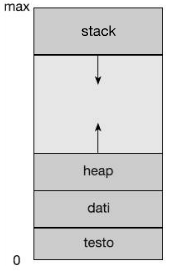
**Sistemi operativi**

**Teoria:**

* **Capitolo 1-2:** Lettura veloce sono semplici concetti introduttivi.
* **Capitolo 3:** Tutto tranne gli ultimi due paragrafi
  + Leggere con attenzione, sono cose che verranno chieste all’esame.
* **Capitolo 4:** Solo il paragrafo introduttivo.
* **Capitolo 5:** Tutto il capitolo tranne fino al 5.4 più:
  + concorrenza dei processi domanda intercorso probabilmente.
  + **Concorrenza** (non ci sta sul libro)
  + **Algoritmo del fornaio** (non ci sta sul libro)
* **Capitolo 5:** Tutto il capitolo da 5.6 tranne il 5.5-5,9 e 10.
* **Capitolo 6:** Tutto tranne 6.4, 6.5, 6.6, 6.7 (di questo solo la parte su linux).
* **Capitolo 7:** Solo 7.1, 7.2, 7.3.
* **Capitolo 8:** Da 8.1 (l’overlay non ci sta) a 8.6.
* **Capitolo 9:** Da 9.1 a 9.6.
* **Capitolo 10:** 10.1, 10.2, 10.4, 10.5, 10.6, 10.7.
* **Capitolo 11:** tutto.

**Capitolo 3:**

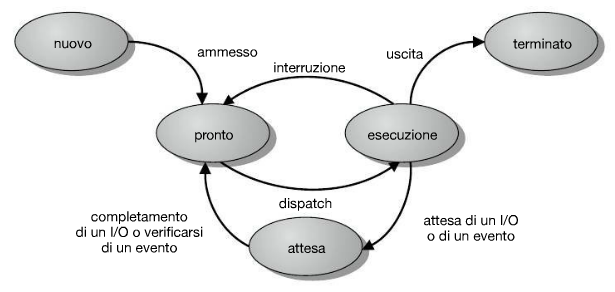
**3.1 Concetto di processo:** Con il termine **job** o **processo** si intende un’attivita svolta dalla CPU, che utilizza risorse di elaborazione. All’interno di un sisitema concorrente i processi sono molteplici, e concorrono per l’utilizzo della CPU.

**3.1.1 Processo:** Informalmente un **processo** è un programma in esecuzione esso è un entità attiva, quindi diversa dal semplice **programma** o **file di testo** che è un entita passiva. Un processo è composto da: 

* **Valore del contatore di programma** e dal contenuto dei **registri della cpu.**
* **Dal proprio Stack:** che contiene i suoi dati temporanei.
* **Una sezione di dati:** contentente le variabili globali del processo.
* **Un heap:** ossia la memoria allocata dinamicamente dal processo.

**3.1.2 Stato del processo:** un processo durante la sua esecuzione passa attraverso a diversi stati:

* **Nuovo:** Il processo viene creato.
* **Esecuzione (running):** le istruzioni vengono eseguite.
* **Attesa (waiting):** il processo attende che si verifichi qualche evento come un’operazione di **I/O.**
* **Pronto (ready):** il processo attende che gli sia assegnata la CPU.
* **Terminato:** il processo termina l’esecuzione.

Diagramma degli stati:

**3.1.3 Blocco di controllo dei processi:** ogni processo è rappresentato nel sistema operativo tramite un **Process Controll Block PCB** (o Task Control Block TCB) che contiene le seguenti informazioni in merito ad un determinato processo: 

* **Stato del processo:** nuovo, pronto, esecuzione ecc.…
* **Contatore del programma:** che contiene l’indirizzo della prossima istruzione del programma.
* **Registri della CPU:** che comprende lo stato dei:
  + Registri **accumulatori**.
  + Registri **indice**.
  + Puntatori allo **stack.**
  + Registri di **uso** **generico.**
  + Registri contenenti i **codici** **di** **condizione.**
* **Informazioni sullo scheduling di CPU:** che contengono informazioni riguardanti la priorità del processo e tutti gli altri parametri inerenti alla coda di scheduling.
* **Informazioni di gestione della memoria:** che contengono informazioni inerenti al valore dei registri di base e di limite oltre alle tabelle dei segmenti.
* **Informazioni sull’accounting:** che contengono informazioni riguardanti la quota di uso della CPU e il tempo di utilizzo della stessa.
* **Informazioni sullo stato dell’I/O:** che contengono informazioni riguardanti i dispositivi di I/O collegati al processo.

**3.2 Scheduling dei processi:** L’obbiettivo della **multiprogrammazione** è quello di tenere il processore sempre a lavoro in modo da non avere tempi morti. L’obbiettivo del **time** **sharing** è quello di commutare l’uso della CPU tra i vari processi in modo di dare all’utente la possibilità di utilizzare più applicazioni contemporaneamente. Per fare ciò i processi prima di essere mandati in esecuzione si trovano in una coda gestista dallo **Scheduler dei processi** che ha il preciso compito di selezionare quale sarà il prossimo processo ad ottenere la CPU.

**3.2.1 Code di scheduling:** All’interno del sistema operativo ogni processo è inserito all’interno di una **coda dei processi (job queue).** I processi già caricati in memoria centrale e quindi pronti all’esecuzione si trovano invece all’interno della **coda dei processi pronti (ready queue).**

Queste code si presentano come delle liste concatenate fra i vari **PCB** dei processi in coda.

Oltre alla ready e alla job queue il sistema operativo ha anche altre code, infatti nel caso in cui più processi abbiano la necessità di accedere ad un qualche dispositivo (per esempio un disco rigido o la tastiera) vengono create delle nuove code in quando i dispositivi (soprattutto quelli di I/O) hanno una velocità estremamente bassa comparata a quella della CPU, questa particolare coda prende il nome di **coda dei dispositivi.**

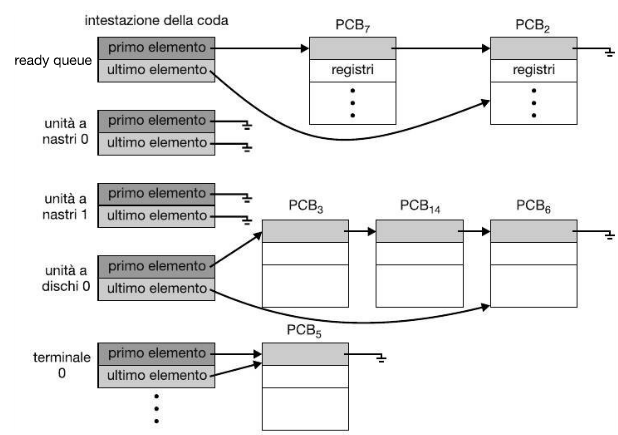
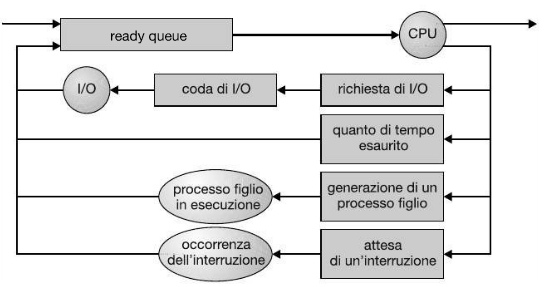
****

Diagramma di accodamento per lo scheduling dei processi (i cerchi rappresentano le risorse necessarie al processo):



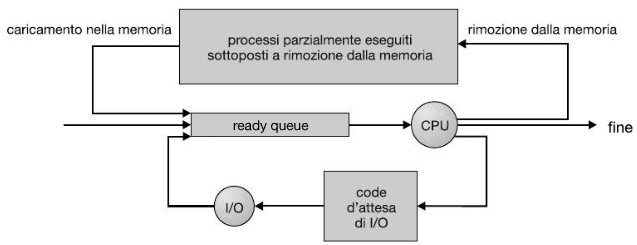
**3.2.2 Scheduler:** è quella parte del sistema operativo che si occupa di selezionare, dalla coda dei processi pronti, i prossimi processi che otterranno la CPU. All’interno dei sistemi batch può però accadere che si creino più processi di quanti se ne possano eseguire. Questi processi vengono per ciò trasferiti nella memoria secondaria dove attendono di essere eseguiti (**spooling**). Per questo motivo esistono vari tipi di **Scheduler:**

* **Scheduler a breve termine** o **Scheduler della CPU:** Si occupa di selezionare i processi presenti nella coda dei processi pronti.
* **Scheduler a lungo termine:** Si occupa di selezionare i processi presenti nella memoria secondaria. Questo scheduler è anche preposto al controllo del **grado di multiprogrammazione** cioè il numero di processi presenti in memoria (processi creati=processi finiti tutto apposto). Questo scheduler si occupa anche di creare un buon mix tra processi di tipo:
  + **I/O Bound:** Impiega la maggior parte del tempo in operazioni di I/O.
  + **CPU Bound:** Impiega la maggior parte del tempo in operazioni di elaborazione.

In modo da massimizzare il tempo di utilizzo di tutti gli apparati dell’elaboratore.

Alcuni sistemi operativo inoltre non utilizzano scheduler a lungo termine, in questi casi la stabilità del sistema dipende:

* **Da limiti fisici dell’elaboratore**
* **Dagli stessi utenti:** che quando si accorgono che il sistema inizia a rallentare chiudono alcuni programmi.

**Scheduler a medio termine:** L’idea alla base di tale scheduler è che a volte può essere vantaggioso eliminare alcuni processi dalla memoria, riducendo il grado di multiprogrammazione del sistema, per poi ricaricarli in secondo momento. Questa operazione prende il nome di **Avvicendamento dei processi in memoria** (**swapping**).

**3.2.3 Cambio di contesto:** Le interruzioni forzano il sistema a sospendere il lavoro attuale della CPU per eseguire routine del kernel. Quando si verifica un’interruzione il sistema deve salvare il **Contesto** del processo attualmente in esecuzione, per poterlo successivamente ripristinare in un secondo momento. Tale operazione prende il nome di **Cambio di contesto** (**context switch**).

**3.3 Operazioni sui processi:** Nella maggior parte dei sistemi operativi i processi si possono eseguire in modo concorrente, e si devono poter creare e cancellare dinamicamente, a tal fine il sistema operativo deve fornire dei meccanismi per permettere di creare e terminare processi.

**3.3.1 Creazioni di un processo:** Durante la propria esecuzione ogni processo può creare numerosi nuovi processi, in questi casi il processo creante si chiama **processo padre**, mentre il processo creato **processo figlio** a sua volta ogni processo figlio può diventare padre di altri processi dando vita ad ‘una struttura ad albero (per esempio in UNIX tutti i processi discendo dal processo **INIT** quello che avvia il sistema)**.** Nella maggior parte dei sistemi operativi ogni processo è identificato tramite un **identificatore univoco del processo** o **PID** (**process indetifier**). All’atto di creazione di un processo figlio abbiamo diverse possibili situazioni:

Per quanto riguarda l’esecuzione:

1. Il processo figlio continua l’esecuzione in modo concorrente al processo padre
2. Il processo padre attende la terminazione di tutti i processi figlio.

Per quanto riguarda l’indirizzamento:

1. Il processo figlio è un duplicato del processo genitore (ha lo stesso programma e dati del padre).
2. Nel processo figlio di carica un nuovo programma.

**3.3.2 Terminazione di un processo:** Un processo termina quando finisce l’esecuzione della sua ultima istruzione e inoltra la richiesta al sistema operativo di essere cancellato **exit ();** a questo punto il processo figlio può inviare un’informazione di stato al processo genitore che la riceve tramite la chiamata di sistema **wait ().** La terminazione di un processo può avvenire anche per mano di un altro processo, per esempio il processo padre può terminare il processo figlio per le seguenti cause:

1. Il processo figlio ha ecceduto nell’uso di alcune delle risorse che sono state assegnate
2. Il compito svolto dal figlio non è più richiesto
3. Il processo genitore termina e il sistema operativo non consente a un processo figlio di continuare in tale circostanza.

In alcuni sistemi se un processo genitore termina si devono terminare anche i suoi figli, questa operazione è detta **terminazione a cascata** ed è di solito avviata dal sistema operativo (Spegnimento del sistema processo init?).

Un processo figlio che ha terminato la sua esecuzione senza che il padre chiamasse la wait () viene detto processo **zombie,** di solito un processo rimane zombie solo per un breve periodo di tempo.

Può anche capitare che un processo padre termini senza mai chiamare la wait () facendo rimanere **orfani** i processi figli, in questo caso sta al sistema operativo identificare i processi orfani e terminarli (per esempio in UNIX il processo init invoca periodicamente la chiama di sistema wait () in modo da raccogliere eventuali processi orfani).

**3.4 Comunicazione tra processi:** I processi eseguiti concorrentemente nel sistema operativo possono essere di due tipi:

* **Indipendenti:** Se non influisce su altri sistemi e non subisce influssi da altri sistemi.
* **Cooperante:** Se influenza o viene influenzato da altri processi.

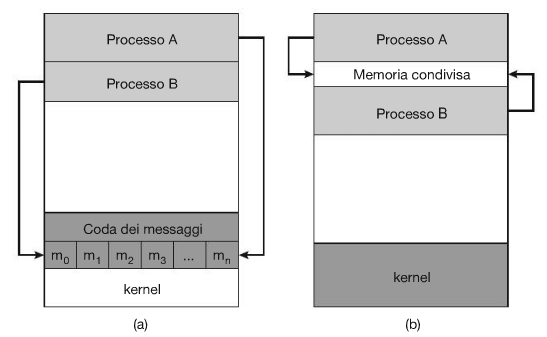
Un abbiente che consente la cooperazione tra processi offre i seguenti vantaggi:

* **Condivisione di informazioni:** più utenti possono essere interessati alle stesse informazioni che devono essere utilizzabili in modo concorrente.
* **Velocizzazione del calcolo:** Alcune operazioni sono realizzabili più velocemente se suddivise in sotto-operazioni.
* **Modularità:** Può essere utile la realizzazione di un sistema modulare che suddividi le funzioni di sistema in processi.
* **Convenienza:** Anche solo un utente può avere la necessità di compiere più attività contemporaneamente.

I meccanismi che regolano lo scambio di dati tra processi vengono detti **IPC** (interprocess communication).

I modelli fondamentali della comunicazione tra processi sono due:

* **Memoria condivisa (buffer):** All’interno del quale si stabilisce una porzione di memoria condivisa dove i processi scrivono e leggono dati. Questo metodo è particolarmente efficiente nel caso in cui si debbano scambiare grandi quantità di dati, ma soffre di problemi di coerenza.
* **Scambio di messaggi:** In questo modello i processi si scambiano messaggi tra di loro, questo metodo e particolarmente efficace in caso di sistemi multicore ma soffre nel caso in cui si debbano scambiare messaggi molto grandi.



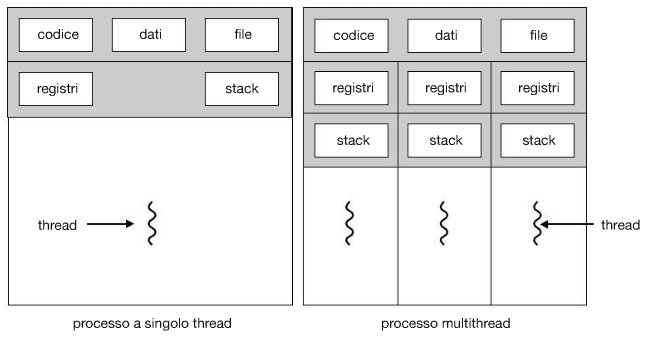
**Capitolo 4:**

**4.1 I Thread:** Nei moderni sistemi di elaborazione ogni processo può eseguire più operazioni concorrentemente (dette per l’appunto Thread). Un thread può essere veduto come l’unità base della CPU, ogni thread condivide con gli altri thread:

* Un identificativo **ID**
* Un contatore al **programma**
* Una serie di **registri**
* Uno **stack**

Oltre a condividere con i thread appartenenti allo stesso processo, tutte le risorse necessarie all’esecuzione (linee di codice, parti di memoria ecc.…).

**4.1.1 Motivazioni:** Il concetto di thread nasce dalla necessità di avere processi che svolgono più operazioni concorrentemente (per esempio un editor di testo ha necessità di svolgere molti compiti alla volta come: creare la grafica, controllare gli errori e gestire gli input), visto che, la creazione di nuovi processi è un’attività onerosa sia in termini di tempo che di risorse (esempio del server con tanti clienti).



**4.1.2 Vantaggi:** I vantaggi della programmazione **multithread** si possono classificare in 4 principali categorie:

* **Tempo di risposta:** Rendere un’applicazione multi thread può diminuire di molto i tempi di risposta, soprattutto per quanto riguarda la creazione di **GUI**. Infatti, se non esistessero i thread quando andremo a cliccare su un programma il sistema rimarrebbe bloccato finché l’applicazione non venga aperta.
* **Condivisione delle risorse:** I normali processi possono scambiarsi informazioni sono tramite memoria condivisa o messaggi, metodi che devono essere implementati dal programmatore. I Thread invece condividono di default lo spazio di memoria del processo a cui appartengono.
* **Economia:** Dal punto di vista delle risorse e di tempo è molto più economico creare un thread (visto che condivide le risorse del processo a cui appartiene) rispetto a creare un nuovo processo.
* **Scalabilità:** I vantaggi della programmazione multithread sono ancora maggiori su sistemi multiprocessore visto che i vari thread possono essere eseguiti contemporaneamente sui vari processori fisici. Cosa che non accade con processi mono-thread che vengono eseguiti su un singolo processore fisico per volta.

**Capitolo 5:**

**La concorrenza** (non presente sul libro ma **importante**): è una caratteristica dei sistemi di elaborazione nei quali può verificarsi che un insieme di processi o sotto processi (thread) computazionali sia in esecuzione nello stesso istante. La concorrenza appare in tre diversi contesti:

* **Applicazioni multiple:** quando più applicazioni si contendono l’uso della CPU per massimizzare il tempo di calcolo.
* **Applicazioni strutturate:** applicazioni programmate come insiemi di processi concorrenti.
* **Struttura del S.O.:** Spesso anche i S.O. sono implementati come insiemi di processi.

In un sistema a singolo processore con multiprogrammazione i processi si alternano l’utilizzo della CPU, questo limita i tempi morti portando ad ‘un aumento di prestazioni, e importante notare che in questo tipo di sistemi i processi possono essere anche sovrapposti. Inoltre, non è possibile predire la velocità relative dei processi in quanto dipendono dalle attività degli altri processi.

In questo tipo di sistema si presentano le seguenti problematiche:

* **Pericolosità della condivisione:** due processi utilizzano la stessa variabile o le stesse risorse, cosa che potrebbe portare a l'inconsistenza dei dati.
* **Gestione dell’assegnazione delle risorse:** per esempio un processo può richiedere l’accesso ad un canale di I/O ed essere bloccato subito dopo prima di poterlo usare.
* **Verifica degli errori:** Trovare un errore di programmazione diventa molto difficile in quanto i risultati sono interlacciati tra i vari processi (vedi l’esercizio di lab lezione 1 sulla fork).

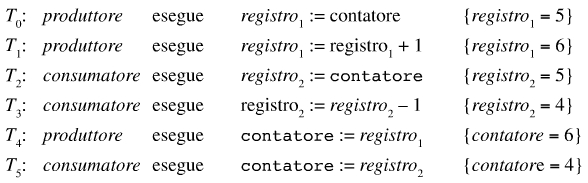
**Principi** di **progettazione** di **gestione** che nascono a causa della concorrenza:

* Il S.O. deve poter tenere traccia dei processi attivi.
* Il S.O. deve allocare e deallocare varie risorse per ogni processo attivo, ed es:
  + Tempo di elaborazione
  + Memoria.
  + File.
  + Dispositivi di I/O.
* Il S.O. deve proteggere i dati e le risorse fisiche di ogni processo da interferenze da parte di altri processi.
* Il risultato di un processo deve essere indipendente dalla sua velocità di esecuzione
  + Relativamente a quelle degli altri processi concorrenti.

**5.1-2 Introduzione e problema della zona critica:**

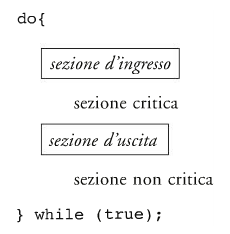
All’interno di sistemi concorrenti la condivisione di informazioni tra processi può portare a situazioni ambigue, visto che all’interno di questi sistemi i processi vengono bloccati e sostituiti all’interno della CPU durante la loro esecuzione.

**Esempio:**

****

Per evitare queste situazioni occorre garantire che un solo processo alla volta possa modificare i dati presenti all’interno dell’area di memoria condivisa, in modo da completare le sue operazioni. La zona che in cui il processo accede alle variabili condivise viene detta:

* **Sezione Critica**: un segmento del codice nel quale il processo potrà:
  + Modificare delle variabili condivisi.
  + Scrivere un file e così via, modificare quindi dati condivisi.

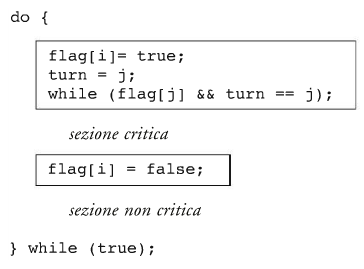
Una caratteristica fondamentale che il sistema deve avere per gestire questo problema è che quando un processo è in sezione critica nessun altro processo può nella sua sezione critica. In particolare, ogni processo per entrare nella propria sezione critica deve ottenere il permesso, la sezione del codice che occupa di richiedere questo permesso viene detta **sezione di ingresso**. Inoltre, la sezione critica può essere seguita da una **Sezione di uscita**, mentre il resto del programma viene detto **Sezione non critica.** In generale una soluzione al problema della sezione critica deve soddisfare tre caratteristiche:

* **Mutua esclusione:** Se un processo P è in sezione critica nessun altro processo deve entrare in sezione critica.
* **Progresso:** Se nessun processo è in sezione critica e qualche altro processo desidera entrarci, possono partecipare alla selezione per entrare in sezione critica solo i processi che non si trovano nella sezione non critica.
* **Attesa limitata:** Se un processo P ha più volte fatto richiesta di entrare in sezione critica, senza successo. Esiste un limite di volte in cui si consente ad altri processi di passare avanti al processo P prima che quest’ultimo venga scelto.

È importante notare che Il problema della sezione critica è al carico del programmatore in quanto sta a lui identificare la sezione critica del programma e quindi implementare di conseguenza la sezione di ingresso e di uscita, oltre che gli algoritmi per organizzare l’ingresso nella sezione critica dei processi.

**5.3 Soluzione di Peterson (si aspetta lo pseudocodice con descrizione e dimostrazione):** Una soluzione del problema della sezione critica è quella di Peterson. Questa soluzione è limitata a due processo **P0** e **P1**, ognuno dei quali esegue alternativamente la propria sezione critica. La soluzione richiede che entrambi i processi condividano i seguenti dati: 

Dove **turn** indica il turno (se vale **0** tocca a **P0**, se vale **1** tocca a **P1**), mentre il **flag** è un vettore che indica se il processo presente nella casella **i** è pronto ad entrare in sezione critica.

Praticamente se il processo **P0** vuole entrare in sezione critica setta il **flag [0]** su **true** e **turn** su **1** (cede il posto al suo compagno). 

Ora se anche **P1** desidera entrare in sezione critica esegue le stesse operazione di **P0** settando **turn** su **0** (anche lui cede il posto al suo compagno).

**A questo punto uno dei due valori verrà sovrascritto** (visto che turn abbiamo detto essere una variabile condivisa), il valore dell’identificatore “sopravvissuto” identificherà il processo ad entrare in sezione critica.

L’algoritmo è verificato perché assolve a tutte le caratteristiche necessarie per risolvere il problema:

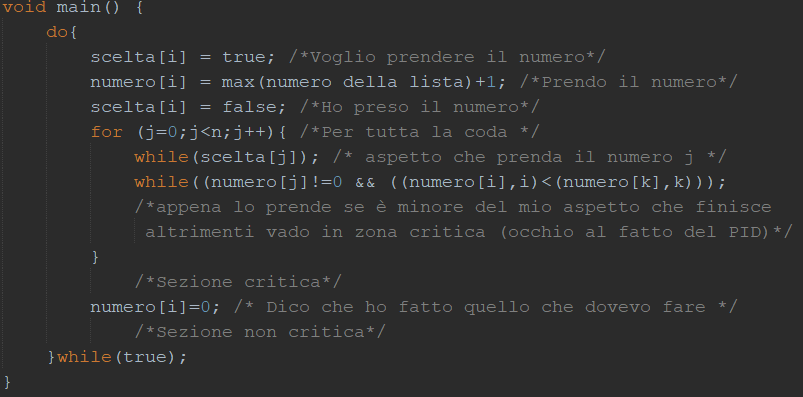
* **Mutua esclusione:** I processi entrano in sezione critica solo se il loro **flag** è uguale a **true** e **turn** è uguale al loro identificatore, che essendo una variabile intera condivisa aggiornata da entrambi i processi può assumere solo un valore.
* **Progresso e Attesa limitata:** Se uno dei due processi non è in sezione critica l’altro ci entra senza problemi istantaneamente, nel caso un processo non sia stato scelto per la sezione critica, al termine della sezione critica del suo compagno (quando imposta il suo flag a false) il processo escluso verrà automaticamente scelto per la sezione critica.

**Soluzione per più processi, algoritmo del fornaio:**

Si basa sull’implementazione di una sorta di salta coda (la fila con il numeretto dal panettiere praticamente), entra in zona critica il processo con il numero più basso. L’algoritmo non è però in grado di assicurare che i processi siano univocamente identificabili (per la natura stessa del problema che stiamo cercando di risolvere) per questo in caso di processi con lo stesso numero usiamo i PID per disambiguare (12-3 16-3 14-3 passa prima quello con il PID più basso).

**Variabili condivise:**

* boolean scelta[n];
* int numero[n];

**Codifica:**

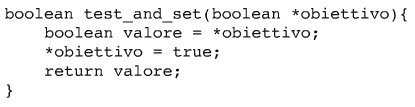
Tutti i processi aspettano di aver preso il numero per poterli confrontare.

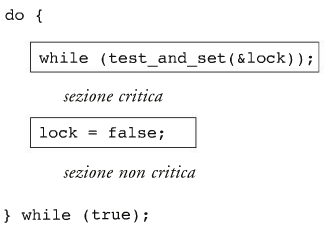
**Dimostrazione:**

* **Mutua esclusione:** Se sono in zona critica vuol dire che sono il processo con il numero **((numero[i], i) <(numero[k], k))** più basso in coda quindi nessuno può disturbarmi finché non dico che ho finito **(numero[i]=0;)**. Se sono un processo in attesa finché non divento il processo con il numero più basso **((numero[i], i) <(numero[k], k))** e quello servito non libera il numero **(while(scelta[j]) ;)** non posso muovermi.
* **Progresso e Attesa limitata:** I processi entrano all’interno della coda con una politica di tipo FCFS (First Come First Served).

**5.4 Hardware per la sincronizzazione: tutto prima di qui vale solo per sistemi mono processore.** Altre soluzioni al problema della sezione critica si basano sull’utilizzo del **lock** ovvero sulla protezione delle sezioni critiche con l’uso di lucchetti. Oltre all’utilizzo dei lock i moderni sistemi operativi mettono a disposizione delle funzioni che permettono di modificare parole chiave contenute nella memoria, queste operazioni sono gestite a livello molto basso dal kernel cosa che ne garantisce l’atomicità (una volta iniziata non può essere interrotta).

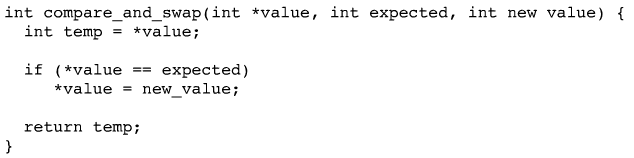
**Istruzione test\_and\_set:** la caratteristica fondamentale di questa istruzione è che viene eseguita

**Mutua esclusione:** per realizzare la mutua esclusione con test\_and\_set si implementa il seguente codice:

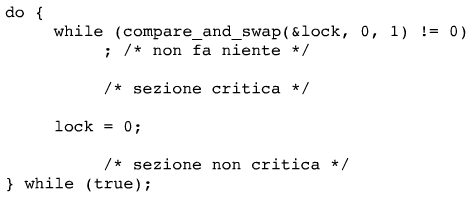
Mentre il lock è falsa entro in sezione critica, informando gli altri che ora è occupata (lock = true ) dal set\_and\_set, quando ho finito la metto su false informando gli altri che ho finito.

**Istruzione compare\_and\_swap:** al contrario della test\_and\_set la compare\_and\_swap utilizza tre parametri:

* **Value:** Valore attuale della cella di memoria che si vuole modificare.
* **Expected:** Valore che ci si aspetta di trovare nella cella di memoria.
* **New\_value:** Nuovo valore da settare nella cella di memoria.



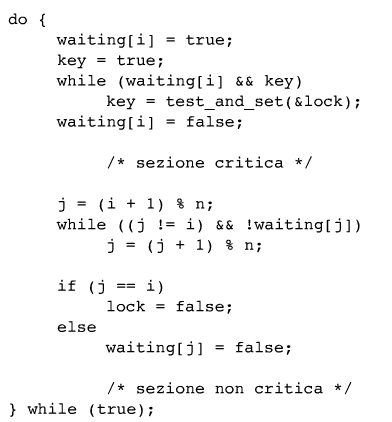
**Mutua esclusione:** Per realizzare la mutua esclusione con il compare\_and\_swap si implementa il codice sotto, mentre il lock è a 0 entro in sezione critica settandolo a 1 con la compare dicendogli che occupo il processore quando ho finito lo rimetto ad 0 liberando il processore.

Questi due algoritmi soddisfano la mutua esclusione ma non l’attesa limitata. Non sono pertanto idonei a risolvere il problema della zona critica.

**Mutua esclusione con attesa limitata con test\_and\_set:** per questo algoritmo le variabili condivise sono due:

* **boolean waiting[n];**
* **boolean lock;**

Entrambe inizializzate al valore **false.**

**Codice:**

**Mutua esclusione:** un processo **P** può andare in sezione critica solo se **waiting[i]** o **key** sono uguali a 0, per questo motivo la **key** viene mossa da un **test\_and\_set** che lo fa diventare false solo nel caso in cui il lock sia falso.

**Progresso:** una volta terminata la sezione critica sia il lock che il waiting[j] vengono impostati su false.

**Attesa limitata:** una volta conclusa la sua zona critica il processo scandaglia il vettore waiting alla ricerca di un processo in attesa di entrare in sezione critica (**waiting[j] = true**), una volta che l’ha trovato lo fa andare in zona critica con **waiting[j] = false**, se non lo trova e ritorna a sé stesso ri-acquisisce il lock ed entra in zona critica.

**Capitolo 5 Sincronizzazione dei processi (già usciti):**

**5.6 Semafori:** Un semaforo S è una variabile intera a cui si può accedere tramite due operazioni atomiche predefinite wait () e signal ()

**5.6.1 Uso dei semafori:** Esistono principalmente due tipi di semafori:

* **Semafori contatori:** il cui valore è un numero intero
* **Semafori binari:** il cui valore è un numero binario

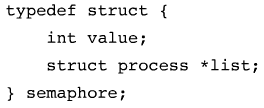
I semafori contatori sono usati principalmente per gestire l’accesso ad una determinata risorsa presente in un numero limitato di esemplari:

1. Il semaforo è inizializzato al numero di risorse presenti.
2. Quando un processo ha necessità di una di queste risorse la prende con wait decrementando il semaforo, quando ha finito la rilascia con signal.
3. Quanto il semaforo è uguale a 0 le risorse sono finite.

In modo analogo questi tipi di semafori possono essere usati per risolvere problemi di sincronizzazione:

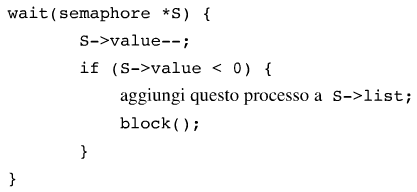
* Il processo uno esegue S1.
* Rilascia la risorsa gestita dal semaforo synch.
* Il processo due blocca la risorsa di synch.
* Il processo due esegue S2.

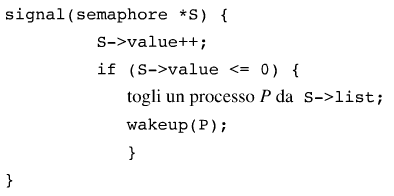
**5.6.2 Implementazione dei semafori:** Il problema principale del Lock mutex era lo stato di attesa continua (cicli infiniti fino alla liberazione del lock) i semafori risolvono questo problema bloccando il processo e mettendolo in una coda di attesa al semaforo. Per fare ciò possiamo definire il semaforo nel seguente modo:



Dove **value** indica il valore del semaforo e **list** una lista di puntatori ai PCB dei processi in attesa che vengono risvegliati dall’operazione signal.

Le nuove signal e wait saranno così definite:





L’operazione **block** presente nella **wait** blocca il processo mettendolo in attesa, mentre l’operazione **wakeup** lo risveglia. Inoltre, è importante notare che così facendo il valore del semaforo può essere anche negativo in quanto i processi presenti nella lista possono essere giustamente di più delle risorse disponibili. (-15 sei in posizione 15 della coda).

Sono sempre i processi a chiamare signal e wait, dove signal togli il processo dalla lista e gli dà la risorsa, e wait lo mette nella lista e lo blocca in attesa del rilascio.

**5.6.3 Stallo e attesa indefinita:** La realizzazione di un semaforo con coda di attesa può portare ad un fenomeno definito **stallo** che avviene quando due processi attendono il rilascio di due risorse che servono ad entrambi, esempio:

S e Q sono due semafori di valore 1.

P0 prende il controllo di S e P1 di Q

P0 per continuare ha bisogno di Q (deve cioè aspettare che P1 dia signal), analogamente P1 ha bisogno di S (deve cioè aspettare che P0 dia signal).

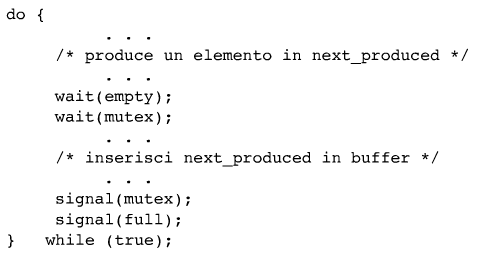
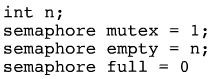
**Siamo fermi…**

Un'altra questione collegata al concetto di stallo è **l’attesa indefiniti o starvation**, e nasce dal fatto che non è possibile definire il tempo di attesa in una coda ad ‘un semaforo creata con una politica LIFO (primo che entra primo ad uscire).

**5.6.4 Inversione di priorità:** Un problema di questo tipo di soluzione sta nella gestione dei processi a priorità più alta, essi infatti nel caso in cui debbano eseguire un’operazione dovranno comunque attendere il rilascio della risorsa dal processo che la sta utilizzando, visto che quest’ultima è bloccata.

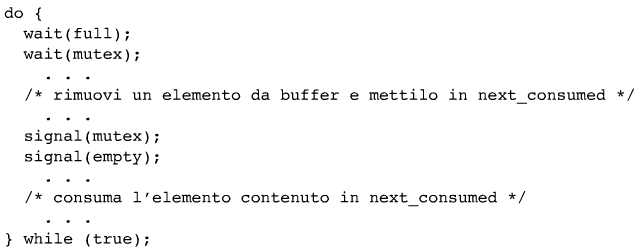
**5.7 Problemi tipici di sincronizzazione:** Di seguito sono riportati alcuni di problemi tipici che riguardano la gestione della concorrenza di processi.

**5.7.1 Produttore/Consumatore con memoria limitata:** Questo problema fa riferimento a una situazione in cui abbiamo due processi (produttore e consumatore) che condividono un buffer di memoria condivisa, dove, il processo produttore inserisce dati e il processo consumatore li preleva. All’interno di questo problema **produttore** e **consumatore** condividono le seguenti informazioni:



**Codice produttore:**

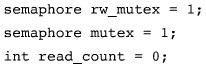
**Codice consumatore:**



È importante notare com il codice del produttore è perfettamente simmetrico rispetto a quello del consumatore.

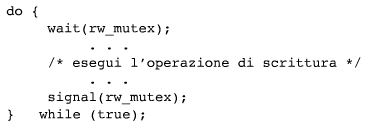
**5.7.2 Problema dei lettori-scrittori:** Questo problema si applica principalmente alla gestione di basi di dati, all’interno delle quali può capitare che più processi vogliano accedere contemporaneamente ai dati, tali processi (nell’ottica di questo problema) si dividono in due categorie:

* **Lettori:** Processi interessati solo a operazioni di lettura (nessun problema generato dalla concorrenza).
* **Scrittori:** Processi interessati a operaizoni di aggiornamento (quindi di lettura e scrittura, cosa che può portare a problemi).

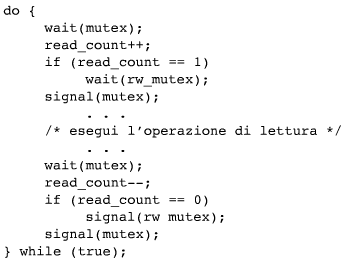
**Variabili:** 

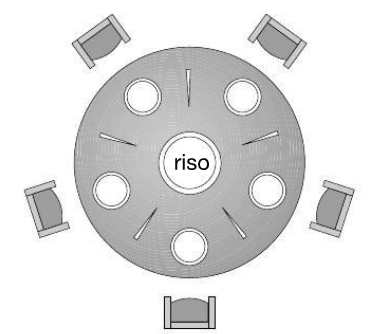
* **rw\_mutex:** Semaforo comune a tutti i processi.
* **mutex:** Utilizzato per assicurare la mutua esclusione quando si aggiorna read\_count.
* **read\_count:** Numero di processi che stanno leggendo l’insiem dei dati.

**Processo scrittore:**

Aspetta che il semaforo sia libero, appena lo è eseguo le operazioni di scrittura e poi libero il semaforo.

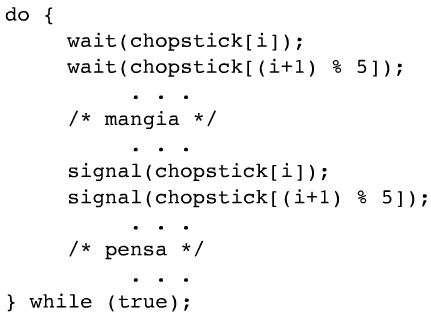
**Processo lettore:**

Aggiorno read\_count (in modo esclusico con la wait) se sono solo aspetto rw mutex, altrimenti leggo, decremento read\_count (sempre in modo esclusivo). Se ero solo libero anche rw mutex.

**5.7.3 Problema dei cinque filosofi (dining philosophers):** Si considerino cinque filosofi che trascorrono la loro esistenza pensando e mangiando. I filosofi condividono un tavolo rotondo circondato da cinque sedie, una per ciascun filosofo. Al centro del tavolo si trova una zuppiera colma di riso, e la tavola è apparecchiata con cinque bacchette (in inglese chopstick). Quando un filosofo pensa, non interagisce con i colleghi; quando gli viene fame, tenta di prendere le bacchette più vicine: quelle che si trovano tra lui e i commensali alla sua destra e alla sua sinistra. Un filosofo può prendere una bacchetta alla volta e non può prendere una bacchetta che si trova già nelle mani di un suo vicino. Quando un filosofo affamato tiene in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette. Terminato il pasto, le posa e riprende a pensare.

**Soluzione:** consiste nell’identificare ogni bacchetta con un semaforo. L’unico dato condiviso è un array di semafori:



**Codice:** 

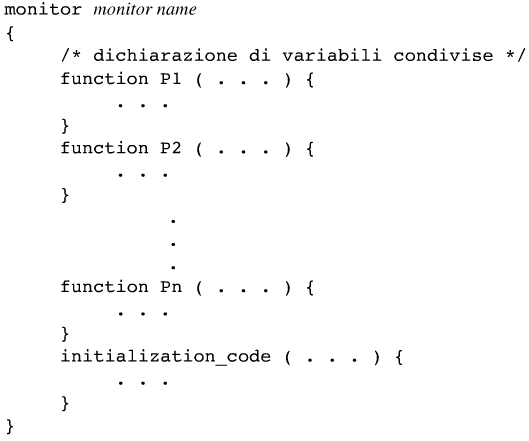
Il filosofo blocca due bacchette, mangia e poi le sblocca.

Questa soluzione ha però un problema: Nel caso in cui tutti i filosofi vogliono mangiare allo stesso momento, andiamo in una situazione di stallo (tutti cercano di prendere due bacchette **starvation**), situazione che può essere evitata con diversi espedienti:

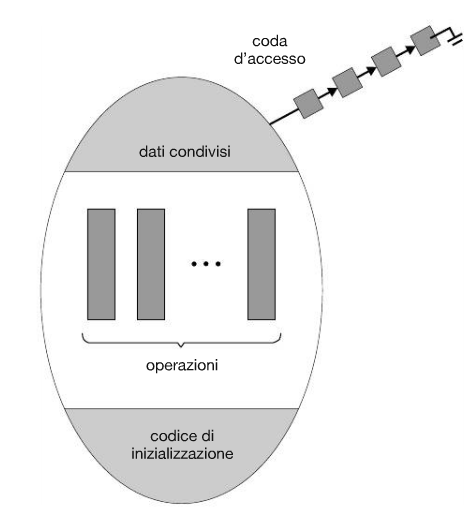
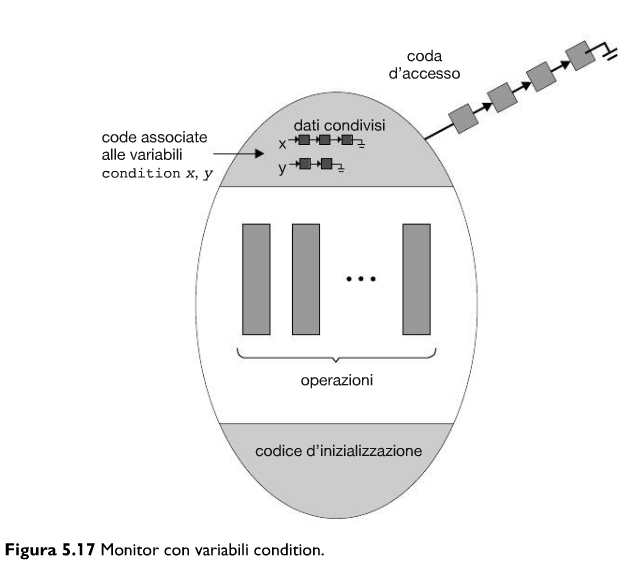
* Solo 4 filosofi possono stare contemporaneamente a tavola.
* Un filosofo può prendere le sue bacchette solo se entrambe disponibili (operazione da eseguire in sezione critica).
* Si adotta una soluzione asimmetrica: un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra.

Una soluzione perfetta per questo problema è realizzabile attraverso i **monitor** è deve garantire che nessu filosofo vada in attesa indefinita (muoia di fame **starvation**).

**5.8 Monitor:** Anche se i semafori costituiscono un meccanismo molto efficace per la sincronizzazione dei processi, possono generare errori molto difficili da individuare. In quanto si verificano sono in presenza di determinate sequenze.

**5.8.1** **Uso del costrutto monitor:** Un monitor è un **tipo di dato astratto (ADT)** che incapsula i dati mettendo a disposizione delle funzioni contraddistinte dalla mutua esclusione oltre che le dichiarazioni alle variabili. Tale costrutto assicura che all’interno di esso un solo processo rimanga attivo contemporaneamente. Un altro costrutto presente all’interno di un monitor è il costrutto **condition** utilizzato per implementare particolari schemi di sincronizzazione definendo variabili di tipo condition. Le operazioni eseguibili su queste variabili sono due:

**condition x;**

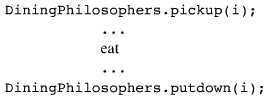
* **x.wait();** sospende il processo che la invoca finché un altro processo non lo risveglia.
* **x.signal();** risveglia un processo sospeso a caso, se non ci sono processi sospesi la signal non ha alcun’effetto.

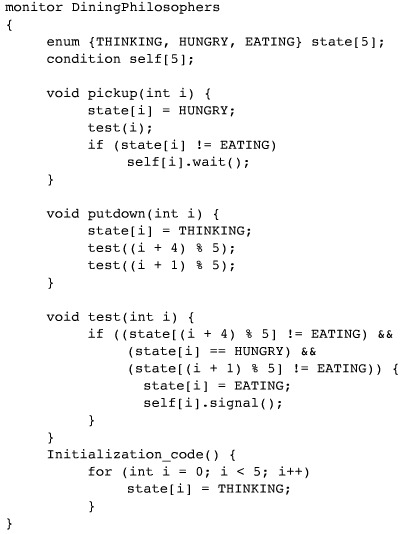
**5.8.1: Problema dei cinque filosofi con i monitor:** la soluzione si basa sulla seguente struttura dati:

**enum {THINKING, HUNGRY, EATING} state[5];**

un filosofo può andare in stato di eating (state[i]=EATING) solo nel caso in cuii suoi due vicini non stanno mangiando. Occorre inoltre dichiarare all’interno del dato le seguenti strutture:

**condition self[5];**

che consente al filosofo di ritardare sé stesso anche quando a fame. La distribuzione delle bacchette viene effettuata dal monitor **DiningPhilosophers**, che mette a disposizione delle funzioni che permettono ai filosofi di prendere e posare le bacchette. Un filosofo i per mangiare e tornare a pensare deve quindi eseguire le seguenti operazioni:

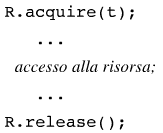
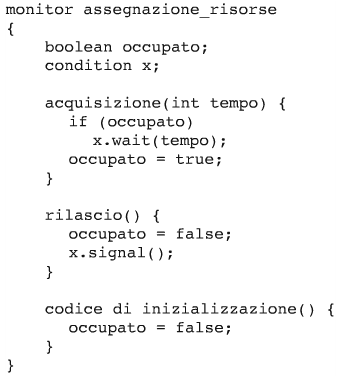
**Codice del monitor:**

**5.8.4: Ripresa dei processi all’interno di un monitor:** quando più processi si trovano in uno stato di sospeso e un altro processo invoca una **x.signal()** diventa necessario stabilire quale processo tra quelli sospesi debba essere risvegliato. Alcune soluzioni:

* Usare un’ordinamento **FCFS** (First Come First Served) secondo il quale il processo che aspetta da più tempo sarà il prossimo a essere eseguito.

Usare il costrutto di attesa condizionale **x.wait(c);** dove **c** chiamato **numero di priorità** (calcolato tramite un espressione creata ad oc) diventa il nome del processo, che verrà utilizzato dalla signal per risvegliare il processo con numero di priorità più basso.

**Esempio**

tale monitor ha il compito di assegnare una particolare risorsa a dei processi in competizione. Quando un processe desidera ottenere tale risorsa deve esegurie le seguenti funzioni:

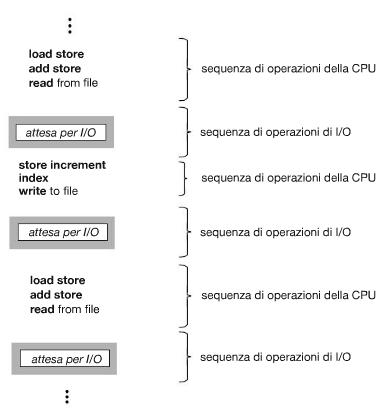
Sfortunatamente il concetto di monitor non può garantire che la precedente sequenza venga rispettata. In particoalre potrebbero verificarsi le seguenti situazioni:

* Un processo può accedere alla risora senza permesso.
* Un processo potrebbe non rilasciare una risorsa.
* Un processo può tentare di rilasciare una risorsa che non ha mai chiesto.
* Un processo può richiedere due volte la stessa risorsa, senza rilasciarla prima della seconda richiesta

**Capitolo 6 Scheduling della CPU:**

**6.1 Concetti fondamentali:** Come abbiamo visto in precedenza all’interno di un sistema monoprocessore può essere eseguito un processo alla volta, questo può portare a situazioni di inattività della CPU (processo in attesa di un evento) per questo motivo quando un processo va in attesa di qualcosa la CPU effettua un cambio di contesto per iniziare a elaborare un altro processo, questa caratteristica prende il nome di **multiprogrammazione.**

**6.1.1 Ciclicità delle fasi d’elaborazione di I/O:** L’esecuzione di un processo all’interno di una CPU consiste in un ciclo d’elaborazione (svolto dalla CPU) e di attesa delle operazioni di I/O.

L’esecuzione di un processo all’interno della CPU inizia con una sequenza di operazioni svolte da quest’ultima dette **CPU burst**,

Successivamente inizia la prima fase di elaborazione.

Seguita da un prima fase di attesa delle operazioni di I/O.

Il seguente ciclo di operazioni va avanti fino all’istruzione di terminazione del processo.

La durata delle operazioni della CPU prende il nome di **burst della CPU**, tale durata varia molto in base alla tipologia di processi presenti nella CPU (I/O bound seguenze corte e frequenti, CPU bound seguenze lunge e meno frequenti.

**6.1.2 Scheduler della CPU:** Ogni volta che la CPU passa in uno stato di inattività, lo **Scheduler della CPU** sceglie dalla ready queue il prossimo processo che impegnera la CPU. È importante notare che la ready queue non è necessariamente una coda di tipo **FIFO,** l’organizzazione di tale coda infatti varia in base al tipo di scheduler utilizzato.

**6.1.3 Scheduling con prelazione:** Le decisioni che riguardano lo scheduler della CPU vengono prese nei seguenti 4 casi:

1. Un processo passa dallo stato di esecuzione allo stato di attesa (per esempio wait()).
2. Un processo passa dallo stato di esecuzione allo stato di pronto (segnale di interruzione).
3. Un processo passa dallo stato di attesa allo stato di pronto (completamento delle operazioni di I/O).
4. Un processo termina.

Quando lo scheduler interviene nelle condizioni 1 e 4, si dice che lo schema di scheduling è **senza prelazione (nonpreemptive)** o **coperativo**, altrimenti viene detto **con prelazione (preemptive).** Nel primo caso quando si assegna un processo alla CPU quest’ultimo rimane in esecuzione finche non viene rilasciato normalmente (non può essere interrotto forzatamente come in presenza di prelazione), questa caratteristica può portare a situazioni di **race condition.**

**6.1.4 Dispatcher:** Si tratta di un modulo dello scheduler che si occupa di passare effettivamente il controllo della CPU al processo, esso svolge le seguenti operazioni:

* Cambio di contesto.
* Passaggio alla modalità utente.
* Salta alla giusta posizione del programma utente per riavviare l’esecuzione.

Poiche si attiva ad ogni cambio di contesto il dispatcher deve essere il più veloce possibile. Il tempo richiesto dal dispatcher per svolgere le sue funzioni viene detto **latenza di dispatch.**

**6.2 Criteri di scheduling:** Esistono diversi criteri utilizzati per confrontare gli algoritmi di scheduling, ecco i principali:

* **Utilizzo della CPU:** La CPU deve essere più attiva possibile, in particolare il carico deve oscillare tra il 40 e il 90 percento.
* **Produttività:** La CPU è attiva quando si svolge un lavoro, tale lavoro viene misurato tramite la **produttività (throughput)** della CPU ovvero il numero di processi elaborati in una determinata unità di tempo.
* **Tempo di completamento:** Dal punto di vista dei processi il criterio più importante è il **tempo di completamento (turnaround time)** ovvero il tempo che intercorre da quando il processo è caricato in memoria a quando è terminato.
* **Tempo di attesa:** è il tempo che un processo attende in coda prima di essere scelto per utilizzare la CPU.
* **Tempo di risposta:** è il tempo che intercorre dall’inzio dell elaborazione del processo e il suo primo output (un processo può generare output anche durante la sue esecuzione).

Un algoritmo di scheduling ideale dovrebbe essere caratterizzato da: un utlizzo della CPU e prodittività molto alta e tempi di risposta, attesa e completamento molto bassi.

**6.3 Algoritmi di scheduling**

**6.3.1 Scheduling in ordine di arrivo:** Questo tipo di scheduler estremamente semplice applica una politica di tipo **FCFS (first come first served)** assegnando, tramite una cosa con politica **FIFO**, la CPU al primo processo che arriva. Un aspetto negatico di questo scheduler è che il tempo medio d’attesa è molto lungo.

**Esempio:**

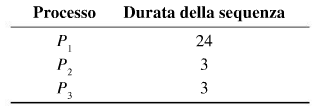
Prendendo in esempio questri tre processi possiamo vedere che utlizzando questo algoritmo il tempo medio di attesa è di (24+27+0/3=17).

Diagramma di Gant (con FCFS):

****

Diagramma di Gante (senza FCFS):

****

In questo caso il tempo medio di attesa e di (0+3+6/3=3) quindi molto più basso.

Un altro problema di questo algoritmo è il cosidetto **effetto convoglio,** tutti i processi attendono che un lungo processo liberi la CPU, causando una riduzione dell’utilizzo della CPU e dei vari dipositivi.

**6.3.2 Scheduling shortest-job-first:** questo algoritmo associa ad ogni processo la lunghezza delle successiva seguenza di operazioni dela CPU. Assegnando alla CPU il processo con la più brece sequenza di operazioni, nel caso di processi con sequenze di lunghezza uguale si applica un apolitica FCFS.

**Esempio:**

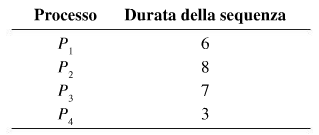
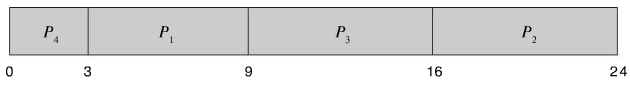
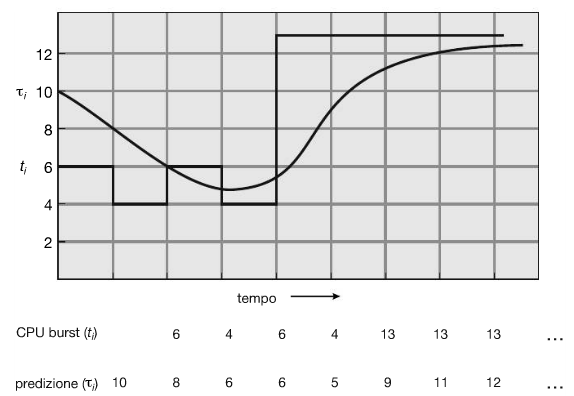
In questo caso il tempo medio di attesa è di (3+9+16/3=7) mentre con FCFS sarebbe stato di circa 10 millisecondi.

Diagramma di Gant (con SJF)

****

Una difficoltà nell’implementazione di questo algoritmo sta conoscere la durata della successiva richiesa della CPU, per questo motivo questo alrgoritmo viene utilizzato principalemnte in scheduler a lungo termine. Nonostante tutto una possibile soluzione consisterebbe nel cercare di approssimare la durata della sequenza tramite il calcolo di una **media esponenziale.**



Un'altra caratteristiche di questo algoritmo è che puo essere sia con **prelazione**  che **senza**, in particolare potrebbe accadere che arrivi all’interno della cosa dei processi pronti un processo che ha una sequenza più corta di quella che resta al processo attualmente in esecuzione.

**Esempio:**

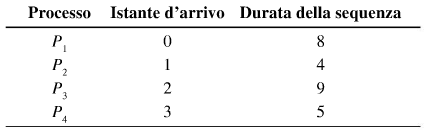
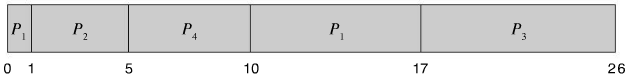
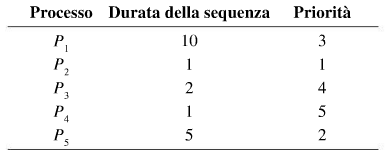
Si noti come il processo P1 viene avviato e poi sospeso per elaborare i processi P2 e P4.

Diagramma di Gant (SJF con prelazione)

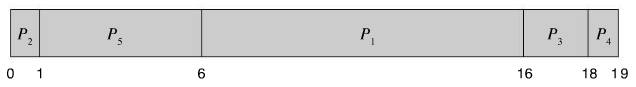
****

**6.3.3 Scheduling con priorità:** Questo algoritmo si basa sull’assegnazione di un priorità ad’ogni processo, i processi con priorità più alta verranno eseguiti per primi, tale priorità viene assegnata solitamente tramite un intervalli di numeri interi. In caso i processi con priotià uguale si applica una politica **FCFS.**

**Esempio:** 

I processi con priorità più alta (numero basso) sono i primi ad ottenere la CPU. Il tempo di attesa medio è di 8.2 millisecondi.

Diagramma di Gant:



Le priorità assegnate ai vari processi si possono definire sia:

* **Internamente:** quando la priorità viene calcolata in base alle caratteristiche del processo
* **Esternamente:** quando viene decisa dagli utenti in base a fattori esterni (quantità di fondi pagati da chi ha generato il processo per esempio).

Un problema importante relativo a questo algoritmo di scheduling è **l’attesa indefinita (starvation),** potrebbe infatti capitare che un processo a bassa priorità non venga mai eseguito. Una possibile soluzione a questo problema e costituista dall’**ivecchiamento (aging),** si tratta di una tecnica he aumenta la priorità di un processo con il passare del tempo in modo da assicurarne l’esecuzione.

**6.3.4 Scheduling circolare:** Questo algoritmo di scheduling circolare o (round robin, RR) è stato progettato appositamente per sistemi **time-sharing** esso utilizza una politica **FCFS** su una coda con politica **FIFO,** concettualmente l’algoritmo è molto simile a quello con lo scheduler di ordine di arrivo, aggiunge però la prelazione e tratta la redy queue come una coda circolare dando la possibilità a i processi di ottenere la CPU solo per determinati lassi di tempo detti **quati di tempo** o **porzioni ti tempo (slice time)** (tra i 10 e i 100 millisecondi).

**Esempio:**

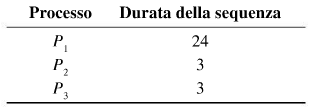
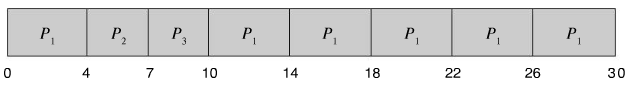
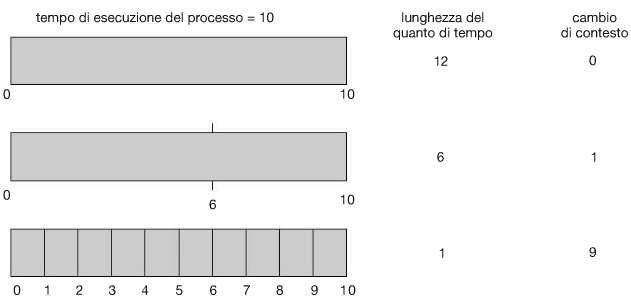
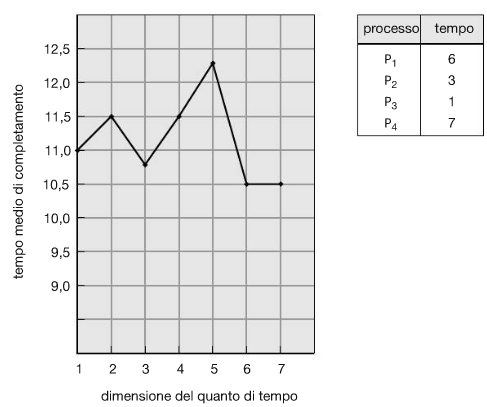
In questo caso il tempo che lo scheduler assegna ai singoli processi è di 4 millisecondi prima di dover rilasciare la CPU. Tempo medio di attesa 5.66 millisecondi

Diagramma di Gant:

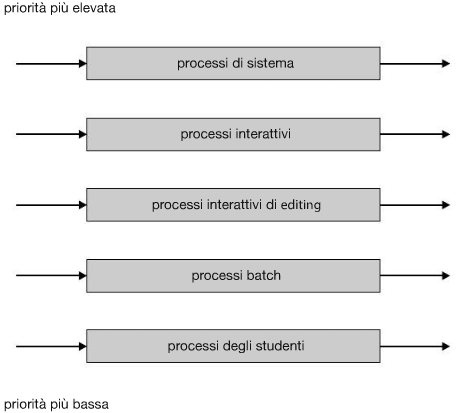


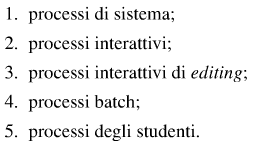
Un aspetto molto importante in qeusto algoritmo è la lunghezza del quanto di tempo, infatti se esso è molto lungo l’algoritmo si comporta come un normlae FCFS se è troppo corto effetuiamo cambi di contesto inutili.

é importante notare che la grandezza del quando tempo impatta anche sul **tempo di completamento**, in quanto in caso di quanti piccoli al tempo di completamento andranno aggiunti anche i tempi dei cambi di contesto. Empiricamente si può stabilire che l’80% delle sequenze di tempo devono essere inferiori all’quanto di tempo.

****

**6.3.5 Scheduling a code mutlivello:** Un’altra soluzione al problema dello scheduling è quella di dividere i processi in code distinte, ognuna con un proprio scheduler personale, oltre a uno scheduler generale con priorità fissa, che favorisce i processi appartenenti ad’una determinata coda. Questo algoritmo prevede la prelazione.

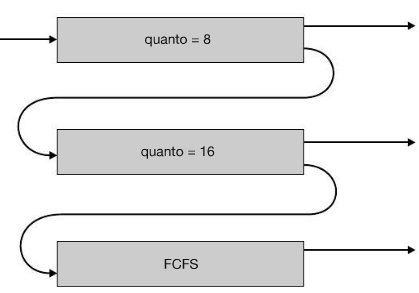
**Esempio:**

****

Una distinzione comune che si fa tra i vari tipo di processi è la seguente:

* **Processi foregroung (primo piano) o interattivi**
* **Provessi backgrounf (sottofonto) o batch**

Un'altra possibile soluzione complementare all’assegnazione delle priorità consiste nell’assegnare ad ogni coda un tempo di CPU per l’esecuzione.

**6.3.6 Scheduling a code multilivello con retroazione:** questo tipo di scheduler è molto simile al precedente, infatti anchesso divide i processi in varie code di scheduler ma lo fa in maniera meno rigida permettendo ai vari processi di spostarsi tra una cosa e l’altro. Per esempio processi di tipo CPU boud vengono messi in code a priorità più bassa mentre processi di tipo I/O bound vengono posizionati in code a priorità più altre. 

**Esempio:**  un’altra possibile soluzione sta nel dividere i processi in base alla lunghezza delle loro seguenze( immagine precedente).

Principali caratteristiche di uno **scheduler a code multilivello con retroazione**:

* Numero di code.
* Algoritmo di scheduling per ciascuna coda.
* Metodo usato per determinare quando far salire un processo di priorità.
* Metodo usato per determinare quando far scendere un processo di priorità.
* Metodo usato per determinare in quale coda un processo dovrà risiedere.

Questo algoritmo rappresenta un ottima soluzione generale è però estremamente complesso da implementare.

**6.7.1 Scheduling in linux:**

Lo scheduler di Linux ricorre ad un algoritmo di scheduling con prelazione basato sulle priorità. Ci sono due gamme di priorità separate:

* Un intervallo **real-time** che va da 0 a 99
* Un intervallo **nice** compreso tra 100 e 140.

I valori di questi due range sono mappati in un valore di priorità globale dove numeri bassi implicano priorità alta.

Il kernel mantiene una lista di tutti i task in una **runqueue** che contiene due array di priorità:

* **Attivo** (active array): contiene tutti i task che hanno ancora tempo da sfruttare
* **Scaduto** (expired array): contiene i task scaduti.

**6.8 Valutazione degli algoritmi:** La valutazione della scelta di un algoritmodi scheduling si effettua attraverso dei criteri dettati dallo scopo e dalle caratteristiche del sistema, un esempio di citeri di scelta può essere in seguente:

* Rendere massimo l’utilizzo della CPU mantenedo un tempo di risposta di massimo 1 secondo.
* Rendere massima la produttività in modo che il tempo di completamento sia direttamente proporzionalre al tempo di esecuzione totale.

**6.8.1 Modellazione deterministica:** Fra i vari metodi di valutazione utilizzati sono particolarmente importanti quelli che fanno parte della classe della **valutazione analitica,** tale valutazione si occupa tramite delle formule di fornire un numero che indica le prestazioni del sistema. Un esempio di valutazione analitica è la **modellazione deterministica** che si occupa tramite una serie di valori di carico predeterminati di valutare le prestazioni con ogni algoritmo.

**Esempio:**

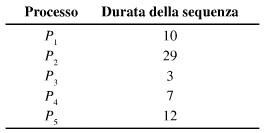
con questi processi andiamo ad applicare i tre principali algoritmi di scheduling, vedendo come si comportano:

Diagramma di Gant(con algoritmo FCFS tm=39):



Diagramma di Gant (con algoritmo FJS tm=13):

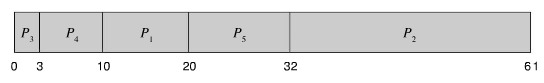
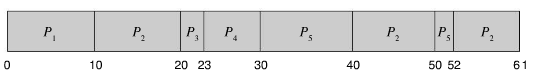
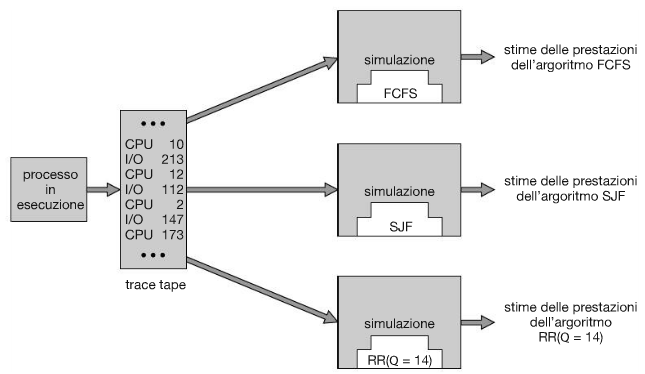
****

Diagramma di Gant(con algoritmo RR tm=23):

****

**6.8.2 Reti di code:** in alcuni sistemi il tipo di processi eseguiti cambia di giorno in giorno, quindi non esiste un insieme statico di processi da usare nella modellazione deterministica. Si può risolvere tale problema studiando la distribuzione delle seguenze di operazioni della CPU e dei dispositivi di I/O in modo ,se sono noti o stimabili l’andamento degli arrivi e dei servizi, da determinare in modo approssimativo i carichi di lavoro. All’interno di questo studio la CPU viene vista come un entità servente con la propria coda di processi pronti, stesso discorso vale per i dispositivi di I/O, per tale motivo questo tipo di studio viene definito **analisi delle reti di code** ed ha come risultato un’approssimazione non sempre esatta del sistema reale.

**6.8.3 Simulazioni:** un’altra soluzione che si può applicare per avere dei dati più precisi è la **simulazione**, dove tramite un modello del sistema di calcolo vengono simulati i processi (generati randomicamente). Per funzioanre il simulatore dispone di una variabile definita **clock** all’umentare di tale variabile il simulatore modifica lo stato del sistema, tenendo traccia dei vari processi all’interno della **trace tape**. Un problema dei simulatori è il loro costo, sia livello di risorse di camclo che di progettazione, oltre a non essere sempre affidabili.

**6.8.4 Realizzazione:** l’unico modo assolutamente sicuro di valutare un algoritmo di scheduling consiste nel codificarlo, inserirlo nel S.O. ed osservarne il comportamento nelle reali condizioni di funzionamento. Si tratta però ovviamente di una soluzione molto onerosa e per certi versi rischiosa (agli utenti non piace).

**Capitolo 7 Stallo dei processi:**

**7.1 Modello del sistema:** Ogni sistema è composto da un numero finito di risorse che vengono distribuite tra i vari processi. Generalmente queste risorse sono suddivise in **classi** differenti ciascuna formata da un certo numero di instanze. Esempio: la classe stampante ha N, una per ognuna delle N stampanti. Nelle ordinarie condizioni di funzionamento un processo può servirsi di una risorsa solo se rispetta la seguente sequenza di passi:

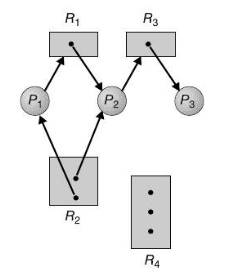
1. **Richiesta:** Il processo richiede la risorsa, se è disponibile se ne appropria altrimenti attende che si libera.
2. **Uso:** Il processo può operare sulla risorsa.
3. **Rilascio:** il processo una volta terminata l’elaborazione rilascia la risorsa.

Nonostante questo vincolo all’iterno dei sistemi operativi ci si trova a volte in **situazioni di stallo,** ovvero quando più processi attendo il liberarsi di una risorsa in modo infinito. Esempio: prendiamo tre processi che possegono ognuno un lettore CD se ognuno di questi tre processi avesse bisogno di un secondo lettore per computare ecco che ci troviamo in una situazione di stallo.

**7.2 Caratterizzazione delle situazioni di stallo:**

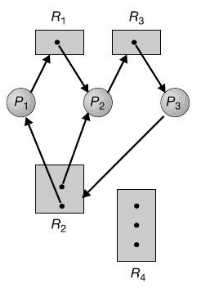
**7.2.1 Condizioni necessarie:** all’interno di un sistema operativo si può avere una situazione di stallo solo se si verificano **contemporaneamente** le sequenti 4 condizioni:

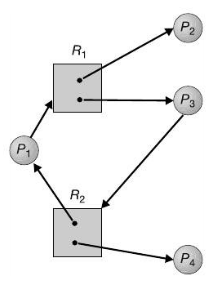
* **Mutua esclusione:** Almeno una risorsa deve essere non codivisibile, ovvero deve essere utilizzabile da un solo processo alla volta.
* **Possesso e attesa:** Un processo deve essere in possesso di una risorsa e attendere di acquisire risorsa già in possesso da altri processi.
* **Assenza di prelazione:** Un processo non può essere obbligato a rilasciare una risorsa.
* **Attesa circolare:** Deve esistere un insieme di processi che si attendono in loop.

**7.2.2 Grafo di assegnazione delle risorse:** Le situazioni di stallo possono essere rappresentate tramite una rappresentazione detta **grafo di assegnazione delle risorse.** All’interno di questa rappresentazione vediamo due tipi di archi:

* **Arco di richiesta:** quando un processo chiede una risorsa.
* **Arco di assegnazione:** quando ad’un processo viene assegnata una risorsa.

**Esempi**

Grafo con stallo a sinistra, e grafo con ciclo senza stallo a destra.

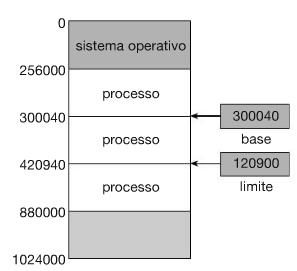


**7.3 Metodi per la gestione delle situazioni di stallo:** essenzialmente il problema dello stallo può essere affrontato in tre modi diversi:

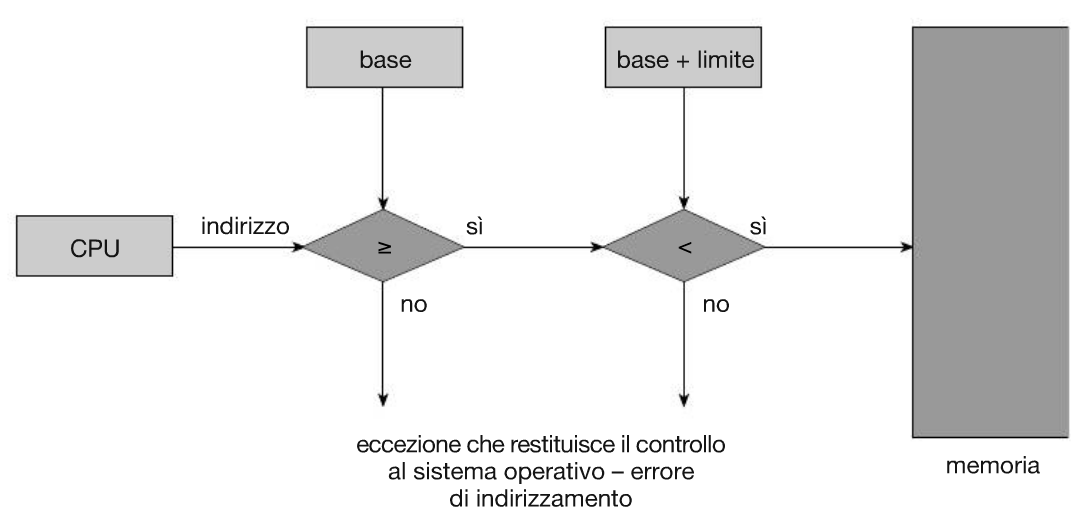
* **Assicurare che il sistema non entri mai in uno stato di deadlock.**
  + **Prevenire i deadlock:** evitare che contemporaneamente si verifichino mutua esclusione, possesso e attesa, impossibiltà di prelazione e attesa circolare  basso utilizzo risorse e throughput ridotto.
  + **Evitare i deadlock:** evitare gli stati del sistema a partire dai quali si può evolvere verso un deadlock.
* **Permettere al sistema di entrare in uno stato di deadlock**, quindi ripristinare il sistema.
  + **Determinare** la presenza di un deadlock.
  + **Ripristinare** il sistema da un deadlock.
* **Ignorare il problema** e “fingere” che i deadlock non avvengano mai nel sistema; impiegato dalla maggior parte dei S.O., incluso UNIX.

**Capitolo 8 Memoria centrale:**

**8.1 Introduzione:** la memoria non è altro che un grande vettore di byte dove il processore preleva e deposita dati in attesa di essere elaborati. Un tipico ciclo di esecuzioni è organizzato in 3 fasi, prelievo dei dati dalla memoria, elaborazione, scrittura dei risultati in memoria.

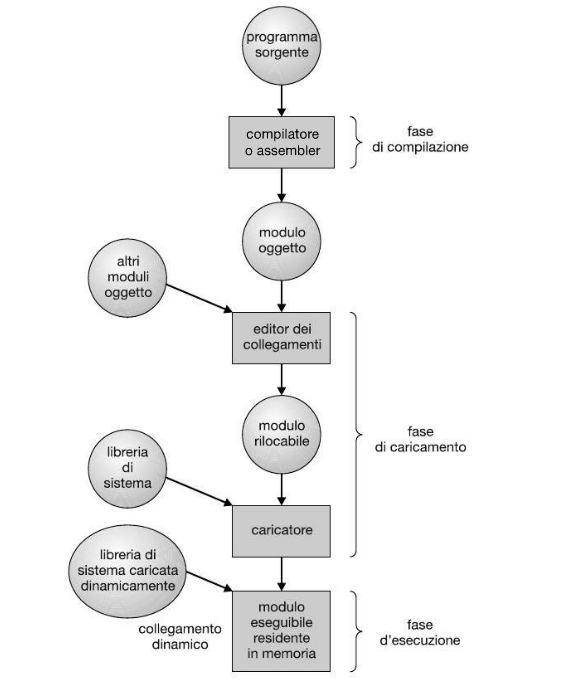
**8.1.1 Hardware di base:** La memoria centrale e i registi incorporati nel processore sono le uniche aree di memoria a cui il processore può accedere direttamente, non esistono infatti istruzioni macchina che accettano indirizzi dei dischi come paramentri. Per quanto riguarda i registi della CPU essi sono di solito accessibili più volte all’interno dello stesso **ciclo di clock**, cosa che non si può però affermare con la memoria centrale (estremamente più lenta dei registri, visto che si deve attraversare un bus esterno), questa differenza di velocità può portare a situazioni di **stallo**, per mancanza di dati. Per risolvere questo problema è stata introdotta la **cache,** un particolare tipo di memoria molto veloce che si interporre tra i registi e la memoria di massa. Oltre alla questione della velocità bisogna anche suddividere la memoria in modo che ogni processo abbia un proprio spazio di indirzzamento, per fare ciò si utilizzano due registi:

* **Registo base:** indirizzo di memoria più basso.
* **Registo limite:** indirizzo di memoria più alto.

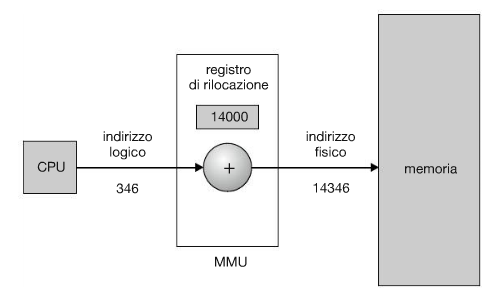
Per mantenere tali spazi la CPU confronta ogni indirizzo generato in modalità utente che i due registri controllando che un processo non tenti di accedere ad aree di memoria che non gli sono concesse, in caso di tentativi di accesso errati viene restituita un **eccezione (trap)** che restituisce il controllo al sistema operativo.

Solo il sistema operativo può caricare i registri base e limite, grazie una speciale istruzione privileggiata che può essere eseguita solo in modalità kernel.

**8.1.2 Associazione degli indirizzi:** In genere un programma quando non è in esecuzione risiede all’interno del disco sottoforma di file binario, all’atto dell’esecuzione il programma viene caricato in memoria e inserito nel contesto di un processo. La coda che contiene tutti i processi pronti ad’essere caricati in memoria prende il nome di **input queue.** È importante notare che la maggior parte dei sistemi consente ai processi di risiedere in qualsiasi parte della memoria (quindi non è detto che il primo indirizzo di memoria del processo sia 00000. Nella maggior parte dei casi un programma per essere eseguito deve passare attaverso una serie di passi (figura affianco) inoltre è importante notare che generalmente gli indirizzi del programma sorgente sono simbolici ( per esempio la variabile **count**) in questi casi sta al compilatore associare (**blind**) gli indirizzi simbolici a quelli fisici, tale operazione di assegnazione di istruzioni e dati a indirizzi di memoria si può compiere nei seguenti passi:

* **Compilazione:** Se nella fase di compilazione si sa già dove il processo risiederà in memoria si può generare **codice assoluto.**
* **Caricamento:** Se nella fase di compilazione non si sa dove il processo risiederà in memoria il compilatore dovra generare un **codice riallocabile** da allocare in un secondo momento.
* **Esecuzione:** Se durante l’esecuzione il processo può essere spostato da un segmento di memoria all’altro, in questo caso si deve ritardare l’associazione degli indirizzi fino alla fase di esecuzione.

**8.1.3 Spazi di indirizzi fisici e logici:** un indirizzo generato dalla CPU viene detto **indirizzo logico**, mentre un indirizzo visto dall’unità di memoria, cioè caricato nel **registro degli indirizzi di memoria (MAR),** viene normalmente chiamato **indirizzo fisico**. Durante la fase di associazione visto che gli indirizzi logici non coincidono con quelli fisici ci si riferisce ad essi come **indirizzi virtuali**. L’insieme di tutti gli indirizzi generati da un programma viene definito, **spazio degli indirizzi logici/fisici**. L’associazione viene svolta dall’**unità di gestione della memoria (MMU).** Un semplice esempio di come quest’associazione avvenga è quello mostrato in figura, dove il programma utente genera un indirizzo logico che viene trasformato in fisco dalla MMU, è importante notare come il programma utente non conosca l’indirizzo fisico a cui fa riferimento.

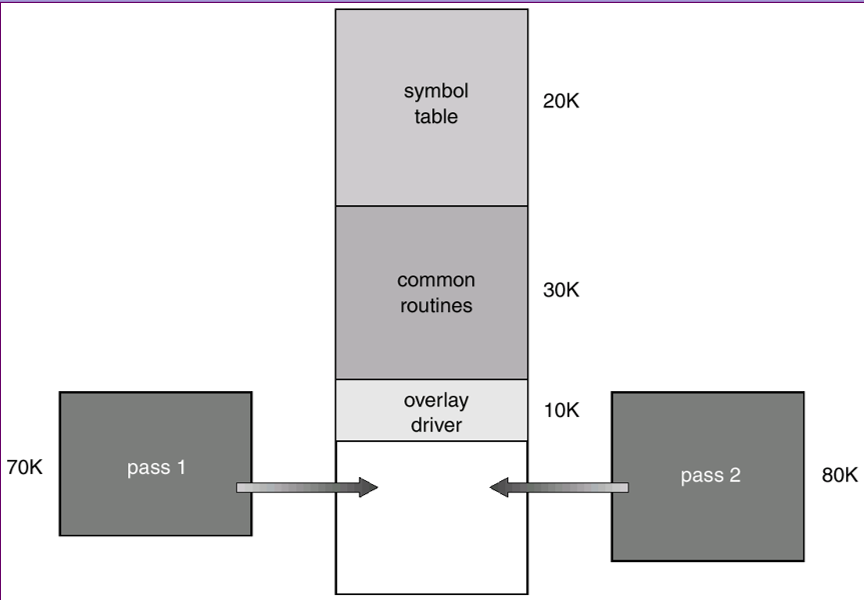


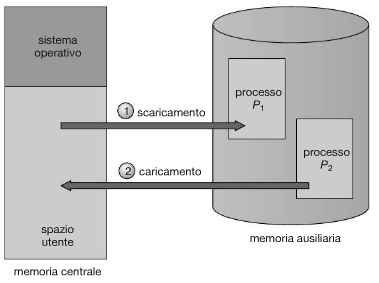
**8.1.4 Caricamento dinamico:** Un’altra possibilità esistente all’atto di caricare un processo in memoria è il **caricamento dinamico**  tale tipo di caricamente ci permette di caricare frammenti del programma solo quando sono necessari, in modo da non dover limitare la grandezza di quest’ultimi alla grandezza della memoria. L’implementazione del caricamento dinamico sta tutta al programmatore che deve gestire le chiamate alle funzioni.

**8.1.5 Linking dinamico e librerie condivise:** Le librerie collegate dinamicamente sono librerie di sistema che vengono collegate ai programmi utente quando questi vengono eseguiti, esistono due tipi di collegamento:

* **Collegamento statico:** In cui le librerie vengono trattate come un qualsiasi altro modulo oggetto e combinate dal caricatore nell’immagine binaria del programma
* **Collegamento dinamico:** Ha un funzionamento molto simile al caricamento dinamico, permettondoci cosi di caricare solo le librerie che diventano necessarie durante l’esecuzione, evitando cosi, tramite dei riferimenti detti **stuf,** di dover conservare una copia delle librerie per ogni programma. Tale proprietà torna anche molto utile nel caso di aggiornamento delle librerie (basta aggiornare gli stuf).

Nel caso in cui un programma utilizzi una libreria di una versione precendente, tale libreria viene caricata in memoria in più versioni, cosi facendo ogni programma tramite lo stuf e la versione utilizza quella con cui è compatibile, tale funzionalità e detta **libreria condivisa.**

**Extra: Sovrapposizione di sezioni (overlay):** Con il termine overlay si intende la possibilità di mantenenere in memoria soltanto le istruzioni e i dati dati che si usano con maggior frequenza. Quando sono necessarie altre istruzioni queste si caricano nello spazio precedentemente occupato dalle istruzioni che non sono più in uso. Tale metodo è necessario quando il processo è più grande della memoria che gli può essere assegnata. Implementato in genere dall’utente. Non vi è supporto da parte del S.O. Utilizzato nei microcalcolatori e in sistemi dotati di memoria limitata.

**8.2 Avvicendamento dei processi (swapping):** Come abbiamo visto un processo per essere eseguito deve essere caricato in memoria, può però capitare che per aumentare il grado di multiprogrammazione del sistema sia necessario spostare un processo dalla memoria centrale ad’una **memoria ausiliaria** tale procedimento è detto **avvicendamento dei processi in memoria** o **swapping.**

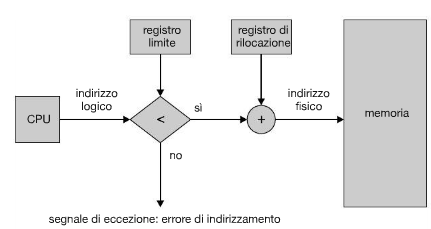
**8.2.1 Avvicendamento standard:** tale avvicendamento riguarda lo spostamento dei processi tra la memoria centrale e quella ausiliaria. Per fare ciò il sistema mantine una **ready queue**  con tutti i processi pronti, quando lo scheduler decide di avviare un determinato processo il controllo passa al **dispatcher** che si occupa di verificare dove il processo si trova (memoria centrale o ausiliaria) ed effettuare le relative operazioni di caricamento o scaricamento. Ovviamente in caso di swappingo il **tempo di cambio di contesto** è estremamente alto. Inoltre è importante notare che un processo può essere scaricato dalla memoria solo se è completamente inattivo.

**8.2.2 Avvicendamento di processi nei sistemi mobili:** La maggior parte dei sistemi operatici PC e server, supporta una qualche versione dell’avvicendamento, ciò non si può però affermare per i dispositivi mobili che utilizzano solitemante memorie flash che visto il loro spazio e numero di scritture limitato non permettono di implementare un buon algoritmo di swapping. Invece di usare lo **swapping** sistemi operativi di questo genere ,come IOS e Andorid, sono in grado di terminare volontariamente i processi per liberare memoria.

**8.3 Allocazione contigua della memoria:** Come abbiamo visto la memoria centrale deve mantenere sia il sistema operativo che i processi utente, per questo motivo la memoria centrale è divisa in due parti:

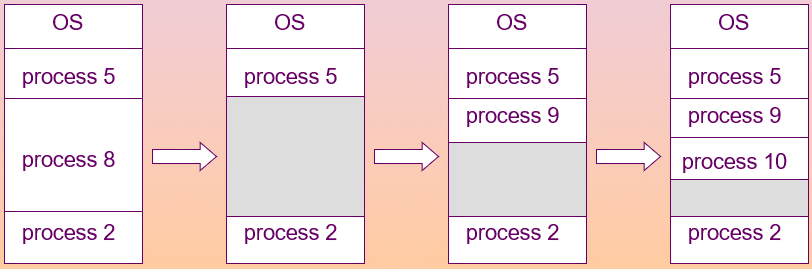
* **Parte riservata al OS:** posta di solito nella parte bassa della memoria.
* **Parte riservata ai processi utenti:** parte alta della memoria.

Generalmente si vuole che più processi utenti risiedano conteporaneamente nella mermoria. Con **l’allocazione contigua della memoria** ciascun processo è contenuto in una singola sezione di memoria contigua a quella che contiene il processo successivo.

**8.3.1 Protezione della memoria:** Il problema della sicurezza della memoria è stato risolto come abbiamo visto in precedenza tramite il **registro limite** e il **registro di rilocazione,** tali registi contengono il pool di indirizzi fisici della memoria in cui il processo può operare. Tale schema permette al sistema operativo di cambiare dinamicamente le proprie dimensioni, eliminando per esempio dati presenti in memoria che non sono utilizzati frequentemente, questo codice prende il nome di codice **transiente** del sistema operativo, poiché viene caricato in memoria solo se necessario.

**8.3.2 Allocazione della memoria:** Uno dei metodi pià semplici per l’allocazione della memoria consiste nel dividere quest’ultima in **partizioni** di dimensione fissa, ogni partizione contiene un processo, quindi in questa soluzione il grado di multiprogrammazione è limitato al numero di partizioni della memoria. In questo schema il sistema operatico conserva una tabella in cui sono indicate le partizioni di memoria diponibili è quelle occupate non che la loro dimensione, inizialmente la memoria è vista come un grande blocco che viene suddiviso in partizioni di dimensione fissa man mano che arrivano i processi. Quando un processo termina l’esecuzione lascia un **buco**, che verrà occupato dal primo processo che con dimensioni minori o uguali a quello del buco, i principali metodi per scegliere il buco libero all’interno della memoria sono:

* **First-fit:** Si assegna il primo buco abbastanza grande.
* **Best-fit:** Si assegna il più piccolo buco in grado di contenere il processo.
* **Wors-fit:** si assegna il buco più grande.

Tramite opportune simulazioni si può affermare che i primi due metodi sono i migliori sia in termini di tempo che utilizzo della memoria.

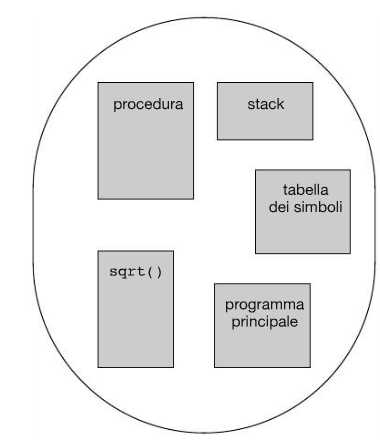
**8.3.3 Frammentazione:** L’utilizzo di questo metodo di allocazione della memoria può portare a dei problemi di frammentazione, essi sono principalmente di due tipo:

* **Frammentazione esterna:** Lo spazio complessivo di memoria disponibile necessario per soddisfare la richiesta esiste, ma non è contiguo.
* **Frammentazione interna:** La memoria allocata è leggermente più grande di quella richiesta, questa memoria residua, interna alla partizione, non viene utilizzata.

La frammentazione esterna può essere ridotta con un’operazione di compattamento. Il contenuto della memoria viene riordinato per riunire la memoria libera in un solo grosso blocco. Tale operazione è possibile solo se la rilocazione è dinamica e si compie in fase di esecuzione.

**Problemi realtivi all’ I/O:** Un job non può essere spostato in un’altra porzione di memoria mentre esegue operazioni di I/O. Una possibile soluzione sta nell’eseguire operazioni di I/O solo usando buffer nella porzione di memoria del S.O..

**8.4 Segmentazione:** La segmentazione è un meccaniscmo che ci permette di mappare la memoria vista dal punto di vista del programmatore sulla memoria fisica.

**8.4.1** **Metodo base:** Dal punto di vista del programmatore, un porgramma non è visto come una seguenza di istruzioni lineari contenute in un array di byte, ma come un insieme di procedure e costrutti di cui la posizione in cui sono memorizzati non è importante. La **segmentazione** è uno schema di gestione della mememoria che supporta questo tipo di rappresentazione ad **“alto livello”**. Essa viene implementata tramite uno spazio di **indirizzi logici** che non è altro che un’insieme di segmenti ognuno dei quali ha un **nome** è un **offset.** Tale indirizzo logico è detto **coppia:**

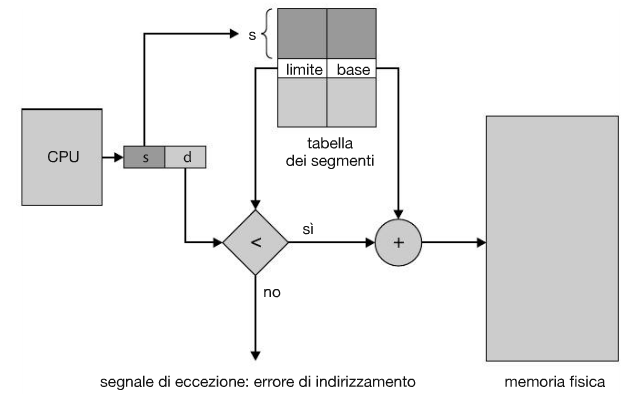
**<numero di segmento, offset>**

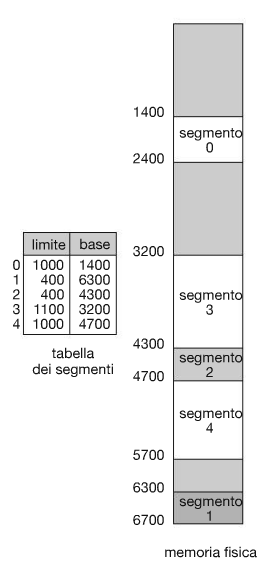
Quando un programma viene compilato il compilatore costruisce automaticamente i segmenti in rapporto al programma sorgente, in particolare un compilatore C crea un segmento per ognuno dei seguenti elementi:

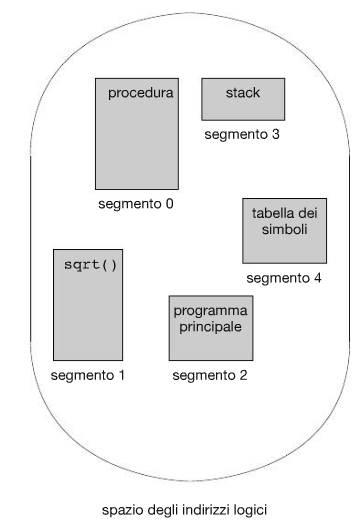
1. **Codice** 3. **Heap**
2. **Variabili locali** 4. **Stack** 5. **Librerie**

**8.4.2 Hardware di segmentazione:** Sebene il programmatore può utilizzare la memoria come se fosse un’oggetto bidimensionale, essa non lo è, per questo motivo è necessario tradurre gli indirizzi bidimensionali utilizzati dal programmatore in indirizzi unidimensionali, tale operazione si svolge tramite una **tabella dei segmenti**, dove per ogni segmento è memorizzato:

* **La base del segmento:** che ne indica l’indirizzo fisico di inizio
* **Il limite del segmento:** che ne indica la lunghezza.

Se la lunghezza dell’offset è compresa tra 0 e la lunghezza del segmento, quest’ultimo viene sommato all’indirizzo base.

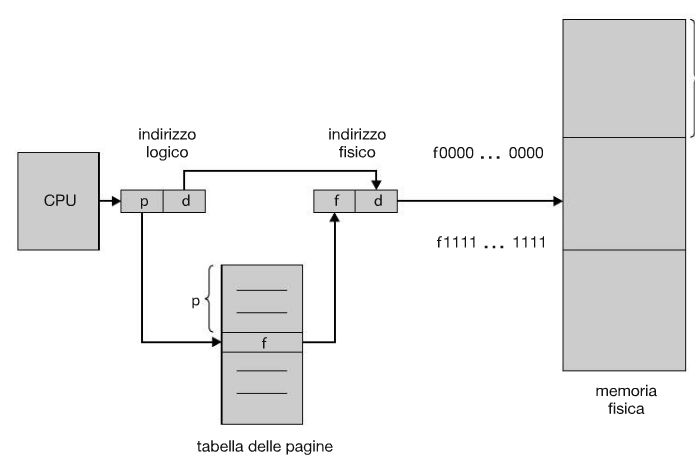
**Esempio:** 



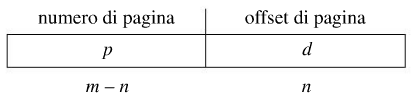
**8.5 Paginazione:** La segmentazione come abbiamo visto permette agli indirizzi fisici del programma di non essere contigui. Un metodo equivalente alla segmentazione è la **paginazione (paging),** che offre importanti vantaggi: evita la **frammentazione esterna** e quindi la necessità di effettuare la compattazione, evita la necessita di scaricare i segmenti di memoria su memorie ausiliari.

**8.5.1 Metodo base:** Il metodo base per implementare tale metodo di gestione delle memoria consiste nel dividere la **memoria fisica** in blocchi di dimensione fissa detti **frame** e nel suddividere la **memoria logica** in blocchi di pari dimensione detti **pagine.** Quando in processo è pronto ad essere eseguito si prendono le sue pagine nei frame di memoria disponibili.

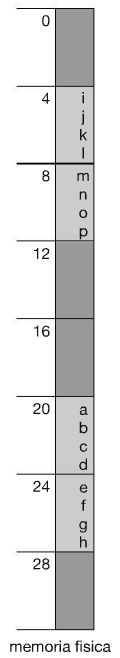
Ogni indirizzo generato dalla CPU è diviso in due parti:

* **Numero di pagina (p):** serve come indice per la **tabella delle pagine** che contiene gli indirizzi base.
* **Offset di pagina (d):** posizione dell’indirizzo di memoria desiderato all’interno del blocco.

**Esempio:** 

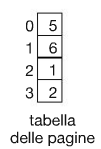
La dimensione di una pagina, cosi come quella di un frame è definita tramite potenza di 2. In questi casi l**’indirizzo logico**  ha il seguente formato: 

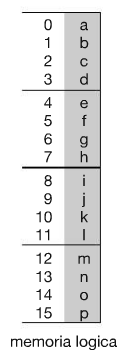
**Esempio:** Supponiamo di avere una memoria fisica di 32 byte con ogni pagina di 4 byte (ossia 8 pagine) . L' indirizzo logico 0 è pagina 0, offset 0. Nella tabella delle pagine, la pagina 0 si trova nel frame 5. Quindi l' indirizzo logico 0 viene mappato nell' indirizzo fisico 20 (20=(5 x 4)+0).

L' indizzo logico 3 (pagina 0, offset 3) viene mappato nell' indirizzo fisico 23 (23=(5x4)+3).

L' indirizzo logico 4 è pagina 1, offset 0; secondo la tabella delle pagine, la pagina 1 è mappato nel frame 6. L' indirizzo 4 viene mappato nell' indirizzo fisico 24 (24 = (6x4) +0).

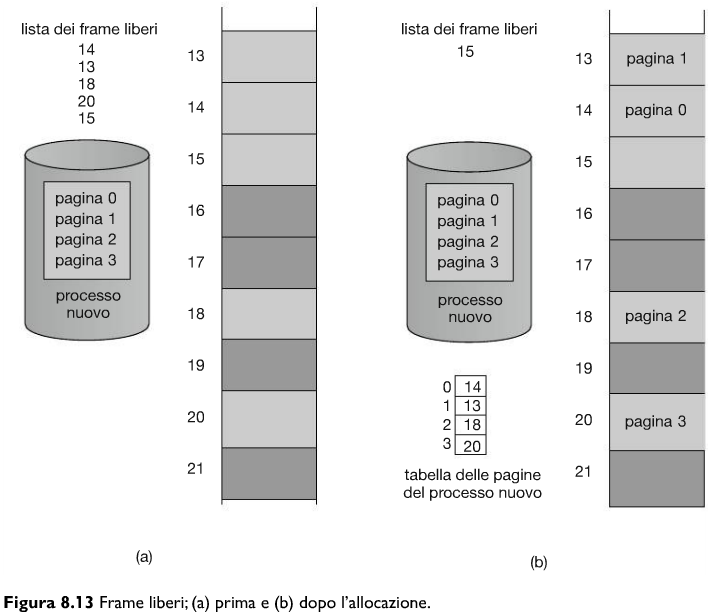
L' indirizzo logico 13 viene mappato nell' indirizzo fisico 9.





Come affermato in precedenza la paginazione risolve il problema della **frammentazione esterna**, ma non quello della **frammentazione interna** infatti potrebbe capitare che un processo non si in grado di occupare per intero una pagina. Per questo motivo è buona norma utilizzare frame di dimensioni ridotte in modo da ridurre l’**ovehead.** Ogni volta che un nuoco processo viene creato il sistema operativo tramite la **tabellad di frame** verifica il numero di frame liberi, se il numero dei frame è uguale al numero delle pagine del nuovo processo, il processo viene caricato in memoria.

**Esempio:**

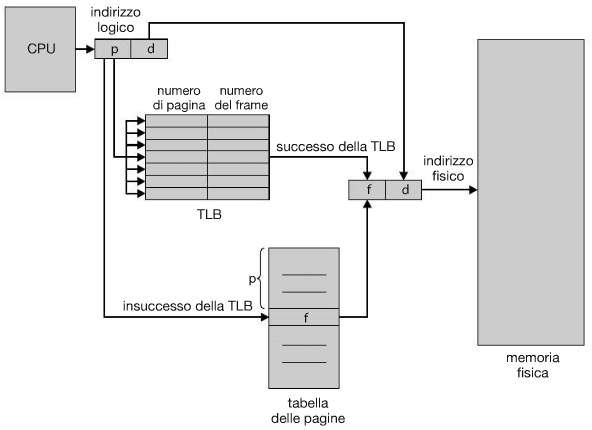
****

**8.5.2 Supporto hardware alla paginazione:** All’interno dei calcolatori moderni la **tabella delle pagine** vista la sua grandezza (nell’ordine di milioni di elementi, processi grandi frame piccoli) viene mantenuta all’interno della memoria centrale (non si puà mettere in registri veloci appunto per la sua grandezza) dove è puntata da un **registro di base della tabella delle pagine (PTBR),** in modo che durante il cambio di contesto, invece di dover ricaricare tutta la tabella basta semplicemente puntare ad’una diversa. Questo metodo ha però un problema: per accedere ad’un indirizzo di memoria dobbiamo effettuare **2 accessi** (uno per accedere alla tabella delle pagine e uno per accedere alla locazione fisica che ci interezza). Per risolvere questo problema consiste nell’impiego di una piccola **chace hardware** detta **TLB** in cui ogni elemento consiste di due parti:

* **Chiave o tag**
* **Valore**

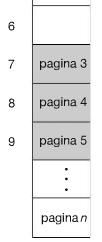
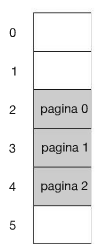
Quando la CPU genera un nuovo indirizzo logico, si presenta il suo numero di pagina alla **TLB** se tale indirizzo (chiave) è presente si ottiene direttamente l’indirizzo fisico (valore) evitando un accesso in memoria.

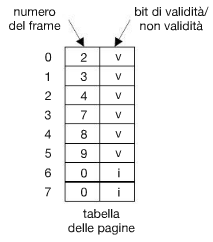
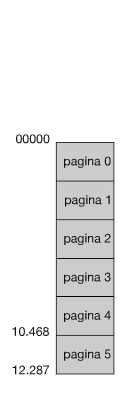
**Esempio:**

****

Nel caso in cui l’indirizzo appena generato dalla CPU non sia presente all’iterno della **TLB** siamo costretti ad’accedere alla memoria, tale situazione prende il nome di **insuccesso della TLB.** Inoltre in caso di insuccesso il riferimento appena recuperato della memoria viene inserito nella **TLB,** nel caso in cui questa sia piena occorre sostituire alcuni elementi. Esistono molti criteri di sostituzione, per sempio alcuni **TLB**  consentono che alcuni elementi siano **vincolati** cioè che non possano essere rimossi dalla TLB. Un altro valore molto importante per le TLB è il **tasso di successi** ovvero il numero di volte in cui il numero di pagina interessato si trova nella TLB.

**8.5.3 Protezione:** in un’abbiente paginato la protezione della memoria è assegnata ad’un bit di protezione associato ad’ogni **frame,** tale bit regola il tipo di operazioni che si possono effettuare con quel frame (lettura o scrittura, e tramite alcune estensioni anche esecuzione). Oltre a tale bit può essere implemetato anche il **bit di validità** che permette di verificare se una pagina è **valida** o **non valida** per quel processo. In entrambi i casi in caso di errore si genera un’eccezione.

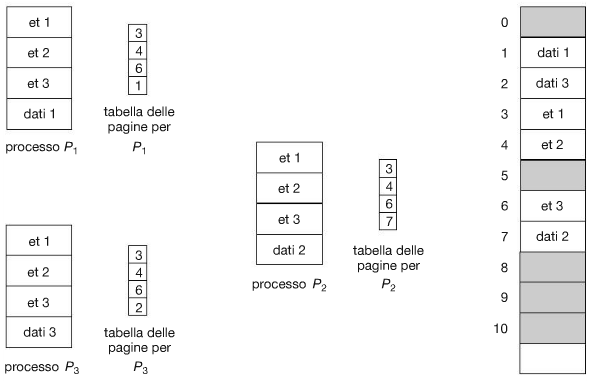
**Esempio:** 



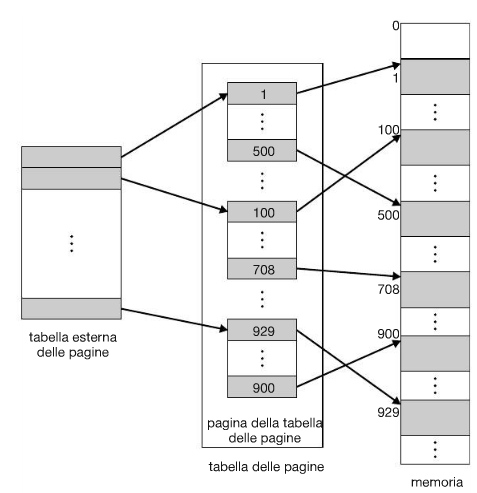
**Osservazione:** in questo esempio possiamo notare come ci sia uno spreco di pagine (la pagina cinque non è utilizzata dal programma), per risolvere tale problema alcune architetture dispongono di alcuni registri detti: **registri di lunghezza della tabella delle pagine (PTRL)** per indicare la lunghezza effettiva delle tabelle.

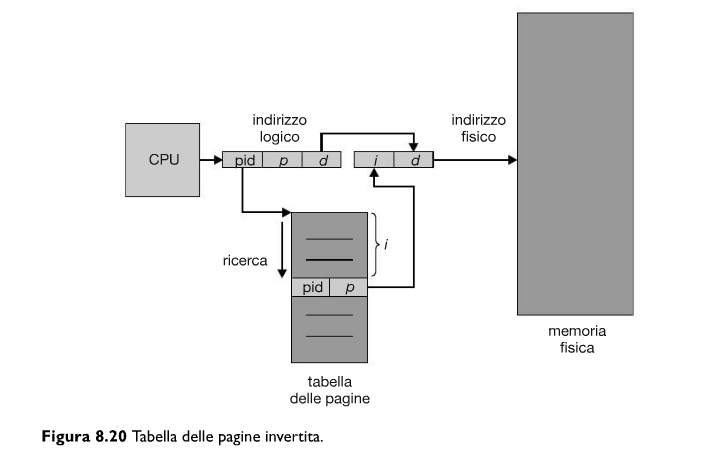
**8.5.4 pagine condivise:** Un altro vantaggio della paginazione è il fatto di poter condividere porzioni di memoria e quindi di codice, il codice che può essere condiviso viene chiamato **codice rientrante** o **codice puro,** ovvero tutta quella parte di codice non auto-modificabile (non si modifica durante l’esecuzione). Se per esempio abbiamo un text editor di 200KB (150 di codice e 50 di dati utente) possiamo risparmiare spazio in memoria utilizzando per tutte le n istanze del text editor sempre gli stessi 150KB di codice, occupando come spazio extra solo 50KB di dati utente.

**Esempio:**

****

**8.6 Struttura della tabella delle pagine:** ecco le tecniche più usate per la paginazione:

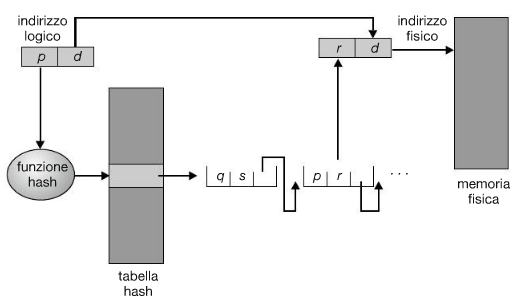
**8.6.1 Paginazione gerarchica:** Generalmente, si associa una tabella delle pagine a ogni processo e tale tabella contiene un elemento per ogni pagina virtuale che il processo sta utilizzando, ossia vi sono elementi corrispondenti ad ogni indirizzo virtuale a prescindere dalla validità di quest’ultimo. Poiché la tabella è ordinata per indirizzi virtuali, il sistema operativo può calcolare in che punto della tabella si trova l’indirizzo fisico associato. Unico inconveniente e che l tabella può contenere milioni di elementi e quindi può occupare una grande quantità di memoria. Per risolvere questo problema si può fare uso della tabella delle pagine invertita.  
La tabella delle pagine invertita ha un elemento per ogni pagina reale (frame). Ciascun elemento è quindi costituito dell’indirizzo virtuale della pagina memorizzata in quella reale locazione di memoria, con informazioni sul processo che possiede tale pagina. Quindi nel sistema esiste solo una tabella delle pagine che ha un solo elemento per ciascuna pagina di memoria fisica



Sebbene questo schema riduca la quantità di memoria necessari per memorizzare ogni tabella delle pagine, aumenta però il tempo di ricerca nella tabella quando si fa riferimento ad una pagina, per trovare una corrispondenza potrebbe essere necessario esaminare ogni elemento della tabella. Per limitare l’entità del problema si può impiegare una tabella hash che riduce la ricerca a un solo, o a pochi, elementi della tabella. I sistemi che adottano questa soluzione hanno difficoltà nell’implementazione della memoria condivisa. Una semplice soluzione consiste nel porre nella tabella delle pagine una sola associazione fra un indirizzo virtuale e l’indirizzo fisico condiviso, ciò comporta un errore dovuto all’assenza della pagina per ogni riferimento agli indirizzi virtuali non associati.

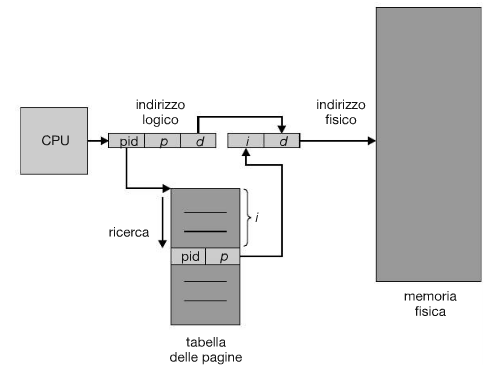
**8.6.2 Tabella delle pagine di tipo hash:** In questo metodo l’argomento di della funzione hash è il numero della pagina virtuale. Per risolvere il problema delle collisioni (savastano che grande) ogni elemento della tabella contiene una lista concatenata per disambiguare. Per qeusto motivo ciascun elemento è composto da tre campi:

* **Numero della pagina virtuale**
* **Indirizzo del frame**
* **Puntatore al successivo elemento della lista**

si applica la funzione hash al numero della pagina contenuto nell’indirizo virtuale, identificando un elemento della tabella, poi si confronta l’indirizzo logico con i vari elementi della lista concatenata. 

**8.6.3 Tabella delle pagine invertita:** Per risolvere il problema della dimensione della tabella delle pagine si può fare uso della **tabella delle pagine invertita** che ha un elemento per ogni pagina reale (frame) di memoria. Ciascun elemento è quindi costituito dall' indirizzo virtuale della pagina memorizzata in quella reale locazione di memoria, con informazioni sul processo che possiede tale pagina. Quindi, nel sistema esiste solo una tabella delle pagine che ha un solo elemento per ciascuna pagina di memoria fisica. Ciascun indirizzo virtuale è formato dalla seguente tripla:

**<id-processo,numero di pagina,offset>**

Ogni elemento della tabella delle pagine invertita è una coppia **<id-processo,numero di pagina>**. Quando viene effettuato un riferimento alla memoria, parte dell' indirizzo virtuale, formato da **<id-processo,numero di pagina>**, viene presentato al sottosistema di memoria. Quindi viene cercata una corrispondenza nella tabella delle pagine invertita. Se tale corrispondenza viene trovata, ad esempio sull' elemento i, viene generato l' indirizzo fisico **<i,offset>**, altrimenti è stato tentato un accesso illegale a un indirizzo.

Sebbene questo schema riduca la quantità di memoria necessaria per memorizzare ogni tabella delle pagine, aumenta però la quantità di tempo necessaria per cercare la tabella quando viene fatto riferimento a una pagina. La tabella delle pagine invertita è ordinata per indirizzo fisico, mentre le ricerche vengono effettuate su indirizzi virtuali. Per trovare una corrispondenza occorre cercare in tutta la tabella, e questa ricerca richiede molto tempo. Per alleviare il problema può essere utilizzata una tabella hash per limitare la ricerca a un solo, o a pochi, elementi della tabella delle pagine.

**Capitolo 9 Memoria Virtuale**

**9.1 Introduzione**

La memoria virtuale si fonda sulla separazione della memoria logica percepita dall’utente dalla memoria fisica. La memoria virtuale facilità la programmazione in quanto il programmatore non deve preoccuparsi della quantità di memoria fisica disponibile. L’espressione spazio degli indirizzi virtuali si riferisce alla collocazione dei processi in memoria dal punto di vista logico. Lo spazio degli indirizzi virtuali è collocato tra l’heap e lo stack ed è un enorme buco vuoto. Uno spazio di indirizzi virtuali che contiene buchi si definisce sparso. Un simile spazio di indirizzi è utile, poiché i buchi possono essere riempiti grazie all’espansione dei segmenti heap o stack, oppure se vogliamo collegare dinamicamente delle librerie durante l’esecuzione del programma. Oltre a separare la memoria logica da quella fisica, la memoria virtuale offre il vantaggio di condividere i file e la memoria fra due o più processi, mediante la condivisione delle pagine. Ciò comporta i seguenti vantaggi:

1. Le librerie di sistema sono condivisibili da diversi processi associando l’oggetto di memoria condiviso a uno spazio degli indirizzi virtuali
2. La memoria può essere condivisa tra processi distinti
3. Le pagine possono essere condivise durante la creazione di un processo mediante la chiamata di sistema fork(), così da velocizzare la generazione dei processi

**9.2 Paginazione su richiesta**

Un’ottima soluzione al problema del caricamento in memoria consiste nel caricare le pagine nel momento in cui servono realmente, questa tecnica è chiamata paginazione su richiesta. Secondo questo schema le pagine sono caricate in memoria solo quando richieste durante l’esecuzione del programma. Si utilizza un metodo d’avvicendamento “pigro” (lazy Swapping): non si carica mai in memoria una pagina che non sia necessaria.

**9.2.1 Concetti fondamentali**

Quando un processo sta per essere caricato in memoria, il paginature ipotizza quali pagine saranno usate, prima che il processo sia nuovamente scaricato in memoria. In questo modo si riduce il tempo d’avvicendamento. Con un tale schema occorre un hardware che sia in grado di identificare quali pagine siano in memoria e quali siano nei dischi. Il bit di validità serve a questo scopo. Se una pagina è contrassegnata come non valida non si ha nessun effetto se il processo non tenta mai di accedervi. Se l’ipotesi del paginatore è esatta si caricano tutte e solo le pagine che servono effettivamente. Durante l’esecuzione, finché il processo accede alle pagine residenti in memoria, l’esecuzione procede come di consueto. Un accesso ad una pagina non valida genera un’eccezione di page fault. Per gestire tale eccezione:

1. Si controlla una tabella interna per questo processo allo scopo di stabilire se il riferimento fosse un accesso alla memoria valido o non valido
2. Se il riferimento non era valido, si termina il processo. Se era un riferimento valido, ma la pagina non era ancora stata portata in memoria, se ne effettua il caricamento
3. Si individua un frame libero
4. Si programma un’operazione sui dischi per trasferire la pagina desiderata nel frame appena assegnato
5. Quando la lettura dal disco è completata, si modificano la tabella interna, conservata con il processo, e la tabelle delle pagine per indicare che la pagina si trova attualmente in memoria
6. Si riavvia l’istruzione interrotta dall’eccezione.

L’Hardware di supporto alla paginazione su richiesta è lo stesso che è richiesto per la paginazione e l’avvicendamento dei processi in memoria:

1. tabelle delle pagine
2. memoria secondaria

Uno dei requisiti cruciali della paginazione si richiesta è la possibilità di rieseguire una qualunque istruzione dopo un page fault. Avendo salvato lo stato del processo interrotto al momento del page fault, occorrerà riavviare il processo esattamente nello stesso punto e con lo stesso stato, ad eccezione per la presenza della pagina in memoria. Nella maggior parte dei casi questo requisito è facile da rispettare.

**9.2.2 Prestazioni della pagina su richiesta**

Attualmente il tempo d’accesso alla memoria varia da 10 a 200 nanosecondi. Finché non si verifichino page fault, il tempo di accesso effettivo è uguale al tempo di accesso alla memoria.

Supponendo che p sia la probabilità che si verifichi un page fault. È probabile che p sia molto vicino allo zero, cioè che ci siano solo pochi fault. Il tempo di accesso effettivo è dato dalla seguente espressione:

tempo di accesso effettivo = (1 - p) – tempo di accesso alla memoria + p x tempo di gestione del page fault  
La gestione di un page fault comporta i seguenti passi:

1. Trap al SO
2. Salvataggio dei registri utente e dello stato del processo
3. Verifica che l’interruzione sia dovuta o meno ad un page fault
4. Controllo della correttezza del riferimento alla pagina e localizzazione della pagina su disco
5. Lettura dal disco e trasferimento
   1. Attesa nella coda al dispositivo di I/O
   2. Attesa dovuta a posizionamento e latenza
   3. Trasferimento della pagina in un frame libero
6. Durante l’attesa: allocazione della CPU ad altro processo utente
7. Ricezione dell’interrupt da disco (I/O completato)
8. Context switch (se è stato eseguito il passo 6))
9. Verifica della provenienza dell’interruzione da disco
10. Aggiornamento della tabella delle pagine (e dei frame) per segnalare la presenza in memoria della pagina richiesta
11. Attesa nella ready queue
12. Context switch di accesso alla CPU

In ogni caso, il tempo di servizio dell’eccezione di page fault è costituito da tre componenti principali

1. Servizio del segnale di eccezione di page fault
2. Lettura della pagina da disco
3. Riavvio del processo

Se ben codificate, le operazioni di servizio constano di poche centinaia di istruzioni, mentre il tempo di accesso a disco costituisce il contributo più significativo.

**9.3 Copiatura su scrittura**

La duplicazione dello spazio di indirizzi causata della fork() può risultare inutile se la prima istruzione eseguita dal processo figlio è la exec(). Per questo si può impiegare una tecnica nota come copiatura su scrittura che permette la condivisione iniziale delle stesse pagine in memoria. Le pagine condivise si contrassegnano come pagine da copiare su scrittura, cioè se un processo scrive su una pagina condivisa, il sistema deve creare una copia di tale pagina. Quando è necessario la duplicazione di una pagina secondo la tecnica di copiatura su scrittura, è importante capire da dove attingerà la pagina libera necessaria. Molti sistemi operativi forniscono, per queste richieste, un gruppo di pagine libere, che di solito si assegnano quando lo stack o lo heap di un processo devono espandersi. L’allocazione di queste pagine avviene secondo una tecnica nota come azzeramento su richiesta; prima dell’allocazione si riempiono di zeri le pagine, cancellandone in questo modo tutto il contenuto precedente.

**9.4 Sostituzione delle pagine**

La sovra locazione si ha quando durante l’esecuzione di un processo utente si verifica un page fault. Il sistema operativo determina la locazione del disco in cui risiede la pagina desiderata, ma poi scopre che la lista dei frame liberi è vuota: tutta la memoria è in uso. Una soluzione è terminare il processo.

**9.4.1 Sostituzione di pagina**

Se nessun frame è libero, ne viene liberato uno attualmente non utilizzato. È possibile liberarlo scrivendo il suo contenuto nell’area di swap e modificando la tabella delle pagine per indicare che la pagina non si trova più in memoria. La procedura della gestione dell’eccezione di page fault è leggermente modificata per permettere di includere la sostituzione della pagina. Se nessun frame è libero occorre effettuare due trasferimenti, questo porta all’aumentare il tempo effettivo d’accesso. Questo sovraccarico si può ridurre utilizzando un bit di modifica. Se viene posto ad 1 significa che nella pagina si scrive un byte. Quando si sceglie la pagina vittima si controlla questo byte, se è uguale a 0 significa che la pagina non è stata modificata quindi non ha senso doverla riscrivere nel disco, dato che è già presente. Per realizzare la paginazione su richiesta è necessario risolvere due problemi principali: occorre sviluppare un algoritmo di allocazione dei frame e un algoritmo di sostituzione delle pagine. Un algoritmo si valuta in base al numero di page fault effettuati su una successione di riferimenti alla memoria.

**9.4.2 Sostituzione della pagine secondo l’ordine d’arrivo (FIFO)**

Questo algoritmo associa a ogni pagina l’istante di tempo in cui quella pagina è stata portata in memoria. Se si deve sostituire una pagina, si seleziona quella presente in memoria da più tempo. È possibile creare una coda FIFO invece di registrare l’istante in cui si carica una pagina in memoria. Questo algoritmo è semplice da realizzare ma le prestazioni fanno un pochino schifo. In alcuni algoritmi è presente la cosiddetta anomalia di Belady per cui la frequenza dei page fault può aumentare con il numero di frame assegnati ai processi.

**9.4.3 Sostituzione ottimale delle pagine**

Questo algoritmo presenta il tasso minimo di page fault e non presenta mai l’anomalia di Belady. Consiste nel sostituire la pagina che non verrà usata per il periodo di tempo più lungo. È difficile da realizzare perché richiede la conoscenza futura della successione dei riferimenti. Si utilizza solo come studio comparativo

**9.4.4 Sostituzione delle pagine meno recentemente (LRU)**

Questo algoritmo sostituisce la pagina che non è stata utilizzata per il periodo più lungo. Simile all’algoritmo ottimale ma si basa sul passato. Questo algoritmo è molto utilizzato, il problema riguarda la sua implementazione. Il problema consiste nel determinare un ordine per i frame definito dal momento dell’ultimo uso. Si possono realizzare due soluzioni:

1. Contatori 🡪 Ad ogni elemento della tabella delle pagine si associa un nuovo campo chiamato “momento di utilizzo” e alla CPU si aggiunge un contatore che si incrementa a ogni riferimento alla memoria. Ogni volta che si fa riferimento ad una pagina si copia il contenuto del contatore della CPU nel campo “momento di utilizzo”. In questo modo è possibile conoscere il momento in cui è stato fatto l’ultimo riferimento a ogni pagina. L’algoritmo sostituisce la pagina con il valore del campo “momento di utilizzo” più piccolo. Questa soluzione prevede una ricerca e una scrittura per ogni acceso alla memoria.
2. Stack 🡪 si utilizza uno stack per immagazzinare i numeri di pagina. Ogni volta che si fa riferimento a una pagina, la si estrae dallo stack e la si colloca in cima a quest’ultimo. In questo modo, in cima allo stack si trova sempre la pagina usata per ultima. L’implementazione viene realizzata con una lista doppiamente concatenata. Ogni aggiornamento è un po' costoso ma per la sostituzione non serva fare una ricerca.

Sia l’algoritmo ottimale sia l’algoritmo LRU non sono soggetti all’anomalia di Belady. Appartengono ad una classe di algoritmi di sostituzione delle pagine, chiamati algoritmi a stack.

**9.4.5 Sostituzione delle pagine per approssimazione a LRU**

Non tutti i sistemi dispongono del supporto hardware per una vera sostituzione LRU delle pagine. Per questo viene usato l’algoritmo LRU con bit di riferimento. Un bit di riferimento è assegnato automaticamente ad una pagina dall’hardware di sistema ogni volta che si fa un riferimento a quella pagina. Inizialmente il sistema operativo imposta tutti i bit a zero. Quando si inizia l’esecuzione di un processo utente, l’hardware imposta a 1 il bit associato a ciascuna pagina cui si fa riferimento. In questo modo è possibile conoscere le pagina che sono usate ma non il loro ordine d’uso.

**5.4.5.1 Algoritmo con bit supplementari di riferimento**

È possibile conservare una tabella in memoria con un byte per ogni pagina. Periodicamente viene salvato il bit di riferimento di tutte le pagine. Intervalli regolai il sistema inserisce il bit di riferimento per ciascuna pagina nel bit più significativo shiftando gli altri bit a destra, in questo modo il bit meno significativo viene scartato. Questo significa che il sistema può tener traccia dell’uso delle varie pagine e quando bisogna sostituire una pagina, si può sostituire la pagina usata meno di recente, ovvero quella che ha il byte con valore più basso.

**9.4.5.2 Algoritmo con seconda change**

Dopo aver selezionato una pagina si controlla il bit di riferimento: se il suo valore è zero, si sostituisce la pagina, se invece il valore è impostato ad 1, si dà una seconda chance alla pagina e si passa alla successiva pagine FIFO. Quando ad una pagina si dà una seconda chance, significa rimpostare il suo bit di riferimento a o e si aggiorna il suo istanze d’arrivo al momento attuale. In questo modo, la pagina con bit 1 ha una seconda chance perché può rimpostare il bit ad 1 prima della prossima verifica, se invece la pagina dopo che gli è stata fornita una seconda chance non viene mai usata il suo bit rimane a = e alla prossima verifica verrà scartata. Un’implementazione di tale algoritmo fa uso di una coda circolare, in cui un puntatore punta alla prima pagina da sostituire. Se tutti i bit sono impostati ad uno l’algoritmo si trasforma in una sostituzione di tipo FIFO

**9.4.5.3 Algoritmo con seconda chance migliorato**

L’algoritmo Seconda Chance può essere migliorato usando come base la coppia:

1. Bit di riferimento: la pagina è stata/ non è stata acceduta di recente.
2. Bit di modifica (dirty bit): la pagina è stata/non è stata modificata dal suo caricamento (e di conseguenza deve/non deve essere salvata).

In questo modo si possono ottenere i seguenti quattro casi:

1. (0, 0) 🡪 né recentemente usata né modificato 🡪 pussa via
2. (0, 1) 🡪 non usato recentemente ma modificato 🡪 bisogna prima salvare le modifiche
3. (1, 0) 🡪 usato recentemente ma non modificato 🡪 con molta probabilità la pagina sarà usata nuovamente
4. (1, 1) 🡪 usata recentemente e modificata 🡪 peggio pagina da sostituire, perché prima bisogna salvare le modifiche e inoltre potrebbe essere usata di nuovo

**9.4.6 Sostituzione delle pagine basata su conteggio**

Si potrebbe usare un contatore del numero dei riferimenti fatti a ciascuna pagina:

1. LFU 🡪 si sostituisce la pagina con il conteggio più basso. Si presuppone che una pagina usata frequentemente abbia un valore di conteggio alto
2. MFU 🡪 si sostituisce la pagina con il conteggio più basso, si basa sul concetto che probabilmente una pagina con un conteggio basso è stata appena inserita

Sti due algoritmi non sono usati molto.

**9.4.7 Algoritmi con buffering delle pagine**

I sistemi in genere hanno uno spazio di memoria chiamato pool of free frames. Quando si verifica un page fault e si è scelta la pagina da sostituire, si traferisce pagina richiesta in questo gruppo libero, in questo modo il processo può riprendere subito l’esecuzione senza dover aspettare che la pagina vittima venga scritta prima in memoria secondaria.

**9.5 Allocazione dei frame**

Occorre stabilire un criterio per l’allocazione della memoria libera ai diversi processi. La strategia di base è quella di assegnare al processo utente qualsiasi frame libero.

**9.5.1 Numero minimo di frame**

Le strategie di allocazione dei frame sono soggette a molteplici vincoli. Uno di questi è quello di assegnare un numero minimo dio frame. Questo vincolo è legato alle prestazioni, al decrescere del numero di frame allocati a ciascun processo aumenta il tasso di page fault. Il numero minimo di frame è definito dall’architettura del calcolatore.

**9.5.2 Algoritmi di allocazione**

Il modo più semplice per suddividere m frame tra n processi è quello per cui a ciascuno si da una parte uguale, m/n frame. Questo schema è chiamato allocazione uniforme. Un’alternativa consiste nel riconoscere che i diversi processi hanno bisogno di quantità di memoria diverse. Questo schema è chiamato all’allocazione proporzionale, secondo cui la memoria disponibile si assegna a ciascun processo secondo la propria dimensione. Gli entrambi schemi di allocazione possono variare rispetto al livello di multiprogrammazione. Se tale livello aumenta, ciascun processo perde alcuni frame per fornire la memoria necessaria per il nuovo processo.

**9.5.3 Allocazione globale e allocazione locale**

Gli algoritmi di sostituzione delle pagine si possono classificare in due gruppi:

1. Sostituzione globale 🡪 permette di scegliere il frame vittima dall’insieme di tutti i frame
2. Sostituzione locale 🡪 permette di scegliere il frame vittima solo dal proprio insieme di frame

L’algoritmo di sostituzione globale ha un problema, un processo non può controllare il proprio tasso di page fault, perché l’insieme delle sue pagine non dipendono solo dal comportamento di paginazione del processo stesso, ma anche dal comportamento degli altri processi. L’algoritmo di sostituzione locale non soffre di questo problema. La sostituzione locale però potrebbe penalizzare un processo, non rendendogli disponibili altre pagina di memoria.

**9.6 Thrashing**

Un processo in thrashing spende più tempo per la paginazione che per l’esecuzione del suo compito

**9.6.1 Cause del Thrashing**

La causa del thrashing è innescata quando il sistema operativo rileva un basso uso della CPU e per aumentare il grado di multiprogrammazione carica nuovi processo in memoria che allocano nuovi frame a discapito dei processi già in memoria. I processi fanno continui swap e l’utilizzo della CPU inizia a calare. Per risolvere questo problema, che porta la produttività del sistema al minimo, occorre diminuire il grado di multiprogrammazione. È possibile usare l’algoritmo di sostituzione locale o l’algoritmo di sostituzione per priorità. Però il problema non è completamente risolto.

Per evitare il verificarsi di queste situazioni, occorre fornire a un processo tutti i frame di cui necessita. Uno degli approcci più utilizzati è il working set che definisce il modello di località d’esecuzione del processo.

Una località è un insieme di pagine usate attivamente. Il modello stabilisce che un processo, durante la sua esecuzione, si sposta di località in località.

**9.6.2 Modello del working set**

Questo modello usa un parametro ∆ per definire la finestra del working set. L’idea consiste nell’esaminare i più recenti ∆ riferimenti alle pagine. L’insieme di pagine nei più recenti ∆ riferimenti è il working set. Se una pagina è in uso attivo si trova nel working set; se non è più usata esce dal working set. Se la somma delle dimensioni dei working set aumenta, superando il numero totale dei frame disponibili il sistema operativo individua un processo da sospendere. Scrive in memoria secondaria le pagine di quel processo e assegna i suoi frame ad altri processi.

**9.6.3 Frequenza dei page fault**

La strategia basata sulla frequenza dei page fault è più diretta rispetto al working set. Se la frequenza è eccessiva, significa che il processo necessita di più frame, se la frequenza fosse bassa il processo potrebbe disporre di troppi frame. SI può fissare un limite inferiore e superiore per la frequenza desiderata dei page fault. In questo modo se la frequenza supera il limite superiore significa che bisogna allocare un altro frame, mentre se la frequenza scende al di sotto del limite inferiore, si sottrae un frame a quel processo.

**Capitolo 10 Memoria secondaria**

**10.1 Struttura dei dispositivi di memorizzazione**

**10.1.1 Dischi magnetici**

I dischi magnetici sono il supporto fondamentale di memoria secondarie per gli elaboratori moderni. I dischi sono formati da: i piatti dei dischi hanno una forma piana e rotonda come quella dei CD, con un diametro che varia tra i 1.8 e i 3.5 pollici e le due superfici ricoperte di materiale magnetico; Le informazioni si memorizzano registrandole magneticamente. Una testina di lettura e scrittura è sospesa su ciascuna superficie d’ogni piatto. Le testine sono attaccate al braccio de disco che le muove in blocco. La superficie di un piatto è divisa in tracce circolari a loro volta suddivise in settori. La velocità di rotazione del disco viene espressa in termini di giri al minuto (RPM): i comuni dischi possono lavorare a 5400, 7200 o 15000 RPM. La velocità di un disco è caratterizzata da due valori:

1. la velocità di trasferimento
2. il tempo di posizionamento caratterizzato:
   1. tempo di ricerca (spostare il braccio del disco in corrispondenza del cilindro desiderato)
   2. latenza di rotazione (tempo impiegato affinché il settore si porti sotto la testina)

L’incidente della testina porta il disco in uno stato di danneggiamento permanente, accade quando la testina urta la superficie del disco.

**10.1.2 Dischi a stato solido**

Un SSD è una memoria non volatile che viene utilizzata come un disco rigido. Ci sono molte varianti di questa tecnologia, dalle DRAM dotate di batteria alle tecnologie di memoria flash come i chip SLC e MLC. Gli SSD hanno le stesse caratteristiche dei dischi rigidi, ma possono essere più affidabili e più veloci. Il costo al megabyte è maggiore e la loro vita è più breve

**10.1.3 Nastri magnetici**

Sono stati i primi supporti di memorizzazione secondaria. Erano caratterizzati da un tempo di accesso molto elevato. Il loro uso principale la creazione di copie di backup. Il nastro è avvolto in bobine e scorre su una testina di lettura e scrittura. Il posizionamento sul settore richiesto può richiedere diversi minuti. Una volta raggiunta la posizione giusta la velocità di scrittura e lettura è simile a quella dei dischi rigidi

**10.2 Struttura dei dischi**

I moderni dischi magnetici sono indirizzati come grandi array monodimensionali di blocchi logici, dove un blocco logico è la minima unità di trasferimento. LA dimensione di questo blocco logico di solito è di 512 byte. L’array monodimensionale di blocchi logici è mappato in modo sequenziale sui settori del disco: il settore 0 è il primo settore della prima traccia sul settore più esterno

**10.4 Scheduling dei dischi**

Ogni volta che si devono compiere operazioni di I/O con un’unità a disco un processo effettua una system call. La richiesta contiene diverse informazioni:

1. se l’operazione sia di immissione o di emissione di dati
2. l’indirizzo nel disco rispetto al quale eseguire il trasferimento
3. l’indirizzo di memoria rispetto al quale eseguire il trasferimento
4. il numero di byte da trasferire

La coda relativa ad un’unità a disco può essere spesso lunga e quindi Il sistema operativo dovrà scegliere quale fra le richieste inevase converrà eseguire prima

**10.4.1 FCFS**

First Come, First Served. Non garantisce la massima velocità di servizio.

**10.4.2 SSTF**

Algoritmo di servizio secondo il più breve tempo di ricerca. SSTF sceglie la richiesta che dà il minimo tempo di ricerca rispetto alla posizione corrente della testina

**10.4.3 SCAN**

Secondo l’algoritmo SCAN il braccio dell’unità a disco parte da un estremo del disco e si sposta verso l’altro estremo, servendo le richieste mentre attraversa o cilindri, finché non completa il tragitto, a questo punto il braccio inverte la marcia e la procedura continua. L’algoritmo viene chiamato anche algoritmo dell’ascensore.

**10.4.4 C-SCAN**

L’algoritmo SCAN circolare è una variante dello scheduling SCAN concepita per garantire un tempo di attesa meno variabile. Funziona esattamente come lo SCAN, tuttavia, quando la testina giunge all’altro estremo del disco, ritorna immediatamente all’inizio del disco, senza servire le richieste durante il viaggio di ritorno.

**10.4.5 LOOK e C-LOOK**

Le versioni dello SCAN e dello C-SCAN comunemente implementate fanno sì che il braccio si sposta solo finché ci sono altre richieste da servire in quella direzione, dopo di che cambia immediatamente direzione, senza giungere all’estremo del disco. Questo tipo di implementazione è chiamata LOOK e C-LOOK, perché guardano se ci sono altre richieste da soddisfare lungo la direzione attuale prima di continuare a spostare la testina in quella direzione.

**10.4.6 Scelta di un algoritmo di scheduling**

Per qualunque algoritmo di scheduling, le prestazioni dipendo comunque in larga misura dal numero e dal tipo di richieste. Le richieste di I/O per l’unità a disco possono essere notevolmente influenzate dal metodo adottato per l’allocazione dei file. Anche la posizione delle directory e dei blocchi indice è importante, perché ogni file deve essere aperto per essere usato, e visto che l’apertura di un file richiede una ricerca attraverso la struttura delle directory, vi saranno frequenti accessi.

**10.5 Gestione dell’unità a disco**

**10.5.1 Formattazione del disco**

Un disco magnetico nuovo è un piatto ricoperto di materiale magnetico; prima che possa memorizzare dati, deve essere diviso in settori che possano essere letti o scritti dal controllore. Questo processo si chiama formattazione di basso livello. La formattazione di basso livello riempie il disco con una speciale struttura dati per ogni settore, tipicamente consiste di una intestazione, un’area(512byte), e una coda. L’intestazione e la coda contengono informazioni per il controllore del disco. E un codice per la correzione degli errori (ECC). ECC è un codice per la correzione degli errori: se solo alcuni bit di dati sono stati alterati, esso contiene sufficienti informazioni affinché il controllore possa identificare i bit in questione e ricalcolare il loro valore coretto (soft error). Prima di poter memorizzare file il sistema operativo deve registrare le proprie strutture dati nel disco, ciò avviene in due passi. Il primo consiste nel creare delle partizioni, ovvero gruppi di uno o più cilindri. Il sistema operativo può trattare ogni partizione come se fosse un’unita a disco a sé stante. Il passo successivo è la formattazione logica, cioè la creazione di un file system. Per una maggiore efficienza, la maggior parte dei file system accorpa i blocchi in gruppi, detti cluster.

**10.5.2 Blocco d’avviamento**

Per far funzionare un sistema operativo occorre avere un programma d’avviamento (bootstrap). Esso inizializza il sistema in tutti i suoi aspetti, dai registri della CPU ai controllori dei dispositivi e al contenuto della memoria centrale, quindi avvia il sistema operativo. Per far ciò il programma d’avviamento trova il kernel del sistema operativo nei dischi, lo carica in memoria, e salta all’indirizzo iniziale per avviare l’esecuzione del sistema operativo. Il bootstrap è comunemente locato in una ROM. Essendo di sola lettura non è possibile modificare il contenuto della ROM quindi per cambiare il bootstrap occorre cambiare l’intera memoria e i circuiti da cui è composta, per questo molti sistemi memorizzano nella rom solo un piccolo caricatore d’’avviamento (bootstrap loader) il cui solo compito è quello di caricare il bootstrap completo dai dischi.

**10.5.3 Blocchi difettosi**

A volte è possibile avere dei malfunzionamenti nei dischi. Più frequentemente, uno o più settori divengono malfunzionanti (bad block). I dischi gestiti tramite controllore IDE i blocchi difettosi sono gestiti manualmente. Una possibile strategia consiste nell’effettuare la scansione del disco durante la formattazione, per rilevare la presenza di blocchi difettosi. Se si trova un blocco difettoso lo si marca come inutilizzabile al fine di segnalare alle procedure di allocazione del file system di non usarlo. Alcuni dischi hanno strategie più complesse. Il controllore mantiene una lista dei blocchi malfunzionanti dell’unità a disco ed è aggiornata per tutto il periodo in cui l’unità è operativa. La formattazione fisica mette da parte dei settori di riserva non visibili al sistema operativo. In questo modo è possibile sostituire dal punto di vista logico un settore danneggiato con uno di riserva. Questa tecnica è chiamata accantonamento dei settori. Un’altra alternativa è la traslazione dei settori. Un hard error porta alla perdita di dati.

**10.6 Gestione dell’area di avvicendamento**

La gestione dell’area di avvicendamento è un altro compito a basso livello del sistema operativo. La memoria virtuale usa lo spazio dei dischi come estensione della memoria centrale: poiché l’accesso alle unità a disco è molto più lento, l’uso dell’area di swapping riduce notevolmente le prestazioni del sistema.

**10.6.1 Uso Dell’area d’avvicendamento**

L’area di avvicendamento è usata in modi diversi da sistemi operativi diversi, in funzione degli algoritmi della gestione della memoria utilizzati. Lo spazio richiesto varia in base alla quantità della memoria fisica, della quantità della memoria virtuale e in modo in cui quest’ultima è usata.

**10.6.2 Collocazione dell’area d’avvicendamento**

Le possibili collocazione per un’are di swapping possono essere due:

1. all’interno del file system
2. partizione del disco a sé stante

Se si utilizza un file system è possibile utilizzare le varie tecniche messe a disposizione da quest’ultimo per crearla, ma porta a molti svantaggi, come la frammentazione esterna e l’aumento di tempo per attraversare, strutture e directory del file system.

Se si utilizza una partizione del disco non formattato si deve usare un gestore dell’are di swapping. In questo caso abbiamo problemi relativi alla frammentazione interna, ma questo non è molto importate dato che i dato che risiedono nell’area di swapping ci rimangono per poco tempo.

**10.7 Strutture Raid**

L’evoluzione tecnologica ha reso le unità a disco a disco progressivamente più piccole e meno costose, tanto che oggi è economicamente possibile equipaggiare un sistema elaborativo con molti dischi. La presenza di più dischi in parallelo rende possibile l’aumento della frequenza a cui i dati possono leggere op scrivere. Inoltre un configurazione di questo tipo permette migliore affidabilità sui dati. Ci sono varie tecniche note col nome di RAID (Redundant Array of Indipendent Disk).

**10.7.1 Miglioramento dell’affidabilità tramite ridondanza**

Nel causo di un guasto al disco le informazioni perse possono essere ricostruite grazie alla memorizzazione di informazioni non necessarie. Uno dei metodi è il mirroring (copiatura speculare): ogni disco logico consiste di due dischi fisici e ogni scrittura si effettua in entrambi i dischi. Quando uno dei duchi si guasta, i dati si possono leggere dall’altro. Se si verificano cali di alimentazioni durante la scrittura, i due dischi si troveranno in uno stato incoerente. Una soluzione prevede prima la scrittura su un disco e poi sul secondo disco. Un’altra soluzione è aggiungere una memoria non volatile a stato solido (NVRAM) alla batteria RAID, protetta dalla perdita di dati causa dalle cadute di alimentazione, se dotata di forme di correzione la scrittura dei dati nella cache può essere considerata sicura

**10.7.2 Miglioramento dell’affidabilità tramite il parallelismo**

Attraverso l’uso di più dischi è possibile anche migliorare la capacità dio trasferimento distribuendo i dati in sezioni su più dichi. La forma più semplice di distribuzione consiste nel distribuire i bit a ciascun byte su più dischi, in questo caso si parla di striping a livello dei bit. Un altro tipo di striping è lo striping a livello di blocchi dove i blocchi di un file si distribuiscono su più dischi. Lo striping porta ad avere l’aumento del throughput per accessi multipli a piccole porzione di dati e alla riduzione del tempo di risposta relativo ad avessi a grandi quantità di dati.

**10.7.3 Livelli RAID**

Esistono diversi schemi per realizzare diversi compromessi tra costi (mirroring) e prestazioni(striping) e sono stati classificati in livelli chiamati livelli RAID.

* **Raid di livello 0 🡪** striping a livello di blocchi, ma senza ridondanza
* **Raid di livello 1 🡪** utilizza la tecnica di mirroring
* **Raid di livello 2 🡪** noto come organizzazione con codici per la correzione di errori di memoria. Utilizzo del bit di parità. Se i bit con valore 1 sono in numero pari (parità = 0) in caso contrario ovvero in numero dispari (parità = 1).
* **Raid di livello 3 🡪** noto come organizzazione con bit di parità intercalati. Considerando che i controllori di dischi possono verificare se un settore è stato letto correttamente, si utilizza un singolo bit di parità per individuare errori e per correggere eventuali errori
* **Raid di livello 4 🡪**  noto come organizzazioni con blocchi di parità intercalati, s’impiega lo striping a livello di blocchi e si utilizzi un blocco di parità in un disco separato
* **Raid di livello 5 🡪** organizzazione con blocchi intercalati a parità distribuita, i dati e le informazioni di parità sono distribuite tra gli n + 1 dischi
* **Raid di livello 6 🡪** schema di ridondanza P + Q simile al livello RAID 5, ma memorizza ulteriori informazioni ridondanti per gestire guasti contemporanei di più dischi
* **Raid di livello 0 + 1 e 1 + 0 🡪**  consiste in una combinazione dei livelli RAID 0 e RAID 1. Il RAID 0 fornisce le prestazioni mentre il livello 1 l’affidabilità

Un sistema a RAID può essere implementato a livello software o a livello hardware. Con una implementazione software, il sistema operativo gestisce l'insieme di dischi attraverso un normale controllore dei dischi. Questa opzione può essere più lenta di un RAID hardware, ma non richiede l'acquisto di componenti extra (anche questi, in ogni caso, soggetti a difetti). Una implementazione hardware del RAID richiede (almeno) un controllore RAID ad hoc. Nei computer desktop, questo può essere una scheda di espansione PCI o può essere usato il controller presente nella scheda madre. Nei RAID più grandi, il controller e i dischi sono sistemati in un alloggiamento esterno. Questa opzione tende a dare le migliori prestazioni e a rendere la gestione nel sistema operativo molto più semplice. Le implementazioni hardware in genere supportano lo scambio a caldo (hot swapping), permettendo di sostituire un disco mentre il sistema rimane in esecuzione.

**10.7.4 Scelta di un livello RAID**

Uno dei criteri di scelta di uno dei tanti livelli RAID potrebbe essere il tempo di ricostruzione, se un disco si guasta, il tempo necessario a ricostruire i dati che contiene può essere rilevante. Questa criterio è importante quando il flusso dei dati è continuo. Il RAID di livello 0 si usa nelle applicazioni ad alte prestazioni in cui le perdite di dati non sono critiche mentre il RAID di livello 1 si usa nelle applicazioni che richiedono un’alta affidabilità ed un rapido ripristino.  
I RAID 0+1 e 1+0 si usano dove entrambe prestazioni ed affidabilità sono importanti, per esempio per piccole basi di dati. Per la memorizzazione di grandi quantità di dati, in genere si preferisce impiegare il RAID di livello 5, non essendo il livello 6 sempre disponibile.

**10.7.5 Estensioni**

I concetti relativi ai sistemi RAID sono stati generalizzati ad altri dispositivi di memorizzazione, comprese le unità a nastri e anche alla diffusione dei dati tramite sistemi senza fili.

**10.7.6 Problemi connessi a RAID**

I sistemi RAID non assicurano sempre la disponibilità dei dati al sistema operativo e agli utenti. Per esempio, un puntatore a file potrebbe essere errato. Per risolvere alcuni problemi si utilizza la cosiddetta somma di controllo (checksum) interna a ogni blocco, dati e metadati inclusi. Se si verifica un problema con i dati, la chesksum darà un valore errato e il file system verrà a conoscenza del problema. Se attivo il mirroring il blocco con checksum errato può essere sostituito. Un altro problema nella maggior parte delle implementazioni riguarda la mancanza di flessibilità. Per risolvere i dischi, o le partizioni dei dischi, sono riuniti in gruppi di memorizzazione attraverso insiemi RAID. Un gruppo può contenere uno o più file system. Tutta l’area libera di un pool è a disposizione di tutti i file system contenuti all’interno di quel pool. Per allocare o rilasciare la memoria si può utilizzare un sistema di gestione basato su calloc(), malloc() e free(). In questo modo non ci sono limiti all’utilizzo della memoria e non sussiste la necessità di ridistribuire i file system tra i volumi né di ridimensionare i volumi.

**Capitolo 11**

**Interfaccia del file system**

**11.1 Concetto di file**

Un file è un insieme di informazioni correlate, registrate in memoria secondaria, cui è assegnato un nome. I file di dati possono essere: alfabetici, numerici, alfanumerici o binari. Un file puro di solito rappresenta un programma in forma sorgente e oggetto. Il concetto di file è estremamente generale. Un file ha una struttura definita secondo il tipo: un file di testo è formato da una sequenza di caratteri organizzati in righe, ed eventualmente pagine. Un file sorgente è formato da una sequenza di funzioni, un file eseguibile consiste in una serie di sezioni di codice che può caricare in memoria ed eseguire.

**11.1.1 Attributi dei file**

Un file ha attributi che possono variare secondo il sistema operativo, ma che tipicamente comprendono i seguenti:

1. **Nome** 🡪 unica informazione tenuta in una forma leggibile dagli utenti
2. **Identificatore** 🡪 etichetta unica, in genere numerica, che identifica il file all’interno del file system: il nome utilizzato dal sistema per il file
3. **Tipo** 🡪 necessario per sistemi che supportano tipi differenti.
4. **Locazione** 🡪 puntatore alla locazione del file sul dispositivo
5. **Dimensione** 🡪 dimensione attuale del file, generalmente espresso in byte
6. **Protezione** 🡪 informazioni di controllo: chi può leggere, scrivere o eseguire
7. **Ora, data e identificazione dell’utente** 🡪 creazione, modifica ed ultimo uso, dati utili alla protezione e per monitorare l’utilizzo

Alcuni sistemi operativi moderni offrono ancora più informazioni come per esempio il checksum.

**11.1.2 Operazioni sui file**

Un file è tipo di dato astratto. Il sistema operativo può offrire chiamate di sistema per creare, scrivere, leggere, spostare, cancellare e troncare un file.

1. **Creazione di un file** 🡪 per creare un file è necessario compiere due passaggi. In primo luogo si deve trovare lo spazio per il file nel file system e successivamente allocare il file, inoltre bisogna creare un nuovo elemento nella directory.
2. **Scrittura di un file** 🡪 per scrivere un file viene effettuata una chiamata di sistema che specifica il nome del file e le informazioni che si vogliono scrivere. Il file system deve mantenere un puntatore di scrittura alla locazione nel file in cui deve avvenire l’operazione di scrittura successiva. Il puntatore si aggiorna ad ogni nuova scrittura
3. **Lettura di un file** 🡪 per leggere un file viene effettuata una chiamata di sistema che specifica il nome e la posizione nella memoria dove leggere. Il sistema cerca la sua positone nella directory, mantenendo un puntatore alla posizione del file in cui deve avvenire la prossima lettura.
4. **Riposizionamento in un** **file** 🡪 Si ricerca l’elemento appropriato nella directory e si assegna un nuovo valore al puntatore alla posizione corrente nel file. Questa operazione è anche nota come posizionamento o ricerca (seek) nel file
5. **Cancellazione di un file 🡪** per cancellare un file si cerca l’elemento della directory associato al file designato, si rilascia lo spazio associato al file e si elimina l’elemento della directory
6. **Troncamento dei un file 🡪** Consente di mantenere immutati gli altri attributi del file, pur azzerando la lunghezza del file e rilasciando lo spazio occupato
7. **Altre operazioni 🡪** rename e append

Il sistema operativo mantiene una tabella contenente informazioni riguardanti tutti i file aperti. Quando si richiede un’operazione su un file, questo viene individuato tramite un indice in tale tabella. Quando il file non è più attivamente usato viene chiuso dal processo, e il sistema operativo rimuove l’elemento a esso associata dalla tabella dei file aperti (chiamate di sistema open() e close()). Il sistema operativo introduce due livelli di tabelle interne: una **tabella per ciascun processo** e una **tabella di sistema**. La tabella del processo contiene i riferimenti a tutti i file aperti dal processo. Ciascun elemento di questa tabella punta a sua volta a una tabella di sistema dei file aperti. In generale la tabella dei file aperti ha anche un contatore delle aperture associato a ciascun file, indicante il numero di processi che hanno aperto quel file

Un file aperto ha le seguenti informazioni:

1. **Puntatore al file** 🡪 ultima posizione di lettura o scrittura sotto forma di puntatore alla posizione corrente nel file
2. **Contatore dei file aperti** 🡪 tiene traccia del numero di open e close, raggiunto il valore 0 il sistema rimuove l’elemento dalla tabella
3. **Posizione nel disco del file** 🡪 informazione per localizzare il file nel disco, viene mantenuta in memoria
4. **Diritti di accesso** 🡪 Ciascun processo apre un file in uno dei mille mila modi di accesso, questa informazione viene mantenuta nella tabella del processo

**11.1.3 Tipi di file**

Un sistema operativo che riconosce il tipo di un file ha la possibilità di trattare il file in modo ragionevole. Una tecnica comune per realizzare la gestione dei tipi di file consiste nell’includere il tipo nel nome del file. Il nome è suddiviso in due parti, una nome e un’estensione, di solito separate da un punto. Il sistema usa l’estensione per stabilire il tipo del file e le operazioni che si possono eseguire su tale file. Il sistema UNIX non fornisce una funzione di questo tipo, ma si limita a memorizzare un semplice codice (noto come magic number) all’inizio di alcuni tipi di file allo scopo di indicarne il tipo.

**11.1.4 Struttura dei file**

Alcuni file devono rispettare una determinata struttura comprensibile al sistema operativo. Alcuni sistemi operativi impongono un numero minimo di strutture di file. Questo orientamento è stato seguito da UNIX, Windows e altri. UNIX considera ciascun file come una sequenza di byte, senza alcuna interpretazione da parte del sistema operativo. Qualsiasi programma applicativo deve contenere il proprio codice per interpretare in modo appropriato la struttura di un file in ingresso.

**11.1.5 Struttura interna dei file**

Per il sistema operativo la localizzazione di un offset all’interno di una file può essere complicata. I dischi hanno una dimensione dei blocchi ben definita, determinata dalla dimensione di un settore. È improbabile che la dimensione del record fisico corrisponda esattamente alla lunghezza del record logico desiderato. Una soluzione consiste nell’impaccamento di un certo numero di record logici in blocchi fisici. Il sistema operativo UNIX definisce tutti i file semplicemente come un flusso di byte. A ciascun byte si può accedere in modo individuale tramite il suo offeset a partire dall’inizio o dalla fine del file. A causa della gestione in multipli di blocchi invece che di byte, porta ad avere dei byte sprecati e ciò costituisce della frammentazione interna che affligge tutti i file system.

**11.2 Metodi d’accesso**

I file memorizzano informazioni; al momento dell’uso è necessario accedere a queste informazioni e trasferirli in memoria.

**11.2.1 Accesso sequenziale**

Le informazioni del file si elaborano ordinatamente, un record dopo, l’altro.  
Le più comuni operazioni sono:

1. read\_next() 🡪 legge la prossima porzione di file e fa avanzare automaticamente il puntatore
2. write\_next() 🡪 fa un’aggiunta in coda al file e avanza fino alla fine delle informazioni appena scritte, che costituisce la nuova fine del file

**11.2.2 Accesso diretto**

In questo caso, un file è formato da elementi logici(record) di lunghezza fissa. Questo metodo si basa sul disco, infatti i dischi permettono l’accesso diretto a ogni blocco di file. Il file si considera come una sequenza numerata di blocchi. Non esistono restrizioni all’ordine di lettura o scrittura di un file ad accesso diretto.

**11.2.3 Altri metodi d’accesso**

Sulla base del metodo di accesso diretto se ne possono costruire altri, che generalmente si basano sulla costruzione di un indice per il file. L’indice contiene puntatori ai vari blocchi, per trovare un elemento del file occorre prima cercare nell’indice e quindi usare i puntatore per accedere direttamente al file e trovare l’elemento desiderato

**11.3 Struttura della directory e del disco**

Ogni entità contenente un file system è generalmente nota come volume. Ogni volume contenente un file system deve anche avere in sé delle informazioni sui file presenti nel sistema. Tali informazioni risiedono in una **directory** del dispositivo o indice del volume.

**11.3.1 Struttura della memorizzazione di massa**

Il numero di file system in un sistema informatico può variare da zero a molti. Alcuni esempi:

1. tmpfs
2. objfs
3. ctfs
4. lofs

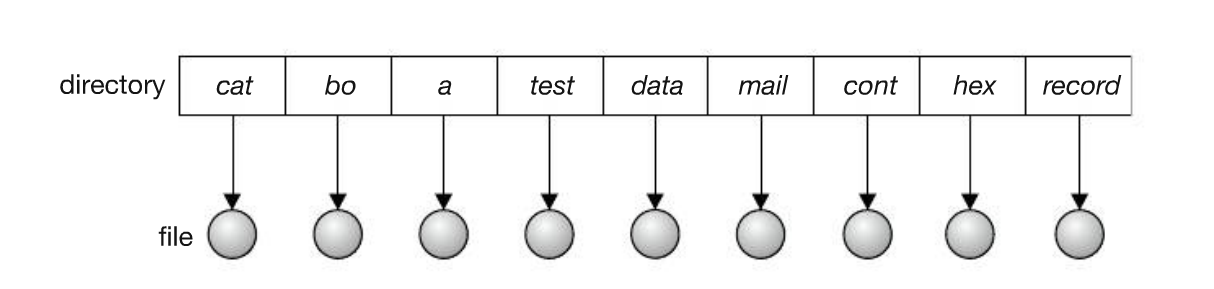
**11.3.2 Aspetti generali delle directory**

La directory si può considerare come una tabella di simboli che traduce i nomi dei file negli elementi in essa contenuti. L’organizzazione deve rendere possibile l’inserimenti di nuovi elementi, la cancellazione di elementi esistenti, la ricerca di un elemento e l’elenco di tutti gli elementi della directory:

1. **Ricerca di un file** 🡪 scorrere una directory per individuare l’elemento associato a un particolare file
2. **Creazione di un file**
3. **Cancellazione di un file**
4. **Elencazione di una directory** 🡪 elencare tutti i file contenuti in una directory
5. **Ridenominazione di un file** 🡪 il nome deve poter essere modificato
6. **Attraversamento del file system** 🡪 salvataggio del contenuto e della struttura del file system su disco

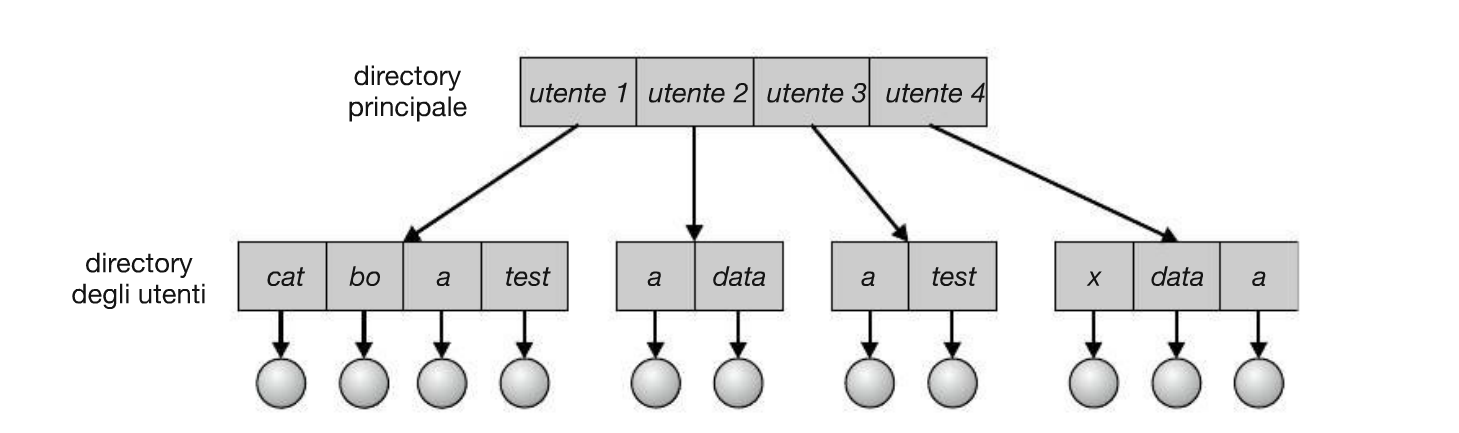
**11.3.3 Directory a un livello**

Tuti i file sono contenuti nella stessa directory, facilmente gestibile e comprensibile. All’aumento del numero di file, questo tipo di schema presenta dei limiti. **Ogni file deve avere un nome unico.**



**11.3.4 Directory a due livelli**

Nella struttura a due livelli, ogni utente dispone della proprio directory utente (UFD userfiledirectory). Tutte le directory utente hanno una struttura simile, ma in ciascuna sono elencati solo i file del proprietario. Quando comincia l’elaborazione di un lavoro dell’utente, si fa prima una ricerca nella directory principale (master file directory, MFD) del sistema. Quando un utente fa riferimento a un file particolare, il sistema operativo esegue la ricerca solo nella directory di quell’utente. Per creare un file per un utente, il sistema operativo controlla che non ci sia un altro file con lo stesso nome soltanto nella directory di tale utente. Per cancellare un file il sistema operativo limita la propria ricerca alla directory utente locale. Le stesse directory utente devono essere cancellare e create. Per far ciò il sistema operativo esegue uno speciale programma di sistema. Questo schema **risolve il problema delle collisioni dei nomi**, **limita gli utenti che vogliono cooperare tra di loro**. Alcuni sistemi permettono, previa autorizzazione, di accedere ai file di altri utenti.



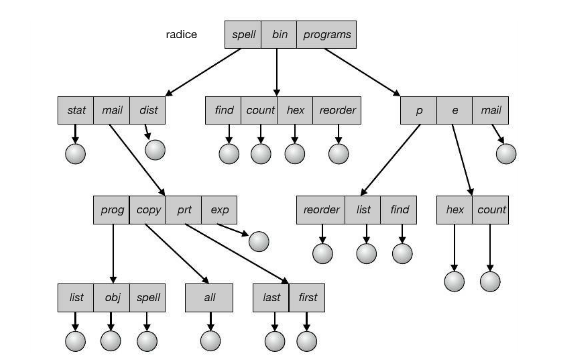
**11.3.5 Directory con struttura ad albero**

Questo tipo di directory permette agli utenti di creare proprie sottodirectory e di organizzare i file di conseguenza. L’albero ha una directory radice e ogni file del sistema ha un unico nome di percorso. **La distinzione tra file e directory è data da un bit di ogni elemento della directory**. Per creare o cancellare directory si usano speciali chiamate di sistema. La directory corrente dovrebbe contenere la maggior parte dei file di interesse corrente per il processo. Quando si fa riferimento a un file, si esegue una ricerca nella directory corrente, se il file non si trova in tale directory, l’utente deve specificare un nome di percorso oppure cambiare directory corrente. Per cambiare directory corrente si usa la chiamata di sistema change\_directory() (cd). I nomi di percorso possono essere di due tipi:

1. **nomi di percorso assoluti** 🡪 comincia dalla radice dell’albero di directory e segue un percorso che lo porta fino al file specificato indicando i nomi delle directory lungo il percorso
2. **nomi di percorso relativo** 🡪 definisce un percorso che parte dalla directory corrente

Una decisione importante in questo schema riguarda la cancellazione di una directory. Se una directory è vuota è sufficiente cancellare l’elemento che la designa nella directory corrente. Tuttavia se la directory da cancellare contiene file o sottodirectory si può procedere in due modi.

1. alcuni sistemi non permettono di cancellare una directory se questa non è vuota, questa regola si applica anche per le sottodirectory
2. Usando una chiamata di sistema (rg per UNIX) la directory con tutto il suo contenuto viene cancellata, questo metodo è più pericoloso perché rimuove l’intera struttura.

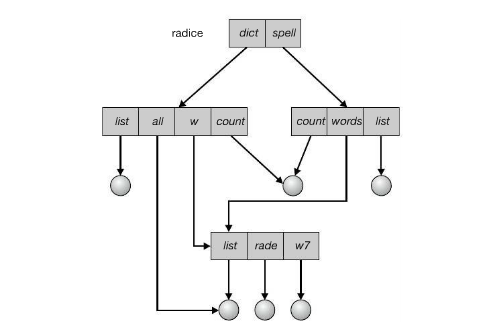
L’accesso ai file da parte di altri utenti è più semplice con una struttura ad albero.

**11.3.6 Directory con struttura a grafo aciclico**

Un grafo aciclico permette alle directory di avere sottodirectory e file condivisi. Lo stesso file o la stessa sottodirectory possono essere in due directory diverse.

I file e le sottodirectory condivisi si possono realizzare in molto modi. Un metodo diffuso prevede la creazione di un nuovo elemento di directory, chiamato **collegamento**, un puntatore a un altro file o un’altra directory.

Questo tipo di struttura è molto più flessibile ma anche più complessa. Ci sono molti problemi:

* Nomi diversi si possono riferire allo stesso file, creando il problema dell’alias. 
* Un altro problema riguarda la cancellazione.

Una possibilità prevede che ad ogni operazione di cancellazione segua l’immediata rimozione del file. Questo però può lasciare puntatori sospesi a un file che ormai non esiste più. Problema ancora più grave, se i puntatori contengono indirizzi effettivi del disco e lo spazio viene poi riutilizzato per altri file, i puntatori potrebbero puntare nel mezzi di questi altri file.

**11.3.7 Directory con struttura a grafo generale**

Un serio problema connesso all’uso di una struttura a grafo aciclico consiste nell’assicurare che non vi siano cicli. Il vantaggio principale di un grafo aciclico è dato dalla sua semplicità degli algoritmi necessari per attraversarlo e per determinare quando non ci siano più riferimenti a un file. Se si permette che nella directory esistano cicli, è preferibile evitare una duplice ricerca di un elemento, per motivi di correttezza e di prestazioni. Un algoritmo mal progettato potrebbe causare un ciclo infinito di ricerca. Una soluzione è quella di limitare arbitrariamente il numero di directory cui accedere durante una ricerca. Per la cancellazione è preferibile adottare un metodo di “ripulitura” (garbage collection) per stabilire quando sia stato cancellato l’ultimi riferimento e quando sia possibile riallocare lo spazio dei dischi. Infine ogni volta che viene aggiunto un collegamento bisogna usare un algoritmo che individui la presenza di cicli, se viene trovato un ciclo, l’operazione di creazione del collegamento è vietata.

**11.4 Montaggio di un file system**

Un file system deve essere montato prima di poterlo utilizzare. In particolare deve essere costruita la struttura directory, composta da volumi, che devono essere montati affinché siano disponibili nello spazio dei nomi del file system. Si fornisce al sistema il nome del dispositivo e la sua locazione (detta punto di montaggio). Nella struttura di file e directory alla quale agganciare il file system. Il passo successivo consiste nella verifica da parte del sistema operativo della validità del file system. Infine il sistema operativo annota nella sua struttura della directory che un certo file system è montato al punto di montaggio specificato

**11.5 Condivisione di un file**

**11.5.1 Utenti multipli**

Se un sistema operativo permette l’uso del sistema da parte di più utenti, diventano particolarmente rilevanti i problemi relativi alla condivisione dei file, alla loro identificazione tramite nomi e la loro protezione. La maggior parte dei sistemi adottano i concetti di proprietario, **utente** e gruppo. Il **proprietario** è l’utente che può cambiare gli attributi, concedere l’accesso e che ha maggior controllo sul file. Il **gruppo** indica un sottoinsieme di utenti abilitati ad utilizzare quel file. Gli identificatori (ID) del gruppo e del proprietario sono memorizzati insieme agli attributi del file

**11.5.2 File system remoti**

I metodi con i quali i file si condividono in una rete sono cambiati di molto:

* Il primo metodo utilizzato consiste nel trasferimento del file richiesto attraverso programmi come **l’ftp**.
* Un secondo metodo è quello dei **file system distribuito**, che permette la visibilità nel calcolatore locale delle directory remote.
* Il terzo metodo è il **WWW**, si utilizza un browser per accedere ai file remoti e il metodo ftp per il trasferimento. Un ultimo metodo diffuso negli ultimi anni è il cloud computing

**11.5.2.1 Modello client-server**

I file system remoti permettono il montaggio di uno o più file system file system di uno o più calcolatori remoti in un calcolatore locale. Il calcolatore contenete i file si chiama **server**, mentre il calcolatore che richiede l’accesso ai file si chiama **client**. Un server può gestire richieste provenienti da più client, e un client può accedere a più server. Le chiave di cifratura sono un ottimo modo per isolare un client che non ha le autorizzazioni adatte

**11.5.2.2 Sistemi di informazioni distribuiti**

Per semplificare il concetto di servizio client server, i sistemi di informazione distribuiti, sono stati concepiti per fornire una accesso unificato alle informazioni necessarie pe il calcolo remoto. Il sistema dei nomi di dominio (DNS) fornisce le traduzioni dai nomi dei calcolatori agli indirizzi di rete per l’intera Internet.

**11.5.2.3 Tipi di malfunzionamento**

I file system locali possono presentare malfunzionamenti per varie cause. Molti di questi malfunzionamenti portano al crash del sistema, al emissione di una condizione d’errore e alla necessità di un intervento umano per risolvere il problema. I file system remoti implicano ancor più problemi, a causa della complessità dei sistemi di rete e dalla necessitò di interazioni tra calcolatori remoti.

**11.5.3 Semantica della coerenza**

La semantica di coerenza è un’importante criterio di valutazione di qualsiasi file system che consente la condivisione dei file, specifica il modo in cui più utenti devono accedere contemporaneamente a un file condiviso, è realizzata come parte del codice del file system.

**11.5.3.1 Semantica UNIX**

* le scritture in un file aperto da parte di un utente sono **immediatamente visibili ad altri** utenti che hanno lo stesso file contemporaneamente aperto
* Un metodo di condivisione permette agli utenti di condividere il **puntatore alla locazione corrente nel file**

Un file è associato a **una singola immagine fisica**, accessibile come una risorsa esclusiva

**11.5.3.2 Semantica delle sessioni**

* le scritture in un file aperto da parte di un utente **non sono immediatamente visibili** ad altri utenti che hanno lo stesso file contemporaneamente aperto
* Una volta chiuso il file, le **modifiche apportate sono visibili solo nelle sessioni che iniziano successivamente.**

Un file può essere contemporaneamente **associato a parecchie immagini**, anche diverse

**11.5.3.3 Semantica dei file condivisi immutabili**

Un file dichiarato condiviso dal suo creatore non può subire modifiche:

1. Il suo nome non può essere modificato
2. Il suo contenuto non può essere modificato

Un file con queste caratteristiche è chiamato file condiviso immutabile

**11.6 Protezione**

Le informazioni contenute in un sistema elaborativo devono essere protette dai danni fisici e da accessi impropri. Generalmente l’affidabilità è garantita dalle mille mila copie di backup. La protezione si può ottenere in molti modi. Per esempio potete mettere il portatile in un comodino.

**11.6.1 Tipi d’accesso**

Il controllo offerto dai meccanismi di protezione si ottiene limitando i possibili tipi di accesso. Gli accessi si permettono o si negano secondo diversi fattori. Si possono controllare diverse operazioni:

1. **Lettura**
2. **Scrittura**
3. **Esecuzione**
4. **Aggiunta**
5. **Cancellazione**
6. **Elencazione**

Sono stati proposti molti tipi di protezione, ognuno ovviamente porta con sé vantaggi e svantaggi

**11.6.2 Controllo degli accessi**

L’approccio più comune al problema della protezione è rendere l’accesso dipendente dall’identità dell’utente. Lo schema più generale consiste nell’associare una lista di controllo degli accessi a ogni directory: in tale lista sono specificati i nomi degli utenti e i relativi tipi di accesso consentiti. Questa tecnica comporta due inconvenienti:

1. La **costruzione di una lista di questo tipo può essere un compito noioso e non gratificante**, soprattutto se la lista degli utenti del sistema non è nota a priori
2. L’elemento della directory, precedentemente di dimensione fissa, deve essere di **dimensione variabile**, quindi anche la gestione dello spazio è più complicata

Per diminuire la lunghezza della lista molti sistemi raggruppano gli utenti di ogni file in tre classi distinte:

1. **Proprietario** 🡪 utente che ha creato il file
2. **Gruppo** 🡪 un gruppo di utenti che condivide il file
3. **Universo** 🡪 tutti gli altri utenti del sistema

**11.6.3 Altri metodi di protezione**

Un altro metodo di protezione consiste nell’associazione di una password a ciascun file. Questo metodo presenta svantaggi: può diventare impraticabile perché le password da ricordare aumentano con l’aumentare dei file, usando la stessa password la protezione non ha senso.

**Capitolo 12**

**Realizzazione del file system**

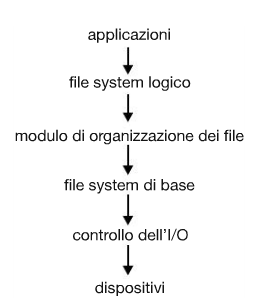
**12.1 Struttura del file system**

I dischi hanno due principali caratteristiche che li rendono adatti al ruolo di memorizzazione del file system:

1. Si possono riscrivere localmente; si può leggere un blocco dal disco, modificarlo e quindi scriverlo nella stessa posizione
2. È possibile accedere direttamente a qualsiasi blocco di informazioni del disco

I trasferimenti tra memoria centrale e dischi si eseguono per blocchi, Ciascun blocco è composto da uno o più settori, la dimensione di solito è di 512 byte. I file system presentono due problemi di progettazioni:

1. **Definizione dell’aspetto agli occhi dell’utente**
2. **Creazione di algoritmi e strutture dati.**

Un file system è generalmente composto da molti livelli distinti. Ogni livello si serve delle funzioni dei livelli inferiori.

* **Controllo dell’I/O**, costituito dai driver dei dispositivi e dai gestori dei segnali d’interruzione, si occupa del trasferimento delle informazioni tra memoria centrale e memoria secondaria.
* **File system di base** deve soltanto inviare dei generici comandi all’appropriato driver di dispositivo per leggere e scrivere blocchi fisici nel disco.
* **Modulo di organizzazione dei file** è a conoscenza dei file e dei loro blocchi logici, così come dei blocchi fisici dei dischi. Questo strato contiene anche il gestore dello spazio libero che tiene traccia anche dello spazio libero in memoria, registrando i blocchi non assegnati.
* **File system logico**, gestiste tutte le strutture del file system, come la struttura delle directory. Mantiene le strutture di file tramite blocchi di controllo dei file (FCB- File Control Block), detti inode nel file system di UNIX.

**12.2 Realizzazione del file system**

I sistemi operativi permettono l’uso delle chiamate di sistema open() e close() per permettere ai processi di accedere a file.

**12.2.1 Introduzione**

Per realizzare un file system si usano parecchie strutture dati, sia nei dischi sia in memoria. Variano a seconda del sistema operativo scelto, ma esistono dei principi generali.

Fra le strutture presenti nei dischi ci sono le seguenti:

1. **Blocco di controllo dell’avviamento**
2. **Blocco di controllo del volume**
3. **Struttura della directory**
4. **Blocco di controllo del file**

Le informazioni tenute in memoria servono sia per la gestione del file system sia per migliorare le prestazioni attraverso l’uso di cache. Le strutture in memoria possono essere:

1. **Tabella di montaggio**
2. **Cache della struttura directory**
3. **Tabella di sistema dei file aperti**
4. **Tabella dei file aperti per ciascun processo**
5. **Buffer per conservazioni di blocchi del file system**

Per la creazione di un nuovo file le applicazioni, eseguono una chiamata al file system logico, il quale conosce il formato della struttura della directory. Per creare un nuovo file, esso crea un nuovo FCB. Il sistema carica quindi la directory appropriata in memoria, la aggiorna con il nome del nuovo file e con l’FCB associato, e la scrive nuovamente sul disco.

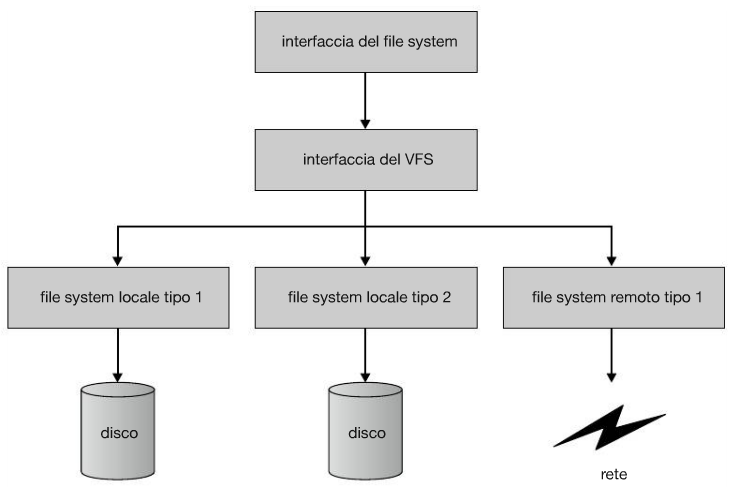
**12.2.2 Partizioni e montaggio**

Una partizione può essere priva di una struttura logica oppure può contenere un file system.Si usa un disco privo di struttura logica se nessun file system è appropriato all’utilizzo che se ne vuole fare, per esempio come area di swapping. La partizione di avviamento consiste in una serie sequenziale di blocchi che vengono caricati in memoria prendendo il nome d’immagine. Il boot loader ha invece abbastanza informazioni sul file system da essere in grado di caricare il kernel e avviarne l’esecuzione. Nella fase di caricamento del sistema operativo, si esegue il montaggio della partizione radice, che contiene il kernel del sistema operativo e in alcuni casi altri file di sistema. Durante il montaggio il sistema operativo controlla che il dispositivo contenga un file system valido. Infine il sistema annota nella tabella di montaggio in memoria che un file system è stato montato.

**12.2.3 File system virtuali**

I sistemi operativi moderni devono gestire contemporaneamente più tipi di file system. Un metodo ovvio ma schifoso è scrivere intere procedure di gestione file e directory differenti per ciascun file system, molti sistemi utilizzano tecniche orientate agli oggetti per organizzare e rendere modulare la soluzione. **In questo modo la realizzazione del file system si articola in tre strati principali:**

1. **Interfaccia del file system**, basata sulle chiamate di sistema open(), read(), write() e close() e sui descrittori di file
2. **File system virtuale (VFS)** è composto da due funzioni importanti:
   1. **Separa le operazioni generiche del file system dalla loro realizzazione definendo un’Interfaccia VFS uniforme.** Possono esistere più interfaccia VFS
   2. **Permette la rappresentazione univoca di un file su tutta la rete.** IL VFS è basato su una struttura di rappresentazione dei file detta vnode che contiene un indicatore numerico unico per tutta la rete per ciascun file.

Il VFS a seconda del file system, attiva le operazioni specifiche di un file system per gestire le richieste locali. Lo strato che realizza il codice di uno specifico file system è il terzo dell’architettura.

**12.3 Realizzazione delle directory**

**12.3.1 Lista lineare**

Il più semplice metodo di realizzazione di una directory è basato sull’uso di **una lista lineare contenete i nomi dei file con puntatori ai blocchi di dati.** È facilmente programmabile ma la sua esecuzione onerosa in termini di tempo. Per creare un nuovo file occorre prima esaminare la directory per essere sicuri che non esista già un file con tale nome, quindi aggiungere un nuovo elemento alla fine della directory. Per cancellare un file occorre cercare nella directory il file con quel nome, quindi rilasciare lo spazio che gli era stato assegnato. Il vero svantaggio è dato proprio dalla ricerca lineare del file

**12.3.2 Tabella hash**

In questo caso una lista lineare contiene gli elementi di directory, ma **si usa una funzione hash. La tabella hash riceve un valore calcolato a partire dal nome del file e riporta un puntatore al nome del file nella lista lineare**. Occorre prendere provvedimenti per quanto riguarda le collisioni. La maggior difficoltà è legata alla sua dimensione, che in genere è fissa, e la dipendenza della funzione hash da tale dimensione.

**12.4 Metodi di allocazione**

La natura ad accesso diretto dei dischi dà flessibilità nella realizzazione dei file Il problema principale consiste nell’allocare spazio per questi file in modo che lo spazio nel disco sia usato efficientemente e l’accesso sia rapido.

**12.4.1 Allocazione contigua**

**Ogni file deve occupare un insieme di blocchi contigui del disco**. Gli indirizzi del disco definiscono un ordinamento lineare nel disco stesso. Il numero di posizionamenti richiesti per accedere a file in cui spazio è allocato in modo contiguo è minimi. Quando si una un accesso sequenziale, il file system memorizza l’indirizzo dell’ultimo blocco cui è stato fatto riferimento e, se necessario, legge il blocco successivo. Nel caso di un accesso diretto al blocco i di un file che comincia in un blocco b si può accedere direttamente al blocco b + i. **L’allocazione contigua presenta alcuni problemi. Una difficoltà riguarda l’individuazione dello spazio per un nuovo file.** La realizzazione del sistema di gestione dello spazio libero determina il modo in cui tale compito viene eseguito. **Un altro problema è collegato all’allocazione dinamica della memoria.** Gli algoritmi di **first-fit** e **best-fit** presenti anche in questo caso, soffrono della **frammentazione esterna**: lo spazio col tempo diventa formato da piccoli buchi nessuno in grado di contenere i dati Una strategia prevede di copiare l’intero file system su un secondo disco, in questo modo si crea un enorme spazio contiguo, successivamente si copiano nuovamente i file nel primo disco. Questa tecnica di compattazione risolve il problema della frammentazione esterna. Per eseguire tale operazione alcuni sistemi richiedono che il file system non sia in linea ovvero non sia montato, questo causa l’impossibilità nell’uso del sistema.

**12.4.2 Allocazione concatenata**

Con questo tipo di allocazione si risolvono tutti i problemi dell’allocazione contigua. **Ogni file p composto da una lista concatenata di blocchi del disco i quali possono essere sparsi in qualsiasi punto del disco stesso**. La directory contiene il puntatore al primo blocco e all’ultimo blocco del file. Per creare un nuovo file si crea un nuovo elemento nella directory, essendo un puntatore si inizializza a null, un’operazione di scrittura provoca una serie di operazioni: si cerca un blocco libero attraverso il sistema di gestione dello spazio libero, si scrive in tale blocco e la concatenazione di tale blocco alla fine del file. **Per leggere basta leggere i blocchi seguendo i puntatori da un blocco all’altro.** Uno degli svantaggi è lo spazio richiesto per i puntatori, **ogni file richiede un po' più di spazio di quanto ne richiederebbe**. La soluzione più comune a questo problema consiste nel **riunire un certo numero di blocchi contigui in cluster e nell’allocare i cluster anziché i blocchi**. In questo modo i puntatori utilizzano una percentuale molto piccola dello spazio del disco. Questa allocazione trae il massimo beneficio dall’accesso sequenziale perché ogni accesso implica una lettura del disco e talvolta un posizionamento della testina di conseguenza l’accesso diretto è inefficiente. **Un altro problema riguarda l’affidabilità, se uno dei puntatori andasse perduto o danneggiato non si potrebbe più accedere al file**. Una variante del metodo di allocazione concatenata consiste nell’uso della tabella di allocazione dei file (FAT). Per contenere tale tabella si riserva una sezione del disco all’inizio di ciascun volume. **La FAT ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco, si usa come una lista concatenata.** L’elemento di directory contiene il numero di blocco del primo blocco del file. L’elemento della tabella indicizzato da quel numero di blocco contiene a sua volta il numero di blocco del blocco successivo del file. Questa catena continua fino all’ultimo blocco, che ha come elemento della tabella un carattere speciale di fine file. I blocchi inutilizzati sono indicati da un valore 0 nella tabella.

**12.4.3 Allocazione indicizzata**

L’allocazione indicizzata risolve il problema dei puntatori sparsi, **raggruppando tutti i puntatori in una sola locazione: il blocco indice.** Si tratta di un array d’indirizzi di blocchi del disco. Simile alla Paginazione. L’allocazione indicizzata soffre di un overhead: lo spazio aggiuntivo richiesto dai puntatori del blocco indice è generalmente maggiore dello spazio aggiuntivo necessario per l’allocazione concatenata. È necessario disporre di un meccanismo per gestire questa situazione:

1. **Schema concatenato 🡪**  Un blocco indice è formato normalmente da un solo blocco di disco perciò **ciascun blocco indice può essere letto e scritto.** Per permettere la presenza di lunghi file è possibile collegare tra loro vari blocchi indice
2. **Indice a più livelli 🡪** Una variante della rappresentazione concatenata consiste nell’impiego di **un blocco indice di primo livello che punta a un insieme di blocco indice di secondo livello** che, a loro volta, puntano ai blocchi dei file.
3. **Schema combinato 🡪** Un’altra possibilità, utilizzata nei sistemi basati su UNIX, consiste nel **tenere i primi 15 puntatori del blocco indice nell’inode del file. I primi 12 di questi 15 puntano a blocchi diretti. Gli altri tre puntatori puntano a blocchi indiretti**. Il primo è un puntatore a un blocco indiretto singolo, si tratta di un blocco indice contenente puntatori a blocchi di dati, il secondo a un blocco indiretto doppio, si tratta di un blocco contenente l’indirizzo di un blocco che a sua volta contiene gli indirizzi di blocchi contenenti puntatori agli effettivi blocchi di dati, il terzo è un puntatore a un blocco indiretto triplo.

**12.5 Gestione dello spazio**

Poiché la quantità di spazio dei dischi è limitata è necessario riutilizzare lo spazio lasciato dai file cancellati per scrivere nuovi file. Per tener traccia dello spazio libero il sistema conserva una lista dello spazio libero. Per creare un file occorre cercare nella lista dello spazio libero la quantità di spazio necessaria e assegnarla al nuovo file.

**12.5.1 vettori di bit**

Spesso la lista dello spazio libero si realizza **come una mappa di bit, o vettore di bit. Ogni blocco è rappresentato da un bit: se il blocco è libero il bit è 1, se il blocco è assegnato il bit è 0**. I vantaggi che derivano da questo metodo sono la sua relativa semplicità ed efficienza nel trovare il primo blocco libero. **I vettori di bit sono efficienti solo se tutto il vettore è mantenuto in memoria centrale, ma è possibile tenere questo vettore solo se il disco è piccolo.**

**12.5.2 Lista concatenata**

Un altro metodo di gestione degli spazi liberi **consiste nel collegarli tutti, tenere un puntatore al primo di questi in una speciale locazione del disco e caricarlo in memoria.** Un puntatore al primo blocco della lista (che a sua volta contiene un puntatore al secondo, e così via) viene mantenuto in una speciale locazione del disco e caricato in memoria. **Questo schema non è efficiente per attraversare la lista è infatti necessario leggere ogni blocco**, con un notevole tempo di I/O.

**12.5.3 Raggruppamento**

Una possibile modifica del metodo della lista dello spazio libero prevede la **memorizzazione degli indirizzi di n blocchi liberi nel primo di questi**.

**12.5.4 Conteggio**

**Un altro approccio sfrutta il fatto che, generalmente, più blocchi contigui si possono allocare o liberare contemporaneamente**, soprattutto quando lo spazio viene allocato usando l’algoritmo di allocazione contigua o attraverso l’uso di cluster.

**12.6 Efficienza**

I dischi tendono ad essere il principale collo di bottiglia per le prestazioni di un sistema essendo i più lenti tra i componenti di un calcolatore. L’uso efficiente di un disco dipende dalle tecniche di allocazione dello spazio su disco e dagli algoritmi di gestione delle directory. Ad esempio, gli i-node di UNIX sono preallocati in una partizione. Anche un disco “vuoto” impiega una certa percentuale del suo spazio per gli i-node. D’altra parte, preallocando gli i-node e distribuendoli lungo la partizione si migliorano le prestazioni del file system. Queste migliori prestazioni sono il risultato degli algoritmi di allocazione e di gestione dei blocchi liberi adottati da UNIX. Questi algoritmi mantengono i blocchi di dati di un file vicini al blocco che ne contiene l’i-node allo scopo di ridurre il tempo di posizionamento. Anche il tipo di dati normalmente contenuti in un elemento di una directory (o di un i-node) deve essere tenuto in considerazione.

Solitamente viene memorizzata la data di ultima scrittura, per fornire informazioni all’utente e per determinare se il file necessita o meno della creazione o aggiornamento di una copia di backup. Alcuni sistemi mantengono anche la data di ultimo accesso per consentire all’utente di risalire all’ultima volta che un file è stato letto. Il risultato del mantenere queste informazioni è che ogni volta che un file viene letto si dovrà aggiornare un campo della directory. Questa modifica richiede la lettura in memoria del blocco, la modifica della sezione e la riscrittura del blocco su disco, poiché sui dischi è possibile operare solamente per blocchi (o gruppi di blocchi). Quindi, ogni volta che un file viene aperto in lettura, anche l’elemento della directory a esso associato deve essere letto e scritto. Questo requisito può risultare inefficiente per file a cui si accede frequentemente, quindi al momento della progettazione del file system è necessario confrontare i benefici con i costi in termini di prestazioni. In generale, è necessario considerare l’influenza sull’efficienza e sulle prestazioni di ogni informazione che si vuole associare a un file

**Capitolo 13**

**Sistemi di I/O**

**13.1 Introduzione**

Il controllo dei dispositivi connessi a un calcolatore è una delle questioni più importanti che riguardano i progettisti di sistemi operativi. **Poiché i dispositivi di I/O sono così largamente diversi altrettanto diversi sono i metodi di controllo**. Tali metodi costituiscono il sottoinsieme di I/O del kernel.

**13.2 Hardware di I/O**

Un dispositivo comunica con un sistema elaborativo inviando segnali attraverso un cavo o attraverso l’etere e comunica con il calcolatore tramite un punto di connessione. **Se più dispositivi condividono un insieme di fili, la connessione è detta bus**. Un bus è un insieme di fili e un protocollo rigorosamente definito che specifica l’insieme dei messaggi che si possono inviare attraverso i fili. I bus sono ampiamente usati nell’architettura dei calcolatori e differiscono tra loro per formato dei segnali, velocità, throughput e metodo di connessione. Un bus PCI connette il sottosistema CPU- memoria ai dispositivi veloci. Altri tipi comuni di bus usati per connettere i principali componenti di un computer includono il PCI Express e HyperTransport. **Un controllore è un insieme di componenti elettronici che può far funzionare una porta, un bus o un dispositivo**. L’unità di elaborazione fornisce comandi e dati al controllore per portare a termine trasferimenti di I/O tramite uno o più registri per dati e segnali di controllo.

**13.2.1 Polling**

**Il polling è la verifica ciclica di tutte le unità o periferiche di input/output da parte del sistema operativo di un personal computer tramite test dei bit di bus associati ad ogni periferica, seguita da un'eventuale interazione** (scrittura o lettura). Questa attività impegna molto del tempo di funzionamento del processore (CPU), rallentando di conseguenza l'intero sistema. l polling presenta pochi vantaggi e alcuni svantaggi.  
**Il vantaggio principale è quello di riuscire a gestire le periferiche sia come struttura hardware che software** (cosa che non fanno le interruzioni), ma presenta degli svantaggi:  
**Potenzialmente la CPU potrebbe sprecare preziose risorse di calcolo: se, ad esempio, nessuna periferica fosse pronta al colloquio**, la CPU farebbe comunque un controllo a vuoto sui bit di stato di tutte le periferiche; L'intervallo tra due interrogazioni del bit di stato non è costante, ma dipende dallo stato delle periferiche; Ci si può trovare in una situazione di emergenza oppure che una periferica mandi un segnale alla CPU per continuare il suo lavoro. Tutti questi svantaggi vengono limitati dall'uso delle interruzioni.

**13.2.2 Interruzioni**

**La CPU ha un contatto detto linea di richiesta dell’interruzione del quale controlla lo stato dopo l’esecuzione di ogni istruzione.** **Quando rileva il segnale di un controllore, la CPU memorizza lo stato del processo corrente e salta alla procedura di gestione delle interruzioni** che si trova ad un indirizzo prefissato, tramite questa procedura gestirà l’interruzione. Quando l’I/O è terminato, un interrupt segnala che i dati sono pronti e il processo può essere ripreso. Nel frattempo, la CPU può mandare in esecuzione altri processi o altri thread dello stesso processo.

**Il vettore delle interruzioni: tabella che associa ad ogni interrupt l’indirizzo di una corrispondente routine di gestione.** In genere esistono due linee di richiesta delle interruzioni:

1. Interruzioni non mascherabili 🡪 **errori di memoria irrecuperabili, non possono essere disattivate.**
2. Interruzioni mascherabili 🡪 **Interruzioni che possono essere disattivate dalla CPU prima di una sequenza critica di istruzioni che non deve essere interrotta.**

Un sistema di livelli di priorità delle interruzioni permette alla CPU di differire la gestione delle interruzioni di bassa priorità e permette ad una interruzione di priorità alta di sospendere l’esecuzione di quella di una bassa. Gli interrupt vengono usati anche per indicare eccezioni.

**13.2.3 Accesso diretto alla memoria (DMA)**

In molti sistemi di elaborazione si utilizza una unità di elaborazione specializzata, detta controllore dell’accesso diretto alla memoria (DMA). **Il controllore di DMA è dedicato al trasferimento diretto delle informazioni dalle periferiche alla memoria centrale**. Il DMA viene utilizzato nelle operazioni di trasferimento di grosse quantità di dati. Per dare avvio ad un trasferimento DMA, **la CPU scrive in memoria un comando che contiene un puntatore alla locazione dei dati ed un altro alla destinazione, ed il numero di byte da trasferire. Poi passa l’indirizzo del comando al controllore del DMA e prosegue con l’esecuzione di altro codice.** Il controllore DMA agisce direttamente sul bus della memoria, presentando al bus gli indirizzi di memoria necessari al trasferimento. Con l’accesso diretto alla memoria virtuale (DVMA) l’accesso diretto avviene allo spazio indirizzi virtuale del processo, e non a quello fisico (per l’I/O associato alla memoria). In questo caso si usano indirizzi virtuali che poi si traducono in indirizzi fisici.

**13.3 Interfaccia di I/O**

È necessario avere un trattamento uniforme dei dispositivi di I/O, quindi **le chiamate di sistema di I/O incapsulano il comportamento dei dispositivi in alcuni tipi generali. Le effettive differenze tra I dispositivi sono contenute nei driver, moduli del kernel dedicati a controllare ogni diverso dispositivo**. Sfortunatamente ogni tipo di S.O. ha le sue convenzioni riguardanti l’interfaccia dei driver dei dispositivi. Così un dato dispositivo sarà venduto con driver diversi, ad es. per MS-DOS, Windows, Unix, etc... **Per l’accesso delle applicazioni ai dispositivi, le chiamate di sistema raggruppano tutti i dispositivi in poche classi generali, uniformando I modi di accesso**. Solitamente queste classi sono:

1. I/O a blocchi o a flusso di caratteri
2. accesso mappato in memoria
3. socket di rete.

Spesso è disponibile una system call scappatoia (back-door) dove si fa rientrare ciò che non rientra nei casi precedenti

**13.3.1 Dispositivi con trasferimento a blocchi e a caratteri**

L’interfaccia per i dispositivi a blocchi **sintetizza gli aspetti necessari per accedere alle unità a disco e ad altri dispositivi basati sul trasferimento di blocchi di dati**. Comandi tipici: read, write, seek. Di solito le applicazioni comunicano con i dispositivi tramite un file system che funge da interfaccia. Altre volte i dispositivi sono trattati come una semplice sequenza lineare di blocchi (I/O a basso livello). I file possono essere associati alla memoria: si fa coincidere una parte dello spazio indirizzi virtuale di un processo con il contenuto di un file. I dispositivi a carattere sono dispositivi che generano o accettano uno stream di dati, ad esempio: tastiera, mouse, porte seriali. Comandi tipici get, put di singoli caratteri o parole. Non è possibile la seek.

**13.3.2 Dispositivi di rete**

I modi di indirizzamento e le prestazioni tipiche dell’I/O di rete sono differenti da quelli dei dispositivi a blocchi o caratteri. La maggior parte dei S.O. fornisce un’interfaccia per l’I/O di rete diversa da quella per i dischi. **L’interfaccia per l’I/O di rete fornita su molti S.O. è l’interfaccia di rete socket. Le system call fornite da questa interfaccia permettono di creare un socket, collegare un socket locale all’indirizzo di un altro punto della rete, controllare se un’applicazione si sia inserita nella socket locale, e inviare o ricevere pacchetti di dati lungo la connessione**. Una funzione select gestisce un insieme di socket, restituendo informazioni sui socket per i quali sono presenti pacchetti che attendono di essere ricevuti e su quelli che hanno spazio per accettare un pacchetto da inviare. select elimina la necessità di interrogazione ciclica.

**13.3.3 Orologi e timer**

La maggior parte dei computer ha temporizzatori e orologi hardware che offrono tre funzioni essenziali:

1. **segnare l’ora corrente.**
2. **segnalare il lasso di tempo trascorso.**
3. **regolare un temporizzatore in modo da avviare l’operazione x al tempo t.**

**Il dispositivo che misura la durata di un lasso di tempo e che può avviare un’operazione è detto temporizzatore programmabile.** Si può regolare in modo da attendere un certo tempo e poi generare un segnale di interruzione. Viene utilizzato anche per segnalare lo scadere della slice di tempo di un processo, per riversare periodicamente nei dischi il contenuto della buffer cache, etc... Per l’ora corrente vi è un contatore di interrupt ad alta frequenza chiamato orologio hardware.

**13.3.4 I/O non bloccante e asincrono**

Un altro aspetto delle chiamate del sistema è la scelta tra I/O bloccante e non bloccante**. Una system call bloccante sospende l’esecuzione dell’applicazione che l’ha invocata**. Questa passa dalla coda dei processi pronti alla coda d’attesa del sistema. Quando la chiamata a sistema termina l’applicazione torna nella coda dei processi pronti. **Alcuni processi necessitano di una forma non bloccante di I/O. Una system call di questo tipo restituisce rapidamente il controllo all’ applicazione che la ha invocata**, fornendo un parametro che indica quanti byte di dati sono stati trasferiti. **Un’ alternativa alle system call non bloccanti è costituita dalle chiamate del sistema asincrone**. Con esse, l’applicazione continua ad essere eseguita e **il completamento dell’I/O viene successivamente comunicato all’ applicazione per mezzo di una variabile** o di un interrupt software o tramite una routine di ritorno.

**13.4 Sottosistema di I/O del kernel**

**13.4.1 Scheduling dell’I/O**

Il kernel fornisce i servizi riguardanti l’I/O. Molti sono offerti dal sottosistema per l’I/O del kernel e sono realizzati a partire dai dispositivi e dai relativi driver. Fare lo scheduling di un insieme di richieste di I/O significa stabilirne un ordine di esecuzione efficace. Lo scheduling è necessario per:

1. concorrere a migliorare le prestazioni globali del sistema
2. distribuire equamente gli accessi ai dispositivi
3. ridurre il tempo di attesa medio per il completamento di un’operazione di I/O associata ad un dispositivo

Ad es. riordinamento delle sequenze di servizio di richieste relative ad operazioni di lettura su disco.

**I progettisti di S.O. realizzano lo scheduling mantenendo una coda di richieste di I/O associata ad ogni dispositivo**. Quando una applicazione richiede l’esecuzione di una chiamata a sistema di I/O bloccante, si aggiunge alla coda relativa al dispositivo la richiesta. Lo scheduler dell’I/O riorganizza l’ordine della coda per migliorare l’efficienza globale del sistema ed il tempo medio di attesa cui sono sottoposte le applicazioni. Lo scheduling dell’I/O è uno dei modi in cui il sottosistema per l’I/O migliora l’efficienza di un calcolatore. Un altro è l’uso di spazio di memorizzazione nella memoria centrale o nei dischi, per tecniche di memorizzazione transitoria, uso di cache e di code per la gestione asincrona dell’I/O.

**13.4.2 Gestione dei buffer**

**Una memoria di transito (buffer) è un’area di memoria che contiene dati mentre essi sono trasferiti tra due dispositivi o tra una applicazione e un dispositivo.** La memorizzazione transitoria viene utilizzata per tre ragioni principali:

1. **Sincronizzare dispositivi caratterizzati da diverse velocità di trasferimento** (ad es. modem disco: doppia memorizzazione transitoria).
2. Gestione dei **dispositivi che trasferiscono blocchi di dati in blocchi di dimensione diverse**: (ad es. scambi di messaggi in rete).
3. Realizzare la semantica delle copie: meccanismo per la coerenza delle informazioni (ad es. applicazione disco). La semantica delle copie garantisce che la versione dei dati scritta su disco sia conforme a quella al momento della chiamata a sistema.

**13.4.3 Cache**

**Una cache è una memoria ad alta velocità utilizzata dai dispositivi di I/O per rendere più efficace il trasferimento di dati**. Essa viene utilizzata per mantenere copie di dati contenuti in un dispositivo. L’uso della cache e l’uso delle memorie di transito sono due funzioni distinte. Buffer e caching differiscono poiché un buffer contiene dati di cui non esiste un’altra copia, mentre la cache contiene, per definizione, copie di informazioni già memorizzate.

**13.4.4 Code di spooling e riservazione dei dispositivi**

**La tecnica dell’accodamento (spooling) si riferisce allo spostamento di dati in un buffer (o in un disco fisico) in attesa di essere smistati verso il dispositivo o l'applicativo che li deve elaborare**. Lo spooling è molto utile quando i dispositivi accedono ai dati a una velocità variabile**. Il buffer rappresenta una stazione di attesa dove i dati possono rimanere fino a che il dispositivo più lento non riesce a gestire i dati in attesa**. La tecnica dello spooling viene utilizzata per ottenere un uso esclusivo dei dispositivi. In alcuni S.O. lo spooling è gestito da un processo di sistema specializzato (daemon di spool), in altri da un thread del kernel.

**13.4.5 Gestione degli errori**

Il S.O. deve proteggersi dal malfunzionamento dei dispositivi. Gli errori possono essere:

1. **transitori** 🡪 ad es. rete sovraccarica
2. **permanenti** 🡪 ad es. disco rotto.

**Nel caso di situazioni transitorie, solitamente il S.O. può tentare di recuperare la situazione** Le chiamate di sistema, quando non vanno a buon fine, segnalano una situazione di errore. In situazioni con errori permanenti, la gestione degli errori è più difficile. Alcuni S.O., come UNIX, forniscono una mappatura degli errori. Alcuni tipi di hardware, poiché gli errori sono in relazione al tipo di dispositivo, forniscono un protocollo per la rilevazione dell’errore.

**13.4.7 Struttura dati del kernel**

Il S.O. utilizza particolari strutture dati interne al kernel per mantenere informazioni riguardo allo stato dei componenti coinvolti nelle operazioni di I/O. Ad esempio la tabella dei file aperti. Sia UNIX che Window risolvono il problema dell’ottenere una struttura dati che consenta di effettuare migliori operazioni, seguendo quello che è l’approccio della programmazione orientata agli oggetti; cioè aggregano nella struttura dati sia dati che metodi (funzioni).

**13.4.8 Concetti principali del sottosistema di I/O del kernel**

In generale un sistema per l’I/O utilizza le seguenti funzioni:

1. gestione dello spazio dei nomi per file e dispositivi
2. controllo dell’accesso ai file e ai dispositivi
3. controllo delle operazioni
4. allocazione dello spazio per il file system
5. allocazione dei dispositivi
6. gestione dei buffer, della cache e delle code di spooling
7. scheduling dell’I/O
8. controllo dello stato dei dispositivi, gestione degli errori e procedure di ripristino
9. configurazione e inizializzazione dei driver dei dispositivi

**13.5 Trasformazione delle richieste di I/O**

In MS-DOS la prima parte del nome di un file identifica in modo univoco una periferica: C: è la parte iniziale di ogni nome di file residente nell’unità dischi principale, per convenzione del S.O. a C: viene associato uno specifico indirizzo di porta per mezzo di una tabella dei dispositivi. In UNIX non c’è nel pathname una vera separazione fra il dispositivo interessato ed il nome del file. Viene impiegata una tabella di montaggio, per associare un prefisso di pathname ad uno specifico nome di dispositivo. Quando deve risolvere il nome di percorso, il sistema esamina la tabella per trovare il più lungo prefisso corrispondente: questo elemento indica il nome del dispositivo voluto. Il nome del dispositivo viene rappresentato come un oggetto del file system, non come un i-node ma come una coppia di numeri <principale, secondario>:

1. principale individua il driver del dispositivo che deve essere usato per gestire l’I/O.
2. secondario individua l’indirizzo della porta o l’indirizzo mappato in memoria del controller del dispositivo.

**13.7 Prestazioni**

L’I/O è un fattore predominante nelle performance di un sistema:

1. **Consuma tempo di CPU per eseguire i driver e il codice kernel di I/O**
2. **Continui cambi di contesto all’avvio dell’I/O e alla gestione degli interrupt.**
3. **Trasferimenti dati da/per I buffer consumano cicli di clock e spazio in memoria.**
4. **Il traffico di rete è particolarmente pesante (es. telnet).**

Per migliorare l’efficienza dell’I/O si possono applicare diversi principi:

1. Ridurre il numero dei cambi di contesto.
2. Ridurre il numero di copiature dei dati nella memoria durante i trasferimenti tra dispositivi e applicazioni.
3. Ridurre la frequenza degli interrupt preferendo trasferimenti di grandi quantità di dati: usare controller intelligenti e l’interrogazione ciclica.
4. Aumentare il tasso di concorrenza usando controllori DMA o bus dedicati, per sollevare la CPU dalle semplici copiature di dati
5. Implementare le primitive in hardware, dove possibile, per aumentare il parallelismo
6. Equilibrare le prestazioni di CPU, del sottosistema per la gestione della memoria, del bus e dell’I/O: Il sovraccarico di un elemento comporta l’inutilizzo degli altri.

**Lezioni di laboratorio:**

**Prima e seconda lezione:** Introduzione alla bash e ad alcuni semplici comandi di Linux.

**Terza lezione di laboratorio:** Utilizzo della fork e dei metodi **Getpid** e **Getppid.**

**Leggere:** capitolo 8 Stevens: Process control.

**Compilazione c in linux:**

1. **gcc nome\_programma.c**
2. **./a.out**

Se non voglio mettere a.out devo fare gcc nome\_programma.c -o nuovo\_out.out.

**Piccola parentesi** su **vi** un editor che non ci piace.

**Metodi getpid** e **getppid:**

* **Getpid:** ritona il nome del processo chiamante
* **Getppid:** ritorna il nome del padre

**Istruzione fork:**

crea un processo (figlio di quello creante) e restituisce il pid (per il padre quello del figlio, per il figlio 0).

* **Caso padre con più figli:** Quando un padre crea più figli dobbiamo stare attenti che i figli non creino altri processi (visto che il codice del padre vine rieseguito dal figlio alla sua creazione) per fare ciò, visto che sappiamo che il pid del filgio è uguale a 0, la seconda chiamata di fork la andiamo a mettere in un if con (pid!=0).

**Terza lezione di laboratorio:**

**Wait:** è un system call che serve ad ‘un processo padre per attendere la terminazione del figlio. È contenuta all’interno della libreria #include <syst/wait.h>. Praticamente quando l’esecuzione all’interno di un processo padre arriva alla wait Il sistema “mette a dormire” il processo per poi “svegliarlo” quando un figlio termina l’esecuzione.

**Sintassi:** PID\_T wait (int\* statlook)

La system call restituisce il PID del figlio che ha terminato per primo, il parametro invece serve ad indicare lo stato di terminazione del figlio tramite passaggio per riferimento

**Waitpid:** versione potenziata della system call wait.

**Sintassi:** Pid\_t waitpid(pid\_t pid ,int \*status, int options);

**Parametri:**

| **Valore pid\_t** | **Descrizione** |
| --- | --- |
| **-1** | Aspetta la terminazione di un processo generico |
| **>0** | Aspetta la terminazione del figlio con quel determinato pid |
| **< -1** | la funzione attende ciascun figlio avente il process group ID uguale al valore assoluto del pid. |
| **=0** | Aspetta la terminazione dell’intero albero di processi (tutti i figli) |

| **Valore di options** | **Descrizione** |
| --- | --- |
| **WNOHANG** | specifica al ritorno immediato se i figli sono usciti |
| **WUNTRACED** | specifica di ritornare anche se il figlio è fermato e lo stato non deve venire riportato. |

**Exec:** è un system call che sostituisce i segmenti codice e dati del processo correntemente in esecuzione nello stato di utente con quelli di un altro programma contenuto in un file eseguibile specificato. Agendo solo su processi in stato di utente la exec non interviene sul segmento di sistema e sui file utilizzati dal processo che la invoca.

Durante la chiamata ad exec e al successivo caricamento nella Ram del nuovo codice compilato il processo mantiene lo stesso pid.

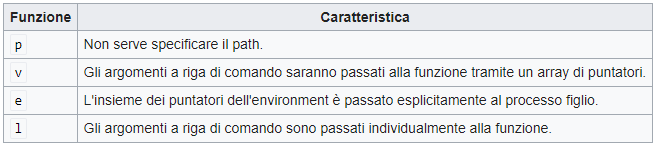
La funzione exec deve passare dei parametri al nuovo programma che viene eseguito. Essi vengono letti dal programma tramite il meccanismo di passaggio usuale argc, argv.

**Sintassi:** int exec(const char \*path, const char \*arg0, ...);

**Parametri:**

* **Path:** specifica il pathname del file da eseguire come processo figlio.
* **Argn:** è lista di puntatori agli argomenti da passare al processo figlio.

**Parametri aggiuntivi:** Il comportamento della system call viene modificato in base alle lettere poste dopo il nome, seguendo la seguente tabella:



**Segnali:** è un interrupt software che permette di gestire eventi asincroni, può essere generato in qualsiasi instante. Ogni segnale comincia con **SIG** (ad es. SIGABRT, SIGALARM) a cui viene associata una costante intera positiva definita in signal.h. Un segnale è un evento asincrono:

* Può arrivare in qualsiasi momento, anche se non è detto che faccia subito effetto.
* Vi sono azioni di default che vengono compiute
* Il processo può decidere se ignorarlo o gestirlo in maniere diversa dal default.

Al sopraggiungere di un segnale un processo può comportarsi nei seguenti modi:

* **Ignorare il segnale:** tranne per **SIGKILL** e **SIGSTOP** (servono al super-user per terminare o stoppare i processi).
* **Catture il segnale:** ovvero eseguire una determinata funzione (**SIGTERM** per esempio cancella tutti i file temporanei del processo.
* **Eseguire l’operazione di default:** ovvero la terminazione del processo nella maggior parte dei casi.

**Tipi di segnali:**

* **SIGABRT:** termina il modo anormale il processo tramite la system-call **abort** da cui è generato
* **SIGCHLD:** segnale generato da un processo figlio (e inviato al padre) al momento della sua terminazione. Nella maggior parte dei casi il padre lo ignora, mentre nel caso siamo implementata una wait viene utilizzato per restituire il pid.
* **SIGCONT:** inviato ad ‘un processo fermo per fargli riprendere l’esecuzione.
* **SIGFPE:** viene inviato quando c’è una divisione per zero.
* **SIGILL:** viene inviato quando l’hardware individua una istruzione illegale (mancanza di strutture per la gestione della virgola mobile).
* **SIGINT:** viene inviato quando viene premuto il tasto DELETE o CTRL-C (default termina).
* **SIGALRM:** generato dalla system-call **alarm**.
* **SIGQUIT:** genera un’immagine in memoria del processo che può essere utilizzata per il debugging.
* **SIGKILL:** termina il processo che lo riceve (non può essere ignorato).
* **SIGSEGV:** il processo ha fatto riferimento ad un indirizzo che non è nel suo spazio di memoria.
* **SIGSTOP:** ferma il processo (non può essere ignorato).
* **SIGSYS:** il processo ha eseguito una system-call senza i parametri adeguati.
* **SIGTERM:** Il segnale di terminazione inviato per dafault dalla system-call **kill**.
* **SIGUSR1 – SIGUSR2:** possono essere utilizzati e definiti dall’utente. A volte sono usati per la comunicazione tra processi.

**Funzione signal:** è una funzione che presi due argomenti, funziona da questo momento in poi, non va messa in un ciclo (per esempio CTRL-C cambia significato se si fa una signal su SIGINT):

* **Signo:** che è il nome del segnale.
* **Func:** funzione da eseguire all’arrivo del segnale signo.

Restituisce il puntatore ad’una funzione che prende come argomento un intero. In caso di errore restituisce SIG\_ERR (cioè -1).

**#include <signal.h>**

**Sintassi:** void (\*signal(int signo,void (\*func)(int)))(int);

Il valore di **func** può essere:

* **SIG\_IGN**: per ignorare il segnale (tranne stop e kill)
* **SIG\_DFL\_** : per settare l’azione associata di default ad’un segnale
* **Lindirizzo di una funzione** che sarà eseguita quando il segnale occorre.

**Kill e Raise:** inviano un segnale **signo** specificato come argomento. Restituiscono, 0 se OK, -1 in caso di errore.

**Sintassi:**

int kill(pid\_t pid, int signo);

int raise (int signo);

in particolare:

* **kill:** manda un segnale ad ‘un processo o ad un gruppo di processi specificato da pid.
* **raise:** consente ad ‘un processo di mandare un segnale a sé stesso.

Se il valore di ritorno di **pid** (funzione kill) è:

* **>0:** il segnale è inviato al segnale con il processo **pid.**
* **==0:** il segnale è inviato al gruppo di processi.
* **<0:** c’è stato un errore.

**sleep:** serve a interrompere un processo per un tot di millisecondi. Restituisce 0 oppure il numero di secondi “non dormiti”.

**Alarm:** fa partire un conto alla rovescia di **secs** secondi (sono un po' fittizi per motivi di schedulazione visto che il processo non è sempre eseguito) , allo scadere di questo conto al processo viene inviato il segnale **SIGALRM**, se secs è a 0 la sveglia precedente viene cancellata.

**Sintassi:** Unsigned int alarm (unsigned int secs);

valori di ritorno:

* 0
* Il numero di secondi mancanti alla sveglia precedentemente settata.

**Pause:** sospende l’esecuzione finché non interviene un segnale.

**Sintassi:** Int pause (void);

valori di ritorno:

* -1 con **errno** settato a **EINTTR.**

**System call capitolo tre sui File:**

**Istruzione open:**

int open (const char \*pathname, int flags, mode\_t mode); --> apre un file

mode --> opzionale

**Librerie usate:**

* #include <sys/types.h>
* #include <sys/stat.h>
* #include <fcntl.h>

**Alcuni Flag:**

* **O\_RDONLY**: solo lettura
* **O\_WRONLY**: solo scrittura
* **O\_RDWR**: scrittura e lettura
* **O\_APPEND**: posiziona il current offset alla fine del file. Le successive operazioni di scrittura verranno accodate alle informazioni già presenti nel pathname.
* **O\_CREAT**: se pathname non esiste viene creato. (bisogna usare il parametro mode)
* **O\_EXCL**: quando usato con O\_CREAT, se il file è esistente viene tornato un errore a la open fallisce.
* **O\_TRUNC**: un file esistente verrà troncato.

**Valori di ritorno:**

* file descriptor (rappresentato dal più piccolo intero non ancora utilizzato)
* -1 se si è verificato un errore

**Per creare un file:**

* open (pathname, O\_WRONLY|O\_CREAT|O\_EXCL, mode);

**Istruzione close:** chiude il file descriptor fd.

int close (int fd);

**Librerie usate:**

* #include <unistd.h>

**Istruzione Iseek:** riposiziona l'offset per la successiva operazione di read/write.

off\_t Iseek (int fd, off\_t offset, int whence);

**Librerie usate:**

* #include <sys/types.h>
* #include <unistd.h>

**current offset:** indica la prossima posizione da leggere/scrivere in un file, è un puntatore

**whence:**

* **SEEK\_SET**: offset è aggiunto dall'inizio del file.
* **SEEK\_CUR**: la nuova posizione è calcolata aggiungendo offset alla posizione corrente.
* **SEEK\_END**: la nuova posizione è calcolata aggiungendo offset dalla fine file.

**Istruzione read:** legge da un file descriptor

ssize\_t read (int fd, void \*buf, size\_t count);

**Librerie usate:**

* #include <unistd.h>

**Istruzione write:**

ssize\_t write (int fd, const void \*buf, size\_t count);

**Librerie usate:**

* #include <unistd.h>

Un Processo apre automaticamente i file descriptor 0, 1 e 3 ovvero i 3 flussi standard:

* **STDIN**(STDIN\_FILENO)
* **STDOU**(STDOU\_FILENO)
* **STDERR**(STDERR\_FILENO)