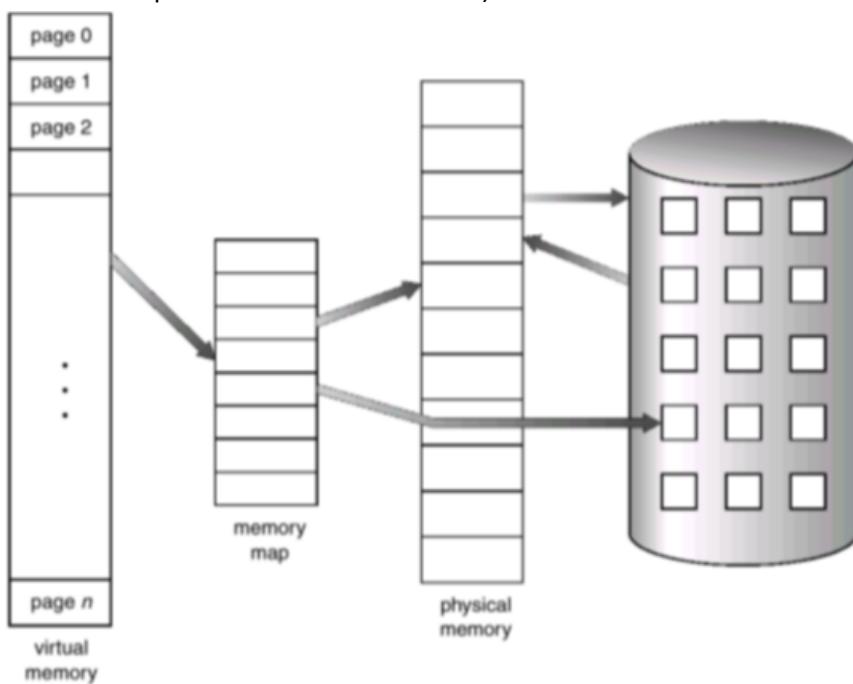
### Memoria virtuale.

Il concetto chiave della memoria virtuale è la possibilità di *swappare* pagine dalla memoria fisica al disco e viceversa, invece che l'intero processo. La memoria virtuale, quindi, permette la separazione della memoria logica (utente) dalla memoria fisica.

## ❖ Paginazione su domanda

### Principi.

La paginazione su domanda è un tipo di *implementazione della memoria virtuale* che si basa sul fatto che una pagina viene caricata in memoria solo quando è necessario. I *vantaggi* di questa implementazione sono che si hanno meno richieste di I/O quando è necessario lo swapping (risposta più rapida) e che si usa meno memoria (più processi hanno quindi accesso alla memoria).



### Stato di una pagina.

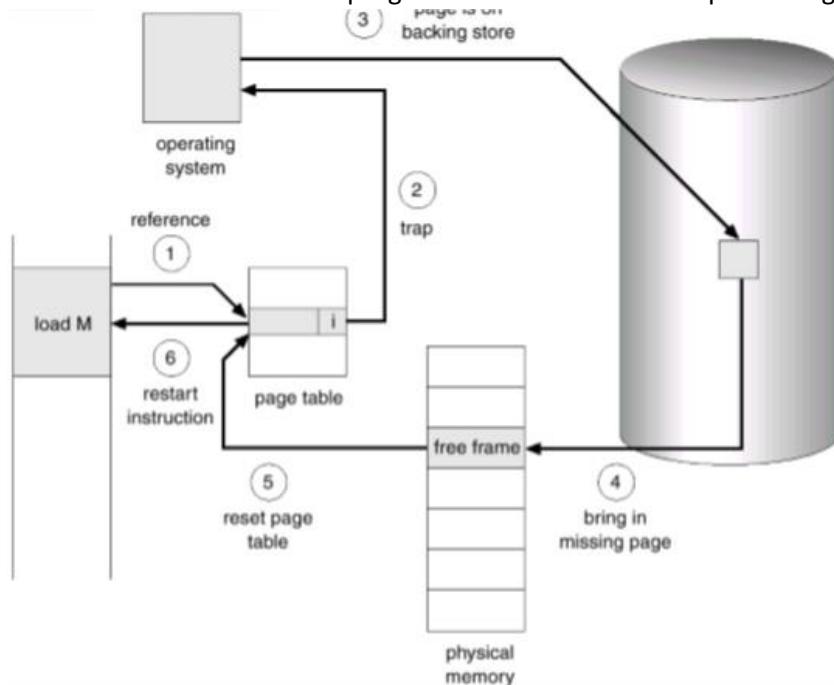
Per realizzare la paginazione su domanda è fondamentale sapere lo stato di una pagina. Quest'ultimo viene mantenuto associando ad ogni entry della page table un *bit di validità* (1 = in memoria, 0 = non in memoria). Inizialmente tutti i bit di validità sono a 0. A questo punto, se durante la traduzione di un indirizzo una entry ha il bit di validità pari a 0, si avrà un *Page fault*.

## Gestione dei Page fault.

Quando si richiede di caricare una pagina, bisogna guardare il bit di validità:

- 1) Se il *riferimento* è valido, si attiva il caricamento della pagina.
- 2) Altrimenti, si avrà un Page fault che causerà un *interrupt*.
- 3) A questo punto, il S.O. deve individuare il *frame del disco* che contiene la pagina da caricare.
- 4) Una volta che l'ha trovato, fa lo swap caricando la pagina in un *frame libero* della memoria.
- 5) Poi, modifica le tabelle aggiornando la *page table* e mettendo il bit di validità a 1.
- 6) Infine, ripristina l'istruzione che ha causato il Page fault.

Ovviamente, il primo accesso in memoria di un programma risulta essere sempre un Page fault.



## Prestazioni.

La paginazione su domanda influenza il tempo di accesso effettivo alla memoria (EAT). Infatti, il tempo di Page fault è dato da 3 componenti principali: servizio dell'interrupt; swap in (lettura della pagina); costo del riavvio del processo. Quindi, per mantenere il peggioramento entro il 10% rispetto al tempo di accesso standard, è fondamentale tenere basso il tasso di Page fault (1 Page fault ogni 10000 accessi).

- $EAT = (1 - p) * t_{mem} + p * t_{page\ fault}$
- Tasso di page fault  $0 \leq p \leq 1$ 
  - $t_{mem} = 100\text{ns}$
  - $t_{page\ fault} = 1\text{ ms} (10^6\text{ ns})$
  - $EAT = (1 - p)*100 + p*10^6 = 100 - 100*p + 1000000*p = 100*(1 + 9999*p)\text{ ns}$

## Rimpiazzamento delle pagine.

Cosa succede se al punto 4 della gestione dei Page Fault *non ci sono frame liberi* in memoria? Il S.O. dovrà rimpiazzare delle pagine tramite un *algoritmo di rimpiazzamento*, il quale sceglie delle pagine in memoria (vittime) e ne fa lo swap sul disco (mettendo il bit di validità a 0). L'obiettivo rimane quello di *ottimizzare le prestazioni* minimizzando il numero di page fault, ma abbiamo appena visto che, in assenza di frame liberi, sono necessari due accessi alla memoria: uno per lo swap out della vittima e uno per lo swap in del frame da caricare. Quindi il risultato è che il *tempo di page fault raddoppia*. Un'ottimizzazione, allora, consiste nell'aggiungere alla page table un *bit di modifica* (dirty bit), che viene messo a 1 quando la pagina viene modificata. A questo punto, solo le pagine che risultano modificate (dirty bit = 1) verranno swappate sul disco se diventano vittime.

## ❖ Algoritmi di rimpiazzamento delle pagine

### Reference string.

Una *problematica* relativa al rimpiazzamento delle pagine è quella della scelta di quale pagina rimpiazzare. Per questo si usano degli algoritmi, detti di rimpiazzamento delle pagine, il cui *obiettivo* è quello di minimizzare il tasso di page fault (che è inversamente proporzionale al numero di frame). Per fare ciò, bisogna eseguire una particolare stringa di riferimenti a memoria (*reference string*) e calcolarne il numero di Page fault. Inoltre, è necessario sapere il *numero di frame* della memoria.

### Algoritmo FIFO (First In First Out).

In questo algoritmo, la prima pagina introdotta è la prima ad essere rimossa. FIFO è un algoritmo “cieco”, in quanto non viene valutata l’importanza della pagina rimossa, ovvero la sua frequenza di riferimento. Inoltre, FIFO tende ad aumentare il tasso di Page fault e soffre dell’*anomalia di Belady*, ovvero che il numero di Page fault può non decrescere all’aumentare del numero dei frame (vedi immagine).

- Reference string

1	2	3	4	1	2	5	1	2	3	4	5
---	---	---	---	---	---	---	---	---	---	---	---

- Con 3 frame 9 page fault

1	1	1	4	4	4	5	5	5	5	5	5
2	2	2	1	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	4	4

- Con 4 frame 10 page fault

1	1	1	1	1	1	5	5	5	4	4
2	2	2	2	2	2	2	1	1	1	5
			3	3	3	3	3	2	2	2
				4	4	4	4	4	3	3

### Algoritmo ideale.

Questo algoritmo garantisce il minimo numero di page fault. L’idea è quella di rimpiazzare le pagine che non verranno usate per il periodo di tempo più lungo. Il problema, però, è come ricavare questa informazione, in quanto si richiede una *conoscenza anticipata* della reference string. La situazione in cui ci si trova è simile a quella che si aveva con l’algoritmo di scheduling SJF, cioè l’*implementazione* è difficile in quanto richiede supporto hardware. Questo algoritmo, quindi, è utile solo come riferimento per gli altri.

### Algoritmo LRU (Least Recently Used).

Questo algoritmo è un’approssimazione dell’algoritmo ideale. L’idea è quella di usare il passato recente come previsione del futuro. Infatti, si rimpiazza la pagina che non viene usata da più tempo (marcata con “\*” nell’esempio sottostante). Questo algoritmo è di solito *migliore del FIFO*, ma *peggiore dell’ideale*. Anche per questo algoritmo, però, rimane sempre il problema dell’*implementazione*, in quanto ricavare il tempo dell’ultimo utilizzo non è un’operazione banale e può richiedere un notevole supporto hardware.

- Esempio: 4 frame → 8 page fault

1	2	3	4	1	2	5	1	2	3	4	5
1	1*	1*	1*	1	1	1	1	1	1	1*	5
2	2	2	2	2*	2	2	2	2	2	2	2
				3	3	3	3*	5	5	5*	4
					4	4	4*	4*	4*	3	3

### Algoritmo LRU – implementazioni hardware.

- *Tramite contatore*: ad ogni pagina è associato un contatore e, ogni volta che la pagina viene referenziata, il clock di sistema viene copiato nel contatore; quindi si rimpiazza la pagina con il valore più piccolo del contatore (che bisogna cercare).

- *Tramite stack*: viene mantenuto uno stack di numeri di pagina e, ad ogni riferimento a una pagina, il numero corrispondente viene estratto e messo in cima allo stack; quindi si rimpiazza la pagina corrispondente al numero che si trova in fondo allo stack (nessuna ricerca della pagina da rimpiazzare).

### Approssimazioni di LRU – bit di reference.

Un'approssimazione dell'algoritmo LRU consiste nell'uso del *bit di reference*. Ad ogni pagina, viene quindi associato un bit (inizialmente a 0) che, quando la pagina è referenziata, viene messo a 1 dall'hardware. Il rimpiazzamento sceglierà una pagina che ha il bit a 0. È un'approssimazione in quanto non viene verificato l'ordine di riferimento delle pagine. In alternativa, posso usare *più bit di reference* per ogni pagina. Tali bit, quindi, si troveranno all'interno di un registro di scorrimento e verranno aggiornati periodicamente. Il rimpiazzamento sceglierà la pagina con il valore del registro di scorrimento più basso.

### Approssimazioni di LRU – second chance.

Un'altra approssimazione dell'algoritmo LRU è il cosiddetto rimpiazzamento *second chance*. Il principio base è quello di una *coda FIFO circolare* basata sui bit di reference. Se il bit vale 0, allora la pagina si rimpiazzava; altrimenti, si mette il bit a 0 ma si lascia la pagina in memoria. Poi si analizzerà la pagina successiva (in ordine circolare e usando la stessa regola) e così via. Anche in questo caso si possono usare più bit di reference.

### Approssimazioni di LRU – conteggio dei riferimenti.

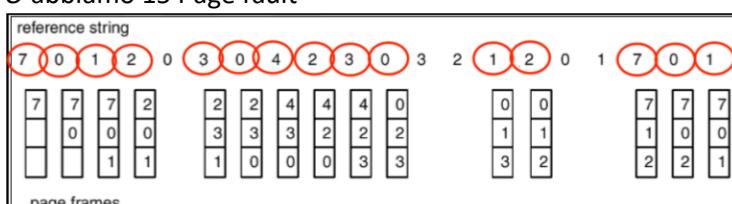
Esistono infine due tecniche (anch'esse approssimazioni dell'algoritmo LRU) basate sul conteggio del numero di riferimenti, che viene mantenuto per ogni pagina:

- *Algoritmo LFU (Least Frequently Used)*, che rimpiazzza la pagina con il conteggio più basso, la quale può non corrispondere alla pagina LRU in quanto, se ho molti riferimenti iniziali, una pagina può avere conteggio alto e non essere eseguita da molto tempo.
- *Algoritmo MFU (Most Frequently Used)*, che è l'opposto di LFU e rimpiazzza la pagina con il conteggio più basso, in quanto è probabilmente stata appena caricata e dovrà essere presumibilmente usata ancora (principio di località dei riferimenti).

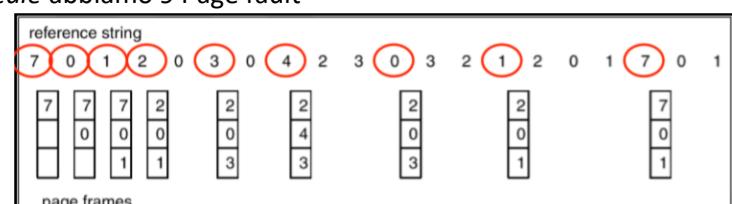
### Confronto degli algoritmi.

Consideriamo il caso di una memoria con 3 frame e applichiamo una certa reference string:

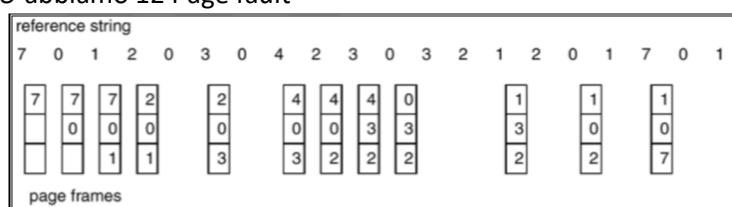
- Con l'*algoritmo FIFO* abbiamo 15 Page fault



- Con l'*algoritmo ideale* abbiamo 9 Page fault



- Con l'*algoritmo LRU* abbiamo 12 Page fault



## ❖ Allocazione dei frame

### Vincoli.

Data una memoria con N frame e M processi, è importante scegliere bene *quanti frame allocare ad ogni processo* al momento dell'esecuzione. Inoltre, bisogna ricordare che ogni processo necessita di un *minimo numero di pagine* per poter essere eseguito. Questo è dettato dal fatto che, se un'istruzione viene interrotta da un Page fault, deve essere fatta ripartire. Di conseguenza, il minimo numero di pagine corrisponde al massimo numero di indirizzi specificabile in una istruzione.

### Schemi di allocazione.

L'*allocazione* dei frame può essere: *fissa*, cioè quando un processo ha sempre lo stesso numero di frame; o *variabile*, cioè quando il numero di frame allocati ad un processo può variare durante l'esecuzione. Inoltre, in caso di page fault, il *rimpiattamento* può essere: *locale*, cioè quando ogni processo seleziona le vittime solo tra i suoi frame; o *globale*, cioè quando un processo sceglie una vittima dall'insieme di tutti i frame (può quindi scegliere i frame di un altro processo). Siccome quest'ultimo migliora il throughput, è più usato rispetto al rimpiazzamento locale.

<i>Contesto</i>	Locale	Globale
<i>Allocazione</i>		
Fissa	X	No
Variabile	X	X

### Allocazione fissa.

L'allocazione fissa può seguire due tipi di approcci:

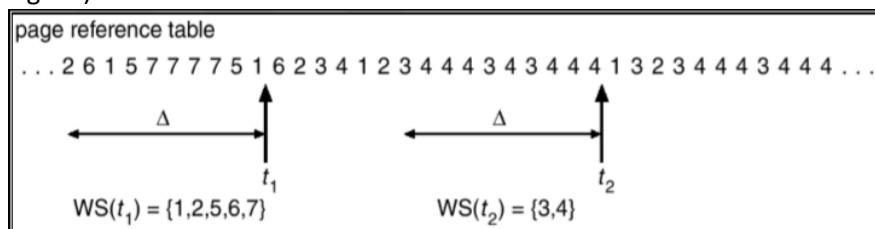
- *Allocazione in parti uguali*, che dati N frame e M processi alloca ad ogni processo  $N/M$  frame.
- *Allocazione proporzionale*, che alloca secondo la *dimensione* del processo (poco significativo) o secondo la *priorità* del processo (più significativo).

### Allocazione variabile.

L'allocazione variabile permette di modificare dinamicamente le allocazioni ai vari processi. Un problema, però, è rappresentato dal definire *in base a cosa* effettuare le modifiche. Due *soluzioni* possono essere: il calcolo del working set e il calcolo della frequenza dei Page fault.

### Modello del working set.

Un criterio per rimodulare l'allocazione dei frame consiste nel calcolare quali sono le *richieste effettive* di ogni processo. In base al modello della località, un processo passa da una località (di indirizzi) all'altra durante la sua esecuzione. Idealmente quindi, un processo necessita di un numero di frame pari alla sua *località*. Per misurarla, viene assegnato ad ogni processo un numero di frame sufficiente a mantenere in memoria il suo working set, ovvero il numero di pagine referenziate nell'intervallo di tempo  $[t-\Delta, t]$  più recente (detto finestra del working set).



### **Thrashing.**

Se la richiesta totale di frame è maggiore del numero totale di frame, si verifica un fenomeno noto come thrashing. Durante questo fenomeno, un processo spende tempo di CPU continuando a swappare pagine da e verso la memoria (*circolo vizioso*). Il thrashing è la conseguenza di un *basso numero di frame*, ovvero se il numero di frame allocati ad un processo scende sotto un certo minimo, il tasso di Page fault tenderà a crescere. Questo porta a:

- *Abbassamento dell'utilizzo della CPU*, in quanto alcuni processi sono in attesa di gestire il Page fault.
- *Aumento, da parte del S.O., del grado di multiprogrammazione* (si aggiungono nuovi processi).
- *Aumento ulteriore del tasso di Page fault*, in quanto i nuovi processi rubano frame ai vecchi processi.

Durante questo circolo vizioso, il throughput precipita. Quindi, è necessario stimare con esattezza il numero di frame necessari ad un processo per non entrare in thrashing.

### **Approssimazione del working set.**

Il working set si può approssimare tramite *timer* e *bit di reference*. Infatti, un timer interrompe periodicamente la CPU e, all'inizio di ogni periodo, i bit di reference vengono posti a 0. Quindi, ad ogni interruzione del timer, le pagine vengono scandite e: quelle con i bit di reference a 1, vengono messe nel working set; quelle con i bit di reference a 0, vengono scartate. L'accuratezza aumenta in base al numero di bit di reference e alla frequenza delle interruzioni.

### **Frequenza dei Page fault.**

Una soluzione alternativa e più accurata è quella di stabilire un *tasso di Page fault accettabile*. Infatti, se il tasso di Page fault effettivo è troppo basso, allora il processo rilascia dei frame perché ne ha troppi; se invece è troppo alto, allora il processo ottiene più frame.

### ❖ Altre considerazioni

#### **Dimensione della pagina.**

La selezione della dimensione della pagina non è semplice. Ciò è dovuto ad una serie di problemi e vincoli:

- Per diminuire la *frammentazione interna*, servirebbero pagine piccole.
- Per diminuire la *dimensione della page table*, servirebbero pagine grandi.
- Per ammortizzare i *costi di lettura e scrittura*, servirebbero pagine grandi.
- Per non dover trasferire anche ciò che non è necessario (*località*), servirebbero pagine piccole.

#### **Struttura dei programmi.**

Anche la struttura dei programmi influisce sul numero di Page fault. Infatti, se consideriamo *due programmi* che iterano su una matrice 1024x1024 e sappiamo che un solo frame è assegnato al processo e che una riga della matrice può essere memorizzata in una pagina, avremo:

– **Programma 1**    for j := 1 to 1024 do  
                      for i := 1 to 1024 do  
                       A[i,j] := 0;  
                       **1024 x 1024 page fault**

– **Programma 2**    for i := 1 to 1024 do  
                      for j := 1 to 1024 do  
                       A[i,j] := 0;  
                       **1024 page fault**

#### **Frame locking.**

Si tratta del blocco dei frame. Infatti, in alcuni casi particolari, esistono *frame che non devono essere (mai) rimpiazzati*. Tali frame possono essere, ad esempio: frame corrispondenti a pagine del kernel; frame corrispondenti a pagine usate per trasferire dati da e verso il sistema di I/O.

# GESTIONE DELLA MEMORIA SECONDARIA

## ❖ Tipologie di supporto

### Nastri magnetici.

Usati per memorizzare dati digitali per la prima volta nel 1951, i nastri magnetici sono formati da una sottile striscia in materiale plastico, rivestita di un materiale magnetizzabile. I nastri magnetici garantiscono la *massima capienza* ma, siccome prevedono un *accesso sequenziale*, sono molto più lenti della memoria principale e dei dischi magnetici. Inoltre, il riposizionamento della testina di lettura richiede decine di secondi e quindi vengono ormai usati solo per scopi di backup. Negli altri ambiti sono stati completamente rimpiazzati dai dischi magnetici e dalle memorie a stato solido.

### Dischi magnetici.

I dischi magnetici sono formati da piatti d'alluminio (o di altro materiale) ricoperti di materiale ferromagnetico. Anche i dischi magnetici garantiscono la *massima capienza*. Inoltre, il loro fattore di forma (diametro) sempre più piccolo, consente di raggiungere velocità di rotazione maggiori. Sui dischi magnetici, sia la lettura che la scrittura avvengono tramite una testina sospesa sulla superficie magnetica: la *scrittura* prevede il passaggio di corrente positiva o negativa attraverso la testina per magnetizzare un'area; la *lettura* prevede il passaggio sopra un'area magnetizzata per indurre corrente positiva o negativa nella testina.

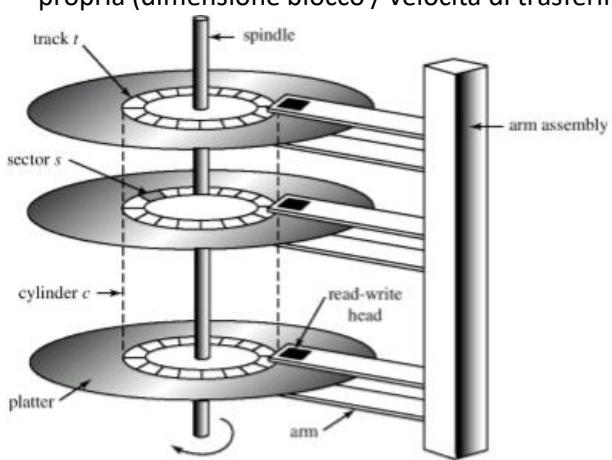
### Struttura fisica di un disco.

La struttura fisica di un disco prevede una serie di superfici (piatti) impilate in modo da formare un *cilindro*. Ogni *piatto* è diviso in *tracce* concentriche, a loro volta suddivise in *settori*. Il settore è la più piccola unità di informazione che può essere letta o scritta su disco e ha dimensione variabile. Generalmente, i settori vengono raggruppati dal punto di vista logico in *cluster* per motivi di efficienza.

### Tempo di accesso a disco.

Per accedere ad un settore del disco bisogna specificare la superficie, la traccia e il settore stesso. Quindi il *tempo di accesso* sarà pari a seek time + latency time + transfer time:

- *Seek time* è il tempo necessario a spostare la testina sulla traccia (da 3 a 15 millisecondi).
- *Latency time* (latenza) è il tempo necessario a posizionare il settore desiderato sotto la testina e dipende dalla velocità di rotazione (0,5 / velocità di rotazione in secondi).
- *Transfer time* è il tempo necessario al settore per passare sotto la testina e rappresenta la lettura vera e propria (dimensione blocco / velocità di trasferimento).



Struttura fisica

- Velocità di trasferimento = 40MB/s
- Velocità di rotazione = 10000 rpm = 166 rps
- Rotazione media =  $\frac{1}{2}$  traccia
- Dimensione blocco = 512 Byte
- Seek time = 5ms
- $T_{\text{accesso}} = 5 \text{ ms} + 0.5/166 + 0.5\text{KB} / 40 \text{ MB}$   
 $= 5 \text{ ms} + 3\text{ms} + 0.0000125 =$   
 $= 8.0000125 \text{ ms}$

Esempio di tempo di accesso

### **Dispositivi a stato solido.**

I dispositivi a stato solido utilizzano dei chip (memorie flash) per memorizzare i dati in modo non volatile. Inoltre, possono rimpiazzare facilmente i dischi fissi in quanto usano la stessa interfaccia. La *capienza* dei dischi a stato solido è generalmente inferiore a quella dei dischi fissi a causa del costo. Tuttavia, rispetto ai dischi fissi, i dischi a stato solido sono:

- *Meno soggetti ai danni* (anche se le memorie flash hanno un limite sul numero di scritture).
- *Più silenziose*, in quanto non si muovono.
- Più efficienti in termini di *tempo di accesso* (0,1 millisecondi).
- Non necessitano di essere *deframmentate*.

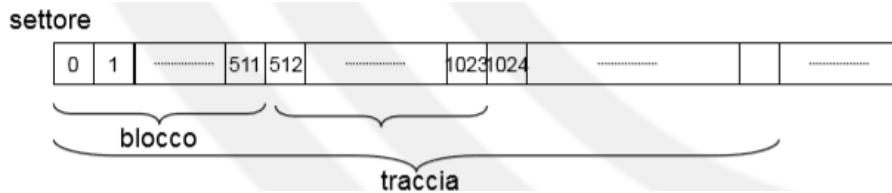
### ❖ Scheduling degli accessi a disco

#### **Considerazioni sul tempo di accesso.**

Nel tempo di accesso a disco, si nota subito che il *seek time* è *dominante*. Quindi, dato il grande numero di processi nel sistema che accedono al disco, è necessario minimizzare il seek time tramite gli *algoritmi di scheduling*. Tali algoritmi ordinano gli accessi al disco in modo da: minimizzare il tempo di accesso totale; ridurre lo spostamento della testina; massimizzare la banda.

#### **Struttura logica di un disco.**

Un disco è logicamente rappresentato come un *vettore* unidimensionale di blocchi logici, dove ogni blocco corrisponde ad un cluster. Tale vettore è *mappato sequenzialmente* e quindi l'indice 0 corrisponde al primo settore del primo cluster della prima traccia del cilindro più esterno. La numerazione degli indici procede gerarchicamente per settori, cluster, tracce, cilindri.



#### **Scheduling degli accessi.**

Un processo che necessita di I/O esegue una *system call*. A questo punto, il S.O. usa la vista logica del disco e di conseguenza vedrà la sequenza degli accessi come una sequenza di indici del vettore. Inoltre, nelle richieste di I/O, vengono incluse altre *informazioni* quali: tipo di accesso (R/W); indirizzo di memoria destinazione; quantità di dati da trasferire.

#### **Algoritmo SSTF (Shortest Seek Time First).**

Questo algoritmo rappresenta la scelta più *efficiente*, in quanto seleziona la richiesta con il minimo spostamento rispetto alla posizione attuale della testina. Questo algoritmo è simile all'algoritmo di scheduling dei processi SJF e perciò è affetto dalla possibilità di *starvation* di alcune richieste. Infine, SSTF migliora l'*algoritmo FCFS*, in cui le richieste vengono processate nell'ordine di arrivo.

#### **Algoritmo SCAN.**

L'algoritmo SCAN (detto anche algoritmo dell'ascensore) si basa sulla *natura dinamica delle richieste*. Infatti, la testina parte sempre da un'estremità del disco e si sposta verso l'altra estremità, servendo tutte le richieste correnti. Una volta arrivato all'altra estremità, riparte nella direzione opposta, servendo le nuove richieste.

#### **Varianti dello SCAN – CSCAN.**

Si tratta di uno *SCAN circolare*. Infatti, si comporta come SCAN ma, quando la testina arriva ad una estremità, riparte immediatamente da 0 senza servire altre richieste. Di fatto il disco è visto come una lista circolare. In

questo algoritmo, il tempo di attesa è più uniforme rispetto a SCAN ma, alla fine di una scansione, ci saranno più richieste all'altro estremo.

#### **Varianti dello SCAN – LOOK e CLOOK.**

In questi algoritmi la testina non arriva sino all'estremità del disco ma, una volta che non ci sono più richieste in quella direzione, *cambia direzione* (LOOK) o *riparte dalla prima traccia* (CLOOK).

#### **Varianti dello SCAN – N-step SCAN.**

In questo algoritmo, per evitare che la testina rimanga sempre nella stessa zona, si partiziona la coda delle richieste in *più code* di dimensione massima N. Se prendiamo un *N grande*, si degenera in SCAN; mentre se prendiamo *N pari a 1*, si degenera in FCFS. A questo punto, quando una coda viene processata per il servizio (SCAN in una direzione), gli accessi in arrivo riempiono le altre code, le quali verranno servite nello SCAN successivo. Infine, va ricordato che, dopo che una coda è stata riempita, non è possibile riordinare le richieste.

#### **Algoritmo LIFO (Last In First Out).**

In certi casi, può essere utile schedulare gli accessi in base all'ordine inverso di arrivo. Questa idea è utile nel caso di accessi con elevata località, ovvero una *serie di accessi vicini*. Anche in questo algoritmo, però, c'è possibilità di *starvation*.

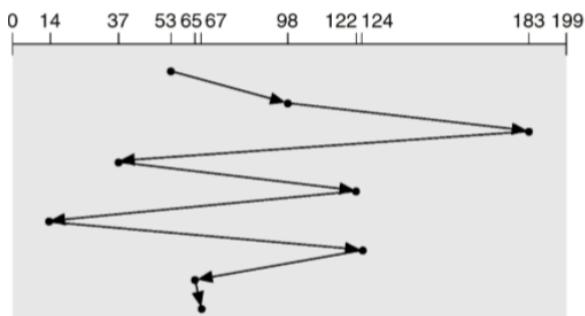
#### **Confronto degli algoritmi.**

Consideriamo il caso di un disco in cui la testina è inizialmente sulla traccia 53 e applichiamo una certa sequenza di accessi:

- Con l'algoritmo *FCFS* abbiamo uno spostamento totale della testina pari a 640 tracce.
- Con l'algoritmo *SSTF* abbiamo uno spostamento totale della testina pari a 236 tracce.
- Con l'algoritmo *CSCAN* abbiamo uno spostamento totale della testina pari a 382 tracce.
- Con l'algoritmo *CLOOK* abbiamo uno spostamento totale della testina pari a 322 tracce.

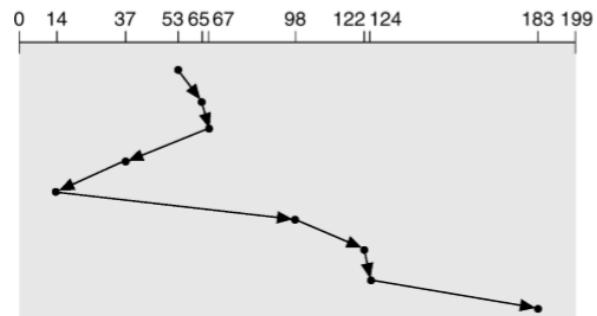
##### Algoritmo FCFS:

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53



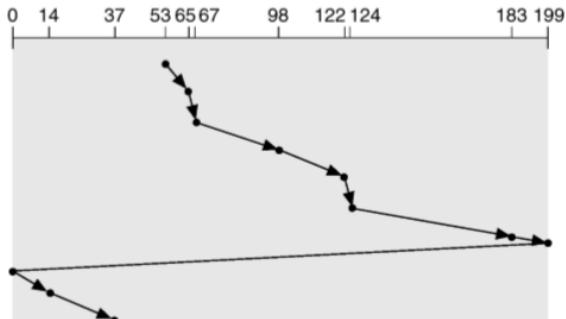
##### Algoritmo SSTF:

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53



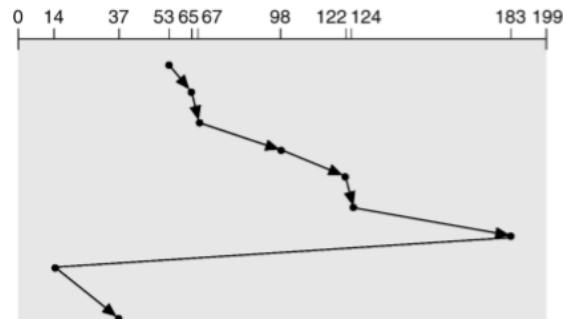
##### Algoritmo CSCAN:

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53



##### Algoritmo CLOOK:

queue = 98, 183, 37, 122, 14, 124, 65, 67  
head starts at 53



## **Analisi degli algoritmi.**

Nessuno degli algoritmi visti è ottimo. Ma comunque un *algoritmo ottimo* sarebbe *poco efficiente*, in quanto il risparmio ottenibile non compenserebbe la complessità di calcolo. Tra i fattori che influenzano l'analisi degli algoritmi troviamo:

- *Distribuzione degli accessi* (numero e dimensioni), in quanto più l'accesso è “grande” più il peso relativo del seek time diminuisce.
- *Organizzazione delle informazioni su disco* (file system), in quanto l'accesso alle directory è essenziale, ma queste sono tipicamente posizionate lontano dalle estremità del disco.

In generale *SCAN* e *CSCAN* sono gli *algoritmi migliori* per sistemi con molti accessi a disco, in cui lo scheduling degli accessi è spesso implementato come modulo indipendente dal sistema operativo.

## ❖ **Gestione del disco**

### **Formattazione dei dischi.**

La *formattazione fisica* (di basso livello) consiste nella divisione del disco in settori che possono essere letti o scritti dal controllore. La *formattazione logica*, invece, consiste nel creare un file system per poter usare il disco come contenitore di file. Per fare questo, il S.O. deve memorizzare le proprie strutture dati sul disco e perciò quest'ultimo viene partizionato in uno o più gruppi di cilindri (*partizioni*). Inoltre, la formattazione logica serve a mantenere sul disco il cosiddetto *programma di bootstrap*, il quale carica i driver dei dispositivi e avvia il sistema operativo. Quando su un *disco nudo* si fa la *formattazione fisica*, tale disco viene suddiviso in settori e, per ognuno, viene aggiunto lo spazio per la correzione di errori (ECC). Quando si fa la *formattazione logica*, invece, alcuni settori vengono riempiti con le strutture dati del file system, ovvero: la lista dello spazio occupato (FAT); la lista dello spazio libero (FL); le directory vuote (DIR).



### **Gestione dei blocchi difettosi.**

L'ECC (Error Correction Code) serve a capire, in lettura o in scrittura, se un settore contiene dati corretti o meno. Infatti, quando il controllore legge un settore, calcola anche l'ECC per i dati appena letti e poi, se il risultato è diverso, segnala un errore dicendo che il blocco è difettoso (*bad block*). Per segnalare tale errore ci sono due metodi di gestione diversi:

- *Gestione off-line* dei bad block, la quale prevede (durante la formattazione logica) di individuare i bad block e metterli in una lista; successivamente si potrà eseguire una utility per isolare i bad block.
- *Gestione on-line* dei bad block, la quale prevede di mappare i bad block su un *blocco di scorta* non ancora usato e opportunamente riservato (*sector sparing*); siccome questa soluzione potrebbe influenzare le ottimizzazioni fornite dallo scheduling, bisogna allocare spare block in ogni cilindro.

### **Gestione dello spazio di swap.**

Per usare il disco come estensione della RAM (memoria virtuale), lo spazio di swap può essere:

- *Ricavato dal normale file system*, per permettergli di utilizzare le primitive di accesso ai file (inefficiente).
- *In una partizione separata* (fuori dal file system), gestita dallo swap daemon (soluzione tipica).

Inoltre, lo spazio di swap può venire *allocato* quando il processo viene creato oppure quando una pagina viene forzata fuori dalla memoria fisica.

# SISTEMI RAID

## **Definizione e obiettivi.**

Il RAID (Redundant Array of Independent Disk) è un sistema per gestire un insieme di dischi che ha l'obiettivo di *migliorare l'affidabilità* e *incrementare le prestazioni*. Le strutture RAID si basano su:

- Copiatura speculare dei dati (*mirroring*), che richiede alta affidabilità e ridondanza (un disco logico corrisponde a due dischi fisici).
- Sezionamento dei dati (*data striping*), che migliora la capacità di trasferimento ma non l'affidabilità.

## **Struttura dei dispositivi RAID.**

Posso avere RAID basati su:

- *Struttura software*, cioè quando più dischi indipendenti sono collegati al bus e il RAID è implementato dal sistema operativo.
- *Struttura hardware*, cioè quando un controllore intelligente gestisce i dischi.
- *Batteria RAID*, cioè quando si ha un'unità a sé stante con cache, batteria e dischi autonomi.

## **Sezionamento dei dati.**

Il sezionamento dei dati può essere:

- *A livello di bit*, in cui i bit di ciascun byte vengono distribuiti su più dischi.
- *A livello di blocco*, in cui i blocchi di un file vengono distribuiti su più dischi.

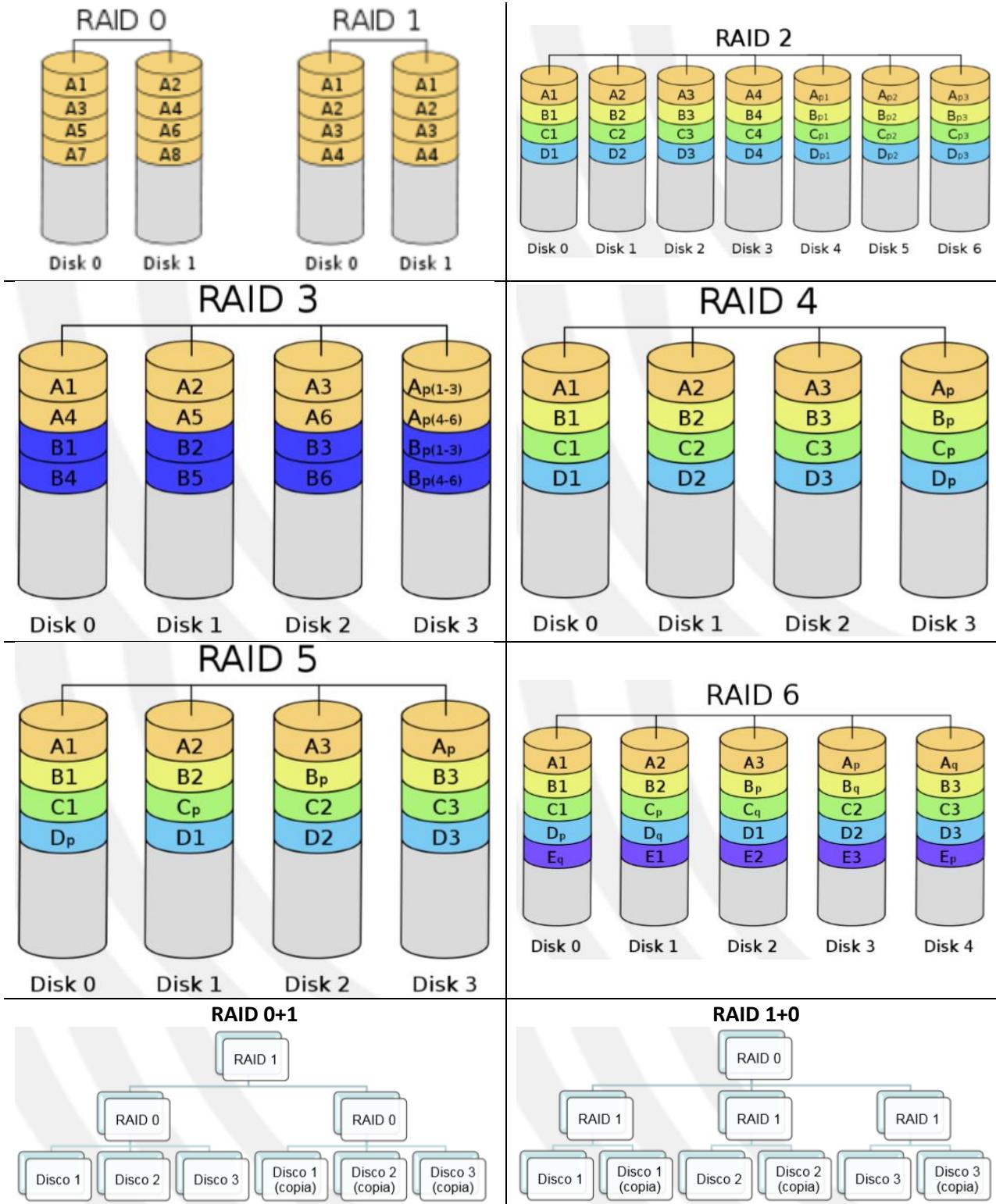
## **Effetti del parallelismo.**

Il parallelismo, tramite il bilanciamento del carico, *aumenta la produttività* per accessi multipli a piccole porzioni di dati e *riduce il tempo di risposta* agli accessi a grandi quantità di dati.

## **Livelli RAID.**

L'utilizzo combinato delle tecniche di mirroring e data striping viene schematizzato dai seguenti livelli RAID:

- *RAID 0*, in cui il sezionamento è a livello di blocco, non c'è ridondanza e si ha un aumento delle prestazioni grazie al parallelismo delle operazioni di lettura/scrittura.
- *RAID 1*, in cui l'affidabilità aumenta linearmente con il numero di dischi (mirroring) e si ha un aumento delle prestazioni in lettura.
- *RAID 2*, in cui il sezionamento è a livello di byte e si utilizzano i codici per la correzione degli errori.
- *RAID 3*, in cui il sezionamento è a livello di byte ma si utilizza un solo disco dedicato al bit di parità.
- *RAID 4*, in cui il sezionamento è a livello di blocco, si utilizza un disco dedicato alla parità e le letture sono più veloci grazie al parallelismo.
- *RAID 5*, in cui il sezionamento è a livello di blocco ma il bit di parità viene distribuito tra tutti i dischi.
- *RAID 6*, che è simile al livello 5 ma grazie alla doppia parità tollera il guasto di due dischi.
- *RAID 0+1*, che sfrutta la velocità del livello 0 implementando la sicurezza come nel livello 1.
- *RAID 1+0*, in cui ogni disco di ogni stripe può guastarsi senza far perdere dati al sistema.



# FILE SYSTEM

## ❖ Interfaccia

### **Componenti.**

Il file system fornisce il meccanismo per l'accesso e la *memorizzazione* di dati e programmi. Il file system consiste di due parti: *collezione di file; struttura di cartelle* (directory).

### **Concetto di file.**

Il S.O. astrae dalle caratteristiche fisiche dei supporti di memorizzazione fornendone una visione logica. Un file è un *insieme di informazioni* correlate identificate da un nome e possiede uno spazio di indirizzamento logico e contiguo. I file possono contenere *dati* (testo, immagini, archivi, ...) oppure essere *eseguibili*.

### **Attributi di un file.**

Sono *informazioni memorizzate* su disco *nella struttura della directory*. Tra gli attributi di un file troviamo: nome, tipo, posizione (cioè il puntatore allo spazio fisico sul dispositivo), dimensione, protezione (che controlla chi può leggere/scrivere/eseguire), data/ora, identificativo dell'utente.

### **Operazioni sui file.**

- *Creazione*: si cerca uno spazio libero sul disco e si inserisce un nuovo elemento nella directory.
- *Scrittura*: si chiama una system call che specifica il nome del file e i dati da scrivere; inoltre, è necessario il puntatore alla locazione della prossima scrittura.
- *Lettura*: si chiama una system call che specifica il nome del file e la posizione in memoria dove mettere i dati letti; inoltre, è necessario il puntatore alla locazione della prossima lettura.
- *Riposizionamento*: si aggiorna il puntatore della posizione corrente.
- *Cancellazione*: si libera lo spazio associato al file e l'elemento corrispondente nella directory.
- *Troncamento*: si mantengono inalterati gli attributi ma si cancella il contenuto del file.
- *Apertura*: si cerca il file nella directory, lo si copia in memoria e si inserisce un riferimento nelle due tabelle dei file aperti (una tabella contiene i riferimenti dei file aperti ed è relativa ad ogni processo; un'altra tabella contiene i dati indipendenti di tutti i file aperti ed è relativa a tutti i processi).
- *Chiusura*: si copia sul disco il file che si trova in memoria.

### **Struttura di un file.**

I tipi di file possono essere adoperati per indicare la *struttura interna del file*, che può essere:

- 1) *Inesistente*, quando il file è visto come una semplice sequenza di byte.
- 2) *A record semplice*, quando il file è visto come un insieme di righe (record) di lunghezza fissa o variabile.
- 3) *Complessa*, nel caso di documenti formattati.

Inoltre, esiste la possibilità di emulare 2 e 3 usando 1, tramite specifici caratteri di controllo.

### **Accesso ai file.**

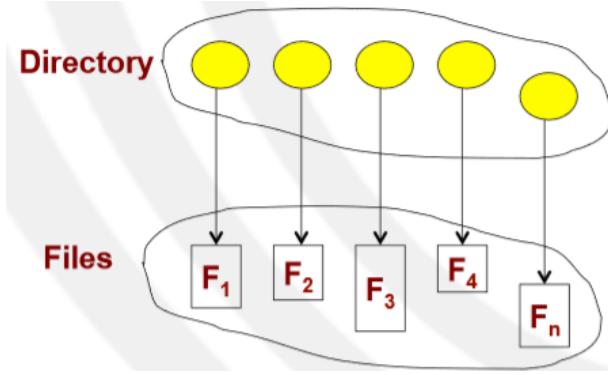
Il *metodo di accesso* ad un file può essere:

- *Sequenziale* (editor, compilatori), che permette operazioni del tipo *read next, write next e rewind*, ma non permette il *rewrite* per evitare il rischio di inconsistenza che si avrebbe se scrivessi qualcosa a metà del file, in quanto potrei cancellare quello che sta dopo.
- *Diretto* (database), in cui il file è visto come una sequenza numerata di blocchi (record) e quindi permette operazioni del tipo *position to N, read N, write N e rewrite N*, oltre a quelle di *read next e write next*.

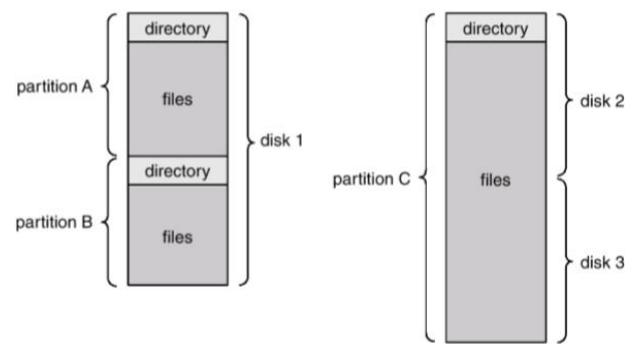
## ❖ Struttura delle directory

### Organizzazione fisica.

Come abbiamo visto, la struttura del file system è determinata da una *collezione di nodi* (directory) contenenti informazioni sui *file* (nome, tipo, lunghezza, ultimo accesso, ultima modifica, possessore). Inoltre, un disco può avere più partizioni e più dischi possono essere uniti per formare un'unica partizione.



Struttura del file system



Dischi e partizioni

### Operazioni sulle directory.

Tra le operazioni che è possibile effettuare sulle directory troviamo: *aggiunta* di un file; *cancellazione* di un file; *visualizzazione del contenuto* della directory; *rinomina* di un file; *ricerca* di un file (tramite una certa espressione); *attraversamento del file system*.

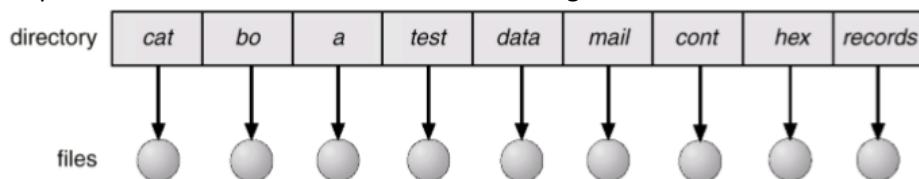
### Organizzazione logica.

Esistono vari tipi di organizzazione logica delle directory, ma tutti devono perseguire i seguenti *obiettivi*:

- *Efficienza*, per garantire un rapido accesso ai file.
- *Nomenclatura*, per permettere agli utenti di usare gli stessi nomi per file diversi avari utenti diversi e nomi diversi per lo stesso file.
- *Raggruppamento*, per garantire la classificazione logica dei file attraverso un criterio (tipo, protezione).

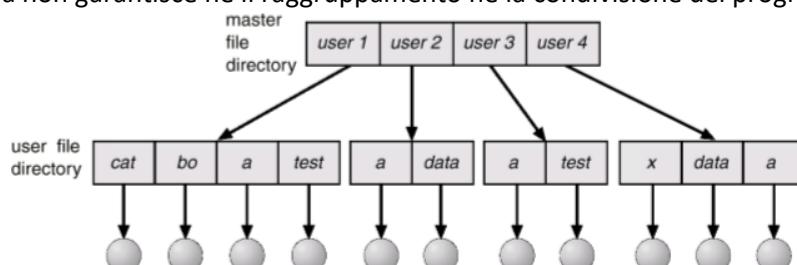
### Directory a un livello.

Prevede una *singola directory per tutti gli utenti*. Di conseguenza, non soddisfa né il raggruppamento né la nomenclatura, in quanto è difficile ricordarsi se un nome esiste già ed è difficile inventare sempre nomi nuovi.



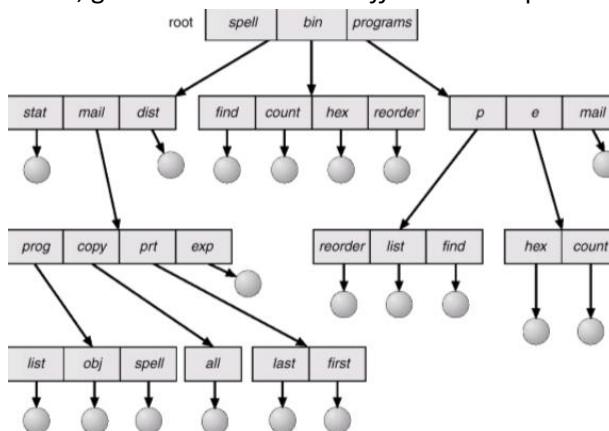
### Directory a due livelli.

Prevede una *directory separata per ogni utente*. Di conseguenza, introduce il concetto di percorso (*path*) e quindi prevede la possibilità di usare lo stesso nome per file avari utenti diversi. Inoltre, garantisce una *ricerca efficiente*, ma non garantisce né il raggruppamento né la condivisione dei programmi di sistema.



### Directory ad albero.

Introduce il concetto di directory corrente (*working directory*) e quindi prevede la possibilità di usare nomi di percorso assoluti o relativi. Inoltre, garantisce una *ricerca efficiente* e la possibilità di *raggruppamento*.



### Directory a grafo aciclico.

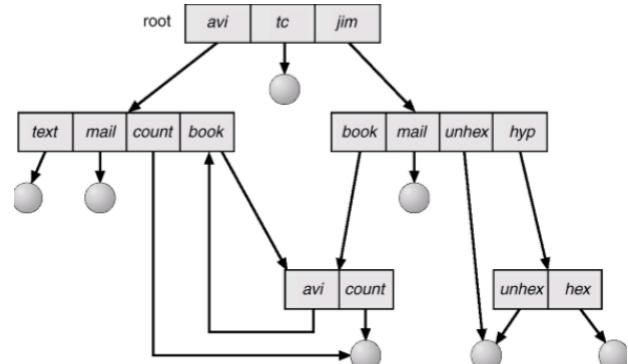
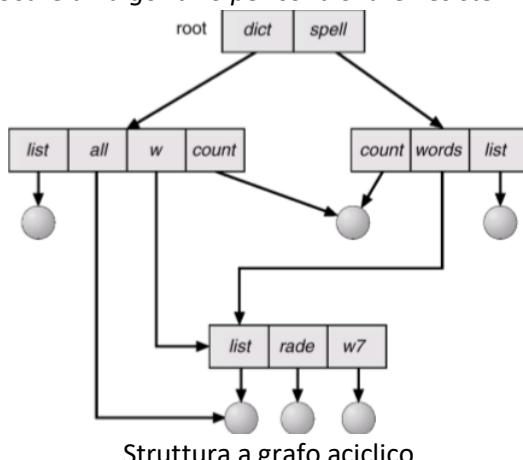
Siccome la struttura ad albero non permette la condivisione di file e directory, si usa la struttura a grafo aciclico. L'implementazione della *condivisione* avviene tramite i collegamenti (link), che possono essere:

- *Link simbolici*, che contengono il path-name del file reale (se cancello il file reale rimangono pendenti).
- *Hard link*, che contengono un contatore di riferimenti il quale viene decrementato ad ogni cancellazione di un riferimento (il file verrà cancellato definitivamente solo se il contatore vale 0).

### Directory a grafo generico.

Se vogliamo passare ad una struttura a grafo generico è necessario garantire che non esistano cicli (per evitare loop infiniti nell'attraversamento). Perché questo avvenga, esistono varie *soluzioni*:

- Non permettere di collegare le directory tra loro.
- Usare il *garbage collector* (costoso).
- Usare un *algoritmo per controllare l'esistenza di cicli* ogni volta che si crea un collegamento (costoso).



### Mount di file system.

Per realizzare *file system modulari*, vi è la possibilità di attaccare (mount) e staccare (unmount), a file system preesistenti, interi file system. In generale, un file system deve essere montato prima di potervi accedere e, in particolare, il punto in cui esso viene montato è detto *mount point*.

### Condivisione e protezione.

La *condivisione*, che è molto importante nei sistemi multiutente, è realizzabile tramite uno schema di protezione. Il possessore di un file, infatti, deve poter controllare cosa è possibile fare su un file

(lettura/scrittura/esecuzione) e anche da parte di chi. Quindi, per realizzare la *protezione*, si è creata una *lista d'accesso* per ogni file e directory, la quale contiene l'elenco di chi può fare che cosa. Poi, siccome si è visto che tale lista può diventare troppo lunga, si è deciso creare delle *classi di utenti*. Questa soluzione è meno generale rispetto alla lista d'accesso, in quanto ci sono solo 3 classi: proprietario; gruppo (cioè gli utenti appartenenti allo stesso gruppo del proprietario); altri. Infine, per ogni classe, i permessi possono essere: r (read); w (write); x (execute). Tali permessi possono essere cambiati mediante il comando *chmod*.

## ❖ Implementazione

### **File system a livelli.**

Per gestire un file system si usano diverse *strutture dati*, di cui una parte stanno su disco e un'altra parte stanno in memoria. Le caratteristiche di queste strutture dati sono fortemente dipendenti dal sistema operativo e dal tipo di file system. Inoltre, esistono caratteristiche comuni a tutte due le tipologie. Tra le strutture che stanno *su disco* si trovano:

- *Blocco di boot*, che contiene le informazioni necessarie per l'avvio del sistema operativo.
- *Blocco di controllo delle partizioni*, che contiene i dettagli riguardanti la partizione quali il numero e la dimensione dei blocchi e la lista dei blocchi liberi.
- *Strutture di directory*, che descrivono l'organizzazione dei file.
- *Descrittori di file* (i-node), che contengono vari dettagli sui file.

Invece, tra le strutture che stanno *in memoria* si trovano:

- *Tabella delle partizioni*, che contiene informazioni sulle partizioni montate.
- *Strutture di directory*, che contengono la copia delle directory a cui si è fatto accesso di recente.
- *Tabella globale dei file aperti*, che contiene le copie dei descrittori di file.
- *Tabella dei file aperti per processo*, che contiene le informazioni di accesso (punta alla tabella globale).

### **Allocazione del disco.**

I metodi di allocazione dello spazio su disco definiscono il modo in cui i blocchi devono essere allocati ai file (o alle directory). Esistono varie *alternative*, ma l'*obiettivo* comune è quello di minimizzare i tempi di accesso e massimizzare l'utilizzo dello spazio.

### **Allocazione contigua.**

Con questo metodo, ogni file occupa un insieme di *blocchi contigui* sul disco. Perciò, la *directory* conterrà solamente l'indirizzo del blocco di partenza e il numero di blocchi. I *vantaggi* di questo metodo sono che:

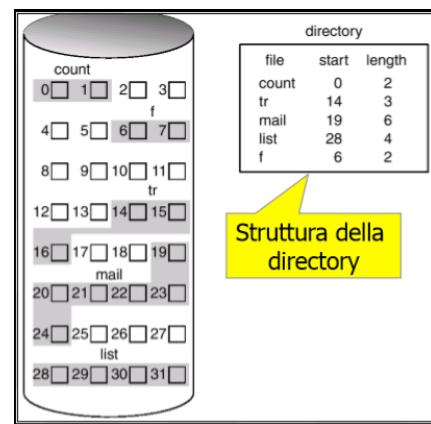
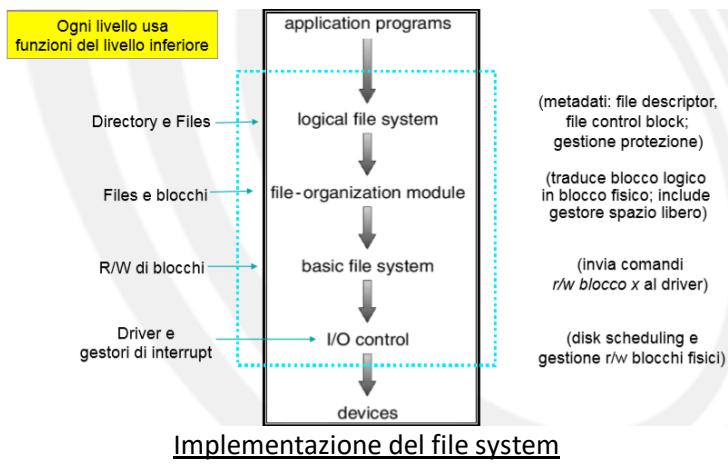
- L'accesso è semplice, in quanto non richiede spostamenti essendo i blocchi contigui sullo stesso cilindro.
- L'accesso sequenziale e l'accesso casuale sono efficienti.

Gli *svantaggi*, invece, assomigliano a quelli dell'allocazione dinamica della memoria e sono:

- Richiesta di algoritmi (best-fit, first-fit, worst-fit).
- Spreco di spazio dovuto alla frammentazione esterna e relativa richiesta di compattazione periodica.
- I file non possono crescere dinamicamente di dimensione, e quindi bisogna usare una soluzione che rallenta il sistema in quanto prevede di rieseguire il programma e trovare un buco più grande dove ricopiare tutto il file ogni volta che il file deve crescere e non c'è spazio.

### **Allocazione contigua – variante.**

Alcuni file system moderni usano uno schema modificato di allocazione contigua basato sul concetto di *extent*, cioè una serie di blocchi contigui su disco (extent-based file system). Questi file system allocano extent anziché singoli blocchi. Quindi, il file è visto come una serie di extent che però, in generale, non sono contigui.



### Allocazione a lista.

Con questo metodo, ogni file è una *lista di blocchi*. Tali blocchi possono quindi essere sparsi ovunque nel disco. Perciò, la *directory* conterrà solo i puntatori al primo e all'ultimo blocco, in quanto ogni blocco contiene il puntatore al blocco successivo. I *vantaggi* di questo metodo sono che:

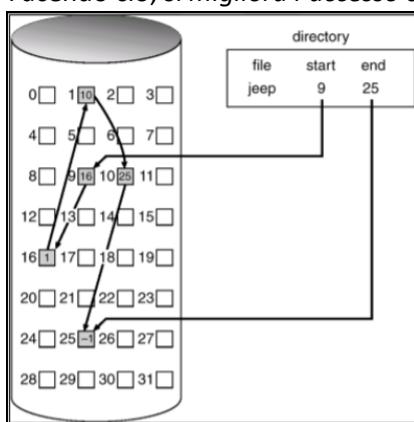
- La creazione di un nuovo file è semplice, in quanto basta cercare un blocco libero e inserire nella *directory* il relativo puntatore.
- L'estensione del file è semplice, in quanto basta cercare un blocco libero e concatenarlo alla fine del file.
- Non c'è nessuno spreco (eccetto lo spazio per il puntatore interno), in quanto potendo usare qualunque blocco elimino il problema della frammentazione esterna.

Gli *svantaggi*, invece, sono che:

- L'accesso casuale non è efficiente, in quanto bisogna per forza scorrere tutti i blocchi a partire dal primo.
- L'efficienza è scarsa, in quanto si richiedono tanti riposizionamenti della testina.
- L'affidabilità è scarsa, in quanto può succedere di prendere il puntatore sbagliato (a causa di una perdita oppure di un errore) e quindi bisogna usare soluzioni che causano overhead, quali liste doppiamente concatenate e salvataggio di nome e numero di blocchi in ogni blocco del file.

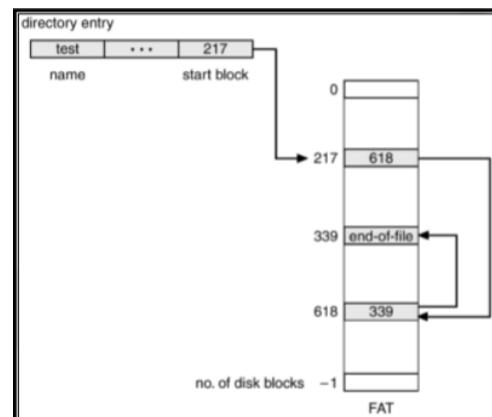
### Allocazione a lista – FAT.

L'allocazione a lista tramite FAT (File-Allocation Table) prevede una *struttura dati* che contiene un elemento per ogni blocco del disco. Gli elementi di questa struttura vengono concatenati per rappresentare i blocchi di un file. Facendo ciò, si *migliora l'accesso casuale*, ma l'efficienza rimane comunque scarsa.



- Indirizzamento
  - X = indirizzo logico
  - N = dimensione del blocco
    - X / (N-1) = numero del blocco nella lista
    - X % (N-1) = offset all'interno del blocco

Allocazione a lista

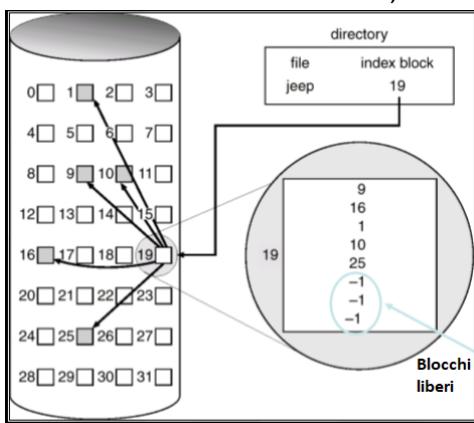


File-Allocation Table

## Allocazione indicizzata.

Con questo metodo, ogni file ha un *blocco indice* (index block) contenente la tabella degli indirizzi (index table) dei blocchi fisici. Perciò, la *directory* conterrà l'indirizzo del blocco indice. I vantaggi di questo metodo sono che l'accesso casuale è efficiente e che non c'è frammentazione esterna, anche se l'overhead del blocco indice per la index table è maggiore di quello richiesto dall'allocazione a lista. Lo svantaggio, invece, è che la dimensione del blocco limita la dimensione del file, in quanto la index table può contenere un numero di indirizzi pari alla dimensione del blocco. Quindi, per avere file senza limiti di dimensione si dovrà usare uno di questi schemi a più livelli:

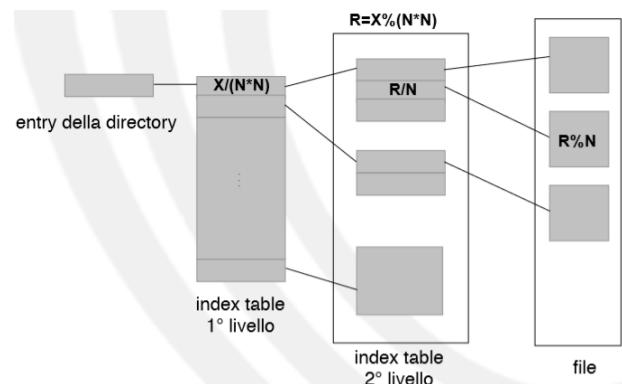
- *Schema con indici multilivello*, dove una tabella esterna contiene i puntatori alle index table interne.
- *Schema concatenato*, in cui si ha una lista concatenata di blocchi indice dove l'ultimo indice punta a un altro blocco indice.
- *Schema combinato*, in cui ogni i-node contiene una parte di indirizzi diretti (allocazione a lista) e un'altra parte di indirizzi con allocazione a 1, 2 e 3 livelli.



### Indirizzamento:

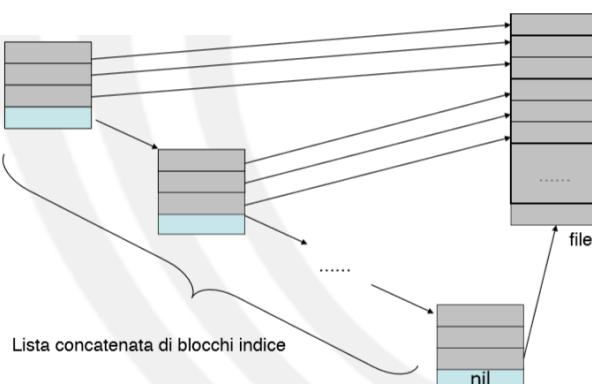
- X = indirizzo logico
- N = dimensione del blocco
  - X / N = offset nella index table
  - X % N = offset all'interno del blocco dati

### Allocazione indicizzata



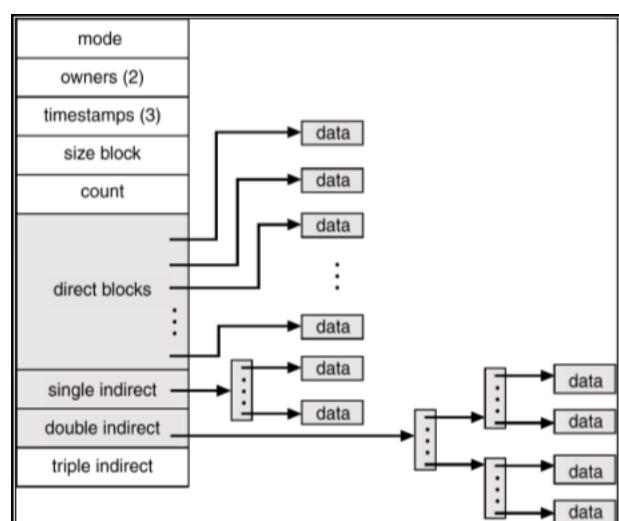
- X = indirizzo logico
- N = dimensione del blocco (in parole)
  - X / (N\*N) = blocco della index table di 1° livello
  - X % (N\*N) = R
- R usato come segue
  - R / N = offset nel blocco della index table di 2° livello
  - R % N = offset nel blocco dati

### Schema con indici multilivello



- X = indirizzo logico
- N = dimensione del blocco (in parole)
  - X / (N(N-1)) = numero del blocco indice all'interno della lista dei blocchi indice
  - X % (N(N-1)) = R
- R usato come segue
  - R / N = offset nel blocco indice
  - R % N = offset nel blocco dati

### Schema concatenato



### Schema combinato

## **Implementazione delle directory.**

Lo stesso meccanismo usato per memorizzare i file si applica alle directory. Le directory non contengono dati, bensì la *lista dei file e delle directory* che essa contiene. Le problematiche riguardano quindi come memorizzare e come accedere a questo contenuto. A tal proposito, si possono individuare due soluzioni:

- *Lista lineare* di nomi di file con puntatori ai blocchi dati, che è facile da implementare ma poco efficiente in quanto lettura, scrittura e rimozione del file richiedono sempre la ricerca del file stesso.
  - *Tabella hash*, che ha il tempo di ricerca migliore ma è anche soggetta a collisioni ovvero situazioni in cui due nomi di file collidono sulla stessa posizione.

#### ❖ Gestione dello spazio libero

## **Lista dei blocchi liberi.**

Per tenere traccia dello spazio libero sul disco si mantiene una lista di blocchi liberi. Per *creare un file* basta cercare blocchi liberi nella lista, mentre per *rimuovere un file* basta aggiungere i suoi blocchi alla lista. Ci sono diversi metodi per gestire lo spazio libero del disco:

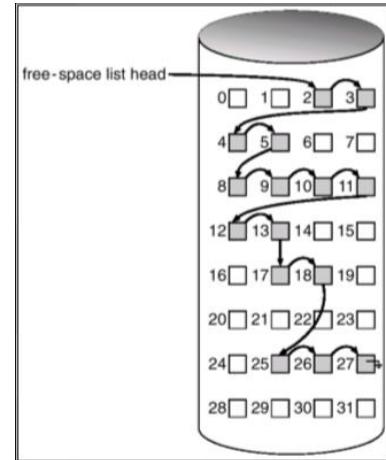
- *Vettore di bit*, in cui ho un bit per blocco (1 = libero, 0 = occupato); il funzionamento prevede che se nel vettore ho un 1 in una parola (es. 00001000) vuol dire che è libera; con questa soluzione posso ottenere facilmente file contigui, ma la mappa dei bit richiede molto spazio.
  - *Lista concatenata* (lista di puntatori ai blocchi liberi), in cui lo spreco è minimo (solo per la testa della lista), ma lo spazio contiguo non è ottenibile.
  - *Raggruppamento*, in cui la lista è concatenata ma fornisce rapidamente un gran numero di blocchi liberi in quanto essi sono raggruppati.
  - *Conteggio*, in cui si mantiene il conteggio di quanti blocchi liberi seguono il primo in una zona di blocchi liberi contigui.

Esempio: n blocchi  
0 1 2 ... n-1

Calcolo del numero del primo blocco libero:

- Cerca la prima parola non 0
  - (<# di bit per parola>) \* (# di parole a 0) + (offset del primo bit a 1)
  - Es: 000000000000000000100001000001111100000000000000000000

### Vettore di bit



## Lista concatenata

#### ❖ Efficienza e prestazioni

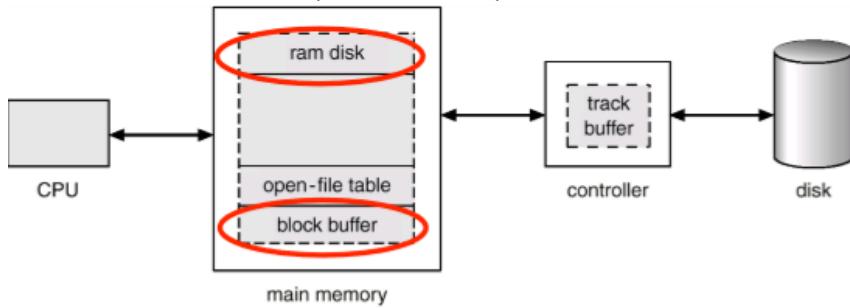
## **Efficienza.**

Nel calcolo dell'efficienza il disco fa da collo di bottiglia, in quanto essa dipende fortemente dall'*algoritmo di allocazione* dello spazio e dal tipo di dati contenuti nella *directory*.

## Prestazioni.

Il controller del disco possiede una piccola cache che è in grado di contenere un'intera traccia ma ciò non basta per garantire prestazioni elevate. Per questo esistono due possibili soluzioni:

- *Dischi virtuali* (RAM disk), in cui parte della memoria RAM viene gestita come se fosse un disco; il driver di un RAM disk accetta tutte le operazioni standard dei dischi eseguendole però in memoria, ciò permette maggiore velocità ma supporta solo i file temporanei (in quanto allo spegnimento si perde tutto).
- *Cache del disco* (detta anche buffer cache), in cui una porzione di memoria RAM memorizza i blocchi usati di frequente; è gestita dal S.O. e sfrutta il principio di località per fare in modo che il trasferimento di dati nella memoria del processo utente non richieda spostamento di byte; le problematiche di questa soluzione riguardano la dimensione e le politiche di rimpiazzamento e scrittura.



### Recupero.

Siccome si potrebbero verificare problemi di inconsistenza tra disco e cache, è bene eseguire un *controllo di consistenza*, cioè confrontare i dati nella directory con i dati sul disco e sistemare le eventuali incongruenze. Inoltre, se si utilizzano programmi di sistema per fare il *backup* del disco, con l'operazione di restore si possono recuperare i file persi.

### File system log structured.

Tali tipologie di file system registrano ogni cambiamento come una *transazione* e scrivono tutte le transazioni su un *log*. Una transazione è considerata avvenuta quando viene scritta sul log, anche se il file system può non essere aggiornato. Infatti, le transazioni sul log vengono scritte nel file system in modo asincrono, e solo quando il file system viene modificato, la transazione viene cancellata dal log. Se il sistema va in crash, le transizioni non avvenute saranno ancora presenti sul log. Usando questa tipologia di file system si ha il vantaggio di ottimizzare il numero di seek.

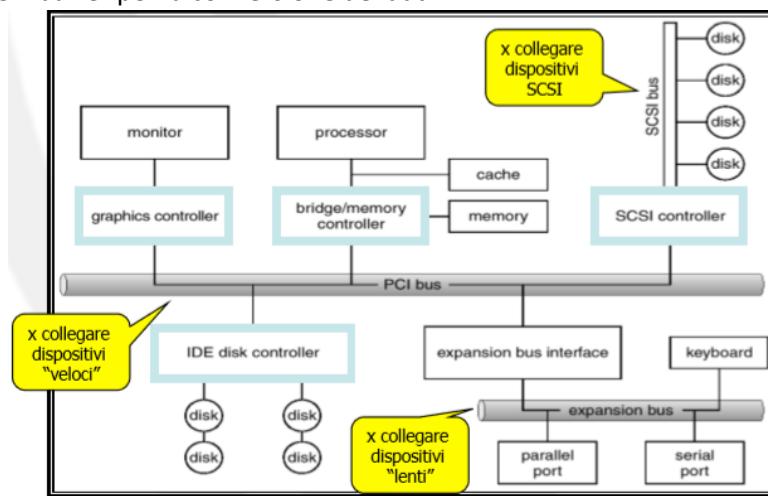
# IL SISTEMA DI I/O

## Obiettivo.

L'obiettivo del sotto-sistema di I/O è quello di fornire ai processi utente un'*interfaccia efficiente e indipendente dai dispositivi*.

## Hardware di I/O.

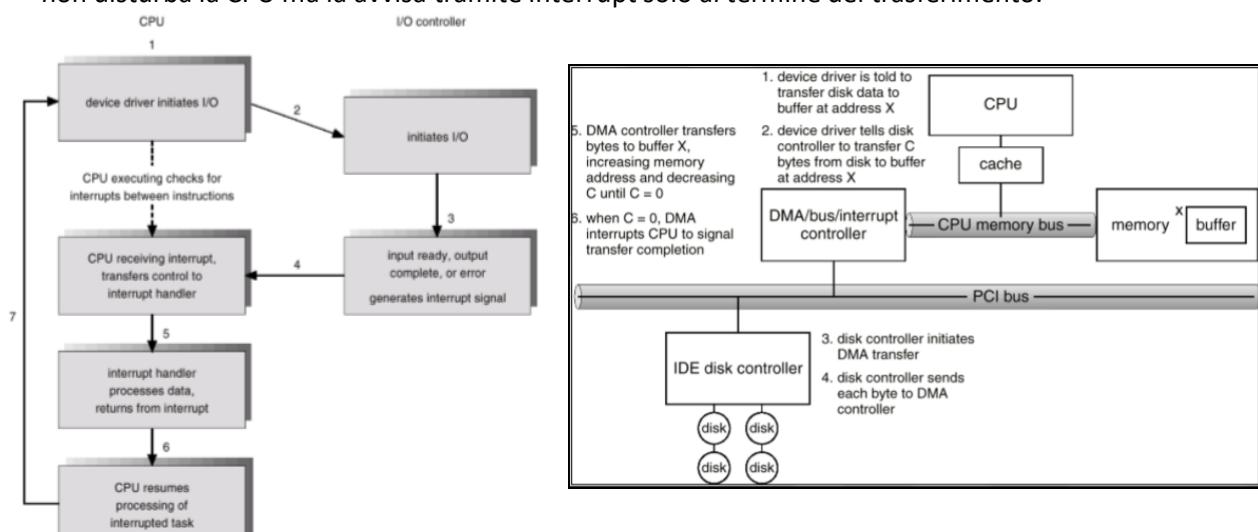
Il controllore dei dispositivi (detto anche *device controller*) è connesso tramite bus al resto del sistema ed è associato ad un indirizzo. Esso contiene i registri per comandare il dispositivo ovvero: il registro di stato, il registro di controllo e il buffer per la conversione dei dati.



## Accesso ai dispositivi di I/O.

Le tecniche di accesso sono:

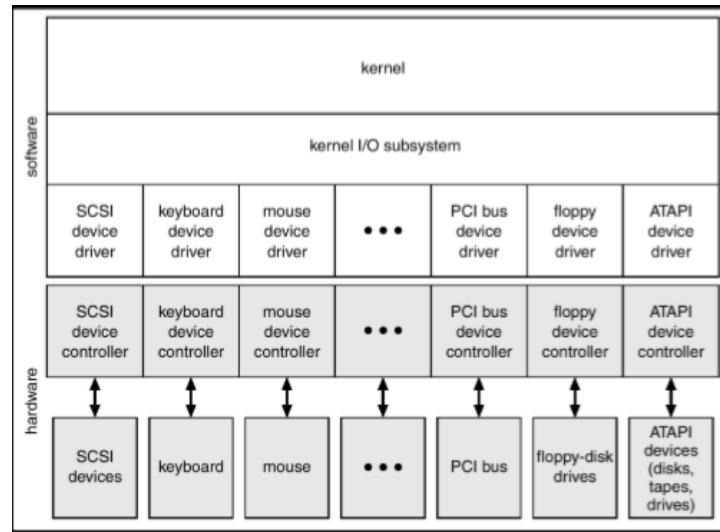
- *Pooling*, che determina lo stato del dispositivo mediante la lettura ripetuta del busy-bit del registro di status; ciò causa uno spreco di CPU.
- *Interrupt*, cioè quando il dispositivo di I/O avverte la CPU tramite un segnale fisico; in questo modo si risolve il busy-waiting ma si creano altre problematiche (gestite dal controllore di interrupt).
- *DMA*, che è stato pensato per grandi spostamenti dati in quanto aggiunge un controllore hardware che non disturba la CPU ma la avvisa tramite interrupt solo al termine del trasferimento.



## Interfaccia di I/O.

Per trattare i differenti dispositivi in maniera standard si utilizzano astrazione, encapsulamento e software layering in modo da nascondere le differenze al kernel del sistema operativo. Quindi si crea un'interfaccia comune con un insieme di funzioni standard (*system call*). Inoltre, le interfacce dei dispositivi possono essere:

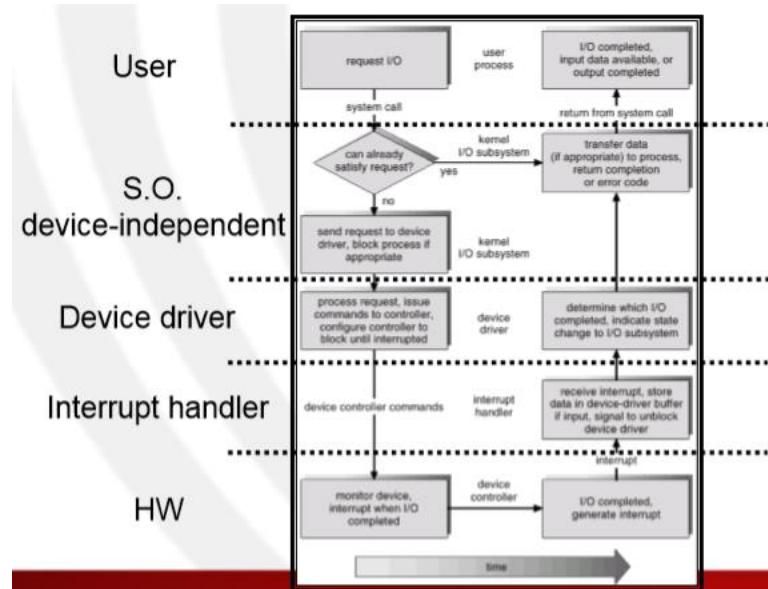
- *A blocchi*, cioè quelle che memorizzano e trasferiscono dati in blocchi e in cui la lettura e la scrittura di un blocco vengono svolte indipendentemente dagli altri.
- *A carattere*, cioè quelle che memorizzano e trasferiscono stringhe di caratteri e in cui non c'è alcun tipo di indirizzamento.



## Software di I/O.

Il software di I/O fa da tramite tra le periferiche e il sistema operativo. I suoi *obiettivi* sono: indipendenza dal dispositivo, notazione uniforme, gestione degli errori, gestione delle operazioni di trasferimento, prestazioni. L'organizzazione dei *livelli di astrazione* è la seguente:

- 1) *Gestori degli interrupt*, che devono essere astratti il più possibile dal resto del S.O. in quanto si occupano del bloccaggio e sbloccaggio dei processi.
- 2) *Device driver*, che traducono le richieste astratte del livello superiore in richieste device-dependent interagendo con i controllori dei dispositivi.
- 3) *SW del S.O. indipendente dal dispositivo*, che svolge le funzioni di naming, protezione, buffering, gestione degli errori, allocazione e rilascio dei dispositivi.
- 4) *Programmi utente*, che tipicamente usano le system call per l'accesso ai dispositivi.



# ESEMPI DI DOMANDE A CROCETTE

## ❖ Introduzione

- Gli obiettivi fondamentali di un sistema operativo sono astrazione ed efficienza: V
  - Astrazione, perché un sistema operativo deve semplificare il più possibile l'uso del sistema.
  - Efficienza, perché un sistema operativo deve essere il più efficiente possibile per permettere l'astrazione del sistema.
- Nei sistemi Batch il tempo di attesa è un parametro fondamentale: F
- I sistemi Batch puntano più alla mole di lavoro che al tempo di risposta: V
- I sistemi Batch sono stati i primi ad introdurre la multiprogrammazione: F
- L'Automatic Job Sequencing è un modulo dei moderni sistemi operativi per gestire i processi: F
  - Nella 2<sup>a</sup> Generazione, c'era un grande spreco della CPU nel caricare i job nel sistema e riavviare l'elaboratore. Si creò quindi l'Automatic Job Sequencing, ossia un programma chiamato "monitor residente" (nella memoria), che si occupava del trasferimento automatico dell'elaboratore da un job all'altro. All'accensione del calcolatore si avviava il monitor, che trasferiva il controllo ad un job; quando esso finiva ritornava il controllo al monitor che faceva eseguire un altro job.
- Il buffering sovrappone operazioni di CPU e I/O di job diversi: F
- Il buffering serve per sovrapporre operazioni di CPU e I/O dello stesso job: V
- Il buffering risolve il problema dello spooling: F
- Lo spooling serve per sovrapporre CPU e I/O dello stesso job: F
- L'utilizzo dello spooling è stato possibile grazie all'introduzione dei nastri magnetici: F
- Polling e interrupt sono due sistemi per accedere ai dispositivi di I/O: V
- DMA e interrupt sono meccanismi alternativi per la gestione delle operazioni di I/O: V
- Il meccanismo di gestione I/O tramite interrupt è stato introdotto per eliminare gli svantaggi del DMA: F
- Il DMA è stato inventato per risolvere i problemi di polling: V
  - Il DMA è stato inventato per risolvere i problemi dell'interrupt e l'interrupt è stato inventato per risolvere i problemi del polling.
- Il DMA è la parte di memoria fisica che viene usata dalla CPU: F
  - Il DMA (Direct Memory Access) è un meccanismo asincrono che permette la sovrapposizione di CPU e I/O sulla stessa macchina.
- Il multitasking è la versione moderna del time sharing: F
- Time sharing è la stessa cosa di multitasking: V
- Multiprogrammazione e multitasking non sono sinonimi per rappresentare lo stesso concetto: V
  - La multiprogrammazione offre un ambiente in cui le risorse del sistema sono impiegate in modo efficiente, ovvero i processi caricati nella RAM vengono eseguiti dalla CPU senza un sistema d'interazione con l'utente.
  - Il multitasking (o time sharing), invece, consente la partizione del tempo d'elaborazione, ovvero la CPU esegue più lavori (job) commutando le loro esecuzioni con una frequenza tale da permettere a ciascun utente l'interazione con il programma durante l'esecuzione, facendo sembrare che i processi siano eseguiti in parallelo.
- Un sistema multitasking funziona solo su architetture multiprocessore: F
- Tutte le operazioni di I/O sono eseguite in modalità protetta (kernel): V
  - Per proteggere l'I/O, tutte le operazioni sono privilegiate e quindi solo il S.O. può accedere alle risorse di I/O, e lo fa tramite delle istruzioni che invocano le system call, ovvero interrupt SW che cambiano la modalità da user a kernel.

- Le system call posso essere usate solo in modalità utente: **F**
- Le system call vengono sempre eseguite in modalità kernel: **V**
- Le system call forniscono un'interfaccia all'utente per l'HW con il passaggio da user a kernel: **V**
- La virtual machine è un particolare sistema operativo che permette di usare istruzioni assembly che non appartengono all'ISA della macchina ospitante: **F**
- La virtual machine permette di eseguire contemporaneamente più copie dello stesso S.O. nella stessa macchina: **V**
- La virtual machine favorisce il time sharing: **V**
- La virtual machine è un sistema monolitico:
  - I sistemi operativi possono essere: monolitici (nessuna gerarchia, componenti tutte allo stesso livello); a struttura semplice (minima organizzazione gerarchica, livelli flessibili); a livelli (organizzazione gerarchica, livelli difficili da definire); virtual machine (approccio a livelli estremizzato); basati su kernel (solo due livelli); client/server (servizi come processi utente).

#### ❖ Gestione dei processi

- In un sistema multi-programmato i processi evolvono sempre in modo concorrente: **V**
- In un sistema multi-programmato i processi vengono eseguiti sequenzialmente: **V**
- I processi non sempre sono eseguiti in modo sequenziale:
  - I processi possono evolvere in modo concorrente, ma vengono sempre eseguiti in modo sequenziale.
- Le variabili globali sono salvate nello Stack: **F**
- L'immagine in memoria di un processo è costituita dalle sezioni PCB, Stack, Codice, Dati, Variabili, Istruzioni: **F**
- L'immagine in memoria di un processo è costituita dalle sezioni Attributi, Stack, Dati e Codice: **F**
  - Il processo ha come immagine: Codice (istruzioni); Dati (variabili globali); Stack (variabili locali, parametri); Heap (memoria dinamica); Attributi (PCB).
- Il PCB contiene le informazioni sullo stato del processo: **V**
- Il PCB di un processo contiene il nome del file dell'eseguibile: **F**
- Il PCB contiene il puntatore allo stato di indirizzamento delle thread associate al processo: **F**
- Un processo è in esecuzione se non è caricato in memoria: **V**
- Un processo durante la sua esecuzione può essere interrotto solo quando esegue operazioni di I/O:
  - Durante l'esecuzione può succedere che un processo: necessita di I/O; termina il suo quanto di tempo; crea un figlio; attende un evento.
- Gli stati di un processo sono: attesa di I/O, esecuzione nell'area di swap: **F**
- L'esecuzione di due processi indipendenti è deterministica ma non riproducibile:
  - Nei processi indipendenti, l'esecuzione è deterministica e riproducibile.
  - Nei processi cooperanti, l'esecuzione non è deterministica e non è riproducibile.
- I moduli del kernel di un sistema operativo sono sempre eseguiti come processi utente ma in modalità protetta: **F**
  - I moduli del kernel possono essere eseguiti in modo diverso in base al sistema adottato (kernel separato, kernel in processi utente, kernel come processo).
- I moduli del kernel di un sistema operativo possono essere eseguiti come processi individuali in modalità protetta: **V**
- Lo Stack dell'immagine del processo è eseguito in modalità kernel: **F**
  - Lo Stack è eseguito in modalità utente, mentre il kernel Stack è eseguito in modalità kernel in quanto gestisce il funzionamento di un processo in modalità protetta.
- Le thread in cui un processo può essere suddiviso condividono lo stesso spazio di indirizzamento: **V**

- Le thread in cui un processo può essere suddiviso non condividono lo stesso spazio di indirizzamento a meno che non siano implementate a livello utente: **F**
- Il tempo di risposta di una thread è sempre superiore al tempo di risposta di un processo: **F**
- Due processi possono condividere una thread: **F**
  - Le thread appartenenti ad un processo, condividono il PCB e lo spazio di indirizzamento del processo stesso. Ogni thread è composta da: TCB; User Stack; Kernel Stack.
- Se si usano thread a livello utente non è necessario passare in modalità utente per passare dall'esecuzione di una thread all'altra quando queste appartengono allo stesso processo: **F**
  - Il kernel ignora le thread a livello utente e quindi non è necessario passare in modalità kernel.
- Le thread a livello kernel hanno gli stessi stati del processo a cui appartengono: **F**
- Se si blocca una thread a livello utente si blocca tutto il processo: **V**
- Se si usano thread a livello kernel il blocco di una thread blocca l'intero processo: **F**
  - A livello kernel, lo scheduling è sulle thread e non sui processi, quindi se una thread si blocca non si blocca anche il processo in quanto viene semplicemente messa in esecuzione un'altra thread.

## ❖ Scheduling

- Lo scheduler a lungo termine controlla il grado di multiprogrammazione: **V**
- Lo scheduler è il modulo del sistema operativo che sceglie il prossimo processo da eseguire: **V**
- Il dispatcher è un modulo del sistema operativo che esegue il cambio di contesto: **V**
- L'operazione di dispatch salva il PCB del processo uscente e passa il sistema in modalità utente: **V**
- Il throughput è dato dalla somma di CPU burst e tempo di risposta: **F**
  - Il throughput è il numero di processi completati per unità di tempo.
  - Il tempo di risposta è il tempo trascorso da quando il processo arriva a quando viene eseguito.
  - Il tempo di attesa è il tempo speso da un processo nella coda di attesa.
  - Il tempo di completamento è il tempo necessario a completare l'esecuzione di un processo.
- L'algoritmo FCFS può portare alla starvation di alcuni processi: **F**
- L'algoritmo HRRN calcola la priorità secondo la formula  $P = 1 + (t_{\text{attesa}} / t_{\text{burst}})$ : **V**
  - Lo scheduling a priorità può portare alla cosiddetta starvation, ovvero al fatto che processi con bassa priorità possano non essere mai eseguiti.
- Quando si usa l'algoritmo Round Robin, l'aumento del quanto di tempo è direttamente proporzionale alla diminuzione del tempo di risposta dei processi: **F**
  - Se il quanto di tempo valesse 2 secondi e i processi entrassero nel sistema ogni 0.5 secondi, il tempo di risposta crescerebbe (perché i context switch diminuirebbero).

## ❖ Sincronizzazione fra processi

- La sezione critica è la porzione di codice dove un processo non deve mai subire interrupt: **F**
- Due processi possono accedere alla mutua esclusione contemporaneamente: **F**
- Se due processi accedono solo in lettura alla stessa variabile non può mai verificarsi una corsa critica: **V**
- Le soluzioni SW al problema della sezione critica fanno uso delle istruzioni atomiche Test&Set o Swap: **F**
- Le soluzioni HW al problema della sincronizzazione dei processi si basano su componenti HW invocati ad hoc dalla CPU per garantire la mutua esclusione tra processi: **F**
- Per evitare il problema delle corse critiche è sufficiente disabilitare gli interrupt: **V**
- L'istruzione Test&Set è atomica: **V**
  - Le soluzioni HW per risolvere il problema delle corse critiche sono: disabilitare gli interrupt mentre una variabile condivisa viene modificata; usare le istruzioni atomiche Test&Set o Swap.
- Le soluzioni HW al problema della mutua esclusione fanno uso di semafori: **F**

- La funzione Test&Set può essere usata per implementare un semaforo: **V**
- L'implementazione dei semafori tramite l'istruzione Swap elimina il problema del busy waiting: **F**
  - I semafori implementati tramite Test&Set o Swap sono basati sul busy waiting.
  - I semafori senza busy waiting mettono il processo nello stato waiting invece di farlo aspettare.
- La variabile associata ad un semaforo può essere incrementata o decrementata tramite opportune primitive atomiche: **V**
- Il valore di un semaforo può essere modificato solo tramite l'uso delle primitive P e V: **V**
- La primitiva V serve per incrementare la variabile di un semaforo: **V**
- Quando un processo bloccato su un semaforo chiama la primitiva Signal (ovvero V), il processo si sblocca se il valore del semaforo diventa positivo: **F**
  - La primitiva V (o Signal) incrementa sempre il valore del semaforo di uno.
  - La primitiva P (o Wait) prova a decrementare il valore del semaforo di uno, altrimenti attende.
- I semafori interi non hanno lo stesso potere espressivo dei semafori binari: **F**
- I semafori binari hanno lo stesso potere espressivo dei semafori interi: **V**
- I semafori vengono inizializzati a 1 per garantire la mutua esclusione tra processi concorrenti oppure a 0 per permettere l'attesa di un evento prima di far avanzare l'esecuzione di un processo: **V**
  - Per la protezione di una sezione critica, si usa un semaforo binario inizializzato a 1.
  - Per l'attesa di evento con 2 operazioni, si usa un semaforo binario inizializzato a 0.
  - Per l'attesa di evento con 1 operazione, si usano due semafori (uno inizializzato a 1 e l'altro a 0).
- Usando i semafori è possibile che si generi un deadlock: **V**
- L'utilizzo di una sincronizzazione basata sui monitor garantisce l'assenza di deadlock: **F**
- Le procedure definite dentro un monitor sono utilizzabili solo in mutua esclusione se il monitor contiene variabili condition: **F**
- Le variabili condition di un monitor possono contenere sia valori interi che booleani: **F**
- I monitor garantiscono implicitamente la mutua esclusione: **V**
  - Le procedure del monitor accedono solo alle variabili definite nel monitor e, siccome un solo processo alla volta può essere attivo in un monitor, la mutua esclusione non va codificata.
  - Per permettere ad un processo di attendere all'interno del monitor è necessario definire variabili di tipo condition accessibili tramite due primitive simili (ma non uguali) a quelle dei semafori.

## ❖ Deadlock

- Quando ogni processo è in attesa di un evento che può essere causato da un processo dello stesso insieme è in deadlock: **V**
- Un deadlock è sempre causato da un errato uso delle primitive di sincronizzazione: **F**
- La condizione di attesa circolare permette di evitare il verificarsi di deadlock: **F**
- Affinché si verifichi un deadlock è sufficiente che sia vera almeno una tra le condizioni di Mutua Esclusione, Possesso e Attesa, Non Prelazione e Attesa Circolare: **F**
- Affinché si verifichi un deadlock devono essere vere contemporaneamente le condizioni di Mutua Esclusione, Possesso e Attesa, Prelazione e Attesa Circolare: **F**
  - Se le condizioni di Mutua Esclusione, Possesso e Attesa, Non Prelazione e Attesa Circolare sono vere contemporaneamente, potrebbe verificarsi un deadlock.
- Un ciclo all'interno di un RAG è sempre l'evidenza del verificarsi di un deadlock: **F**
  - Se il RAG non contiene cicli, non si ha sicuramente il deadlock; altrimenti, potrebbe verificarsi il deadlock (ma se ci sono più istanze dipende comunque dallo schema di allocazione).
- I deadlock possono essere prevenuti effettuando una prelazione forzata delle risorse detenute dai processi: **V**

- Le tecniche di prevenzione statica e dinamica per la gestione dei deadlock riducono eccessivamente l'uso delle risorse: **V**
- Le tecniche per la prevenzione statica dei deadlock garantiscono sempre un uso delle risorse maggiore rispetto agli algoritmi di rilevazione, ma rallentano il sistema: **F**
- Le tecniche per la prevenzione statica dei deadlock garantiscono sempre un uso delle risorse maggiore rispetto agli algoritmi di rilevazione, ma possono portare a starvation: **F**
  - Sia la prevenzione statica che quella dinamica rallentano il sistema (basso utilizzo delle risorse). Con rivelazione e ripristino, invece, si ha un uso maggiore delle risorse.
- Un processo nella prevenzione statica può richiedere risorse solo in ordine crescente/decrescente di priorità: **F**
  - Solamente quando si impedisce l'attesa circolare, un processo può richiedere risorse solo in ordine crescente di priorità.
- Una sequenza di esecuzione è safe se e solo se i processi eseguono rispettando la mutua esclusione: **F**
- Tutte le situazioni di unsafe sono deadlock: **F**
  - Se non esiste una sequenza safe (stato unsafe), possiamo andare in deadlock.
- In un RAG gli archi di reclamo e gli archi di richiesta hanno lo stesso verso: **V**
- In un RAG gli archi di reclamo indicano quali risorse sono assegnate ai processi: **F**
- In un RAG gli archi di reclamo sono indicati con freccia tratteggiata: **V**
- In un RAG gli archi di reclamo collegano risorse con risorse: **F**
- In un RAG gli archi di reclamo collegano processi con processi: **F**
- L'algoritmo del banchiere mantiene sempre il sistema in uno stato safe tranne quando si verifica deadlock: **F**
- L'algoritmo del banchiere mantiene sempre il sistema in uno stato safe: **V**
- L'algoritmo del banchiere rileva la presenza di un deadlock indipendentemente dal numero di istanze di risorse usate dai processi: **F**
- L'algoritmo del banchiere previene il verificarsi di un deadlock perché forza i processi a liberare le risorse per evitare il verificarsi di una attesa circolare: **F**
- L'algoritmo del banchiere ripristina il sistema ad una condizione precedente al verificarsi del deadlock: **F**
- L'algoritmo del banchiere è una tecnica di prevenzione statica dei deadlock: **F**
- L'algoritmo del banchiere necessita di conoscere need: **V**
- Quando uso gli algoritmi del banchiere e di rilevazione c'è un eccessivo spreco di risorse: **F**
- I processi P1 e P2 richiedono al banchiere una risorsa senza ottenerla e poi P3 ne richiede 2 o 3 può ottenerla: **V**
  - Può esserci il caso che P1 e P2 portino ad uno stato unsafe mentre P3, grazie alla sua terminazione, lascia il sistema in uno stato safe.
- L'algoritmo del banchiere garantisce che un sistema possa passare da uno stato safe ad uno unsafe senza cadere in deadlock: **F**
- Gli algoritmi di rilevazione dei deadlock non richiedono di conoscere quale sarà il massimo utilizzo delle risorse che ciascun processo potrà fare: **V**
- Le tecniche di recovery per la risoluzione di deadlock possono essere soggette al fenomeno della starvation di qualche processo: **V**
- L'algoritmo del banchiere è più efficiente dell'algoritmo di rilevazione: **F**
  - Nessuna tecnica è superiore alle altre: la prevenzione è costosa, il ripristino è costoso e gli algoritmi sono spesso sbagliati.

## ❖ Gestione della memoria

- Il binding degli indirizzi può essere effettuato dal compilatore per associare indirizzi logici a quelli fisici: **V**

- La rilocazione dinamica permette ad un processo di essere caricato su locazioni diverse della memoria dopo ogni operazione di swap: **V**
- La presenza di un meccanismo di swap impedisce di effettuare il caricamento statico dei processi: **V**
- La rilocazione dinamica degli indirizzi può essere fatta solo con la paginazione: **F**
- Se la traduzione tra indirizzo simbolico e indirizzo fisico avviene a tempo di caricamento si parla di binding degli indirizzi statico: **V**
- Nel binding a run-time l'indirizzo fisico e logico potrebbero essere diversi: **V**
- Un processo che fa un link statico alle librerie impiega più tempo a caricarsi in memoria rispetto ad un processo che fa un link dinamico: **V**
  - Nel linking statico, l'immagine del processo contiene una copia delle librerie usate.
  - Nel linking dinamico, il link delle librerie è posticipato al tempo di esecuzione.
- L'immagine di un processo creato con binding statico contiene tutto il codice delle librerie usate: **V**
  - Nel loading statico, tutto il codice è caricato in memoria al tempo di esecuzione.
  - Nel loading dinamico, il caricamento del codice si fa in corrispondenza del primo utilizzo.
- L'MMU trasforma indirizzi logici in indirizzi fisici: **V**
- L'MMU utilizza il registro di rilocazione per passare dagli indirizzi logici a quelli fisici: **F**
  - L'MMU prevede che il valore di un registro base venga aggiunto ad ogni indirizzo logico.
- Nella frammentazione esterna esiste lo spazio disponibile in memoria ma non è contiguo: **V**
- La tecnica di allocazione contigua della memoria con partizione fisse è vantaggiosa solo se si usa con la strategia best fit: **F**
- La tecnica delle partizioni variabili risolve il problema della frammentazione interna: **V**
- Best-fit è migliore di first-fit perché ottimizza l'utilizzo della memoria: **F**
  - La strategia first-fit alocca la prima buca libera (è tipicamente la strategia migliore).
  - La strategia best-fit alocca la più piccola buca libera (richiede la scansione della lista).
  - La strategia worst-fit alocca la più grande buca libera (richiede la scansione della lista).
- La compattazione della memoria viene usata per eliminare il problema della frammentazione interna: **F**
- La tecnica del buddy system elimina il problema della frammentazione interna: **F**
- Il buddy system elimina la frammentazione interna ma non quella esterna: **F**
- Il buddy system necessita di tabella delle pagine per effettuare il mapping tra memoria logica e memoria fisica: **F**
  - La tecnica del buddy system per la riduzione della frammentazione esterna è un compromesso tra partizioni fisse e variabili, in quanto alocca buche di dimensione pari ad una potenza di 2 (suddividendo e ricompattando la memoria in base alla dimensione del processo e ai rilasci). La frammentazione interna esiste ancora, ma solo nelle buche di dimensione più piccola.
- La paginazione è una tecnica di allocazione della memoria contigua: **F**
- La paginazione permette che lo spazio di indirizzamento logico di un processo sia non contiguo: **V**
- La presenza di un meccanismo di swap elimina la necessità di usare la paginazione: **F**
- Il numero di pagine della memoria logica è sempre superiore al numero di frame della memoria fisica: **F**
- La memoria logica può essere minore della memoria fisica: **F**
  - Lo spazio di indirizzamento virtuale (memoria logica) è molto maggiore dello spazio fisico.
- Il TLB è una cache lenta che permette di ridurre il numero di page fault: **F**
- La paginazione migliora i tempi di accesso alla memoria rispetto all'allocazione contigua se lo hit ratio del TLB è maggiore del 90%: **F**
- La paginazione migliora i tempi di accesso alla memoria solo se si usa la tabella delle pagine invertita e il TLB: **F**
- Il sistema operativo memorizza una sola tabella delle pagine invertita anche se sono in esecuzione contemporaneamente due o più processi: **V**

- Nella paginazione, per migliorare i tempi di accesso alla memoria, si usa una cache veloce detta TLB che contiene una parte delle entry della tabella delle pagine. Inoltre, quest'ultima tabella può essere implementata anche tramite tabella delle pagine multilivello o invertita.
- La segmentazione è una tecnica di gestione della memoria che non elimina il problema della frammentazione esterna: **V**
- Le dimensioni della memoria logica e fisica sono uguali se si usa la paginazione ma non lo sono se si usa la segmentazione paginata: **F**
- 32 bit sono sufficienti per indirizzare 4 GB di memoria: **V**
- L'algoritmo per il rimpiazzamento delle pagine LRU può essere approssimato usando bit di reference: **V**
- Esiste un massimo numero di frame che devono essere assegnati ad un processo affinché questo possa essere eseguito: **F**
- Il thrashing si verifica quando si sceglie sempre la stessa pagina come vittima a fronte di più page fault:
  - Il thrashing si verifica quando la richiesta totale di frame è maggiore del numero totale di frame. Durante questo fenomeno, un processo spende tempo di CPU continuando a swappare pagine da e verso la memoria (circolo vizioso), in quanto se il numero di frame allocati ad un processo scende sotto un certo minimo il tasso di Page fault tenderà a crescere.
- Il settore è l'unità massima di informazione che può essere letta/scritta su disco: **F**
  - Il settore è la più piccola unità di informazione che può essere letta o scritta su disco.
- La formattazione fisica di un disco suddivide il disco in settori: **V**
  - La formattazione fisica consiste nella divisione del disco in settori che possono essere letti o scritti (viene aggiunto anche uno spazio per la correzione di errori).
  - La formattazione logica consiste nel creare un file system per poter usare il disco come contenitore di file e quindi alcuni settori vengono riempiti con le seguenti strutture dati: lista dello spazio occupato (FAT); lista dello spazio libero (FL); directory vuote (DIR).

## **DOMANDE SULLA TEORIA**

**Si descrivano in dettaglio i seguenti concetti e si spieghi in modo organico come sono correlati tra loro: multiprogrammazione; time-sharing; prelazione; dispatching; scheduling; context switch.**

Lo *scheduling* è il meccanismo di selezione del processo da eseguire nella CPU, che deve garantire:

- *Multiprogrammazione*, per massimizzare l'uso della CPU nel caso di più processi in memoria.
- *Time-sharing* (o multitasking), per commutare frequentemente la CPU tra i processi in modo che ognuno creda di avere la CPU tutta per sé.

Il *dispatching* ha il compito di effettuare il *context switch*, ovvero il passaggio della CPU ad un nuovo processo:

- Salvataggio dello stato (PCB) del vecchio processo e caricamento dello stato del nuovo processo.
- Passaggio alla user mode, in quanto all'inizio della fase di dispatch il sistema è ancora in kernel mode.
- Salto all'istruzione da eseguire nel processo appena arrivato nella CPU.

Per eseguire queste due operazioni, il S.O. dispone di due moduli:

- Il *dispatcher*, che passa il controllo della CPU al processo scelto dallo scheduler.
- Lo *scheduler*, che seleziona (tramite un algoritmo) un processo tra quelli in memoria pronti per l'esecuzione e gli alloca la CPU.

In particolare, gli algoritmi di scheduling possono essere:

- Con *prelazione*, quando il processo che detiene la CPU può essere forzato a rilasciarla prima del termine.
- Senza prelazione, quando il processo che detiene la CPU non la rilascia fino al termine.

**Si introduca il concetto di sezione critica e corsa critica (race condition). Si descrivano quindi i criteri che devono essere soddisfatti per implementare una soluzione al problema della sezione critica. Si descriva infine la differenza tra soluzioni SW e soluzioni HW al problema della sezione critica. Non è necessario descrivere gli algoritmi.**

La *sezione critica* è una porzione di codice in cui si accede ad una risorsa condivisa. La *corsa critica*, invece, è la possibile inconsistenza dei dati che si verificherebbe se più processi possono modificare una variabile contemporaneamente. Per risolvere questo problema, devono essere soddisfatti alcuni *criteri*:

- Mutua esclusione, ovvero un processo alla volta può accedere alla sezione critica.
- Progresso, ovvero solo i processi che stanno per entrare nella sezione critica possono decidere chi entra.
- Attesa limitata, ovvero deve esistere un massimo numero di volte per cui un processo può entrare.

Ci possono essere due tipologie di *soluzioni* al problema della sezione critica:

- Soluzioni software, che aggiungono codice alle applicazioni e non necessitano di supporto hardware.
- Soluzioni “hardware”, che aggiungono codice alle applicazioni ma necessitano di supporto hardware.

Nel secondo caso, infatti, si usano le istruzioni HW atomiche (TestAndSet o Swap) per fare in modo che l'accesso alla risorsa occupi un unico ciclo di istruzione (non interrompibile).

#### **Si descrivano le principali differenze tra semafori e monitor.**

I *semafori* sono variabili intere a cui si accede attraverso due primitive atomiche:

- V, che incrementa il semaforo di 1.
- P, che tenta di decrementare il semaforo di 1; se non si può decrementare è necessario attendere.

I semafori possono essere binari (true-false) o interi (0...), tuttavia essi hanno lo stesso potere espressivo.

Tra i problemi derivanti dall'utilizzo dei semafori troviamo la difficoltà nella scrittura dei programmi e la scarsa visibilità della correttezza delle soluzioni. Un'alternativa è quella di utilizzare i *monitor*, ovvero dei costrutti per la condivisione sicura ed efficiente di dati tra processi. Il programmatore non deve codificare esplicitamente la mutua esclusione in quanto nei monitor è possibile che un solo processo alla volta sia attivo. Per permettere ad un processo di attendere all'interno del monitor è necessario definire variabili di tipo condition accessibili solo tramite due primitive simili (ma non uguali) a quelle dei semafori:

- wait, che blocca il processo fino all'invocazione della sua corrispondente signal da parte di un altro.
- signal, che sveglia esattamente un processo.

#### **Si descriva in dettaglio il concetto di deadlock e si spieghi cosa si intende per stato sicuro (safe).**

Un insieme di processi è in *deadlock* quando ogni processo è in attesa di un evento (rilascio di una risorsa) che può essere causato da un processo dello stesso insieme. Le condizioni necessarie (tutte vere in contemporanea) affinché possa verificarsi un deadlock sono: *mutua esclusione* (risorsa non condivisibile), *hold-and-wait* (un processo detiene una risorsa e attende di acquisirne un'altra), *no-preemption* (le risorse possono essere rilasciate solo volontariamente), *attesa circolare* (i processi attendono ciclicamente il liberarsi di una risorsa). Se anche solo una di queste condizioni non si verifica (obiettivo della *prevenzione statica*), non si avrà mai il deadlock. La *prevenzione dinamica* (per cui è richiesta la conoscenza del massimo numero di istanze di ogni risorsa richiesta da ogni processo) si attua mediante degli algoritmi (con RAG per 1 istanza, del banchiere per N istanze) che lasciano il sistema sempre in uno stato safe, ovvero assegnano una risorsa ad un processo se e solo se si rimane in uno stato safe. Il sistema si trova in uno *stato safe* se esiste una sequenza safe ovvero se, usando le risorse disponibili, il sistema può allocare risorse ad ogni processo (in qualche ordine) in modo che ciascuno di essi possa terminare la sua esecuzione. Se tale sequenza non esiste, siamo in uno stato unsafe (non tutti gli stati unsafe sono deadlock, ma da uno stato unsafe posso andare in deadlock).

**Si descriva l'algoritmo di rilevazione dei deadlock basato sul RAG.**

La prevenzione è conservativa e riduce eccessivamente l'utilizzo delle risorse. Un approccio alternativo è rappresentato dalla cosiddetta *rilevazione e ripristino*. Se abbiamo una sola istanza per ogni risorsa, l'algoritmo di rilevazione consiste nell'analizzare periodicamente il *RAG* per verificare se esistono deadlock (detection) e, in tal caso, iniziare il ripristino (recovery). Il vantaggio di questa tecnica è che non richiede la conoscenza anticipata delle richieste, mentre lo svantaggio è il costo del ripristino che può essere effettuato tramite uccisione (generale o selettiva) dei processi coinvolti oppure tramite prelazione delle risorse (totale o fino allo stato safe) dei processi coinvolti.

**Si descrivano le fasi necessarie per trasformare un programma in processo, si spieghi quindi il concetto di binding degli indirizzi e infine si presentino le differenti tipologie di collegamento (linking) e caricamento (loading).**

Ogni programma deve essere portato in memoria e trasformato in processo per essere eseguito. La trasformazione da programma a processo avviene attraverso varie *fasi*:

- La CPU preleva le istruzioni da eseguire dalla memoria in base al valore del program counter.
- L'istruzione viene codificata e può prevedere il prelievo di operandi dalla memoria.
- Al termine dell'esecuzione dell'istruzione, il risultato può essere scritto in memoria.
- Quando il processo termina, la sua memoria viene rilasciata.

In ogni fase si ha una diversa semantica degli indirizzi, infatti, gli indirizzi del programma sorgente sono indirizzi logici che devono essere trasformati in indirizzi fisici. Il collegamento tra indirizzi simbolici e indirizzi fisici viene detto *binding* e può avvenire in tre momenti distinti:

- Compile-time (statico), quando si sa in quale parte della memoria risiederà il processo.
- Load-time (statico), quando è necessario generare codice rilocabile (indirizzi relativi).
- Run-time (dinamico), quando il processo può essere spostato in posizioni diverse della memoria.

In quest'ultimo caso, si ha anche la presenza di un dispositivo hardware chiamato MMU, il quale prevede che il valore di un registro base venga aggiunto ad ogni indirizzo logico generato da un processo. Il *linking* (collegamento) può essere:

- Statico (tradizionale), se l'immagine del processo contiene una copia delle librerie usate.
- Dinamico, se il codice del programma non contiene il codice delle librerie ma solo un riferimento.

Il *loading* (caricamento) può essere:

- Statico (tradizionale), se tutto il codice è caricato in memoria al tempo dell'esecuzione.
- Dinamico, se il caricamento dei moduli è posticipato in corrispondenza del primo utilizzo.

**In un contesto di allocazione contigua, la strategia worst-fit alloca sempre la buca più grande. Confrontare i vantaggi e gli svantaggi di questa tecnica con quelle first-fit e best-fit.**

Nell'allocazione contigua i processi sono allocati in memoria in posizioni contigue all'interno di una partizione. Per cui, la memoria deve essere divisa in partizioni, che possono essere fisse o variabili. La tecnica delle *partizioni variabili* prevede che la memoria sia divisa in partizioni di dimensioni variabile (buche) in base alla dimensione dei processi (si elimina la frammentazione interna). Quando arriva un processo, si cerca una buca che lo possa contenere e gli viene allocata tale memoria. Per fare questo, il S.O. mantiene informazioni sulle buche e sulle partizioni allocate attraverso una lista di buche libere. Esistono quindi varie strategie per soddisfare le richieste dei processi:

- *First-fit*, che alloca la prima buca grande a sufficienza (è tipicamente la strategia migliore).
- *Best-fit*, che richiede la scansione della lista e alloca, tra quelle grandi a sufficienza, la buca più piccola.
- *Worst-fit*, che richiede la scansione della lista e alloca, tra quelle grandi a sufficienza, la buca più grande.

**Si descriva in dettaglio la tecnica della paginazione mostrando in particolare: motivazioni; meccanismo di funzionamento; schema architettonale per la trasformazione da indirizzo logico a indirizzo fisico; vantaggi e svantaggi rispetto all'allocazione contigua e alla segmentazione; impatto sul tempo effettivo di accesso alla memoria.**

La *paginazione* permette di eliminare la frammentazione esterna facendo in modo che lo spazio di indirizzamento fisico di un processo sia non contiguo, e quindi allocando memoria fisica dove essa è disponibile. La memoria fisica è divisa in blocchi di dimensione fissa detti *frame*, mentre la memoria logica è divisa in blocchi della stessa dimensione detti *pagine*. La frammentazione interna esiste ancora, ma solo nell'ultima pagina. L'*indirizzo logico* viene diviso in due parti:

- Numero di pagina (*p*), che viene usato come indice nella tabella delle pagine per ottenere l'indirizzo base.
- Offset (*d*), che combinato con l'indirizzo base definisce l'indirizzo fisico che viene inviato alla memoria.

*Vantaggi e svantaggi* rispetto alle altre tecniche:

- + Non esiste frammentazione esterna.
- + L'allocazione dei frame non richiede algoritmi specifici.
- C'è una minima frammentazione interna.
- C'è una separazione tra la vista utente e la vista fisica della memoria.

Infine, per limitare l'impatto sul tempo effettivo di accesso alla memoria (*EAT*) si può implementare una cache veloce (ma costosa) detta TLB, la quale contiene una piccola parte delle entry della tabella delle pagine. Quindi, durante un accesso alla memoria: se la pagina cercata è nel TLB, esso ritorna il numero di frame con un singolo accesso; altrimenti, è necessario accedere alla tabella delle pagine in memoria (due accessi).

**Si spieghi come funziona l'indirizzamento nel caso di tabella delle pagine multilivello.**

Nelle architetture moderne lo spazio di indirizzamento virtuale è molto maggiore dello spazio fisico, e quindi sono necessari dei meccanismi per gestire il problema della dimensione della tabella delle pagine. Uno di questi è quello di paginare la tabella delle pagine (*tabella delle pagine multilivello*). In pratica, solo alcune parti della tabella delle pagine sono memorizzate esplicitamente in memoria (le altre sono su disco). Quindi, per tradurre un indirizzo logico in un indirizzo fisico bisognerà:

- Recuperare la tabella di secondo livello corrispondente all'indirizzo che si trova alla riga indicata da *p1* (indice della tabella delle pagine esterna).
- Recuperare, dalla tabella di secondo livello, l'indirizzo che si trova alla riga indicata da *p2* (offset all'interno della pagina della tabella delle pagine interna).
- Sommare *d* (offset) al valore fornito dalla tabella di secondo livello.

Siccome ogni livello è memorizzato come una tabella separata in memoria, la traduzione dell'indirizzo logico in quello fisico può richiedere fino ad un massimo di 4 accessi in memoria (nel caso di una tabella delle pagine a 3 livelli).

**Si descriva cosa si intende per segmentazione paginata commentando in particolare i vantaggi rispetto a paginazione e a segmentazione. Si commenti lo schema architettonale per la traduzione degli indirizzi.**

È possibile combinare le tecniche di paginazione e segmentazione per migliorarle entrambe. Questa soluzione è detta *segmentazione paginata* e prevede che: ogni segmento venga suddiviso in pagine; ogni segmento possieda la sua *tabella delle pagine*; ci sia una *tabella dei segmenti* che contiene l'*indirizzo base* delle tabelle delle pagine per ogni segmento.

*Vantaggi* della segmentazione paginata:

- + L'allocazione dei frame non richiede algoritmi specifici.
- + Non esiste frammentazione esterna.
- + C'è consistenza tra vista utente e vista fisica della memoria.

**Si descriva in dettaglio il concetto di memoria virtuale facendo esplicito riferimento a un contesto di allocazione dello spazio tramite paginazione.**

Solo una parte del programma può essere caricata in memoria. Di conseguenza lo spazio degli indirizzi logici può essere molto più grande dello spazio degli indirizzi fisici e, così facendo, più processi possono essere mantenuti in memoria. Il concetto chiave della *memoria virtuale* è la possibilità di swappare pagine dalla memoria fisica al disco e viceversa, invece che l'intero processo. La memoria virtuale, quindi, permette la separazione della memoria logica (utente) dalla memoria fisica. La *paginazione su domanda* è un tipo di implementazione della memoria virtuale che si basa sul fatto che una pagina viene caricata in memoria solo quando è necessario. Per realizzare la paginazione su domanda è fondamentale sapere lo stato di una pagina (*bit di validità*). Se per lo swap-in non ci sono frame liberi in memoria, il S.O. dovrà rimpiazzare delle pagine tramite un algoritmo di rimpiazzamento, il quale sceglierà una pagina in memoria (vittima) e ne farà lo swap-out sul disco (mettendo il bit di validità a 0).

**Cos'è l'anomalia di Belady? Mostrare che un algoritmo di rimpiazzamento delle pagine implementato tramite Stack è immune dall'anomalia di Belady.**

Nell'algoritmo FIFO, la prima pagina introdotta è la prima ad essere rimossa. FIFO è un algoritmo "cieco", in quanto non viene valutata l'importanza della pagina rimossa, ovvero la sua frequenza di riferimento. Inoltre, FIFO tende ad aumentare il tasso di Page fault e soffre dell'*anomalia di Belady*, ovvero che il numero di Page fault può non decrescere all'aumentare del numero dei frame. Un algoritmo migliore del FIFO è l'algoritmo LRU, che può essere implementato tramite *Stack* ed è quindi immune all'anomalia di Belady. In particolare, viene mantenuto uno Stack di numeri di pagina e, ad ogni riferimento a una pagina, il numero corrispondente viene estratto e messo in cima allo Stack; quindi si rimpiazza la pagina corrispondente al numero che si trova in fondo allo Stack (nessuna ricerca della pagina da rimpiazzare).

**Indicare due possibili implementazioni (eventualmente approssimate) dell'algoritmo LRU in un contesto di rimpiazzamento delle pagine.**

L'algoritmo LRU è un'approssimazione dell'algoritmo ideale (minimo numero di page fault, si rimpiazzano le pagine che non verranno usate per il periodo di tempo più lungo, si richiede una conoscenza anticipata della reference string). L'idea alla base di LRU è quella di usare il passato recente come previsione del futuro. Infatti, si rimpiazza la pagina che non viene usata da più tempo. Per ricavare il tempo dell'ultimo utilizzo si può ricorrere a delle approssimazioni di LRU realizzate tramite:

- *Bit di reference*, in cui ad ogni pagina viene quindi associato un bit che, quando la pagina è referenziata, viene messo a 1 dall'hardware; il rimpiazzamento sceglierà una pagina che ha il bit a 0.
- *Second chance*, in cui c'è una coda FIFO circolare basata sui bit di reference, e quindi se il bit vale 0, la pagina si rimpiazza; altrimenti, si mette il bit a 0 ma si lascia la pagina in memoria.
- *Conteggio dei riferimenti*, in cui viene mantenuto per ogni pagina il conteggio del numero di riferimenti e si rimpiazza la pagina con il conteggio più basso (LFU) oppure quella con il conteggio più alto (MFU).

**Si descriva il modello del working set per l'allocazione di frame ai processi.**

Al momento dell'esecuzione, è importante scegliere bene quanti frame allocare ad ogni processo. Tale *allocazione* può essere fissa (in parti uguali o proporzionale) oppure variabile. Quest'ultima permette di modificare dinamicamente le allocazioni ai vari processi. Un problema, però, è rappresentato dal definire in base a cosa effettuare le modifiche. Due soluzioni possono essere: il calcolo del *working set* e il calcolo della frequenza dei Page fault. Nel primo caso, l'idea è che un processo necessita di un numero di frame pari alla sua località. Per misurarla, viene assegnato ad ogni processo un numero di frame sufficiente a mantenere in memoria il suo working set, ovvero il numero di pagine referenziate nell'intervallo di tempo  $[t-\Delta, t]$  più recente (detto finestra del working set). Infine, il working set si può approssimare con timer e bit di reference.

**Si descrivano i concetti di base legati al RAID e si descriva il funzionamento del livello RAID 0+1.**

Il *RAID* è un sistema per gestire un insieme di dischi che ha l'obiettivo di migliorare l'affidabilità e incrementare le prestazioni. Le strutture RAID si basano su: *mirroring* (copia speculare dei dati) e *data striping* (sezionamento dei dati). Il sezionamento (RAID 0) aumenta le prestazioni grazie al parallelismo garantito nelle operazioni di lettura/scrittura. Il mirroring (RAID 1) aumenta le prestazioni in lettura ma richiede affidabilità (che aumenta linearmente con il numero di dischi). Infine, esistono due soluzioni combinate:

- RAID 0+1, che sfrutta la velocità del livello 0 implementando la sicurezza come nel livello 1 (3 dischi sezionati secondo RAID 0 copiati in altri 3 dischi secondo RAID 1).
- RAID 1+0, in cui ogni disco di ogni stripe può guastarsi senza far perdere dati al sistema (3 insiemi formati da disco e copia secondo RAID 1 sezionati secondo RAID 0).

**Si descrivano in dettaglio le tecniche (indicizzata, concatenata, combinata) per l'allocazione dello spazio su disco ai file.**

Con il metodo dell'*allocazione indicizzata* ogni file ha un blocco indice (contenuto nella directory) contenente la tabella degli indirizzi dei blocchi fisici. I vantaggi di questo metodo sono che l'accesso casuale è efficiente e che non c'è frammentazione esterna. Gli svantaggi sono l'overhead per la index table e il fatto che la dimensione del blocco limita la dimensione del file (la index table può contenere un numero di indirizzi pari alla dimensione del blocco). Quindi, per avere file senza limiti di dimensione si dovrà usare uno di questi schemi a più livelli:

- *Schema con indici multilivello*, dove una tabella esterna contiene i puntatori alle index table interne.
- *Schema concatenato*, in cui si ha una lista concatenata di blocchi indice dove l'ultimo indice punta a un altro blocco indice.
- *Schema combinato*, in cui gli indirizzi sono in parte diretti e in parte con allocazione a 1, 2 e 3 livelli.