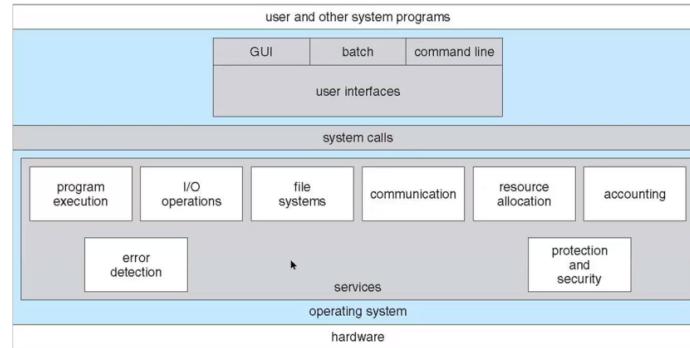


PER ALTRI APPUNTI CONSULTARE IL SITO:
https://luigi-v.github.io/Appunti_Universita/

1.0 SERVIZI PER L'UTENTE GENERICO

La visione della macchina che abbiamo, e che l'utente generico osserva, è la seguente:



Alla base di tutto c'è l'hardware (CPU, parti meccaniche, etc...). Al di sopra di queste parti meccaniche è presente il sistema operativo, il quale mette a disposizione le risorse che hanno l'obiettivo di far eseguire un dato programma all'utente (comprende tutta la parte che nell'immagine viene vista in azzurro). In particolare, i servizi che il sistema operativo mette a disposizione sono:

- **eseguire programmi**: il SO deve caricare in memoria ed eseguire i programmi e garantire la terminazione dell'esecuzione in modo normale o anormale (eventualmente indicandone l'errore).
- Come noi sappiamo, i programmi che scriviamo risiedono nell'Hard Disk, però, ogniqualvolta un esecutivo deve essere eseguito, bisogna caricarlo in memoria principale perché l'unico modo che ci permette di eseguire un programma, è che esso risieda in memoria principale, in quanto la CPU comunica solo con essa. Una volta che il programma è in memoria principale, la CPU esegue riga dopo riga il programma.
- **effettuare operazioni di input/output**: quando stiamo eseguendo un programma in memoria principale, il SO deve garantire che in un momento qualsiasi del programma si necessiti di un file (che naturalmente risiede sul disco), anche quest'ultimo deve essere portato all'interno del pezzo di memoria principale che è stato assegnato precedentemente al programma.
- **gestire il file system**: significa che il SO, non solo deve mettere a disposizione la possibilità di effettuare input/output su un file, ma anche creare file, creare una directory ed all'interno inserire un file, sapere quali sono i permessi di accesso di un file. Il SO deve garantire questi servizi.
- **comunicazione tra processi**: il SO deve garantire che più processi possono scambiarsi informazioni, sullo stesso computer o tra computer su una rete.
- **rilevazione dell'errore**: il SO deve essere sempre informato sugli errori di tipo hardware o software.
- **allocazione risorse**: il SO deve garantire il corretto passaggio dell'esecutivo dal disco alla memoria principale.
- **contabilizzazione**: contare quanti utenti ci sono, e quanta memoria stanno utilizzando, utile per la gestione totale del sistema.
- **protezione e sicurezza**:

Tutti questi servizi che sono stati elencati, vengono realizzati mediante le **system calls**, che sono l'unica modalità che ci consente di andare in kernel mode ed eseguire fisicamente ciò di cui abbiamo bisogno.

Al disopra del sistema operativo ci sono le interfacce utente che ci consentono di comunicare col SO e che fondamentalmente possono essere di 2 tipi: **interfaccia CLI** (Command Line Interface), in cui si comunica al sistema operativo attraverso comandi. Un'operazione del genere è garantita dalla shell. Ci sono poi le **interfaccia grafiche** (GUI) che mettono a disposizione ad esempio, un desktop amichevole, i comandi sono forniti tramite mouse, tastiera e monitor.

1.1 SERVIZI PER IL PROGRAMMATORE: SYSTEM CALL

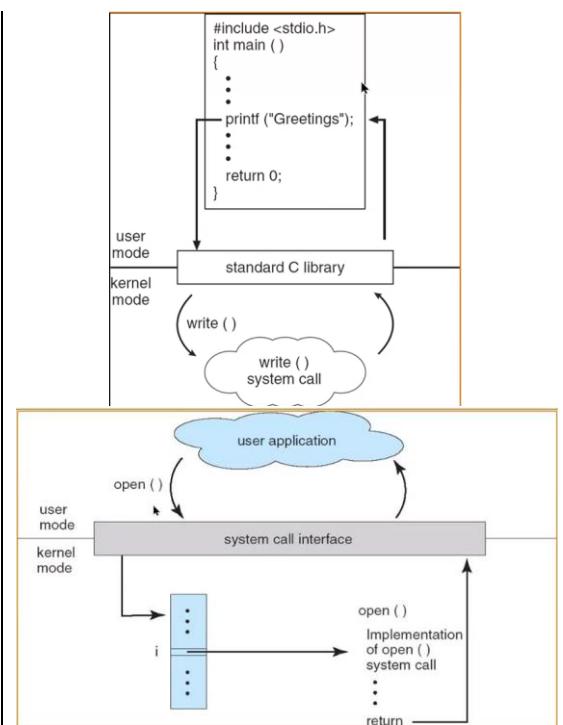
Quando l'utente chiede al SO di voler effettuare un servizio, questo viene realizzato attraverso le system call. Tipicamente le system call sono delle interfacce che il SO mette a disposizione del programmatore in maniera tale che una qualsiasi richiesta venga fatta al sistema operativo, viene gestita mediante system call. Usando le system call, si usano delle funzioni di libreria (API) che sono più facili da gestire nella pratica e che nascondono i dettagli implementativi che la system call utilizza effettivamente.

Ad esempio, la printf è una funzione di libreria →

Immaginiamo di avere questa porzione di codice, e ad un certo punto viene invocata la printf. Quando viene invocata la printf, il SO va a prendere l'effettivo codice della printf scritto da qualche parte, lo porta in memoria principale e inizia ad interpretarlo. Ad un certo punto, nel codice della printf appare la funzione `write()`, che è una system call, e di conseguenza passa in kernel mode e ciò permetterà di stampare a schermo l'informazione che l'utente richiede. La `write()` ha esattamente l'aspetto di una normale funzione C, solo che essendo una system call *non esiste una porzione di codice che implementa tale funzione*, infatti le system call sono qualcosa di fisico che il kernel utilizza per poter realizzare quello che è stato chiesto.

Poiché non esiste una porzione di codice che implementa le system call, chiaramente esiste un altro tipo di gestione che viene adesso descritto →

Il SO associa ad ogni system call un numeretto. Ogni qualvolta si chiede al SO di mandare una system call, nell'esempio la funzione `open()` e per esempio sa che la funzione ha numero 10, nel kernel del SO c'è una sorta di tabella al cui interno ci sono i numeri contenenti ciascuna system call, per cui va nell'entry 10 ed è presente un link ad una sequenza di istruzioni che devono eseguire per l'implementazione della system call `open()`. Di conseguenza noi non avremo mai a disposizione l'implementazione della system call `open`, ma sappiamo naturalmente ciò che fa.



2. FILE

Il File System è la parte più visibile del SO e fornisce i meccanismi per la memorizzazione e l'accesso ai dati e ai programmi del SO e degli utenti del sistema di calcolo. Un File System consiste di due parti:

- Un insieme di file
- Una struttura di directory che permette di organizzare tutti i file del sistema

Un file è una collezione di informazioni che sono correlate tra di loro e vengono conservate su una memoria non volatile a cui è stato assegnato un nome che lo identifica univocamente. Rappresentano un'unità logica di memorizzazione. I file contengono dati (numerici, caratteri, binari) e programmi (sorgenti, linkabili, eseguibili). La cosa che è importante da ricordare come descritto in precedenza è che sono conservati su memoria non volatile (dischi, memorie esterne, etc...). I file possono essere di vario tipo:

- **File di testo:** sequenza di caratteri, parole, linee, pagine.
- **File sorgente:** sequenza di subroutine e funzioni.
- **File oggetto:** sequenza di byte organizzate in blocchi che risultano comprensibili al linker del SO.
- **File eseguibile:** serie di sezioni di codice binario che il caricatore porta in memoria ed esegue.

Per capire un file di che tipo è, s'inserisce alla fine del nome un'estensione. Ad esempio, se un file ha estensione .txt o .doc allora sappiamo che è un file di testo.

Mac OS X conserva, all'atto della creazione di un file, anche il nome del programma che ha creato il file (questo gli dà modo di poter riconoscere il tipo del file). **Unix** memorizza un codice (numero magico) all'inizio di alcuni tipi di file in modo da indicarne in modo generico il tipo (eseguibile, script shell, PostScript). Ciascun file è identificato attraverso un nome naturalmente, però ha anche associato un insieme di **attributi** che vanno a specificare le caratteristiche di un file. Questi attributi sono:

- **Nome:** unica informazione in una forma leggibile dagli esseri umani.
- **Identificatore:** tag unico (spesso numerico) che identifica il file all'interno del file system.
- **Tipi:** nei sistemi che supportano differenti tipi di file.
- **Locazione:** la locazione può essere di due tipi:
 - *Posizione fisica:* puntatore alla locazione fisica del file nel dispositivo.
 - *Posizione logica:* il pathname del file, ossia la sequenza delle directory che bisogna percorrere per raggiungere il file (questa informazione NON è memorizzata esplicitamente da nessuna parte eccetto che in casi particolari).
- **Dimensione:** dimensione corrente del file.
- **Protezione:** determina chi può leggere, scrivere, eseguire.
- **Ora, data e identificativo dell'utente:** dati utili per la protezione, la sicurezza ed il monitoraggio d'uso.

Tutte queste informazioni appena elencate sono memorizzate insieme al file, naturalmente sul disco. Capiremo più avanti che gli attributi di un file andranno a formare il **File Control Block**. Vediamo ora quali sono le operazioni che il SO deve garantire sui file:

- **Creazione:** devono essere garantite delle system call che sono capaci di creare un file. Naturalmente il SO deve anche garantire che venga trovato dello spazio nel disco per la corretta creazione.
- **Scrittura / Lettura:** il SO mette a disposizione una opportuna system call per specificare il nome del file su cui operare. Il SO deve gestire il puntatore di scrittura/lettura al punto del file in cui si vuole scrivere/leggere (ad esempio una system call che ci permetta di posizionare il cursore in una qualsiasi posizione del file) e di occuparsi di trovare spazio sufficiente per ospitare l'eventuale espansione del file, in caso di scrittura.
- **Riposizionamento all'interno di un file:** per leggere o scrivere a partire dal punto specificato.
- **Cancellazione:** recupera lo spazio occupato dal file sul supporto di memoria secondaria e lo spazio occupato nella directory che lo conserva.
- **Troncamento:** cancella i dati memorizzati e recupera lo spazio occupato, ma mantiene tutti gli attributi del file.

Un processo, quando utilizza i file, deve tenere in uso una tabella, che viene chiamata **Tabella dei file aperti dal processo**: ogni processo, man mano che apre dei file per poterli utilizzare, annota in una tabella tutte le caratteristiche dei file che sono stati aperti. Questo descritto è relativo al singolo processo.

Il sistema, nella sua interezza deve avere anche lui una tabella dei file, che viene chiamata **Tabella dei file aperti**. Supponiamo che il Processo1 apre file1, file2 e file3 e che il Processo2 apre file4 e file5, nella Tabella dei file aperti ci stanno tutti e 5 questi file che sono stati aperti, in cui file1, file2 e file3 sono associati al Processo1 e file4 e file5 sono associati al Processo2.

2.1 FILE IN UNIX

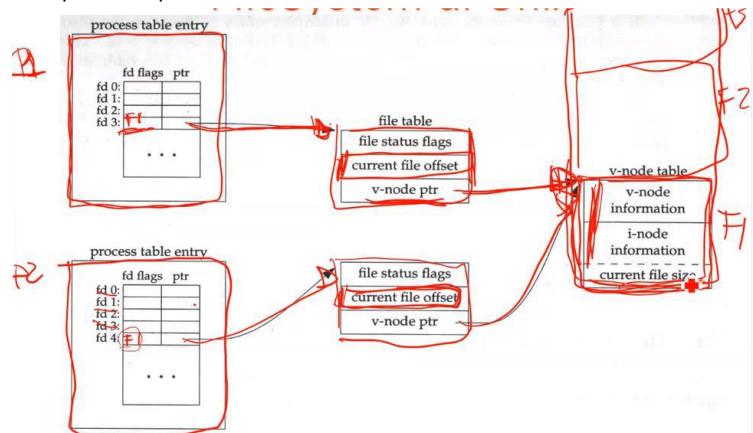
Dal punto di vista di Unix quello che succede quando 2 processi utilizzano lo stesso file è questo:

Supponiamo di eseguire un primo processo P1. Per quanto descritto fino ad ora, quello che succede è che viene messo a disposizione del processo P1 uno spazio in memoria principale. In questo spazio è sicuramente presente il codice del processo P1. Supponiamo che durante l'esecuzione di P1 si incominciano ad aprire dei file (file1, file2, ...), allora nello spazio messo a disposizione in memoria principale per P1, esisterà una tabella che è la tabella dei file aperti dal processo, e al suo interno ci sarà una entry per file1, file2, file3 e così via quanti file verranno aperti. Il SO anche lui nel suo spazio avrà una tabella di tutti i file aperti nel sistema e scriverà tutti i file che i processi apriranno.

Supponiamo di eseguire un nuovo processo P2, al quale verrà naturalmente dedicato dello spazio in memoria principale e avrà la propria tabella dei file aperti dal processo. Supponiamo che P2 apre file1 (lo stesso aperto da P1). Dal punto di vista del SO, poiché file1 è stato già salvato nella propria tabella (perché già P1 lo ha aperto precedentemente), dovrà solamente tenere in considerazione che da questo momento sia P1 che P2 utilizzeranno quel file (tutto ciò mediante contatore).

Osservando l'immagine, associamo al primo process table entry, la tabella dei file aperti dal processo di P1, e la seconda di P2.

P1 ha aperto, nell'immagine, 4 file e a ciascuno di questi file viene dato un numeretto locale alla tabella, che viene chiamato **File Descriptor**. P2 ha aperto, nell'immagine, 5 file e anch'esso ha associato ai file un file descriptor.



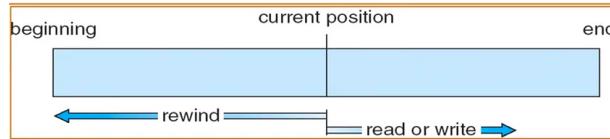
Unix, per ciascuno dei file aperti nel processo, ha una sorta di puntatore agli attributi generali del file (nell'immagine **file table**), per esempio **file status flags**, **current file offset**. All'interno degli attributi del file, c'è anche un **v-node ptr**, che rappresenta un puntatore alla **v-node table** che sarebbe la tabella dei file aperti dal sistema. Nell'immagine si vede che fd3 del processo P1 e fd4 del processo P2 hanno entrambi un puntatore v-node ptr allo stesso campo nella v-node table: questo perché contemporaneamente hanno aperto lo stesso file, cioè F1.

Quando si accede ad un file, ci possono essere due modalità di accesso:

- **Accesso sequenziale:** si accede ai byte presenti in un file uno dopo l'altro in modo sequenziale (un po' come viene fatto per le vecchie cassette).

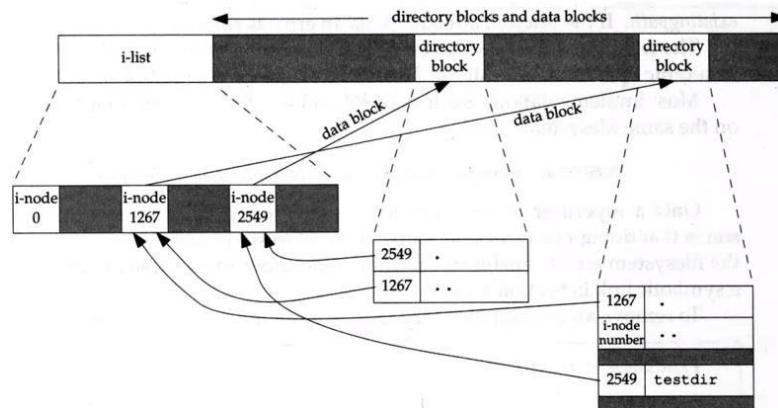
Non si può saltare in una posizione specifica del file, ma bisogna leggere tutti i byte precedenti fino a che non si raggiunge quello x.

- **Accesso diretto:** se l'utente vuole accedere al blocco x, l'accesso avviene direttamente, senza passare per i precedenti.



2.2 ORGANIZZAZIONE DI UN FILE SYSTEM IN UNIX

La struttura dell'organizzazione di un file system di Unix è la seguente:



Supponiamo che tutto il primo rettangolo sia lo spazio a disposizione messo sul disco per il file system. Il file system è partitionato nel seguente modo: c'è un **i-list** che rappresenta tutta la lista degli attributi dei file che vengono riferiti mediante i-node (campo presente nella v-node table visto precedentemente). L'i-node dunque contiene gli attributi del file (i-node 0 contiene gli attributi del file 0 e così via...).

In Unix, tutti gli attributi dei file vengono posizionati all'inizio del file system dunque. Fisicamente i file sono poi memorizzati all'interno di **directory block**, che come suggerisce il nome, sono delle directory che contengono al loro interno i file stessi. Ad esempio, nell'immagine, all'interno della prima directory block sono presenti due file identificati con 2549 e 1267 (che sono gli **i-node number**). Su 2549 è presente un solo puntino e questo indica la directory del file è la corrente, mentre per 1267 ci sono due punti è questo indica la directory padre.

Ogni file contenuto nelle directory è identificato mediante i-node number, e questi punteranno all'i-node contenuto nell'i-list, il quale contiene al suo interno tutti gli attributi di un file.

Organizzare logicamente i file in directory permette di ottenere:

- **Efficienza:** localizzazione rapida dei file.
- **Assegnazione di nomi conveniente per gli utenti:** due utenti possono scegliere lo stesso nome per file differenti.
- **Raggruppamento:** raggruppamento logico di file in base alle proprietà

2.3 STRUTTURE DIRECTORY

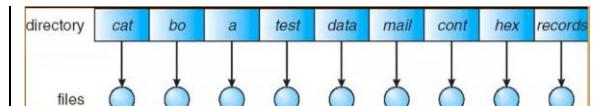
Partiamo dal caso di gestione banale fino ad arrivare a quelli più complessi ed utilizzati nei nostri sistemi:

DIRECTORY A SINGOLO LIVELLO:

Tutti i dati sono contenuti in una sola directory per tutti gli utenti. Di conseguenza non abbiamo la possibilità di creare altre. Tutti i file che abbiamo sono quindi aggregati.

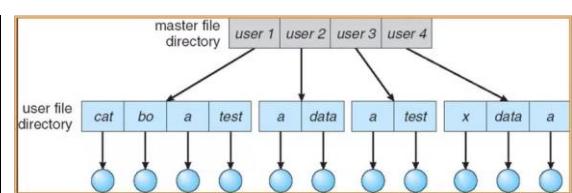
Banalmente, questa gestione "estrema" crea molti problemi perché siamo costretti ad utilizzare nomi diversi per ciascun file, non si possono differenziare se non per il nome. I file non possono essere raggruppati in maniera logica.

Non viene mai usato, viene mostrato solo per definizione.



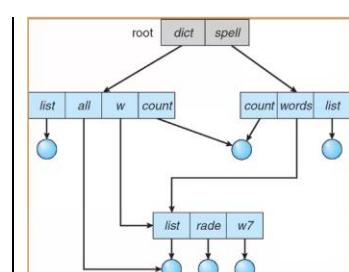
DIRECTORY A DUE LIVELLI:

In questo caso si crea un primo strato di directory (per esempio una per ciascun utente), e i file degli utenti vengono dunque separati. In questo caso c'è un minimo di miglioramento rispetto al caso precedente, è possibile applicare una ricerca efficiente.



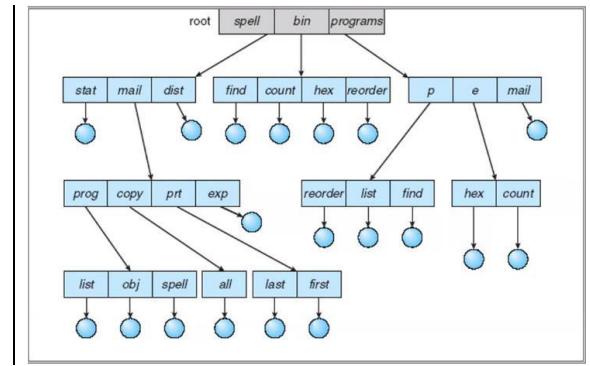
STRUTTURA A GRAFO ACICLICO:

Rappresenta un ulteriore avanzamento della struttura ad albero in cui ci sono file o sottodirectory condivise. Per esempio, osservando l'immagine, si consideri la directory list, questo si trova all'interno di tre directory. Viene implementata sotto UNIX con gli hard link.



STRUTTURA AD ALBERO:

Rappresenta la struttura a cui noi siamo abituati, in cui c'è una root iniziale dell'albero che ha le sue directory, ogni directory contiene i suoi file con eventuali sottodirectory poste all'interno. In questo modo la ricerca è efficiente, c'è alta capacità di raggruppamento, etc...



2.4 MONTAGGIO DI UN FILE SYSTEM

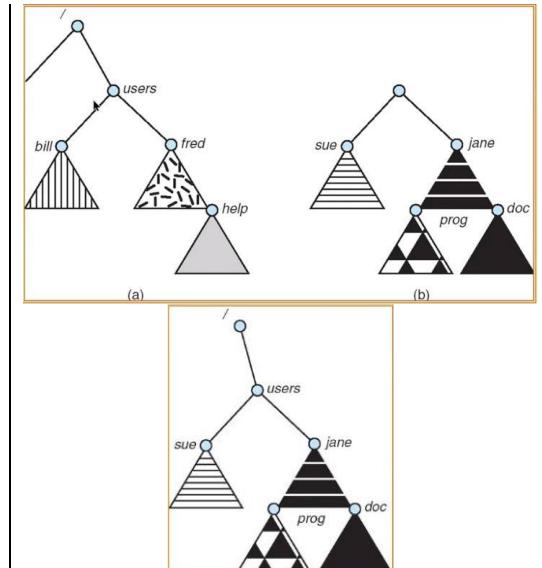
Avviene un montaggio di un file system quando si vuole agganciare un file system nuovo ad un file system esistente. Un po' come quando si inserisce una penna usb.

Sotto Windows, quando s'inserisce una penna usb, viene assegnata una porta con una lettera al dispositivo esterno (per esempio "E:"). In questo modo il file system che era presente prima dell'inserimento della penna usb, rimane staccato da quello che viene inserito esternamente.

Sotto UNIX, invece, la penna usb viene agganciata ad una delle directory che sono già presenti all'interno del file system. Il file system della penna usb verrà visto come parte integrante dell'intero file system del sistema operativo. In UNIX si ha la libera possibilità di scegliere la cartella sotto la quale agganciare il nuovo file system. A quel punto cosa succede però? Assumiamo di fare l'aggancio sotto ad una directory, ma se in quella directory era presente qualcosa (file, directory, ...), che fine fanno queste informazioni? Vengono oscurate. Non si vedranno più le informazioni che c'erano sotto quelle directory e vengono momentaneamente sostituite (non perse) da quelle del dispositivo esterno appena inserito. Nel momento in cui si rimuove il dispositivo esterno, si rivedranno le informazioni che erano state precedentemente oscurate.

Facciamo un esempio →

Osservano la figura (a), che rappresenta il file system esistente, mentre la figura (b) rappresenta il file system di una penna USB. Supponiamo di voler effettuare il mount di (b) all'interno della cartella users della figura (a). Quello che succede, come descritto precedentemente, è che il contenuto di users (bill, fred, help) viene temporaneamente oscurato e si vede sostituito da quello del file system della penna USB.



Il risultato è dunque il seguente →

Per proteggere file e directory, viene assegnato a ciascuno di essi un **User ID** e un **Group ID**.

I permessi di accesso di un file/directory sono: lettura, scrittura, esecuzione. Ci sono tre classi di utenti:

RWX

- Accesso proprietario 7 → 1 1 1
- Accesso gruppo 6 → 1 1 0
- Accesso pubblico 1 → 0 0 1

3.0 ORGANIZZAZIONE DEL FILE SYSTEM

Il File System rappresenta la definizione dell'aspetto del sistema agli occhi dell'utente, algoritmi e strutture dati che permettono di far corrispondere il file system logico ai dispositivi fisici. Il File System risiede in un'unità di memorizzazione secondaria (disco):

- Per **leggere** un blocco dal disco, lo si deve portare in memoria principale.
- Per **modificarlo**, lo si deve portare in memoria principale e dopo averlo modificato, lo si riscrive nella stessa posizione del disco.
- Al blocco ci si può accedere con accesso diretto, muovendo la testina di lettura.

In generale, il file system è organizzato a strati, che segue questa linea:

Supponiamo che il file che stiamo cercando si trovi sulla **devices**, per poter accedere al dispositivo fisico, c'è un **controllo dell'input-output** che si occupa del trasferimento delle informazioni dal disco alla memoria principale e viceversa.

Una volta che il file entra nella memoria principale, il **file system di base** da i comandi al driver, cioè quello che dispone la necessità di leggere o scrivere blocchi fisici nel disco.

Il **modulo di organizzazione dei file** traduce gli indirizzi dei blocchi logici in quelli dei blocchi fisici. Gestisce lo spazio libero, cioè, quando si vuole leggere il contenuto di un file (il quarto byte precisamente), allora il modulo di organizzazione dei file si deve porre il problema di capire dove fisicamente si trova il quarto byte.

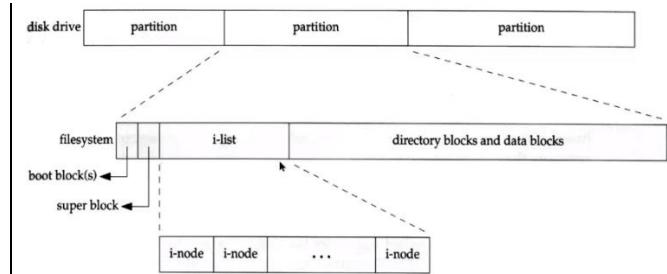
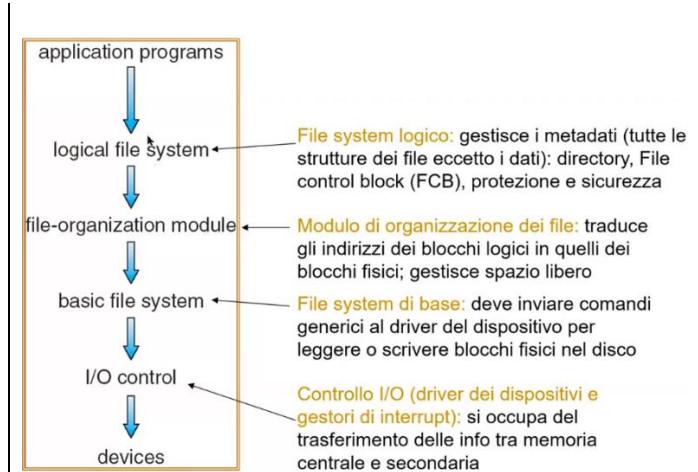
Tutto questo viene gestito grazie al **File system logico** che gestisce i metadati (tutte le strutture dei file eccetto i dati): directory, File control block (FCB), protezione e sicurezza.

Sul disco, sono presenti le seguenti strutture:

- **Boot control block**: informazioni necessarie per il caricamento del sistema operativo (*UFS – boot block, NTFS – partition boot sector*).
- **Volume control block**: contiene dettagli del volume, numero di blocchi per partizione, taglia dei blocchi, blocchi liberi ... (*UFS, NTFS*).
- **Struttura delle directory**: usata per organizzare i file ... (*UFS – nomi file e i-node associato, NTFS – master file table*).
- **File control block (FCB)**: informazioni sul file (*UFS – inode*).

Dopo aver esposto le precedenti informazioni, possiamo ora vedere in più dettaglio quella che è l'organizzazione di un file system in UNIX:

Abbiamo detto che c'è l'i-list, seguito dalla zona effettiva in cui sono immagazzinati i dati. Quello che prima non avevamo visto, è che prima dell'i-list ci sono altre due informazioni che vengono salvate: la prima, il **boot block(s)** in cui vengono caricate le informazioni principali del SO, mentre il **super block** che tiene conto dei dettagli della partizione, del volume che stiamo considerando.



3.1 FILE CONTROL BLOCK

L'insieme degli attributi di un file/directory prende il nome di **File control block** e contiene diverse informazioni, quali i permessi del file, le date del file (creazione, accesso, scrittura), il proprietario, la size e i puntatori ai vari blocchi del file. È quello che in UNIX è detto i-node.

Quando si effettua una richiesta di **open** e **read** di un file, accade ciò:

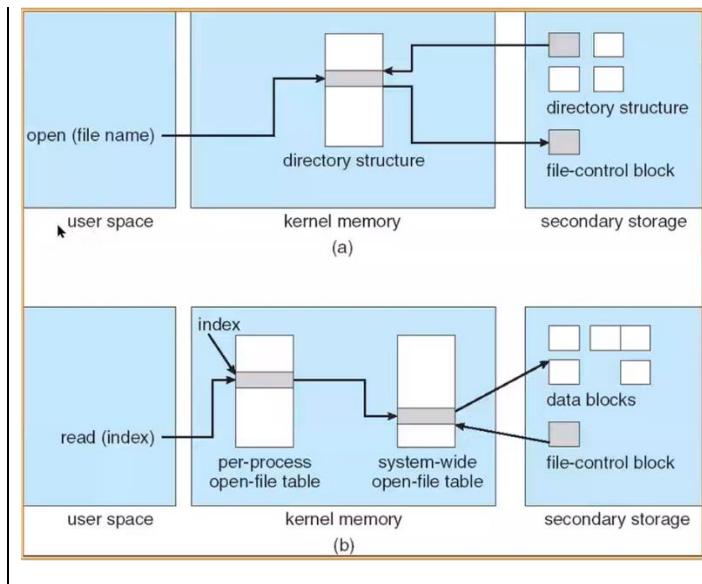
Assumiamo che il nostro programma in esecuzione faccia la *open* di un file che nominiamo *file1*. Quando all'interno del campo *file name* della funzione *open* scriviamo *file1*, oltre a specificare il file che vogliamo aprire, andiamo anche a specificare qual è la directory all'interno della quale si trova *file1*. Se non mettiamo la directory, ma scriviamo solo *file1*, stiamo cercando il suddetto file all'interno della directory corrente, ossia quella in cui è stato salvato il codice.

La prima cosa che il sistema fa quando viene invocata la *open*, si mette in kernel mode, va a recuperare la directory in questione (si prende il file control block della directory, in cui c'è la lista dei file all'interno della directory, e se lo porta in memoria principale). Nella struttura della directory cerca *file1* secondo un criterio. Se non lo trova, la funzione *open* ritorna un errore, se invece è presente, la directory structure va a recuperare il **file control block** di *file1* e lo porta in memoria principale.

Successivamente, all'interno della tabella dei file aperti dal sistema presente in memoria principale, si sarà creata un'entry contenente le informazioni di *file1*. Verrà creata anche un'entry nella tabella dei file aperti dal processo, a cui verrà assegnato un file descriptor che corrisponde al valore di ritorno della funzione *open* (rispettivamente le due tabelle nella figura b in memoria principale).

Una volta aperto il file, scorrendo il codice, c'è un'operazione di *read*. Da questo momento in poi, il riferimento al file non avviene più con il suo nome, bensì attraverso il file descriptor. Per la *read* a cui si passa in input il file descriptor, si entra in kernel mode, si va nella posizione relativa al file descriptor di *file1*, all'interno di quest'ultima c'è un campo che punta al riferimento del file nella tabella dei file aperti dal sistema. Si entra nella tabella dei file aperti del sistema al cui interno ci sarà il **File control block** del file di nostro interesse. Tra le cose che ci interessano ci saranno i puntatori ai blocchi di dati veri e propri che ci serviranno per poter leggere il file.

Quando si effettua la *open* di *file1* e si cerca il file all'interno della directory structure, la ricerca può essere fatta in maniera intelligente: per esempio, tenendo una lista ordinata e si effettua una ricerca, oppure organizzare i file contenuti nella directory in una tabella Hash e così via... L'idea è di non cercare sequenzialmente il file altrimenti si perderebbe molto tempo inutilmente.



3.2 METODI DI ALLOCAZIONE

Ci sono 3 tipi di allocazioni standard che ci consentono di effettuare la traduzione da indirizzo logico ad indirizzo fisico: **contigua, linkata e indicizzata**.

3.2.1 ALLOCAZIONE CONTIGUA:

Ogni file occupa un certo numero di blocchi contigui (uno dietro l'altro) sul disco. È un'allocazione facile, in quanto una volta che sappiamo dove si trova il primo blocco di un certo file, so facilmente individuare per esempio qual è il quarto blocco del suddetto file. L'accesso può essere di due tipi:

- **Accesso sequenziale**: se per esempio si vuole accedere al decimo blocco di un certo file, quello che si fa è che si legge prima il primo blocco, poi il secondo, ..., fino ad arrivare al decimo.
- **Accesso diretto**: se si vuole accedere al decimo blocco di un certo file, l'accesso è diretto, senza passare per i primi 9.

Nel **File Control Block** di **allocazione contigua**, le informazioni che ci interessano sono: **blocco iniziale, lunghezza del file in blocchi**.

Supponiamo di avere a disposizione il disco nell'immagine, in cui sono allocati 3 file posizionati in una directory: file1, programma.c e documento.

Il **File control block** della directory ci dà le informazioni riguardo ai blocchi dei file (sappiamo che il file1 ha come blocco iniziale 0 e la lunghezza è di 4 blocchi), inoltre abbiamo a disposizione un singolo puntatore che punta al blocco iniziale. Naturalmente questo accade perché, sapendo che l'allocazione è contigua, non si ha la necessità di dover avere puntatori a tutti i blocchi di un file, ma basta avere solo quello che punta al primo blocco.

Immaginiamo di esaminare programma.c, sappiamo che comincia al blocco 4 e la lunghezza del file è di 8 blocchi. Se vogliamo accedere al terzo byte del file, tenendo conto che parliamo di allocazione contigua, come facciamo a sapere qual è il blocco che contiene al suo interno il byte 3 del file? La prima cosa che dobbiamo sapere è quanto è grande un blocco (per convenzione supponiamo che ogni blocco è grande 512 byte).

Il calcolo che dobbiamo fare è il seguente:

LA/512 dove LA indica il **Logical Address**, ossia il byte che vogliamo cercare (nel nostro esempio quattro). Quando effettuiamo la divisione, avremo a disposizione un quoziente ed un resto.

Il **quoziente** ci dirà qual è il blocco che contiene il byte di interesse.

Il **resto** ci dirà lo spiazzamento all'interno del blocco, all'interno del quale si trova il byte che stiamo cercando.

ESEMPIO:

La numerazione dei byte dei file parte da 0 e la numerazione dei byte all'interno di ogni blocco parte da 0.

$3/512 = 0$ con resto 3 → significa che il byte 3 è presente nel primo blocco e all'interno del blocco il byte che cerchiamo sta nella posizione 3 (che corrisponde al quarto elemento del vettore),

$520/512 = 1$ con resto 8 → significa che il byte 520 è presente nel secondo blocco (tenendo conto dell'immagine precedente devo effettuare $4+1=5$ blocco, in quanto programma.c parte dal quarto blocco) e all'interno del blocco il byte che cerchiamo sta nella posizione 8, di conseguenza, tenendo conto che la numerazione all'interno di ogni blocco comincia da 0, arrivo all'ottavo (che è in posizione 9) e quello è il byte che mi interessa.

Dal punto di vista del disco, tenendo conto che il file programma.c comincia dal blocco 4, allora dobbiamo effettuare il seguente calcolo: $4 + Q$ che ci indicherà il blocco effettivo all'interno del quale si trova un certo byte che stiamo cercando.

Gli **svantaggi** dell'allocazione contigua sono:

- **Frammentazione esterna**: tenendo conto dell'immagine precedente, supponiamo che si cancellano ed aggiungono file. Quello che può succedere è che si formano dei buchi, per esempio rimangono 2 blocchi contigui liberi da una parte, altri 2 in un'altra parte e un altro in un'altra parte del disco. Se si riceve una richiesta di un file che richiede 5 blocchi, anche se effettivamente abbiamo 5 blocchi liberi, questi non sono contigui, e di conseguenza il file che richiede 5 blocchi non può essere salvato nel disco.
- **La taglia dei file non può crescere**: supponiamo di prendere come riferimento l'immagine precedente e di focalizzarci su *programma.c* che va dal blocco 4 al blocco 11. Se per qualche motivo si vuole continuare a scrivere all'interno di questo file, questa è una operazione che ci viene vietata in quanto il blocco successivo (ossia il 12) è occupato da *documento*.

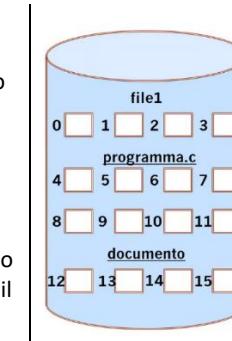
Ci sono alcuni sistemi che usano uno schema di allocazione contigua modificato. Ogniqualvolta non si riesce a far crescere un file in maniera contigua, si cerca una serie di blocchi contigui liberi e si partiziona il file in una serie di blocchi contigui che si trovano sul disco. Naturalmente nel **File Control Block** dobbiamo tenere traccia di dove si ferma un file e da quale blocco riparte. È una tecnica un po' complessa che ha avuto breve vita.

3.2.2 ALLOCAZIONE LINKATA

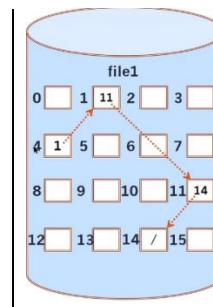
Con l'allocazione linkata si risolvono i problemi dell'allocazione contigua (frammentazione esterna). Per ogni blocco del disco, si consuma una parte di ogni blocco per contenere un campo puntatore. Di conseguenza, ogni blocco non verrà utilizzato per contenere totalmente dati, ma una piccola parte dovrà essere consumata da un puntatore. Naturalmente il puntatore punterà al blocco successivo presente sul disco.

Nel **File Control Block** di **allocazione linkata**, le informazioni che ci interessano sono: **blocco iniziale, blocco finale oppure lunghezza in blocchi (quanti blocchi sono richiesti da un file)**, quest'ultima più usata.

Supponiamo che *MyFileConcatenato* comincia dal blocco numero 4. All'interno del blocco numero 4 ci sono ovviamente le informazioni del file, e ci sarà anche un riferimento al blocco successivo, che in questo caso è il blocco 1. Stesso discorso vale per i blocchi successivi del file. Nel caso in cui, in un secondo momento, si ha la necessità di voler ampliare *MyFileConcatenato*, si cerca un blocco libero sul disco e si va puntare l'ultimo blocco del file a questo nuovo blocco. L'ultimo blocco naturalmente punterà a NULL (/).



File	Blocco iniziale	Lunghezza (in blocchi)
file1	0	4
programma.c	4	8
documento	12	4



File	Blocco iniziale	Blocco finale
MyFileConcatenato	4	14

Vediamo ora come avviene la conversione da indirizzo logico ad indirizzo fisico:

Si tenga conto che per la numerazione dei blocchi, il primo byte, quello che contiene il puntatore non viene considerato. La numerazione parte dal successivo (quindi da 0 a 510).

LA/511 dove LA indica il **Logical Address**, ossia il byte che vogliamo cercare. Quando effettuiamo la divisione, naturalmente avremo a disposizione un quoziente ed un resto. Si osserva che in questo caso il divisore è 511 in quanto stiamo assumendo che il blocco sia di 512 byte di cui 1 byte è occupato dal puntatore al prossimo blocco. Il quoziente ci dirà qual è il blocco che contiene il byte di interesse. Il resto ci dirà lo spiazzamento all'interno del blocco, all'interno del quale si trova il byte che stiamo cercando.

ESEMPIO:

$520/511 = 1$ con resto 9 → ciò significa che il byte 520 è presente nel secondo blocco e all'interno del blocco il byte che stiamo cercando sta alla posizione 9 (10 elemento del vettore non tenendo conto del campo puntatore).

Gli svantaggi dell'allocazione linkata sono:

- **Gestione difficoltosa dei puntatori:** se viene perso un puntatore, non ci si può più riferire al continuo del file.
- **Tempi di accesso:** si è obbligati ad accedere sequenzialmente ad ogni blocco del file, causando continui accessi al disco (in quanto nel File Control Block abbiamo a disposizione solo il riferimento al primo blocco).
- **Consumo di memoria:** per ogni blocco siamo costretti a perdere una quantità definita di byte per il campo puntatore.

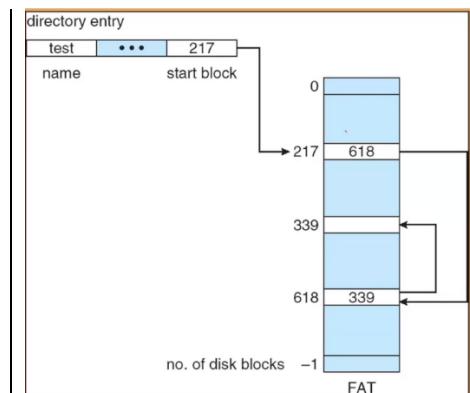
3.2.3 TABELLA DI ALLOCAZIONE DEI FILE (FAT)

FAT sta per **File Allocation Table** e funziona nel seguente modo:

Supponiamo che nel File Control Block del nostro file si trovi semplicemente un numeretto (**start block**) che rappresenta il riferimento all'interno della FAT dentro cui è contenuto il primo blocco del nostro file. La FAT è un array che ha tante entry quanti sono i blocchi utilizzabili del nostro disco.

All'interno delle entry dell'array c'è l'indice del successivo blocco di un certo file. Per esempio, tenendo conto del file *test*, nel suo File Control Block all'interno del campo start block c'è il numero 217. Per sapere qual è il blocco successivo, si va all'interno della FAT in corrispondenza dell'elemento con indice 217, si osserva il valore al suo interno che in questo caso è 618, il quale rappresenta quindi il blocco successivo. 618 a sua volta contiene 339 che è il terzo blocco del file. Quando si arriva ad un elemento del vettore che al suo interno contiene il valore NULL allora tale blocco rappresenta l'ultimo di un certo file.

La FAT risiede normalmente sul disco, quando avviene la formattazione di una periferica con modalità FAT, una parte della periferica viene utilizzata per immagazzinare dati e una parte verrà usata per servizio.



Quando viene montato il File System (per esempio quando si mette una penna USB) formattato FAT 32, succede che la tabella FAT viene ricopiata dal disco alla memoria principale. Di conseguenza, in memoria principale abbiamo da subito a disposizione tutte le informazioni sui file contenuti sul disco. Quando per esempio apriamo il file *test* in didascalia, prendiamo il suo File Control Block e lo portiamo in memoria principale, e se per esempio vogliamo accedere al secondo blocco di questo file, questa operazione viene fatta immediatamente dalla memoria in quanto, come detto in precedenza, abbiamo da subito a disposizione tutta la tabella della FAT in memoria principale. Una volta che sappiamo che il secondo blocco è 618 andremo poi sul disco e lo preleveremo. Abbiamo fatto quindi un solo accesso al disco (se fosse stata l'allocazione linkata avremmo dovuto fare 2 accessi a disco).

Lo **svantaggio** della FAT è dovuto alla grandezza, in quanto già di suo la FAT occupa tanto spazio all'interno del disco e nella memoria principale. Se consideriamo FAT 32, questo 32 rappresenta la lunghezza in bit dell'indirizzo, di conseguenza 4 byte vengono utilizzati per l'indirizzo per individuare ciascuno dei blocchi. Il numero di entry della tabella FAT è 2^{32} e la grandezza di ogni entry della tabella è 4 byte, di conseguenza la grandezza della tabella FAT è $4 \cdot 2^{32} = 2^{34}$ che sarebbero 16 GB. 16 GB vengono quindi occupati solamente per la FAT.

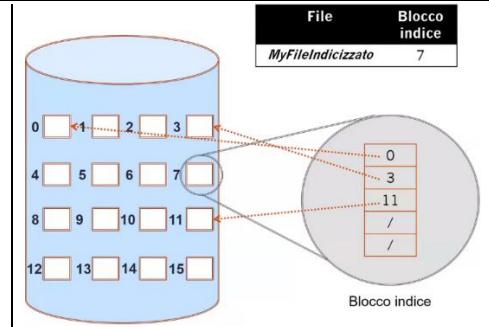
3.2.4 ALLOCAZIONE INDICIZZATA

In questo tipo di allocazione si prende uno dei blocchi che si assegna ad un file, e all'interno di questo blocco, anziché mettere i dati, lo utilizziamo per mettere gli indici di tutti i blocchi utilizzati per un certo file. Lo possiamo vedere come blocco di servizio.

Possiamo dunque vederlo come raffigurato nell'immagine. Supponiamo di avere un file chiamato *MyFileIndicizzato*. All'interno del File Control Block abbiamo a disposizione il **blocco indice** ossia il riferimento al blocco che contiene al suo interno tutti gli indici del file. Il blocco 7 lo possiamo quindi vedere come un array contenente in modo ordinato tutti i blocchi del file.

Uno svantaggio di questo tipo di allocazione è che per ogni file, dobbiamo perdere un blocco, che dovrà contenere gli indici dei blocchi che costituiscono un file. Se ad esempio un file è costituito da un solo blocco, dovremo avere comunque a disposizione un ulteriore blocco indice.

Vediamo come avviene il mapping da indirizzo logico ad indirizzo fisico:



LA/512 dove LA indica il **Logical Address**, ossia il byte che vogliamo cercare (nel nostro esempio quattro). Quando effettuiamo la divisione, naturalmente avremo a disposizione un quoziente ed un resto. Il quoziente ci dirà qual è il blocco che contiene il byte di interesse.

Il resto ci dirà lo spiazzamento all'interno del blocco, all'interno del quale si trova il byte che stiamo cercando.

ESEMPIO:

La numerazione dei byte dei file parte da 0 e la numerazione dei byte all'interno di ogni blocco parte da 0

$520/512 = 1$ con resto 8 → significa che il byte 520 è presente nel secondo blocco (tenendo conto dell'immagine precedente devo entrare nel blocco con indice 3) e all'interno del blocco il byte che cerchiamo sta nella posizione 8, di conseguenza, tenendo conto che la numerazione all'interno di ogni blocco comincia da 0, arrivo all'ottavo (che è in posizione 9) e quello è il byte che mi interessa. Il blocco indice lo si porta una sola volta dal disco alla memoria principale.

Un altro **svantaggio** dell'allocazione indicizzata si ha nel momento in cui il file è molto grande, infatti se è molto grande lo spazio contenuto all'interno di un blocco indice non è sufficiente per immagazzinare tutti gli indici dei blocchi che costituiscono un file. Si rimedia a tutto ciò con l'allocazione indicizzata a schema concatenato.

ALLOCAZIONE INDICIZZATA – SCHEMA CONCATENATO:

Supponiamo dunque che un singolo blocco indice non sia sufficiente ad immagazzinare tutti gli indici dei blocchi che costituiscono un file. Quello che si fa in questo caso, è di prendere l'ultimo indice del primo blocco indice e, anziché contenere al suo interno un riferimento ad un blocco del file sul disco, esso conterrà un puntatore ad un ulteriore blocco indice. Se il secondo blocco indice è sufficiente a contenere tutti gli indici dei blocchi del file sul disco, allora l'ultimo indice di tale blocco verrà contrassegnato con "/" che indica un valore NULL. Per essere ancora più chiari, se il primo blocco indice si chiama a e il secondo blocco indice si chiama b allora all'interno dell'indice 511 del blocco a , ci sarà scritto b che rappresenta il puntatore al blocco b .

In questa modalità, la conversione da indirizzo logico ad indirizzo fisico risulta essere un po' più complesso, scelto il **LA** di interesse, si effettuano questi calcoli:

$LA/(511 \times 512)$, la seguente divisione ci fornirà un quoziente (Q_1) e un resto (R_1). Come divisore consideriamo (511×512) perché ogni blocco indice mette a disposizione 511 entry contenenti riferimenti a blocchi del file (in quanto 1 indice è perso per il campo puntatore al successivo blocco indice), mentre ogni blocco contenente dati di file contiene al suo interno 512 byte → il numero di byte che posso individuare attraverso un unico blocco indice sono proprio (511×512) .

Preso Q_1 , bisogna addizionargli 1 per individuare qual è il blocco indice che dobbiamo considerare.

R_1 viene invece usato per un ulteriore calcolo:

$R_1/512$, la seguente divisione ci fornirà un ulteriore quoziente (Q_2) e un resto (R_2).

Q_2 rappresenta l'indice da considerare nel blocco della tabella indice individuato in precedenza.

R_2 rappresenta l'indice nel blocco del file.

ESEMPIO1:

Scelto **LA**=200, si effettuano i seguenti calcoli:

$200/(511 \times 512)$, $Q_1=0$, $R_1=200$.

Questo significa che dobbiamo considerare il primo blocco indice (in quanto $Q_1=0 + 1= 1$). Adesso sfruttiamo R_1 :

$200/512$, $Q_2=0$, $R_2=200$.

Questo significa che all'interno del primo blocco indice individuato mediante Q_1 dobbiamo considerare il primo indice (in quanto $Q_2=0$) e all'interno del blocco del file considerare il 200 byte contenuto in esso.

ESEMPIO2:

Scelto **LA**=300000, si effettuano i seguenti calcoli:

$300000/(511 \times 512)$, $Q_1=1$, $R_1=38368$

Questo significa che dobbiamo considerare il secondo blocco indice (in quanto $Q_1= 1 + 1= 2$). Adesso sfruttiamo R_1 :

$38368/512$, $Q_2=74$, $R_2=480$

Questo significa che all'interno del secondo blocco indice individuato mediante Q_1 dobbiamo considerare l'indice 74 (in quanto $Q_2=74$) e all'interno del blocco del file considerare il 480 byte contenuto in esso.

ALLOCAZIONE INDICIZZATA A 2 LIVELLI:

In questo caso c'è un blocco indice esterno e dei blocchi indici interni. Il blocco indice esterno contiene 512 entry ad ulteriori blocchi indici (interni). Ogni blocco indice interno avrà 512 entry in cui ogni indice conterrà un riferimento al blocco del file contenuto nel disco.

In questo caso, il numero di blocchi massimo del file sono 512×512 , mentre il numero di byte massimi del file sono $512 \times 512 \times 512$.

Vediamo come avviene la conversione da indirizzo logico ad indirizzo fisico:

Scelto il **LA** di interesse si effettuano i seguenti calcoli:

$LA/(512 \times 512)$, la seguente divisione ci fornirà un quoziente (Q_1) e un resto (R_1).

Q_1 ci indica quale indice all'interno del blocco indice esterno dobbiamo considerare.

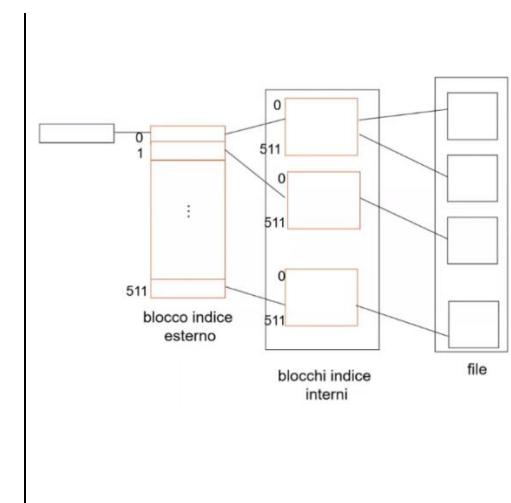
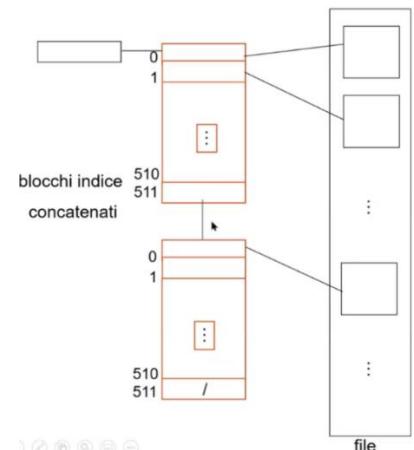
Preso Q_1 , bisogna addizionargli 1 per individuare qual è il blocco indice che dobbiamo considerare.

R_1 viene invece usato per un ulteriore calcolo:

$R_1/512$, la seguente divisione ci fornirà un ulteriore quoziente (Q_2) e un resto (R_2).

Q_2 rappresenta l'indice da considerare nel blocco indice interno individuato in precedenza.

R_2 rappresenta l'indice nel blocco del file.



3.2.5 SCHEMA COMBINATO: UNIX (4K BYTE PER BLOCCO)

UNIX propone un mix di tutte le allocazioni che abbiamo analizzato precedentemente. Ciò viene fatto per massimizzare la grandezza del file da gestire e per garantire sicurezza e velocità di accesso.

Come mostrato nell'immagine, la parte blu costituisce il FCB di UNIX, cioè l'i-node, che contiene varie informazioni del file. Successivamente, la parte grigia, contiene le informazioni per l'accesso al file. In questa zona ci sono 15 puntatori gestiti nel seguente modo:

- 12 puntatori sono diretti a blocchi (si può vedere come un'allocazione indicizzata).
- 3 puntatori a blocchi indiretti:
 - Il primo punta ad un blocco indiretto.
 - Il secondo punta ad un blocco indiretto doppio (2 strati di blocchi indice).
 - Il terzo punta ad un blocco indiretto triplo (3 strati di blocchi indice).

Assumiamo di avere a disposizione un file F, il cui numero di blocchi è minore o uguale a 12. In questo caso l'accesso ai dati è velocissimo, in quanto vengono sfruttati solo i blocchi contenuti nel **direct blocks**.

Supponiamo che il file F abbia un numero di blocchi maggiore o uguale a 13. In questo caso ad una parte del file si accederà in maniera diretta, mentre l'altra parte attraverso un accesso a disco utilizzando **single indirect**.

Sostanzialmente più è grande il file, più tempo ci vorrà ad accedere ad una parte del file.

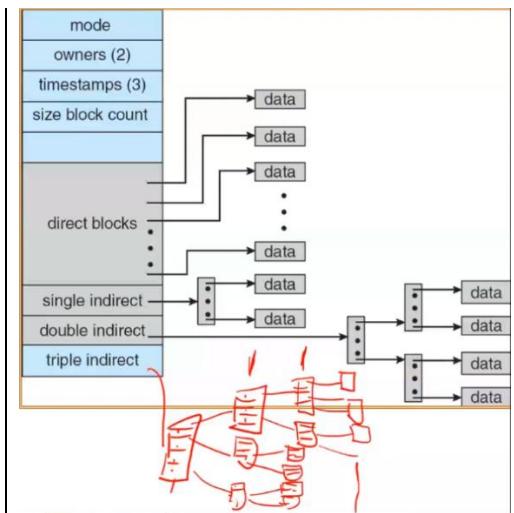
Questa organizzazione consente di immagazzinare file veramente molto grandi.

Effettuiamo alcuni conti:

Sappiamo che la size di un blocco (data nell'immagine) sia $4kb = 4 \times 2^{10} = 2^{12}$ byte. Adesso bisogna individuare quanto ci costa un campo puntatore all'interno di un blocco indice: tenendo conto che la size di un blocco è 2^{12} e che supponiamo la size di un puntatore sia 4 byte. Allora il numero di puntatori che possono essere messi all'interno di un blocco indice è $2^{12} b / 2^2 b = 2^{10}$.

La size di un file con questo schema può essere al più grande:

$$|F| \leq 12 \times \text{size blocco} + 2^{10} \times \text{size blocco} + 2^{10} \times 2^{10} \times \text{size blocco} + 2^{10} \times 2^{10} \times 2^{10} \times \text{size blocco} = \\ \leq 4kb(12 + 2^{10} + 2^{20} + 2^{30})$$



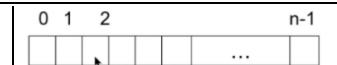
3.3 GESTIONE DELLO SPAZIO LIBERO

Abbiamo visto come poter allocare un file sul disco in vari modi. Nel caso in cui ci sia la necessità di dover far crescere un file c'è bisogno di andare ad individuare un nuovo blocco libero, a seconda di come viene gestita l'allocazione.

Qualunque sia l'allocazione utilizzata bisogna sempre avere delle informazioni sui blocchi liberi. Sapere dove sono per poterli rintracciare. Ci sono diverse tecniche per la gestione dello spazio libero:

3.3.1 GESTIONE DELLO SPAZIO LIBERO – VETTORE DI BIT

Abbiamo un vettore con n bit numerati da 0 a n-1. Se all'interno di un indice del vettore c'è uno 0 allora significa che il blocco[i]-esimo è occupato. Se invece c'è un 1, allora tale blocco è libero.



Questo vettore di bit richiede uno spazio extra: ad esempio si supponga che la dimensione di un blocco sia di 2^{12} byte, mentre la dimensione del disco è 2^{30} byte (1 gigabyte). Da queste due informazioni possiamo ricavare n, ossia dei blocchi che possibile inserire che è uguale a $2^{30}/2^{12}=2^{18}$. Il nostro vettore di bit, di conseguenza, avrà 2^{18} bit che sono occupati solamente per la gestione dei blocchi liberi. Questo rappresenta uno svantaggio.

3.3.2 GESTIONE DELLO SPAZIO LIBERO – LISTA DEI BLOCCHI LIBERI

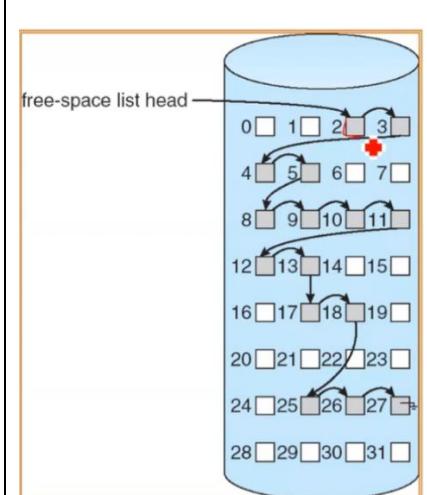
Lo spazio libero viene gestito mediante una lista a puntatore in cui i blocchi liberi vengono puntati uno dopo l'altro.

Si conserva il puntatore al primo blocco (e solo a questo) in una locazione speciale del disco che viene caricato in memoria quando si vuole accedere ad un blocco libero. In questo modo non c'è spreco di spazio.

Nel caso in cui si richiede un blocco libero, quello che si fa è prendere il blocco 2 in quanto è il primo libero, si porta in memoria principale, si legge qual è il successivo blocco libero (in questo caso il 3), si scrive il 3 nella locazione speciale contenuta all'interno della memoria, in modo tale che da questo momento il blocco 2 è libero e può essere usato e il blocco 3 diventa primo blocco libero della lista dei blocchi liberi.

Altri metodi usati per la gestione dello spazio libero sono:

- **Raggruppamento:** si memorizzano in un blocco gli indici di n blocchi: di questi i primi n-1 sono realmente liberi mentre l'ultimo contiene gli indirizzi di altri n blocchi, e così via. In questo modo si permette di trovare rapidamente molti blocchi liberi.
- **Conteggio:** spesso più blocchi contigui possono essere rilasciati. Quindi ogni elemento della lista dello spazio libero è formato da un indirizzo ed un contatore: l'indirizzo punta al primo dei blocchi liberi mentre il contatore dice quanti blocchi contigui ci sono.



4.0 DISCO MAGNETICO

Per memoria di massa intendiamo la memoria secondaria la quale può essere associata ai vecchi nastri magnetici che possono contenere grosse quantità di dati, poco usato al giorno d'oggi. Il problema del nastro il tempo di accesso è sequenziale, di conseguenza si perde molto tempo.

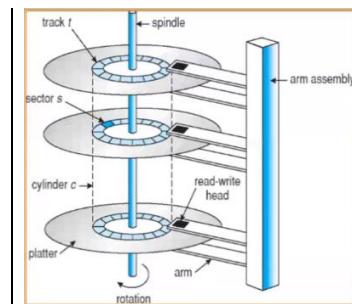
Un disco magnetico è fatto da una serie di piatti impilati uno nell'altro che ruotano attorno ad un perno centrale e ognuno dei piatti è suddiviso in tracce che sarebbero i cerchi concentrici in didascalia ed ogni traccia è divisa a sua volta in settori. La lettura da un settore avviene attraverso una testina di lettura che si muove lungo il raggio dei piatti.

Quando per esempio si deve prelevare un blocco la prima cosa che bisogna fare è andare a muovere il braccio lungo il raggio per andare a mettersi nella posizione relativa alla traccia dove è presente il settore che ci interessa. Dopodiché bisogna aspettare il tempo di rotazione in maniera tale che il settore di interesse vada a finire esattamente sotto la testina e dopo che questo è avvenuto c'è la caduta della testina e dunque avviene la lettura.

La cosa che pesa di più quando bisogna andare a cercare un dato è il movimento del braccio lungo il raggio.

Altre caratteristiche sono:

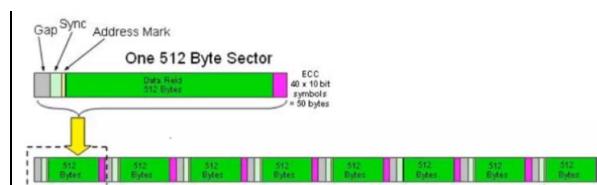
- I piatti del disco ruotano da 60 a 200 volte al secondo.
- **Velocità di trasferimento:** è la velocità con cui i dati vengono trasferiti dal disco al computer.
- **Tempo di posizionamento (o tempo di accesso):** è il tempo per muovere la testina sul settore desiderato (tempo di ricerca + latenza di rotazione).



In generale il disco viene visto come un grosso array monodimensionale di blocchi, malgrado la sua struttura. È importante capire come vengono considerati i settori nelle tracce. Ci sono due tipi di politiche a riguardo: ci sono i dispositivi **CVL** (constant linear velocity) in cui la densità dei bit per traccia è uniforme, cioè le tracce esterne contengono più settori. Poi ci sono i dispositivi **CAV** (constant angular velocity) in cui la densità di bit decresce dalle tracce interne verso quelle esterne.

Lo spazio all'interno di un settore è gestito nel seguente modo: inizialmente c'è una serie di informazioni relative al settore, per esempio il numero del settore in analisi.

Successivamente c'è la zona dati che viene utilizzata per immagazzinare dati. Una parte importante nel settore è quella che in figura è rappresentata come **ECC** che è l>Error Correcting Code che serve a verificare che la zona dati interessata dal settore non abbia subito errori o problemi.



Quando si memorizzano dei dati all'interno del settori, viene calcolato un numero che viene depositato all'interno dell'ECC. Questo serve per esempio quando all'interno di questo blocco vengono effettuate operazioni di lettura/scrittura. Viene ricalcolato questo numeretto e viene confrontato con quello precedentemente calcolato: se sono diversi sorge il dubbio che c'è stata qualche anomalia e si avverte il SO. Questo numeretto è una sorta di codice di sicurezza.

La prima cosa che il SO fa con un disco è quella di formattarlo (crea i settori e le numerazioni etc...). Dopo la creazione delle partizioni occorre formellarle logicamente. Per esempio, il sistema operativo ha bisogno di registrare su disco le partizioni e le corrispondenti directory, nonché le proprie strutture dati per la gestione dello spazio non allocato (creazione di un file system).

Per la gestione del disco è importante considerare la gestione dei blocchi difettosi. Quello che può succedere è che man mano che si utilizza il disco, si possono identificare una serie di blocchi che sono difettosi (per difettoso s'intende un problema di natura fisica). Questi blocchi naturalmente non possono essere utilizzati. In questi casi ci sono varie tecniche:

- **Sector sparing** (accantonamento dei settori): il controllore accantona i blocchi difettosi e li sostituisce con dei blocchi di riserva. Quando si formatta un disco, quello che succede è che non tutti i blocchi vengono messi a servizio dei dati. Una parte viene accantonata come settore di riserva. Laddove ci sono poi dei blocchi difettosi, si usano poi questi blocchi di riserva. Se si fa richiesta di scrittura ad un blocco difettoso, il controllore traduce la richiesta al blocco di riserva associato a quello difettoso.
- **Sector slipping** (traslazione dei settori): non c'è un vero e proprio settore di riserva contenente tutti i blocchi di riserva, bensì questi sono posizionati in posizioni casuali del disco. In questo caso tutti i settori compresi tra il settore danneggiato e quello di riserva immediatamente successivo, i dati, vengono spostati avanti di un settore.

4.1 SCHEDULING DEL DISCO

Vediamo ora come il driver del disco gestisce le richieste che arrivano al disco stesso. Supponiamo che nella macchina stiano girando una serie di processi e si stanno usando vari file. Al driver del SO possono arrivate ad un certo istante una serie di richieste di blocchi (ad esempio, il processo 1 richiede il blocco 10, il processo 2 il blocco 15 e così via). Il driver del SO deve gestire la richiesta di questi blocchi. Lo scheduling rappresenta una sorta di regola che deve essere rispettata dal driver del SO che gli consente di scegliere secondo un certo criterio quale processo deve essere servito per prima e quindi quale blocco mettere a disposizione per primo.

Si potrebbe pensare banalmente l'idea "servo il primo che richiede", ma questa non è detta che sia la politica migliore. Per capire qual è l'algoritmo di scheduling più giusto, si deve capire qual è atto pesa di più in termini di tempo per accedere ai vari blocchi.

Analizziamo dunque il **tempo di posizionamento** che ha due componenti principali:

- **Tempo di ricerca** (seek time): il tempo che impiega il braccio del disco a muovere le testine fino al cilindro contenente il settore desiderato. Questa componente pesa molto in termini di tempo.
- **Latenza di rotazione**: è il tempo aggiuntivo speso in attesa che il disco faccia ruotare il settore desiderato sotto la testina. Questa componente pesa poco in termini di tempo.

L'**ampiezza di banda** del disco è data dal numero totale di byte trasferiti diviso per il tempo totale che intercorre fra la richiesta di servizio e il completamento dell'ultimo trasferimento.

Quando si analizzano gli algoritmi di scheduling del disco si tiene conto quasi esclusivamente del seek time. L'obiettivo è minimizzarlo.

Supponiamo di avere un disco in cui le tracce sono numerate da 0 a 199 e supponiamo si presenti la seguente coda di richiesta sulle tracce: 98, 153, 37, 122, 14, 124, 65, 67.

Sappiamo inoltre che la testina è posizionata sul cilindro 53. Vediamo i vari algoritmi che ci permettono di gestire queste tracce:

4.1.0 FCFS (FIRST COME FIRST SERVED)

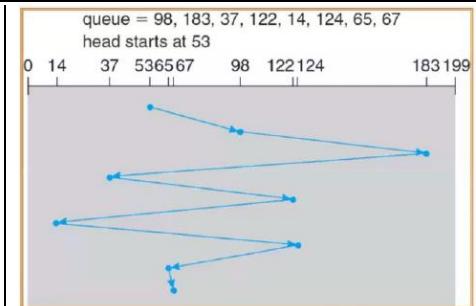
L'idea è quella di servirli nell'ordine in cui le richieste sono arrivate.

Di conseguenza viene servito prima il 98, poi il 183, poi il 37 e così via.

Dal punto di vista dell'immagine vediamo come inizialmente, da traccia, la testina sta sulla posizione 53, successivamente si sposta verso il 98. Da 98 a 183 e così via.

Dovendo valutare tutti questi movimenti della testina quello che si fa è vedere quanti spostamenti si sono accumulati. Per esempio, mi pongo la seguente domanda: quanto tempo ho perso per andare da 53 a 98? $98-53= 45$. Da 98 a 183, 85.

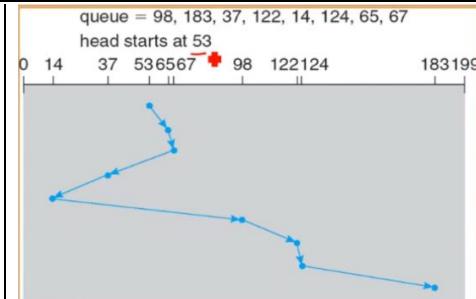
Continuando su questa idea alla fine scopriremo che il movimento totale della testina è stato di 640 cilindri. Si può fare di meglio?



4.1.1 SSTF (SHORTEST SEEK TIME FIRST)

A partire dalla posizione attuale del cilindro, ci si va a spostare nei posti più vicini alla posizione attuale del cilindro, tra tutte le richieste che devono essere soddisfatte. Tenendo conto che la posizione iniziale del cilindro è 53, guardo tutte le richieste che ho nella queue e mi rendo conto che la richiesta più vicina alla posizione 53 è la 65. Mi sposto quindi alla posizione 65, poi vedo la richiesta che è più vicina alla posizione attuale della testina ed è la 67, mi sposto lì e così via.

Se consideriamo il tempo, il movimento totale della testina è 263 che, confrontato con il 640 di prima, è molto meglio. Attenzione però al fatto che questo tipo di schedulazione può causare attesa indefinita di richieste (**starvation**). Supponiamo che arrivino una serie di richieste che oscillano tra 60 e 70. Se c'è una richiesta alla posizione 180, questa non verrà mai servita.

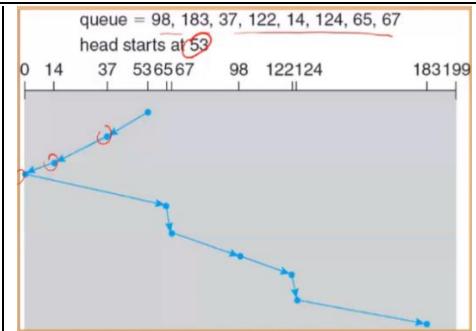


4.1.2 SCAN

Per garantire che tutte le richieste possano sempre essere servite, si muove il braccio del disco da un estremo all'altro. Per esempio, supponendo che la posizione iniziale della testina sia 53, e che il movimento della testina sia verso lo 0. Allora nel tragitto verso lo 0 vengono servite man mano tutte le varie richieste (37-14). Arrivato alla posizione 0 verranno servite in ordine crescente. In questo modo tutte le richieste sono potenzialmente raggiungibili, evitando chiaramente lo **starvation**.

In questo caso il movimento totale della testina è 208 cilindri.

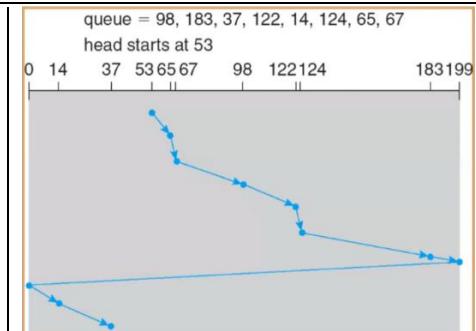
Ma una volta soddisfatta la richiesta alla posizione 14, che senso ha dover andare anche alla posizione 0? Da 14 si potrebbe pensare di andare a soddisfare direttamente la richiesta alla posizione 65. Un altro difetto è che dalla posizione 0 alla posizione 65 sicuramente non ci saranno altre richieste che verranno servite e dunque questo è tutto tempo che viene sprecato inutilmente.



4.1.3 C-SCAN

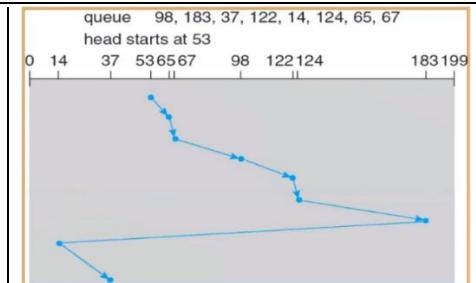
Il braccio si muove da un capo all'altro del disco, servendo le richieste lungo il percorso quando raggiunge l'altro capo, ritorna direttamente all'inizio del disco (gestione mediante lista circolare) senza servire alcuna richiesta durante il ritorno. Le tracce vengono sempre viste come all'interno di una lista circolare e vengono servite **sempre in una direzione**. Dunque, la posizione della testina parte da 53 e si muove verso destra. Vengono servite tutte le richieste fino a 183. Da 183 si sposta a 199, dopodiché avviene un passaggio immediato da 199 a 0 (proprio perché è visto come una lista circolare). Da 0 ricomincia sempre in senso crescente e serve le ultime richieste che aveva.

Anche in questo algoritmo il problema è il seguente: una volta servita la richiesta 183, che senso ha dovermi spostare fino a 199, e da 199 andare a 0? Non potrei pensare di spostarmi da 183 a 14 direttamente?



4.1.4 C-LOOK

C-LOOK risolve il problema presentato qui sopra: arrivato alla richiesta 183, successivamente si riparte dalla prima richiesta più piccola e cioè 14 e si va in senso crescente fino a terminare le richieste.



4.2 UNITA' A STATO SOLIDO

All'interno delle SSD (Solid State Drive) non sono presenti dei dischi, ma si basano su memoria flash. Non è richiesta alcuna componente meccanica o magnetica ed è un dispositivo di memoria secondaria che ha una struttura totalmente differente rispetto a quelle viste precedentemente. L'unico difetto delle SSD è il numero delle possibili scritture che possono essere fatte (difetto relativo in quanto sono nell'ordine dei milioni).

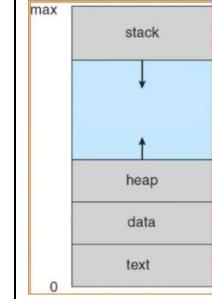
Caratteristiche delle SSD sono:

- Meno soggetti a danni.
- Più silenziosi (non c'è movimento meccanico).
- Più veloci (non c'è seek).
- Non necessitano di defrag (per ridurre il tempo di seek).

5.0 STATI DI UN PROCESSO

Supponiamo di avere un programma in C e di effettuarne una compilazione. Quando generiamo il nostro file a.out e lo mandiamo in esecuzione, stiamo creando un'**istanza del programma**, ossia un **processo** che deve eseguire delle istruzioni scritte all'interno di un programma. Un processo rappresenta un'esecuzione dell'eseguibile. Di un medesimo programma possono esserci più processi associati.

Dal punto di vista del SO, la prima cosa che succede, quando a.out viene mandato in esecuzione, è che deve essere trovato un posto all'interno della memoria principale. Una volta trovato questo pezzo in memoria principale all'interno di questo spazio sono presenti le seguenti informazioni: il codice eseguito, i vari dati del programma e poi viene lasciato uno spazio completamente libero, il quale verrà utilizzato come spazio di servizio, per esempio nel momento in cui avviene un'allocazione dinamica della memoria, questo spazio allocato dinamicamente è presente in questo spazio libero. Ci sarà anche uno stack all'interno dello spazio del processo.



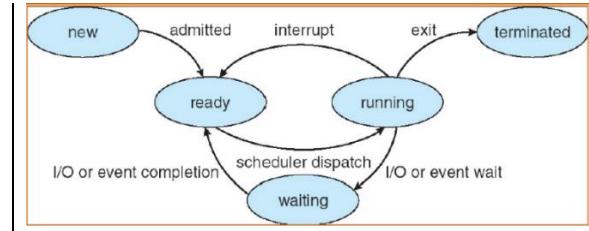
Supponiamo che nel nostro codice sono presenti delle chiamate a funzione, quando chiamiamo una funzione, tutte le informazioni contenute nel programma principale vengono conservative all'interno dello stack. Dopo che la funzione chiamata è stata eseguita, tutte le informazioni del programma principale vengono recuperate dallo stack e l'esecuzione può procedere normalmente.

Una volta eseguito il primo step (trovare spazio nella memoria principale), si deve caratterizzare il processo. Un processo è caratterizzato da: lo spazio allocato, il program counter (ci dice a che punto siamo dell'esecuzione del codice), contenuto registri CPU (in un certo istante, quali contenuti dei registri della CPU sono usati), sezione testo (codice), sezione dati, heap, stack.

Tutte queste informazioni ci servono per sapere ad un certo istante, un processo in che stato si trova.

Un processo durante la sua esecuzione può trovarsi in uno di 5 stati:

- **New**: il processo è stato creato (appena do il comando a.out);
- **Running**: le sue istruzioni vengono eseguite. Questo è il momento in cui gli viene assegnata la CPU;
- **Waiting**: il processo è in attesa di qualche evento (ad esempio fine di un'operazione di I/O);
- **Ready**: il processo è in attesa di essere assegnato ad un processore;
- **Terminated**: il processo ha terminato l'esecuzione.



Appena si crea un processo, siamo nello stato **new**, in questo stato si assegna lo spazio di memoria. Una volta "apparecchiato", il processo entra nello stato **ready**, in questo stato è pronto per l'esecuzione e si aspetta che venga assegnata la CPU. Dallo stato **ready** si esce e si va nello stato **running** in cui viene assegnata la CPU e si comincia con l'esecuzione del programma. Dallo stato **running** si può uscire per vari motivi: quando c'è un'operazione di I/O il processo viene fatto uscire dalla CPU, si passa allo stato **waiting** in cui si aspetta la computazione. Appena l'operazione finisce si torna allo stato **ready** in cui si indica che il processo è pronto e che rimane in attesa della CPU. Un altro modo per uscire dallo stato **running** è quando arriva un **interrupt**. Un esempio di interrupt si ha quando il tempo di un processo all'interno della CPU è limitato, superato un certo tempo scatta un interrupt che fa passare il processo allo stato **ready**. Mediante un interrupt, si torna allo stato **ready**, in attesa che venga riassegnata la CPU. Dallo stato **running** si passa allo stato **terminated** mediante una **exit** nel momento in cui ho terminato il codice da eseguire.

Quando un processo perde la CPU, in qualche modo si deve effettuare una fotografia per sapere a che punto si trova il processo, per poi poter ripartire dal punto in cui ci si è fermato in precedenza. Queste informazioni sono presenti all'interno del **Process Control Block**.

Il **Process Control Block** rappresenta una serie di informazioni sullo stato del processo, su quella che è la situazione attuale, e sono:

- **Stato del processo**: a quale degli stati si trova un processo;
- **Numero del processo**: ad ogni processo viene assegnato un identificatore numerico che lo identifica univocamente. Questo viene chiamato il **PID** di un processo;
- **Program counter**: a che punto dell'esecuzione del codice sono arrivato. Utile nel momento in cui si riprende la CPU si sa da dove riprendere l'esecuzione;
- **Registri**;
- **Memory limits**: da dove comincia e dove finisce il pezzo di memoria assegnato ad un processo;
- **Lista dei file aperti dal processo**;
- **Informazioni di contabilizzazione** (quanto tempo ho usato la CPU), **informazioni sullo stato dell'I/O**, etc...

process state
process number
program counter
registers
memory limits
list of open files
...

Supponiamo di avere un processo P_0 in esecuzione (stato **running**). Per un qualche motivo arriva un interrupt, dunque il processo perde la CPU a favore di un altro processo.

Quando il processo esce dalla CPU deve essere aggiornato il suo **Process Control Block**.

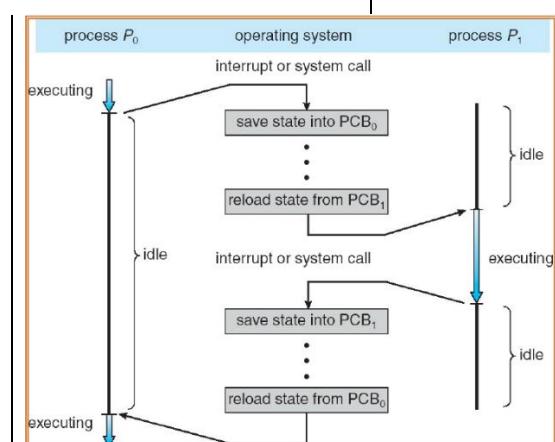
Esce dalla CPU, vengono salvate le sue informazioni nel PCB $_0$.

Il SO deve scegliere tra tutti i processi **ready**, quello che deve prendere la CPU. Supponiamo che tra tutti i processi **ready**, viene scelto P_1 .

Dal PCB di P_1 devono essere prese tutte le informazioni per poter ripartire da dove era stato fermato in precedenza.

Questo lasso di tempo viene chiamato **cambio di contesto** (rappresenta un **tempo perso**).

Una volta fatto ciò il processo P_1 viene eseguito fino a quando per un qualche motivo c'è un interrupt e gli viene tolta la CPU e si ripete quanto detto in precedenza.

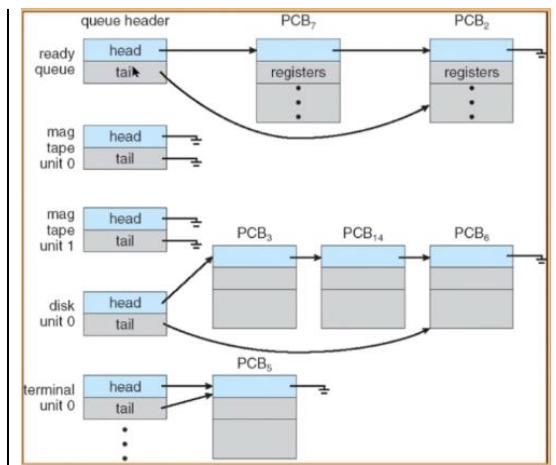


Un concetto importante sono le **Code di scheduling per i processi**, cioè, quando un processo esce dalla CPU, può capitare che non è l'unico processo che magari aspetterà nuovamente la CPU.

I processi che sono nello stato **ready** e sono in attesa vengono messi all'interno di una **Ready queue**. È presente anche una **coda dei dispositivi**, ossia l'insieme dei processi in attesa di qualche dispositivo I/O.

Più in generale abbiamo una **job queue**, ossia l'insieme di tutti i processi nel sistema. I processi, durante la loro vita, migrano tra varie code.

Questo è un esempio di ready queue in cui ogni processo è in attesa ed è conosciuto mediante il suo PCB →



5.1 CICLO DI VITA DI UN PROCESSO & SCHEDULATORI

Un processo, dopo lo stato **new**, arriva nella **ready queue** e sta lì fino a che non gli viene assegnata la CPU. Dalla CPU può uscire quando il codice del programma è terminato, oppure perché c'è una richiesta di I/O, e se c'è una richiesta di I/O allora si deve entrare all'interno della coda dei dispositivi e aspetta che possa essere eseguita una certa operazione.

Completata questa operazione si ritorna nella **ready queue**.

Dalla CPU si può anche uscire perché è finito il tempo assegnato a priori all'interno della CPU.

Non si passa per altre code ma si ritorna subito alla **ready queue**.

Si può uscire dalla CPU perché è stata fatta una **fork** per creare un figlio.

L'ultimo caso per cui si esce dalla CPU è quando c'è un **interrupt**.

Nella gestione dei processi sono importanti gli **schedulatori**, che sono degli strumenti che servono a selezionare i processi dalle varie code, quando si è per esempio nella **I/O queue**, ci sono vari criteri per stabilire quali delle richieste servire per prima. Lo scheduler sceglie tra i processi che devono entrare nell'I/O quale eseguire. La scelta avviene secondo vari algoritmi. Ci sono vari scheduler:

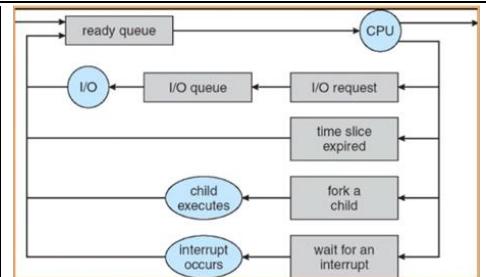
- **Schedulatore a lungo termine (job scheduler)**: seleziona i processi dal disco che devono essere caricati in memoria centrale (**ready queue**). Questo avviene perché non tutti i processi possono stare contemporaneamente in memoria principale altrimenti si consuma molto spazio.
- **Schedulatore a breve termine (CPU scheduler)**: seleziona il prossimo processo che la CPU dovrebbe eseguire.

Sistemi come Unix e Windows, sono privi dello schedulatore a lungo termine e si limitano a caricare in memoria tutti i nuovi processi ed a gestirli con lo scheduler a breve termine. È l'utente che, se vede che le prestazioni della macchina diminuiscono, decide di chiudere alcune applicazioni.

Altro tipo di schedulatore è quello a **medio termine** che viene usato quando non esiste quello a lungo termine.

Una volta che si rende conto che la situazione si sta intasando, prende il processo che ha la CPU e lo mette sul disco.

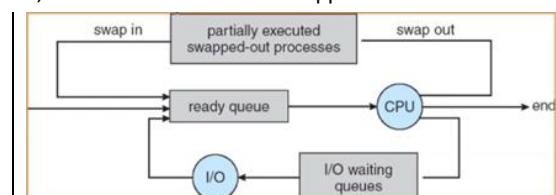
Una volta che si libera dello spazio in memoria principale, lo riporta sulla memoria.



I processi possono essere descritti come:

- **Processi I/O bound**: consumano più tempo facendo I/O che computazione, contengono molti e brevi CPU burst.
- **Processi CPU-bound**: consumano più tempo facendo computazione; contengono pochi e lunghi CPU burst.

È compito dello **schedulatore a lungo termine** quello di scegliere una giusta combinazione tra i processi di I/O bound e quelli CPU-bound che sono presenti in memoria centrale.



6. SCHEDULING DELLA CPU

Verranno illustrati quelli che sono gli algoritmi utilizzati dallo **schedulatore a breve termine**.

In realtà, durante l'esecuzione di un processo, ci sono una serie di cicli d'esecuzione della CPU (per un certo tempo uso la CPU) e attese di I/O.

Il ciclo di vita di un processo può essere visto come un'alternanza di CPU burst e I/O burst, come si vede nell'immagine al lato.

Se si considerano i CPU-burst di piccola entità sono molto più frequenti.

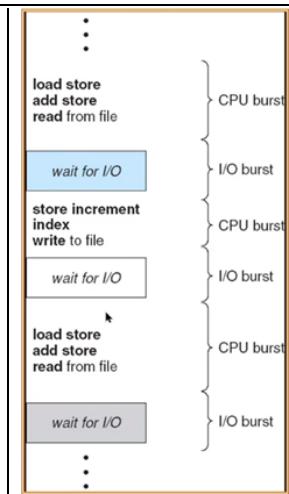
Lo scheduler della CPU può avvenire quando:

1. Si passa dallo stato di **esecuzione** allo stato di **attesa** per una operazione I/O o per una wait. In questo caso la schedulazione è **non preemptive** (senza diritto di prelazione).
2. Si passa dallo stato di **esecuzione** allo stato di **pronto** per l'arrivo di un segnale di interruzione. In questo caso la schedulazione è **preemptive** (con diritto di prelazione).
3. Si passa dallo stato di **attesa** allo stato di **pronto** perché è terminata un'operazione di I/O. In questo caso la schedulazione è **preemptive** (con diritto di prelazione).
4. **Termina**. In questo caso la schedulazione è **non preemptive** (senza diritto di prelazione).

Quando parliamo di schedulazione **non preemptive** intendiamo che non si ha il diritto di interrompere chi è in esecuzione. Al contrario si dice **preemptive**.

Per capire qual è l'algoritmo migliore che seleziona uno dei processi pronti ad entrare nella CPU, si adottano questi criteri:

- **Utilizzo della CPU**: mantenere la CPU il più possibile impegnata. Si vuole massimizzare tale criterio.
- **Frequenza di completamento (throughput)**: numero di processi completati per unità di tempo. Si vuole massimizzare tale criterio.
- **Tempo di completamento (turnaround time)**: intervallo che va dal momento dell'immissione del processo nel sistema al momento del completamento. Si vuole minimizzare tale criterio.
- **Tempo di attesa (waiting time)**: somma dei tempi spesi in attesa nella coda dei processi pronti. Si vuole minimizzare tale criterio.
- **Tempo di risposta**: tempo che intercorre dalla formulazione della prima richiesta fino alla produzione della prima risposta, non l'output (per gli ambienti time-sharing). Si vuole minimizzare tale criterio.



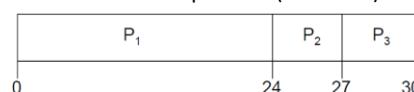
6.0.1 FIRST-COME, FIRST-SERVER (FCFS)

Il primo processo che arriva è il primo che viene servito. In generale si hanno vari processi in coda, per ogni processo c'è il proprio CPU Burst, ossia per quanto tempo richiede la CPU.

Il processo P_1 ha bisogno di 24ms nella CPU, P_2 e P_3 per 3ms.

Siccome l'algoritmo è di tipo **FCFS**, entrano i processi in base all'ordine di ingresso.

Si parte dall'istante 0 in cui si comincia. Naturalmente entra il processo P_1 e rimane nella CPU per 24ms, e di conseguenza esce a 24. Subito dopo entra P_2 per un tempo che è pari a 3 (esce a 27). Rimane P_3 che entra e rimane per 3ms (esce a 30). Tutto ciò si rappresenta nel seguente diagramma:



Se voglio valutare il tempo di attesa medio, effettuo i seguenti calcoli:

Chiamiamo **WT** (Waiting Time) il tempo di attesa di ogni Processo:

$$WT - P_1 = 0; \quad WT - P_2 = 24; \quad WT - P_3 = 27$$

Chiamiamo **Turnaround** il tempo totale in cui processo è stato all'interno del sistema:

$$T - P_1 = 24; \quad T - P_2 = 27; \quad T - P_3 = 30$$

Il **Tempo medio di attesa** è il seguente:

$$\text{Tempo medio } WT = (0+24+27)/3 = 17$$

$$\text{Tempo medio Turnaround} = (24+27+30)/3 = 27$$

Supponiamo di considerare l'ordine di arrivo in maniera differente rispetto a prima. In questo caso i processi arrivano nel seguente ordine: P_2 , P_3 , P_1

In questo caso il diagramma è il seguente:



Valutiamo il WT:

$$WT - P_1 = 6; \quad WT - P_2 = 0; \quad WT - P_3 = 3$$

$$\text{Tempo medio } WT = (6+0+3)/3 = 3 \rightarrow \text{questo valore è ben diverso da quello di prima.}$$

Questo ci dice che il tempo di attesa dipende dall'ordine in cui i processi vengono serviti. Il fatto di aver messo P_1 alla fine ci consente di accorciare i tempi di attesa di quelli che lo precedono, dato che P_1 ha un CPU-burst molto grande. FCFS è una scelta poco oculata.

6.0.2 SHORTEST-JOB-FIRST (SJF)

Viene schedulato il processo con il prossimo CPU burst più breve. Spostando un processo breve prima di un processo lungo, il tempo di attesa del processo breve diminuisce più di quanto aumenti il tempo d'attesa per il processo lungo. Di conseguenza il tempo d'attesa medio diminuisce.

Immaginiamo il seguente scenario, in cui oltre ad avere il classico Burst Time, abbiamo anche i tempi di arrivo dei vari processi →

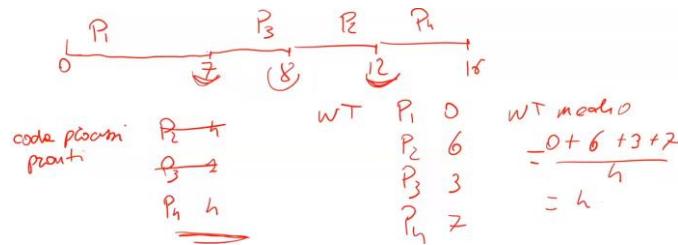
In questo caso il criterio è di scegliere, quando possibile, il processo con Burst Time più piccolo.

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Al tempo 0 naturalmente non c'è scelta, entra il processo P_1 in quanto è l'unico disponibile. Facciamo attenzione a questa cosa: stiamo considerando un caso **non-preemptive**, di conseguenza il processo P_1 ottiene la CPU e rimane per tutto il tempo che gli serve, dunque fino a 7.

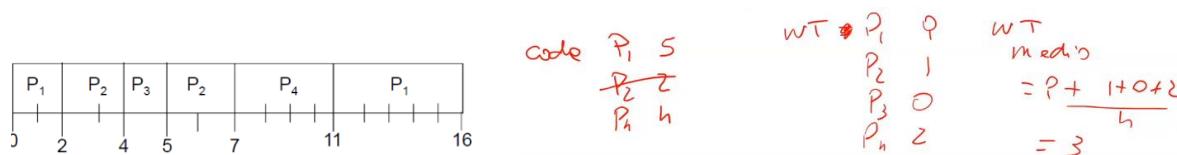
Siamo al tempo 7, di conseguenza tutti i processi sono arrivati (dunque in una coda ho i processi $P_2 - P_3 - P_4$), in questo caso, dato l'algoritmo SJF, scelgo quello che ha Burst Time minore, dunque faccio entrare il processo P_3 . Una volta processato P_3 , nella coda mi rimangono P_2 e P_4 , i quali hanno entrambi Burst Time pari a 4. In questo caso scelgo in maniera FIFO, il primo che è entrato in coda.

Di conseguenza servirà prima P_2 e poi P_4 . Il diagramma di Gantt è il seguente:



Quando invece esaminiamo il caso **preemptive**, un processo che entra in CPU, può essere interrotto. Di conseguenza, esaminando l'esempio precedente, il processo P_1 che vuole stare in CPU per 7 secondi, non posso fare questa cosa immediatamente in quanto mi devo sempre porre il problema di cosa succede quando un nuovo processo arriva.

In questo caso, il processo P_1 è il solo a tempo 0 e dunque parte. All'istante 2 è arrivato P_2 che momentaneamente si mette in coda. Devo confrontare il tempo residuo di P_1 e quello di P_2 e devo scegliere quello che ce l'ha minore: considerando che P_1 ha tempo residuo 5 e P_2 ha Burst Time 4, scelgo P_2 che dunque all'istante 2 entra nella CPU. P_1 si mette in coda con tempo rimanente 5. Adesso devo ripetere la stessa identica idea nel momento in cui all'istante 4 arriva il processo P_3 E arrivo alla conclusione che entra P_3 all'istante 4 e P_2 va in coda con tempo residuo 2.



Tale algoritmo può essere:

- **Nonpreemptive**: quando un processo ha ottenuto la CPU, non può essere prelazionato fino al completamento del suo cpu-burst.
- **Preemptive**: quando un nuovo processo è pronto, ed il suo CPU-burst è minore del tempo di cui necessita ancora il processo in esecuzione, c'è la prelazione. Questa schedulazione è detta *shortest-remaining-timefirst*.

STIMA DEL PROSSIMO CPU BURST

Quando abbiamo visto gli algoritmo di scheduling, abbiamo detto che un processo arriva con la sua richiesta di CPU burst. Consideriamo un processo che arriva e si mette in coda con il suo PCB che contiene tutte le sue informazioni. Nel PCB di un processo ci sta scritto a che punto sta il program counter, tabella dei file aperti, contenuto dei registri, etc.... Ma il processo non conosce il futuro, nel senso che dire "a me serve un CPU burst di 7", significa che sta prevedendo che quando entrerà in CPU gli serviranno 7ms.

Questo concetto di prevedere per quanti millisecondi verrà dedicata la CPU ad un processo, è un numero stimato. Per fare questa stima, ci si basa sulla seguente relazione di ricorrenza:

$$\tau_{n+1} = \alpha t_n + (1-\alpha) \tau_n \text{ dove:}$$

- τ_n = valore reale dell'ennesimo CPU burst.
- τ_{n+1} = valore previsto per il prossimo CPU burst.
- α = parametro con valore $0 \leq \alpha \leq 1$
- τ_1 = previsione primo CPU burst (valore di default)

Sviluppiamo la relazione di ricorrenza scritta in precedenza:

$$\begin{aligned} \tau_{n+1} &= \alpha t_n + (1-\alpha) \tau_n & | m \tau_n = \alpha t_{n-1} + (1-\alpha) \tau_{n-1} \\ &= \alpha t_n + (1-\alpha)(\alpha t_{n-1} + (1-\alpha) \tau_{n-1}) & = \\ &= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \tau_{n-1} & | m \tau_{n-1} = \\ &= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 (\alpha t_{n-2} + (1-\alpha) \tau_{n-2}) & \alpha t_{n-2} + (1-\alpha) \tau_{n-2} \\ &= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + (1-\alpha)^3 \tau_{n-2} & \\ &= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + (1-\alpha)^3 \tau_{n-2} & \end{aligned}$$

Notiamo che i coefficienti diventano sempre più piccoli, dunque io sommo tutta una serie di cose che vanno a decrescere. Nella stima che si fa per il prossimo CPU burst, si dà più peso ai CPU burst più recenti, rispetto a quelli più vecchi.

6.0.3 SCHEDULING A PRIORITÀ

In questo tipo di scheduling si assegna a ciascun processo, indipendentemente dal suo CPU burst, una priorità. Un processo all'interno del suo PCB mantiene il valore della sua priorità.

Ci può essere anche in quest'algoritmo il caso **preemptive** (durante l'esecuzione entra un processo con priorità maggiore) e **nonpreemptive**.

Anche lo **SJF** è un algoritmo a priorità, dove in quel caso la priorità è l'inverso della lunghezza del prossimo CPU burst (previsto).

Per questo tipo di algoritmo potrebbero esserci dei problemi: supponiamo di avere un processo in coda che ha una priorità bassa, se in un certo lasso di tempo continuano ad arrivare processi che hanno una priorità maggiore rispetto a quello con bassa priorità, quest'ultimo processo non entrerebbe

mai in CPU (questo fenomeno prende il nome di **starvation**). Una soluzione per un caso del genere è quello di effettuare un'operazione di **invecchiamento** (aging) secondo cui si accresce gradualmente la priorità dei processi nel sistema in modo che ad un certo punto avrà una priorità tale da entrare nella CPU. Naturalmente tutte queste informazioni sono contenute nel PCB di un processo.

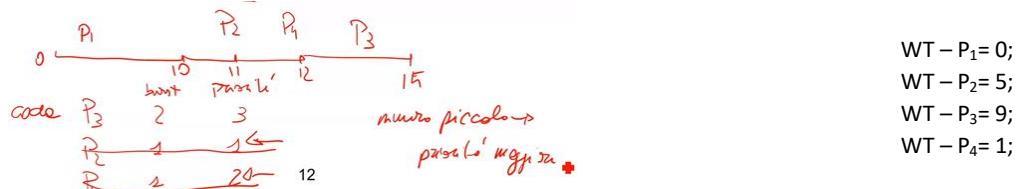
La priorità da assegnare può essere:

- **INTERNA** se dipende da fattori del processo governati dal SO (numero di file aperti, lunghezza media delle operazioni di I/O, etc.)
- **ESTERNA** (importanza del processo).

Vediamo come può funzionare un algoritmo di scheduling a priorità, caso **nonpreemptive**:

Teniamo a mente questa informazione: **più il numero è piccolo più la priorità è maggiore**.

In questo caso all'istante 0 abbiamo solo il processo P_1 che di conseguenza entra in CPU e rimane al suo interno per tutto il tempo richiesto dal suo Burst Time (in quanto stiamo analizzando il caso non preemptive). All'istante 10 nella coda dei processi in attesa sono entrati tutti i restanti processi ($P_2 - P_3 - P_4$). La scelta, tenendo conto che più il numero è piccolo più la priorità è maggiore, ricade nel processo P_2 . Una volta servito questo processo ci rimangono da scegliere P_3 e P_4 . Procediamo in tal senso con P_4 e successivamente con P_3 .

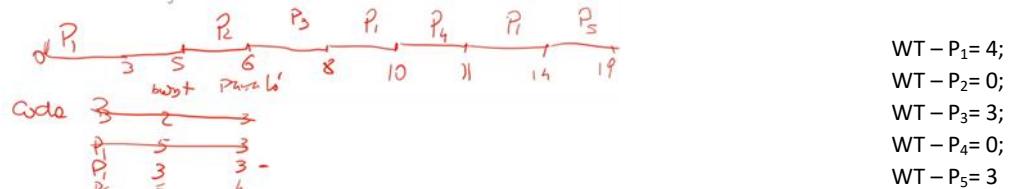


Il **Tempo medio di attesa** è il seguente: $WT = (0+5+9+1)/4 = 3,75$

Vediamo come può funzionare un algoritmo di scheduling a priorità, caso **preemptive**:

Process	Arrival Time	Burst Time	Priorità
P_1	0	10	3
P_2	5	1	1
P_3	3	2	3
P_4	10	1	2
P_5	11	5	4

In questo caso il processo P_1 entra nella CPU all'istante 0. Arrivato all'istante 3 devo fermarmi e considerare che è arrivato il processo P_3 con priorità 3. Siccome P_1 e P_3 hanno la stessa priorità, faccio continuare il processo P_1 , mentre P_3 va in coda. Arrivato all'istante 5 mi fermo e considero il processo P_2 . Poiché quest'ultimo ha priorità 1 allora il processo P_1 esce dalla CPU e va in coda con tempo rimanente pari a 5 ed entra nella CPU il processo P_2 . Ripeto ragionamenti analoghi fino alla fine... **Attenzione**: quando ho due processi in coda con la stessa priorità, faccio entrare nella CPU quello che è entrato per primo nella coda dei processi in attesa.



Il **Tempo medio di attesa** è il seguente: $WT = (4+0+3+0+3)/5 = 2$

6.0.4 ROUND ROBIN (RR)

Nel RR ad ogni processo viene assegnato un quanto di tempo. Cioè si stabilisce che per ogni processo che entra nella CPU viene assegnato un quanto di tempo q . Qualunque processo sia, qualunque priorità, rimarrà nella CPU per un tempo fissato q . Abbiamo detto che un processo rimane nella CPU per un certo tempo q . Finito questo tempo, se il CPU burst non è terminato, il processo va in coda ed entra il successivo, secondo un ordine FIFO.

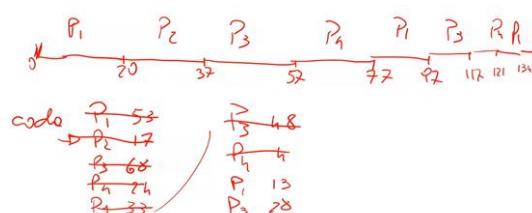
Adesso dobbiamo analizzare quanto deve essere grande questo valore q . Se q è molto grande ai fini pratici è come se stessimo analizzando un algoritmo FIFO. Se q è molto piccolo, si creano problemi in quanto ogni processo rimane nella CPU per un lasso di tempo molto piccolo. Per consumare tutto il suo CPU Burst, dovrebbe entrare e uscire un numero elevato di volte, generando troppi cambi di contesto (tempo perso inutilmente). Bisogna trovare dunque una buona misura di q .

Vediamo un esempio di applicazione di RR considerando $q=20$:

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

Al tempo 0 arriva il processo P_1 . Rimane in CPU per 20ms, dopodiché poiché avrà un tempo rimanente pari a 33, entra in coda e lascia spazio al processo P_2 . Poiché P_2 ha un Burst Time pari a 17 e $q=20$, riesce a consumare tutto il suo Burst Time e di conseguenza non verrà più considerato.

Ripeto lo stesso ragionamento per i successivi step...



7.0 MULTITHREADING

Il **Multithreading** è l'estensione del concetto di processo. Alla base del concetto di thread sta la constatazione che la definizione del processo è basata su due aspetti: possesso delle risorse ed esecuzione. Questi 2 aspetti sono indipendente e il SO può gestirli in maniera indipendente, dunque l'elemento che viene eseguito è detto **thread** mentre l'elemento che possiede le risorse è il **processo**. Il termine **multithreading** è utilizzato per descrivere la situazione in cui ad un processo sono associati più thread. Supponiamo di mandare in esecuzione un web browser potrebbe avere:

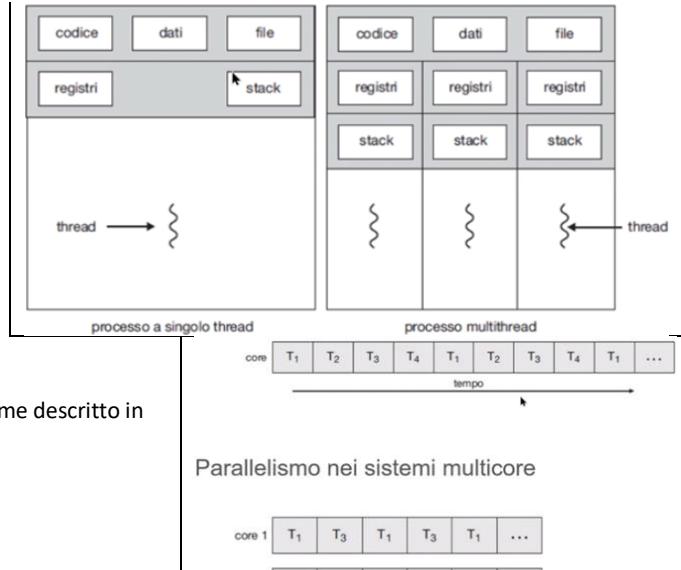
- Un thread per la rappresentazione sullo schermo di immagini e testo.
- Un thread per reperire i dati dalla rete.

Anche un Word processor potrebbe avere un thread per ciascun documento aperto (i thread sono tutti uguali ma lavorano su dati diversi).

Da quest'immagine si vede un processo a singolo thread (quello che abbiamo visto fino ad ora), in cui al processo viene assegnata una certa zona dati che conteneva codice, dati, etc., e vi era un unico thread di esecuzione.

Per quanto riguarda invece un processo multithread, l'esecuzione viene partizionata in tanti thread, ognuno dei quali si occupa di qualcosa di specifico. Localmente ogni thread ha a disposizione dei registri e uno stack (per favorire l'esecuzione).

Una caratteristica del Multithreading è che si adatta perfettamente alla programmazione nei sistemi multicore, in cui abbiamo 2 CPU posizionati in uno stesso chip. Si può partizionare la computazione sui due Core, in modo da migliorare i tempi di esecuzione.



Vediamo la differenza tra concorrenza e parallelismo.

Piuttosto che partizionare i processi nel tempo, decido di assegnarli ai 2 core, come descritto in precedenza.

Il supporto del SO ai thread avviene in 2 modi:

- Supporto **user-level**: librerie di funzioni (API) per gestire n thread in esecuzione.
- Supporto **kernel-level**: tabella dei thread del sistema.

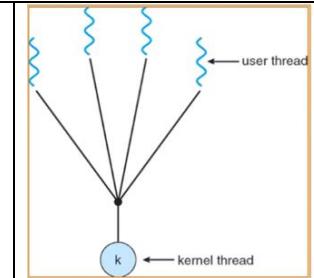
I Thread a livello utente sono gestiti come uno strato separato sopra il nucleo del sistema operativo, il kernel non ne è a conoscenza. Vengono realizzati tramite librerie di funzioni per la creazione, lo scheduling e la gestione dei thread. Il vantaggio è che può essere implementato un pacchetto di thread anche su SO che non supportano i thread. Uno svantaggio è che una chiamata di sistema bloccante da parte di un thread bloccherebbe tutti gli altri thread (il kernel blocca il processo). I Thread a livello kernel sono gestiti direttamente dal SO: il nucleo si occupa di creazione, scheduling, sincronizzazione e cancellazione dei thread nel suo spazio di indirizzi. Un vantaggio si ha perché quando un thread si blocca, il kernel può eseguire altri thread (anche dello stesso processo). Vediamo che tipo di relazione può esserci tra i thread a livello utente e i thread a livello kernel:

7.1 MODELLO MOLTI-A-UNO (M:1)

In questo tipo, ci sono molti thread a livello utente che però corrispondono ad un unico thread a livello kernel. I thread sono implementati a livello di applicazione, il loro scheduler non fa parte del SO che continua a vedere il tutto come un unico processo.

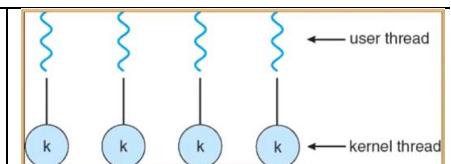
I vantaggi di questo modello è che la gestione dei thread è efficiente nello spazio utente (scheduling poco oneroso), non richiede un kernel multithread per poter essere implementato.

Gli svantaggi sono che l'intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante, inoltre, i thread sono legati allo stesso processo a livello kernel e non possono essere eseguiti su processori fisici distinti.



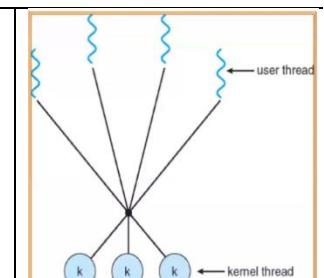
7.2 MODELLO UNO A UNO (1:1)

Ciascun thread a livello utente corrisponde ad un thread a livello kernel. Il kernel "vede" una traccia di esecuzione distinta per ogni thread. I thread vengono gestiti dallo scheduler del kernel (come se fossero processi). I vantaggi sono che lo scheduling è molto efficiente, se un thread effettua una chiamata bloccante, gli altri thread possono proseguire nella loro esecuzione. I thread possono essere eseguiti su processori fisici distinti. Gli svantaggi sono che può esserci inefficienza per il carico di lavoro dovuto alla creazione di molti thread a livello kernel, richiede un kernel multithread per poter essere implementato.



7.3 MODELLO MOLTI A MOLTI (M:M)

Si mettono in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel. In altre parole, il sistema dispone di un insieme ristretto di thread (detti anche *worker*), ognuno dei quali viene assegnato di volta in volta ad un thread utente. I vantaggi sono che c'è la possibilità di creare tanti thread a livello utente quanti sono necessari per la particolare applicazione (e sulla particolare architettura) ed i corrispondenti thread a livello kernel possono essere eseguiti in parallelo sulle architetture multiprocessore. Inoltre, se un thread invoca una chiamata di un sistema bloccante, il kernel può fare eseguire un altro thread. Uno svantaggio è che c'è difficoltà nel definire la dimensione del pool di worker e le modalità di cooperazione tra i due scheduler.



8. SINCRONIZZAZIONE TRA PROCESSI

Nei SO moderni, i processi vengono eseguiti in maniera concorrente, cioè si hanno più processi che vengono mandati in esecuzione e questi concorrono all'utilizzo della CPU e possono essere interrotti in qualunque momento (per esempio per priorità, etc.). I processi cooperanti possono condividere uno spazio logico di indirizzi, cioè codice e dati, oppure semplicemente dati, attraverso i file. Come si stabilisce chi deve manipolare per prima i dati? Ciò avviene mediante la sincronizzazione, che serve a mantenere la consistenza dei dati.

PRODUTTORE – CONSUMATORE:

Supponiamo di avere un processo produttore che produce informazioni che sono poi consumatore. I due processi hanno un buffer in comune in cui possono scrivere e dal quale possono leggere. Un esempio pratico di questa idea è il server-client: un server web fornisce (produce) pagine html e immagini, le quali vengono lette (consumate) dal browser web (client) che le richiede.

Definiamo il buffer nel seguente modo →

Supponiamo di avere a disposizione una certa struttura contenente al suo interno un certo numero di informazioni. Dichiariamo il *buffer* come un array di tanti dati di tipo elemento, dove il numero di elementi è 10.

Abbiamo altre variabili in comune tra il produttore e il consumatore che sono, oltre al buffer: *in*, *out*, *contatore* che vengono inizializzate a 0, capiremo a breve il loro funzionamento. Per adesso ci basta capire che questi sono i dati condivisi tra produttore e consumatore.

Vediamo ora come il processo **PRODUTTORE** è definito:

Supponiamo che il produttore abbia a disposizione un elemento appena prodotto che è di tipo elemento. Il produttore deve scrivere questo oggetto di tipo elemento all'interno del buffer. Per fare ciò, il produttore fa sempre un while(1) in quanto il suo compito è quello di inserire continuamente prodotti all'interno del buffer. All'interno di questo while notiamo che c'è subito l'uso della variabile *contatore* che abbiamo visto precedentemente come variabile comune che conta il numero di elementi che sono già presenti all'interno del buffer. Poi abbiamo una variabile *in* che rappresenta l'indice della successiva posizione libera nel buffer. Dunque, *contatore* ci dice quanti ce ne sono dentro, mentre *in* mi dice la prima posizione libera.

Nel codice del produttore c'è: `while(contatore == BUFFER_SIZE); /*do nothing*/`, significa che deve girare a vuoto nel momento in cui il buffer è pieno. Supponiamo che per un qualche motivo, il buffer comincia a svuotarsi, dunque il *contatore* è diverso da *BUFFER_SIZE*, di conseguenza il ciclo while fallisce e si procede nel codice. Quello che succede è che *buffer[in]* che ricordiamo indica la prima posizione libera all'interno di buffer, viene egualato ad *appena_prodotto*. Dopodiché si incrementa *in* in modulo *BUFFER_SIZE* (per poter ricominciare dall'inizio eventualmente scrivo nell'ultima posizione) *in* quanto la posizione *in* è stata appena occupata. Viene incrementato *contatore* in quanto ho incrementato anche il *buffer*.

Vediamo adesso come è definito il processo **CONSUMATORE**:

Naturalmente lo scopo del consumatore è quello di prelevare elementi dal buffer. Per fare ciò, fa sempre un while(1). Appena entra in questo while nel momento in cui il *contatore* vale 0, e dunque non c'è nessun elemento all'interno del buffer, si rimane bloccati all'interno perché non deve fare niente.

Appena *contatore* diventa diverso da 0, significa che è stato inserito un elemento dal buffer, dunque esce dal while e procede con il codice. Quello che fa è andare a mettere all'interno della variabile *da_consumare*, il valore *buffer[out]*, dove *out* è l'indice della prima posizione piena del buffer.

Naturalmente viene incrementato *out* per indicare che l'elemento in posizione *out* è stato prima preso correttamente. Viene decrementato *contatore* in quanto è stato eliminato un elemento da *buffer*.

Prese separatamente, le procedure del produttore e del consumatore sono corrette, ma possono "non funzionare" se eseguite in concorrenza, sebbene siano corrette. In particolare, le istruzioni: *contatore--*; e *contatore++*; possono causare problemi se non atomiche (o si fa tutto o niente).

RACE CONDITION:

Quando si effettua l'incremento del contatore, quello che succede realmente è che: si prende il valore del contatore e lo si mette in un registro, si incrementa il valore del registro e si mette il valore del registro incrementato all'interno del contatore. Stesso ragionamento per il decremento.

L'incremento e il decremento del contatore non sono operazioni atomiche, dunque se il processo produttore/consumatore perde la CPU in una posizione intermedia dell'operazione di incremento/decremento del contatore, questo può causare problemi, del tipo:

Supponiamo che contatore valga inizialmente 5. La CPU viene assegnata al produttore al tempo T_0 e mettiamo che è arrivata al punto in cui deve fare *contatore++*. Per fare questa operazione deve fare le 3 operazioni descritte precedentemente, supponiamo che riesce a fare solo le prime 2 e poi perde la CPU, il valore di *registro* è 6. Subito dopo, la CPU viene presa dal processo consumatore la quale deve effettuare *contatore--*. Anch'esso fa solo le prime due operazioni di un decremento, quindi *registro* varrà 4. Una volta persa la CPU, viene assegnata nuovamente a produttore che deve fare il terzo passo dell'operazione di incremento, secondo cui deve assegnare a contatore, il valore di *registro*, dunque contatore vale 6.

Il produttore riperde la CPU e viene assegnata al processo consumatore il quale deve completare l'operazione precedente e consumatore varrà 4.

Quello che è successo, e che, tenendo conto che inizialmente contatore valesse 5, quello che ho fatto è stato aggiungere e togliere un elemento, di conseguenza ci si aspetta che contatore continui a valere 5. In realtà contatore vale 4, ma è un valore errato chiaramente. Tutto dipende dallo scheduling della CPU.

SEZIONE CRITICA:

In casi di questo genere, cioè quando si hanno parti di codice in comune (come nel nostro caso sull'alterazione di contatore) vengono chiamate **sezioni critiche o corse critiche**. Per evitare le sezioni critiche occorre **sincronizzare** i processi. Questo è un fenomeno di **Race Condition** secondo cui più processi accedono in concorrenza e modificano dati condivisi, l'esito dell'esecuzione dipende dall'ordine nel quale sono avvenuti gli accessi.

```
#define BUFFER_SIZE 10
typedef struct {
    .
    .
} elemento;
elemento buffer[BUFFER_SIZE];

#define BUFFER_SIZE 10
typedef struct {
    .
    .
} elemento;
elemento buffer[BUFFER_SIZE];

int in = 0;
int out = 0;
int contatore = 0;

elemento appena_prodotto;
while (1) {
    while (contatore == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = appena_prodotto;
    in = (in + 1) % BUFFER_SIZE;
    contatore++;
}
```

```
elemento da_consumare;
while (1) {
    while (contatore == 0)
        ; /* do nothing */

    da_consumare = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    contatore--;
}
```

```
registro1 = contatore
registro1 = registro1 + 1
contatore = registro1
```

```
LOAD R1, CONT
ADD 1, R1
STORE R1, CONT
```

```
registro2 = contatore
registro2 = registro2 - 1
contatore = registro2
```

```
LOAD R2, CONT
SUB 1, R2
STORE R2, CONT
```

Esempio: inizialmente contatore=5
T₀ produttore: registro1=contatore (registro1=5)
T₁ produttore: registro1=registro1+1 (registro1=6)
T₂ consumatore: registro2=contatore (registro2=5)
T₃ consumatore: registro2=registro2-1 (registro2=4)
T₄ produttore: contatore=registro1 (contatore=6)
T₅ consumatore: contatore=registro2 (contatore=4)

Supponiamo di avere n processi che competono per utilizzare dati condivisi. Ciascun processo è costituito da un segmento di codice chiamato sezione critica in cui accede a dati condivisi. Il problema è assicurarsi che, quando un processo accede alla propria sezione critica, a nessun altro processo sia concessa l'esecuzione sulla propria sezione critica. L'esecuzione di sezioni critiche da parte di processi cooperanti è mutuamente esclusiva nel tempo.

La **soluzione** è quella di progettare un protocollo di cooperazione fra processi, a livello di codice:

- Ogni processo deve chiedere il permesso di accesso alla sezione critica, tramite una **entry section**, in modo che nessun altro processo può accedere alla propria sezione critica.
- La sezione critica è seguita da una **exit section**, in maniera tale da far capire agli altri processi che c'è la possibilità di accedere alla propria sezione critica.
- Il rimanente codice è non critico.

```
do {
    entry section
    sezione critica
    exit section
    sezione non critica
} while (1);
```

Le **entry section** ed **exit section** devono garantire:

- **Mutua esclusione**: se il processo P_i è in esecuzione nella sezione critica, nessun altro processo può eseguire la propria sezione critica.
- **Progresso**: se nessun processo è in esecuzione nella propria sezione critica ed esiste qualche processo che desidera accedervi, allora la sezione del processo che entrerà prossimamente nella propria sezione critica non può essere rimandata indefinitamente (evitare il deadlock).
- **Attesa limitata**: se un processo ha effettuato la richiesta di ingresso nella sezione critica, è necessario porre un limite al numero di volte che si consente ad altri processi di entrare nelle proprie sezioni critiche, prima che la richiesta del primo processo sia stata accordata (politica fai per evitare la starvation).

8.1 SOLUZIONE CON ALTERNANZA STRETTA

Questa è una soluzione semplice, in cui ci sono 2 processi P_0 e P_1 che utilizzano una variabile *turn* inizializzata a 0 inizialmente che serve a gestire la entry section e la exit section. Questi 2 codici garantiscono la stretta alternanza di ingresso alla sezione critica. Nel codice di P_0 inizialmente si va a verificare qual è il valore di *turn*, se è diverso da 0 allora non è il suo turno e cicla sul while. Appena *turn* vale 0 (viene settato dal processo P_1) allora è arrivato il suo turno, attende che la CPU gli venga data, e quando si entra nel primo while naturalmente fallirà e P_0 può entrare nella sua sezione critica.

```
do {
    while (turn != 0);
    sezione critica
    turn = 1;
    sezione non critica
} while (1);
```

algoritmo per il processo P_0

```
do {
    while (turn != 1);
    sezione critica
    turn = 0;
    sezione non critica
} while (1);
```

algoritmo per il processo P_1

Quando finisce, nella sua exit section, settnerà *turn* uguale a 1 in modo da permettere al processo P_1 di entrare nella sua sezione critica.

Molti degli aspetti precedenti vengono risolti.

Ma, se tocca a P_0 (cioè P_1 ha finito con la sua sezione critica ed ha messo *turn=0*) ed intanto P_0 si attarda ancora nella sua sezione non critica (per un qualsiasi motivo) → P_1 rimane bloccato → si può violare il progresso; quindi, possibile deadlock. Di conseguenza, è stata violata una condizione per la buona scrittura di un codice, dunque questa non è corretta.

8.2 SOLUZIONE DI PETERSEN (1981)

Oltre ad utilizzare la variabile *turn*, si utilizza un vettore contenente due variabili booleane (*boolean flag[2]*) dove:

l'array *flag* si usa per indicare se un processo è pronto ad entrare nella propria sezione critica, se *flag[i]=TRUE* questo implica che il processo P_i è pronto per accedere alla propria sezione critica.

Rispetto alla soluzione con alternanza stretta, è stata modificata la entry section e la exit section di entrambi i processi. Il flag di P_0 risulta essere a true quando P_0 è pronto ad entrare nella propria sezione critica, setta *turn=1*. Queste due istruzioni ci fanno capire che P_0 vorrebbe entrare nella sua sezione critica (*flag[0]=true*), in questo momento è il turno del processo P_1 (*turn=1*), posso entrare nella sezione critica?

Questa domanda si traduce nel seguente modo: si va a controllare se *flag[1]* è uguale a true e *turn==1*. Se queste due condizioni sono vere allora P_1 si trova nella sua sezione critica e dunque cicla questo while (rimane bloccato a questa condizione).

Poiché P_1 si trova nella sua sezione critica, quando la finisce setta *flag[1]=false* e procede normalmente con la sua sezione non critica, successivamente perde la CPU che viene nuovamente data a P_0 .

P_0 , che ricordiamo era rimasto bloccato in *while(flag[1] && turn == 1)*. Siccome ora *flag[1]=false* il while fallisce e il processo P_0 può entrare nella sua sezione critica.

Questa modalità risolve il problema della soluzione con alternanza stretta perché il processo P_1 termina la propria sezione non critica e ritorna alla prima istruzione della sua porzione di codice, dunque setta *flag[1]=true* e *turn=0*. Nello stesso tempo il processo P_0 è rimasto nella sua sezione non critica. Mentre nella soluzione precedente P_1 rimaneva bloccato, ora invece lui ha messo *flag[1]=true* per dire che è interessato all'utilizzo della sezione critica, ha messo *turn=0* per indicare che attualmente dovrebbe essere il turno di P_0 , però quando fa a fare il test del while è ovviamente duplice e, *turn* vale ovviamente 0, invece, avendo supposto precedentemente che il processo P_0 si trovasse nella propria sezione non critica (quindi non ha ricominciato la propria porzione di codice), *flag[0]=false*, questo comporta che il while fallisce e il processo P_1 può entrare nella propria sezione critica. Questa soluzione è perfettamente valida per 2 processi che tentano di utilizzare variabili comuni, ma se invece di essere 2 i processi fossero di più? In questo caso la soluzione di Petersen non sarebbe più corretta.

In generale, qualsiasi soluzione al problema della sezione critica richiede l'uso di un semplice strumento, detto **lock** (lucchetto). Per accedere alla propria sezione critica, il processo deve acquisire il processo di un lock, che restituirà al momento della sua uscita e uno dei tanti processi in attesa lo prenderà.

```
P1
do {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
    sezione critica
    flag[1] = false;
    sezione non critica
} while (1);
```

```
P0
do {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
    sezione critica
    flag[0] = false;
    sezione non critica
} while (1);
```

Molti sistemi hanno dei sistemi hardware che mettono a disposizione per evitare i problemi descritti precedentemente, per esempio interdire le interruzioni mentre si modificano le variabili condivise. Inoltre, molte architetture forniscono speciali istruzioni atomiche implementate in hardware: queste permettono di controllare e modificare il contenuto di una parola di memoria (**TestAndSet**), o di scambiare il contenuto di due parole di memoria (**Swap**).

```
do {
    acquisisce il lock
    sezione critica
    restituisce il lock
    sezione non critica
} while (1);
```

8.3 HARDWARE DI SINCRONIZZAZIONE (TestAndSet)

Questo è il codice della funzione *TestAndSet*. Viene eseguita atomicamente, ossia viene tutto eseguito insieme. La funzione prende in input un puntatore ad un oggetto booleano e restituisce un valore booleano. La funzione salva in una variabile booleana *value* il valore della variabile booleana *object* passata in input. Successivamente setta a true il contenuto della variabile booleana *object* passata in input e come ultima istruzione ritorna la variabile *value* ossia il valore booleano passato in input alla funzione.

Vediamo questa porzione di codice →

Inizialmente abbiamo una variabile booleana *lock* (rappresenta il **token**) inizializzata a **FALSE**.

Successivamente c'è do-while: nell'entry section c'è un while il cui contenuto è la chiamata a funzione definita precedentemente a cui gli passiamo in input il puntatore della variabile *lock*. Per quello che abbiamo detto prima sappiamo che la funzione restituirà **FALSE** e assegnerà a *lock* il valore **TRUE**. Poiché *TestAndSet* restituisce **FALSE** si esce dal while e si entra nella sezione critica.

lock=TRUE quando un processo è nella sua sezione critica.

Nella exit section *lock* viene settato a **FALSE** in modo tale che un altro processo può entrare nella propria sezione critica (viene restituito il **token**).

Tutto funziona però bisogna considerare che il sistema operativo deve supportare tale operazione atomica (*TestAndSet*).

```
boolean TestAndSet (boolean *object)
{
    boolean value = *object;
    *object = TRUE;
    return value;
}
```

```
boolean lock = FALSE;

do {
    while TestAndSet(&lock);
    sezione critica
    lock = FALSE;
    sezione non critica
} while (1);
```

8.4 SEMAFORI

Un semaforo non è nient'altro che una variabile intera, chiamata *S*. Dopo l'inizializzazione si può accedere al semaforo solo attraverso due **operazioni atomiche predefinite: wait() – signal()** (da non confondere con le omonime funzioni Unix).

Il codice della **wait** è il seguente: prende come argomento il semaforo e quindi la variabile *S*. Non fa altro che andare a verificare il contenuto di *S*. Se è minore o uguale a 0 allora non fa niente. Se *S* è maggiore di 0, viene decrementata di uno e se ne esce. La **signal**, che prende come argomento il semaforo e quindi la variabile *S* non fa altro che incrementare *S* di uno.

L'obiettivo della variabile semaforo è quello di contare il numero di risorse che si hanno a disposizione.

Immaginiamo il caso in cui abbiamo 3 stampanti a disposizione e condivise.

Poiché le stampanti sono condivise, decido di mettere un semaforo a guardia di queste 3 stampanti. Di conseguenza la variabile *S* varrà 3. Se arriva un processo che decide di effettuare una stampa e dunque esegue la **wait** per verificare se c'è una risorsa (stampante) disponibile. Siccome *S* vale 3, il while fallisce e decremento *S* di uno, quindi le stampanti a disposizione sono diventate 2 in quanto una è stata occupata.

Supponiamo che altri 2 processi richiedono una stampa: di conseguenza viene chiamata due volte la **wait**, per due volte il while fallisce e alla fine ci ritroveremo che la variabile *S* vale 0. Quando *S* vale 0 significa che le 3 stampanti sono occupate. Se arriva un quarto processo che richiede una stampante, naturalmente invoca la **wait** però in questo caso rimarrà bloccato nel while. Questo processo quarto processo si sblocca quando uno dei 3 processi che stava utilizzando la stampante, finisce di utilizzarla. Appena finisce di utilizzarla viene invocata la **signal** che incrementa *S* di uno. In questo modo una stampante diventa nuovamente disponibile. Tutte le modifiche al valore del semaforo contenute nelle operazioni **wait()** e **signal()** devono essere eseguite in modo atomico. Mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore. Nel caso di **wait()** devono essere effettuate atomicamente sia la verifica del valore del semaforo che il suo decremento.

```
wait(S) {
    while S<=0 ; // do notop
    S--;
}
```

```
signal(S) {
    S++;
}
```

8.4.1 SEMAFORI BINARI

Un semaforo binario assume soltanto i valori 0 e 1.

Inizialmente la variabile viene inizializzata a 1 in quanto è in guardia di un'unica risorsa. Quando diventa uguale 0 allora l'unica risorsa è in mano ad un processo e gli altri processi che la richiedono devono aspettare. Generalmente i semafori binari vengono chiamati *mutex*.

Quando un processo vuole utilizzare questo semaforo, nell'entry section chiama la **wait** passando come parametro *mutex*, mentre nella exit section chiama la **signal** passandogli *mutex*.

Il semaforo binario, come lo abbiamo appena visto, risulta essere utile nel momento in cui si vuole andare a proteggere una sezione critica. Può essere utilizzato anche per la **sincronizzazione**: stabilire un ordine di esecuzione tra i processi.

Vogliamo in qualche modo condizionare l'esecuzione dei processi, utilizzando i semafori. Supponiamo che si voglia prima far eseguire il processo P1 e poi quello P2, indipendentemente da come lo scheduling della CPU selezioni i processi. Allora quello che si fa in questo caso è di inizializzare un semaforo *synch* a 0. Il processo P1 è composto da tutte le istruzioni di cui è composto e solo alla fine ci sta una signal su *synch* che non fa nient'altro che settare *synch* a 1.

Il processo P2 è fatto in modo che la sua prima istruzione sia la **wait** su *synch* in modo tale che se venisse schedulato prima del processo P1, si bloccherebbe al while della **wait** (perché *synch* vale 0).

Se il while fallisce allora significa che P1 è stato eseguito per prima, e dunque abbiamo applicato la **sincronizzazione**.

BUSY WAITING:

In generale, quando il valore di un semaforo è maggiore di 1, allora parliamo di **semaforo contatore** (esempio precedente riguardo le stampanti). I processi che desiderano utilizzare un'istanza della risorsa, invocano una **wait()** sul semaforo, decrementandone così il valore. I processi che ne rilasciano un'istanza, invocano **signal()**, incrementando il valore del semaforo. Quando il semaforo vale 0, tutte le istanze della risorsa sono allocate e i processi che le richiedono devono sospendersi sul semaforo fino a che esso ridiventa positivo.

Questo fenomeno viene chiamato **busy waiting**. Come fare per risolvere questo problema? L'idea potrebbe essere la seguente: quando un processo chiama la **wait** passando come parametro un semaforo che vale 0, anziché essere bloccato in quel while, si potrebbe pensare di far perdere la CPU e di farlo accodare (tramite il Process Control Block) ad una lista linkata di tutti i processi che stanno aspettando per una certa risorsa.

```
Semaphore mutex=1; // inizializzazione del semaforo mutex = 1
```

```
do {
    wait(mutex);
    sezione critica
    signal(mutex);
    sezione non critica
} while (1);
```

Processo *P_i*

```
Semaphore synch=0;
```

```
P1:      S1;
         signal(synch);
```

```
P2:      wait(synch);
         S2;
```

Ricorda:

```
wait(S) {
    while S<=0;
    S--;
}
```

```
signal(S) {
    S++;
}
```

Ricorda:

```
wait(S) {
    while S<=0;
    S--;
}
```

```
signal(S) {
    S++;
}
```

Tutto ciò si traduce nella seguente struttura: in questo caso il semaforo non è più costituito da un singolo valore numerico, ma piuttosto da un record costituito dal valore numerico e da una lista di record, dove ogni elemento della lista è un processo (PCB del processo stesso) che vuole utilizzare le risorse messe a disposizione dal semaforo, ma che in un certo momento non sono disponibili.

```
typedef struct {
    int valore;
    struct processo *lista;
} semaphore;
```

Un semaforo del genere appena descritto utilizza due operazioni:

- **block**: posiziona il processo che richiede di essere bloccato nell'opportuna coda di attesa, ovvero sospende il processo che la invoca.
- **wakeup**: rimuove un processo dalla coda di attesa e lo sposta nella ready queue.

Il codice della **wait**, come conseguenza di quanto abbiamo appena detto, subisce delle modifiche: innanzitutto in questo caso si prende in input alla funzione un puntatore al semaforo che ricordiamo essere un record. Come prima cosa che si fa è decrementare $S->value$, ossia il numero di risorse a disposizione. Se $S->value$ è diventato un valore minore di 0 allora significa che non ci sono più risorse a disposizione e allora viene aggiunto il processo alla lista e invoca la **block** in modo da fargli perdere la CPU.

Per quanto riguarda la **signal**, la prima cosa che si fa è incrementare $S->value$. Se adesso $S->value$ è un valore minore o uguale a 0 (perché significa che c'è ancora qualche processo nella coda dei processi in attesa della risorsa) allora si rimuove il processo P da $S->list$ e si invoca la **wakeup**.

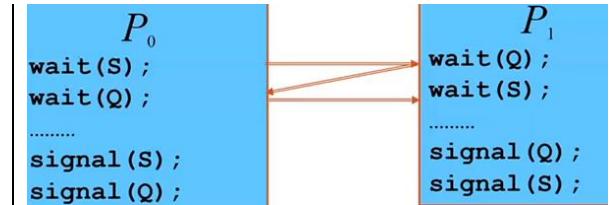
```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        aggiungi il processo P a S->list;
        block(P);
    }
}
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        rimuovi il processo P da S->list;
        wakeup(P);
    }
}
```

Un'osservazione: se $S->value$ è negativo, allora il valore assoluto di $S->value$: $|S->value|$ rappresenta il numero dei processi che attendono il semaforo. Ciò avviene a causa dell'inversione dell'ordine delle operazioni di decremento e verifica del valore nella **wait()**.

La realizzazione di un semaforo con coda di attesa può condurre a situazioni in cui ciascun processo attende l'esecuzione di un'operazione **signal()**, che solo uno degli altri processi in coda può causare. Più in generale, si verifica una situazione di **deadlock**, quando due o più processi attendono indefinitamente un evento che può essere causato soltanto da uno dei due processi in attesa.

Supponiamo ci siano due semafori **S** e **Q** inizializzati entrambi ad 1 e che ci siano due processi **P0** e **P1** che hanno le seguenti porzioni di codice:

Supponiamo venga eseguito prima il processo **P0**, che fa la **wait** su **S**: tenendo conto che **S** vale 1, riesce a prendere la risorsa. Immaginiamo che dopo aver fatto la **wait** su **S**, perde la CPU in favore di **P1**: **P1** fa la **wait** su **Q** che inizialmente vale 1 dunque continua. Successivamente perde la CPU che ritorna a **P0**. **P0** fa la **wait** su **Q** che in questo momento vale 0, dunque decrementa **Q** e viene piazzato in coda.



Supponiamo che ora perde la CPU che ripassa a **P1** che farà la **wait** su **S** che valeva 0, verrà decrementato il valore e andrà nella coda dei processi **S**. Per come abbiamo costruito i codici, nessuno li sbloccherà (**P0** può essere sbloccato da **P1** e viceversa) e dunque ci siamo messi in una situazione di deadlock. È importante stare attenti quando si scrive il codice.

Bisogna stare anche attenti al fenomeno della **Starvation**, questa si può verificare qualora i processi vengano rimossi dalla lista di attesa associata al semaforo in modalità LIFO. In teoria si potrebbe gestire mediante FIFO.

Facciamo ora un passo indietro e vediamo come risolvere il problema **produttore-consumatore** utilizzando i semafori:

```
semaphore full, empty, mutex;
// inizialmente full = 0, empty = n, mutex = 1
```

Abbiamo a disposizione 3 semafori **full**, **empty** e **mutex**. Il buffer ha **n** posizioni, ciascuna in grado di contenere un elemento. Il semaforo **mutex** garantisce la mutua esclusione sugli accessi al buffer. I semafori **empty** e **full** contano, rispettivamente, il numero di posizioni vuote ed il numero di posizioni piene nel buffer.

Questo è il codice di produttore, che continuamente produce un elemento. Dopo aver prodotto l'elemento fa una **wait** su **empty**. Successivamente esegue una **wait** su **mutex**. Si può osservare come **mutex** sia a guardia del buffer. Successivamente c'è una **signal** su **buffer** con cui si indica che le operazioni sul buffer sono terminate e poi una **signal** su **full**, in maniera tale da incrementare il numero delle posizioni piene.

```
do {
    ... /* produce un elemento in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ... /* inserisce next_produced nel buffer */
    ...
    signal(mutex);
    signal(full);
} while (true);

do {
    ...
    wait(full);
    wait(mutex);
    ... /* sposta un elemento dal buffer in next_consumed */
    ...
    signal(mutex);
    signal(empty);
    ... /* consuma un elemento in next_consumed */
} while (true);
```

Consumatore esegue le operazioni simmetriche secondo cui: innanzitutto le operazioni sul buffer sono controllate dal semaforo **mutex**. Inizialmente il consumatore può andare ad eseguire le operazioni quando all'interno del buffer c'è qualcosa al suo interno, dunque si effettua una **wait** su **full**. Una volta effettuate le operazioni sul buffer c'è una **signal** su **empty** in modo da far capire che è stato preso un elemento, c'è una posizione libera in più nel buffer.

In questo modo sono risolti tutti i problemi e abbiamo di conseguenza risolto il problema **produttore-consumatore**.

9. GESTIONE DELLA MEMORIA CENTRALE

Un programma in genere risiede su disco in forma di un file binario eseguibile e deve essere portato (dal disco) in memoria e “inserito” all’interno di un processo per essere eseguito e quindi il sistema operativo assegna al processo uno spazio di memoria.

La parte alta della memoria principale viene assegnata al kernel del sistema operativo e in quel momento il kernel viene caricato e poi c’è il resto della memoria principale che libera.

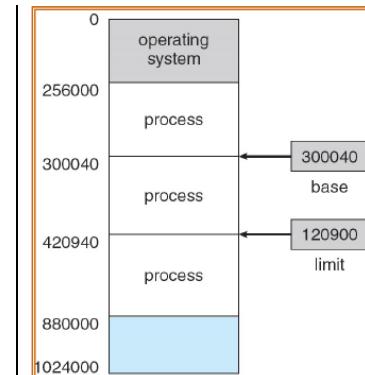
Man mano che si creano i processi a questi viene assegnato un pezzo di memoria principale e all’interno di essi viene immagazzinato tutto il necessario di tale processo.

Al processo in questione gli viene assegnato uno spazio ben definito.

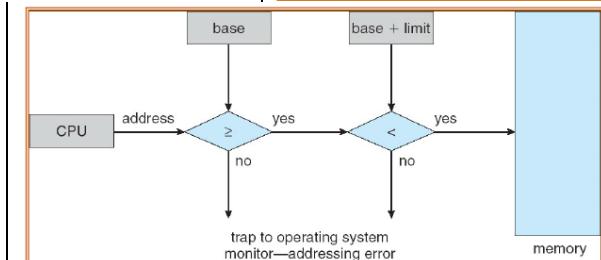
Il sistema operativo deve però mettere un sistema di protezione a questo spazio dedicato al processo, in modo che nessuno invada il suo spazio o che egli invada altri spazi.

Ogni qualvolta che la CPU sta eseguendo un certo processo emette un indirizzo, il SO deve garantire che quell’indirizzo che è stato richiesto è parte dello spazio che è stato assegnato a quel processo specifico.

Per garantire ciò il SO assegna ad ogni processo una coppia di registri (registro **base** e **limit**) che definiscono lo spazio di indirizzamento fisico del processo.



Questi registri **base** e **limit**, vengono utilizzati con questi controlli →



Il collegamento delle istruzioni e dei dati agli indirizzi di memoria può essere effettuato in:

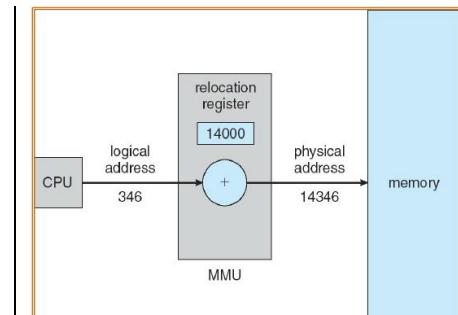
- **Fase (o tempo) di compilazione:** se in fase di compilazione si conosce dove il processo risiederà in memoria, allora si può generare un codice assoluto. Se tale indirizzo cambia per una qualche ragione bisogna ricompilare;
- **Fase di caricamento:** il compilatore genera un codice rilocabile. Il caricatore associa indirizzi rilocabili ad indirizzi di memoria. Se l’indirizzo iniziale cambia è sufficiente ricaricare in memoria il codice;
- **Fase (o tempo) di esecuzione:** se il processo può venire spostato, durante l’esecuzione, da un segmento di memoria a un altro, allora il collegamento deve essere ritardato fino al momento dell’esecuzione. Necessita di supporto hardware.

Indirizzo logico – indirizzo generato dalla CPU, anche definito come indirizzo virtuale.

Indirizzo fisico – indirizzo visto dalla memoria.

Fase di compilazione e di caricamento: indirizzi logici e fisici identici.

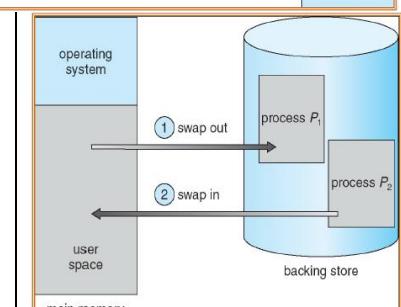
Fase di esecuzione: indirizzi logici e fisici diversi.



CARICAMENTO DINAMICO:

Una procedura non è caricata finché non è chiamata, vengono mantenute in memoria secondaria in un formato rilocabile.

Un problema che si risolve in questo modo è lo **swapping**, ovvero un processo può essere temporaneamente spostato e poi riportato in memoria centrale (swap out, swap in). La maggior parte del tempo di swap è tempo di trasferimento, il tempo totale di trasferimento è direttamente proporzionale alla quantità di memoria interessata, ma anche problemi di I/O.



ALLOCAZIONE CONTIGUA DI MEMORIA:

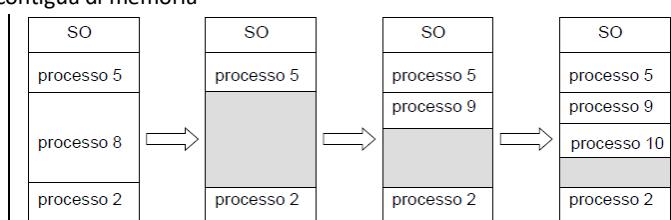
La memoria centrale è normalmente divisa in due parti:

- Sistema operativo residente, collocato nella memoria bassa con il vettore degli interrupt.
- Processi dell’utente collocati nella memoria alta.

Con l’allocazione contigua, ciascun processo è contenuto in una singola sezione contigua di memoria

Quando un processo arriva, il sistema cerca nell’insieme un blocco libero abbastanza grande per questo processo.

Supponiamo che il processo 8 termini e lascia un buco (**hole**) nella memoria, arrivano altri processi e vengono allocati in quello spazio, arriva il processo 11 che vorrebbe entrare anche lui ma non può perché non c’è abbastanza spazio, in realtà non c’è spazio abbastanza spazio contiguo (**frammentazione esterna**).



Si prende la memoria principale piuttosto che vederla come una sequenza di Byte la si suddivide in blocchi di dati di potenza di 2, così quando un processo arriva gli si dà un certo numero di blocchi contigui. Esistono varie politiche per assegnare blocchi liberi:

- **First-fit** (più veloce perché non deve esaminare l’intera lista): assegna il primo blocco libero abbastanza grande per contenere lo spazio richiesto.
- **Best-fit**: assegna il più piccolo blocco libero abbastanza grande.
- **Worst-fit** (meno efficiente sull’utilizzo della memoria): assegna il più grande blocco libero.

9.1 PAGINAZIONE

Dobbiamo capire come il sistema operativo assegna spazio ai vari processi quando vengono mandati in esecuzione. I principali approcci sono i First-fit, Best-fit e Worst-fit, ma questi possono provocare frammentazione esterna.

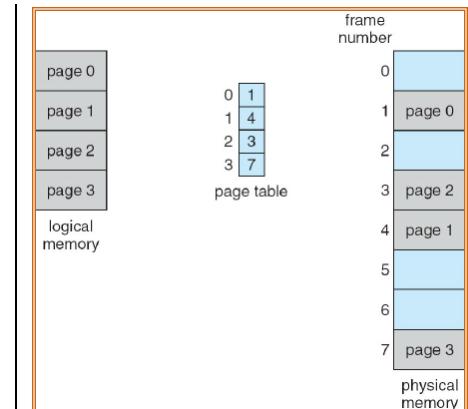
Per ovviare a tali problemi si usa la **paginazione**:

Si suddivide la memoria principale in blocchi chiamati **frame** che sono potenze di 2, si prende il pezzo di memoria che il processo necessita e si suddivide anche questo in blocchi chiamate **pagina**.

La size della **pagina** è uguale alla size di un **frame**, così che ciascuna pagina può calzare all'interno di un frame della memoria principale.

Quindi l'idea è di assegnare le varie pagine del processo a frame liberi in memoria principale.

Per non perdere i vari pezzi del processo per la memoria principale, si imposta una **tavella delle pagine**, si va a scrivere dove sono state collocate tutte le pagine del processo.



TRADUZIONE INDIRIZZO LOGICO IN FISICO:

Ogni indirizzo generato dalla CPU è diviso in due parti:

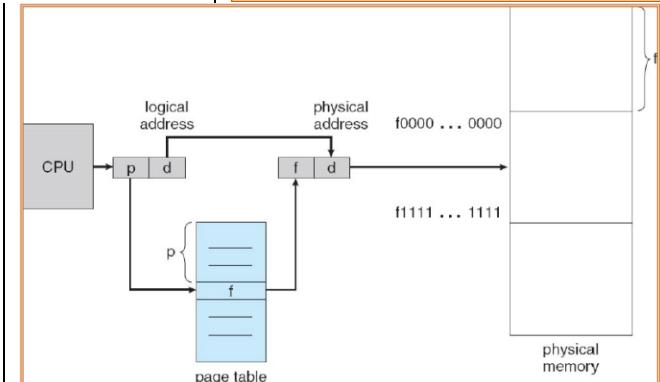
- Numero di pagina (p)**, usato come indice nella tabella delle pagine che contiene l'indirizzo di base di ogni frame nella memoria fisica.
- Spiazzamento nella pagina (d)**, combinato con l'indirizzo di base per calcolare l'indirizzo di memoria fisica che viene mandato all'unità di memoria centrale.

numero di pagina	offset di pagina
<i>p</i>	<i>d</i>

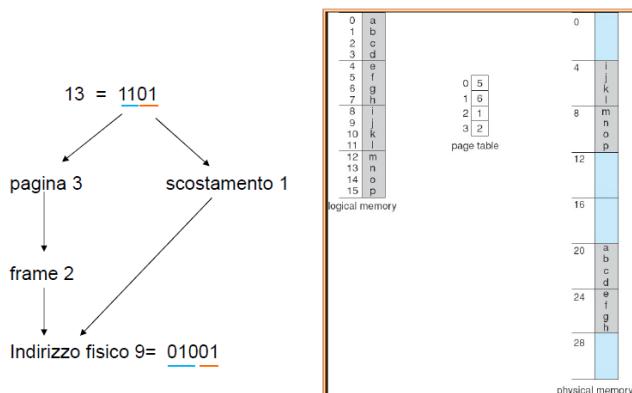
($m - n$) bit n bit

$$2^m = \text{spazio degli indirizzi logici}$$

$$2^n = \text{size di una pagina}$$



Esempio di paginazione per una memoria centrale di 32 byte con pagine di 4 byte



size di un frame= size di una pagina = 2048 = 2^{11}

size del processo = $10469 = (2^{11} \times 5) + 229 \Rightarrow \# \text{ pagine} = 6$ (cioè 12288 = $2^{11} \times 6$ byte)

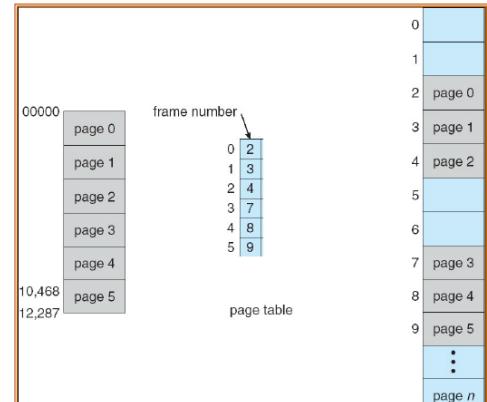
11 bit = bit necessari per rappresentare una posizione in una pagina (o frame), cioè per *d*

3 bit = bit necessari per rappresentare il numero di una pagina, cioè per *p*

numero di pagina	offset di pagina
<i>p</i>	<i>d</i>

3 bit 11 bit

Esempio di paginazione



2^{14} = spazio degli indirizzi logici

2^{11} = size di una pagina

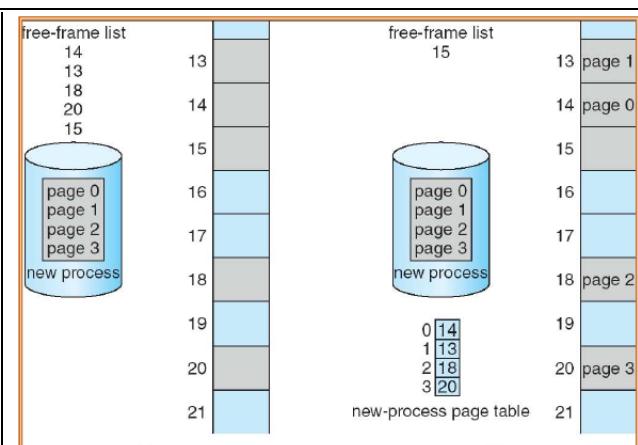
9.2 IMPLEMENTAZIONE DI UNA TABELLA DELLE PAGINE

Il SO conserva una **lista dei frame liberi**, quando un processo parte e richiede un numero di pagine allora il sistema va nella lista e prende tutti i frame necessari.

La tabella delle pagine è mantenuta nella memoria centrale assieme al processo. Il registro base della tabella delle pagine (**PTBR**) punta alla tabella, quando vi è un cambio di contesto basta cambiare il contenuto del PTBR.

Ogni accesso a dati/istruzioni richiede **due accessi alla memoria**: uno per la tabella delle pagine e uno per dati/istruzioni (indirizzo fisico).

Il problema dei due accessi alla memoria può essere risolto attraverso l'uso di una speciale piccola cache per l'indicizzazione veloce detta memoria associativa (o translation look-aside buffer – **TLB**).



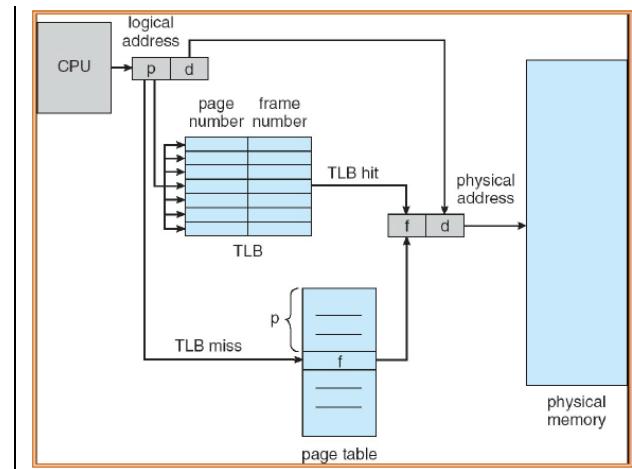
Prima dell' allocazione

Dopo l'allocazione

La **TLB** è una memoria associativa che viene posta vicino alla CPU, e quindi facilmente accessibile e non si trova in memoria principale, che contiene un numero limitato di entry, si segna le coppie accedute frequentemente.

Esiste un ASID che serve a identificare il processo e quindi evitare che un processo faccia riferimento a pagine conservative in tale tabella cache ma non appartenenti al processo in uso, più processi possono essere mantenuti nella TLB.

Se la TLB non consente l'uso dell'ASID, viene cancellato il contenuto di TLB ad ogni cambio di contesto.



TEMPO DI ACCESSO EFFETTIVO:

- *Tempo di accesso alla TLB (associative lookup)* = ε unità di tempo;
- *Tempo di accesso a memoria* = 1 unità di tempo;
- *Tasso di accesso con successo (hit ratio)*, frequenza delle volte in cui un particolare numero di pagina viene trovato nella TLB = α ;
- *Tempo di accesso effettivo (EAT)*:

$$EAT = (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) = 2 + \varepsilon - \alpha$$

La prima parte rappresenta il caso di successo, in cui con probabilità α trovo l'indirizzo nella TLB e quindi il tempo effettivo sarà $(1 + \varepsilon)$, la seconda parte rappresenta il caso di insuccesso, con probabilità $(1 - \alpha)$, il tempo effettivo sarà $(2 + \varepsilon)$.

PROTEZIONE DELLA MEMORIA PRINCIPALE:

La protezione della memoria centrale in ambiente paginato è ottenuta mediante **bit di protezione** associati ai frame.

Un bit di validità/invalidità è associato ad ogni elemento della tabella:

- “**valido**” indica che la pagina associata è nello spazio degli indirizzi logici del processo ed è quindi una pagina legale.
- “**non valido**” indica che la pagina non è nello spazio degli indirizzi logici del processo.

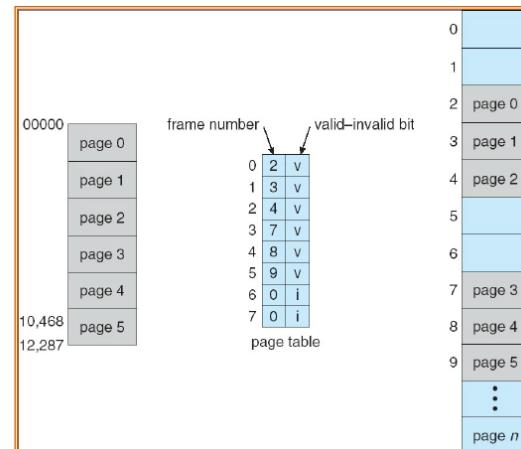
Capita raramente che un processo faccia uso di tutto il suo intervallo di indirizzi, gran parte della tabella rimane inutilizzata. Il registro della lunghezza della tabella delle pagine (PTLR) indica la dimensione della tabella delle pagine del processo. Permette di evitare grosse tabelle delle pagine con molte entrate associate a pagine invalide.

14 bit = lunghezza dell'
indirizzo logico
 2^{14} = spazio logico

size di un frame= size di
una pagina = 2048 = 2^{11}

size del processo =
 $10469 = (2^{11} \times 5) + 229 \Rightarrow$
pagine = 6 (cioè 12288 =
 $2^{11} \times 6$ byte)

$2^{14} / 2^{11} = 2^3 = 8$ pagine =>
nella PT ci sono 8 entry; di
queste ne vengono usate
solo 6 (quelle che possono
contenere 12288 byte)



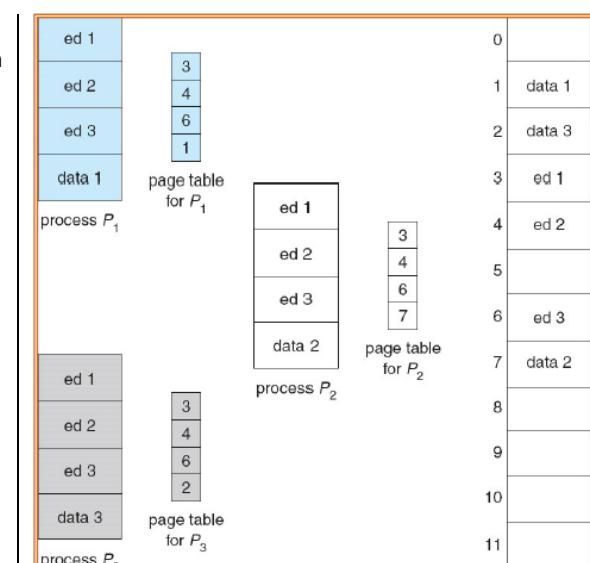
PAGINE CONDIVISE DAI PROCESSI:

Codice condiviso:

- Una copia di sola lettura di codice rientrante (i.e., non automodificante = non cambia durante l'esecuzione) condiviso fra processi (ad esempio editor, basi di dati).
- Le tabelle delle pagine dei processi associano le pagine logiche del codice agli stessi frame fisici.

Codice privato e dati:

- Ogni processo possiede una copia separata del codice e dei dati.
- Le pagine per il codice privato ed i dati possono apparire ovunque nello spazio di indirizzo logico



9.3 STRUTTURA DELLA TABELLA DELLE PAGINE

La tabella delle pagine ha un problema, ovvero che queste tabelle possono essere molto grandi, relative agli indirizzi logici.

Con pagine da 4 KB= 2^{12} ci sarebbero PT da $2^{32}/2^{12}=2^{20}$ entry, chiaramente sarebbe meglio non collocare in maniera contigua in memoria tale tabella.

Soluzioni: Tabella gerarchica, Tabella delle pagine con hashing e Tabella delle pagine invertita.

TABELLA GERARCHICA:

Suddivide lo spazio degli indirizzi logici in più tabelle di pagine, ad esempio la tabella delle pagine a due livelli.

Un indirizzo logico (su una macchina a 32-bit con pagine di 4K) è diviso in:

- un numero di pagina di 20 bit;
- uno spiazzamento della pagina di 12 bit.

Il numero di pagina è ulteriormente diviso in :

- un numero di pagina da 10 bit;
- uno spiazzamento nella pagina da 10 bit.

page number	page offset
p_1	p_2
10	10 12

dove p_1 è un indice per la tabella esterna, e p_2 rappresenta lo spostamento nella pagina della tabella interna.

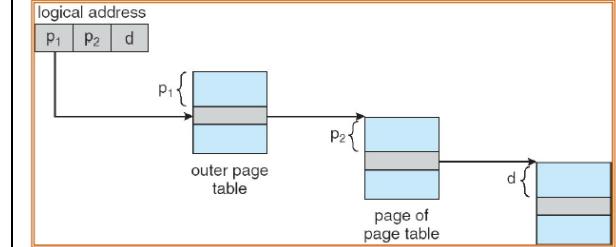
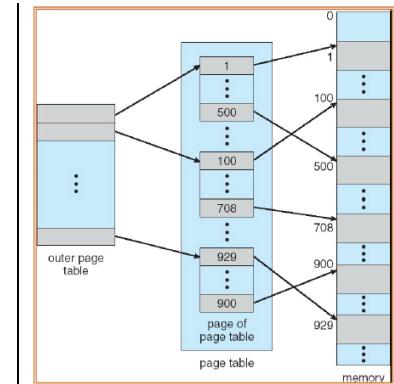


TABELLA DELLE PAGINE CON HASHING:

Comune per trattare gli spazi di indirizzamento più grandi di 32 bit. Ottima per spazi "sparsi".

Il numero di pagina virtuale p è l'input di una funzione hash. Il valore hash prodotto viene usato come indice nella tabella. Ogni elemento della tabella contiene una **lista** di coppie (numero pagina, frame) **concatenata**.

Il numero di pagina p viene confrontato con i numeri di pagina di questa lista, cercando una corrispondenza. Se viene trovata, il numero di frame fisico associato al numero di pagina viene estratto.

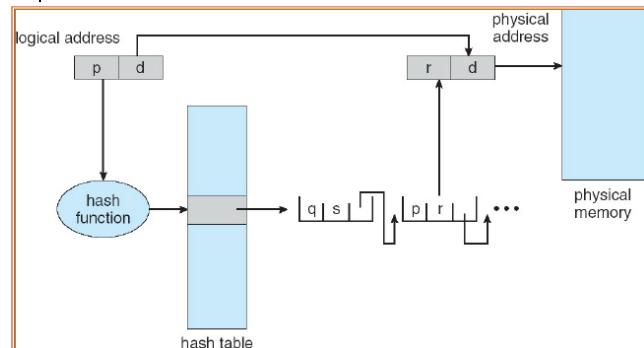


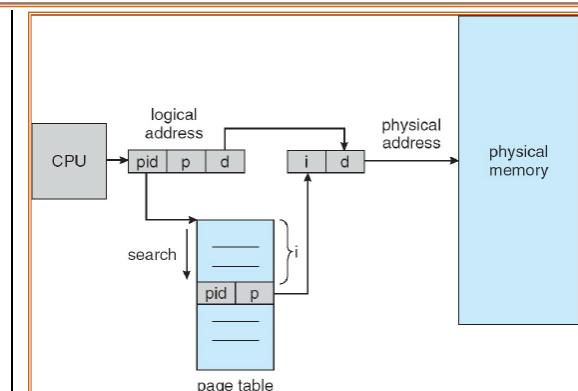
TABELLA DELLE PAGINE INVERTITA:

Un elemento per ogni frame della memoria centrale.

Ciascun elemento contiene il numero della pagina logica memorizzata in quel frame fisico, con informazioni sul processo cui appartiene quella pagina.

Questo schema permette di diminuire la quantità di memoria centrale necessaria per memorizzare le tabelle ma aumenta il tempo necessario per cercare nella tabella quando si verifica un riferimento alla pagina.

Usare una tabella con hashing per limitare la ricerca a uno o al più a pochi elementi nella tabella delle pagine.



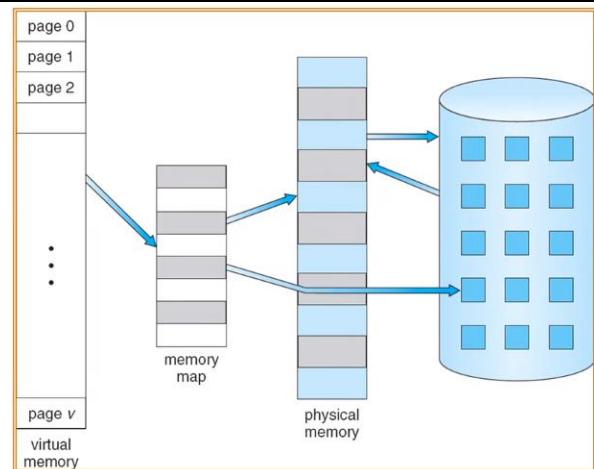
10. MEMORIA VIRTUALE

Quando un processo viene mandato in esecuzione gli si deve assegnare dello spazio (o contiguo o in frame) nella memoria principale. L'idea della memoria virtuale è che solo **una parte** del programma necessita di essere in memoria per l'esecuzione.

Supponiamo di avere un processo suddiviso in tante pagine (memoria virtuale), una memory map (page table), memoria fisica e poi il disco.

L'idea è caricare nella memoria principale solo un numero limitato di pagine del processo. Durante l'esecuzione, se c'è necessità di una nuova pagina la si prende dal disco, la si porta in memoria principale e si aggiorna la page table del processo, ovviamente bisogna avere un frame libero nella memoria principale per poter caricare la nuova pagina richiesta dal processo.

La memoria virtuale può essere implementata attraverso la **paginazione su richiesta (demand paging)** e **segmentazione su richiesta (demand segmentation)**.



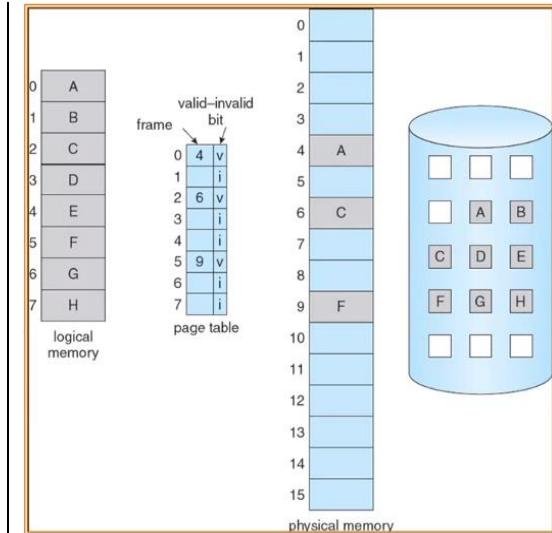
10.1 PAGINAZIONE SU RICHIESTA (DEMAND PAGING)

Un processo quando necessita di una pagina la richiede e il SO trova il modo di prendere la pagina, ancora sul disco, e portarla in memoria principale.

Per poter gestire quali pagine del processo sono realmente in memoria principale o meno, si utilizza il **bit di validità**.

Ad ogni elemento della page table è associato un bit (**v** → in-memoria, **i** → non-in-memoria). Inizialmente il bit è impostato a **i** per tutti gli elementi. Durante la traduzione dell'indirizzo, se il bit è **i** nell'elemento della tabella delle pagine, significa una mancanza di pagina e viene a crearsi un **page fault**.

Supponiamo un processo che richiede 8 pagine, con la sua page table, successivamente il processo quando parte ha caricato in memoria solo 3 pagine.



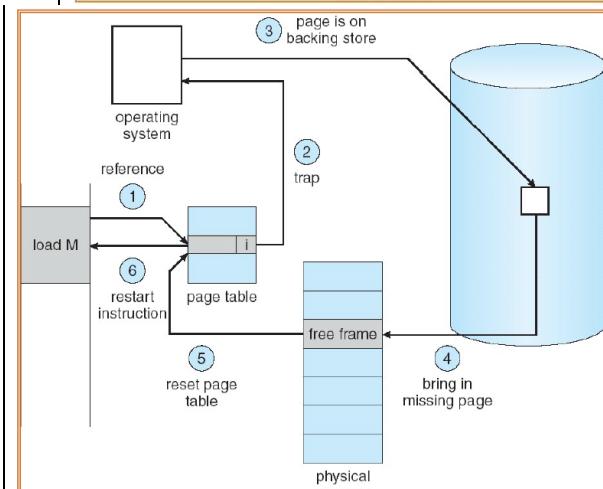
PAGE FAULT:

Quando avviene un **riferimento** ad una pagina non presente in memoria (cioè una pagina contrassegnata come non valida), si provoca una **trap** (segnale) al SO.

Il SO esamina una sua tabella interna per decidere cosa fare:

- Se si tratta di una pagina che non fa parte di quel processo, allora tale processo termina;
- Se la pagina è di quel processo ma c'è il bit di validità ad **i**, vuol dire che è in **page fault**.

In quest'ultimo caso, il SO deve porsi il problema di prendere quella pagina dal disco e portarla in memoria principale, così cerca un frame libero e sposta la pagina nel frame, dopodiché modifica la page table (scrivendo il numero del frame e mettendo a **v** il bit di validità), fatto tutto ciò bisogna riavviare l'istruzione che ha causato tale **page fault**.



PRESTAZIONE DELLA PAGINAZIONE SU RICHIESTA:

Probabilità di page-fault $0 \leq p \leq 1$:

- se $p = 0$ non ci sono mancanze di pagine
- se $p = 1$, ogni riferimento è una mancanza di pagina

Tempo di accesso effettivo (EAT):

$$EAT = (1 - p) \times \text{accesso alla memoria} + p (\text{page fault overhead} + \text{lettura della pagina} + \text{overhead ripresa})$$

Esempio:

Tempo di accesso alla memoria = 200 nanosecondi

Tempo medio di gestione di un page fault = 8 milisecondi = 8,000,000 nanosecondi

$$EAT = (1 - p) * (2 * 200) + p * 8,000,000 = (2 * 200) + p * 7,999,600 = 8199,8 \text{ nanosecondi}$$

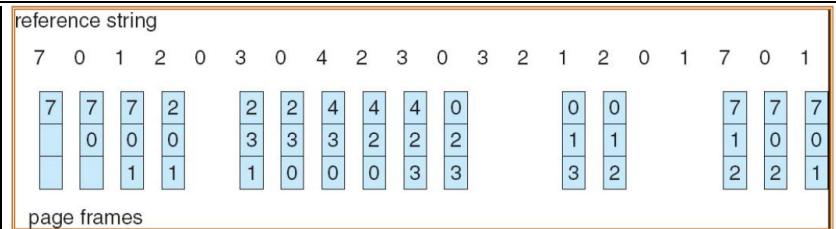
SOVRALLOCAZIONE:

Questa situazione si verifica quando non sono più disponibili frame liberi in memoria. L'unica soluzione è sostituire delle pagine già presenti in memoria principale, in base a determinati criteri. Un algoritmo che implementa questa soluzione: individua la posizione della pagina desiderata sul disco, se non c'è un frame libero si usa un algoritmo di sostituzione (altrimenti si procede normalmente) che sceglie il **frame vittima**, carica la pagina desiderata nel frame libero (aggiornando la page table) e si riprende il processo.

Se bisogna sostituire una pagina, in teoria, si prende questa pagina, la si porta sul disco e si inserisce la pagina nuova, ma questo è necessario solo quando tale pagina viene modificata, così si aggiunge alla page table il **bit di modifica**, oltre al bit di validità, che indica se la pagina è stata modificata.

10.1.1 SOSTITUZIONE FIFO DELLA PAGINA

Avendo un numero fissato di frame, se si verifica un page fault, il SO sceglie il frame vittima seguendo il criterio FIFO, la prima pagina inserita sarà la prima ad essere sostituita, e così via...



Avendo la seguente stringa di riferimento: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Se ho 3 frames (3 pagine possono essere in memoria per ogni processo):

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

Se ho 4 frames:

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

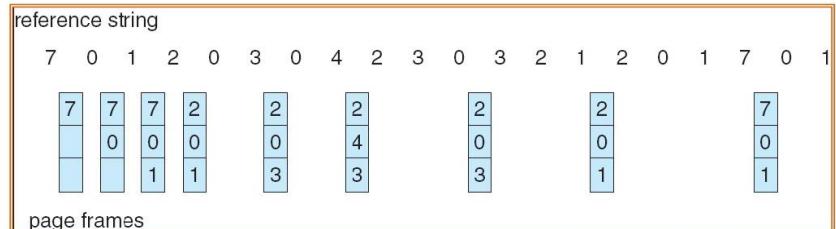
In questo caso si ha 9 page fault.

Con tale algoritmo di sostituzione si ha la **Belady's Anomaly**, ovvero più frame più page fault.

10.1.2 SOSTITUZIONE OTTIMALE DELLE PAGINE

Sostituisce la pagina che non si userà per il periodo di tempo più lungo.

Questo algoritmo richiede la conoscenza della sequenza di riferimenti futuri. Usato per misurare quanto sono buone le prestazioni di un algoritmo.



Avendo la seguente stringa di riferimento: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

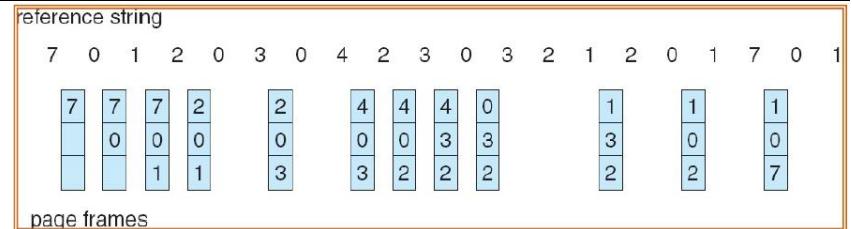
Se ho 4 frames:

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

In questo caso ho 6 page fault.

10.1.3 SOSTITUZIONE LRU (LAST RECENTLY USED) DELLE PAGINE

Sostituisce la pagina che non è stata usata per il periodo di tempo più lungo.



Avendo la seguente stringa di riferimento: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Se ho 4 frames:

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

IMPLEMENTAZIONE LRU:

L'implementazione di tale algoritmo, esistono diverse tecniche, la prima è attraverso un **contatore**:

- Si assegna alla CPU un contatore che si incrementa ad ogni riferimento alla memoria;
- Si assegna ad ogni entry della tabella delle pagine un ulteriore campo;
- Ogni volta che si fa riferimento ad una pagina viene copiato il valore del contatore della CPU nel campo corrispondente per quella pagina nella tabella delle pagine;
- Quando una pagina deve essere sostituita si guardano i valori dei contatori, scegliendo la *pagina con il valore del contatore più basso*.

La seconda implementazione è attraverso uno **stack**:

- Mantenere uno stack dei numeri di pagina in una lista a doppio collegamento;
- Riferimento ad una pagina: se è una pagina non presente nei frame la si mette in cima allo stack, ogni volta che si fa riferimento ad una pagina già presente nello stack bisogna prelevare tale riferimento dalla sua posizione e metterla in cima, nessuna ricerca per la sostituzione: basta *prelevare la pagina dal fondo*;
- Richiede di cambiare al più 6 puntatori.

Le implementazioni con contatore e stack richiedono molte risorse, una soluzione più semplice è quella di assegnare alla page table, oltre al bit di validità e modifica, un **bit di riferimento** che inizialmente è 0 quando la pagina viene caricata la prima volta in memoria, quando viene fatto riferimento nuovamente a questa pagina, viene messo il bit ad 1 e nel momento in cui si deve scegliere una pagina da eliminare si va a scegliere la pagina che ha questo bit di riferimento a 0, perché significa che oltre al momento iniziale che si setta a 0, tale pagina non è stata proprio usata.

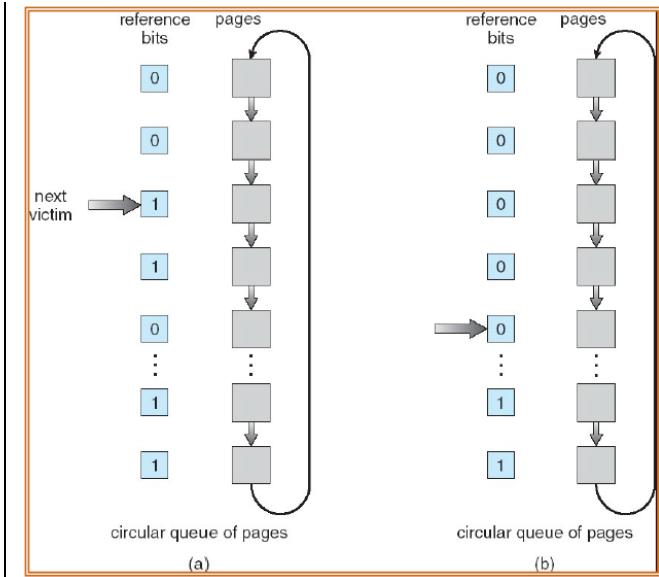
Questa soluzione semplice è una approssimazione, perché non è detto che si trova solo un bit posto a 0, ma possono essere molteplici, e nasce il problema di quale scegliere. Tale approssimazione potrebbe essere migliorata con l'algoritmo della seconda chance.

ALGORITMO DELLA SECONDA CHANCE:

Le pagine vengono predisposte in una sorta di lista circolare, e quando si seleziona una pagina vittima si inizia la scansione della lista.

Si parte da una certa posizione, quando si cerca una pagina vittima si cerca la pagina che ha il bit di riferimento ad 1, lo si fa diventare 0 (seconda chance) e si prosegue con la scansione, così con tutti gli altri bit ad 1.

Alla prima pagina che ha bit 0, questa sarà la vittima.



RAFFINAMENTO DELL'ALGORITMO DELLA SECONDA CHANCE:

Invece di considerare solo il *bit di riferimento*, si valuta anche il **bit di modifica** in una coppia (**bit riferimento, bit modifica**):

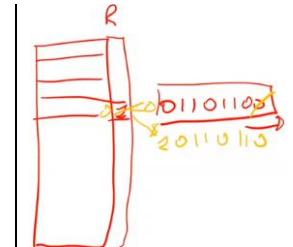
- (0,0) non usato di recente, non modificato
- (0,1) non usato di recente, modificato
- (1,0) usato di recente, non modificato
- (1,1) usato di recente, modificato

La pagina che verrà eliminata sarà quella con la **coppia di valore minore**, chiaramente la pagina che non è stata modificata costa meno perché consente di avere un accesso in meno al disco durante la sostituzione.

Un altro raffinamento potrebbe essere quello di avere più *bit di riferimento* (tenuti in un **registro a 8 bit**).

In corrispondenza ad una pagina p si ha il bit di riferimento ed un registro a 8 bit, associato alla pagina.

Ad intervalli regolari, si prende il contenuto del registro e lo si shifta a destra di una posizione, shiftando anche il bit di riferimento. La pagina da eliminare, in questo caso, sarà quella col valore di registro più basso.



ALGORITMO DI CONTEGGIO:

Esistono altri tipi di algoritmi, a parte LRU, che tengono un contatore del numero di riferimenti che sono stati fatti ad ogni pagina:

- **Algoritmo LFU (Least Frequently Used)**: sostituisce la pagina con il più basso conteggio;
- **Algoritmo MFU (Most Frequently Used)**: sostituisce la pagina con il conteggio più alto, è basato sulla assunzione che la pagina col valore più basso è stata spostata recentemente in memoria e quindi deve ancora essere impiegata.

BUFFERIZZAZIONE DELLE PAGINE:

Per semplificare la scelta delle *pagine vittima* si utilizza un buffer, ovvero si considera un **pool di frame liberi** per soddisfare le richieste velocemente.

Quando si sceglie una pagina vittima, invece di inserirla nel disco, la si trasferisce nel pool dei frame liberi, in questo modo se si fa riferimento a tale pagina successivamente il SO va vedere se è presente nel pool, in caso affermativo la prende e la porta in memoria evitando l'accesso al disco.

Il processo può essere riattivato rapidamente, senza attendere la fine dell'operazione di salvataggio del frame vittima sulla memoria di massa.

Le pagine modificate del pool dei frame liberi sono scritte sul disco periodicamente in background.

10.2 ALLOCAZIONE DEI FRAME

Ogni processo ha bisogno di un numero **minimo** di pagine.

Esempio: IBM 370 – 6 pagine per gestire l'istruzione MVC:

- L'istruzione richiede 6 byte, che possono estendersi su 2 pagine.
- 2 pagine per gestire il "from" dell'istruzione.
- 2 pagine per gestire il "to" dell'istruzione.

Quindi significa che per gestire tale operazione servono almeno 2 pagine.

Però bisogna stabilire i criteri per poter assegnare i frame ai vari processi, i principali sono **allocazione fissa** e **allocazione a priorità**.

ALLOCAZIONE FISSA:

- **Allocazione uniforme**, avendo m frame ed n processi, si assegnano m/n frame a ciascun processo.

Esempio: se 100 frame e 5 processi, ognuno prende 20 frame.

- **Allocazione proporzionale**, si assegna la memoria disponibile ad ogni processo in base alle dimensioni di quest'ultimo, ovvero più un processo richiede spazio e più sarà lo spazio assegnatogli.

$$s_i = \text{size del processo } p_i$$

$$S = \sum s_i$$

m = numero totale di frames

$$a_i = \text{numero di frame allocati per } p_i = \frac{s_i}{S} \times m$$

Esempio:

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

ALLOCAZIONE A PRIORITÀ:

Usare uno schema di allocazione proporzionale basato sui valori delle priorità piuttosto che delle dimensioni, esempio processo di sistema, più frame si assegna al processo e maggiore sarà la sua velocità di esecuzione.

SOSTITUZIONE GLOBALE E LOCALE:

Quando si sceglie la pagina vittima la si può scegliere in due ambiti:

- **Sostituzione locale**, vengono considerati solo i frame allocati al processo.
- **Sostituzione globale**, permette di selezionare un frame di sostituzione a partire dall'insieme di tutti i frame, anche se quel frame è correntemente allocato a qualche altro processo, cioè un processo può prendere un frame da un altro processo.

In quest'ultimo caso, un processo non può controllare la propria frequenza di page fault ed il tempo di esecuzione di ciascun processo può variare in modo significativo.

10.3 THRASHING

Se un processo non ha abbastanza pagine, il tasso di page fault aumenta. Questo comporta:

- riduzione utilizzo della CPU;
- il sistema operativo ritiene che sia necessario aumentare il livello di multiprogrammazione;
- un altro processo aggiunto al sistema.

Thrashing ≡ si spende più tempo nella gestione della paginazione che nella esecuzione dei processi.

Per evitare il thrashing occorrerebbe sapere quali, oltre che quante pagine, servono.

Modello di località: Una località è un insieme di pagine che vengono accedute insieme, che sono contemporaneamente in uso attivo, il processo si muove da una località all'altra e le località possono sovrapporsi.

Perché si verifica il thrashing? **dimensione della località > numero di frame assegnati**

Esempio:

Quando viene invocato un sottoprogramma, si definisce una nuova località: vengono fatti riferimenti alle sue istruzioni, alle sue variabili locali ed a un sottoinsieme delle variabili globali. Quando il sottoprogramma termina, il processo lascia la località corrispondente.

Le località sono definite dalla struttura del programma e dei dati.

MODELLO DEL WORKING-SET:

Per tentare di gestire il thrashing è utilizzare il **modello del working-set**:

- **Δ**: Si considera una finestra temporale Δ chiamata **finestra working-set**, fondamentalmente è un numero fisso di riferimenti di pagina, ad esempio 10.000 istruzione, cioè ogni 10.000 riferimenti a pagina viene eseguita un'istruzione;
- **WSS_i**: Si individua il numero totale di pagine cui un processo P_i fa riferimento nell'intervallo Δ , cioè si considera il tempo partizionato in queste finestre (intervalli di 10.000 istruzioni) e si va a contare il numero delle pagine che ciascun processo P_i fa riferimento.
 - Se Δ è troppo piccolo non comprenderà l'intera località;
 - Se Δ è troppo grande può sovrapporre parecchie località;
 - Se $\Delta = \infty \rightarrow$ il working set è l'insieme delle pagine toccate durante l'esecuzione del processo.

A questo punto, si prende il WSS di ciascun processo e li si somma: $D = \sum WSS_i \equiv$ richiesta globale dei frame.

Dopo si va a confrontare D con il numero reale di frame nel disco : **m = numero di frame**.

- Se $D > m \rightarrow$ Thrashing, cioè il numero totale di pagine che sono state richieste dai vari processi è maggiore rispetto al numero di frame disponibili;
- Se $D < m$, allora occorre sospendere uno dei processi.

Per individuare quali sono le pagine che sono state utilizzate nella finestra di tempo Δ :

Man mano che la finestra scorre può aumentare o diminuire il numero di frame dati al processo.

Nell'esempio in t_1+1 la pagina 2 scompare, quindi il frame ad essa allocato viene considerato libero (può eventualmente essere usato da un altro processo il cui working set invece cresce).

