Appunti SO

La shell è un’interfaccia per comunicare con la macchina attraverso comandi in un terminale, ed è l’interfaccia con il sistema operativo stesso. In Linux è rappresentato dal terminale, mentre ora, in windows il terminale è solo un’emulazione di una shell; quindi, si avvicina di più ad un linguaggio ad alto livello piuttosto che ad una shell.

La shell è quindi un programma che è composto da un ciclo che aspetta l’input del client, interpretandolo e offrendo il servizio richiesto dall’utente.

Le varie versioni delle shell variano per la qualità dell’interfaccia offerta, per esempio l’autocompletamento dei comandi tramite il tasto TAB, o anche l’history dei comandi accessibili con freccia su o giù. Insomma, aggiunge utility per il client.

La shell è utile per poter accedere a computer remoti, operare su file che altrimenti avrebbero un peso troppo grosso sul calcolo

I comandi della shell:

sintassi: **comando opzioni argomenti**

Le istruzioni si dividono in:

* Comandi built-in della shell (es. ls/cd) = non hanno un eseguibile ma sono inglobati nella shell
* Script
* Codice eseguibile

Es. sort -n num\_telefoni > num\_tel.ordinati:

* Sort = funzione di ordinamento
* -n = opzione di ordinamento -> ordinarli come numeri, ovvero il numero più piccolo viene prima
* Num\_telefoni = argomento in input
* > = ridirezione dell’output in un file specifico
* Num\_tel.ordinati = nome del file di ridirezione, che se non è esistente allora viene creato, ma se esiste viene troncato a zero(dimensione 0) e poi viene scritto

Es. lp -d lp1 -h file1 file2 file3

* Lp =line print
* -d
* Lp1 = argomento dell’opzione -d
* -h
* File1 file2 file3 = argomenti in input

Nella shell l’analizzatore lessicale analizza la stringa inserita, cercando e riconoscendo dei **token**, come appunto il primo token: il nome del programma. Oppure un secondo token può essere un’opzione individuata dal carattere **–** che li precede

Riconosce che esiste o meno un comando grazie al PATH, che è una variabile che indica le directory nel quale cercare, eseguendo il primo trovato.

Una directory non è un contenitore fisico ma logico, e contiene soltanto i nomi dei file, non i file stessi. Quindi muovendo un file da una directory ad un’altra è necessario soltanto spostare il nome il nome del file dalla prima alla seconda.

Esempi di comandi della shell:

**File system:** ls, cp, mv, rm, mkdir.

**Accesso:** chmod, chown. -> chmod modifica i permessi, chown modifica il proprietario

**Controllo del disco:** df, mount. -> mount aggiunge all’albero principale ( C: ) un sottoalbero (D:) che è un dispositivo diverso da quello principale nel quale è scritto il file system principale

**Controllo dei processi:** ps, kill, bg, fg. ->

**Rete:** ifconfig, ping, tracepath.

**File System**

Il file system è la parte più vista dall’utente del SO, un file non è altro che un “contenitore” di dati: caratteri, immagini, programmi e altro.

Il file è caratterizzato da un insieme di attributi (file control block):

* Nome
* Identificatore = identificatore numerico assegnato dal SO al file
* Locazione:
  + Posizione fisica = dove si trova fisicamente il file
  + Posizione logica = il path
* Dimensione
* Protezione = identifica i permessi per leggere, scrivere o visualizzare il file

Operazioni sui file

* Creazione di un file = il SO deve trovare uno spazio adatto per poter ospitare questo file (sia per il contenuto del file che del file control block), trovando e componendo l’accesso alla directory, stabilendo le modalità di accesso al file (file control block)
* Scrittura/Lettura = il SO deve: trovare lo spazio per ampliare il file, deve avere una system call per potersi posizionare nella posizione desiderata dall’utente
* Riposizionamento
* Cancellare un file = recuperare lo spazio precedentemente occupato dal file, cancellando anche il nome all’interno delle directory nelle quale era inserito
* Troncamento = cancellare il contenuto del file, ovvero lasciare intatto il file control block e il nome del file nella directory

Apertura dei file:

Il SO deve avere due strutture dati per poter ordinare i processi aperti:

* Tabella dei processi aperti (in generale nel pc)
* Tabella dei processi aperti dal programma

Un file aperto richiede diversi dati:

* Contatore di open = Ovvero quanti programmi hanno aperto ed utilizzano un file, per sapere se modificare o meno il file
* Locazione dei file su disco
* Diritti di accesso
* File pointer = Puntatore all’ultima locazione letta/scritta da ogni processo (ognuno indipendente)

In Unix, nella **tabella dei processi aperti** (del sistema = process table entry) ad un’entry precisa viene associato un puntatore che indica ad una tabella (unica per ogni entry) che contiene dei **flag di stato** del file (solo lettura, scrittura), il **current offset** che indica la locazione del file, e infine **un secondo puntatore** che punta ad un’ennesima tabella che contiene gli attributi del file, unica per ogni file (V-node table): **informazioni control block** (i-node information) che tra l’altro contiene anche la **dimensione del file**, **informazioni del file** (v-node information).

Metodi di accesso:

* Sequenziale
* Accesso diretto = accedere ad un settore senza per forza dover leggere tutti gli altri, un settore (o blocchi) è una locazione di memoria nei quali sono memorizzati i file o i dati richiesti

Attributi di una directory:

* Nome
* Indirizzo
* Lunghezza corrente (numero di file contenuti)
* Data ultimo accesso = Data ultima lettura/accesso alla directory
* Data ultimo aggiornamento = Data ultima modifica ad uno dei file della directory (creandone o rimuovendone uno)
* ID proprietario
* Informazioni sulla protezione

Unix è organizzato in modo che tutti gli i-node vengono memorizzati in una locazione di memoria all’inizio, memorizzati nell’i-node list, poi vengono memorizzati in dei blocchi di memoria dei directory block i nomi dei file e le locazioni dei file nell’i-list, usato per accedere ad una locazione di memoria precisa dove si trovano fisicamente i dati del file

Unix poi indica la locazione corrente un’entry per se stesso(.) e uno per il padre(..)

Operazioni su una Directory:

* Ricerca di un file
* Creazione di un file
* Cancellazione di un file
* Elencare il contenuto della directory
* Rinomina di un file
* Attraversamento del File system

Directory a singolo livello: tutti i dati e i file contenuti in una singola directory

Directory a due livelli: permette di avere più utenti che lavorano in locazioni di memoria diversi, quindi una directory punta ad una seconda

Directory ad albero: in una directory, organizzata gerarchicamente partendo dalla root, vengono memorizzati file e sottodirectory

Directory a grafo aciclico: A differenza della struttura ad albero possiede dei file condivisi tra più sottodirectory (hard link), ovvero ad un file si può accedere attraverso più di una directory differenti

Grafo Ciclico: una sottodirectory può contenere anche un collegamento alla sua directory padre

**Montaggio di un file system:**

Viene “montato” il file system contenuto di una periferica esterna (USB) sul file system centrale del PC, ovvero viene aggiunto l’albero di sistema della memoria esterna come sottoalbero a quello di sistema centrale, in UNIX viene montato in una directory specificata o in una di default, queste due sono detti punti di montaggio

Montando un file system in una directory che conteneva oggetti, questi dopo il montaggio scompaiono fino allo smontaggio del file system esterno

**Protezione dei file:**

User ID: id di un utente preciso, utili per poter associare un insieme di processi di accesso ad un utente preciso

Group ID: id che specificano un gruppo di utenti, servono per poter modificare i permessi di accesso ad un gruppo di utenti

Esistono tre tipo di classi di utenti: RWX (read, write, execute)

* Accesso proprietario 7 -> 111
* Accesso gruppo 6 -> 110
* Accesso pubblico 1 -> 001

Chmod 761 file -> modifica i permessi di accesso del file in modo che il proprietario abbia 7 di accesso, il gruppo 6 e il pubblico 1

7 identifica l’accesso concesso per il proprietario (essendo root può sempre modificare)

6 accesso concesso ad un gruppo

1 accesso concesso al gruppo

Implementazione del File System

Il file system definisce l’aspetto del sistema per gli utenti, è memorizzato in un’unità di memorizzazione secondaria: per leggere quindi è necessario portarlo in memoria principale, e per modificarlo va prima portato in memoria principale e poi riscritto nella posizione precisa prima della modifica nella memoria secondaria

Viene organizzato in livelli:

1. **Application program**
2. **Logical file system** -> gestisce i metadati, directory, sicurezza (controllando se si hanno i permessi per modificare o accedere al file), strutture e gestisce quindi il file control block da trasferire al livello inferiore
3. **File-organization module** -> traduce gli indirizzi dei blocchi logici in quelli fisici (ovvero attraverso le informazioni logiche del file che vogliono essere tradotte e assegna un indirizzo fisico nel disco), gestendo lo spazio libero
4. **Basic file system** -> comunica ai driver le posizioni precise per leggere o scrivere nel disco fisico
5. **I/O control** -> è il driver del dispositivo si occupa del trasferimento di informazioni tra memoria secondaria e principale, gestisce gli interrupt
6. **Devices**

Sul disco sono presenti:

* Boot control block -> informazioni necessarie per il caricamento del SO
* Volume control block -> Contiene le informazioni sul volume, il numero, la taglia e i blocchi liberi di una partizione

In memoria principale sono presenti invece:

* Mount Table -> punti di montaggio
* Cache per le strutture delle directory recentemente accedute
* Tabella dei file aperti di sistema
* Tabella dei file aperti per processo

**Metodi di allocazione**

L’allocazione su disco può avvenire in 3 metodi, le principali differenze tra questi sono i tempi di accesso ai file, ma una non esclude l’altra in un SO:

* Allocazione contigua:
  + Vantaggi:
    - Ogni file occupa un certo di numero di **blocchi contigui su disco** (necessariamente posti uno dopo l’altro)
    - **ha come pro di essere semplice** poiché basta conoscere la posizione del primo blocco per poter risalire agli altri nota la lunghezza (in blocchi) del file
    - l’accesso può essere sequenziale o diretto:
      * sequenziale: il file system memorizza l’ultimo blocco a cui si è acceduto
      * diretto: dato il blocco di inizio b e il blocco desiderato i allora si accede direttamente all’indirizzo del blocco b+i-1
  + Mapping logici/fisici:
    - sia LA un indirizzo logico di un byte, e 512 byte la dimensione di un blocco fisico, data LA/512, il **quoziente** se **sommato** al **blocco di partenza** rappresentano il **blocco al quale accedere**, mentre il **resto** **l’indice** del blocco (es. LA=500, 500/512-> Q= 0 allora il blocco da accedere è b+0=b mentre l’indice del blocco è 500)
  + **Svantaggi**:
    - L’allocazione contigua, per via della sua natura lascia liberi spazi in seguito all’allocazione e la dislocazione dei file che possono non essere abbastanza per permettere di allocare altri file, rendendo lo spazio di questi buchi inutilizzabile (**frammentazione esterna**), inoltre un file potrebbe non avere la possibilità di espandersi = la taglia non potrebbe crescere. (**frammentazione interna**)
* Allocazione linkata:
  + Nasce per risolvere la frammentazione dell’allocazione contigua
  + Vantaggi:
    - Ogni file è una lista linkata di blocchi: ogni singolo blocco ha un puntatore al suo successivo = i blocchi del file possono essere sparpagliati in giro per il disco, per questo metodo nel file è necessario che il file nel suo file control block sia inserito la sua dimensione in blocchi, con l’ultimo blocco che ha come puntatore NULL al suo successivo per indicare la fine della lista
  + Svantaggi:
    - L’efficienza nell’accesso è rallentata dallo scorrimento di tutta la lista
    - Uno spazio dei blocchi viene *persa* per la memorizzazione dei puntatori
    - Se per qualche errore viene corrotto uno dei blocchi, viene perso tutto il resto del file dopo quel blocco in quanto il puntatore diventa nullo
  + Mapping logici/fisici:
    - Considerando sempre lo spazio di 512 byte per il blocco, togliamo 1 byte usato per il puntatore, e sia LA sempre un indirizzo logico: LA/511 produce un quoziente Q e un resto R, dove Q+1(numero di accessi al disco) indica il blocco al quale accedere, mentre R rappresenta l’indice del blocco (es. sia LA=500, 500/511 -> Q=0, R=11, allora Q+1=1 il blocco a quale accedere e 11 l’indice del byte a cui vogliamo accedere)
* Allocazione FAT:
  + Per risolvere questa inefficienza, Windows utilizza la FAT (**File Allocation Table**), la memorizzazione resta sparsa per il disco in modo uguale, ma l’importante differenza è appunto la tabella con tutti gli indici che **puntano ad ogni blocco**, quindi migliorando **l’accesso diretto**, ed essendo già in **memoria principale** non ha bisogno di accessi ulteriori che peggiorano l’efficienza.  
    Quindi si scorre la tabella con i puntatori della memoria principale seguendone gli indirizzi scritti in ogni cella, e arrivati al blocco richiesto **si accede** al disco **direttamente a quel blocco**.
    - **Vantaggi**: risiede nello scorrimento della tabella contenuta nella memoria principale che ha costo nullo
    - **Svantaggi**: la sua dimensione, infatti nel caso della FAT32 si hanno 32bit per la memorizzazione degli indirizzi = 2^32 = **4GB**
* Allocazione indicizzata
  + Ad ogni file viene allocato un **blocco extra** utilizzato come **indice** per tutti gli altri blocchi che lo compongono, e nel FCB (File control block) viene specificato **l’indirizzo** di questo **blocco indice**, che viene **portato** subito in **memoria principale**, evitando accessi inutili, permettendo quindi di **accedere** poi **direttamente** al **blocco desiderato** dal disco
    - Svantaggi: Se un file cresce troppo, oltre la possibilità del blocco indice, questo non può essere memorizzato
* Una **soluzione** per l’allocazione indicizzata è la concatenazione dei blocchi indice, riservando l’ultima parte del blocco per puntare a quello successivo.
  + **Mappatura**: LA/(511\*512) -> Q e R come risultato, dove Q indica il blocco indice al quale accedere, mentre poi bisogna eseguire R/512 -> la divisione produce Q2 ed R2; Q2 indica il blocco della tabella indice ed R2 indica l’indice nel blocco del file
* Un’ennesima soluzione è l’indicizzazione a due livelli, ovvero: bisogna allocare un blocco indice per i blocchi d’indice che hanno funzionamento analogo a quello prima descritto, solo che invece di 511 vengono utilizzati a pieno in quanto l’indice di questi blocchi viene memorizzato nel blocco indice più esterno
  + **Mappatura**: LA/(512\*512) -> Q1 e R1, dove Q1 indica qual è il blocco indice interno, poi R1/512 produce -> Q2 ed R2, Q2 indica qual è il blocco nel disco ed R2 indica l’indirizzo nel blocco file

UNIX utilizza 15 puntatori per indirizzare ai blocchi nel FCB, 12 sono ad indirezione diretta, 1 a singola indirezione, 1 a doppia e 1 a tripla

**Gestione dello spazio libero**

* Vettore di bit:
  + Assegno un bit per ogni blocco, con tanti bit quanti sono i blocchi sul disco, dove in ogni cella 0 indica blocco occupato ed 1 libero, viene caricato in memoria principale all’avvio della macchina e viene scritto su disco allo spegnimento
    - Svantaggi: utilizzo di memoria direttamente proporzionale alla dimensione del disco, es:
      * dimensione blocco = 212, dimensione disco 232, 232/212=218 bit = 32Kb utilizzati
* Lista dei blocchi liberi:
  + Si conserva un puntatore ad una lista di puntatori che mirano ai blocchi liberi della memoria, e nella memoria principale è salvato la testa della lista

**Gestione della memoria di massa (Silberschatz Cap. 12)**

Esistono due tipi di dispositivi:

* CVL (cosntant linear velocity) -> la densità di bit per traccia è uniforme
* CAV (constant angular velocity) -> la densità di bit delle tracce più interne è maggiore rispetto alla densità di quelle più esterne

La formattazione fisica – o a basso livello – divide il disco in **settori** letti e/o scritti dal controller, di taglia solitamente 256, 512, 1024 byte

La formattazione logica è la suddivisione in partizioni

**Gestione del disco**

Esiste una partizione di avviamento che contiene il SO, e il programma di avviamento è contenuto in un **boot block**

Per risolvere il problema dei **settori difettosi** esiste la tecnica del **sector sparing**, ovvero di sostituire ad ogni richiesta di un settore definito e identificato come difettoso, propone un blocco di rimpiazzo alla richiesta, questi presi dai **blocchi di riserva** che vengono riservati durante la formattazione

Un’altra tecnica è quella del **sector slipping** ovvero, slittare tutti i settori danneggiati di uno

**Scheduling del disco**

Per efficientare il lavoro del disco, che ha come difetti quelli del tempo di ricerca (**seek time**) = tempo che impiega la testina per arrivare alla lunghezza del blocco desiderato

Per descrivere il lavoro della testina esistono vari algoritmi di scheduling:

* **FCFS** = first comes first served, ovvero esegue e cerca ogni blocco cercato in ordine in cui si sono posizionati nella coda di richieste **poco efficiente**
* **SSTF** = shortest seek time first, ovvero in base alla posizione corrente, viene soddisfatta la richiesta che ha il minor tempo possibile di seek time, rendendo più efficiente la coda evitando movimenti superflui della testina **più efficiente** **ma** di **contro**, questa schedulazione può causare un’attesa indefinita per richieste di blocchi molto lontani dalla testina poiché verrà sempre e solo soddisfatta la richiesta più vicina alla testina – algoritmo comune e naturale
* **SCAN** = parte da un estremo e lo soddisfa, e scende fino ad arrivare all’estremo più piccolo, poi ritorna ad andare all’esterno e man mano che incontra richieste le soddisfa, come **difetto** presenta uno spreco di tempo in quanto passa più volte su tracce con richieste appena svolte.
  + Una soluzione è quindi farlo partire dall’esterno del disco e farlo arrivare all’inizio, facendolo ripartire dalla fine invece che invertire la marcia, questo algoritmo è detto **C-SCAN** – sia scan che c-scan sono usati in sistemi a carichi pesanti sui dischi
* **C-LOOK** = è un raffinamento della C-SCAN, invece di arrivare all’estremo del disco, si sposta tra gli estremi delle richieste invece che tra gli estremi minimo e massimo del disco – insieme a SSTF quelli fra i più usati

**Unità a stato solido**

A differenza dell’hard disk questi non presentano i problemi di scheduling per la testina, in quanto non ha componenti meccaniche o elettromagnetiche, si basa su una memoria flash (portandoli più vicini ad una ram piuttosto che ad un hard disk)

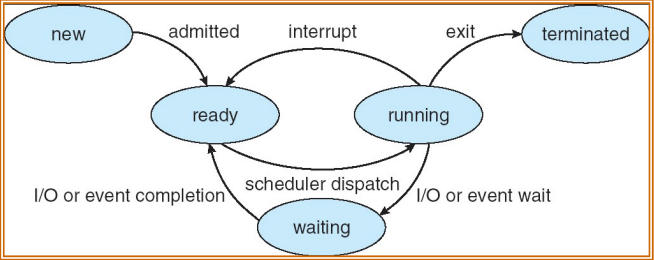
**Processi**

Un processo nasce dalla richiesta di un programma in esecuzione, un programma quindi può dare vita a più processi.

All’esecuzione di un programma, il SO cerca lo spazio nella memoria principale per poterlo avviare, nel caso sia disponibile viene allocato uno spazio per un processo, organizzato (partendo dal basso):

* Il codice
* Le variabili = heap
* Uno spazio libero (contiguo) per poter memorizzare dati che possono servire (che sono necessari) all’esecuzione del programma, come anche per le chiamate di una funzione (Stack)

Il processo segue una serie di stati:

* **New** = processo appena creato -> allocato lo spazio in memoria e scritte le informazioni base scritte sopra
* **Ready** = pronto per l’esecuzione del programma, aspetta soltanto gli venga assegnata la CPU per lavorare
* **Running** = nel caso venga assegnata la CPU al processo (**scheduler dispatch**), comincia l’esecuzione -> vengono caricati vicino alla CPU i registri che erano memorizzati nella locazione del processo, dallo stato running può uscire per tre motivi:
  + Uscita per la fine del programma -> **fase terminated**
  + Uscita per interrupt -> All’arrivo di un interrupt la CPU finisce di lavorare sul processo e lo fa tornare allo stato **ready**
  + Uscita per un evento I/O o di attesa (system call, o input/output) -> porta il processo nello stato di **Waiting**
* Se il processo si trova in **Waiting**, finito l’evento di attesa il processo può tornare ad aspettare nello stato **ready**

All’uscita dallo stato di running, nel caso del rientro nello stato di ready, al processo viene assegnato un PCB (Process Control Block) che contiene una serie di informazioni (come un record) che riguardano il processo, contenendo informazioni necessarie per riperendere l’esecuzione del programma dall’ultima esecuzione, organizzato così:

* Process state -> stato del processo
* Process number -> identificatore (univoco) del processo
* Program Counter -> puntatore all’ultima posizione della CPU in quel processo prima dell’uscita dallo stato di ready
* Registers -> registri della CPU riguardanti quel processo durante lo stato di running
* Memory Limits -> Impone gli estremi dello spazio di memoria nel quale eseguire il programma, insomma per informare la CPU dove trovare lo stack, l’heap e tutte le altre informazioni del processo
* List of open files -> Lista dei file aperti necessari per l’esecuzione del processo

L’utilità del PCB è quella di “fotografare” il processo nel momento dell’uscita dallo stato di running, per poter riprendere l’esecuzione del processo come prima della perdita della CPU

La CPU quando esegue un cambio di processo in esecuzione:

1. Salva lo stato del processo P0 in un PCB
2. Carica il PCB (nel caso il processo sia nuovo, il PCB è vuoto e viene riempito in seguito) del secondo processo (P1) da eseguire (cambio di contesto)
3. Esegue il processo P1
4. Nel caso avvenga un interrupt o qualche evento che ne causi l’uscita, viene salvato nel PCB lo stato di P1
5. Intanto P0 è in fase di attesa per ritornare ad essere eseguito
6. Supponendo gli venga assegnata la CPU: Viene caricato il PCB di P0 per ritornare allo stato del programma in esecuzione (cambio di contesto)

Esistono tre tipi di code di processi:

* **Job Queue** = coda di tutti i processi nel sistema
* **Ready Queue** = coda di tutti i processi pronti per l’esecuzione (in memoria centrale)
* **Code dei dispositivi** = insieme dei processi che attendono un evento I/O

Ogni processo nel corso della sua vita migra da una coda all’altra più volte, il suo ciclo di vita allora si può schematizzare:

1. Ready Queue
2. CPU (dopo essere scelto dallo scheduler), nel caso finisca il processo, questo termina
3. Varie opzioni:
   1. **Time slice expired**, la sua esecuzione supera i tempi che la CPU concede ad ogni processo -> torna a **Ready Queue**
   2. **I/O request** -> **I/O Queue**, ovvero il processo aspetta un evento di input output -> **I/O** -> ritorno alla **Ready Queue**
   3. **Fork a child**, ovvero la chiamata alla system call **fork** che serve per poter avviare un **programma figlio** a quello corrente -> esecuzione del processo figlio (che va in **Ready Queue**) -> il processo (padre) torna in **Ready Queue**
   4. **Wait for an interrput**, arrivo di un interrupt alla CPU che prende priorità superiore al processo in esecuzione, facendogli perdere la CPU -> occorrenza e risoluzione dell’interrupt -> ritorno in **Ready Queue**

**Schedulatore**

Uno schedulatore è un software che ha il processo di schedulare i processi (in stato *ready*) e quindi permettere la **multiprogrammazione**, e si dividono in:

* **Schedulatore a lungo termine** (Job Scheduler) = gestisce i processi che devono essere caricati in memoria principale (Ready Queue), in un sistema batch questo ha il compito di prendere uno dei tanti processi (più di quanti se ne possano servire) dal disco per portarlo in memoria principale (Unix e Linux non ce l’hanno) -> invocato raramente ed è generalmente più lento (secondi o min)
* **Schedulatore a breve termine** (CPU Scheduler) = seleziona uno dei processi dalla Ready Queue per portarlo all’esecuzione nella CPU -> invocato molto frequentemente (ms)
* **Schedulatore a medio termine**, diminuisce il numero di processi che richiedono I/O prendendone uno e depositandoli su disco (**swap out**), e una volta risolto il rallentamento dei processi lo fa tornare nella **Ready Queue** (**swap in**)

I processi, quindi, possono essere divisi in:

* **I/O-bound** -> processi che contengono molti ma brevi CPU burst (ovvero il tempo in cui il processo ha assegnato la CPU) e altrettante richieste I/O, consumando più tempo per questo che per la computazione
* **CPU-bound** -> processi che consumano molto più tempo per la computazione, che se potessero occuperebbero continuamente la CPU senza uscire, quindi, hanno pochi ma lunghi CPU burst, di cui ognuno di questi è causato dall’uscita del processo dalla CPU per uno dei tanti motivi

Un processo, come detto prima, ha la capacità di poter creare altri processi figli, che a loro volta ne sono capaci; creando un proprio albero di processi, dove, nel caso di un SO ha come radice fondamentale il processo **init** che da vita a processi figli che avviano il SO. Processi padre e figli condividono le risorse secondo tre situazioni diverse:

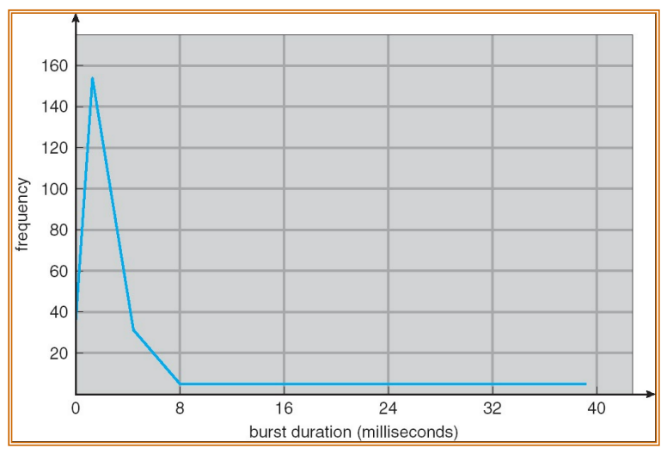
* Fra entrambi sono condivise tutte le risorse
* Il figlio condivide solo un sottoinsieme di risorse del padre
* Non condividono nulla fra loro

In Unix:

* **Fork** crea un nuovo processo
* **Exec**, usata dopo una fork per copiare (nel suo spazio di memorizzazione del processo) il codice di un programma anche diverso da quello del padre
* **Wait** permette al padre di aspettare la fine dell’esecuzione del figlio

**Scheduling della CPU**

La vita di un processo è un alternarsi di I/O burst e CPU burst, ovvero, rispettivamente: periodi di tempo usati puramente per input/output e periodi di tempo usati puramente per il calcolo

Generalmente i CPU burst sono molto frequenti e molto brevi, il loro rapporto è inversamente proporzionale, ad una frequenza di CPU burst molto elevata corrisponde una durata del burst molto breve, viceversa una frequenza molto bassa corrisponde ad una durata più lunga:

Il CPU scheduler con prelazione (**preemptive**) per la schedulazione dei processi si basa anche sulla priorità di questi, e ha il potere di togliere la CPU ad un processo per assegnarlo ad un altro che ha una priorità maggiore

Dispatcher è il modulo del SO che esegue:

* Cambio di contesto (passaggio da un PCB di un programma x all’altro del programma y)
* Passaggio alla modalità utente
* Permette alla CPU di tornare alla locazione corretta del programma

Latenza del dispatcher = tempo impiegato per il cambio di contesto

**Utilizzo della CPU** = mantenere la CPU il più impegnata possibile

**Throughput** = numero di processi completati per unità di tempo -> massimizzare

**Turnaround time** = tempo impiegato tra l’immissione del processo nel sistema e il suo completamento -> da minimizzare

**Waiting time** = tempo speso da un processo nella coda ready -> da minimizzare

**Tempo di risposta** = lasso di tempo tra la richiesta e la produzione della prima risposta per soddisfarla -> minimizzare

**Algoritmi di scheduling**

Ogni richiesta di un processo si presenta con la propria quantità di tempo di CPU burst

**FCFS (First Come, First Served)**

Le richieste dei processi vengono svolte nell’ordine in cui si presentano, es:

P0 ha 24 CPU Burst

P1 ha 3 CPU Burst

P2 ha 3 CPU Burst

Verranno eseguiti in ordine P0, e dopo 24 CPU Burst (o quando finirà) partirà P1 e a sua volta dopo la fine di P1 partirà P2,

P0 ha aspettato 0 per l’esecuzione

P1 ha aspettato 24 per l’esecuzione

P2 ha aspettato 27 per l’esecuzione

Quindi il **waiting time medio** è = (0+24+27)/3 = 17

Mentre il **turnaround time** è:

* P0 = 24
* P1 = 27
* P2 = 30

Nel caso fossero arrivati nell’ordine P1,P2,P0

Il waiting time:

* P1 = 0
* P2 = 3
* P0 = 6

Il waiting time medio allora è = (3+6+0)/3 = 3 -> molto più efficiente del primo caso, perché viene evitato l’effetto “**convoglio**” in cui processi dall’esecuzione più veloce (P1,P2) devono subire un tempo di attesa lungo per via dell’esecuzione di un processo più lento prima di loro (P0)

**SJF ( Shortest Job First)**

Tra tutti i processi in coda vengono scelti i processi dalla durata più breve, questo ha un waiting time medio sempre migliore del FCFS, gli algoritmi SJF si dividono in **nonpreemptive** (senza prelezione) e **preemptive** (con prelezione)

Nel caso non preemptive, lo SJF selezione tra un gruppo di processi, il processo dalla durata minore e gli da la priorità per l’esecuzione. Es:

Burst time = P1= 7, P2= 4, P3= 1, P4= 4

Arrival time = P1 = 0, P2= 2, P3= 4, P4= 5

Al tempo 0 viene quindi eseguito P1, alla fine della sua esecuzione sono già arrivate le altre 3 richieste, e viene quindi eseguito P3 che ha durata di esecuzione inferiore agli altri, e poi P2 e P4 che hanno la stessa durata.

Waiting time:

* P1 = 0
* P2 = 6
* P3 = 3
* P4 = 7

Waiting time medio = (6+3+7)/4 = 4

Turnaround time:

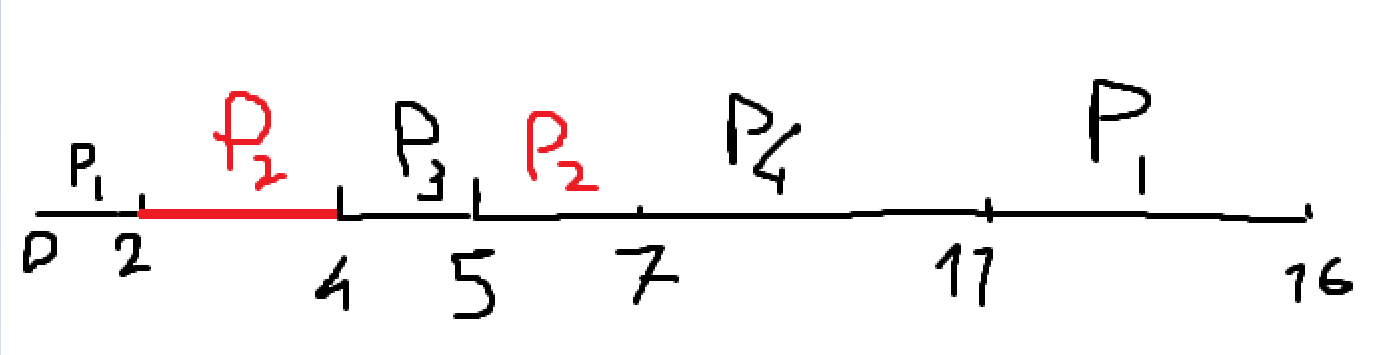
* P1=7
* P2=10
* P3=4
* P4=11

Turnaround medio: (11+10+4+7)/4 =8

Nel caso preemptive si parla anche di STRF (Shortest Time Remaining First), ovvero lo scheduler ha sempre il potere di togliere un processo dall’esecuzione se nel gruppo di processi ready esiste un processo dalla durata minore

Con gli stessi parametri di P1 P2 P3 P4

A tempo 0 viene eseguito P1



Waiting Time:

* P1 = 0+11-2= 9
* P2= 0+5-4=1
* P3= 0
* P4= 2+0=2

Waiting Time Medio = (9+1+0+2)/4 = 3

Turnaround Time:

* P1= 16
* P2= 5
* P3= 1
* P4= 6

Può essere stimato il prossimo CPU Burst in base alla lunghezza dei CPU Burst precedenti, pesandone l’importanza in basa alla loro età, ovvero quelli più recenti sono più importanti rispetto a quelli molto più vecchi (dimostrato sviluppando la formula considerandola come relazione di ricorrenza):

* Tn valore reale (tempo realmente impiegato) dell’ennesimo CPU Burst (può essere diverso da τn perché quella è soltanto una stima e non sempre rappresenta la realtà con precisione)
* τn+1 = valore previsto per il prossimo CPU Burst
* α = parametro compreso tra 0 ed 1 (deciso dal SO)
* τ0 = previsione del primo CPU Burst (valore default)

secondo la formula: τn+1= α \*Tn + (1-α)\* τn  --sviluppo-> τn+1= α \*Tn + (1-α)\*a\* Tn-1 + … + (1-a)i\*a\*Tn-1+ … + (1- α)n+1a\*τ0

il risultato di questa formula, quindi, indica la quantità (in tempo) di CPU Burst in via del tutto probabilistica e teorica, l’obiettivo di un SO è quello di avvicinarsi sempre di più alla previsione, α viene quindi deciso in base al SO che viene scelto in base alla previsione del compito che dovrà svolgere

**Scheduling a priorità**

Generalizzando ed implementando lo **SJF** lo si fa basare sulla priorità, questa **non** dipende solamente dal suo **CPU Burst** anche se è un fattore molto importante, viene quindi creato un nuovo **campo numerico** che indica la sua **priorità** (numero più **piccolo** = priorità **maggiore**, numero più **grande** = priorità **minore**). Per evitare il problema della ***starvation*** (**blocco indefinito**), ovvero il caso in cui un processo con **priorità** **minore** **viene** continuamente **bloccato** dall’arrivo di processi con priorità **maggiore**, **bloccandolo** per un **tempo non definito**, per **risolvere** si ricorre al concetto di ***aging*** (**invecchiamento**), quindi, al **processo** di prima **che avrebbe aspettato un tempo indeterminato**, viene **aumentata** **la sua** **priorità** gradualmente **dopo un lasso di tempo** di attesa portando prima o poi il processo ad essere **eseguito** sicuramente per via della sua **priorità sempre** più **crescente**.

La priorità ad ogni processo può essere assegnata in due modi:

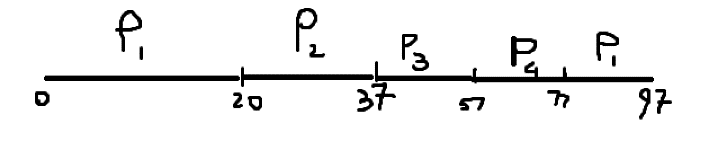
* **Interna** nel caso dipenda da fattori del processo stesso (CPU Burst, operazioni I/O et.)
* **Esterna** Se viene assegnato in base al dipartimento che ha promosso quel processo, ovvero arbitrariamente se un processo necessariamente essere eseguito gli viene assegnata una priorità maggiore anche se il SO avrebbe deciso in modo diverso

**Round Robin (RR)**

Il round robin è socialista

Il round robin è comunista

Il round robin non si basa su un sistema a priorità ma stabilisce un intervallo di tempo standard (di solito 10-100ms) concesso ad ogni processo, se questo viene completato, esce ed entra il prossimo, se invece superato questo lasso e il processo non è terminato, questo è forzato ad uscire portandolo nella coda dei ready, e quando gli verrà assegnata di nuovo la CPU avrà di nuovo lo stesso quanto di tempo (q) standard, se q è molto grande ci ritroviamo nella situazione di FCFS, mentre in q troppo piccolo porta a troppi cambi di contesto

Es. P1=53 P2=17 P3=68 P4=24

E così via

**Scheduling a code multiple**

La coda ready viene divisa in:

* Foreground (interattivi)
* Background (batch ­­­­­­­­­­­– sullo sfondo)

Ognuna di queste due ha il proprio algoritmo:

* Foreground -> Round Robin
* Background -> FCFS

Bisogna poi implementare un algoritmo di scelta tra le code per decidere quale dei processi proposti da ognuna portare in CPU, questi possono basarsi su:

* Priorità fissa, tipo prima quelli in foreground e poi background. Causa starvation
* Time Slice, ogni processo in base alla coda da cui proviene ha un intervallo di tempo in cui gli viene assegnata la CPU, es. 80% della slice per foreground e 20% per i background

**Code multiple con Feedback**

Si basa sulla divisione tra lo scheduling a code multiple, solo che implementa anche il passaggio da una coda all’altra, “elevando” o “abbassando” la priorità di un processo portandolo ad una coda con più o meno “benefici”

Es.:

Q0 gestita da RR con q=8ms

Q1 gestita da RR con q=16ms

Q2 gestita FCFS

Schedulazione tra queste:

* Un nuovo processo entra in Q0­­­­­­­­­­­­­, quando gli viene assegnata la CPU, ha 8ms per i suoi CPU Burst. Se non terminato viene spostato in Q1 in cui poi quando verrà schedulato avrà 16ms, e se ancora non avrà finito verrà infine spostato in Q2 dove avrà la possibilità di terminare

**Time Sharing**

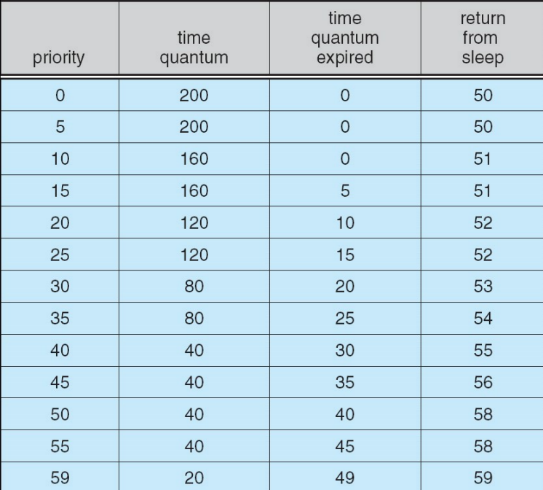
Un sistema Time Sharing è paragonabile ad un Round Robin, nel senso che ad ogni processo viene assegnato un quantum di tempo per la CPU ma con l’aggiunta di un sistema di priorità, in questo caso esistono sistemi **con prelazione** (un processo con priorità maggiore che entra in Ready forza l’uscita di un processo con priorità minore) e sistemi **senza prelazione**.

**Scheduling di Solaris**

Usa un algoritmo preemptive con priorità a code multiple:

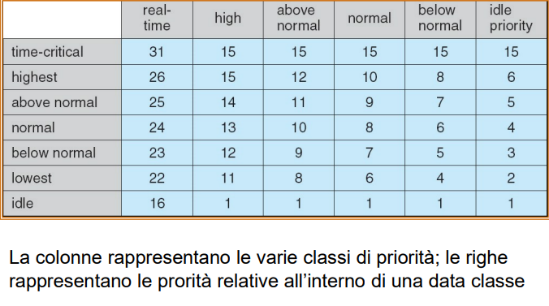
* Real-time (priorità alta)
* Processi di sistema (priorità alta ma immediatamente sotto quella real-time)
* Time-sharing (suddivisa in altre code a feedback, una volta eseguiti real-time e processi di sistema vengono eseguiti questi)
* Interattivi (priorità maggiore rispetto ai processi che hanno un CPU Burst alto)

**Solaris Dispatch Table**

vengono istituiti 60 livelli di priorità in cui:

* Il quanto di tempo è inversamente proporizonale alla priorità
* Quanto di tempo esaurito: rappresenta la priorità di un processo dopo che ha superato il lasso di tempo assegnato
* Ripresa dell’attività: priorità di un processo che ha appena finito di attendere (tipo attesa I/O)

**Scheduling Windows XP**

Usa un algoritmo preemptive a priorità a 32 livelli, divise in:

* Variabile (1 a 15)
* Real-Time (16 a 31)
* 0 è riservato ai processi che gestiscono la memoria

Esiste una coda per ogni priorità

**Scheduling di Linux**

Uno scheduling preemptive a priorità, dividendo il range delle priorità in due classi:

* Real-Time da 0 a 99
* Nice da 100 a 140

I quanti di tempo sono organizzati per essere direttamente proporzionali alla priorità, priorità più alta = quanti più lunghi

Lo scheduling si basa su uno stato di attivo o scaduto, un processo è nello stato attivo se è pronto all’esecuzione e non ha ancora superato il quanto di tempo per l’esecuzione della CPU, superato q, il processo viene considerato scaduto e dovrà aspettare che vengano eseguiti tutti i processi active prima

**Multithreads**

Il processo è costituito da:

* Possesso delle risorse
* Esecuzione

La prima è il processo stesso, mentre, l’esecuzione è la parte dell’esecuzione del processo.

Considerati più esecuzioni del processo si parla di multithreading, ovvero esecuzione multipla con le stesse risorse di base, per esempio in un browser web:

* partendo dalle stesse risorse ci saranno più thread, magari uno che rappresenta a schermo testo e immagini e un altro ricerca i dati sul web

Il processo rappresenta un master e tutti i thread rappresentano gli slave che lavorano sui dati, infatti ogni thread appartiene ad un processo unico (con lo stesso codice) ma lavorano su parti diverse di questo

Ogni thread poi vengono allocati e utilizzati per ognuno:

* dei registri
* uno stack locale
* una parte del codice o dei dati su cui lavorare, derivati dal codice del master (risorse del processo)
* un proprio descrittore
* un program counter
* uno stato di esecuzione

Per via delle poche informazioni (come carico) associate ad ogni thread, il cambio di contesto avviene più velocemente ed efficientemente

Il multithreading può avvenire anche in parallelo, in cui ogni thread lavora su una parte diversa del programma, mandando poi il risultato delle loro operazioni ad una zona di memoria comune per il funzionamento del programma

Nell’organizzazione multithreading, lo scheduling si adatta molto bene ai sistemi multicore, infatti un unico processo può essere assegnato per via di thread diversi su cpu core diversi, velocizzando di molto l’esecuzione di un singolo processo

Il sistema dei thread a livello di sistema operativo viene supportato in base al livello di utilizzo:

* user level, attraverso delle API (librerie di funzioni) utilizzate per gestire n thread
* kernel level, tramite una tabella dei thread di sistema gestita attraverso delle system call

**Thread a livello utente**

La gestione è affidata completamente al codice dell’utente, ovvero il sistema operativo vede il processo come un unico processo anche se questo viene lavorato in più thread attraverso l’uso di API

* Vantaggi:
  + Si può utilizzare un pacchetto di thread anche in un SO che non li supporta
* Svantaggi:
  + All’avvenire di una system call che blocca il processo di uno dei thread, questo blocca anche tutti gli altri thread del processo

**Thread a livello kernel**

Sono gestiti direttamente dal sistema operativo, vengono quindi organizzati e gestiti i singoli thread piuttosto che tutto il programma, risolve il problema dei thread a livello utente; infatti, se un thread esegue una system call “bloccante” verrà bloccato solo quel thread mentre gli altri continuano la loro esecuzione

I thread a livello utente e quelli a livello kernel, esistono diversi modelli di relazione:

* Molti ad uno (M-1):
  + Come descritto prima, più thread a livello utente vengono eseguiti a livello kernel come un unico thread
  + Vantaggi:
    - Gestione efficiente dei thread a livello utente
    - Non richiede un kernel multithread per l’applicazione
  + Svantaggi:
    - L’intero processo si blocca se uno dei thread esegue una system call bloccante
    - I thread non possono essere gestiti su più CPU
* Uno ad uno (1-1):
  + Ciascun thread a livello utente corrisponde ad un thread a livello kernel
  + Vantaggi:
    - Scheduling molto efficiente
    - Ogni system call è legata ad ogni singolo thread, se uno di questi esegue una system call bloccante gli altri possono continuare a lavorare
    - I thread possono essere lavorati su CPU fisicamente distinte
  + Svantaggi:
    - Svantaggio in base al carico di lavoro, nel peggiore dei casi consiste nella creazione e gestione di molti thread
    - Richiede necessariamente un kernel multithread
* Molti a Molti (M-M):
  + Si fanno corrispondere più thread a livello utente ad un numero minore od uguale di thread a livello kernel, esistono quindi dei thread working che vengono assegnati di volta in volta a thread utente
  + Vantaggi:
    - Possono essere creati tanti thread a livello utente quanti ne servono, e questi vengono gestiti efficientemente a livello kernel anche in parallelo su architetture a più processori fisicamente distinti
    - Se un thread invoca una system call bloccante non ferma l’esecuzione di tutti gli altri thread
  + Svantaggi:
    - È resa difficile la gestione della pool di thread worker e la gestione di corrispondenza tra thread utente e thread kernel

**Sincronizzazione dei processi**

Nelle code dei processi possono esistere **processi cooperanti**, definiti così per la condivisione di risorse, dati/file/memoria/thread e la competizione per la CPU crea un problema specialmente tra questi due, perché magari può portare a risultati inaspettati. Il SO deve quindi promettere la coerenza dei dati.

Es.

processi: **Produttore** e **Consumatore**

*Questi due utilizzano gli stessi dati, nello specifico il produttore è un sito web e fornisce una pagina o dei dati in generale e il consumatore è un client che si connette, quindi, questi due utilizzano lo stesso buffer da cui possono leggere e scrivere.*

*In cui produttore “produce” un elemento per inserirlo nel buffer (se possibile = se c’è lo spazio per inserire l’elemento nel buffer), grazie al puntatore* ***in*** *che punta alla prima posizione libera, incrementando il contatore di elementi*

*Il consumatore invece toglie elementi dal buffer (se ci sono) grazie al puntatore* ***out*** *che punta alla prima posizione occupata del buffer, decrementando il contatore di elementi del buffer*

*Il problema tra questi due processi è l’incremento o il decremento del contatore (degli elementi) che non è un’operazione atomica ma ne sono tre: carico in un registro il valore del contatore, incremento/decremento il registro e infine sovrascrivo il registro*

*Questo può portare a risultati inaspettati per via della perdita della CPU nel corso di una di queste operazioni*

Il sistema operativo deve garantire quindi la **mutua esclusione**, ovvero che due processi non devono poter trovarsi in una situazione simile

Il problema dell’interfogliamento dipende dalla **race condition**, ovvero più processi concorrono per accedere e modificare dati comuni. La soluzione è la sincronizzazione l’accesso a questi dati comuni

La parte del processo in cui si accede a dati comuni è detta **sezione critica**, l’obiettivo è quindi rendere impossibile ad altri processi l’accesso a quel dato quando un altro processo ha raggiunto la parte critica

Il problema di sincronizzazione avviene anche a livello kernel, per esempio la tabella dei processi aperti deve essere modificata ogni volta da ogni processo che viene aperto o chiuso

**Sezione critica**

Per risolvere questa problematica bisogna progettare un protocollo di cooperazione, denotando l’accesso e l’uscita di un processo ad una sua sezione critica attraverso **un’entry section** e **una exit section**, ovvero avere un controllo sull’entry section ad ogni sezione critica, e l’exit section alla fine della sezione critica per tornare alla normale esecuzione del processo

Entry Section e Exit Section devono garantire:

* **Mutua Esclusione**: ovvero se un processo P esegue la sua sezione critica nessun altro processo, anche se assegnata la CPU, può eseguire la propria sezione critica
* **Progresso**: Una volta che un processo P è uscito dalla sua sezione critica (si trova nella sezione non critica) non deve causare un deadlock agli altri processi che vogliono accedere alla propria sezione critica
* **Attesa Limitata**: evita la starvation degli altri processi, assegnando un quanto di tempo massimo per la sezione critica di un processo

Possibili soluzioni:

* **Alternanza Stretta**: utilizzare una variabile comune (turn) che indica quale processo deve eseguire la sua sezione critica, la variabile viene poi cambiata in quella dell’altro processo alla fine della sezione critica.
  + **Vantaggi**: Garantisce ***progresso****(apparentemente)* e ***attesa limitata***
  + **Svantaggi**: Funziona solo nel caso i due processi P0 e P1 devono funzionare in sequenza prima uno e poi l’altro, per via dell’exit section (turn=1 o turn=0)
* **Soluzione di petersen**: si basa sull’alternanza stretta, avendo in comune una variabile turn che indica quale processo “è di turno”, con l’aggiunta di un array flag che in corrispondenza del numero del processo (es cella 0 in corrispondenza di P0) che indica se un processo è pronto per eseguire la sua sezione critica quando è true
  + **Svantaggio**: funziona solo su due processi
* **Soluzione generale**: bisogna utilizzare uno strumento **lock** per eseguire la propria sezione critica, per poi restituirlo alla fine di questa
* **Hardware di sincronizzazione**: vengono rese atomiche delle istruzioni attraverso la modifica dell’architettura del processore, aggiungendo per esempio l’istruzione atomica ***Swap*** e ***TestAndSet***
* **Semafori**: il semaforo è una variabile intera che viene inizializzata ad 1 (disponibile) e può avere uno stato <=0 (non disponibile) utilizzando due operazioni atomiche **wait()** e **signal()**:
  + **wait** corrisponde ad un ciclo while infinito che non fa nulla finché S<=0, ovvero finché S non è disponibile. Uscendo dal ciclo infinito S viene decrementato
  + **signal** incrementa S

il semaforo ha l’utilità di fermare l’esecuzione di un processo quando un altro ha raggiunto la sua sezione critica, viene quindi inizializzato ad 1, ed incrementato o decrementato in base alla necessità del processo;

Quando il semaforo S>1 significa che

* + **Svantaggi**:
    - **Spinlock**: ovvero l’attesa la perdita di tempo per la CPU nel while infinito del wait per i processi che aspettano un altro processo finisca la sezione critica **(busy waiting)**
* Il busy waiting può essere eliminato aggiungendo un puntatore alla lista di processi in attesa, il semaforo viene quindi definito come un record, e viene modificata la wait e la signal:
  + Wait: viene diminuito il valore di S, e se il suo valore è <0 il processo che invoca la wait viene aggiunto alla lista dei processi
  + Signal: viene incrementato il valore di S, se il suo valore poi è <=0 viene tolto il processo che invoca signal dalla lista dei processi
  + **Svantaggi**: Starvation, Deadlock(due processi si bloccano l’un l’altro)

**Gestione memoria centrale**

La memoria centrale è gestita come un array, in cui sono memorizzati il sistema operativo, dove vengono caricate le informazioni di bootstrap e soprattutto le locazioni di memoria per l’esecuzione dei processi, questo è gestito in registri:

* Il registro e il suo puntatore (e indirizzamento) viene associato ad un processo dalla CPU, che emette un puntatore del processo, a cui a livello di memoria centrale vengono associati 2 registri **base** e **limit** chesono utilizzati per controllare la posizione e il corretto indirizzamento (spaziale e di memorizzazione), infatti viene controllato che l’indirizzo prodotto dalla CPU sia compreso tra l’indirizzo base e l’indirizzo limit, per controllare che quel processo scriva esclusivamente in quella locazione di memoria (**base** = **indirizzo limite inferiore** e **limit** = **indirizzo limite superiore**) , proteggendo di conseguenza tutta la memoria da scritture involontarie e impreviste in aree non volute

L’associazione di un’istruzione di un processo ad uno degli indirizzi di memoria può avvenire in fasi diverse:

* A tempo di compilazione:
  + Si conosce già dove il processo risiederà (ovvero quando viene compilato un programma viene già portato in memoria principale prima ancora dell’esecuzione, usato sui vecchi DOS [ai tempi di pappagone] )
    - **Svantaggi**: a qualsiasi variazione(l’indirizzo può variare in base a varie cose, una di queste può essere per via dello scheduling della memoria) dell’indirizzo assegnato, il programma deve essere ricompilato
* In fase di caricamento:
  + Evoluzione del collegamento a tempo di compilazione, si basa sulla generazione di un codice **rilocabile** a cui il caricatore (loader) associa **indirizzi rilocabili** nella memoria centrale
    - A differenza di quello prima, se varia l’indirizzo iniziale, il loader deve soltanto aggiustare gli indirizzi, invece di dover ricompilare per ottenere altri indirizzi
    - **Svantaggi**: caricamento ripetuto troppe volte = spreco di tempo e di memoria
* A tempo di esecuzione:
  + Evoluzione ancora di quello di prima, vengono portate in memoria segmenti del processo a mano a mano che vengono eseguiti, questo avviene in base ad un supporto hardware, e l’indirizzo in memoria viene calcolato sommando al all’indirizzo generato dalla CPU + indirizzo base

Spazio di indirizzamento logico: indirizzo generato dalla CPU (detto anche indirizzo virtuale)

Spazio di indirizzamento fisico: indirizzo a livello della memoria

Il supporto hardware necessario per il collegamento a tempo di esecuzione è detto MMU (Memory Management Unit), e il suo compito è quello di sommare l’indirizzo logico all’indirizzo di rilocamento (indirizzo base)

L’indirizzo fisico può variare in base allo swapping (cambio di contesto) per via dello scheduling della CPU, non è detto quindi che un processo dopo un cambio di contesto, se rientrato in memoria principale, abbia lo stesso indirizzo di base di prima

**Caricamento dinamico**

È una tecnica di caricamento che consiste nel caricare in memoria una procedura solo quando questa viene chiamata, per poter ridurre lo spazio occupato nella memoria principale, invece di caricare tutte le procedure di tutto il programma. Basandosi su un’allocazione temporanea ogni volta riempita dalla/e procedure chiamate, per poi essere liberate quando non più necessarie.

**Allocazione contigua di memoria**

Con partizioni multiple:

* Nella memoria centrale sono presenti molti hole (blocchi liberi di dimensione variabile), all’arrivo di un processo in memoria centrale il sistema cerca un buco abbastanza grande da poterlo ospitare, se non lo trova lo inserisce dopo
  + **Svantaggio**: possono crearsi dei buchi troppo piccoli per ospitare processi, riducendo quindi la capacità di memorizzazione
* Algoritmi per la memorizzazione:
  + **First-fit**: assegna il processo nel primo hole abbastanza grande per poter contenere il processo (più efficiente)
  + **Best-fit**: assegna il processo all’hole che è abbastanza grande da ospitare il processo ma che lascia meno blocchi liberi possibili (il sistema legge prima tutti i blocchi, risulta meno efficiente però riduce il rischio di frammentazioni)
  + **Worst-fit**: assegna il processo all’hole più grande possibile (poco efficiente, e anche questo riduce *teoricamente* il rischio di frammentazione)

Come per l’allocazione contigua nel disco, **l’allocazione contigua** per la memoria **causa** **frammentazione**:

* **Esterna**: lo spazio libero per poter ospitare un processo c’è ma non è contiguo, quindi, non è utilizzabile
* **Interna**: lo spazio assegnato ad un processo può essere sempre almeno un po’ più grande dello spazio richiesto dal processo

La frammentazione può essere risolta manualmente con una compattazione dei processi in alto nella memoria eliminando tutti i piccoli hole uniformandoli in uno solo alla fine. Questa procedura però può generare altri problemi per i processi che sono in fase di caricamento, perciò è sconsigliata in favore alla paginazione.

**Paginazione**

La memoria fisica viene divisa in **frame** (blocchi), la cui dimensione è una potenza di 2

La memoria logica in blocchi delle stesse dimensioni (dei frame), chiamati **pagine**

Si mantiene traccia di tutti i frame liberi, e quindi per eseguire un processo di **n** pagine, sono necessari **n** frame liberi, vengono caricate quindi le pagine in questi frame

Per evitare la memorizzazione contigua, viene creata una **tabella delle pagine** per tradurre gli indirizzi logici in indirizzi fisici, che memorizza quindi i frame nei quali sono memorizzate le pagine

**Traduzione da indirizzo logico ad indirizzo fisico**:

Ogni indirizzo generato dalla CPU viene suddiviso in:

* Numero di pagine (spazio indirizzo logico/size pagina-> m-n, dove 2m è lo spazio di indirizzi logici e 2n è la size di una pagina -> 2m/2n = m-n)
* Spiazzamento nella pagina: corrisponde alla size di una pagina, usato per calcolare l’indirizzo fisico. Rappresenta la posizione nella pagina.

Il calcolo per la traduzione avviene a livello hardware attraverso un dispositivo che riceve in input l’indirizzo logico. Ogni volta che viene emesso un indirizzo logico dalla CPU, avvengono 2 accessi in memoria principale.

Es:

Size memoria centrale =32 Byte size frame/pagine= 4 Byte =22 Byte

indirizzo logico = 13 = 1101

l’indirizzo si divide in 11 e 01 (perché l’esponente della size della pagina è 2 e corrisponde, quindi 2 bit)

da 11 nella tabella delle pagine vediamo che è assegnato il frame 2, quindi 010

allora l’indirizzo fisico è 01001 perché vengono aggiunti i bit di offset (gli ultimi 2 in questo caso)

I frame liberi sono gestiti attraverso una lista che punta ad ogni frame non ancora allocato, che viene consultata ogni volta che bisogna scrivere in un frame

**Tabella delle pagine e TLB**

Innanzitutto, risiede in memoria principale. Però esiste l’implementazione **TLB** che funziona come una cache che memorizza ogni volta le ultime associazioni pagine – frame per rendere potenzialmente il calcolo degli indirizzi meno dispendiosi perché essendo vicina alla CPU è la prima memoria da guardare per accedere ai frame, e potenzialmente se si vuole accedere ad uno degli indirizzi scritti nella TLB non bisogna ricalcolarli per accedere, la sua funzionalità si basa sul principiò della località, ovvero che una pagina già visitata è più probabile sia rivisitata piuttosto che una nuova. **In breve**:

* Quando la CPU emette un indirizzo logico, passa prima ad un controllo per tutta la TLB per vedere se è già presente il risultato della conversione in indirizzo fisico, se non è presente la CPU procede a calcolarlo dalla tabella delle pagine

**Tempo di accesso effettivo**

Tempo di accesso alla TLB = ε

Hit ratio = α (rappresenta la frequenza delle volte in cui un numero di pagina viene trovato nella TLB)

L’effective access time (EAT) si calcola:

EAT = (1 + ε)α + (2 + ε)(1 - α) =

**EAT =** **2 + ε - α**

**Protezione della memoria centrale (nella page table)**

Si basa sulla presenza di un bit di protezione associato ad ogni frame che può assumere 2 valori:

* Valido = ovvero che la pagina a cui è associato questo valore è nello spazio degli indirizzi logici del processo, quindi, ci si può accedere
* Non valido = la pagina non è presenta nello spazio associato degli indirizzi logici del processo

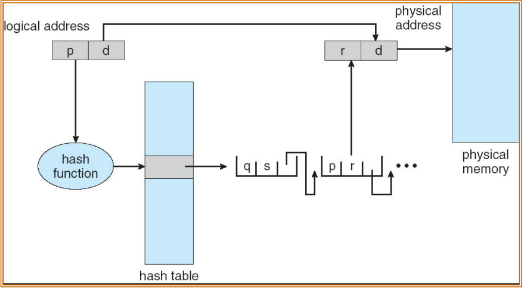
**Pagine condivise**

Se processi diversi hanno bisogno e usano pagine comuni, nelle loro personali page table è presenta l’indirizzo comune della memoria centrale che corrisponde a quella risorsa comune.

Il vantaggio della paginazione è quello di poter gestire efficientemente la condivisione di risorse e soprattutto superare il problema della frammentazione

**Struttura della page table**

La tabella delle pagine può creare un problema nella memorizzazione se di dimensioni troppo grandi nel caso questa venga allocata in modo contiguo, le possibili soluzioni a quest’allocazione sono:

* **Paginazione gerarchica**:
  + Il numero di pagina viene diviso in due parti, in cui la prima parte rappresenta l’indirizzo logico nella prima tabella (la più esterna), la seconda parte lo spiazzamento nella prima tabella, il risultante da questa rappresenta l’indirizzo logico nella seconda tabella (la più interna) e l’ultima parte dell’indirizzo prodotto dalla CPU rappresenta l’offset con cui viene calcolato poi l’indirizzo fisico.
* Attraverso **l’hashing**:
  + L’indirizzo virtuale prodotto dalla CPU si usa come input per una funzione hash che da come prodotto un indirizzo che viene usato come indice nella tabella, dove ogni entry della tabella ha associato una **lista** di **coppie di elementi**:
    - numero di pagina
    - frame
  + Dalla lista poi viene preso il numero di pagina, per poi estrarre il suo frame corrispondente per poter calcolare l’indirizzo fisico:
* **Tabella delle pagine invertita**:
  + Simile al funzionamento della FAT, ovvero viene memorizzato in una tabella tutti i frame della memoria centrale, e in corrispondenza di ogni frame è memorizzato il numero della pagina logica, in aggiunta ad informazioni riguardanti il processo a cui appartiene quella pagina, ovvero in 0

**Memoria Virtuale**

Lo spazio di indirizzamento virtuale è fisicamente lo spazio allocato situato tra l’heap e lo stack di un processo e la sua dimensione varia in base alla crescita di questi due.

La memoria virtuale, quindi, può memorizzare e rendere possibile la memorizzazione di uno spazio logico più grande di quello fisico, caricando in memoria centrale man mano parti di programma riducendo lo spazio occupato in memoria principale da processi con risorse molto dispendiose

La memoria virtuale è quindi uno spazio comune che porta in memoria centrale i frammenti di memoria (pagine) necessarie all’esecuzione di un processo, basandosi sul principio di località per sostituire le pagine già usate che con molta probabilità non verranno riutilizzate (nel caso poi ci sia un miss, la pagina richiesta viene portata in memoria). Rendendo quindi eseguibili programmi con un peso maggiore di quello disponibile fisicamente in memoria centrale.

La memoria virtuale può essere implementata attraverso:

* Paginazione su richiesta
* Segmentazione su richiesta

**Paginazione su richiesta**

Introduce una pagina in memoria solo se necessario, perché un processo al lancio vengono portate poche (o anche nessuna) pagine in memoria. E nel caso abbia bisogno di altre pagine, vengono cercate seguendo il processo:

Pagina necessaria -> riferimento alla pagina necessaria -> se la pagina non è in memoria -> portata in memoria

Nel caso il riferimento alla pagina sia non valido il programma termina in errore

Nel caso la pagina sia già presente in memoria l’esecuzione continua normalmente:

Questo viene controllato per via del bit di validità, che può assumere due valori:

* i = non in memoria
* v = in memoria

All’ avvio del processo tutti i bit vengono impostati ad i e poi aggiornati di conseguenza

La mancanza di pagina in memoria è controllata quindi attraverso il bit di validità, che nel caso incontrato, durante la traduzione dell’indirizzo logico, è nello stato **i** allora causa una **trap** (avviso al sistema operativo = primo step) al sistema chiamata **page fault**, viene salvato lo stato del processo e i suoi registri (secondo step), viene distinta la page fault tra i due possibili casi:

* Riferimento alla pagina non valido -> termine del processo
* Pagina riferita non in memoria:

1. Viene cercato un frame libero
2. Viene spostata la pagina riferita nel frame trovato, marcandolo come occupato
3. Viene cambiato lo stato del bit di validità a **v**
4. Riavvio istruzione che ha richiesto la pagina (causando page fault)

Il riavvio dell’istruzione può sollevare a sua volta altre page fault, in quanto viene rieseguita ogni volta tutta l’istruzione, esempio:

add(a,b,c) dove c è la destinazione della somma. Nel caso sia sollevato page fault per c, viene caricata in memoria (seguendo il processo di prima) e poi rieseguita l’operazione (con tanto di caricamento in memoria di a, di b e l’operazione) per poi finire con la scrittura nel frame assegnato alla memorizzazione di c

**Prestazione della paginazione su richiesta:**

Sia p la probabilità di page fault (compresa tra 0 e 1)

EAT = (1 – p) x accesso alla memoria + p (page fault overhead + lettura della pagina + overhead ripresa)

**Sostituzione di pagina**

Se alla necessità di scrivere una pagina in un frame, viene attuata una sovrallocazione (o sovrascrizione) di un frame, per eseguire questo processo si necessità però di un algoritmo per la selezione del frame “vittima” che risulta non usato in quel momento o non strettamente necessario in un tempo x, spostando la pagina vittima sul disco, per poi poterla riprendere nel caso diventi utile o venga richiesta

Riassumendo quindi:

* Se un **processo** in esecuzione ha bisogno di **memorizzare** una nuova pagina in memoria, ma questa **risulta piena** (nessun frame libero), la soluzione è la **sostituzione di pagina**, ovvero sostituire una delle pagine scritta in uno dei frame, scrivendo la pagina sovrascritta nel disco:
  + **Questa sostituzione** avviene attraverso **l’individuazione di un frame** con una **pagina non utile** in quel momento (o non utilizzata), con poi conseguente **scrittura sul disco**, questa **avviene** solo nel caso la pagina **sia stata precedentemente modificata**, infatti, risulta inutile (e dispendioso) accedere al disco scrivendo di nuovo gli stessi dati sul disco. Questa condizione è identificata attraverso il **modify bit**

Il fine dell’algoritmo è quello di ottimizzare e ridurre al minimo il numero di page fault:

* **FIFO**
  + Ogni pagina come si presenta viene inserita nei frame liberi, e se non sono liberi vengono gradualmente sostituiti quelli scritti
    - Poco efficiente -> anomalia di Belady
* **LRU (Last Recently Used)**
  + L’ottimizzazione della sostituzione pretende di conoscere già le prossime pagine che verranno richieste, la sostituzione delle pagine avviene quindi su quelle non usate per un lungo periodo di tempo. Quindi viene sostituita la pagina che non viene richiesta in tempo recente

**LRU**

L’implementazione avviene:

* **attraverso un** **contatore**:
  + contatore assegnato alla CPU che determina quale processo sia stato usato meno recentemente, viene incrementato ad ogni riferimento ad una pagina della memoria
* **Attraverso uno stack**:
  + Vengono pushate nello stack tutti i numeri di pagina riferite, e messe in cima tutti i numeri già presenti in mezzo quando vengono richiamati, per determinare poi la pagina usata meno recentemente si guarda il bottom dello stack, evitando la ricerca per un frame libero

L’algoritmo di LRU può essere approsimato nel caso il SO non supporti l’esistenza di uno stack o di un contatore per le pagine. Viene quindi utilizzata un’architettura basata su un **bit di riferimento** a livello della tabella delle pagine in corrispondenza dell’entry. Inizialmente il bit viene assegnato a 0 e quando viene referenziato, il bit cambia di stato ad 1, sostituendo quelli impostati a 0.

**FIFO migliorata**

La FIFO può essere migliorata di conseguenza per via del bit di riferimento, definendola come una lista circolare, come per la FIFO si vuole sostituire la pagina che è presente da più tempo.

Questo avviene scorrendo la lista delle pagine in ordine FIFO, e la si sostituisce alla pagina usata meno recentemente nel frame con bit di riferimento 0, mentre se una pagina ha bit di riferimento ad 1, questo viene decrementato a 0 (**algoritmo di seconda chance**).

L’algoritmo di seconda chance può essere a sua volta raffinato:

* Vengono memorizzati 2 bit, (bit di riferimento, bit di modifica) che indicano rispettivamente:
  + Bit di riferimento:
    - 1 = usato di recente
    - 0 = non usato di recente
  + Bit di modifica:
    - 1 = modificato
    - 0 = non modificato