


# Emiddio Ingenito - Sistemi Operativi - Parte B

 Autore	Emiddio Ingenito @emikodes
--	----------------------------

## Memoria Centrale

### Collegamento degli indirizzi

1. Collegamento in fase di compilazione
2. Collegamento in fase di caricamento
3. Collegamento in fase di esecuzione

## Paginazione

### Come funziona la paginazione

### Protezione della memoria

### Paginazione gerarchica

### Tabella delle pagine con hashing

### Tabella delle pagine invertita

### Pagine condivise

## Segmentazione

## Memoria Virtuale

### Vantaggi della Memoria Virtuale

### Allocazione dei Frame e Trashing

## Sottoinsiemi di I/O

### Comunicazione tra CPU e periferiche

### Modalità di gestione dell'I/O

### Classificazione dei dispositivi di I/O

### Tecniche per ottimizzare l'I/O

### Rilevamento e gestione degli errori

## Protezione

## Memoria Centrale

La **memoria centrale** è un componente essenziale di un moderno elaboratore, e consiste in un vettore di **parole** di memoria (o *byte*), ognuna delle quali ha un proprio indirizzo. Il processore caricherà nei propri registri istruzioni e dati presi direttamente da essa, in particolare da quella posizione indicata dal *program*

*counter*. Il numero di accessi alla memoria durante la normale esecuzione di un processo può essere elevatissimo, dunque la sua gestione deve essere efficiente e rigorosa.

La memoria centrale vede solo flussi di indirizzi, ciò significa che ignora chi e come li generi e con quale scopo.

## Collegamento degli indirizzi

I programmi risiedono generalmente su memorie secondarie sotto forma di file binari eseguibili, pronti ad essere attivati come processi. L'insieme dei processi in attesa di essere caricati in memoria centrale forma la cosiddetta **coda di entrata**, dalla quale ne verrà selezionato uno (o più) da caricare e mandare in esecuzione. Una volta terminata l'esecuzione, le risorse allocate (inclusa la memoria occupata dal processo) verranno rilasciate per essere utilizzate da altri processi.

I processi non possono essere caricati a partire da un indirizzo qualunque, ma devono seguire una serie di passaggi per il corretto collegamento degli indirizzi. Durante questi passaggi, i dati e le istruzioni vengono mappati in memoria con modalità diverse a seconda del tipo di collegamento scelto.

### 1. Collegamento in fase di compilazione

In questa modalità, l'indirizzo in cui il programma verrà caricato è già noto al momento della compilazione. Il compilatore genera quindi un **codice assoluto**, con indirizzi di memoria fissi e immutabili. Se il programma viene caricato in una posizione diversa da quella prevista, l'esecuzione risulterà errata o impossibile, rendendo necessaria una ricompilazione per adattarlo alla nuova posizione.

#### ◆ Esempio:

Un programma Assembly con indirizzi assoluti potrebbe contenere un'istruzione come:

```
MOV AX, [1000H] ; Legge un valore dalla cella di memoria 1000H
ADD AX, 10      ; Aggiunge 10 al valore letto
MOV [1002H], AX ; Salva il risultato nella cella 1002H
```

Se il sistema operativo lo carica in un'area diversa, ad esempio a **3000H**, continuerà comunque a leggere da **1000H**, causando errori ( **1000H** non si trova

nello spazio di indirizzamento del programma in esecuzione)

## 2. Collegamento in fase di caricamento

Se al momento della compilazione non si sa ancora in quale posizione della memoria centrale verrà caricato il programma, il compilatore genera un **codice rilocabile**. In questa modalità, il sistema operativo decide l'indirizzo di partenza solo al momento del caricamento del programma in RAM. Gli indirizzi assoluti vengono calcolati partendo da questo indirizzo base.

### ◆ Esempio:

Un programma scritto in Assembly potrebbe usare indirizzamento relativo invece di assoluto:

```
MOV AX, [SI]    ; Legge un valore dalla posizione relativa SI
ADD AX, 10      ; Aggiunge 10 al valore
MOV [SI+2], AX  ; Salva il risultato nella posizione relativa SI+2
```

Il sistema operativo imposta il registro **DS** con l'indirizzo base assegnato (ad esempio **3000H**), e gli accessi a memoria avvengono con riferimenti **relativi** a questo segmento.

## 3. Collegamento in fase di esecuzione

Questa modalità è tipica dei sistemi in cui i processi possono essere **spostati dinamicamente** in memoria anche durante l'esecuzione, ad esempio per ottimizzare l'uso della RAM o per implementare la memoria virtuale. In questi casi, il processo non può affidarsi a indirizzi statici e deve usare una traduzione dinamica degli indirizzi. Questo è possibile grazie alla **Memory Management Unit (MMU)**, un'unità hardware che traduce gli indirizzi logici generati dal programma in indirizzi fisici reali.

### ◆ Esempio:

Un processo può avere un'istruzione che accede alla memoria con un indirizzo virtuale:

```
MOV AX, [2000H] ; Indirizzo virtuale generato dal programma
```

La MMU converte automaticamente **2000H** in un indirizzo fisico reale, che può essere **5000H** in un momento e **8000H** successivamente, se il processo viene spostato in un'altra area di memoria.

Questa modalità è ampiamente usata nei moderni sistemi operativi con **paginazione** e **memoria virtuale**, dove il programma non ha bisogno di conoscere la sua posizione fisica effettiva.

Il collegamento in fase di esecuzione è il più utilizzato nei moderni sistemi operativi grazie alla sua flessibilità e capacità di gestire in modo efficiente la memoria.

### Spazio di indirizzamento logico e fisico

L' **indirizzo fisico** è l'indirizzo nello spazio di memoria centrale che individua in modo univoco una parola in esso contenuta. Lo **spazio di indirizzamento fisico** è un vettore lineare che va da 0 all'ultimo spazio indirizzabile del chip di memoria.

L' **indirizzo logico** è invece un indirizzo astratto, consistente solo all'interno dello spazio di indirizzamento del processo, rappresentandone lo spiazamento rispetto alla prima parola. Lo **spazio di indirizzamento logico** è dunque lo spazio indirizzabile dal nostro processo, e la sua prima parola corrisponde ad un certo indirizzo fisico detto *indirizzo di base*. Al processo non interessa se al di là dei confini del proprio spazio di indirizzamento siano caricate parole associate ad altri processi, tanto lui non potrà accedervi come gli altri non possono accedere al suo.

I metodi di collegamento degli indirizzi in fase di compilazione e caricamento generano indirizzi logici e fisici identici, cosa che invece non accade in fase di esecuzione. In quest'ultimo caso la responsabilità di trasformarli è demandata a un dispositivo ausiliario hardware che abbiamo già nominato, ovvero la **MMU**. Il sistema di trasformazione semplice consiste nell'utilizzo del **registro di rilocalizzazione**, il cui valore viene aggiunto ad ogni indirizzo generato dal processo nel momento in cui viene inviato in memoria. La formula generale è:  $\text{indirizzo fisico} = \text{indirizzo logico} + \text{offset}$ , dove l'offset è lo spiazamento tra lo 0 logico del processo e la sua mappatura effettiva in memoria centrale. Grazie a questo mappaggio il processo non vede mai gli indirizzi fisici reali, ma tratta solo quelli logici che sono molti meno del totale quindi più semplici e veloci da manipolare.

Ricapitolando: gli indirizzi logici vanno da 0 a un valore massimo, mentre i corrispondenti indirizzi fisici vanno da  $R+0$  a  $R+\text{valore massimo}$  (dove  $R$  è il valore del registro di rilocalizzazione). Nella fase di **linking** dei programmi, viene eseguito il calcolo degli indirizzi logici a partire da quelli simbolici generati dalle chiamate di sistema invocate. Nella fase di **binding** si passa invece dall'indirizzo logico a quello fisico; può avvenire durante la compilazione, o nel momento del loading del programma, o anche durante l'esecuzione (soprattutto se ho librerie dinamiche).

### **Caricamento dinamico**

Dato che per essere eseguito un programma deve risiedere in memoria centrale, come fare quando le sue dimensioni superano quelle della memoria stessa? La soluzione è il **caricamento dinamico**, che prevede inizialmente il caricamento di un insieme di procedure fondamentali e il mantenimento delle altre su disco in formato di caricamento rilocabile; nel caso in cui queste ultime si rivelino necessarie, verrà richiesto a un *loader* di caricarle e di aggiornare la tabella degli indirizzi del programma. Il vantaggio di tale sistema è che non si spreca inutilmente spazio di memoria per procedure che vengono utilizzate raramente (come quelle di gestione degli errori). Il sistema operativo non gestisce direttamente il caricamento dinamico, la sua reale implementazione è lasciata ai programmatori.

### **Collegamento dinamico e librerie condivise**

Utilizzare collegamenti dinamici piuttosto che statici alle librerie di sistema consente di risparmiare spazio sia su memorie di massa che in quella centrale. Il motivo è semplice: un collegamento statico considererebbe le librerie come un qualsiasi altro modulo da caricare, quindi ogni programma dovrebbe incorporarne una copia nel file eseguibile e nel suo spazio di indirizzamento. Con i **collegamenti dinamici** (attuati in fase di esecuzione) viene invece inclusa nell'eseguibile un' *immagine (stub)* che indica come individuare la procedura di libreria desiderata già residente in memoria, o come reperirla dal disco se non è presente. In questo modo tutti i processi che usano una stessa libreria eseguiranno una sola copia del codice.

Il sistema a **librerie condivise** estende i vantaggi del collegamento dinamico, consentendo inoltre di aggiornare le versioni delle librerie e di fare in modo che ogni programma utilizzi quella a sé compatibile.

Al contrario del caricamento dinamico, il collegamento dinamico può richiedere l'intervento del sistema operativo, ad esempio in quei casi in cui un processo avrebbe bisogno di una procedura contenuta nello spazio di indirizzamento di un altro.

## **Overlay**

Se il caricamento dinamico offriva una soluzione all'esecuzione dei processi con dimensioni maggiori della memoria centrale, la tecnica dell' **overlay** consente a un processo di essere più grande della quantità di memoria a esso allocata, ovvero al suo spazio di indirizzamento. Il principio è simile: mantenere in memoria solo le istruzioni e i dati che sono necessari in un certo momento, rimpiazzando man mano quelle più vecchie. Si può dire che un overlay è una sorta di partizione del programma di partenza, indipendente dagli altri e quindi in grado di compiere autonomamente tutta una serie di operazioni che lo caratterizzano. Quando la sua esecuzione parziale del programma si esaurisce, viene deallocato e al suo posto ne viene caricato un altro.

Gli overlay non richiedono alcun supporto speciale da parte del sistema operativo, è compito del programmatore definirne il numero e le suddivisioni. Questo si rivela però un compito improbo, dato che i programmi in questione sono generalmente molto grandi (per quelli piccoli questa tecnica non è evidentemente necessaria) e suddividerli in parti implica una conoscenza puntigliosa della loro struttura. Inoltre i moduli possono avere dimensioni diverse, con conseguenti sprechi di spazio quando se ne caricheranno alcuni di grandezza più ridotta. Troppe responsabilità al programmatore e sfruttamento della memoria centrale poco efficiente fanno così dell'overlay una tecnica utilizzata solo per sistemi con poca memoria fisica e supporto hardware poco avanzato.

## **Swapping**

In un sistema operativo multitasking è naturale che tra i processi caricati in memoria centrale alcuni non stiano facendo niente, o perché sono in attesa o perché hanno bassa priorità. Ciò però comporta uno spreco di spazio, cui si può far fronte adottando la tecnica dello **swapping**. In breve essa consiste nello scambiare temporaneamente un processo in attesa con un altro in stato di pronto o running che si trova in una *memoria temporanea (backing storage)*, tipicamente un disco veloce. Nello specifico avremo quattro fasi:

- *identificare* i processi in stato di attesa
- *salvare sulla memoria temporanea* i loro dati sensibili (dati globali, heap, stack)
- rimuovere dalla memoria centrale tali processi (*scaricamento*)
- caricare nello spazio appena liberato quei processi in stato di pronto o running che si trovavano nella memoria temporanea (*caricamento*).

Una variante di questa tecnica è il **roll out/roll in**, che individua i processi da caricare e scaricare in base alla loro priorità.

Lo swapping si applica sugli interi processi ed impone almeno due vincoli. Il primo è che si possono spostare solo quei processi in stato di riposo; il secondo è che non si possono spostare quei processi che hanno invocato una chiamata di funzione e sono ancora in attesa di una risposta, o il processo che gli subentrerebbe la riceverebbe al posto suo. Possibili soluzioni sono impedire lo swap di processi in attesa di I/O oppure permetterlo solo a quelli che usano i buffer condivisi del sistema operativo.

Lo swapping standard ha come vantaggio l'aumento del grado di multiprogrammazione del sistema, ma essendo gestito automaticamente dal sistema operativo ed essendo applicato agli interi processi comporta un netto rallentamento delle prestazioni. Una versione modificata viene invece utilizzata in molti sistemi operativi UNIX, in cui ad esempio viene attivato solo quando esaurisce lo spazio in memoria centrale e solo fintanto non se ne libera un po'.

### **Allocazione contigua di memoria**

L' *allocazione contigua di memoria* è un metodo per allocare nel modo più efficiente possibile sia il sistema operativo che i processi degli utenti in memoria centrale. Il primo viene generalmente memorizzato nella parte bassa della memoria centrale, separato dalle sezioni riservate ai processi da un *vettore di interrupt*.

### **Protezione della memoria centrale**

Proteggere la memoria centrale significa garantire che non avvengano al suo interno *accessi illegali* da parte dei processi, dove per accessi illegali si intendono quelli al di fuori del proprio spazio di indirizzamento (che sconfinano quindi in

quello del sistema operativo o di altri processi). La protezione avviene attraverso l'utilizzo congiunto di due registri che fanno parte del contesto di ogni processo:

- il **registro di rilocalizzazione**, che contiene il valore del più piccolo indirizzo fisico
- il **registro limite** che contiene l'intervallo degli indirizzi logici, quindi la dimensione in byte dello spazio di indirizzamento

Quando lo schedulatore seleziona un processo da mandare in esecuzione, la *Memory Management Unit* si occuperà di verificare che ogni indirizzo logico sia inferiore del registro limite associato, dopodiché lo mapperà dinamicamente aggiungendogli il valore del registro di rilocalizzazione. L'indirizzo così ottenuto è inviato in memoria centrale.

Il registro di rilocalizzazione fa inoltre in modo che le dimensioni del sistema operativo cambino dinamicamente, rendendo possibile l'utilizzo o meno di parte del codice (detto *transiente*).

### **Allocazione della memoria centrale**

Uno dei metodi più semplici per allocare memoria centrale consiste nel suddividerla in *partizioni*, ciascuna delle quali può contenere al più un processo. Dal loro numero dipende il grado di multiprogrammazione del sistema. In particolare, con il *metodo delle partizioni multiple* quando un processo termina la sua computazione viene sostituito con uno selezionato dalla coda dei processi pronti.

Abbiamo due schemi di partizionamento:

- **schema a partizione fissa**, in cui si hanno partizioni di dimensione *statica* (definite al bootstrap) e una tabella aggiornata che indica quali parti della memoria centrale sono disponibili e quali occupati (inizialmente sono tutti liberi). Man mano che i processi si attivano, vengono messi in una coda di entrata gestita con un qualsiasi algoritmo di schedulazione; il sistema operativo valuterà le loro richieste di memoria e cercherà di allocarli in blocchi abbastanza grandi da ospitarli. Se tale ricerca fallisce, il processo entra in stato di attesa e vi rimarrà finché un altro non avrà terminato la sua computazione e quindi reso disponibile la partizione che occupava
- **schema a partizione variabile**, che a differenza dello schema precedente consente di modificare l'indirizzo di base delle varie partizioni rendendo di fatto



possibile variarne le dimensioni, ad esempio unendo blocchi contigui precedentemente distinti o suddividendone uno particolarmente sovradimensionato.

Nello schema a partizione variabile la configurazione dello spazio disponibile continua a variare, dunque il sistema dovrà controllare spesso se la situazione è diventata favorevole per il caricamento di uno dei processi in attesa nella coda di entrata. Questo viene anche chiamato *problema dell'allocazione dinamica della memoria centrale*, per il quale sono percorribili tre strategie:

- **first-fit**, che assegna il primo blocco libero abbastanza grande per contenere lo spazio richiesto
- **best-fit**, che assegna il più piccolo blocco libero che lo può contenere
- **worst-fit**, che assegna il più grande blocco libero che lo può contenere

Le simulazioni hanno dimostrato che le strategie migliori sono le prime due, in particolar modo la *first-fit* che è la più veloce.

### Frammentazione

Il problema del *first-fit* e del *best-fit* è che soffrono di **frammentazione esterna**, ovvero lasciano dei blocchi liberi fra quelli occupati, blocchi che se fossero uniti e contigui potrebbero ospitare un altro processo. Questo accumularsi di spazio non sfruttato alla lunga comporta un considerevole abbassamento delle prestazioni, cosa particolarmente grave se si pensa che nel caso peggiore potrebbe soffrire di frammentazione esterna ogni coppia di blocchi. Il problema non è evitabile: la *regola del 50 per cento* dice che su N blocchi allocati con la first fit ne andranno persi N/2 a causa della frammentazione. Bisogna dunque fare in modo di ottimizzare la situazione a posteriori.

Una prima soluzione è la *compattazione*, ovvero la fusione di tutti i blocchi liberi in uno solo. Tuttavia questa tecnica può essere applicata solo se la rilocalizzazione è dinamica ed è fatta al momento dell'esecuzione, dato che in tal caso basterebbe spostare programma e dati. Si tratta inoltre di un procedimento piuttosto costoso.

Altra contromisura è permettere allo spazio di indirizzo logico di un processo di essere non contiguo, un'idea che verrà usata con profitto dalle tecniche di *paginazione* e *segmentazione* che vedremo poi.

Se abbiamo la *frammentazione esterna* non possiamo certo farci mancare anche la **frammentazione interna**, che si presenta quando carichiamo all'interno di un blocco un processo più piccolo della sua dimensione. Lo scarto di spazio tra la dimensioni del processo e del blocco rappresenta la frammentazione.

---

## Paginazione

La **paginazione** è un meccanismo di gestione della memoria adottato dalla maggior parte dei sistemi operativi. Il suo scopo è superare molte delle limitazioni dei metodi tradizionali, garantendo:

- Il caricamento e scaricamento solo di piccole porzioni di memoria, riducendo il costo dello swapping dell'intero processo.
- La permanenza in memoria solo delle parti del processo necessarie nel breve periodo.
- La riduzione al minimo degli sprechi di memoria.
- La possibilità di utilizzare aree di memoria non contigue per lo stesso processo.
- Una gestione automatica della memoria, indipendente dal programmatore, integrata con l'hardware e il sistema operativo.

### Come funziona la paginazione

- La **memoria fisica** viene divisa in **frame di dimensione fissa** (o pagine fisiche).
- La **memoria logica** viene divisa in **pagine** (o pagine logiche), della stessa dimensione dei frame.

Quando la CPU genera un **indirizzo logico**, questo è composto da due parti:

1. Il **numero di pagina logica** ( $p$ )
2. Lo **spiazzamento all'interno della pagina** ( $d$ ), che indica la posizione specifica del dato dentro la pagina.

La traduzione di un indirizzo logico in un indirizzo fisico è gestita da un'unità hardware (MMU), che fa uso di una **tabella delle pagine, univoca per ogni**

**processo**, che associa ogni pagina logica a un frame fisico.

1.  $p$  è usato come indice nella tabella delle pagine per trovare il corrispondente indirizzo del frame  $f$ .
2. L'indirizzo fisico del dato viene calcolato sommando  $(f, d)$

Il sistema operativo deve inoltre monitorare lo stato della memoria fisica, per conoscere quali frame sono occupati e quali liberi per il caricamento di nuove pagine. Per farlo, utilizza una **tabella dei frame**.

### **Vantaggi e svantaggi della paginazione**

- La paginazione elimina la **frammentazione esterna** perché pagine e frame hanno la stessa dimensione.
- Tuttavia, introduce **frammentazione interna**: se un processo non riempie completamente i frame assegnati, parte della memoria rimane inutilizzata.

Se piccola, la tabella delle pagine può essere caricata in registri ad alta velocità, tuttavia a causa delle grandi dimensioni, si sceglie nei sistemi moderni di memorizzarla in memoria centrale.

Il problema di questo approccio è che ogni accesso alla memoria richiede prima un'operazione aggiuntiva per consultare la tabella, rallentando il sistema di un fattore 2.

Per velocizzare gli accessi, si utilizza una cache associativa chiamata **Translation Lookaside Buffer (TLB)**, che mantiene le traduzioni di indirizzi recenti.

Quando la TLB è piena, si utilizza una politica di sostituzione, come **Least Recently Used (LRU)**, che rimpiazza l'elemento meno usato di recente.

Il **tasso di successo della TLB** (*hit ratio*) misura la frequenza con cui la TLB trova direttamente il numero di frame senza dover accedere alla tabella in memoria.

### **Protezione della memoria**

Per proteggere la memoria da accessi non autorizzati, ogni elemento della tabella delle pagine include dei **bit di protezione**, che definiscono le operazioni consentite sulla pagina:

- **Sola lettura** (es. costanti).
- **Lettura e scrittura** (es. dati).
- **Sola esecuzione** (es. codice).

Inoltre, viene mantenuto un **bit di validità**, che indica se la pagina è attualmente disponibile per l'accesso in memoria centrale.

Tentativi di accesso illegale generano una trap gestita dal SO.

Per ottimizzare la gestione di grandi tabelle delle pagine, possono essere utilizzate alcune tecniche avanzate:

## Paginazione gerarchica

La tabella delle pagine stessa viene paginata. L'indirizzo logico è suddiviso in tre parti:

- *p1*: indice nella tabella principale.
- *p2*: indice nella sottotabella.
- *d*: spiazzamento nella pagina.

Per i sistemi a **64 bit**, le paginazioni gerarchiche diventano poco efficienti perché richiederebbero troppi accessi ai vari livelli.

## Tabella delle pagine con hashing

Utilizza una **tabella di hashing**, così utilizzata:

1. Il numero della **pagina virtuale** viene passato a una **funzione di hash**, che lo trasforma in un indice per accedere a una tabella più compatta.
2. A causa delle **collisioni** (cioè più pagine virtuali che finiscono nello stesso indice), ogni posizione della tabella contiene una **lista concatenata** di elementi che associano una pagina virtuale ad un frame fisico.
3. Il sistema scorre la lista per trovare la pagina cercata:
  - Se la trova, usa il **numero di frame** associato per ottenere l'indirizzo fisico.
  - Se non la trova, significa che la pagina non è in memoria e potrebbe essere necessario un **page fault**.

Se un processo ha meno frame delle pagine di cui ha bisogno attivamente, il sistema subisce il *trashing*: un numero eccessivo di *page fault* che rallenta drasticamente le prestazioni. Per evitarlo, si utilizza il *modello del Working Set (WS)*, che stima la località di riferimento di un processo contando il numero di pagine a cui ha acceduto negli ultimi *N* riferimenti. Il sistema operativo monitora il *WS* di ogni processo e gli assegna un numero di frame sufficiente a contenerlo. Se ci sono abbastanza frame disponibili, può essere avviato un nuovo processo; in caso contrario, un processo potrebbe essere sospeso per liberare memoria.

## Tabella delle pagine invertita

Invece di una tabella per ogni processo, esiste una **unica tabella globale**.

Ogni elemento associa ad ogni frame fisico:

- **ASID** (identificatore del processo).
- **Numero di pagina logica memorizzata in quel frame.**

Questa tecnica riduce drasticamente l'uso della memoria, ma rende più lenta la ricerca di una pagina. Per migliorarne l'efficienza, si può combinare con **hashing**, riducendo il numero di accessi richiesti.

## Pagine condivise

La paginazione permette la condivisione di codice comune. Per evitare di duplicare le pagine, queste possono essere condivise tra processi, mappando lo stesso frame nelle tabelle delle pagine di processi diverse.

Si usa anche per la comunicazione con memoria condivisa.

Il codice non dev'essere automodificante.

---

# Segmentazione

La **segmentazione** è una tecnica di gestione della memoria che permette a un processo di:

- Utilizzare uno spazio di indirizzi fisici non contiguo.

- Tipizzare le diverse porzioni di indirizzamento logico, come, ad esempio, distinguere tra l'area del codice e quella dei dati.
- Supportare una divisione della memoria dal punto di vista dell'utente.
- Conservare in memoria solo i segmenti necessari nel breve periodo, scambiando i segmenti non utilizzati tra la memoria centrale e l'area di swap quando richiesto.

Con la segmentazione, lo spazio di indirizzamento logico di un processo può essere molto grande, teoricamente pari a quello complessivo supportato dalla macchina (ad esempio,  $2^{32}$  indirizzi per un sistema a 32 bit,  $2^{64}$  per un sistema a 64 bit). Questo spazio logico viene separato dallo spazio di indirizzamento fisico, che è limitato dalla quantità effettiva di memoria disponibile nel sistema.

Un limite della **paginazione** è che non consente di differenziare le varie porzioni dello spazio di indirizzamento logico. In altre parole, non permette di distinguere facilmente l'area del codice da quella dei dati, il che rende complessi i controlli mirati su programmi di grandi dimensioni.

La **segmentazione** risolve questo problema, mantenendo la separazione tra memoria logica e fisica come nella paginazione:

- **Memoria centrale fisica:** è divisa in **frame fisici**. La dimensione di ciascun frame è tipicamente fissa e non dipende dai segmenti.
- **Spazio di indirizzamento del processo:** è suddiviso in **segmenti logici** di dimensione **variabile**, che dipende dalle necessità del processo.

I segmenti sono generati automaticamente dal compilatore, e allocati in modo non contiguo all'interno della memoria fisica, mentre i segmenti non attivi vengono spostati nell'area di swap

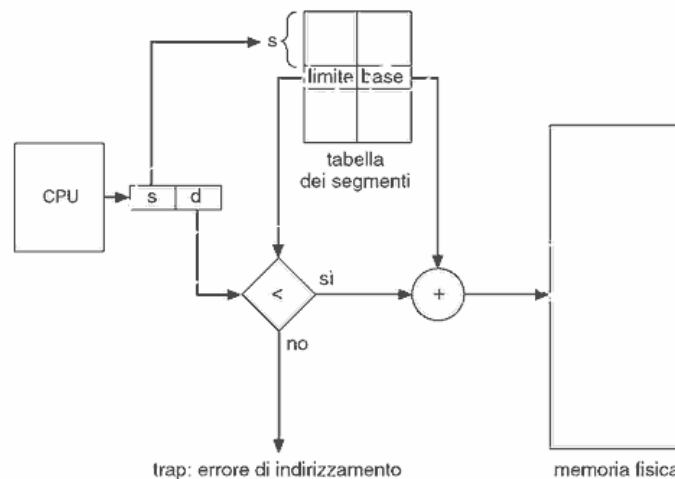
ad es. un compilatore C crea i segmenti:

- Il codice
- Variabili globali
- Variabili locali
- Heap
- Librerie standard del C

L'indirizzo logico consiste in un numero di segmento ( $s$ ) e uno spiazzamento ( $d$ ).

Analogamente alla paginazione, la traduzione degli indirizzi logici avviene tramite una **tabella dei segmenti**. Ogni entry della tabella contiene due informazioni principali: l'indirizzo fisico di partenza del segmento, e il *limite del segmento*, che definisce la sua lunghezza..

Il numero di segmento è usato come indice nella tabella dei segmenti, l'offset viene sommato alla base, ottenuta dalla tabella, risultando così nell'indirizzo fisico. Se lo spiazzamento supera il valore del limite, l'accesso viene bloccato.



L'hardware dedicato per il supporto alla segmentazione è anche in questo caso la **MMU**, stavolta orientata alla gestione dei segmenti.

### Protezione e condivisione

Un processo può accedere solo ai suoi segmenti dunque la protezione dagli accessi di altri processi è implicita. Ogni segmento contiene una porzione diversa del programma, ed è associato a dei bit di protezione che lo definiscono in sola lettura, lettura/scrittura e sola esecuzione.

La condivisione di codice è semplice e avviene mappando lo stesso frame nelle tabelle dei segmenti di più processi. Per condividere codice vi sono due vincoli: il codice non deve essere automodificante; il segmento condiviso deve avere lo stesso numero logico nei vari processi. Non ci sono vincoli per la condivisione di dati, che può essere usata anche per comunicazione con memoria condivisa

### Frammentazione

La frammentazione esterna è un problema per la segmentazione dal momento che i segmenti sono di dimensione variabile e non sempre coincidono con la dimensione dei frame fisici. Dato però che per sua stessa natura la segmentazione avviene con una rilocalizzazione dinamica, è possibile adottare sistemi di compattazione che riducono il problema. Se in aggiunta vengono effettuate valutazioni ben ponderate da parte dello schedatore a lungo termine, si può mantenere la frammentazione esterna sotto controllo.

### **Segmentazione con paginazione**

Paginazione e segmentazione hanno entrambi vantaggi e svantaggi. È possibile combinare i vantaggi di entrambi

Si divide la memoria fisica in frame di dimensione fissa, evitando frammentazione esterna. Si divide la memoria logica in segmenti tipizzati di dimensioni

diverse; a loro volta i segmenti sono divisi in pagine, ognuna di stessa dimensione, pari a quella dei

frame. Una pagina di un segmento è caricata in un frame. Gli indirizzi logici sono formati da numero

di segmento, pagina nel segmento, offset nella pagina; quelli fisici da numero di frame e offset nel frame.

La traduzione è svolta dall'MMU, gestita dal SO.

## **Memoria Virtuale**

Le istruzioni dei processi per poter essere eseguite devono risiedere in memoria centrale, con le limitazioni sulle dimensioni fisiche che ne conseguono. Spesso è però inutile caricare tutto il codice, o perché contiene procedure che riusano raramente (ad esempio gestione degli errori), o perché è difficile che diverse porzioni del programma servano contemporaneamente. L'esecuzione di un programma caricato solo parzialmente in memoria ha invece diversi vantaggi, dato che non è più vincolato alla dimensione della memoria centrale e permette il caricamento di più processi aumentando così il livello di multiprogrammazione (dunque lo sfruttamento della CPU).

La **memoria virtuale** è una tecnica che consente l'esecuzione dei processi che non sono completamente in memoria, astraendo la memoria centrale in un vettore



di memorizzazione molto più grande e uniforme.

In un sistema con **memoria virtuale**, gli indirizzi che un processo utilizza per accedere ai dati non corrispondono direttamente agli indirizzi della memoria fisica, vi è infatti una netta separazione tra memoria logica e fisica.

Tale separazione può avvenire mediante due tecniche principali:

- **Paginazione:** suddivide lo spazio di indirizzamento logico in **pagine** di dimensione fissa e la memoria fisica in **frame** della stessa dimensione. La traduzione di indirizzi logici in fisici è affidata ad un'unità di gestione della memoria MMU che fa uso di una tabella delle pagine.
- **Segmentazione:** suddivide la memoria fisica in frame di dimensione fissa, quella logica in segmenti di dimensione variabile, ciascuno corrispondente a una porzione significativa del programma (es. codice, dati, stack). La traduzione di indirizzi logici in fisici è affidata ad un'unità di gestione della memoria MMU che fa uso di una tabella dei segmenti.

## Vantaggi della Memoria Virtuale

L'adozione della memoria virtuale porta diversi benefici:

- **Espansione dello spazio di indirizzamento:** I processi possono sfruttare più memoria di quella fisicamente disponibile grazie allo swapping.
- **Multiprogrammazione efficiente:** Più processi possono coesistere in memoria, ottimizzando l'uso della CPU.
- **Condivisione del codice:** Le librerie condivise possono essere caricate una sola volta in memoria, riducendo il consumo di risorse.
- **Gestione flessibile della memoria:** Le tecniche di paginazione e segmentazione permettono una gestione dinamica dello spazio disponibile.

In caso di memoria paginata, la memoria virtuale è implementata mediante la **richiesta di paginazione**, che consiste nel caricare in memoria solo le pagine richieste durante l'esecuzione. Se un processo tenta di accedere a una pagina non caricata, si verifica un **page fault**: il sistema operativo interrompe l'esecuzione, carica la pagina dallo spazio di swap e riprende il processo. Nel caso di **richiesta di paginazione pura**, il processo inizia con zero pagine in memoria e genera page fault potenzialmente ad ogni istruzione. Grazie al principio

di **località dei riferimenti**, sappiamo che i riferimenti in memoria si concentrano su regioni di memoria spesso contigue o vicine, pertanto il numero di **page fault** è ridotto

La tecnica di **copy-on-write** permette ai processi di condividere inizialmente le pagine, copiandole solo quando una di esse viene modificata. Questo è utile nelle operazioni di **fork()**, dove il processo figlio spesso esegue subito **exec()**, rendendo inutile la copia immediata delle pagine.

Quando un processo richiede una nuova pagina ma non ci sono frame liberi, il sistema operativo deve selezionare un **frame occupato** e rimpiazzare la **pagina** che contiene.

Questo introduce diversi **algoritmi di sostituzione**:

- **FIFO (First-In, First-Out)**: Sostituisce la pagina più vecchia (calcolata con un timestamp o grazie a una coda FIFO).  
Soffre dell'anomalia di Belady, ovvero il suo tasso di page fault all'aumentare del numero di frame a disposizione può aumentare invece che diminuire come ci si aspetterebbe.
- **LRU (Least Recently Used)**: Sostituisce la pagina non usata da più tempo, approssimando l'ottimo ma con un alto costo hardware.
- **Sostituzione approssimata di LRU**: Utilizza un **bit di riferimento** per tracciare l'uso recente delle pagine, riducendo il costo hardware.
- **Politiche basate sul conteggio**: Tengono traccia della frequenza d'uso delle pagine, sostituendo la meno o la più frequentemente usata.

Per ridurre l'overhead della sostituzione, si usa il **bit di modifica**, che indica se una pagina è stata modificata: se non lo è, può essere sovrascritta direttamente senza essere risalvata nello swap.

## Allocazione dei Frame e Trashing

Maggiore è il numero di frame, minore è la quantità di page fault. Per spartire  $m$  frame su  $n$  processi vi sono due metodi:

- Allocazione omogenea: ogni processo ottiene  $m/n$  frame. È la più semplice. Alcuni processi potrebbero usare poca memoria sprecando il resto.
- Allocazione proporzionale: i frame sono divisi in modo direttamente proporzionale alla dimensione o alla priorità

Un'altra tecnica per prevenire il *trashing* è il controllo della *frequenza dei page fault (PFF)*. Se la frequenza dei *page fault* di un processo supera una certa soglia, gli viene assegnato un frame aggiuntivo (se disponibile); se scende sotto una soglia inferiore, gli viene rimosso un frame. Se non ci sono frame liberi, il sistema può sospendere un processo per recuperare memoria.

---

## Sottoinsiemi di I/O

La gestione delle periferiche in un sistema di elaborazione monoprocesso: si descriva l'organizzazione del software di sistema per la loro gestione e le principali funzioni che esso deve realizzare, evidenziandone caratteristiche, vantaggi e limiti.

La gestione delle periferiche in un sistema monoprocesso è un aspetto fondamentale del sistema operativo, che permette l'interazione tra il processore e i dispositivi di input/output (I/O). Le periferiche possono avere caratteristiche molto diverse tra loro: alcune supportano solo lettura o scrittura, altre sono condivisibili tra più processi, alcune hanno un accesso sequenziale (come i nastri magnetici), altre permettono l'accesso diretto ai dati. Inoltre, possono trasferire dati a caratteri o a blocchi, in modalità sincrona o asincrona, e con velocità variabili.

Per gestire questa eterogeneità, il sistema operativo organizza il software di gestione dell'I/O su tre livelli principali:

- **Gestione del canale di comunicazione:** astrae il modo in cui viene gestito il flusso di informazione tra calcolatore e periferiche.
- **Driver dipendenti dal dispositivo:** moduli del kernel che incapsulano le differenze tra dispositivi dello stesso tipo, unificando il modo in cui il sistema interagisce con dispositivi dello stesso tipo.
- **Driver indipendenti dal dispositivo:** rende trasparenti le differenze tra dispositivi di tipi diversi relativamente a gestione degli errori, bufferizzazione, caching e spooling.

## Comunicazione tra CPU e periferiche

Dal punto di vista hardware, le periferiche comunicano con il processore attraverso il **bus di sistema** e i **controller**, circuiti specializzati che gestiscono il funzionamento del dispositivo. Il processore interagisce con i controller tramite registri specifici:

- **Registro di stato:** indica la disponibilità della periferica e segnala eventuali errori.
- **Registro di controllo:** permette alla CPU di inviare comandi alla periferica.
- **Registri dati:** gestiscono il trasferimento delle informazioni tra CPU e dispositivo.

Una tecnica per semplificare l'accesso alle periferiche è il **mappaggio in memoria**, che consente di trattare i registri delle periferiche come se fossero celle di memoria, permettendo l'uso di normali istruzioni di accesso alla memoria centrale.

## Modalità di gestione dell'I/O

Il sistema operativo può gestire le operazioni di I/O con diversi meccanismi:

- **Attesa attiva (polling):** la CPU controlla periodicamente lo stato della periferica, verificando se è pronta a ricevere o inviare dati. Questo metodo è semplice ma inefficiente, perché la CPU rimane occupata in un ciclo di attesa.
- **Interrupt:** la periferica segnala direttamente alla CPU quando è pronta, generando un **interrupt** che sospende temporaneamente il processo in esecuzione e attiva la routine di gestione dell'I/O. Il sistema utilizza un **vettore**

**degli interrupt** per associare ogni segnale alla sua routine specifica, con un meccanismo di priorità per gestire più richieste simultanee.

- **Accesso diretto alla memoria (DMA):** il **controller DMA** (DMAC) consente alla periferica di accedere direttamente alla memoria interna per scambiare dati, senza coinvolgere continuamente la CPU. Questo riduce l'overhead e migliora le prestazioni del sistema.

## Classificazione dei dispositivi di I/O

A seconda della modalità di trasferimento dei dati, i dispositivi di I/O possono essere classificati in:

- **Dispositivi a caratteri:** trasmettono dati in modo sequenziale, un carattere alla volta (es. tastiere, mouse).
- **Dispositivi a blocchi:** organizzano i dati in blocchi di dimensione fissa, permettendo un accesso diretto alle informazioni (es. hard disk, SSD).

## Tecniche per ottimizzare l'I/O

Per migliorare le prestazioni delle operazioni di I/O, il sistema operativo adotta strategie come:

- **Buffering:** i dati coinvolti in un'operazione di I/O vengono temporaneamente memorizzati in un buffer prima della trasmissione: ciò permette di ridurre il numero di operazioni I/O, adattare la differenza di velocità tra 2 dispositivi; adattare periferiche con blocchi di dimensione diversa; supportare la semantica della copia, ovvero evitare che un processo alteri i dati che deve mandare a un device mentre procede con la sua richiesta in coda: si copiano i dati in un buffer, e questo sarà scritto sul device.

- **Caching:** una cache è una memoria veloce che conserva una copia dei dati letti da una periferica per il riuso veloce.

L'obiettivo è evitare di effettuare accessi multipli alle periferiche, per la lettura di stessi dati.

- **Spooling:** tecnica utilizzata per gestire dispositivi non condivisibili (come le stampanti), Lo spooling considera il **disco** come un enorme **buffer** in grado di

memorizzare tanti lavori per il dispositivo finché i dispositivi di output non sono pronti ad accettarli.

- **Prenotazione delle periferiche (device locking):** impedisce l'accesso simultaneo di più processi a dispositivi non condivisibili, evitando conflitti.
- **Schedulazione delle operazioni di I/O,** ottimizzando l'ordine delle richieste per ridurre i tempi di attesa.

## Rilevamento e gestione degli errori

Il sistema operativo deve anche gestire eventuali errori hardware o di trasferimento dati. Gli errori vengono registrati in strutture dati del kernel, come la **tabella dei file aperti**, che tiene traccia delle risorse di I/O utilizzate dai processi.

---

## Protezione

La protezione dei file in un sistema di elaborazione monoprocesso: si descrivano il concetto di protezione dei file e sue principali realizzazioni, evidenziandone caratteristiche, vantaggi e limiti.

Per **protezione** si intende la messa in sicurezza delle *risorse* (dette anche *oggetti*) da parte di accessi non autorizzati di utenti o processi. Stabilito quest'obiettivo, bisogna distinguere le *regole* (specificano chi e come può utilizzare una risorsa) dai *meccanismi* (gli strumenti che le applicano) per garantire maggior flessibilità.

### Domini di protezione

Un sistema è normalmente composto da risorse sia fisiche (l'hardware) che informative (i file, i programmi), ognuna delle quali è caratterizzata da un nome univoco e da un insieme di operazioni consentite.

Utenti e processi dovrebbero poter accedere a tali risorse secondo il *principio di minima conoscenza*, cioè *accedere* solo a quelle strettamente necessarie per la

propria computazione.

### Struttura del dominio di protezione

Per meglio applicare il principio di minima conoscenza ogni processo viene fatto operare in un **dominio di protezione**, che definisce l'insieme di risorse a cui può accedere ed il tipo di operazioni che è autorizzato a compiere. Per ognuno di essi vengono definiti una serie di *diritti di accesso*, composti da una coppia <nome oggetto, insieme-diritti>. Notare che i domini non devono essere necessariamente disgiunti, ma possono condividere alcuni dei loro diritti.

L'associazione di un dominio ad un processo può essere *statica* o *dinamica*. Nel primo caso il set di risorse disponibili per il processo è fissato all'inizio per tutta la durata dell'esecuzione; Con l'associazione dinamica invece vengono forniti meccanismi per cambiare dominio di protezione, il che meglio applica il principio di minima conoscenza a fronte però di una maggiore complessità.

Un dominio può essere associato a:

- un utente, quindi il cambio di dominio avviene quando un altro utente effettua l'accesso al sistema.
- un processo, quindi si ha un cambio di dominio in concomitanza di un cambio di contesto
- una procedura, in cui si ha un cambio di dominio ad ogni nuova invocazione di procedura

Va però ricordato che nei cambi di dominio (*switch*) la loro struttura rimane inalterata, comprese le operazioni abilitate sulle risorse. Per poter modificare diritti e struttura bisogna averne l'autorità.

### Matrice d'accesso

Il modello di protezione può essere rappresentato in modo astratto con la **matrice di accesso**, in cui le righe indicano i domini di protezione e le colonne gli oggetti.

La cella  $(i, j)$  definisce i diritti d'accesso che il processo nel dominio  $D_i$  ha sulla risorsa  $O_j$

La modifica controllata del contenuto degli elementi della matrice d'accesso è resa possibile da tre operazioni:

- **Copia:** copia l'autorizzazione che un dominio ha su un oggetto a un altro dominio, con o senza diritto di propagarlo.
- **Proprietà:** chi ha il diritto di proprietà su un oggetto può cambiare i diritti che altri domini hanno su quell'oggetto.
- **Controllo:** Consente a chi ha il diritto di controllo su un dominio di cancellare i diritti d'accesso

### **Implementazione della matrice d'accesso**

- **Tabella globale:** è l'implementazione più semplice. Memorizza una lista di terne  $\langle \text{dominio}, \text{oggetto}, \text{diritti} \rangle$ . Un processo in un dominio  $Di$  è autorizzato a eseguire un'operazione  $M$  su un oggetto  $Oi$  se e solo se esiste una terna  $\langle Di, Oi, Ri \rangle$  con  $M \in Ri$ . Occupa molto spazio e non permette di raggruppare oggetti o domini.
- **Liste di controllo degli accessi (Access Control List, ACL):** per ogni oggetto si memorizza una lista di coppie  $\langle \text{dominio}, \text{diritti} \rangle$ . Risponde ai bisogni degli utenti e memorizza informazioni globali ma è inefficiente su grandi sistemi poiché le ACL sono scandite spesso.
- **Liste di capacità dei domini (Capability List, CL):** per ogni dominio si memorizza una lista di coppie  $\langle \text{oggetto}, \text{diritti} \rangle$ . Un oggetto è rappresentato dal suo nome fisico o dall'indirizzo, ed è detto capacità. I domini non sono direttamente accessibili ai processi ma solo dal SO. Rende semplice l'accesso a informazioni sui processi e memorizza informazioni locali ma la revoca è poco efficiente.
- **Meccanismo serratura-chiave:** compromesso tra ACL e CL, ogni oggetto ha una lista unica di bit detta lock, mentre ogni dominio ne ha un'altra detta key. Un processo in esecuzione in un dominio può accedere a un oggetto solo se la sua key è in grado di aprire il lock interessato.

In un sistema di protezione dinamico, può essere necessario revocare dei permessi.



La revoca può essere:

- immediata o ritardata;
- selettiva su un numero limitato di utenti o generale su tutti;
- parziale su un numero limitato di diritti o totale su tutti;
- temporanea o permanente.

Le tecniche per gestire la revoca includono:

- **Riacquisizione:** i permessi vengono periodicamente cancellati e devono essere richiesti nuovamente;
- **Puntatori ai permessi:** si mantiene una lista dei puntatori a ogni permesso associato a un oggetto; per cancellare il permesso mi basta rimuovere il puntatore
- **Indirezione:** i permessi puntano a un valore intermedio che può essere rimosso per revocare l'accesso;
- **Chiavi crittografiche:** ogni oggetto ha una chiave principale unica, che viene associata quando si concede un permesso. La revoca avviene cambiando la chiave principale dell'oggetto, rendendo invalide tutte le abilitazioni precedenti, poiché le chiavi precedenti non possono più essere usate per accedere all'oggetto.

Un ulteriore livello di protezione è offerto dalla **protezione basata sul linguaggio**, che integra i controlli di sicurezza già a livello di compilazione, garantendo maggiore efficienza e flessibilità rispetto ai meccanismi tradizionali basati sul sistema operativo.