Architettura del Sistema Operativo
Il kernel, il gestore dei processi, il gestore della memoria

1. Introduzione

Quando si parla di architettura dei sistemi operativi si fa riferimento, in genere, alla progettazione generale delle varie componenti hardware e software ma, soprattutto, a come queste interagiscono tra di loro e, di conseguenza, alla loro efficacia operativa nel complesso.

Per essere giudicato davvero efficace, un sistema operativo deve essere innanzitutto consapevole dei moduli hardware e software che gestisce, ma deve essere progettato anche alla luce dei programmi che dovrà coordinare e degli utenti che lo utilizzeranno.

In base alle modalità di funzionamento un Sistema Operativo può essere:

- **Mono Tasking e mono utente:**
  - Esecuzione un solo programma applicativo alla volta
  - Viene utilizzato da un solo utente per volta
  - Esempio: MS-DOS
- **Multi Tasking e mono utente:**
  - Consente di eseguire contemporaneamente più programmi applicativi
  - Esempio: Windows.
- **Multi Tasking e multi utente:**
  - Consente l'utilizzo contemporaneo da parte di più utenti
  - Esempio: Unix/ Linux, VMS, MVS (per main frame)

2. Il kernel

Il **kernel** è un componente basilare di ogni sistema operativo con lo scopo di far comunicare le componenti hardware di un computer e i programmi in esecuzione su di esso.

Data la sua importanza, è il primo programma ad essere caricato in memoria quando si accende un computer e l'ultimo ad essere chiuso in fase di spegnimento.

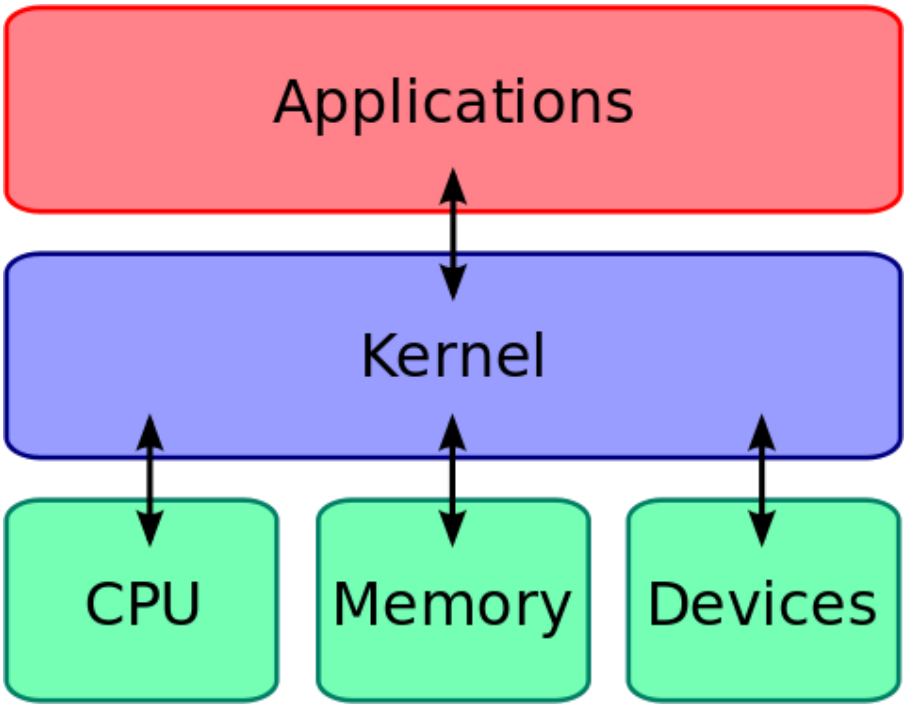

Figura 1: Schema di funzionamento generale di un kernel

Il kernel è responsabile della gestione e dell'allocazione delle risorse del computer, consentendo ad altri programmi di utilizzare queste risorse.

Inoltre, il kernel definisce:

- lo spazio degli indirizzi di memoria per i vari programmi in esecuzione;
- carica in memoria i file con il codice dell'applicazione;

- imposta l'ordine di esecuzione delle varie applicazioni e dei programmi attivi.

2.1 Tipologie di kernel

È possibile individuare 5 tipologie di kernel a seconda di come viene avviene l'accesso alle risorse:

- monolitico;
- microkernel;
- kernel ibrido;
- nanokernel;
- esokernel.

2.1.1 Kernel monolitico

Un **kernel monolitico** comprende in un unico file tutte le funzioni principali di un sistema operativo nonché i driver delle varie periferiche collegate al computer.

Questa tipologia di kernel è costituita da vari moduli, uno per ciascuna delle funzioni da svolgere o per ciascuna delle periferiche da gestire. Questo può portare ad avere un numero di moduli importante con la conseguenza che maggiore sarà il numero di moduli che lo compongono, più grande sarà il file risultante e maggiore sarà il tempo richiesto al sistema per caricarlo in memoria.

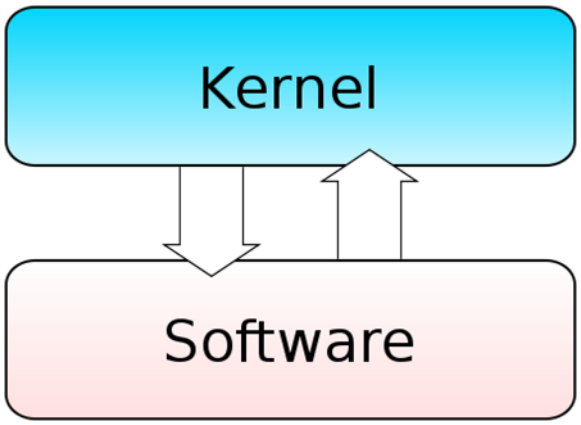

Figura 2: Schema di funzionamento di un kernel monolitico

Poiché si tratta di un kernel totalmente personalizzabile è possibile eliminare in modo semplice tutti quei moduli non necessari o relativi a periferiche non installate. Un kernel di questo tipo, se ben configurato, può occupare pochissimo spazio sul disco rigido garantendo così una riduzione dei tempi di caricamento. La maggior parte del lavoro nei kernel di questo tipo viene svolto utilizzando il meccanismo delle chiamate di sistema, in particolare, i programmi in esecuzione inviano al kernel delle richieste di accesso a determinate risorse hardware e sarà poi il kernel a decidere quali risorse concedere e in che ordine.

Questo approccio "minimalista" porta però con sé diversi svantaggi. Innanzitutto la difficoltà che si incontra ogni qual volta si deve aggiungere una nuova periferica al sistema. Se si volesse utilizzare, ad esempio, uno scanner e nella prima "scrittura" del kernel non fosse stato inserito il relativo modulo, lo scanner non riuscirebbe ad interfacciarsi con le componenti fondamentali del computer e di conseguenza con il sistema operativo. Inoltre, essendo tale kernel realizzato come una struttura in cui i vari moduli sono strettamente collegati l'uno all'altro, un malfunzionamento in una qualsiasi delle parti che lo compongono potrebbe portare un malfunzionamento generale, mettendo così a rischio la stabilità del sistema stesso.

2.1.2 Microkernel

A differenza del kernel monolitico che contiene tutti i moduli necessari al funzionamento del computer, il microkernel implementa solo alcune funzioni basilari quali ad esempio la gestione della memoria, il multitasking e la comunicazione tra processi, affidando le altre funzionalità a programmi terzi chiamati server.

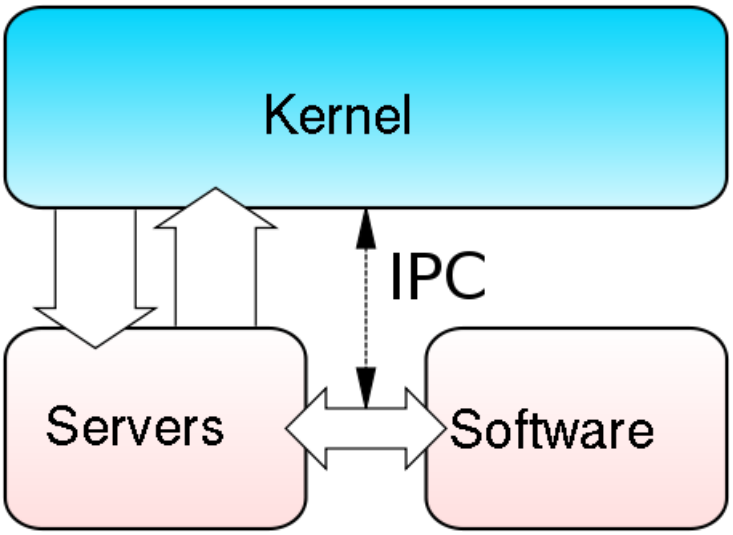

Figura 3: Schema di funzionamento di un microkernel

In un sistema del genere, i driver vengono gestiti come elementi esterni al kernel, facilitando così l'installazione di nuove periferiche. I server, indipendenti e separati l'uno dall'altro, comunicano tra loro attraverso il meccanismo di scambio di messaggi: ogni server può accedere alle risorse e alle funzionalità di un altro server solo attraverso uno scambio di messaggi informativi filtrati dal kernel.

La struttura modulare rappresenta un grande vantaggio in quanto se uno dei server dovesse smettere di funzionare, il sistema può continuare a funzionare, ricaricando il modulo danneggiato nel momento in cui ce ne fosse bisogno.

2.1.3 Kernel ibridi

I kernel ibridi rappresentano una combinazione delle due tipologie di kernel appena descritti, unendo i punti di forza dell'uno e dell'altro.

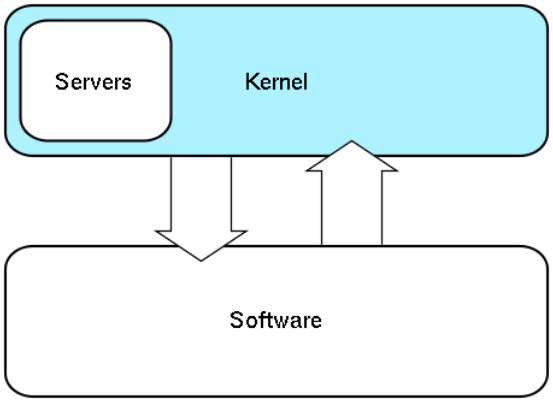

Figura 4: Schema di funzionamento di un kernel ibrido

Tali kernel possono essere considerati come un'estensione dei microkernel poiché l'approccio strutturale è più o meno lo stesso, ma il kernel ibrido è formato da un numero maggiore di "moduli primari". I kernel ibridi sono impiegati in molti sistemi operativi commerciali come ad esempio Microsoft Windows e Mac OS.

All'interno di un kernel ibrido sono caricati alcuni moduli "non essenziali" che consentono di migliorare la gestione delle risorse.

Il gestore del sistema, o lo sviluppatore del sistema operativo, dovrà assicurare che il numero di moduli sia il minore possibile.

2.1.4 Exokernel

L'approccio strutturale e funzionale degli **esokernel** è totalmente differente rispetto a quello utilizzato dai kernel visti sino ad ora. Questi ultimi, infatti, operano una sorta di filtro tra software e

hardware, "nascondendo" così ai programmi le risorse hardware a loro disposizione. Ad esempio, un programma in esecuzione non potrà conoscere in quale parte di memoria è stata allocato o in quale settore del disco è stato scritto un file ad esso collegato.

Tale scelta è legata principalmente alla sicurezza in quanto il kernel, occupandosi sia della gestione delle risorse sia dei rischi connessi all'esecuzione di programmi dannosi, in questo modo cerca di minimizzare l'esecuzione di questi ultimi.

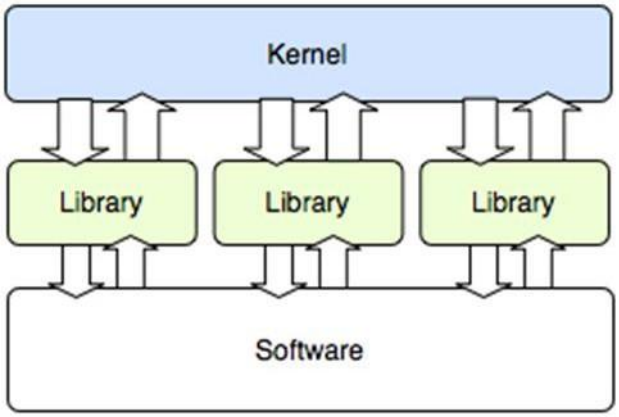

Figura 5: Schema di funzionamento di un Exokernel

L'exokernel ha lo scopo di eliminare il livello di astrazione che di nasconde le risorse hardware ai processi software in esecuzione. Ad esempio un'applicazione può richiedere l'accesso a determinate risorse del sistema scegliendo la porzione di memoria dove essere allocata o il settore del disco rigido su cui essere salvata.

L'unico compito dell'exokernel è quello di assicurarsi che le risorse richieste siano disponibili e che il programma possa accedervi senza problemi.

3. Gestore dei processi

Un **processo** è un'istanza in esecuzione di un programma che richiede determinate risorse, quali CPU, memoria e dispositivi di I/O.

Ogni processo, per l'esecuzione necessita di risorse hardware e software.

Tali risorse possono essere:

- "**consumabili**": si tratta di una risorsa che non può essere riutilizzata. Esempi di risorse consumabili sono ad esempio i segnali;
- "**riusabili**": si tratta di una risorsa che può essere utilizzata da un singolo processo alla volta e non viene distrutta dopo l'utilizzo. Esempi di risorse riusabili sono i dischi (risorse hardware) o i file (risorse software).

Gestire queste risorse è compito del Sistema Operativo.

Il ***gestore dei processi*** si occupa di come i programmi, residenti contemporaneamente in memoria centrale, sono eseguiti utilizzando il processore (o i processori) di cui è dotato il sistema così da ottimizzare tale risorsa critica.

La gestione dei processi coinvolge varie attività come:

- creazione e cancellazione dei processi;
- sospensione e riesumazione dei processi;
- fornire meccanismi per sincronizzazione dei processi;
- comunicazione tra processi;
- evitare, prevenire e risolvere le situazioni di stallo (deadlock).

3.1 Lo scheduler

Il gestore dei processi è detto **Scheduler**. Lo scheduler ha il compito di scegliere quale processo tra quelli attivi deve essere scelto per l'esecuzione.

Esistono tre tipi di scheduler:

1. **scheduler a breve termine**: conosciuto anche come CPU Scheduler è, lo scheduler che decide quale dei processi in memoria deve essere assegnato alla CPU;
2. **scheduler a medio termine**: sposta temporaneamente i processi dalla memoria principale e alla memoria secondaria e viceversa.
3. **scheduler a lungo termine**: questo tipo di scheduler stabilisce quali processi devono essere ammessi alla coda pronta; ovvero, quando si cerca di eseguire un programma, il suo inserimento nella lista dei processi attualmente in esecuzione viene autorizzata o ritardata da questo scheduler. Lo scheduler a lungo termine è anche responsabile del controllo del grado di multielaborazione.

Lo **scheduling** è l'algoritmo di gestione della risorsa, che consente la scelta di assegnare o togliere una risorsa al processo che l'ha richiesta.

Quando un nuovo processo viene creato, viene inserito in quella che viene chiamata la **tabella dei processi**. La tabella dei processi serve al sistema per tener traccia di quali sono i processi esistenti al fine di determinare qual è il prossimo processo da far avanzare per un po' di tempo così da dare l'illusione all'utilizzatore che l'esecuzione di tutti i processi venga fatta avanzare simultaneamente.

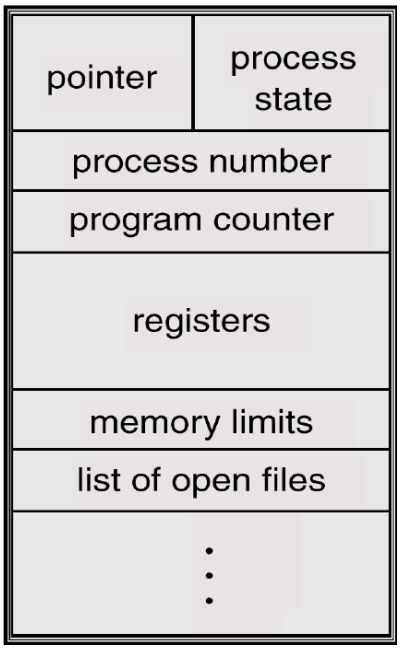

Figura 6: Tabella dei processi

I processi possono eseguire delle chiamate a funzioni di sistema per ottenere dei servizi dal sistema operativo relativi alla gestione dei processi stessi, in particolare le chiamate sono:

- **exec** per l'esecuzione di altri processi;
- **fork** per la replica del processo in esecuzione;
- **wait/signal** per l'invio di segnali da un processo ad un altro;
- **kill/terminate** per terminare un processo.

Mediante la system call "fork" il sistema operativo, a partire da un processo iniziale di avvio del sistema stesso, è in grado di eseguire, in base alla sua configurazione, altri processi in cascata, creando così una gerarchia ad albero "padre/figlio" tra i processi eseguiti sul computer: la radice è il processo di inizializzazione del sistema.

3.2 Stati di un processo

Ogni processo può attraversare diverse fasi, definite più precisamente **stati**, di seguito riportati:

- **nuovo**: quando l'utente lancia un programma e la CPU lo carica nella RAM, in questa fase il processo viene di fatto creato;
- **pronto**: una volta creato, il processo è, per l'appunto, pronto all'esecuzione da parte dello scheduler del sistema operativo;
- **in esecuzione**: una volta che il processo è pronto, lo scheduler comincia ad eseguirlo attraverso il processore, è una fase attiva in cui il programma sta effettivamente lavorando.
- **terminato:** quando il programma è stato eseguito, le risorse macchina utilizzate (RAM e CPU) vengono liberate e rese disponibili per altri task;
- **Sospeso/bloccato/in attesa**: in processo è di fatto fermo, in attesa di eventi che ne alterino lo stato corrente, può succedere a causa di errori o bug nel programma.

Figura 7: Diagramma degli stati di un processo

Alla sua di creazione un processo si trova nello stato New e passerà nello stato di Ready, pronto per essere eseguito solo quando il sistema operativo lo riterrà opportuno.

Nello stato di Ready ci possono essere più processi in attesa di passare allo stato di Running.

Lo stato Running è lo stato di effettiva esecuzione; ovviamente un computer monoprocessore potrà avere un solo task in questo stato. Durante la fase di Running un processo possiede in modo esclusivo la risorsa CPU. Dallo stato di Running un processo può passare in due stati diversi:

1. stato di **waiting**:
    a. quando non ha più bisogno della CPU perché deve effettuare un'operazione di I/O (fase di I/O-burst);
    b. oppure perché il processo è in attesa di un dato che deve essere prodotto da un altro processo che al momento non è ancora pronto. In ogni caso il sistema operativo mette il processo in attesa finché non si ripresentano le condizioni per riavviare la fase di CPU-burst;
2. stato di **terminate**: quando il processo ha finito le istruzioni da eseguire. In questo caso il processo termina di esistere.

Il cambiamento di stato da running a terminate determina che il processo ha terminato definitivamente l'esecuzione; esso rilascia oltre alla risorsa CPU anche la memoria centrale.

Quando un processo rilascia la CPU, per uno dei motivi esposti, questa è assegnata, dal sistema operativo, ad un altro processo che è nello stato di ready.

Quando un processo che è in stato di waiting ottiene quanto richiesto, per riavere a disposizione la CPU, deve ritornare allo stato di ready e qui attendere il proprio turno.

In un sistema multitasking le scelte fatte per la condivisione delle risorse (CPU, memoria e periferiche) sono di fondamentale importanza poiché le prestazioni del computer sono strettamente legato alla qualità di questo lavoro.

4. Il gestore della memoria

La gestione della memoria è uno degli aspetti più importanti dei sistemi operativi.

Per poter essere eseguito un processo deve risiedere necessariamente in memoria centrale.

Alcune definizioni sono necessarie, in particolare definiamo:

- lo **spazio fisico** come l'insieme degli indirizzi accessibili fisicamente che costituisce la memoria fisica;
- lo **spazio logico** come uno spazio astratto di indirizzi che costituisce la memoria logica.

Affinché un processo venga eseguito è necessario mappare lo spazio logico sullo spazio fisico, cioè sulle effettive locazioni di memoria che conterranno dati e istruzioni.

Il sistema operativo deve quindi creare una corrispondenza tra il processo virtuale e il processo vero e proprio e lo fa attraverso un componente detto **MMU** (*Memory Management Unit*) il SO, con il quale in particolare converte l'indirizzo logico della memoria virtuale in indirizzo fisico.

Per analizzare il funzionamento i questo componente è importante conoscere il concetto di **memoria virtuale**.

In genere i programmi eseguiti da un calcolatore occupano una dimensione maggiore rispetto a quella disponibile in memoria, quindi come è possibile che un programma, qualsiasi sia la sua dimensione, venga comunque caricato in memoria?

La prima soluzione adottata, consisteva nel ridurre la dimensione dei programmi, in particolare venivano scelti algoritmi che prevedevano elaborazioni più lente così da dover richiedere meno risorse.

Una soluzione alternativa e più efficiente consisteva nel dividere il programma in un numero finito di pezzi detti **overlay.** Quando iniziava l'esecuzione del programma, il primo overlay veniva prelevato e caricato in memoria, per poi essere sostituito dal successivo dopo la sua esecuzione e, così via fine al termine del programma.

Una tecnica oggi ancora utilizzata e conosciuta con il termine di memoria virtuale va a sostituire le precedenti nel 1961.

Tutte le tecniche di gestione della memoria viste prevedono che il codice di un processo risieda interamente in memoria centrale in una zona contigua.

Con la tecnica della memoria virtuale tale vincolo cade, riducendo notevolmente i problemi di frammentazione ed eliminando del tutto il problema della riorganizzazione della memoria. In tale contesto, il caricamento di un programma non riguarda più l'intero suo codice, bensì solo una prima parte, il resto è allocato in memoria di massa (in una zona riservata) pronto ad essere portato in memoria centrale qualora contenga qualche istruzione da eseguire.

La memoria virtuale si basa su un semplice concetto: se si deve eseguire un processo il cui codice è costituito, per esempio, da 100.000 istruzioni, è inutile caricarle tutte in memoria centrale; infatti, se il computer, in un determinato istante, sta eseguendo l'istruzione numero 35 probabilmente per un certo periodo di tempo non avrà bisogno dell'istruzione numero 95.366, per cui quest'ultima può essere tranquillamente riposta in memoria di massa e richiamata all'occorrenza.

Le principali tecniche di implementazione della memoria virtuale sono la **paginazione** e la **segmentazione**. La differenza sostanziale tra le due tecniche sta nella formazione della partizione che deve contenere il codice di programma da eseguire; nella tecnica della paginazione essa è costante, mentre in quella della segmentazione le partizioni sono variabili.

4.1 La paginazione

Con la paginazione la memoria centrale è suddivisa in blocchi aventi tutti la stessa dimensione e lo spazio degli indirizzi di un programma è suddiviso logicamente in parti chiamate **pagine**. In particolare, nella paginazione la suddivisione del programma in blocchi avviene in base a criteri geometrici, la singola pagina non ha alcun significato per il programmatore e non c'è

nessun collegamento logico tra il contenuto di un blocco e quello del blocco successivo. Quindi la paginazione è invisibile al programmatore.

I blocchi hanno la stessa dimensione delle pagine (in genere multipli di 2k) e l'allocazione del programma avviene quando in memoria centrale esiste un numero di blocchi liberi pari o superiori al numero di pagine da allocare. Mentre le pagine sono logicamente contigue, i blocchi di memoria in cui verranno caricate possono non esserlo. Se un programma è costituito da 4 pagine, al momento della sua allocazione il gestore della memoria dovrà trovare 4 blocchi (anche sparsi) liberi. Per fare questo il sistema operativo deve gestire alcune tabelle di abbinamento come ad esempio la **tabella delle pagine** che contiene dettagli su dove sono archiviate le pagine.

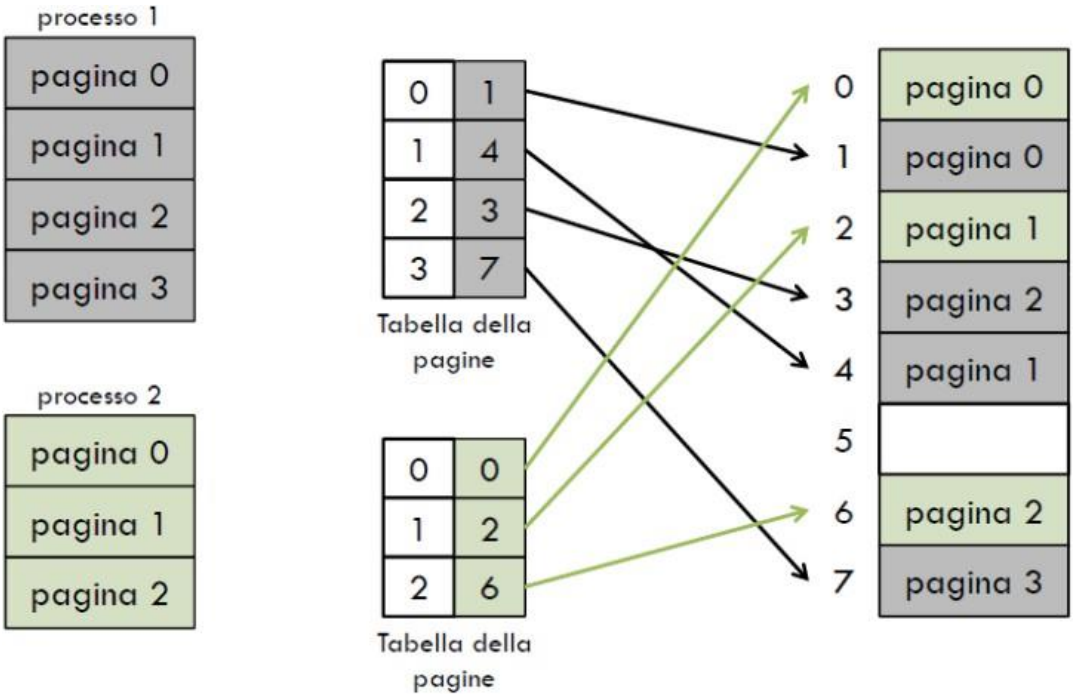

Figura 8: Un esempio di paginazione

La tabella delle pagine contiene dettagli su dove sono archiviate le pagine.

4.2 La segmentazione

Nella **segmentazione** i processi vengono suddivisi in blocchi di dimensioni differenti detti **segmenti**.

Con questa tecnica un processo è suddiviso in blocchi seguendo criteri logici che dipendono da come il software è organizzato, ad esempio un programma potrebbe essere composto da tanti segmenti quanti sono i moduli che lo compongono: un segmento per il programma principale, uno per ogni sottoprogramma e così via.

La segmentazione riflette quindi la visione che il programmatore ha del proprio programma.

Per memorizzare un segmento, il sistema operativo deve cercare una zona libera di dimensione pari a quella del segmento da allocare, per fare questo il sistema operativo deve gestire quella che viene chiamata **tabella dei segmenti**. Tale tabella contiene tutte le informazioni sul posizionamento dei segmenti di un processo ed è usata dalla MMU per costruire l'indirizzo fisico di una cella di memoria.

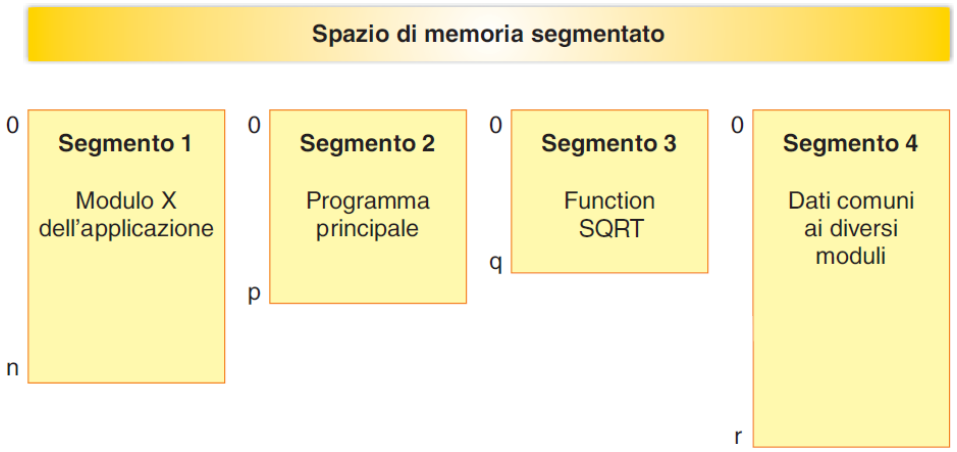

Figura 9: Esempio di segmentazione

I vantaggi della segmentazione sono:

- i dati da condividere possono essere inseriti in un unico segmento condiviso poi tra i processi quindi la gestione degli spazi di memoria condivisi risulta più semplice;

- la protezione della memoria può essere effettuata in modo puntuale in base alle esigenze e alle caratteristiche funzionali del singolo segmento;
- Se nell'esecuzione del processo lo spazio occupato dai dati aumenta è sufficiente aumentare la dimensione di quello specifico segmento per cui è più semplice gestire strutture dati dinamiche allocandole in un segmento.