*Appunti di Gabriele P. (aremi)*

*Più o meno dovrebbe esserci tutto*

# Introduzione ai Sistemi Operativi

## Prime Definizioni e concetti:

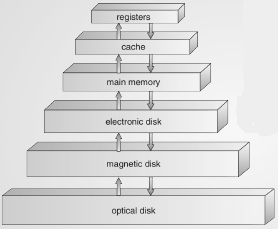
* + **Astrazione**: rendere semplice qualcosa di complicato.  
    Es: Il concetto di file, dal punto di vista Hardware, non esiste, è in realtà una astrazione software.
  + **Multiplexing nel tempo**: una risorsa viene richiesta da più processi ma in un dato istante può essere usata da una entità soltanto.  
    Grazie al multiplexing i processi hanno l'impressione di avere una CPU dedicata, mentre invece i vari processi vengono alternati in intervalli di tempo nel registro.
  + **Processore**:
    - *Ciclo di base*: prelevamento (*fetch*), decodifica, esecuzione.
    - *Registri particolari*: **PC**, **PS**, **PSW**.
      * *PC*: Indirizzo prossima istruzione.
      * *PS*: Stack pointer, punta alla cima dello stack.
      * *PSW*: Program status word, composto da una collezione di flag con cui indica anche la modalità di esecuzione della CPU.
        + Modalità di esecuzione doppia: *user process* e *kernel*.
  + **Chiamate di Sistema (*trap*)**: modifica un bit del registro PSW da *utente* a *kernel* e viceversa.

Anche nelle **Interrupt Hardware** cambia la modalità di esecuzione a *kernel*.

Un processore può eseguire due processi contemporaneamente, ma non è una esecuzione parallela vera e propria (multithreading).

Oppure posso avere fisicamente più processori per aumentare le prestazioni e l'affidabilità.

## Memorie

Gerarchia, dalla memoria più veloce alla più lenta:

## Dispositivi I/O

Il **Driver** permette al *Sistema Operativo* di comunicare con il **Controller**.

I dischi SATA sono invece universali e economici.

Tutti i *driver* dello stesso *sistema operativo* sono standard e le istruzione di input/output del driver possono essere usate solamente in **modalità kernel**.

Disabilitare gli Interrupt Hardware è conveniente per piccole porzioni di tempo per processo. Non è bene invece disabilitarli perché ci sarebbero tecnologici che non funzionerebbero. Tipicamente ogni Controller ha la propria Interrupt.

## Corel Cache Core2 Cache DDR3 Memo PCIe slot PCIe slot PCIe slot PCIe slot Shared cache GPu Cares Memo controllers OMI Platform Controller Hub More PCIe devices pCle Gra hics DDF3 Memo SATA USB 2.0 orts USB 3.0 orts Gi abit Ethernet pole BUS

* Bus PCI: 32bit contemporaneamente.
* USB (Universal Serial Bus): non ha sempre bisogno del Controller, ha bisogno del driver ma non è universalmente disponibile per tutti i sistemi operativi.

## Strutture Sistemi Operativi

Possibili strutture:

* + Monolitica.
  + A livelli.
  + Microkernel.
  + A moduli.
  + Macchine Virtuali.

### Monolitica:

Unico kernel con tutto dentro, nessun supporto hardware. Qualunque cosa può chiamare qualunque cosa. Inizialmente non c'era divisione tra modalità utente e kernel, solo in seguito venne sviluppata questa divisione.

### A livelli:

Lo strato inferiore è la struttura hardware, priva di bug. Superiormente troviamo una serie di livelli di astrazione con dati e metodi.

Più vado verso l'alto e più ho servizi e più problemi vengono risolti dai problemi sottostanti. I diversi strati possono essere implementati anche da team diversi.

Un livello può interagire direttamente solo con il livello immediatamente inferiore. Un bug in uno strato può andare ad intaccare anche gli strati inferiori.

Una variante di questa struttura e la Struttura ad Anelli Egocentrici:

Al centro si trova l'hardware e man mano che vado verso l'esterno gli strati diventano più potenti.

### Microkernel:

Alcuni servizi vengono tirati fuori dal kernel, che sarà di dimensioni ridotte, e considerati come entità esterne.

All'interno del microkernel rimangono la gestione dei processi. Il microkernel diventa anche più semplice da scrivere e da debuggare. Qualunque servizio al di fuori del microkernel non fa danno, al massimo crasha.

Il microkernel comunica tramite chiamate di sistema, la comunicazione avviene tramite un modello a messaggi.

Uno svantaggio è che le entità esterne mandano messaggi esterni che devono comunque passare per il microkernel.

### Struttura a moduli:

Ho un kernel con un numero minimo di servizi, tutti gli altri servizi sono gestiti da moduli. Tali moduli implementano servizi specifici (filesystem, driver, ecc…) e sono caricabili dinamicamente a *run-time*. Ogni modulo può invocare qualunque altro modulo direttamente.

### Macchine Virtuali:

Può essere considerato una sorta di estensione ulteriore al concetto di astrazione già visto.

Questo concetto di astrazione può essere esteso fino ad arrivare ad una sorta di copia completa di un calcolatore.

Guest OS process 
Guest OS 
Type 2 hypervisor 
Kernel 
module 
oo 
Host operating system Excel Word Mplayer Apollon 
Windows 
Linux 
Type 1 hypervisor Abbiamo un sistema ospitante e un sistema ospitato, il sistema ospitante è tipicamente corredato da un ***Hypervisor*** che gestisce il meccanismo di virtualizzazione. Al di sopra dell' *Hypervisor* vive il sistema ospitato (anche più istanze).

#### *La para-virtualizzazione…*

E' un modello più specifico. Nella virtualizzazione generale, la macchina virtuale è una sorta di copia interale del sistema e il sistema ospitato non ha bisogno di essere progettato per girare in un ambiente virtuale.

Nell'ambiente para-virtualizzato invece il sistema operativo ospitato è conscio di girare in un sistema ospitante e il suo kernel è adattato a questo scopo, utilizzando alcuni servizi messi a disposizione dal sistema di virtualizzazione (Hypervisor) per collaborare con il sistema ospitante e gestire le risorse in maniera più efficiente.

Tipicamente i sistemi para-virtualizzati sono della stessa natura (windows con windows ecc…).

#### *Nella simulazione…*

si fa qualcosa di simile alla virtualizzazione. Nella virtualizzazione l'unica ipotetica CPU fisica del sistema, in un dato momento esegue; o il software della macchina ospitante o software della macchina ospitata. Il codice dei processi virtualizzati gira direttamente sulla CPU, quindi potenzialmente è molto efficiente.

Nella simulazione invece l'astrazione offerta simula una macchina fisica, ma questa potenzialmente può avere una architettura hardware diversa dalla macchina ospitante. La differenza sta nel come viene eseguito il codice all'interno dell'ambiente virtualizzato. L' *Hypervisor* dell'emulatore eseguirà il codice delle istruzioni via software, perché non può eseguirlo direttamente sull'architettura CPU del sistema Ospitante.

La simulazione permette sì di mischiare le carte ma con un grosso prezzo in termini di efficienza.

Esistono due tipologie di *Hypervisor*:

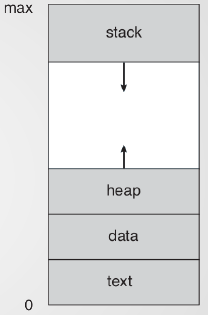
* + Tipo 1: gira direttamente sull'hardware, è l' *Hypervisor* stesso che funge da gestore delle risorse hardware.
  + Tipo 2: abbiamo un sistema operativo ospitante e un software che implementa un *Hypervisor*.   
    L' *Hypervisor* viene eseguito in modalità utente e per funzionare in modo efficiente ha bisogno di installare dei moduli kernel all'interno del kernel del SO ospitante, permettendo di gestire le risorse in maniera più efficiente.

# Processi

**Programma:** Un programma è una sequenza più o meno organizzata di istruzioni comprensibili all' architettura del sistema operativo.

Ogni volta che eseguiamo un programma creiamo una sua istanza di utilizzo, questa istanza di utilizzo è proprio il concetto di processo.

**Un processo**: è una sorta di copia attiva di un programma, che ha un suo stato interno e che a partire dal codice del programma può svilupparsi in vari modi.



## Stato interno di un processo

**Lo stato interno** di un processo è composto da:

* **Spazio degli indirizzi**, che a sua volta è composto da:
  + - Text, contiene il codice binario da eseguire.
    - Dati, strutture dati statiche come variabili.
    - Stack, contiene le chiamate a funzioni e le variabili al loro interno.
    - Heap, contiene allocazioni dinamiche di oggetti, vettori o nodi.

Heap e Stack crescono e si riducono dinamicamente. 

* + Copia dei registri della CPU;

A un processo è tipicamente associato una CPU che lo esegue, e all'interno della CPU abbiamo dei registri che possono avere, in un dato istante, dei valori. Questi valori fanno parte dello stato interno del processo.

Sarà quindi necessario salvare questi valori perché se a un certo punto quella stessa CPU cambia processo (***Context-Switch***) allora quei valori verranno sovrascritti con quelli del novo processo in entrata.

* + Lista de file aperti;

Un processo per lavorare probabilmente avrà bisogno di aprire diversi file. Bisogna tenere traccia di tutti questi file, del punto in cui li si stanno leggendo ecc… e lo si fa grazie alla lista dei file aperti.

* Allarmi pendenti;

Sono dei segnali, una sorta di interrupt di processi software, che fanno eseguire una certa routine.

* Processi imparentati; relazioni con altri processi.

## Tabella dei Processi

process state 
process number 
program counter 
registers 
memory limits 
list of open files Tutte queste informazioni, soprattutto quelle che non fanno parte dello spazio di indirizzamento del processo, devono stare da qualche parte all'interno del sistema.

Una struttura dati preposta a questo scopo è la ***tabella dei processi.***

E' una tabella il cui generico record riguarda un processo attualmente in esecuzione all'interno del sistema. Quindi per ogni processo vivo, ancora all'interno del sistema, esiste un record all'interno della tabella dei processi. Questo record si chiama ***PCB***, che è una sorta di carta di identità del processo e permette di raggiungere qualunque informazione del processo del record.

L' idea è che quando un processo termina, il suo PCB viene eliminato dalla tabella. E' una tabella dinamica.

### Un PCB contiene:

1. Process state (*ready*, *running*, *blocked*).
2. Process number: *PID.*
3. Registro PC (*Program Counter*)
4. Uno spazio Register, ovvero uno spazio idoneo a mantenere una copia integrale dei registri della CPU. In questo modo posso salvare, nel caso di *context-switch*, i valori attuali dei registri nel PCB del processo
5. Memory Limits, ovvero i limiti che indicano dove inizia lo spazio degli indirizzi del processo e dove finisce.
6. Lista dei file aperti.
7. ecc…

### Cosa avviene quando l'hardware notifica un Interrupt alla CPU?

1. L' hardware salva nell' attuale *Stack* il *Program Counter* e il *PSV* (registro che contiene tanti flag, tra cui modalità di esecuzione).
2. L' hardware si occupa di rintracciare e caricare la procedura associata all' Interrupt triggerato, dal vettore degli Interrupt.
3. Il software salva tutti i registri nel *PCB* del processo che si è interrotto.
4. Esecuzione della procedura di servizio per l'Interrupt.
5. Interrogazione dello *scheduler* per sapere con quale processo proseguire.
6. Ripristino dal PCB dello stato di tale processo (registri, ecc…).
7. Ripresa nel processo corrente.

## One program counter Process switch Process Modello dei processi

Ogni processo ha una CPU virtuale dedicata interamente a lui.

La CPU fisica, tramite Process-Switch, esegue i processi in *pseudo-parallelismo* ma i processi, fisicamente, vengono eseguiti in sequenza, uno di seguito all'altro.

## Creazione e Terminazione dei processi

### Creazione:

All'avvio di un sistema operativo multi-programmato vengono lanciati dei processi, a prescindere dalla volontà dell'utente.

I processi, in generale, vengono creati ad opera di qualcuno.

In fase di avvio, ovvero l'atto in cui il kernel viene portato in memoria (in caso di sistemi unix), viene creato una sorta di processo progenitore di tutti, chiamato ***init*** e a partire da questo vengono creati in cascata tutta una serie di altri processi di servizio.

Un nuovo processo viene creato tramite chiamata di sistema su richiesta di un altro processo già esistente.

La metodologia di creazione di un nuovo processo è riconducibile a due approcci:

* 1. Approccio universale dei sistemi UNIX.
  2. Approccio dei sistemi Windows.

#### *Approccio sistemi Windows:*

Windows, nelle API 32, prevede una chiamata ***CreateProcess*** che viene invocata da un processo esistente e che prende molti parametri, per creare un nuovo processo. La nuova entità sarà indipendente dal padre, avrà un proprio stato interno e condividerà poco con il processo padre. Il processo figlio partirà da zero, non avrà un stato interno.

#### *Approccio sistemi UNIX:*

I sistemi UNIX utilizzano due meccanismi, ***fork*** ed ***exec***.

##### fork:

La *fork* è una chiamata semplice, che non prende nessun argomento e che viene invocata da un processo esistente.

Questa chiamata duplica il processo padre, creando una nuova istanza, un nuovo processo indipendente che sarà un clone del processo padre. Avrà quindi un proprio spazio di indirizzamento proprio e indipendente che sarà uguale (stessi valori nei registri ecc…) a quello del padre.

Il processo figlio non partirà da zero perché avrà già uno stato interno.

##### exec:

La chiamata *exec* è una chiamata di sistema che, a partire dal punto di biforcazione tra processo padre e processo figlio, permette di differenziare i flussi di esecuzione e far eseguire al figlio qualcosa di diverso dal padre.

Infatti quando chiamata da un processo esistente X, azzera lo spazio di indirizzamento del processo X e questo spazio viene predisposto per eseguire una applicazione specificata tra gli argomenti della *exec*. Dopodiché salterà alla prima istruzione del codice del programma

* La combinazione di *fork* ed *exec* permette di creare un nuovo processo e inizializzarlo con un programma.
* Il processo padre prima della *fork* avrà dei dati (ad esempio una variabile x=10), il processo figlio dopo la *fork* si ritroverà gli stessi dati del padre, ma questi dati saranno del tutto indipendenti da quelli del padre.
* La *fork* crea un processo con un suo PID, la *exec* ricicla il processo (stesso PID) per eseguire un programma.

### Terminazione:

Un processo termina per vari motivi:

1. Uscita volontaria, in cui il processo stesso, dopo aver concluso le sue istruzioni, decide di auto-terminarsi, tramite la chiamata di sistema ***exit*** (UNIX), ***ExitProcess*** (Windows).

L'entità processo viene distrutta e le risorse occupate vengono liberate e rese riutilizzabili.

1. Uscita volontaria del processo con un codice di errore, tramite chiamata di sistema.  
   Ad esempio tentare di aprire un file che non esiste.
2. Uscita involontaria, errori critici che non possono essere controllati e gestiti dal processo.  
   E' il sistema operativo che interrompe brutalmente il processo, senza alcuna chiamata di sistema.
3. Uscita Involontaria, terminazione a causa di una richiesta esterna.  
   E' un processo X che termina un processo Y tramite una chiamata di sistema, ***kill*** (UNIX) e ***TerminateProcess*** (Windows)

In caso di sistema multi-utente, un utente X non può killare i processi di un utente Y.

## new admitted ready interrupt exit running terminated scheduler dispatch I/O or event completion I/O or event wait blocked Stato di un processo:

3 stati principali: **ready**, **running** e **blocked**.

2 stati addizionali (iniziale e finale): **new** e **terminated**.

### Esaminiamo gli stati cruciali:

* **ready**: il processo è pronto ad essere eseguito, cioè ad avere assegnato una CPU.  
  In un dato istante, in un ambiente multi-programmato, cioè con tanti processi, c'è ne sono tanti in stato di ready. Quando la CPU si libera, questa deve scegliere a quale processo assegnarsi. Questa scelta viene fatta dallo ***scheduler*** tra un insieme di processi ready. Tutti i processi ready sono candidati ad avere assegnata la CPU.

La transizione tra **ready** a **running** viene chiamata ***scheduler dispatch*** e si intende l'azione del predisporre un processo ad utilizzare la CPU.

* **running**: il processo sta utilizzando effettivamente la CPU. In dato istante ci può essere solo un processo in esecuzione, per CPU, ovvero quello scelto dallo *scheduler* e caricato sulla *CPU*.

Se un processo termina o incombe in un errore non critico, eseguendo la ***exit***, passa allo stato **terminated**.

La transizione ***I/O or event wait*** porta il processo nello stato di **blocked**.

Questa transizione si riferisce allo scenario in cui il processo compie una operazione che lo porta ad interagire con un dispositivo Hardware di input/output.

* **blocked**: un processo passa volontariamente, tramite chiamata di sistema, allo stato di *blocked* perché queste operazioni di *I/O* sono tipicamente lente. Per evitare di sprecare cicli di CPU questo processo che ha fatto la richiesta si pone in uno stato transitorio di attesa, fino a quando il **Controller** del dispositivo di *I/O* non notificherà la risposta della richiesta innescata, tramite un **Interrupt**.

Nello stato di *blocked* si impedisce allo *scheduler* di selezionare il processo in attesa.

Quando il meccanismo di *Interrupt* (nel caso di dispostivi I/O o in generale di un evento) notificherà l'arrivo dell'esito della richiesta, ci sarà una transizione del processo di ***ready I/O or event completion*** e il processo potrà essere assegnato alla CPU quando verrà scelto dallo *scheduler*.

#### *Prelazione:*

Un processo in *running* passa allo stato di *ready* (***transizione Interrupt***), ma perché?

Il motivo è riconducibile ad una caratteristica di come la CPU viene schedulata (ripartita fra i vari processi).

Un programma che non esegue operazioni di *I/O* potrebbe usare la CPU per un tempo infinito, non applicando la prelazione e monopolizzando la CPU per se stesso.

La prelazione è un meccanismo che permette al sistema operativo di interrompere l'esecuzione di un processo, dopo un certo tempo (**quanto di tempo**), togliendogli la CPU e evitando la monopolizzazione, portando il processo nello stato di *ready*.

Il sistema operativo prende possesso della CPU tramite chiamate di sistema, ma se un processo non effettua chiamate di sistema come fa il sistema operativo a prendere possesso della CPU?

Il meccanismo di prelazione viene implementato tramite un *Interrupt* legato al **clock di sistema**, che viene programmato in modo tale da innescarsi ad una data scadenza.

Alla scadenza verrà triggerato questo Interrupt che periodicamente darà il controllo della CPU al sistema operativo.

## Code e accodamento:

ready queue 
I/O 
I/O queue 
child 
executes 
CPU 
I/O request 
time slice 
expired 
fork a 
child L'insieme dei processi **ready**, essendo un insieme dinamico, viene gestito tramite una coda chiamata **coda dei processi ready** (*Ready Queue*).

Sostanzialmente è un struttura dinamica composta dai PCB dei soli processi nello stato ready. Quando un processo cambia stato dal *ready*, viene eliminato dalla coda e verrà reinserito quando riassumerà lo stato di *ready*.

#### *I/O Queue:*

Quando un processo passa allo stato *blocked* il suo PCB viene inserito nella coda I/O Queue del Controller del dispositivo I/O a cui il processo ha fatto richiesta.

(Coda dei processi bloccati da richieste I/O)

# Thread

E' un modello di sviluppo per applicazioni moderne.

Consideriamo un processo come un raggruppamento di risorse, attorno ad un flusso di esecuzione. L'idea è generalizzare il concetto e utilizzare lo stesso gruppo di risorse, in particolare lo stesso *spazio di indirizzamento*, tra più flussi d'esecuzione (*Thread*).

Una caratteristica fondamentale è che i *Thread* di un processo condividono le risorse tra di loro.

In particolare i vari ***Thread* imparentati** (ovvero i Thread che lavorano in uno stesso processo) condivideranno tra di loro tutto ciò che è presente nello spazio di indirizzamento del Processo.

repisters 
Stack 
•egs'er-s 
Slack 
Stock 
thread 
Processo process 
Thread 
Kernel I Thread possono comunicare e scambiarsi dati tra di loro. Tra Thread imparentati non c'è protezione, perciò un Thread può pasticciare i dati degli altri.

### Elementi caratterizzanti di un Thread:

1. **PC** (Program Counter), **Registri**, **Stack** e **Stato** sono distinti per ogni Thread.
2. **Spazio degli indirizzi**, che è condiviso tra Thread Imparentati:
   * + **Text, contiene il codice binario da eseguire.**
     + **Dati, strutture dati statiche come variabili.**
     + **Heap, contiene allocazioni dinamiche di oggetti, vettori o nodi.**
3. Scheduling dei Thread:  
   I vari Thread imparentati, in generale, vengono gestiti dal sistema operativo e questo, quando assegnerà la CPU, si troverà a scegliere non solo a quale Processo ma, se questi sono composti da più Thread, anche a quale Thread andare ad assegnare la CPU. Quindi vi è anche uno scheduling tra Thread.  
     
   Quando la CPU passa da un Thread ad un altro Thread Imparentato abbiamo un Context-Switch, perciò dobbiamo salvare i registri del Thread uscente e caricare quelli del Thread entrante. La differenza tra un Context-Switch tra Thread Imparentati e Thread separati (quindi processi distinti) sta nel fatto che i Thread Imparentati si trovano nello stesso processo perciò l'operazione di Context-Switch è molto più semplice e veloce dato che non dobbiamo riprogramma (rimappare) la **MMU** (Unità di Gestione della Memoria, gestisce le richieste di accesso alla memoria generate dalla CPU).  
     
   Per questo motivo, nello scheduling, i Thread Imparentati hanno più probabilità di essere scelti come candidati, proprio perché è più conveniente in termini di efficienza.
4. Testo alternativo generato dal computer:
   Running 
   Blocked 
   4 
   Ready Cambiamento di stato dei Thread:

Valgono gli stessi stati e transizioni dei Processi.

1. Operazioni sui Thread;

* *thread\_create*: serve per creare un Thread. Un processo, quando viene creato, ha implicitamente un Thread nativo in esecuzione al suo interno. Questo Thread può creare Thread secondari con questa chiamata di sistema.
* *thread\_exit*: permette ad un Thread di terminare spontaneamente. Quado un processo è composto da più Thread, continuerà ad esistere fintanto che c'è un Thread attivo al suo interno.
* *thread\_join*: Serve per sincronizzare l'azione di un thread con un altro, bloccando il Thread chiamante fintanto che il Thread indicato come parametro alla chiamata non ha finito (*thread\_exit*).
* *thread\_yield*: meccanismo collaborativo che permette di condividere spontaneamente la CPU, permettendo ad un Thread di cedere la CPU ad un suo Thread fratello.

## Implementazione dei Thread

Come viene implementato il modello a Thread in un sistema?

### thread user Kernel Kernel Th read Process Run-time system table Modello 1-a-molti: Thread a livello Utente

* Associazione tra l'unico flusso di esecuzione (un solo Thread kernel) e la molteplicità di task individuati dal programmatore.
* Modello basato sui Thread a livello utente.

Prima che i sistemi operativi supportassero nativamente i Thread, esistevano le **librerie run-time** che permettevano di gestire i Thread nell'ambito di processi utente. Il SO non sapeva neppure cosa fosse un Thread.

L' idea è che sarà il processo ad occuparsi di effettuare il Context-Switch spontaneamente tra un suo Thread Utente ed un altro, senza passare mai in modalità kernel, salvando i registri della CPU nella **tabella dei Thread** (che sta dentro la *run-time system*, che sta dentro il processo) del Thread uscente e caricare quelli del Thread entrante.

Il sistema operativo assegnerà un *quanto di tempo* al processo, il processo dividerà questo *quanto di tempo* tra i suoi Thread interni, intervallando la CPU tra di essi in maniera autonoma rispetto al SO.

Questo è un modello a Thread puramente collaborativo (*thread\_yield*), non può esserci prelazione perché solo il kernel può implementarla, e qui non ci sono chiamate di sistema.

**Pro**:

* il Context-Switch tra Thread utenti fratelli è rapidissimo, perché non vi è alcuna chiamata di sistema e non interviene il kernel (non c'è commutazione tra modalità utente -> kernel -> utente).
* Scheduling dei Thread personalizzato.

**Contro**:

* Se il flusso di esecuzione (uno dei Thread utente) fa una chiamata bloccante, questa blocca tutto il processo, perché transiterà nello stato di *blocked*.
* Se mal programmato, la CPU potrebbe essere monopolizzata da un Thread.

### p rocess Thread Kernel Process table Thread table k user thread kernel thread Modello 1-a-1: Thread a livello Kernel

* Associazione fra i task individuati dal programmatore e i Thread kernel usati dall'applicazione (5 Task => 5 Thread).
* Modello basato sui thread a livello kernel.

Il SO supporta, conosce e gestisce i Thread.

Lo spazio di indirizzamento dei Thread contenuto all'interno del kernel (**Thread Table**) e le operazioni sui Thread diventano chiamate di sistema.

**Pro**:

* Un Thread che fa una chiamata bloccante non blocca l'interno processo.

**Contro**:

* Il Context-Switch tra Thread imparentati è più veloce rispetto a quello tra processi, è però più lento rispetto a quello tra Thread utente. Perché in quel caso non interviene il kernel, qui c'è invece una commutazione tra modalità utente -> kernel -> utente che richiede tempo.
* Le operazioni di creazione e distruzione dei Thread a livello kernel sono più costose rispetto a quelle per Thread utente, anche se meno rispetto alla creazione e distruzione di un processo.

### l'i user thr on a kernel User Korrel Kernel 'n read Modello molti-a-molti: Modello Ibrido

E' il programmatore che gestisce come dividere i task tra Thread kernel e Thread utente. Prevedendo un certo numero di Thread Kernel, posso assegnare a ciascuno di essi uno (nel caso di task importante) o più Thread Utente.

# Comunicazione fra processi

Un primo modello per far comunicare due o più processi è utilizzare i canali di input/output, mappando l'output del primo processo come input di un altro processo. Questa procedura viene chiama pipe.

programma1 | programma2 => pipe

Nel modello a pipe il programma2 è bloccato fin quando non riceve l'input dal programma1, bisogna quindi coordinare le operazioni tra i vari processi.

### Problematiche sulla comunicazione:

* + *Corse Critiche (race condition):*

Due processi operano sulla stessa variabile. Il processo A e il processo B fanno, ad esempio, 10 incrementi ciascuno.

Alla fine dovrei trovarmi con la variabile incrementata di 20 volte… Ma cosa può succedere alla variabile in caso di sistema con *Prelazione*?

L' operazione di incremento (x = x+1), per quanto semplice, non è atomica, sarà perciò composta da almeno due operazioni macchina (una di *fetch* dalla memoria al registro, una di incremento, una di store dal registro alla memoria).

Potrebbe succedere che il SO rimuova la CPU al processo A, prima che questo abbia effettuato l'operazione di *Store*.

A questo punto la CPU, per caso, passa al processo B che viene caricato in memoria. La variabile X caricata in memoria dal processo B sarà però il vecchio record, non modificato dal processo A.

A questo punto immaginiamo che la CPU passi nuovamente al processo A, che viene caricato in memoria e riprende da dove aveva interrotto, ovvero l'operazione di *store*. Il processo A sovrascriverà la variabile X con un valore vecchio, non corrispondente agli incrementi passati.

Non è plausibile che un programma ogni tanto faccia cilecca… La probabilità che si verifichino questi effetti negativi aumenta con l'aumentare della complessità delle operazioni.

##### Soluzione: Mutua Esclusione

La soluzione passa dalla definizione delle **sezioni critiche**; ovvero quei pezzi di codice che vanno ad interagire con la struttura dati condivisa da più processi, isolarle dal resto del codice e garantirvi la **mutua esclusione** nell'esecuzione delle sezioni critiche.

**Mutua Esclusione**: garantire che in ogni dato istante ci sarà al più un processo nella rispettiva sezione critica. In questo modo avremo la garanzia che ci sarà al più un processo che lavora sulla struttura dati condivisa.

Se i due processi si trovano su due CPU diverse (parallelismo vero), bisogna bloccare il processo che tenta di entrare nella sezione critica se già un processo è nella sua sezione critica => ***Logica Bloccante***.

##### Come realizzare la Mutua Esclusione in ambito di processi Utente:

1. **Soluzione Disabilitare gli *Interrupt***: Se riuscissimo ad inibire transitoriamente gli *Interrupt* per un processo, giusto il tempo di eseguire la sua sezione critica, evitiamo che il processo venga bloccato dalla Prelazione (*Interrupt del clock*) o da richieste I/O, prima di aver completato la sua sezione critica.  
     
   **Problema**: come disabilito gli *Interrupt* in un generico processo Utente, se per farlo devo essere in modalità kernel? E' sicuro farlo per un generico processo Utente?  
     
   **Uso**: questa soluzione viene usata invece nel caso di processi kernel, dando per scontato che i programmatori kernel siano esperti, in processori con una sola CPU.  
   Disabilitare gli Interrupt non aiuta nel caso di sistemi multi-core, perché in questo caso bisogna prima adottare la **Logica Bloccante**.
2. **Soluzione Variabile di lock**: mettere una sorta di semaforo all'accesso della struttura dati, ad esempio una variabile booleana, che indica ai vari processi se c'è qualcuno dentro la sezione critica.   
   Inserendo nel codice, prima della sezione critica, una routine *enter\_region* che controlla se la variabile di lock se è 0 o 1.  
   Se 0 allora imposto la variabile a 1 e puoi entrare. Se è 1 fa **busy-waiting**, bloccando il processo con un ciclo while, anche se male perché sprecherà cicli di clock.  
   Prima di uscire dalla sezione critica si esegue una *leave\_region* che ripristina la variabile a 0.  
     
   **Problema**: la variabile di lock può diventare essa stessa il problema, perché è una variabile condivisa e nel caso sfortunato in cui due processi eseguono la *enter\_region* contemporaneamente allora entrambi saranno dentro le sezioni critiche contemporaneamente.
3. int N=2 
   int turn 
   function enter_region(int process) 
   while (turn != process) do 
   nothing 
   function process) 
   turn = 1 
   - process **Soluzione Alternanza stretta**:

Abbiamo delle variabili condivise; ***N*** indica quanti sono i processi, ***turn*** stabilirà di chi è il turno per entrare nella sezione critica.  
L'idea che gli N processi si alterneranno in uno specifico ordine per entrare nella propria sezione critica. Il parametro ***process*** è l'identificativo del processo chiamante.  
  
**Pro**: Si può generalizzare a N processi.  
**Problema**:

* + Si fa *busy-waiting*.
  + Implica dei turni rigidi: se un processo A non ha necessità di entrare nella sua sezione critica, mentre un altro B si, allora B non potrà entrarvi fin quando A non sarà entrato prima nella sua, consumando il suo turno e passandolo poi a B.

1. Testo alternativo generato dal computer:
   int N=2 
   int turn 
   int interested[N] 
   function enter_region(int process) 
   other = 1 
   process 
   interested[process] 
   = true 
   turn = process 
   while (interested[otherl 
   = true and turn 
   nothing 
   function leave_region(int process) 
   interested[process] 
   = false 
   = process) do **Soluzione di Peterson**:  
   Abbiamo le stesse variabili condivise ma in più c'è un array ***interested[N]*** che indica per ogni processo, se quest'ultimo è interessato ad entrare nella sua sezione critica.  
     
   **Pro**:
   * Si può generalizzare a N processi.
   * Risolve il problema dei turni rigidi.

Problema:

* + Ancora *busy-waiting*.
  + Può avere problemi sui moderni multi-processori a causa del riordino degli accessi alla memoria casuale.

1. **Soluzione Istruzioni TSL o XCHG**Abbiamo un supporto Hardware, ovvero delle istruzione apposite all'interno del set di quelle disponibili nelle CPU moderne. Queste istruzioni (non kernel) assumono due semantiche possibili (**Test and set lock**) e (**Xchange**).  
   Ancora *busy-waiting*!  
   * + leave_region : 
       MOVE 
       RET _region : 
       enter 
       TSL 
       REGISTER , LOCK 
       CMP 
       enter _ region 
       JNE 
       RET **TSL: è una istruzione atomica (non scindibile, ovvero viene effettuata con un solo ciclo di clock, una sola istruzione macchina, quindi non può essere interrotta a metà) che prende come argomenti un *registro* e una *locazione di memoria*, usata come variabile di lock.  
       L'idea è che per ogni struttura dati condivisa io vi associo una variabile di lock, la corsa critica sulla variabile non si può avere perché?  
       La variabile di lock è condivisa come la struttura dati, la TSL ha due effetti: (*lettura*) fetch dalla memoria della variabile di lock nel registro -> (*scrittura*) la variabile di lock viene automaticamente impostata ad un valore diverso da 0.  
         
       In caso di *multi-CPU* le due CPU potrebbero lavorare sulla stessa variabile di lock, ostacolandosi… ma la TSL, quando in esecuzione, ha l'effetto di bloccare il bus della memoria a tutte le altre CPU, evitando che ci sia un interlacciamento all'interno del bus della memoria sulla variabile di lock a opera di più CPU.**

* + **Xchange**: prende come argomenti un *registro* e una *locazione di memoria*, usata come variabile di lock, e scambia i valori del registro con quelli della variabile di lock.

###### *Cause dello Spin-lock:*

Tutte le soluzioni presentate fino ad ora fanno ***spin-lock*** (*busy-waiting*), ciò potrebbe comportare il verificarsi del fenomeno di **inversione di priorità**: ovvero del bloccaggio della CPU ad un processo H a più alta priorità che attende, facendo spin-lock usando attivamente la CPU, che la variabile di *lock* torni a 0. Assumendo che la variabile di lock sia stata settata ad 1 da un processo L a bassa priorità su cui è stata eseguita la prelazione.

**H** attende, usando la CPU, che *lock=0*.

**L** attende che la CPU passi a lui per uscire dalla sua sezione critica e settare *lock=0*.

Lo Scheduler non vede perché dover passare la CPU ad un processo a più bassa priorità.

H e L sono bloccati.

1. **Soluzione Primitive Sleep e Wakeup**

Una soluzione che garantisca mutua esclusione senza fare pause attive (*busy-witing*) prevede che ci sia un modo assistito dal sistema operativo (tramite chiamate di sistema), che permetta ad un processo utente di auto-addormentarsi o risvegliare un processo.

### Semafori

La generalizzazione e l'aggiustamento del concetto di *Sleep* e *Wakeup* dà vita a quella che è la primitiva ***Semaforo***.

L'idea è che se arriva una *Wakeup* ad un processo che non è dormiente, questa non si deve perdere, anche se arrivano più *Wakeup*.

Il *Semaforo* è quindi un meccanismo di auto-addormentamento in cui le *Wakeup* non consumate vengono memorizzate in quantità progressiva.

I semafori vengono utilizzati per:

* Garantire mutua esclusione. (semaphore mutex)
* Sincronizzare i processi Producer e Consumer. (semaphore empty, full)

#### *Definizione e proprietà:*

* Il *Semaforo* è una primitiva che ha uno stato interno, una variabile intera S, che terrà il conto delle *Wakeup* non consumate.
* Lo stato interno, la variabile **S**, non deve mai diventare negativa.
* Lo stato interno verrà modificato da due operazioni ***down*** a ***up*** (o rispettivamente *wait* e *signal*), che incrementano o decrementano il suo valore.

L'implementazione della *down* (*wait*), di fronte ad un semaforo nullo (S=0) deve bloccare il tentativo di decremento al fine di evitare la negatività dello stato interno.

Questa chiamata potrà pertanto diventare bloccante, nel senso che non ritornerà al chiamante finché non si creano i presupposti per cui il decremento si possa applicare, cioè fino a quando qualcuno non farà una *up* (*signal*).

La *wait* e la *signal* sono primitive del sistema operativo al livello kernel, quindi eseguite in modalità kernel, perciò nel caso di blocco provocato dalla *wait* su un semaforo nullo questo sarà un blocco passivo che non sprecherà cicli di clock.

Il blocco sarà passivo perché la *wait* andrà a rimuovere il PCB del processo dalla coda dei processi ready, mentre una futura *signal* lo reinserirà.

Il SO deve però garantire l'atomicità delle due primitive, cioè nell'esecuzione della *wait* e della *signal* ci deve essere mutua esclusione nell'accesso allo stato interno del semaforo, ma come viene ottenuta?

#### *Come si ottiene la mutua esclusione nelle chiamate wait e signal?*

Il kernel, prima di eseguire la *wait* e la *signal* si assicurerà che nessuno si possa interlacciare:

* Su un sistema single-core si disabilitano temporaneamente gli interrupt.
* int N=IOO 
  semaphore 
  semaphore 
  semaphore 
  function producer() 
  while (true) do 
  item = produce_item() 
  down(empty) 
  down(mutex) 
  insert_item(item) 
  up(mutex) 
  up(full) 
  mutex = 1 
  empty = N 
  full = O 
  function consumer() 
  while (true) do 
  down(full) 
  down(mutex) 
  item = remove_item() 
  up(mutex) 
  up(empty) 
  consume_item(item) Su un sistema multi-core si implementa uno *spin-lock* tramite le istruzioni ***TSL/XCHG***.  
  Questo spin-lock (pausa attiva della CPU) sarà legato solo all'esecuzione della *wait* e della *signal*, quindi trascurabile in termini di sprechi di CPU. Cioè il lock serve solamente per evitare la *race condition* nella scrittura dello stato interno (zona critica) del semaforo.

Il ***mutex*** viene inizializzato ad 1, cosicché il primo processo a richiedere di entrare in sezione critica, chiamando la *wait*, decrementerà il suo stato interno a 0 e nel caso lo richieda anche un secondo mentre vi è già il primo, questo verrà bloccato. Prima di uscire dalla *sez. crit.* chiamerà una *signal* sulla *mutex*, sbloccando quel secondo processo.  
  
I semafori ***empty*** e ***full*** si riferiscono invece alla quantità di memoria condivisa usata e disponibile. Un produttore che vuole scrivere dovrà chiamare una *wait* su *empty* per vedere se ci sono slot disponibili e una *wait* sulla *mutex* per chiedere di entrare nella sua sezione critica. Prima di uscire chiamerà una *signal* su *full* per segnalare di aver riempito uno slot e una *signal* sulla *mutex*. Un consumatore dovrà innanzitutto verificare (*wait* su *full*) che vi siano effettivamente degli slot da leggere e, dopo averli letti in mutua esclusione, svuotare lo slot letto (*signal* su *empty*) ed uscire dalla *sez. crit*.

##### Mutex e Thread Utente:

Se un *thread* utente effettua una *down* su un *semaforo*, e questa *down* è bloccante, allora in realtà si viene a bloccare l'interno processo e quindi tutti i suoi *thread*.

Per risolvere questo problema è possibile implementare efficientemente i *mutex* facendo uso delle istruzioni **TSL/XCHG**, ma anziché ciclare attivamente (*spin-lock*) si cede la CPU (*thread\_yield*).

#### *Futex: (Sistemi linux only)*

E' una implementazione del meccanismo dei *mutex*, che combina una parte utente e una parte kernel per ottenere una implementazione più veloce dei *mutex*.

**In che senso più veloce?**

L'aspetto negativo dei *mutex* è legato all'overhead introdotto dalle chiamate *down* e *up*, che sono chiamate di sistema che quindi innescano il meccanismo di *trap* per il passaggio da modalità *utente* a modalità *kernel* (salvataggio dei registri e del contesto di esecuzione, …).

Frazioni di secondo che però se chiamate spesso, come capita nella comunicazioni tra *thread*, si fanno sentire.

Nei sistemi linux è implementato questo meccanismo chiamato ***futex*** che si pone di limitare al minimo l'intervento del *kernel*.

L'idea è la seguente: un futex ha due componenti principali.

* Una componente lavora all'interno del kernel, gestendo le chiamate di sistema offerte dal sistema operativo.
* Una libreria utente.

Il cuore di questo meccanismo è l'implementazione di una variabile di lock in modalità utente, basata sulle istruzioni di **TSL/XCGH**, ma sostituendo lo spin-lock di queste istruzioni con una chiamata di sistema che, solo nel caso in cui un processo rilevi la necessità di bloccarsi (mutua esclusione), invoca l'aiuto del kernel, che ne provoca l'addormentarsi.

Questo meccanismo permette di evitare il ciclo di **spin-lock** e di addormentare un processo solo quando necessario. In caso di non contesa invece, il futex riesce ad entrare nella sezione critica senza effettuare nessuna chiamata di sistema.

### Monitor

Sono un costrutto per la risoluzione dei problemi di sincronizzazione e di garanzia di mutua esclusione alternativo ai semafori.

I monitor nascono per sopperire alla "difficoltà" di utilizzo dei semafori, infatti sono dei costrutti ad alto livello disponibile su alcuni linguaggi recenti (Java, C#, …).

Sono oggetti che possiedono dati e metodi e che godono di una proprietà innata fondamentale, ovvero la **garanzia di mutua esclusione all'accesso al monitor**.

Ovvero i dati interni del monitor sono accessibili esclusivamente dal monitor stesso (private) attraverso dei metodi, **questi metodi sono protetti dalla mutua esclusione**.

Il monitor è un oggetto condiviso tra i *thread* di una stessa applicazione in esecuzione e questa garanzia di mutua esclusione garantisce che all'interno del monitor stia operando al più un *thread*.

Il monitor contiene tra i suoi dati la zona critica a cui vogliono accedere i *thread*, questi ultimi per accedervi devono sostanzialmente entrare dentro il monitor, che a sua volta garantisce di natura la mutua esclusione.

Il codice generato dal linguaggio di programmazione usa gli strumenti più a basso livello che già conosciamo (semafori, futex, …).

#### *Monitor vs Semafori:*

* I monitor prescindono dal linguaggio, i semafori invece sono presenti in ogni sistema operativo e possono essere usati in qualsiasi linguaggio.
* I monitor possono essere utilizzati per gestire i *thread* solo per un singolo processo, mentre i semafori possono essere usati da processi distinti.
* I monitor sono un approccio più comodo ma ci sono una serie di restrizione che non lo rendono universale.

#### *Meccanismo di sincronizzazione:*

Il modello dei monitor prevede dei costrutti addizionali che gestiscono la sincronizzazione dei *thread* che operano su un monitor, sono le **variabili condizione**.

Ovvero delle variabili interne definite dentro un monitor che sono usate dai *thread* per accoppiare le operazioni di *signal* e *wait*.

Un *thread* può quindi auto-addormentarsi (pausa passiva->*blocked*) dentro il monitor, modificando opportunamente una ***variabile condizione*** e sbloccando l'accesso al monitor a qualcun altro. Per risvegliarsi si aspetta una *signal* da un altro *thread* sulla stessa variabile su cui il *thread* si era addormentato.

**Ma se un *thread* esegue una *signal* su un altro processo addormentato, non avremo due processi attivi dentro lo stesso monitor violando così la sua proprietà di mutua esclusione?**

La *signal* viene infatti implementata con diverse semantiche:

* *monitor Hoare*, **signal & wait**: Il *thread* che chiama la *signal* viene bloccato per risvegliare il *thread* addormentato, quando quest'ultimo uscirà dal monitor, verrà risvegliato il *thread* di prima.
* *monitor Mesa*, **signal & continue**: Il *thread* che chiama la *signal* sul *thread* dormiente, prima finisce di operare dentro il monitor poi, una volta uscito, tenendo conto della sua precedente *signal*, risveglierà quel determinato *thread*.

* compromesso, **signal & return**: il *thread* chiamerà la *signal* come metodo uscente dal monitor.

### Modello a memoria condivisa: scambio messaggi tra processi

Il sistema operativo fornisce due primitive:

* send(destinazione, messaggio) {**POTENZIALMENTE BLOCCANTE**}
* receive(sorgente, messaggio) {**BLOCCANTE**}

L'idea è che se scrivo un algoritmo che, tramite queste chiamate di sistema, riesce a prendere un problema e suddividerlo in tanti sotto-problemi da elaborare all'interno di un *thread* e poi, sempre tramite lo scambio di messaggi, ricostruire il risultato finale, ottengo una applicazione che essere distribuita si una serie di *thread* che collaborano.

Il vantaggio è che un programma scritto con questa logica può essere scalato su tante macchine, ad esempio un cluster composto da 100 macchine.

I messaggi vengono imbucati dalla *send* all'interno di una ***mailbox***, la *receive* ha invece come riferimento la *mailbox* stessa e non il mittente.

La *mailbox* funge quindi da ***buffer***, perciò anche la *send* po' diventare bloccante se la *mailbox* è piena.

Il fatto di avere questa logica bloccante risolve il problema della sincronizzazione e non ho concorrenza perché è il *kernel* che interagisce con la *mailbox*, tramite le primitive *send* e *receive* (che sono atomiche).

* Questo meccanismo ha però lo svantaggio di avere un elevato ***overhead***, che rallenta parecchio la velocità di esecuzione nel caso di scambio di molti messaggi.

### Problema dei 5 Filosofi

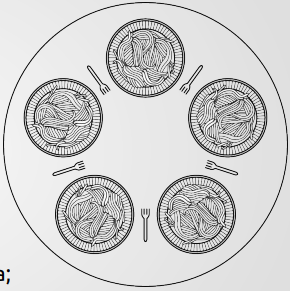
5 Filosofi sono riuniti attorno ad una tavola.

Essi possono: Pensare o Mangiare. Per mangiare hanno bisogno di 2 risorse, due forchette.

Ogni filosofo dispone di due stati: mangiare e pensare.

La soluzione che andiamo a cercare è una soluzione che permette ai 5 filosofi di acquisire correttamente le risorse forchette di cui ha bisogno per mangiare, e che in particolare massimizzi la possibilità di mangiare per tutti (che permetta ad esempio a più di un filosofo di mangiare, al più 2 contemporaneamente in questo caso).

Problema dei 5 filosofi: 
soluzione bQSQtQ sui semafori 
int N=5; int THINKING=O 
int HUNGRY-I; int EATING-2 
int state[N] 
semaphore mutex=l 
semaphore ... , O} 
function philosopher(int 
while (true) do 
think() 
take_forks(i) 
i) 
eat ( ) 
put _ forks (i) 
function left(int i) 
= i-l mod N 
function right(int i) = 
i+l mod N 
function test(int i) 
if and 
function take_forks(int i) 
down(mutex) 
state [i] -HUNGRY 
test(i) 
up(mutex) 
down(s [i] ) 
function put_forks(int i) 
down(mutex) 
test(left(i)) 
test(right(i)) 
up(mutex) 
statetleft(i)] I-EATING and statetright(i)] I-EATING Problema dei 5 filosofi: 
soluzione bQSQtQ sui monitor 
int N=5; int THINKING=O; int HUNGRY-I; int EATING-2 
monitor dp_monitor 
int state[N] 
condition self [N] 
function take_forks(int i) 
state[i] = HUNGRY 
test(i) 
if state[i] EATING 
function put_forks(int i) 
state[i] = THINKING; 
function test(int i) 
function philosopher(int i) 
while (true) do 
think() 
dp_monitor. take_forks (i) 
eat() 
dp_monitor. 
if ( EATING and state[i] 
and statecright(i)] EATING ) 
state[i] 
= EATING 
= HUNGRY 

### Problema dei lettori e scrittori: soluzione n.2 bQSQtQ sui monitor monitor rw_rnonitor int rc O; boolean busy_on_write false condition read,write function if (busy_on_write OR in_queue(write)) wait(read) rc = rc+l signal(read) function end_read() rc = rc-l if (rc O) signal(write) function start_write() if (rc > O OR busy_on_write) = true function end_write() busy_on_write = false if (in_queue(read)) signal (read) else signal(write) function reader() while true do rw_monitor. start_read() read_database() rw_monitor. end_read() use_data_read() wait (write) function writer() while true do think_up_data() rw_monitor. start_write() write_database() rw_monitor. end_write() Problema dei lettori e scrittori su un DB

# Scheduling

E' comodo distinguere due tipologie estreme di comportamenti dei processi.

* Processi **CPU-bounded**:  
  E' un processo molto interessato ad utilizzare la *CPU* piuttosto che fare operazioni di *input/output*. Questo tipo di processo ha periodi di *CPU burst* relativamente lunghi e poco frequenti.
* Processi **I/O-bounded**:  
  E' un processo più interessato ad effettuare operazione di input/output piuttosto che ad usare attivamente la CPU. Questo tipo di processo ha periodi di *CPU burst* relativamente brevi e più frequenti.

Long CPU 
WaimG to' 10 
CPU ***CPU burst***:

Lasso di tempo che trascorre tra una operazione di input/output e l'altra, di un processo.

### Algoritmo di Scheduling:

Un *algoritmo di scheduling* viene chiamato quando la *CPU* si libera per scegliere a quale processo della coda dei processi pronti assegnare la *CPU*, ma quando si libera?

* Terminazione e creazione di processi.
* Chiamata bloccante (es. *I/O*) e arrivo del relativo interrupt.
* *Interrupt* periodici:
* sistemi non-preemptive (**non applicano la prelazione,** quindi ogni processo userà la *CPU* fino a quando non avrà completato).
* sistemi preemptive (**applicano prelazione del *SO***).

Gli algoritmi di scheduling si differenziano sulla base dei differenti ambienti di lavoro: *batch*, *interattivi* e *real-time*. Ma indifferentemente da questo, avranno due obiettivi comuni…

#### *Obiettivi degli algoritmi di scheduling:*

* **Principio di equità** nell'assegnazione della *CPU*: ovvero trattare i processi con la stessa importanza e concedere loro lo stesso tempo di utilizzo.
* **Bilanciamento** nell'uso delle risorse: permettere di bilanciare non solo l'uso di tutti i core in modo similare, ma anche il reparto di input/output.

#### *Come si misura l'efficienza di un algoritmo di scheduling?*

Bisogna guardare a questi valori…

##### Sistemi Batch:

* metrica **Throughput**: indica la produttività di un sistema, ovvero quanti compiti riesce a smaltire in un lasso di tempo.
* **Tempo di Turnaround** (tempo di completamento): quanto tempo passa tra l'immissione all'interno del sistema del processo ed il suo completamento. Istante in cui il processo viene immesso nel sistema **+** Istante in cui gli viene data la *CPU* dallo Scheduler **+** Tempo di elaborazione;
* **Tempo di Attesa**: Tempo di attesa del processo all'interno della coda dei processi pronti. Istante in cui il processo viene immesso nel sistema **+** Istante in cui gli viene data la *CPU* dallo Scheduler;

##### Sistemi Interattivi:

* **Tempo di Risposta**: quanto tempo il sistema impiega a reagire ad un input.

##### Sistemi real-time:

* **Rispetto delle scadenze**.
* **Prevedibilità**.

### Scheduling nei Sistemi batch:

*Un sistema Batch è caratterizzato da dei processi chiamati Job che devono essere calcolati. Sono delle operazioni che non interagiscono con l'utente.*

#### *First-Come First-Served (FCFS) {o scheduling per ordine di arrivo}*

Soddisfa le richieste dei processi che vogliono essere schedulati esattamente rispettando l'ordine di arrivo.

* Implementato tramite una semplice coda FIFO.
* Non-preemptive.

#### *Shortest Job First (SJF) {o scheduling per brevità}*

Andare a scegliere sempre il processo che comporta un costo minimo, cioè con la durata minore.

* Non-preemptive.
* Dobbiamo conoscere o supporre quando tempo impiego ogni lavoro.
* Ottimale solo se i lavori sono tutti subito disponibili.

#### *Shortest Remaining Time Next (SRTN)*

E' una variante dello SJF in cui viene introdotto il concetto di prelazione.

Nello specifico, mentre vi è un processo in esecuzione sulla CPU, se si aggiunge un nuovo processo nella coda dei processi pronti, confronto la durata del nuovo processo con il tempo rimanente del processo in corso e in caso di vantaggio verrà applicata la prelazione.

* Schedulino nei sistemi batch 
  FCFS: 
  11 12 
  16 
  tempi di attesa: P P P (media 475); 
  tempi di completamento: P P P P (media 8.75); 
  SJF: 
  12 
  tempi di attesa: P —O; P P P (media 4); 
  tempi di completamento: P P =10; P P (media 8); 
  AcciYQ 
  16 
  16 
  Qurata 
  SRTN: 
  p 
  p 
  p 
  11 
  tempi di attesa: Pl=9; P P P.=2 (media 3); 
  tempi di completamento: 6; P P (media 7). Versione preemptive dello SJF.

### Scheduling nei Sistemi Interattivi:

*Un sistema Interattivo è un sistema che, di fronte ad una chiamata di I/O deve reagire velocemente evitando che la CPU venga monopolizzata.*

#### *Round-Robin (RR)*

Si basa sul concetto di **quanto di tempo** (*timeslice*), durante il quale il processo di turno viene allocato nella CPU e concessogli un quanto di tempo paritario ad ogni processo, un processo alla volta.

* Versione con prelazione del FCFS.
* Preemptive.
* Ogni processo ha la garanzia che userà la CPU, al più ogni:

|  |
| --- |
| ***(n-1)q ms*** con [n: #processi, q: timeslice] |

* Un valore tipico di *quanto di tempo* (*timeslice*) si attesta tra i 20-50ms.

Alla scadenza del quanto di tempo, il processo viene rimesso alla fine della coda.

Se il *timeslice* è troppo lungo il programma andrà a scatti, se è troppo corto perdo prestazioni della CPU per via dei troppi context-switch e i conseguenti overhead.

#### *Scheduling a Priorità*

Introduce l'idea che non tutti i processi siano per forza uguali, introduce quindi l'idea della priorità, che produce processi più importanti di altri.

* **Regola di base**: si assegna la CPU al processo con più alta priorità.

|  |
| --- |
| Priorità alta: servizio di scheduling migliore.  Priorità bassa: servizio di scheduling peggiore. |

##### Assegnamento delle priorità:

* **Statica**: un processo mantiene la stessa priorità fino alla sua terminazione.
* **Dinamica**: un processo può cambiare priorità durante il corso della sua vita, ad esempio per i Sistemi Interattivi è opportuno favorire i processi *I/O bounded*.

La differenza sostanziale tra un processo *I/O bounded* ed uno *CPU bounded* è che:

Un processo *I/O bounded* tende a non usare per intero il suo quanto di tempo, dato che effettuerà una chiamata di sistema. Al contrario un processo *CPU bounded* tenderà ad usare per intero il suo quanto di tempo.

Se osservassimo negli ultimi quanti di tempo, quali processi hanno usato per intero o meno il quanto a loro disposizione allora potremo legare dinamicamente ogni processo ad un valore numerico di priorità.

##### Come viene gestita la priorità?

Un modo semplice su come l'algoritmo di scheduling potrebbe prendere le proprie scelte su una lista di processi con un valore di priorità sarebbe quello di scegliere sempre il processo con priorità più alta (**SJF o SRTN**).

###### *Problema e Soluzione:*

In un caso estremo potrei però avere una situazione di **starvation** in cui processi a bassa priorità, a causa della continua attività di processi a più alta priorità, usano poco e nulla la CPU. Per mettere una pezza a questo problema si può usare la tecnica dell'**aging**:

Se un processo sta da molto tempo in coda gli viene incrementata di una certa quantità la sua priorità, una volta finita di usare la CPU la sua priorità si resetta a quella normale.

##### Prelazione vs non-prelazione:

**Senza prelazione** significa che, una volta assegnata la CPU, questa non gli viene tolta se il processo non termina o effettua una chiamata di sistema, anche se viene inserito in corsa un processo con più alta priorità. (**SJF**)

**Con prelazione** invece avviene il context-switch tra processi. (**SRTN**)

##### Variante; scheduling a code multiple (classi di priorità):

Usato dai Sistemi Operativi (Linux, Windows, …).

Fissata una gerarchia delle priorità con N priorità, allora sostanzialmente avrò N code dei processi pronti. L'algoritmo di scheduling a questo punto avrà due principi di scelta:

1. Principio di scelta verticale, ovvero del valore di priorità:  
   Si applica la regola di base + prelazione.
2. Principio di scelta orizzontale, sceglierà un candidato all'interno della coda:  
   Si può applicare un algoritmo di scheduling secondo il grado di priorità verticale, ad esempio applicare:

* Alle alte priorità (dove potenzialmente ci stanno i processi *I/O bounded*) un *Round-Robin* con un quanto più basso. (Alta reattività - ~~Alto overhead~~ -> i processi *I/O bounded* usano quanti molto piccoli)
* Queue 
  headers 
  Priority 4 
  Priority 3 
  Priority 2 
  Priority 1 
  Runable processes 
  (Highest priority) 
  (Lowest priority) Alle basse priorità (processi *CPU bounded*) un *Round-Robin* con un quanto più alto. (Bassa reattività - Basso overhead)

#### *Shortest Process Next (SPN):*

L'idea è quella di indentificare il processo, tra quelli attualmente presente nella coda dei processi pronti, che avrà il ***CPU burst*** futuro più breve e assegnargli la *CPU*.

La durata del prossimo *CPU burst* verrà stimata dinamicamente sulla base dei *burst* precedenti.

#### *Scheduling garantito:*

Viene stabilita una percentuale di utilizzo e viene fatta rispettare. Si tiene perciò conto, per ogni processo, del rapporto del tempo di utilizzo effettivo della *CPU* e del tempo che gli è stato "promesso" dallo scheduler. I processi per cui questo rapporto è più basso saranno favoriti ad essere scelti dallo scheduler, alla prossima assegnazione.

#### *Scheduling a lotteria:*

La misura in cui ogni processo utilizzerà la *CPU* si baserà sulla quantità di ticket posseduti dallo stesso. Un processo a più alta priorità possiederà un più alto numero di ticket.

#### *Scheduling fair-share:*

Realizza un equo uso tra gli utenti del sistema, piuttosto che tra i processi.

### Scheduling su sistemi multiprocessore:

Vi sono due possibili approcci:

* **Multielaborazione asimmetrica**:  
  Una delle N CPU assume il ruolo di *Master*, mentre le altre di *Slave*.
* La CPU Master, su cui girerà il *kernel* e le sue routine, avrà il compito di coordinare e gestire i processi.
* Le CPU Slave eseguiranno processi solo in modalità utente.

In questo scenario non si possono creare *race-conditions* sulle strutture dati del *kernel*, ma si "spreca" una *CPU*. Inoltre il *Master* potrebbe diventare un collo di bottiglia se vi sono parecchie *Slave* da gestire.

* **Multielaborazione simmetrica** (*SMP*):  
  Tutte le CPU sono uguali e si possono occupare di processi *kernel* e processi *utente*.  
  Può essere implementata tramite **coda unificata** dei processi pronti o **code separate** per ogni *CPU*.
* **Coda unificata**, dato che più *CPU* modificheranno questa coda devo gestire le *race-conditions*. Il sistema è poco scalabile.
* **Code separate**, è più scalabile perché non c'è concorrenza nell'accesso a tale struttura dati.

Il problema dei sistemi multiprocessore è che un processo potrà essere computato in parte da una *CPU* e in altre parti da altre CPU.

##### Perché è vantaggioso che ogni processo venga computato per intero dallo stesso processore?

Il vantaggio è legato al sistema delle cache delle *CPU*:

Supponiamo che nella *CPU-1* un processo **X** si auto-sospenda o che venga prelazionato, avverrà il *context-switch* (con relativi aggiornamenti dei registri) con il processo **Y**.

La cache della *CPU-1* potrà: o essere azzerata (male) oppure no (meglio).

Il processo **Y** verificherà se la cache della *CPU-1* contiene parte dei dati utili alla sua computazione:

* Se non sono presenti allora farà un certo numero di **cache-miss**, che gli permettono di caricare i dati necessari dalla *Memoria Centrale* alla cache della *CPU-1*.
* Se sono presenti allora si farà un certo numero di cache-hit, guadagnando sulla velocità di esecuzione.

Se nel frattempo il processo **X** di partenza viene pescato dalla CPU-3 allora le speranze del processo **X**-**Y**-**…** di avere già dati nella cache della CPU-N sono scarsissime.

La predilezione di un processo per un dato processore è la politica che gestisce questo fenomeno.

* **Predilezione debole**: si cerca di eseguire un processo sempre sullo stesso processore ma, alla necessità (bilanciamento del carico) cambiare questa affiliazione.
* **Predilezione forte**: i processi hanno la possibilità di assegnare se stessi o dei suoi pari, tramite apposite chiamate di sistema, un vincolo preciso tra una CPU e un processo.

E' desiderabile che in generale tutti i processori siano utilizzati più o meno in equa misura, questo bilanciamento del carico va in contrapposizione alla predilezione. Si deve raggiungere un compromesso tra le due…

Questo compromesso si ottiene tramite un meccanismo di **Migrazione** del PCB del processo da una coda ad un'altra. (I due tipi di migrazione possono essere usati assieme)

* **Migrazione guidata**: è una routine interna al kernel che periodicamente si attiva e controlla lo stato di bilanciamento delle varie code. Se viene osservato un certo sbilanciamento viene innescata una migrazione da una coda ad un'altra per uno o più processi.
* **Migrazione spontanea**: si attiva nel momento in cui una CPU si ritrova con una coda dei processi pronti vuota. Anziché andare in *idle*, la CPU attiva una routine di controllo delle altre code ed andrà a "rubare" alcuni processi dalle CPU più occupate.

E' palese che la **migrazione** può causare delle *race-conditions*, dato che il kernel (Migrazione Guidata) o le altre CPU (Migrazione Spontanea) possono leggere e scrivere sulle code di altre CPU in concorrenza al loro utilizzo da parte della CPU proprietaria. Pertanto è necessario prevedere mutua esclusione anche nei sistemi con code separate.

## Scheduler di Windows 8 (Slide 47)

E' un sistema che fa prelazione, basato sul concetto di *Thread* e sulle classi di priorità.

Le API Win32 prevedono le chiamate di sistema:

* **SetPriorityClass**: imposta la classe per l'intero processo (gruppo di *Thread*).
* classi di prioritå 
  time-critical 
  highest 
  above normal 
  normal 
  below normal 
  lowest 
  idle 
  real- 
  time 
  31 
  26 
  25 
  24 
  23 
  22 
  16 
  high 
  15 
  15 
  14 
  13 
  12 
  11 
  above 
  normal 
  15 
  12 
  10 
  normal 
  15 
  10 
  below 
  normal 
  15 
  idle 
  priority 
  15 **SetThreadPriority**: imposta la priorità relativa per un singolo *Thread*.

Viene utilizzato, verticalmente, l'algoritmo di selezione per priorità più alta e, orizzontalmente, il Round-Robin.

* Vi è un apriorità di base (*normal*).
* Euristiche per sistemare i problemi:
* cambio di priorità sulla base dell'utilizzo del quanto di tempo (si privilegiano i processi I/O bounded).
* I processi I/O della tastiera, del mouse e del disco hanno priorità ancora più alta.
* I Thread del processo in primo piano (in focus dall'utente) vengono privilegiati incrementando il quanto di tempo a loro disposizione.
* Il problema dell' **Inversione di priorità** viene risolto tramite la tecnica dell' **AutoBoost**:
* Si cercano tutte le dipendenze tra processi ad alta priorità e processi a più bassa priorità che collaborano su una struttura dati condivisa e si incrementa la priorità dei processi più bassi.

## Scheduling sui sistemi Linux

I sistemi Linux sono basati sul concetto di ***Task***, ovvero una visione del *Thread* (quindi un flusso di esecuzione) che dispone di una serie di risorse che possono essere condivise o meno con altri *Task*.

Grazie a questa ulteriore astrazione è stata introdotta una chiamata di sistema: **clone**, che permette una creazione più granulare dei vari task. Infatti la chiamata clone, tramite degli *sharing flag*, permette di decidere quali strutture dati condividere con il task chiamante.

Task di uno stesso processo avranno lo stesso **PID**, essi saranno identificati tramite il **TID** (*Task Identifier*).

##### Algoritmi di Scheduling:

I sistemi Linux si basano principalmente su due algoritmi di scheduling, inizialmente si usava **scheduler O(1)** che è poi stato rimpiazzato con **CFS** (*Completely Fair Scheduler*)

I sistemi Unix in generale gestiscono 3 classi di *Thread*:

* **real-time FIFO** (non-preemptive)
* **real-time Round-Robin** (preemptive)
* **timesharing** (preemptive) {Approccio usato nei SO Linux}

###### *Priorità numeriche:*

* **real-time** da 0 (altissima) a 99.
* **timesharing** da 100 a 139 (bassissima).

Abbiamo un meccanismo che permette di avere una priorità di base, che tipicamente è ereditata dal *thread* padre, che può essere influenzata da due meccanismi:

* **variazione statica**:  
  Innescata da una chiamata di sistema **nice** che permette di variare la priorità di un processo da -20 a +19.  
  Per evitare l'abuso, i processi utente possono chiamarla solo con valori positivi (peggiorando la priorità). Solo l'amministratore può migliorare la classe di priorità di un processo.
* **variazione dinamica**: migliora o peggiora (+5 a -5) la priorità in base al tipo di comportamento (*I/O bounded* piuttosto che *CPU bounded)*

### Scheduler O(1)

Algoritmo di scheduling per code di priorità, può funzionare su sistemi multicore dove ogni core ha una propria **run-queue** (di fatto una coda dei processi pronti) che presenta due code: la **active queue** e la **expired queue**.

#### *Active Queue e Expired Queue:*

La *active queue* è un array di 140 classi di priorità a cui a ciascuno di essi è associata una coda contenente il *PCB* dei vari processi.

I quanti di tempo delle classi di priorità più alte (peggiori, *CPU bounded*) saranno più lunghi dei quanti di tempo assegnati alle priorità più basse (migliori, *I/O bounded*).

##### Questo concetto teorico è opposto a quello visto fino ad ora, perché?

La differenza sta nel fatto che i task che si bloccano prima di aver consumato il proprio quanto di tempo, quando ritorneranno nella coda dei processi pronti, avranno assegnato il quanto residuo e non uno nuovo.

Flags 
Static_prÉ 
Expir ed 
prjonty O 
ArraylOl 
Array[ll Quando invece il quanto viene effettivamente esaurito allora il processo viene messo **coda expired**, con la stessa classe di priorità (ma anche no).

Quando la coda active sarà vuota (quindi tutti i processi hanno avuto la possibilità di usare il proprio quanto di tempo) allora vi sarà uno switch tra i riferimenti della coda active e la coda expired, di fatto invertendo l'una con l'altra.

Non ho problemi di **starvation** e la complessità dell'algoritmo di selezione è O(1)!

### Scheduler CFS (Completely Fair Scheduler)

Questo algoritmo implementa l'idea dello **scheduling garantito**, ossia di dare in una misura equa una porzione della risorsa CPU ad ogni processo.

L'idea è quella di tenere traccia di quanta *CPU* ogni processo ha utilizzato, attraverso il concetto di **Virtual Runtime**, e concedere di volta in volta la *CPU* al *task* l'ha usata di meno.

Il **Virtual Runtime** è una sorta di contatore associato ad ogni task/processo che conta il tempo (in nanosecondi) per cui quel task ha usato concretamente la CPU (tralasciando quindi i tempi morti, ovvero in cui il task era bloccato). Tramite questa misura viene scelto di volta in volta dello Scheduler il task con il valore di *Virtual Runtime* più piccolo.

47 
10 
30 
27 
più bisogno della CPU 
38 
NI L 
86 
meno bisogno della CPU Per fare questa scelta in modo efficiente si utilizza un **Albero Rosso-Nero**, ossia una struttura auto bilanciante. Questa è una struttura le cui istruzioni sono abbastanza efficienti, e che per sua natura (***BST***) andrà a posizionare il minimo nella foglia più a sinistra ed il massimo nella foglia più a destra.

Se pensiamo che l'albero contenga proprio il valore di *Virtual Runtime* allora i task saranno ordinati per il tempo in cui hanno utilizzato concretamente la *CPU*. Il minimo sarà la scelta che l'algoritmo di selezione andrà a selezionare.

Il processo selezionato viene spostato in *CPU* ed il suo contatore *Virtual Runtime* comincerà a crescere, infatti con una certa periodicità si andrà ad aggiornare il contatore all'interno della struttura dati del *task* che attualmente sta usando la CPU.

Mano a mano che il valore di una foglia di questa struttura incrementa allora questa comincerà a spostarsi verso destra e ad un certo punto potrebbe non essere più la foglia con il valore minimo.

Non appena non sarà più il minimo allora, dopo aver superato anche un gap (**granularità**) di sorpasso, la CPU sarà assegnata al nuovo *task* con l'etichetta minima.

Questa **granularità** (gap) va ad influire sulla reattività del sistema:

* In particolare per un sistema pensato per il desktop si sceglierà una granularità minima -> più *context-switch*.
* Su sistemi batch una granularità maggiore -> meno *context-switch*.

Non ci sono dei quanti di tempo prestabiliti e non è predeterminato per quanto tempo un task userà la CPU. Questo sistema, in modo naturale, gestisce una sorta di equità nel servizio offerto a prescindere dalla natura del task.

In particolare i processi ***I/O Bounded*** saranno naturalmente avvantaggiati, perché bloccandosi relativamente subito il loro *Virtual Runtime* non crescerà molto, quindi quando saranno di nuovo ***ready*** il loro *Virtual Runtime* sarà uno dei minimi presenti nella struttura, perciò saranno prossimi candidati ad essere scelti dall'algoritmo.

Se arriva un nuovo processo gli si assegna un *Virtual Runtime* ragionabile tra quelli presenti (il minimo, la media, il massimo, …).

In generale i sistemi **CFS** non usano priorità per cercare di bilanciare il tutto, però si ha comunque la possibilità di stabilire le priorità statiche dei processi con l'uso della chiamata **nice**. Questa chiamata di sistema è implementata usando i **Fattori di Decadenza**, ovvero dei numeri in stretta relazione con le priorità che vengono tirate in gioco nel momento in cui bisogna aggiornare il *Virtual Runtime*.

Il *Fattore di Decadenza* è infatti un fattore moltiplicativo che viene applicato al valore che si va ad addizionare al *Virtual Runtime*, influendo quindi sulla posizione del *task* nell'albero.

Questo algoritmo implementa anche il concetto del *fair-share*, ossia equità nei confronti degli utenti, attraverso delle schedulazioni basate su gruppi di processi.

# Gestione della Memoria

## Memoria centrale (RAM) e processi utente

#### Typical access time 1 nsec 2 nsec 10 nsec 10 msec 100 sec Registers Cache Main memory Magnetic disk Magnetic tape Typical capacity KB 512-2048 MB 200-1000 GB 400-800 GB *Gerarchia di memoria:*

La memoria centrale (formata da **RAM**) rappresenta il più basso livello direttamente utilizzabile dalla CPU.

Vedremo utilizzare la memoria centrale inizialmente senza alcuna astrazione e via via aggiungendo astrazioni sempre più complesse…

#### *Senza alcuna astrazione (sistema a indirizzi fisici):*

* Modello usato negli anni 60-80.
* I programmi utilizzano direttamente gli **indirizzi fisici**.
* Difficile eseguire due programmi contemporaneamente.

La memoria centrale veniva semplicemente divisa tra il processo utente (shell) e il processo del sistema operativo (molto basilare).

L'utente poteva leggere e scrivere su ogni indirizzo fisico della memoria, non c'era alcun tipo di protezione del SO.

##### E' possibile mantenere più processi in memoria in questo sistema?

* **Problema della rilocazione**: ovvero di stabilire come gestire gli indirizzi fisici ad opera di diversi processi che coesistono in memoria.  
  Cioè il compilatore in fase di creazione del codice dei processi deve decidere gli indirizzi fisici che utilizzeranno e in molti casi questi indirizzi non li può stabilire a priori.
* Soluzione **Rilocazione a compile-time**: ovvero rilocare il codice a lavorare nelle aree di memoria informandolo a tempo di compilazione (crea un codice molto rigido e controverso).
* Soluzione **Rilocazione Statica**: ovvero un aggiustamento degli indirizzi di memoria presenti nel codice del programma che viene effettuato in fase di *loading* (fase di lancio del programma).
* Effetto: Rallentamento del *loader*.
* **Problema della protezione della memoria**: chi vieta ai programmi di creare problemi ad altri processi o al sistema operativo? Nulla.
* Soluzione **lock & key**: suddividere la memoria blocchi di memoria con delle chiavi di protezione e *PSW* con la chiave del processo in esecuzione.

#### *Spazio degli indirizzi:*

Il sistema era basato su due registri presenti all'interno della CPU, chiamati rispettivamente **registro base** e **registro limite**.

Era implementata la tecnica della **Rilocazione Dinamica**, in quanto implementava la tecnica di Rilocazione Statica, ma a *Runtime*, cioè l'aggiustamento del codice veniva fatto in corsa dalla **MMU**.

L'idea è che ogni volta che un particolare processo all'interno di una partizione veniva caricato nella CPU, l'indirizzo a partire del quale si sviluppava la partizione del processo veniva caricata all'interno del registro base.

Il registro limite invece veniva caricato con il valore corrispondente all'ampiezza in byte della partizione stessa.

Con questi due registri, che in fase di *context-switch* venivano caricati all'interno della *MMU*, si implementava nella stessa fase sia l'obbiettivo raggiunto dalla Rilocazione sia la Protezione. In quanto l'indirizzo che proviene dalla CPU (Indirizzo Logico) viene passato alla MMU, questa effettua una duplice azione:

1. **Azione di controllo**: confrontando l'*indirizzo logico* con il valore contenuto nel registro limite (se il programma tenta di accedere ad una locazione al di fuori della propria allora verrà terminato il processo).
2. **Azione di aggiustamento**: somma il valore del registro base all'indirizzo stesso, in corsa. (Shift di tutti gli indirizzi)

Se il sistema prevedeva la separazione tra modalità *utente* e modalità *kernel*, allora sarebbe stato parecchio robusto, dato che il processo utente non avrebbe potuto modificare i due registri.

#### *Swapping:*

Il livello di multiprogrammazione (la capacità di introdurre livelli di astrazione per eseguire più programmi in contemporanea) è seriamente limitato dalla dimensione della memoria centrale.

Lo *Swapping* è un processo in esecuzione, in cui un app in corsa poteva essere parcheggiata sulla memoria secondaria (**disco**) se le esigenze lo richiedevano, ovvero se la memoria centrale risultava satura e l'utente voleva lanciare una ulteriore app, allora In questo modo si liberava spazio dalla memoria centrale per soddisfare l'utente.

Lo spazio di indirizzamento veniva copiato quindi sul disco, in un'area chiamata di ***swap***, ovvero uno spazio del disco in cui vengono parcheggiate per interno le informazioni relative ad un processo che fino all'istante prima era in esecuzione in memoria centrale e che adesso è invece nello stato *blocked*.

Questo processo sfortunato, che viene bloccato, viene scelto tramite un apposito algoritmo chiamato ***swapper*** che aveva il compito di selezionare il processo di sacrificare. La selezione avveniva sulla base della natura del processo, tipicamente venivano scelti servizi che non interagivano con l'utente.

##### Come viene gestita lo swap?

Questo spazio di memoria viene detto **contiguo**, cioè la memoria allocata ad un processo parte da un indirizzo e si sviluppa verso un indirizzo limite. La *contiguità* presenta delle problematiche (**frammentazione interna** e **esterna**), che vedremo a breve…

Inoltre la partizione assegnata ad un nuovo processo può essere **fissa** o **dinamica**:

* *Fissa*: modello molto limitato perché si deve quantificare a priori la dimensione massima di ogni blocco.
* *Dinamica*: un processo può richiedere ulteriore memoria in maniera dinamica, questo processo potrebbe però avere lo spazio di allocazione sopra e sotto di lui già occupato, quindi non gli si potrà accettare la richiesta. Questo è il problema principale della contiguità…
* *Non-Soluzione*: allora potrei allocare un processo con più memoria di quella richiesta? potrebbe non essere abbastanza o comunque sarebbe uno spreco (**frammentazione interna**).
* La **frammentazione esterna** consiste invece nel fatto che si possono venire a creare dei buchi di memoria non allocata frammentati, ognuno di pochi Kb.
* aperating 
  Cperatng *Soluzione*: per sistemare questo problema viene usato, ad esempio, la *memory compaction*, che va a compattare lo spazio non allocato spostando (processo dispendioso) i vari blocchi e unendo quelli non allocati.

#### *Gestione della allocazione:*

Come fa un sistema operativo a tenere traccia di quale è lo stato della memoria centrale, ovvero di rendersi conto di dove poter allocare un ulteriore processo?

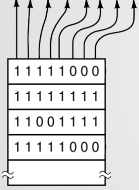
A tal fine si usano due metodologie: *Bitmap* e *Liste Dinamiche*;

##### Bitmap:

Usare una bitmap di dimensioni fisse, ovvero un vettore di bit caratterizzato da una allocazione contigua. Di rado può crescere dalla coda, se possibile.

L'idea è di mantenere una bitmap in cui ho tanti bit quanti sono i blocchi di memoria allocabili in memoria:



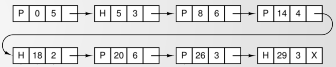


##### Liste Dinamiche:

Usare liste dinamiche, attraverso dei blocchi che non hanno vincoli sulla contiguità. (Una lista è per natura *non-contigua*)

L'idea è che, rispettivamente ogni nodo rappresenta o una partizione assegnata ad un processo (*P*) o ad un buco vuoto (*H*).

Come informazione satellite avrò l'indirizzo del nodo iniziale della lista e la sua estensione.

Questa lista viene dinamicamente manipolata alla allocazione e disallocazione di un processo.  
Quando deve essere allocato un nuovo processo allora si usano vari algoritmi di ricerca per buchi sufficientemente grandi per le necessità del nuovo processo.

#### *Memoria virtuale:*

Questo concetto è utilizzato nei sistemi contemporanei e si basa su un meccanismo di astrazione che fornisce uno spazio di memoria virtuale con le seguenti caratteristiche:

* Ogni singolo processo dispone di un proprio spazio di indirizzamento in una memoria virtuale, abbondante (più grande della memoria fisica; nei sistemi a 32bit circa 4gb, nei 64bit 2^64 byte).  
  L'implementazione avviene tramite un meccanismo basato sulla ***paginazione***:
* Di base si osserva che un processo in esecuzione attraversa varie fasi di vita (esecuzione di una porzione di codice rispetto che un'altra), tramite le quali si può osservare il **fenomeno della localizzazione della porzione di codice** in esecuzione dal processo. Ovvero la localizzazione (nello spazio di indirizzamento [nella memoria virtuale]) di quella porzione limitata di codice e dati che il processo necessita in quella determinata fase di vita.  
  L'ideale sarebbe quindi di mantenere in memora centrale solamente quella porzione limitata di codice e dati. Questa implementazione deve però adattarsi alle nuove esigenze per ogni volta che il processo cambia fase (necessitando quindi di altri dati ed altre porzioni di codice da eseguire), procurandosi la porzione di spazio di indirizzamento necessaria, portandola in memoria centrale, permettendo al processo di continuare a lavorare, il tutto in maniera trasparente al processo stesso (cioè lui non si deve accorgere di nulla).  
    
  Tutti gli spazi di indirizzamento di un processo che non stanno in memoria centrale perché non attualmente utili al processo, stanno invece nella memoria secondaria (disco) come in una sorta di *swap*, solo che a differenza di spostare l’intero spazio di indirizzamento di un processo e bloccare il processo stesso (che passerà allo stato *blocked*, penalizzandolo parecchio) in questo nuovo sistema basato su *impaginazione* andiamo a “parcheggiare” sul disco delle porzioni di codice non utili alla fase di vita attuale del processo. Perciò il processo persisterà nello stato di *ready* o di *running*, continuando quindi ad essere potenzialmente eseguibile dalla *CPU*.  
  Quando il processo necessiterà di spazi di indirizzamento parcheggiati nel disco, a causa di un cambio di fase di vita, questo evento prende il nome di *Page Fault*, si occuperà il sistema operativo di gestire il problema, di nascosto.

##### Come funziona la Paginazione?

Questo ampio spazio di memoria virtuale viene gestito tramite un sistema di indirizzi (virtuali):

Per comodità questa memoria virtuale viene suddivisa in pezzetti (**pagine**) tutte di dimensione uguale (dell'ordine di qualche kb, regolabile tramite un parametro), ciascuna delle quali viene etichettata da un numero (*numero di pagina virtuale*).

La paginazione prevede che anche la memoria fisica centrale venga suddivisa in pezzettini (**frame**) anch'esse numerate, tipicamente della stessa dimensione della *pagina*.  
L'idea è quella di ospitare le *pagine* all'interno dei *frame*, ma il vantaggio è che queste *pagine* non hanno il vincolo della contiguità. Ovvero, se debbo mantenere le *pagine* 0, 1, 2, 3 (contigue in memoria virtuale) di un processo X in memoria fisica centrale, non è detto che in essa siano fisicamente contigue.  
Ciò significa che le posso mettere in un qualsiasi *frame* libero della memoria fisica centrale, eliminando il problema della frammentazione esterna.  
Ogni processo ha il proprio spazio di indirizzamento indipendente, cioè ognuno di essi avrà il proprio spazio di indirizzamento virtuale suddiviso in pagine, cioè sia il processo X che Y avranno la propria pagina virtuale 0, 1, … . Inoltre il processo X non ha un modo per riferirsi allo spazio di memoria virtuale del processo Y, applicando una sorta di protezione implicita tra processi. Poi vedremo comunque che ci sarà un modo per far comunicare i processi, in maniera "autorizzata".

##### Come viene gestita l'associazione tra indirizzi virtuali e indirizzi fisici? *Mappatura*

Questo meccanismo mi dice per ogni pagina della memoria virtuale:

* Se sta in memoria centrale oppure no
* Se sta in memoria centrale, dove?

Tramite la mappatura gli indirizzi virtuali generati dalla *CPU* verranno, all'occorrenza, tradotti/mappati sui corretti indirizzi fisici.

###### *Vantaggi e svantaggi della Mappatura:*

***Il fenomeno della frammentazione interna:***

La paginazione alloca memoria virtuale (ma anche fisica) arrotondando tutto in pagine *frame*. Ad esempio, immaginiamo che una struttura dati di *7kb* sia immagazzinata in memoria virtuale con pagine da *4kb* ciascuna, questa allora occuperà due pagine contigue, in cui nella seconda ci sarà *1kb* sprecato.

Se il processo necessita di questa struttura dati, allora queste due pagine verranno portate in memoria centrale e conservate in due frame (non necessariamente contigui ma comunque collegate logicamente da un puntatore) della memoria fisica centrale anch'essi di *4kb*, sprecando realmente *1kb* di spazio (frammentazione interna).

Se uso pagine grandi, questo fenomeno potrebbe accentuarsi,

Se uso pagine più piccole allora in teoria questo si ridurrebbe. Ma usare pagine più piccole presenta dei problemi; ovvero hanno bisogno di strutture di supporto (tabella delle pagine) più grandi perché ci saranno più pagine.

*Curiosità:  
Esistono sistemi operativi minori (Solaris) che fanno uso di pagine di dimensione variabile*

###### *Esempio pratico del processo di Mappatura nella Paginazione:*

Testo alternativo generato dal computer:
Virtual 
address 
space 
60K-64K 
56K-60K 
52K-56K 
48K-52K 
40K_44 K 
36K-40K 
32K-36K 
28K-32K 
28K-32K 
24K-28K 
24K 28K 
20K-24K 
16K 20K 
16K-20K 
12K 16K 
12K-16K 
12K 
x 
x 
x 
x 
7 
5 
x 
x 
3 
4 
0 
6 
2 
Virtual page 
20K 
K_4K 
physical 
memory 
address 
81<-12K 
o K_4K 
Page frame 

Alla sinistra è rappresentato lo spazio di indirizzamento virtuale da *64kb* che si riferisce ad un singolo processo (quindi ne avrò tanti quanti sono i processi), la prima *word* ha indirizzo virtuale *0*, l'ultima *64k*.

Alla destra è rappresentata la memoria fisica di appena *32kb* di memoria. La memoria fisica invece è condivisa per tutti i processi (**MA** in questo esempio semplificato è usata da un solo processo).

Le pagine sono da *4kb*, di conseguenza anche i *frame*. Con *64kb* di memoria ho esattamente 16 pagine virtuali, numerate. La memoria fisica con *32kb* ha invece 8 *frame* numerati.

Lo spazio di indirizzamento virtuale non può stare tutto in memoria fisica, infatti le '**X**' rappresentano quegli spazi di indirizzamento non necessari attualmente al processo e che stanno quindi in memoria secondaria (disco "*swap*").

###### *Come avviene questo raccordo tra gli indirizzi generati dalla CPU e gli indirizzi che effettivamente devo usare per rintracciare le informazioni sparse qua e là nella memoria centrale?*

E' il problema della traduzione da *indirizzo virtuale* a *indirizzo fisico* che, presentandosi frequentemente, viene gestito dall'hardware, in particolare dall'***MMU*** (*Memory Management Unit*).

###### *Quando ha senso effettuare questa traduzione?*

L'esercizio di traduzione da *indirizzo virtuale* a *indirizzo fisico* può funzionare per quelle word del processo presenti in memoria virtuale. Ricordando infatti che per il processo ogni sua porzione di codice è accessibile in qualsiasi momento, allora ha pienamente diritto di richiedere una word che non si trova in memoria virtuale ('**X**').

Questo evento, statisticamente più raro rispetto gli ordinari accessi in cui riesco ad effettuare la traduzione, è dato dal fatto che alcune informazione stanno in memoria centrale e altri no e prende il nome di ***page fault*** (*fault di pagina*).

La gestione di questo evento è deputata al Sistema Operativo. In questi casi infatti, quando l'*MMU* capisce di trovarsi di fronte un fault di pagina, passa il controllo al Sistema Operativo che gestirà il problema individuando la pagina richiesta sul disco (operazione che richiederà una certa quantità di tempo e che bloccherà transitoriamente il processo *[blocked]* fino a quando non verrà risolto il *page fault*, e sarà del tutto trasparente al processo), deve anche individuare un frame libero in memoria centrale in cui inserire la pagina nel disco (se lo trova allora effettuerà una operazione I/O che porterà l'intera pagina contenente la word richiesta dal processo dal disco alla memoria centrale), e aggiornare in fine la mappatura.

A questo punto, una volta risolto il *page fault*, il Sistema Operativo si occuperà di rendere nuovamente disponibile il processo *[ready]*, e la prima cosa che farà il processo quando riprenderà il controllo sarà rieseguire la stessa istruzione che ha generato il *page fault*, cui a questo punto la nuova mappatura renderà possibile la traduzione.

###### *Come può l'MMU capire quando poter effettuare la traduzione o se si trova di fronte ad un page fault?*

A questo scopo l'*MMU* utilizza la ***tabella delle pagine*** che presenta per ogni pagina presenta un **bit di presenza/assenza** in memoria centrale e, nel caso in cui la pagina sia presente mi indica il *numero di frame* che ospita quella data pagina.

###### *Ma come avviene effettivamente la traduzione?*

Supponiamo di avere un *indirizzo virtuale* numericamente pari a 8196.

La prima cosa da fare è prendere questo *indirizzo virtuale* e dividerlo per la dimensione della pagina, ottenendo quindi il *numero di pagina* cui è contenuta (8196 / 4096 = 2,001 [indirizzo virtuale 2]).

Otterremmo 2 con il resto di 4; queste due informazioni hanno un significato ben preciso, il quoziente intero indicherà esattamente il *numero di pagina* che ospita l'*indirizzo virtuale* che è oggetto della traduzione, il resto invece indica l'***offset***… cioè lo spiazzamento (distanza) che c'è tra la ***word*** che il processo ha richiesto e l'inizio della *pagina* che lo contiene (in questo esempio la pagina 2 inizia all'indirizzo 8192 [4096\*2] ovvero 4 byte prima della word richiesta).

A questo punto debbo consultare la tabella delle pagine all'indice calcolato e controllare:

* In primis il *bit di presenza/assenza* e, nel caso in cui è confermata la presenza in memoria centrale, vedere in quale *numero di frame* è mappata.

A questo punto devo calcolare l'indirizzo fisico della *word* nel *frame* (perché io conosco dove si trova in memoria virtuale ovvero a pagina 2 all'indirizzo virtuale 8196, ma non in memoria fisica).

Per calcolare l'*indirizzo fisico* della *word*, acquisito il *numero di frame*, moltiplico quest'ultimo per la dimensione del frame (6 \* 4096 = 24.576) che corrisponde all'*indirizzo fisico* in cui comincia il *frame* che ospita la *word* richiesta. Infine per ottenere l'indirizzo dell'inizio della *word* basta sommare l'*offset* ottenuto in precedenza all'inizio dell'*indirizzo del frame* (24576 + 4 = 24.580).

**IMPORTANTE:**

Ogni processo ha il proprio *spazio di indirizzamento virtuale*, quindi ogni spazio di indirizzamento virtuale di ogni processo viene mappato parzialmente sull'unica memoria fisica, quindi per fare questa mappatura, che è del tutto indipendente tra un processo e l'altro, ho bisogno di una tabella delle pagine per ogni processo.

###### *Uso di una tabella delle pagine:*

Se consideriamo che gli indirizzi di memoria sono rappresentati in binario, dove ogni indirizzo è una potenza del 2, possiamo notare come tutte le operazioni di divisione e moltiplicazione per la traduzione tra *indirizzi virtuali* e *fisici* risultino ingegnerizzate e semplici per un calcolatore.

In particolare si noti appunto che se devo dividere per 4096, ovvero 2^12, posso farlo semplicemente shiftando verso destra di 12 posizioni l'*indirizzo virtuale* della *word* (0010000000000100 >> 12 <==> 8196 / 4096) analogamente per la moltiplicazione.

Questo è il resto di 4 byte (offset)

Questo è il risultato dello shift, ovvero il numero di pagina (2)

Tutto il numero binario è invece l'indirizzo virtuale della word

***Esempio di domanda di esame***

*Dati:*

* *Spazio Virtuale: 4 MB.*
* *Tabella pagine con 2^13 voci.*
* *Numero di frame a 8 bit.*

*Domanda: di quanta memoria fisica dispongo? ==> 2^17 byte ovvero 128 KB.*

*Risposta:*

*Osserviamo che 4 MB = 2^22 byte.*

*Calcoliamo la grandezza di una pagina virtuale… dividiamo quindi la dimensione dello spazio di indirizzamento virtuale per il numero di pagine: 2^22 / 2^13 = 2^9 byte sarà la dimensione di una pagina e di conseguenza anche del frame.*

*La memoria fisica è formata da tanti frame, ma quanti frame?*

*Posso risalire a questa informazione conoscendo da quanti bit è composto un numero di frame… un numero di frame è composto da 8 bit quindi esisteranno 2^8 frame (256 frame), se ogni frame è da 2^9 byte io ho 2^8 \* 2^9 = 2^17 byte ovvero 128 KB.*

###### *Struttura della tabella delle pagine:*

Nei sistemi reali, con numeri decisamente più grandi, come questa traduzione è resa comunque gestibile? Ovvero tenendo conto:

* **Dell'efficienza nella traduzione:**  
  Il fatto che la traduzione sia eseguita dall'hardware non significa massime prestazioni, infatti prima o poi l'*MMU* dovrà consultare la *tabella delle pagine* che si trova in *memoria centrale*, in base a come strutturiamo la tabella vedremo che dovremo fare o un solo accesso o più accessi alla *memoria centrale*. In generale più accessi impattano sull'efficienza.
* **Spazio occupato:**  
  La dimensione della tabella cresce al crescere del numero delle pagine, che cresce al diminuire della dimensione di ogni pagina. Considerando che vi è una tabella per ogni processo si capisce bene che avere in *memoria centrale* tabelle troppo grandi può essere un problema.

###### Caching disabled Modified Referenced Present/absent Page frame number Protection *La generica voce della tabella delle pagine presenta:*

Oltre il già visto *bit di presenza/assenza* ed il numero di frame la tabella delle pagine presenta altre informazioni:

* Un campo ***maschera di permessi*** (*campo protezione*) composta da 2 o 3 bit in cui ognuno indica un permesso relativo a quella pagine (*lettura, scrittura, esecuzione*).  
  Perché porre dei vincoli su porzioni che appartengono a un processo?  
  Questo porre dei limiti è collegato ad alcune strategie legate alla robustezza del sistema e alla sicurezza.  
  In generale si ha il permesso di *lettura/scrittura* nelle porzioni relative alle strutture dati e sola *lettura* nelle porzioni in cui è presente il codice.  
  Questo meccanismo permette anche di risparmiare *memoria centrale* ad esempio nel caso in cui vengano eseguite più istanze di uno stesso processo, tutte queste istanze si rifaranno ad una sola pagina in *lettura* di una istanza contenente il codice piuttosto che conservare il codice in memoria per ogni processo.  
    
  Questo controllo viene fatto in modo efficiente dall'MMU che, ancora prima di controllare il *bit di presenza/assenza* controlla se il tipo di operazione richiesta dal processo è compatibile con i permessi relativi alla pagina richiesta.  
  Il diritto di *esecuzione* si riferisce all'operazione di *fetch* del ciclo di base ***fetch-decode-execute***, ed è formalmente una lettura motivata dal prelevare una *word* ci si aspetta contenga del codice.  
  Perché entrare nel dettaglio e specificare che una pagina è leggibile allo scopo di essere eseguita?  
  E' un modello motivato da ragioni di sicurezza, legato al fatto che alcuni attacchi a software mal scritto prevedono di eseguire un injection di codice passato come dati e, sfruttando degli scarsi controlli dati dalla bassa qualità del codice, si può indurre la CPU, tramite buffer overflow, a saltare in termini di *Program Counter*, a porzioni di codice malevolo.
* Il ***bit di referenziamento*** è un singolo bit gestito dall'hardware che serve per tener traccia dell'utilizzo di una pagina in un certo lasso di tempo.  
  Ogni volta che l'*MMU* fa un accesso ad una pagina va ad aggiornare questo bit sulla pagina, che passerà ad 1 alla prima visita e rimarrà tale per le altre visite.   
  Questo bit ritornerà a 0 ad opera del Sistema Operativo, che si occuperà di azzerare il bit di referenziamento di ogni processo ad una data frequenza.  
  Questo bit serve per capire se la pagina "è importante" per il processo, infatti, prima di ogni azzeramento dispongo infatti di una statistica a brevissimo termine che mi dice quali sono le pagine usate o meno dal processo e questa informazione servirà più avanti per effettuare determinate scelte.
* Il ***bit di modifica*** (*dirty bit*) indica se una pagina, da quando è stata portata in memoria centrale, è stata o meno modificata.  
  Quando avviene un *page fault* si ha che una porzione di dati viene copiata dalla *memoria secondaria* alla *memoria centrale* (dal *frame* viene copiata nella *pagina*) e all'istante di caricamento in *memoria centrale* le due copie coincidono.  
  Per tenere traccia se in futuro coincidano o meno viene usato questo bit, che viene gestito dall'hardware.  
  L'*MMU* infatti, per le operazioni di lettura non considera neanche questo bit, mentre per le operazioni di scrittura viene posto ad 1 e saprò che la copia della *pagina* che sta in memoria centrale è potenzialmente differente dalla copia che sta sul disco.  
  E' importante perché si potrà avere l'esigenza di spostare le pagine dalla *memoria centrale* al disco, però se io ho traccia se una pagina è "sporca" o meno allora posso rendere più leggera l'operazione di scrittura sul disco.
* Il ***caching bit***, che segnala all'hardware se applicare o meno le ottimizzazioni legate alla cache della CPU a quella pagina.  
  Un motivo per cui voler disabilitare tale ottimizzazione può essere ad esempio il caso di accesso alla mappatura delle porte *input/output*, in cui potenzialmente un tentativo di lettura di queste porte potrebbe non vedere mai il risultato.
* Il ***bit di validità*** (o di *allocazione*) è utilizzato per amministrare l'immenso spazio di indirizzamento.  
  Nello specifico serve per tener traccia di quegli spazi di indirizzamento allocati dinamicamente che sono stati concessi al processo (memoria dinamica *Heap*).

PARENTESI

###### *Tabella dei Frame:*

La tabella dei Frame è una tabella globale, ovvero una tabella usata da tutti i processi.

A cosa serve? fornisce delle informazioni sui frame, nello specifico contiene un record per ogni frame con informazioni elementari del tipo se è occupato o libero e se occupato da quale processo.

Viene consultata e aggiornata:

1. Ogni volta che viene creato un nuovo processo, allocando una certa quantità di frame.
2. Quando il processo chiede di allocare nuove pagine.
3. Quando si liberano risorse allocate.

FINE PARENTESI

--> Tornando alla tabella delle pagine…

###### *Progettazione di una tabella della pagine:*

Gli aspetti principali da curare sono:

###### *Memoria Associativa:*

* **La velocità nell'accesso**, poiché questa tabella verrà consultata ad ogni accesso alla memoria centrale per effettuare la traduzione.

Una prima strategia per non creare un collo di bottiglia è stata mantenere la tabella delle pagine all'interno della stessa *MMU* attraverso un vettore di registri interni alla *MMU* sufficienti a contenere l'intera tabella.

Il *SO* in caso di *context-switch* avrebbe dovuto riprogrammare l'*MMU* riempiendo questa tabella con le informazioni della tabella delle pagine del processo che sta per entrare.

L'approccio che si usa oggi invece mantiene stabilmente la tabella delle pagine all'interno della memoria centrale e si farà riferimento ad essa solo se necessario. L'*MMU* accederà infatti alla memoria centrale facendo un ulteriore *fetch*, addizionale a quello che ha scatenato la traduzione, per consultare la tabella delle pagine.

L'indirizzo fisico dalla quale si sviluppa la tabella delle pagine viene conservato in un registro all'interno della *CPU* chiamato ***PTBR*** (*Page-table base register*) e questa informazione verrà usata per calcolare l'indirizzo fisico della word richiesta.

Un aiuto per evitare di effettuare un *fetch* addizionale alla memoria centrale, ad ogni richiesta di traduzione, viene direttamente dall'hardware che cerca di mitigare questo *overhead* attraverso il meccanismo della ***memoria associativa*** (*translation lookaside buffer* o **TLB**) che possiamo considerare come una sorta di *cache* all'interno della *MMU* stessa.  
La *TLB* attraverso un evento chiamato ***TLB-hit*** permetterà di evitare di dover accedere alla tabella delle pagine, l'idea è che se l'*MMU* trova le informazioni che gli servono all'interno della *TLB* allora potrà evitare di accedere alla memoria centrale e prelevarle direttamente da lì. Se invece non le trova (***TLB-miss***) dovrà accedere alla tabella delle pagine, in memoria centrale.

La *TLB* cercherà di mantenere all'interno dei suoi registri le voci della tabella delle pagine utilizzate più di frequente dal processo stesso.

La struttura di una generica voce della *TLB* è molto simile alla generica voce della tabella delle pagine.

Se la richiesta di accesso alla pagina che si trova nella *TLB* (*TLB-hit*) è di scrittura, e quindi si andrà a modificare la pagina contenuta nella *TLB* piuttosto che la stessa contenuta in memoria centrale, allora il *dirty-bit* di quella voce nella *TLB* verrà posto ad 1, per indicare che prima di azzerare quella voce dalla TLB sarà necessario aggiornare quello stesso dato in memoria centrale con quello più aggiornato.

Cosa succede quando avviene un *context-switch*, passando quindi a lavorare su un'altra tabella delle pagine?

Poiché ogni processo ha il proprio spazio di indirizzamento si avrà che il numero di pagina virtuale non è una informazione univoca, ogni processo avrà il suo numero di pagina 2 ad esempio.  
  
  
  
  
Per evitare di violare lo spazio di indirizzamento di altri processi con *TLB-hit* fasulli sarebbe necessario azzerare la *TLB*, ma effettuare questo processo ad ogni *context-switch* azzererà ogni possibilità che un processo che torni ad essere schedulato trovi voci che lo riguardano al suo interno. L'ideale sarebbe che la *TLB* non si debba azzerare e sperare che magari delle voci restino a disposizione, utili di nuovo quando poi quel processo tornerà ad utilizzare la *CPU*.

Affinché ciò sia possibile è necessario accoppiare la chiave di ricerca "numero di pagina" con un'altra informazione che la renda univoca, si utilizza a tal fine un identificativo del processo.

Ogni voce della *TLB* manterrà quindi oltre al numero di pagina virtuale anche un identificativo del processo (ad esempio l'*address-space identifiers* [***ASID***]). A questo punto, disponendo di due chiavi di ricerca (una che riconosca il processo, l'altra la pagina della word richiesta) non è più necessario azzerare la *TLB* in caso di *context-switch*.

In caso di *TLB-miss*, se la *TLB* è piena bisognerà capire quale voce scartare.

Il sistema operativo può, in alcuni casi, vincolare alcune voci della *TLB* affinché queste non vengano mai eliminate dalla *TLB*, ad esempio quando ha bisogno di accedere in modo efficiente ad alcune strutture interne.

* **Le dimensioni, la sua struttura ed il costo di accesso**.

###### *Tabella delle pagine Multilivello:*

Nei sistemi reali a 32bit, la dimensione della tabella delle pagine diventa una problematica perché supponendo di usare pagine da 4KB avrò 1 milione di pagine (2^20), per un totale di 4MB.

Ogni tabella corrisponde ad un segmento di memoria fisica continua in memoria centrale. Il problema sta nel fatto che anche se la singola tabella occupa relativamente poco, questa deve essere contigua e ne avrò tante quante sono i processi. Ancora peggio, nel caso di sistemi a 64bit ancora si parlerebbe di 2^52 pagine da 4KB per ogni processo.

Da questo punto di vista siamo costretti ad adottare strategie alternative, in particolare la prima strategia è la cosiddetta **Tabella Multilivello** (*Approccio gerarchico*). L'idea è di utilizzare non una tabella lineare ma una struttura più simile ad un albero il cui scopo è limitare il problema dell'allocazione continua e cercare di ridurre lo spazio necessario per memorizzare le informazioni della *tabella delle pagine*.

L'idea viene realizzata suddividendo -nel caso di sistemi a 32bit- i 20bit del numero di pagina virtuale, ad esempio, in 2^10 gruppi (1024) da 2^10 voci (1024) ciascuno. Ottengo quindi un albero in cui la tabella radice viene chiamata ***Tabella di primo livello*** e le varie foglie saranno le ***Tabelle di secondo livello***, in particolare queste ultime riguarderanno un gruppo di pagine.  
La *tabella di primo livello* servirà come indice di ricerca per le *tabelle di secondo livello*, ogni sua voce conterrà infatti l'indirizzo di sviluppo di ogni gruppo, ad esempio l'indice 2 della *tabella di primo livello* conterrà l'indirizza fisico da cui si sviluppa la tabella di gruppo 2, ecc…

Le *tabelle di secondo livello* conterranno invece le informazioni per rintraccia ogni pagina in memoria centrale (numero di frame, bit di validità, bit protezione, ecc…)

Il processo di traduzione funzionerà in questo modo:

Il numero a 20bit viene diviso in due da 10bit in cui:

* i 10bit più significativi -contrassegnati nell'immagine dalla sigla PT1- indicano l'indice nella *tabella di primo livello*.
* PTI 
  10 
  p T2 
  Second-levell 
  page tables 
  Top-level 
  page table 
  1023 
  6 
  5 
  4 
  3 
  2 
  1 
  1023 
  6 
  5 
  4 
  3 
  2 
  1 
  table for 
  the top 
  4M of 
  memory 
  Too 
  pages i 10bit meno significativi -contrassegnati nell'immagine dalla sigla PT2- danno invece l'offset da sommare all'indirizzo della tabella per ottenere la pagina richiesta.

Questa strategia a multilivello, nel nostro esempio a 2 livelli, appesantisce l'operazione di traduzione, che infatti richiederà più accessi alla memoria centrale, più sono i livelli più saranno gli accessi necessari:

Il primo accesso alla memoria centrale avviene per ottenere l'indirizzo in memoria fisica della posizione da cui si sviluppa la *tabella di primo livello* (radice). Questa informazione è contenuta all'interno del registro *Page-Table Base Register*.  
A partire da questo si utilizza l'indirizzo fisico del nodo radice, combinato con PT1 (moltiplicato) per ottenere l'indirizzo da cui si sviluppa il gruppo che contiene la word richiesta, ovvero l'indirizzo della *tabella di secondo livello* interessata e sommando PT2 (*offset*) per ottenere la posizione della pagina all'interno del gruppo.

Il secondo accesso avviene proprio con la lettura di tale pagina, contenente le informazioni che permettono di tradurre l'indirizzo virtuale della word nel suo indirizzo fisico.

Il terzo accesso consiste invece nella lettura della word richiesta dalla CPU, prelevare il dato.

Da un lato si ha un peggioramento delle prestazioni, ma il vantaggio di un tale approccio sta nel fatto che; osservando il fatto che un processo tipicamente utilizza solo una porzione del proprio spazio di indirizzamento, porzione con un pattern molto specifico: ([Stato interno di un processo](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={1C876E5E-EE13-048A-0D3D-C7E7920C85C7}&D1&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one), Capitolo sui Processi, pagina 5)

|  |  |  |
| --- | --- | --- |
| Lo stato interno di un processo è composto da:   * Spazio degli indirizzi, che a sua volta è composto da: * Text, contiene il codice binario da eseguire. * Dati, strutture dati statiche come variabili. * Heap, contiene allocazioni dinamiche di oggetti, vettori o nodi. * Stack, contiene le chiamate a funzioni e le variabili al loro interno.   Heap e Stack crescono e si riducono dinamicamente. |  | Top-level  page table  1023  6  5  4  3  2 |

Si può notare come in particolare le aree concretamente utilizzate dal processo siano la *text*, la *data* e lo *stack*, mentre l'*heap* può essere potenzialmente vuoto. Possiamo quindi immaginare che il che la *tabella di primo livello* venga intelligentemente riempita in modo che il gruppo 0 copra il codice della sezione *text*, il gruppo 1 la sezione *data* ed il gruppo 1023 lo *stack*.

Avremo quindi che il sistema operativo non dovrà tenere in memoria le tabelle di secondo livello relative ai gruppi corrispondenti all'*heap* perché sostanzialmente vuoti. Ogni volta che il processo richiederà di allocare nuovo spazio dinamicamente, questo non farà altro che reclamare una nuova tabella di secondo livello nella "sezione" dell*'heap.*

E' chiaro che basterebbe che scatti una sola voce nel nuovo gruppo che dovrò però allocare l'intera *tabella di secondo livello*, per limitare lo spreco di memoria derivante da questo tipo di evento è possibile suddividere diversamente i 20bit del *numero di pagina virtuale* (o anche diminuire la dimensione delle pagine). Ad esempio si può dimezzare dedicando 11bit per la *tabella di primo livello* che andrà quindi ad identificare 2048 gruppi, e 9bit per le *tabelle di secondo livello* che saranno perciò composte da 512 pagine.

Usare una struttura gerarchica permette di risparmiare memoria dato che le *tabelle di secondo livello* relative ai gruppi vuoti (non utilizzati dal processo) non sono allocati in memoria centrale.

Nei sistemi a 64bit, usando pagine da 4KB, avrei un numero di pagina virtuale di 2^52. Per gestire questa enormità di spazio è necessario aumentare il numero di livelli, penalizzando però le prestazioni perché saranno necessari più accessi alla memoria centrale. Questo approccio multilivello è però compatibile con il meccanismo di memoria associativa della *TLB*, che nei sistemi a 64bit sarà generosamente capiente, ed in caso di *TLB-hit* verranno risparmiati tutti gli accessi extra causati dall'approccio multilivello.

###### *Tabella delle pagine Invertita:*

Esistono anche delle architetture a 64bit che usano una strategia del tutto diversa per il problema di gestire lo spazio di indirizzamento.

L'idea è che questa tabella ha una struttura diversa rispetto la classica tabella, infatti presenta una voce per ogni frame fisico presente in memoria, e per certi versi può essere considerata come una tabella dei frame estesa ed univoca, in cui ogni voce contiene l'informazione del *Address-Space Identifiers* (o il *PID*), il *numero di pagina*, *bit di allocazione*, *bit di modifica*, *bit di referenziamento*, ecc…

La traduzione avviene mediante una ricerca all'interno di questa tabella, usando come chiavi il *PID* ed il *numero di pagina virtuale* richiesta. Se la ricerca non trova nessun riscontro si otterrà un *page-fault*, in quanto la pagina non si trova in memoria fisica, altrimenti l'indice del risultato della ricerca della *tabella invertita* corrisponde esattamente al *numero di frame*.

I vantaggi di questo approccio sono:

* Le dimensioni ridotte della tabella.
* La tabella è univoca per tutti i processi e non dipende dal numero di essi.

Gli svantaggi sono:

* La ricerca è assolutamente inefficiente, ma può essere migliorata indicizzando le chiavi di ricerca con una funzione *hash*.
* Le molte tabelle classiche forniscono più informazioni dell'unica *tabella invertita*, infatti quest'ultima contiene informazioni solamente sulle pagine presenti in memoria centrale, per quelle invece "parcheggiate" sul disco non presenta nulla, perciò non ho nessuna informazione sul dove andarle a cercare.

#### *Cache della memoria:*

Una memoria *cache* è composta da *linee di cache* (lunghe 32 o 64 word) ognuna delle quali conterrà una copia speculare di una porzione di memoria centrale e sarà identificata dall'indirizzo a partire dal quale parte la linea di cache inizia.

In particolare questa chiave permette di capire se, a fronte di una richiesta, se quest'ultima ricade o no all'interno di una delle *linee di cache*.

La natura di questa chiave, e quindi del tipo di indirizzo, dipende dal dove la memoria cache è inserita, distinguendosi in;

* мми 
  сри 
  cache 
  Метоту 
  DRAM **cache basata sugli indirizzi fisici:** posta dopo l'*MMU*.  
  La ricerca avverrà per indirizzo fisico e risulterà penalizzata nell'efficacia perché, essendo la cache in cascata solo dopo l*'MMU,* verrà pagato il costo di una traduzione da indirizzo virtuale a indirizzo fisico che agirà da collo di bottiglia alla potenzialità effettiva della cache.

* **cache basata sugli indirizzi virtuali:** posta prima l*'MMU.*

мми 
Метогу 
НАМ La ricerca avverrà per indirizzo virtuale e tutte le *linee di cache* proverranno dagli spazi di indirizzamento virtuale dei singoli processi. Potenzialmente la ricerca può essere più efficace poiché nel caso fortunato di *cache-hit* non dovrò pagare il costo della traduzione. Le *linee di cache*, un po' come per la *TLB*, per non essere azzerate ad ogni *context-switch* devono essere supportate dall'hardware permettendo una ulteriore chiave di ricerca: la chiave ***ASID*** (*address-space identifiers*).   
Questo modello scala male all'aumentare della dimensione della cache.

I due approcci sono entrambi plausibili e nella pratica vengono usati entrambi; cache di primo livello basata su indirizzi virtuali e cache di secondo livello basata su indirizzi fisici.

сри 
addr 
мми 
ТЕВ 
cache 
Метогу 
DRAM 

***Algoritmi di sostituzione delle pagine:***

In caso di *page fault* ed in assenza di frame liberi l'hardware cede il controllo al sistema operativo (trap) che commuterà la *CPU* ad eseguire, in modalità kernel, una procedura preposta a gestire questo evento.

Se tutti i frame della memoria fisica sono occupati allora bisognerà scegliere, tramite un *algoritmo di sostituzione delle pagine*, una pagina virtuale *vittima* da spostare dalla memoria centrale verso il disco per far spazio alla pagina che è stata richiesta.

A tale fine è bene ricordare il [fenomeno della localizzazione del codice](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={FBCE36E2-EFB0-0E96-1194-A46671E01EC1}&17&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one), per cui posso ipotizzare che le pagine utilizzate nel più recente passato sono più utilizzate dal processo e quindi peggiori candidate ad essere eliminate.

**Algoritmo Ottimale (OPT):**

La scelta ottimale (teorica) della pagina vittima sarebbe quella per cui l'algoritmo dovrebbe individuare una pagina che idealmente non è più utile al processo e che quindi verrà riutilizzata più in là nel futuro. Ma nella pratica è una soluzione teorica difficilmente realizzabile.

**Algoritmi NRU (Not Recently Used):**

L'idea mira al cercare di scegliere la pagina in memoria che non è stata usata di recente.  
Il *bit di referenziamento* ed il *bit di modifica* (*bit R* e *bit M*) sono molto importanti per il funzionamento di tale algoritmo. In particolare il *bit R*, venendo azzerato periodicamente dal sistema operativo, mi darà uno specchio nel brevissimo periodo di quali pagine, prima dell'ultimo azzeramento sono state utilizzate dal processo e quali no.

L'algoritmo distingue la pagine in 4 classi:

* **classe 0**: pagine non referenziate e non modificate (R=0, M=0).
* **classe 1**: pagine non referenziate e modificate (R=0, M=1).
* **classe 2**: pagine referenziate e non modificate (R=1, M=0).
* **classe 3**: pagine referenziate e modificate (R=1, M=1).

L'algoritmo sceglierà una pagina dalla classe non vuota di numero più basso.

*\**[*Soffre dell'anomalia di Belady*](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={1D6D264B-4683-0820-282D-5BE411F711AD}&3D&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one)*.*

**Algoritmo FIFO e 'Della Seconda Chance':**

Si basa sull'idea di considerare l'età delle pagine, intendendo per età l'istante in cui la pagina viene caricata in memoria centrale.

Per implementare una simile strategia posso mantenere una lista concatenate che rispetta l'ordinamento *FIFO* (*first-in first-out*), inserendo in testa le nuove pagine caricate in memoria e mantenendo in coda le pagine più vecchie. L'idea è di eliminare la pagina in coda ma questa scelta non sempre risulta felice, in quanto può rimuovere pagine vecchie ma magari molto usate.

*\**[*Soffre dell'anomalia di Belady*](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={1D6D264B-4683-0820-282D-5BE411F711AD}&3D&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one)*.*

Una sua semplice variazione consiste nel considerare il *bit R* della pagina, ovvero nel caso in cui la pagina più vecchia ha il *bit R* ad 1 questa verrà risparmiata ed anzi reinserita in testa alla lista, ponendo poi il suo *bit R* a 0.

Verrà quindi rimossa la pagina più vecchia e non usata di recente.

*\**[*Soffre dell'anomalia di Belady*](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={1D6D264B-4683-0820-282D-5BE411F711AD}&3D&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one)*.*

**Algoritmo Clock:**

L'idea è la medesima dell'algoritmo *'Della Seconda Chance'* ma implementata in maniera più efficiente utilizzando una **lista circolare** con ordinamento FIFO. Infatti in questo modo piuttosto che estrarre e reinserire posso semplicemente aggiornare il puntatore della testa all'elemento successivo, ottenendo lo stesso risultato ma in maniera più efficiente.

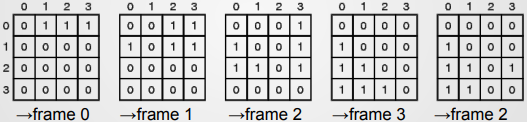
*\**[*Soffre dell'anomalia di Belady*](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={1D6D264B-4683-0820-282D-5BE411F711AD}&3D&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one)*.*

**Algoritmo LRU (Least Recently Used):**

L'idea è di rimuovere la pagina meno utilizzata di recente e può essere implementato;

Tramite un contatore nella *CPU* che conta, ad esempio, il numero di cicli della *CPU*, ed avere un campo corrispondente all'interno della tabella delle pagine. Questa soluzione è implementabile solo con il supporto dell'hardware.  
Idealmente, ogni volta che la *CPU* accede ad una pagina posso immaginare che sia l'*MMU* a leggere il valore del contatore che si trova nel registro della *CPU* e che lo copierà nel campo corrispondente all'interno della pagina. Il contatore agirà come un timestamp, più è alto più indicherà un evento recente. Al momento del *page fault*, in mancanza di frame libero, l'algoritmo *LRU* andrà a scegliere la pagina che è stata utilizzata meno di recente nel passato, andando ad attenzionare tutti i contatori delle pagine in memoria, scegliendo quella con il contatore più piccolo.

Un'altra idea basata sull'hardware per implementare il criterio di scelta *LRU* prevede di mantenere lo stato di aggiornamento delle pagine tramite una *matrice di bit* all'interno della *MMU* dalla dimensione limitata poiché numero di righe e colonne corrisponde al numero di frame concretamente in memoria fisica.  
Partendo dall'ipotesi che inizialmente la matrice sia vuota, ogni si effettua l'accesso all' i-esimo frame viene impostato ad 1 l'intera riga i-esima e poi azzerata l'intera i-esima colonna.  
Il criterio di scelta della pagina vittima consiste nel guardare tutte le righe e scegliere quella che presenta il numero minore di bit posti ad 1.



Ogni volta che viene effettuato un aggiornamento succede che, il porre ad 1 l'i-esima riga pone il frame come più importante e contemporaneamente il porre a 0 l'i-esima colonna sottrare un bit a 1 ad ogni altra pagina.

In ogni caso questo approccio non è attuabile in quanto l'hardware attuale non supporta il criterio di scelta LRU.

*\**[*Gode della proprietà di inclusione*](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={1D6D264B-4683-0820-282D-5BE411F711AD}&70&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one)*.*

**Algoritmo NFU (Not Frequently Used):**

Nel caso concreto l'hardware non fornisce nessun supporto per implementare l'algoritmo *LRU* se non il bit di referenziamento.

L'algoritmo *NFU* è una poco buona approssimazione software dell'algoritmo *LRU*.

L'idea è di mantenere una sorta di contatore per ogni pagina virtuale e periodicamente, poco prima che il *bit R* debba essere azzerato, sommare il suo valore a tale contatore. Verrà rimossa la pagina con il contatore più basso, ma questo approccio può erroneamente privilegiare pagine che sono state molto utilizzate in passato e scarsamente usate di recente e che quindi potenzialmente saranno scarsamente usate anche in futuro.

*\**[*Eredita la proprietà di inclusione*](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={1D6D264B-4683-0820-282D-5BE411F711AD}&70&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one)*.*

**Algoritmo di Aging:**

Un semplice aggiustamento dell'algoritmo *NFU* permette di ottenere una buona approssimazione del criterio *LRU*.

Presenta le stesse dinamiche dell'algoritmo *NFU*, da cui differisce per il metodo di accorpamento del *bit R* al contatore.

Nella tabella delle pagine, ogni qualvolta si debba azzerare il *bit R* infatti, si esegue uno shift a destra di una posizione del contatore di ogni pagina per poi apporvi il valore del *bit R* di ogni pagina come bit più significativo del contatore, prima suo azzeramento. Verrà sacrificata la pagina con il contatore minore.

*\**[*Eredita la proprietà di inclusione*](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={1D6D264B-4683-0820-282D-5BE411F711AD}&70&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one)*.*

**Riepilogo sugli algoritmi:**

OPT: non implementabile, ma utile come termine di paragone; 
NRU*: approssimazione rozza dell'LRU; 
FIFO*: può portare all'eliminazione di pagine importanti; 
Seconda chance*: un netto miglioramento rispetto a FIFO; 
Clock*: come S.C. ma più efficiente; 
LRU: eccellente idea (vicina a quella ottima) ma difficilmente 
realizzabile se non in hardware; 
NFU: approssimazione software abbastanza rozza dell'LRU; 
Aging: buona approssimazione di LRU con implementazione 
software efficiente. 
* soffre dell'anomalia di Belady 

***Anomalia di Belady:***

Disponendo di maggiore *RAM*, quindi di più *frame*, ci si aspetterebbe che un algoritmo di sostituzione delle pagine si comporti più efficientemente amministrando meglio i *frame* stessi… In realtà il ricercatore Belady osservò che su particolari pattern alcuni algoritmi si comportavano in maniera non intuitiva, aumentando il numero di *page fault*. L'anomalia di Belady si verifica quando aumentando i *frame* disponibili aumentano anche i *fault di pagina*!

***Proprietà di inclusione:***

La proprietà di inclusione si ha quando, al variare del tempo: in ogni istante l'insieme delle pagine caricate negli *n frame* disponibili rimarrebbero comunque presenti se in quello stesso istante ci fossero invece *n+1 frame* disponibili.

#### *Altri aspetti da curare su sistemi basati su paginazione*

**Allocazione dei frame:**

Come decide il sistema operativo quanti frame assegnare ad ogni singolo processo? Cosa succede quando nu processo viene creato? Quali pagine vengono caricate in memoria centrale e quali invece lasciate sul disco?

***Avvio di un processo***

Quando un processo viene avviato i SO spesso si appoggiano ad una strategia chiamata *Paginazione su Richiesta* (*Pure Demand Paging*). Il sistema operativo cioè non compie nessuna scelta al posto del processo, ma lo avvia senza allocarvi alcuna *pagina virtuale* e gestirà perciò questa fase di *bootstrap* del processo esattamente come farà in generale.

Il primo tentativo di accesso che farà il processo, che probabilmente sarà alla prima pagina che conterrà la prima istruzione del programma, che risiederà all'interno dello spazio di indirizzamento: provocherà un *page fault* che permetterà all'intera pagina contenente anche quella prima istruzione richiesta di essere caricata in memoria, e così anche per i dati che probabilmente il processo richiederà. Quindi osserveremo un picco di *page fault* all'avvio del processo, che man mano si assesteranno ad una frequenza minore.

***Suddivisione dei frame tra i processi***

Un altro aspetto legato alla gestione dei frame è quello di come il SO suddivida la disponibilità dei frame ai vari processi.

Questi *n frame*, rimossi quelli dedicati esclusivamente al SO, possono essere divisi tra i processi utente secondo diverse strategie:

* **Allocazione equa**;  
  Assegna a *run-time* una equa quantità di *frame* agli *n processi* in esecuzione, ridistribuendoli non appena un processo viene creato o terminato.  
  Questa strategia non tiene conto delle diverse esigenze di ogni processo; infatti, un numero di *frame* che possono andare bene per un programma di piccole dimensioni (lunghezza del codice e quantità di dati) possono risultare insufficienti per processi più pesanti.
* **Allocazione proporzionale**;  
  Adegua a *run-time* (perché le necessità di un processo variano nel tempo) il numero di *frame* alle necessità di ogni processo, ridistribuendo in base ai cambi di stato e alle terminazioni e creazioni di processi. Assegna più *frame* ai processi più pesanti e meno frame a quelli più piccoli.
* **Allocazione per priorità**;  
  Stesse politiche dei precedenti ma adegua a *run-time* il numero di *frame* in base anche alla priorità di ogni processo.

***Numero minimo di frame che posso assegnare***

Esiste un numero minimo di *frame* che posso assegnare ad un processo? Se assegnassi, ad esempio, un solo *frame* allora molto probabilmente il processo andrà in *page fault* ad ogni istruzione provocando il [*fenomeno del trashing*](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={1A564C22-C610-0690-0BB6-70A68A14EFA1}&30&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one).

**Trashing del sistema**:

Un sistema va in *trashing* quando la quantità di tempo che la *CPU* dedica ad un processo è minore della quantità di tempo che invece spende per l'*overhead* legato ai *page fault* del processo stesso, andando a rallentare anche tutti gli altri processi.

In una situazione reale assegnare un solo *frame* ad un processo non ne permetterebbe fisicamente la sua l'esecuzione. Infatti una singola istruzione del processo potrebbe provocare potenzialmente due o più riferimenti alla memoria…

Esempio:

L'istruzione *Assembly ADD* e l'istruzione *Assembly MOVE*

|  |  |
| --- | --- |
| ADD tmp, 1 | Il primo *fetch* sarà di lettura della pagina contente l'istruzione stessa. Il secondo *fetch* riguarderà invece sicuramente una pagina diversa contenente i dati [*tmp*].  Il secondo *fetch*, essendo disponibile solamente un *frame*, andrà a sovrascrivere il frame contenente l'istruzione provocando poi un *page fault* che richiederà nuovamente la lettura dell'istruzione dalla pagina contenente il codice che verrà sovrascritto e si andrà avanti in un loop di *page fault*. |
| MOVE &1000, &2000 | Questa istruzione potenzialmente potrebbe richiedere 3 frame diversi, una per l'istruzione, una per l'indirizzo &1000 ed una per l'indirizzo &2000 |

Abbiamo capito che in realtà il minimo dipende dall'architettura e dall'istruzione, ma ci sono anche altri fattori che aumentano il minimo teorico…

*Accesso alla memoria indiretto*: posso avere istruzione in cui il singolo operando si riferisce alla locazione di *memoria virtuale* X che contiene il valore dell'*indirizzo virtuale* Y. In questo caso il numero di pagine caricate dipende dal numero di indirizzamenti indiretti.

***Quali pagine considerare per la rimozione?***

* **Allocazione locale**: considero solo le pagine appartenenti al processo.  
  In questo modo viene garantito che ogni processo userà esattamente il numero di *frame* massimo assegnatogli.
* **Allocazione globale**: tutte le pagine (anche di altri processi).  
  In questa strategia non si può più parlare di ripartizione di *frame* in quanto un processo potrebbe "rubare" *frame* ad altri processi che contengono pagine che a livello globale sono le meno usate.

***Cosa succede se ci sono pochi frame assegnati ad un processo?***

* Sotto il minimo strutturale: il processo verrà sospeso e swapperà sul disco.
* Poco sopra il minimo: può andare in *Trashing*.

Bisognerebbe quindi assegnare un numero di *frame* commisurato alle necessità del processo.

**Modello di località:**

Il *modello di località* consiste in un insieme di locazioni di memoria contenenti le singole istruzioni e i singoli dati utili al processo.

In generale vale la regola per cui, se il numero di *frame* assegnati ad un processo sono sufficienti a contenere l'attuale località del processo, allora il processo genererà pochi *page fault*.

**Working set:**

Per capire se un processo possiede un numero di frame adeguato alle sue esigenze ci serve un metodo per misurare il *concetto di località*:

Il concetto di *working set* (*WS*) cerca di approssimare il *concetto di località* di un processo. Un *WS* è un insieme di pagine e l'idea è che, in un dato istante, venga definito un *Working Set* algoritmicamente e in modo tale che contenga l'intera *località del processo*.

Astrattamente possiamo immaginare che un *WS* sia formato da tasselli abbastanza ampi (pagine) da contenere la *località del processo* - quest'ultima visualizzabile come singoli puntini contenenti singoli indirizzi, variabili o istruzioni.

Bisogna quindi tener traccia, in termini di pagine, di quale sia il *Working Set* di ogni singolo processo e di fare poi tutte le valutazioni su di esso piuttosto che sulla *località* - perché tener traccia della località intesa come collezione di indirizzi è molto più dispendioso per il sistema.

page reference table 
,2615777751623412344434344413234443444 
WS(t,) = (1 
= {3,41 Un *Working Set* di un istante è composto da un parametro *Δ* e si definisce come l'insieme delle pagine consultate negli ultimi *delta* accessi alla memoria centrale:

In questo modo riesco a fornire una sorta di approssimazione della località in un dato istante.

Conoscere il *WS*, inteso come approssimazione delle esigenze del processo, permette di fare alcune valutazioni e applicare strategie che permettono di prevenire il *fenomeno del trashing*.

Il *SO* può andare a considerare ad esempio quante sono le pagine che appartengono al *WS* di ogni processo in esecuzione, calcolare la dimensione del *WS* stesso in quel dato istante per ogni processo e ottenere dalla somma di tutte queste esigenze una valutazione globale della popolazione di processi, ovvero la quantità di frame che in generale i miei processi hanno bisogno per andare avanti.

In più posso ad esempio paragonare questa somma alla quantità di *RAM* a disposizione e preliminarmente individuare situazioni di pericolo in cui si rischia di saturare la disponibilità concreta la memoria centrale rischiando il blocco totale.

***Come tengo algoritmicamente traccia del working set di ogni processo?***

Posso utilizzare il bit di referenziamento ed il suo azzeramento periodico per contare quanti bit conservare.

**Page fault frequency: (senza approfondire troppo)**

Esiste un approccio più diretto sostituivo al WS: *Page fault frequency*.

Si basa sulla relazione che lega la frequenza dei *page fault* con la disponibilità dei *frame* del sistema.

Si osserva come in generale se le esigenze del sistema cominciano ad aumentare allora anche i *page fault frequency* aumenteranno e come un numero più alto di *frame* tende a diminuire il numeri di *page fault*.

Mettendo in relazione le due osservazioni si possono stabilire un limite superiore ed uno inferiore all'interno del quale sarebbe ottimale mantenersi.

**Politica di pulitura delle pagine:**

Il tempo di gestione dei *page fault* è legato alle operazioni che il sistema deve compiere come conseguenza della situazione che ritrova…

La situazione migliore è quella in cui si hanno *frame* liberi e, senza dover invocare l'algoritmo di sostituzione delle pagine a rallentare ulteriormente, si gestisce velocemente il *page fault*.

Con una specifica strategia si può rendere più probabile il riscontrare *frame* liberi disponibili attraverso. Tipicamente questo compito è assegnato ad un ***Paging Daemon*** che periodicamente si attiva liberando preventivante dei frame, creando una *pool di frame liberi*, che permetterà di gestire rapidamente futuri *page fault*.

Tipicamente questo *Daemon* viene lanciato quando la *CPU* si trova in *IDLE*, in questo modo si sfrutterà uno stato di inutilizzo per diminuire il carico di lavoro per quando invece ci sarà una invasione di richieste.

Per scegliere quale frame liberare verrà invocato l'algoritmo di sostituzione delle pagine, che verrà usato "impropriamente", per individuare la pagina che globalmente tra tutte è meno interessante per il processo in esecuzione - andandola eventualmente a sincronizzare in caso il *bit di modifica* fosse a 1 - ed etichettando il frame come libero per mantenere questa *pool di frame liberi*.

In caso di **ripescaggio** di una stessa pagina, ovvero il caso in cui l'algoritmo di sostituzione andasse a selezionare -erroneamente - una pagina che l'istante successivo verrà richiesta nuovamente dal processo, allora: formalmente il frame è stato etichettato come libero ma fisicamente questo non è mai stato azzerato e conterrà ancora la pagina richiesta, cosicché il *SO* sia ancora in tempo per restituire il frame contenente la pagina richiesta al processo in questione, il tutto senza fare input/output extra.

**Dimensione delle pagine:**

* Vantaggi di una pagina grande
  + Tabelle delle pagine più piccole.
  + Migliore efficienza del trasferimento I/O perché si ha meno frammentazione di dati sparpagliati nel disco.
  + Tende a minimizzare il numero di *page fault*. - Ipoteticamente se avessi una sola pagina non avrei mai nessun page fault
* Vantaggi di una pagina piccola
  + Minore frammentazione interna
  + Migliore precisione nel definire il *Working Set* in memoria.

**Pagine condivise:**

Se avessi più istanziazioni di uno stesso programma allora idealmente le pagine contenente il codice del processo saranno identiche e possono essere condivise tra tutte le istanziazioni, facendo puntare le rispettive voci delle tabelle delle pagine delle varie istanze allo stesso frame contenente il codice.

Ciò è possibile se le pagine sono condivise in sola lettura e cioè se nessuno può violare lo spazio di indirizzamento di altre istanze.

Se invece i processi hanno richiesto di comunicare in lettura/scrittura allora viene utilizzato il meccanismo di *Inter Process Comunication* (*IPC*). In questo caso il *SO* farà puntare una *pagina* della tabella delle pagine di ogni processo che ha richiesto la *IPC* ad un unico *frame* condiviso.

1. Permette al *SO* di risparmiare *frame*, quindi memoria fisica.
2. Permette al SO di implementare un meccanismo di *IPC*.
3. Facilmente implementabile sia su [tabelle multilivello](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={74D8A210-4E66-08E1-1979-80DD46B29A4F}&B&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one) che a livello singolo (vettore).

***Problemi delle pagine condivise***

* [Gestione della Cache](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={52E1B5C4-88B7-0EEE-18D4-E796FB76D257}&14&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one): nel caso di pagina condivisa in lettura/scrittura

Supponiamo ad esempio di avere tre processi: P1, P2 e P3 che condividono una pagina:

*{P1 indicizza il frame condiviso 7 nella sua pagina 1 della sua tabella delle pagine ecc…}*

|  |  |  |
| --- | --- | --- |
| P1 | *frame* 7 | *pagina* 1 |
| P2 | *frame* 7 | *pagina* 1 |
| P3 | *frame* 7 | *pagina* 22 |

Avremo che la prima *word* del *frame* 7, a seconda di chi la guarda, avrà un indirizzo virtuale uguale per P1 e P2 ma diverso per P3 (fenomeno dell'aliasing).

Per la *cache* organizzate per *Indirizzi Virtuali* che usa la doppia chiave [*indirizzo virtuale*, *ASID*] questo è un problema perché supponiamo che per primo il processo P1 acceda alla prima *word* della pagina condivisa: la *cache* porterà la *linea di cache* che copre quella *word* al suo interno e da quel momento in poi il processo P1 non farà accesso alla *memoria centrale* poiché saranno tutti *cache-hit* e si andrà ad utilizzare quella *linea di cache*.

Supponiamo adesso che sulla stessa *CPU*, senza azzerare la *cache* perché corredata di *ASID*, passo al processo P3 che vorrebbe leggere la stessa *word* condivisa con il processo P1. Succederà che la *cache* riceverà una richiesta per un *ASID* e soprattutto per un *indirizzo virtuale* differente. La *cache* non può accorgersi del problema e lo tratterebbe come un *cache-miss*, andando a popolare un'altra *linea di cache* che riguarderà lo stesso pezzo di memoria fisica andando a scindere quella condivisione tra P1 e P2 con P3.

Per risolvere si può:

* Disabilitare la cache per quanto riguarda le *pagine condivise*.
* Se vi è il supporto hardware, usare *cache* con ricerca basata su *indirizzi virtuali* e *tag fisici*.
* [Tabella delle Pagine Invertita](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={B1228C95-45B7-00F1-3DD0-8E1388B7FBEF}&CF&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one):  
  Essendo la stessa *word* vista con *alias* diversi a seconda di quale processo legga/scriva, allora non è chiaro come la tabella della pagine invertita dovrebbe gestire le *pagine condivise*.
* Singolo core: il *SO* riesce a riconoscere che non si tratta di *page fault* ma di un problema di *aliasing* e può sovrascrivere l'alias di P1 della tabella delle pagine invertita con l'alias generato da P3.
* Multi core: in questo caso la tabella invertita è usata in modo concorrente da più *CPU*, a questo punto non è plausibile che la tabella venga corretta simultaneamente.

**Altre tecniche di condivisione:**

***Copy-on-write***

Consideriamo lo scenario della chiamata *fork()*. Essendo il processo figlio una copia speculare del padre allora questi condivideranno, finché possibile - ovvero finché uno dei due non effettua una scrittura - tutti i tipi di pagine (codice e dati) marcate però con soli permessi di lettura, risparmiando così memoria.

Ma cosa succede se un processo dei due tenta di modificare una pagina?  
L'*MMU* intercetterà il tentativo di scrittura bloccandolo perché le pagine condivise erano state marcate in sola lettura, e invocare una routine del *SO* per gestire l'eccezione. Il *SO* si renderà conto di ciò che è avvenuto e creerà una copia del contenuto condiviso sul primo tentativo di scrittura cercando un altro *frame* libero per conservarvi la pagina modificata, aggiornerà le voci delle tabelle delle pagine dei due processi per farli puntare a questo punto e due *frame* distinti e manipolerà anche i permessi dei due processi sulle loro pagine distinte come in scrittura/lettura.

Una possibile *exec()* nel figlio andrà ad azzerare lo spazio di indirizzamento del processo chiamante e il *SO* andrà a manipolare nel padre i permessi delle pagine adesso non più condivise con il figlio ripristinandole in lettura/scrittura.

***Zero-fill-on-demand:***

Consideriamo lo scenario in cui un processo alloca memoria (*new* o *malloc*). Normalmente in questi casi se lo spazio all'interno dell'*heap* è esaurito in generale la soluzione è quella di farlo crescere verso l'alto su indirizzi crescenti.

Supponendo che il processo abbia ad esempio bisogno di ulteriore 1000 pagine allora questo starebbe richiedendo una nuova porzione di memoria al *SO* che fino a quel momento non poteva utilizzare, come gestisco i *frame*?

La strategia è quella di soddisfare la richiesta del processo immediatamente, sfruttando la *copy-on-write* su un contenuto speciale che il *SO* mantiene in memoria fisica: ovvero la *read-only-static-zero-page*, cioè un frame prefissato vuoto che viene mappato e marcato in sola lettura per tutte le nuove pagine richieste. Tutte le pagine punteranno quindi ad uno stesso *frame* vuoto e qualora il processo volesse scrivere su una di esse, essendo marcate in lettura, l'*MMU* intercetterà il tentativo di scrittura bloccandolo ed il *SO* gestirà l'eccezione cercando un frame vuoto per soddisfare la richiesta.

Similmente a come avevamo visto per la [pool dei frame liberi](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={08847D3F-C650-0D95-28FB-5DFF2BBDFF13}&C2&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one), anche in questo caso verrà mantenuta una pool dei frame vuoti, che verranno svuotati solamente nei momenti in cui una CPU è in IDLE, per velocizzare la gestione dell'eccezione al momento propizio.

**Librerie Condivise:**

Si tratta di uno scenario di condivisione di codice tra programmi distinti che ricalca un modello di programmazione ormai consolidato: ovvero l'uso di *librerie condivise*.

Una libreria è una collezione di procedure, oggetti e componenti scritte da team, che possono essere utilizzate come riciclo di codice preesistente (*#include*).

Quando un programma che usa una libreria viene compilato: il codice scritto viene tradotto dal compilatore in codice binario, la gestione delle parti di codice che fanno riferimento alla libreria in questione si distingue in due scenari;

* **Linking statico** (scenario più semplice e meno comune)  
  Il *linker* preleva dalla libreria in questione la porzione di codice che il programma ha richiesto dalla libreria e la accorperà al file binario che il compilatore creerà, accorpando quindi il tutto in un unico blocco binario.  
  Questo approccio tende a creare file binari più grandi in quanto va a caricare fisicamente quelle porzioni di codice all'interno del file binario, inoltre non permette la condivisione di tali segmenti di codice comune qualora fossero in esecuzione più programmi che usino quella tessa libreria. Andando ad occupare tanta memoria sul disco quanti file binari vengono compilati e tanta memoria *RAM* quanti processi che usino tale approccio e librerie comuni vi siano in esecuzione. L'unico aspetto positivo è che programmi compilati in questo modo sono totalmente autonomi a prescindere dall'ambiente in cui vengono lanciati, perché dispongono di ciò che necessitano al loro interno.
* **Linking dinamico**  
  Il compilatore, insieme al *linker*, produce un file binario in cui nelle porzioni di codice dove si fa riferimento alla libreria viene inserito uno ***stab*** e ne verrà inserito uno per ogni riferimento alla libreria. Uno *stab* è una porzione minimale di codice standard che viene invocato nel momento in cui, per la prima volta, la libreria viene richiesta da un qualsiasi processo. Quando il programma verrà lanciato in un processo, questo *stab* avrà il compito di mappare, se non lo era già, l'intera libreria di cui si ha bisogno nello spazio di indirizzamento del programma che lo ha richiesto. Successivamente il *SO* sovrascriverà a *run-time* quello stesso *stab* nell'indirizzo virtuale a cui il programma dovrà saltare per utilizzare la funzionalità richiesta nella libreria. Questo approccio fa risparmiare sia memoria sul disco, perché i file binari compilati di fatti non conterranno al loro interno alcun codice di librerie, ma anche memoria centrale perché stesse librerie richieste da programmi in esecuzione verranno mappate una sola volta alla prima richiesta e marcate in sola lettura in un frame comune a tutti i processi richiedenti, e inoltre essendo mappate verranno caricate solamente quelle pagine della libreria in cui ricadono le richieste dei processi e non l'intera libreria.  
  Un altro aspetto positivo di questo approccio è dato dal fatto che lo sviluppo della libreria diventa indipendente dal programma, cioè se ad esempio dopo la compilazione del file binario la libreria venisse aggiornata allora non è necessario ricompilare il programma (al contrario dell'approccio *Linking statico*) in quanto gli *stab* fanno riferimento al nome della libreria indipendentemente dalla sua versione.

[**File mappati**](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={B85B2E0F-F518-023D-04BA-2BD2596BFF2E}&23&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one)**:**

Come visto è possibile mappare file (anche pesanti) sullo spazio di indirizzamento virtuale e beneficiare della potenza di tale strumento, ovvero il caricamento spezzettato in pagine e su richiesta dal processo.  
L'atto della mappatura non provoca alcun input/output e l'intero file verrà mappato nell'enorme spazio di indirizzamento virtuale. Un primo tentativo di accesso in lettura/scrittura di una parte di questo file provocherà un *page fault* che porterà al caricamento in memoria della singola pagina su contenente quella porzione di file piuttosto che l'intero file.

Se per caso anche altri processi richiedessero la mappatura dello stesso file, già presente in memoria virtuale, allora sarà formalmente condiviso in lettura (potrebbe anche essere usato come strumento di ***IPC*** [*Inter process Comunication*]).

Quando un nuovo processo viene creato, a prescindere dal *SO*, questo crea uno spazio di indirizzamento virtuale completamente vuoto, poi mapperà il file eseguibile per intero sulla parte bassa dello spazio di indirizzamento e successivamente lancerà il processo appena creato. Questo eseguirà la prima istruzione del programma e questo primo tentativo di *fetch* per ottenere la prima istruzione provocherà un *page fault* che caricherà i dati statici (es. variabili globali) del programma e l'*entry point* (es. *main*) del codice eseguibile del programma in *RAM*.

## Memoria centrale (RAM) per il kernel

Il kernel, come entità presente in memoria centrale, ha le sue esigenze in termini di allocazioni dinamica di memoria per le sue strutture dati. Queste esigenze vengono identificate da dei connotati che le distinguono da quelle dei normali processi utente e né permettono una gestione ad-hoc.  
Alcune richieste possono essere gestite utilizzando la stessa strategia della paginazione, come abbiamo visto per i processi utente, altre richieste devono invece necessariamente essere gestite tramite una diretta manipolazione della memoria centrale fisica.  
Per chiarire le idee, finora molte eccezioni viste nell'astrazione dello "spazio di indirizzamento virtuale" dei processi utente sono state risolte "magicamente" grazie all'intervento del kernel, ma il kernel non può sempre usare la stessa astrazione che lui stesso sta implementando per risolvere le sue stesse eccezioni/problemi.



I kernel moderni dispongono quindi di una doppia gestione della memoria, da un parte un qualcosa di simile a uno spazio di indirizzamento virtuale gestito su paginazione e dall'altra una sorta di allocazione diretta della memoria fisica.

### Gestione delle allocazioni dinamiche dirette del kernel

#### *Prima soluzione: Buddy system*

Inizialmente usato sui sistemi Linux poi abbandonato in favore dello *Slab Allocation*, serve per gestire un segmento di memoria contigua fisica. Affinché possa essere utilizzato parte dall'ipotesi che il segmento da gestire sia di una dimensione della potenza del due. Quando il kernel ha bisogno di allocare una struttura dati farà richiesta per un blocco idoneo a contenere una già nota taglia, ad esempio 3kb. A questo punto l'idea alla base del *Buddy System* è quella di spezzettare man mano in due il blocco di partenza in blocchi sempre più piccoli fino ad ottenere quel blocco di dimensione uguale o arrotondata alla potenza di due superiore della taglia richiesta, nel nostro esempio 4kb.

Tutti i resti di pezzetti rimasti sono liberamente allocabili per altre richieste che soddisferanno le varie taglie.

* Questo approccio produce frammentazione interna dato che a prescindere dalla richiesta il blocco viene sempre allocato in potenze di due.
* Ma teoricamente produce anche frammentazione esterna (anche se poco probabile data le tipiche dimensioni ridotte delle taglie) perché man mano che spezzetto posso avere blocchi troppo piccoli per soddisfare taglie più grandi che se invece potessi fondere soddisferei.
* physically contiguous pages 
  128 KB 
  64 KB 
  256 KB 
  128 KB 
  AR 
  64 KB 
  32 KB 
  32 KB L'aspetto positivo sta nella sua semplicità perché questo modo di suddividere la memoria permette di gestire la coalescenza dei blocchi liberi, ovvero il meccanismo di fusione di blocchi fratelli, in maniera semplice ed efficiente.

#### *Soluzione attuale: Slab Allocation:*

L'idea parte dal concetto e dalla gestione degli ***slab***. Uno *slab* è una sequenza di pagine contigue che, con riferimento a delle taglie di kernel prestabilite, si ha che la sua dimensione è sempre un multiplo sia della taglia stessa sia della dimensione della pagina (perché in realtà la memoria fisica è gestita per *frame* perciò per il *SO* ci deve essere una corrispondenza) {ES: struttura 3K, frame 4K ==> *slab* 12K).

Grazie a questa proprietà lo *slab* di taglia X è idoneo a contenere molteplici strutture del kernel di tipo *X*, *slab* di taglia *Y* per molteplici strutture di tipo *Y* e così via per tutti i tipi di strutture del kernel.

* Questo implica che non ci sarà frammentazione interna perché esisteranno solamente *slab* di dimensioni uguali al tipo di richieste del kernel.
* Non ci sarà neanche frammentazione esterna perché qualsiasi buco tra pagine dello *slab* potrà comunque contenere richieste di taglia *Z* di molteplicità più piccola.

Questi *slab* sono organizzati in *cache* intese come collezione di più *slab*. Una *cache* di taglia 3kb sarà composta da tanti *slab* di taglia 3kb, una *cache* di 7kb…

Se lo *slab* in uso si satura allora il kernel andrà ad individuare nella memoria fisica un altro *frame* che verrà allocato come nuovo *cache* di *slab* vuoti.

Nessuno spreco e massima efficienza.

kernel objects 
3-KB 
objects 
74<8 
objects 
caches 
slabs 
physically 
contiguous 
pages 

## Segmentazione:

La segmentazione è un modo diverso di presentare la memora centrale ai processi. Finora i processi, grazie all'astrazione spazio di indirizzamento, hanno consultato la memoria centrale tramite l'indirizzo virtuale; ma un altro modello di astrazione è quello della *Segmentazione*, che prevede di presentare al processo la propria memoria come una collezione di segmenti in cui il singolo segmento, a differenza dello spazio di indirizzamento virtuale, è specializzato e omogeneo a contenere uno specifico contenuto.

Esiste quindi idealmente un segmento per lo stack, uno per l'heap ecc… ed ogni segmento è separato dagli altri.

Questo modello, per riferirsi ad un oggetto della memoria, usa un sistema di coordinate bidimensionali *{numero di segmento, offset}* che andranno ad identificare all'interno del segmento stesso l'esatta word richiesta dal processo.

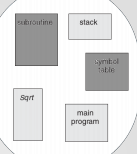
I vantaggi di tale modello sono:

Il livello di dettaglio nella "classificazione" di ogni segmento asseconda meglio il lavoro del compilatore e la visione logica del programmatore.

Si può predisporre un segmento per ogni tipologia di oggetto: un segmento per libreria, uno per funzione, oggetto ecc… e l'indirizzo del segmento sarà l'*entry point* dell'oggetto stesso.

Questo modello può essere combinato con quello della paginazione in un approccio misto, ottenendo i benefici di entrambi.

*Ma come faccio a raccordare la memoria fisica centrale, che segue un modello monodimensionale basato sull'indirizzo fisico, con questo bidimensionale?*

La tabella dei segmenti raccorda queste due entità, essa indica per ogni segmento dove si trova in memoria fisica poiché ogni entry della tabella riporta la coppia *{#segmento ==> (base, limite)}* che appunto indicano dove si trova nella memoria fisica e quanto è grande.

Problemi:

1. Un segmento, che può crescere e diminuire di dimensioni come un *heap*, potrebbe essere limitato nella crescita perché essendo tutti i segmenti contigui allora potrebbe non esserci spazio per la crescita.
2. C'è potenzialmente frammentazione esterna.

### Segmentazione su *Pentium Intel* (*32bit*) (non approfondito):

Gli Intel Pentium supportano siano *segmentazione* che *paginazione*; supponendo di usare entrambi i meccanismi:

la *CPU* genera un indirizzo logico (bidimensionale), questa informazione viene passata alla *Segmentation Unit* che linearizza l'indirizzo - ovvero trasforma tramite la tabella dei segmenti l'indirizzo logico bidimensiole in indirizzo logico lineare-, questo viene passato alla *Paging Unit* che trasformerà l'indirizzo lineare -che si muove all'interno di uno spazio di memoria virtuale contiguo - in indirizzo fisico che in realtà è spezzettato in pagine e che può essere presente in memoria anche solo in parte.

logical 
address 
linear 
address 
physical 
address 
CPU 
segmentation 
unit 
paging 
unit 
physical 
memory 

Queste *CPU* supportavano al più 16.000 segmenti con indirizzamenti a 4gb (limite massimo teorico).

Si distinguono due tipi di segmenti:

* **Locali**: segmenti dedicati ad ogni singolo processo e "privati".
* **Globali**: segmenti potenzialmente utilizzabili anche da altri processi, "condivisibili".  
    
  Questa distinzione nella tipologia dei segmenti avveniva tramite una duplice *tabella dei segmenti*, ognuna composta da circa 8000 voci:
* **Local Descriptor Table** (*gdt*): è una unica tabella globale condivisa tra tutti i processi.
* **Local Descriptor Table** (*ltd*): ne esiste una per ogni processo.

Per individuare a quale segmento ci si vuole riferire si utilizzavano e usano ancora dei ***registri di segmenti*** che sono dei registri di *CPU* che contengono l'identificativo di alcuni segmenti.

Ogni segmento supportava 4 livelli di protezione (quindi non soltanto *kernel mode* e *user mode*).

La *Paging Unit* utilizza una *tabella delle pagine multilivello* con 2 livelli in associazione con la **memoria associativa** (*TLB*).

Le architetture a *64bit* utilizzano invece *tabelle multilivello* fino a 4 livelli.

### Gestione della memoria su *Linux* (non approfonditio):

Linux è uno tra i casi più diffusi di *SO* multi-architettura. Architetture differenti offrono supporti diversi a *paginazione segmentazione*, il problema sta nel fatto di avere una gestione software comune che si possa adattare alle varie architetture.

In linux questo compromesso si è raggiunto non offrendo di fatto ai processi il modello della segmentazione ma sfruttare comunque il supporto hardware per ottimizzare alcune operazioni del *SO*. Avremo così un *SO* che riesce a girare su architetture che non supportano la segmentazione ma sfruttarla non-nativamente in quelle che la supportano.

Per lo stesso motivo di adattabilità tra varie architetture linux fornisce supporto solamente ai due classici livelli di protezione (*kernel mode*, *user mode*)

La tabella delle pagine è una t*abella multilivello* fino a 4 livelli con recente supporto a 5.

Aumentando il numero di livelli aumenta l'overhead (più accessi alla memoria per consultare la tabella) questo problema viene risolto tramite ***TLB*** (*memoria associativa*) e anche troncando gli indirizzi virtuali teorici di 2^64bit a 2^48bit, che è molto più pratico per una tabella a 4 livelli.

questo indirizzo virtuale, come visto nella teoria, è suddiviso 5 parti di cui 4 per la navigazione tra le tabelle e l’ultima per l'offset della pagina. Il so può anche girare utilizzando meno delle 4 tabelle, per questo su architetture a 64bit possono anche girare so a 32bit.

Strumenti usati da linux:

* *copy-on-write* e *read-only- static-zero-page.*
* struttura tipica dello *spazio di indirizzamento* (non segmentazione pur essendo supportata dall'hardware).
* *file mappati* (codice e librerie condivise).
* allocazione dinamica dell'heap tramite la chiamata **brk()**.
* un algoritmo di sostituzione delle pagine ibrido tra *clock* e *NRU.*
* *Slab Allocation* per la memoria del kernel.

# I File System - memoria secondaria (disco)

Il *file system* è una astrazione non fornita dall'hardware ma offerta dal *sistema operativo*, che permette di avere un modello basato su file, directory e permessi che finora abbiamo dato per scontato.

Esso si pone l'obiettivo di risolvere il problema di gestire grandi quantità di informazioni in modo **persistente** e **condiviso** tra più processi.

Ogni file system avrà delle caratteristiche standardizzate e uguali per tutti ma anche una propria implementazione e gestione dei dettagli che lo possono differenziare da altri *file system*.  
Alcune differenze possono essere:

* nomenclatura
* tipi di file
* tipi di accesso
* metadati
* operazioni supportate dai file
* strutture dati per la gestione dei file

Il set di operazioni sui vari file è invece standardizzato e prescinde dal *file system.*Ad esempio l'accesso condiviso ai file, essendoci potenzialmente concorrenza tra più processi, viene gestito dal *sistema operativo* mediante il meccanismo del *lock* (**semafori**) ma poi sarà il *file system* a gestire l'implementazione dei permessi al file stesso.

I *file system* usano strutture dati per la gestione dei file, in particolare si distinguono due tipi di *tabelle dei file aperti*:

* **Tabella Globale**, in cui il *SO* tiene traccia di quali sono i file aperti in generale da tutti i suoi processi.  
  In particolare la generica entry conterrà le info relative a quell'oggetto ma che non sono legate allo specifico processo (nome del file, il percorso della posizione, la dimensione, …)
* **Tabella per processo**. Le voci presenti in questa tabella sono meno dettagliate e conterranno informazione specifiche utili al processo (puntatore alla voce della Tabella Globale, puntatore *file pointer*, …)

La lettura del disco avviene per blocchi.

**Struttura di un file system**

Tipicamente un disco viene organizzato in partizioni, ovvero non viene visto come un unico contenitore ma lo si può suddividere in parti in cui possono essere memorizzati diversi *file system*, ad esempio per separare il *sistema operativo* dai dati (trattando questi ultimi in un *file system* diverso).

**Master Boot Record (MBR):**

Partition table 
Entire disk 
Disk partition 
Boot block 
Superblock 
Free space mgmt 
I-nodes 
RW dir 
Files and directories Gli standard che governano i pc gestiscono come creare e come delimitare queste partizioni. La standard uscente è il cosiddetto *Master Boot Record* (***MBR***) che lascia il suo posto alla odierna *GUID Partition Table* (***GPT***).

Il funzionamento del *Master Boot Record* è il seguente: identificherà nel disco il suo primo blocco e, suddividendolo in due parti, il *MBR* conserverà nella prima sezione le istruzione per eseguire l'avvio mentre nella seconda parte una tabella delle partizioni utile a gestire il partizionamento del disco.

Questo standard era pensato per partizionare dischi di dimensioni ridotte, infatti la tabella che si componeva di sole 4 voci permetteva al più 4 partizioni che a loro volta erano limitate ad una dimensione massima di 2TB.

La tabella indica semplicemente dove inizia e dove termina ogni partizione e quale, tra queste, fosse una partizione attiva (ovvero contenente un *SO* e avviabile).

Nel tempo si è cercato di estendere questi limiti mantenendo la retro-compatibilità, introducendo le partizioni estese che prevedevano di estendere concettualmente la tabella utilizzando una della 4 partizione come un appendice della tabella stessa, bypassando il limite delle 4 partizioni.

**file system in generale**

***avvio***

La componente hardware del computer all'avvio esegue delle istruzioni già presente al suo interno su memorie non volatili, il cosiddetto ***BIOS***, che ha il compito di attivare le prime periferiche, mettere in funzione il controllo dei dischi e in generale individuare ed avviare il *sistema operativo*. Ciò avviene tramite una serie di step prefissati che iniziano proprio dal contenuto del *Master Boot Record* e in particolare della sua componente ***Boot Blok***.

Il *BIOS* andrà a caricare in memoria *RAM* il *boot record* e a questo verrà affidata la CPU.

Questo blocco ha il compito a sua volta di individuare all'interno del disco la partizione del *sistema operativo* ed avviarne conseguentemente il suo ***kernel***. Questa individuazione avviene attraverso la lettura della tabella delle partizioni.

***struttura***

La struttura della singola partizione presenta delle componenti fisse:

* **Boot Block**: ogni partizione presenta questo campo in quanto potenzialmente potrebbe contenere un *sistema operativo* avviabile. Se invece in esso sono contenuti dei dati allora semplicemente questo blocco sarà vuoto.
* **Superblocco**: è per standard il secondo blocco ed il suo ruolo è di rendere disponibili, in una posizione fissa nota, alcuni parametri della partizione stessa (tra cui l'indicazione del tipo di *file system* usata, la dimensione della partizione, la dimensione di un *cluster* - insieme di più blocchi).
* Tutti i blocchi che seguono il *superblocco* memorizzano tutti i dettagli relativi all'astrazione *file system* e le strategie per farlo cambiano da *file system* a *file system*.

***implementazione dei file***

Un qualunque file di una dimensione arbitraria viene memorizzato utilizzando un certo numero di *blocchi del disco*, andandosi così a spezzettare in unità di sequenze di byte che la cui somma ricrea il contenuto logico del file.

Vi sono vari modi per gestire il come e dove memorizzare queste sequenze di blocchi;

File A 
(4 blocks) 
(3 blocks) 
File C 
(6 blocks) 
File E 
(12 blocks) 
File a 
(3 blocks) 
File D 
(5 blocks) 
File 
(6 blocks) **Allocazione contigua**

La prima strategia è quella di semplicemente allocare i vari blocchi che compongono il file in maniera contigua sul disco.

Problemi:

* **Frammentazione interna**: ipotizzando blocchi da 1024byte ed un file di 3900byte allora mi serviranno 4 blocchi in cui nell'ultimo spreco circa 100byte.
* **Frammentazione esterna**: l'eliminazione di file porterebbe alla liberazione di un certo numero di blocchi ma in maniera dislocata e non compatta.
* Il file è una entità dinamica che può aumentare e diminuire di dimensione, se i blocchi precedenti e successivi sono occupati allora il file non può espandersi. Potrei fixare imponendo lo spostamento del file in coda, dove vi sono blocchi liberi ma è una operazione assurdamente dispendiosa se il file cambia di dimensione continuamente.

Vantaggi:

* La semplicità nella gestione dell'allocazione, cioè le informazioni necessarie per tener traccia di dove è memorizzato un file saranno semplicemente l'indirizzo di partenza del file e la sua dimensione del file.
* ***Accesso Sequenziale***, il file venendo letto dal processo in modo sequenziale sarà agevolato essendo questo memorizzato contiguamente nei blocchi.
* ***Accesso Diretto***, se il processo volesse spostarsi in uno specifico punto all'interno del file senza dover leggere tutto fino a quel punto, ecco che nell'allocazione contigua ciò è facilmente implementabile.
* Nei *dischi elettromeccanici* possiamo immaginare che anche sulla superficie del disco il file vi sia inciso "contiguamente", pertanto la testina lo leggerà naturalmente con la rotazione del disco, senza dover saltare da un punto ad un altro del cilindro. In questo modo pago il ritardo del ***Sick-time*** (posizionamento della testina) solamente una volta e la ***latenza di rotazione*** del disco è nulla in quanto il file è inciso contiguamente.

L'allocazione contigua oggigiorno è utilizzata nei *file system* di sola lettura dei dischi ottici (CD rom e DVD), perché si parla di un *file system* immutabile, che una volta scritto (masterizzato) non può più essere modificato.

File 
block 
4 
File 
block 
File 
block 
File 
block 
10 
Fila 
block 
12 
physical 
block **Allocazione con liste collegate**

Si rinuncia all'idea di imporre la contiguità nella memorizzazione del file, andandolo a memorizzare in pezzetti in parti distinte del disco. I blocchi del disco non necessariamente contigui vengono considerati nodi di una lista linkata, collegati logicamente l'uno all'altro da un riferimento al nodo successivo. La dimensione del campo che fa riferimento al successore dipende dalla dimensione del disco (quindi dal numero di blocchi del disco) e più questo è capiente e più spazio occupa questo porzione di blocco.

Vantaggi:

* Ogni file può incrementare la sua dimensione semplicemente aggiungendo e collegando logicamente il numero di blocchi necessari, come è naturale che debba essere.
* Niente frammentazione esterna.
* Niente frammentazione interna.

Problemi:

* Una porzione del blocco deve essere utilizzata per memorizzare il riferimento al blocco successivo e ciò deve avvenire per ogni blocco del disco,
* per dischi molto capienti è un bello spreco.
* andrò a memorizzare all'interno del blocco una quantità di dati effettivi che non è più una potenza di due, creando una sorta di disallineamento tra la visione standard che hanno i processi e la quantità di dati memorizzate all'interno del blocco. Ad esempio se un processo richiede un blocco di dati di 1024byte di fatti questo si concretizza in termini di *i/o* in una lettura di due blocchi perché ogni blocco conterrà 1020byte di dati e 4byte di riferimento al successivo.
* L'***Accesso Diretto*** ai file è supportato in modo pessimo, in quanto per leggere un file all'i-esimo blocco di questa lista è necessario leggere tutti i blocchi precedenti.

**Allocazione con lista tabellare (FAT - *file allocation table*)**

aaaq 
OJaq Sims V OLI La strategia è simile alla allocazione con *liste collegate*, ma l'idea è di mantenere le informazioni dei riferimenti logici successivi tra i vari blocchi, non nel blocco stesso ma su un'unica tabella.

Questa tabella contiene essenzialmente metadati, fa parte integrante delle strutture chiave del *file system* che la impiega e sarà memorizzata da qualche parte sul disco ma per ragione di efficienza in fase di apertura della partizione (montaggio del *file system*) verrà caricata in memoria centrale.

La tabella è composta da un numero di voci indicizzate equivalente al numero di blocchi allocabili all'interno della partizione che deve essere gestita, l'i-esima voce indicherà il blocco successivo al blocco i-esimo. Il numero di blocco di partenza ci verrà fornito tramite i metadati delle directory.

L'***Accesso Diretto*** avviene tramite operazioni *i/o* sulla tabella **FAT**, ma trovandosi questa in memoria centrale l'*overhead* generato è trascurabile.

Problemi:

* E' poco scalabile in quanto la tabella **FAT** comincia a diventare ingombrante se deve gestire dischi di dimensioni odierne, e mantenere costantemente in memoria centrale una tabella di centinaia di megabyte è un problema.

Address of disk 
Address ot disk dock 
Address ot disk 2 
Address ot disk 3 
Address ot disk 
Address ot disk dock S 
Address ot disk 
Address ot disk 
Address ot block of pointers 
DisÅ block 
containing 
addresses **Allocazione con *i-node*:**

Un *i-node* è una struttura dati di supporto alla memorizzazione dei file che nasce per tener traccia delle informazioni di un singolo file, ci sarà quindi un *i-node* per ogni file e sono strutture persistenti che all'occorrenza vengono portate in memoria centrale.

Tipicamente sono strutture di piccole dimensione, più piccole un blocco del disco e tipicamente un blocco riesce a contenere più *i-node*.

Il fatto di avere una struttura dati specializzata per il singolo file permette di minimizzare la quantità di dati da portare in memoria centrale allo stretto indispensabile, ovvero solo quando ci sarà almeno un processo che sta consultando il file stesso.

L'*i-node* contiene due generi di informazioni, una prima parte sono informazioni collegate all'entità file (dimensione del file, utente proprietario, …). Il nome del file non è invece presente in quanto si troverà già tra i metadati della directory insieme al numero di *i-node*.

La seconda parte dell'*i-node* contiene invece una lista di indirizzi di *blocchi del disco* contenenti le varie parti del file. Essendo un *i-node* molto piccolo la lista sarà limitata nel numero di voci; per i file che vanno oltre il numero di voci standard allora è possibile prevedere una strategia di estensione della lista:

***Variante collegata:***

Esaurite le voci sfrutto l'ultima per estendere la lista facendola puntare invece ad un *blocco del disco* contenente non una porzione del file ma puntatori ad altri blocchi del disco, che conterranno porzioni del file.

Questa strategia permette perfettamente l'***Accesso Sequenziale***, per quanto riguarda l'***Accesso Diretto*** richiede di portare l*'i-node* in memoria ma per file molto grandi potrei avere una lista abbastanza lunga e in generale non è l'ideale. Per file di molto piccoli è invece ottimo.

***Variante multilivello:***

Ogni voce dell'*i-node* punterà ad un blocco che conterrà, in una struttura ad albero, puntatori ad altri blocchi del disco, che conterranno porzioni del file.

L'***Accesso Diretto*** è ben implementato in quanto la struttura ad albero permette di calcolare il l'i-esimo blocco richiesto con un fattore costante.

Il lato negativo è lo spazio occupato per file molto piccoli, ma che pone un limite alla dimensione massima di ogni file.

I-node 
Attributes 
Singlg 
block 
Addresses of 
Double 
data blocks 
indirect 
block 
Triple 
b I cuck 
basato su i-node: ***Variante ibrida:***

Approccio concretamente utilizzato da alcuni *file system*.

L'*i-node* dedicherà una certa quantità di byte per i metadati e 13 voci alla lista.

In le prime 10 voci conterranno indirizzi a blocchi diretti, che punteranno cioè alle prime 10 porzioni del file (*variante collegata*, per file piccoli).

L'11a voce conterrà invece un indirizzo di *blocco indiretto singolo*, cioè l'indirizzo ad un blocco che conterrà *indirizzi diretti* a porzioni del file.

In una struttura ad albero sbilanciato, la 12a voce un *indirizzo indiretto doppio*, cioè l'indirizzo ad un blocco che conterrà *indirizzi indiretti singoli* che punteranno ad *indirizzi diretti* a porzioni del file.

L'ultima voce, con lo stesso ragionamento, conterrà un indirizzo ad un *blocco indiretto triplo*.

Questa strategia permette di effettuare un ***Accesso Diretto*** per file molto grandi (di svariati giga) con al più 3 accessi di lettura dal disco, mentre per file molto piccoli posso cavarmela anche con la sola lettura dell'*i-node* dalla memoria centrale.

**Implementazione delle directory (*non approfondito*):**

La directory è una collezione di riferimenti a file e in base alla strategia che viene usata per gestire l'allocazione dei file la directory deve a sua volta avere una struttura più o meno complessa.

Nel caso più semplice degli *i-node* che contengono tutti i metadati, la directory deve mantenere solo il nome del file ed il numero dell'*i-node*.

Nel caso invece di un file system in cui non ho una struttura dati associata ad ogni singolo file, allora tutti i metadati relativi al singolo file sono memorizzati nella directory.

Se il numero totale di metadati di un file è noto e di grandezza massima nota, il nome del file essendo di lunghezza variabile non può essere memorizzato in un campo prestabilito.

Prima strategia: vedere una directory come una sequenza di voci formate da una componente di dimensione fissa ed una di dimensione variabile. Presenta il fenomeno della frammentazione e della poca adattabilità alla dinamicità dei file.

Seconda strategia: cercare di separare la componente di dimensione di fissa da quella di dimensione variabile allo scopo di gestire le parti di dimensioni fisse senza frammentazione e gestendo e compattando la componente dinamica nell'heap.

**Condivisione di un file (Hard-link e Soft-link):**

I *file system UNIX* mettono a disposizione un meccanismo che permette di riferirsi ad un dato file tramite uno o più riferimenti;

***soft-link***

I *link simbolici* sono dei file speciali che contengono al loro interno il riferimento del file a cui puntano. Se viene cancellato il *soft-link* il file target rimarrà inalterato mentre il link diverrà inconsistente. E’ possibile creare link anche a file che si trovano in file system diversi.

***hard-link***

I *file system UNIX* rappresentano ogni oggetto tramite *i-node* memorizzato tra i *meta-dati* delle cartelle. Un *hard-link* agisce creando nella stessa cartella un ulteriore puntatore al medesimo *i-node*. Dato un file e creato un *hard-link* ad esso, a posteriori sarà impossibile distinguere qual era il nome originale e quale il *link*, quindi si ha una trasparenza totale. Poiché gli *hard-link* lavorano a livello di *i-node* sarà possibile creare solo tra file dello stesso *file system*. Il file system, per ogni *i-node*, mantiene un contatore del numero di *link fisici* che puntano ad esso.  
Quando di cancella un *hard-link*:

1. Il SO controlla il contatore dei *link* all’interno dell’*i-node* del file che si vuole cancellare:
   * Se tale contatore è uguale ad 1 allora viene cancellato sia il *link* della directory che l’*i-node* stesso (compreso il contenuto del file).
   * Se il contatore è strettamente maggiore di 1 allora verrà decrementato e cancellato solo il riferimento all*’i-node* (il *link*).

**Gestione dei blocchi liberi:**

Lo spazio libero è considerato anche esso una risorsa da gestire e da tracciare, in quanto il *SO* si ritroverà a rispondere a richieste di *blocchi liberi* e allo stesso tempo cercherà, per quanto possibile, di memorizzare file contigui in memoria virtuale in sequenze contigue di *blocchi* del disco (a favore dei dischi elettromeccanici).

Questa risorsa può essere gestita attraverso varie soluzioni:

***bitmap***

L'idea è di avere una *bitmap* in cui ogni singolo *bit* rappresenta un blocco allocabile al file, rappresentando con 1 un blocco allocato e con 0 un blocco disponibile.

La dimensione della *bitmap* è relativamente piccola, dipende dal numero di blocchi da gestire e viene conservata su una posizione nota del disco.

* **I vantaggi:**

E' possibile, sotto richiesta, caricare porzioni di questa bitmap in memoria centrale sotto richiesta pensando anche di paginare la *bitmap* stessa su uno spazio di indirizzamento virtuale.

Disponendo quindi in *RAM* di una porzione di bitmap è possibile individuare sequenze di *blocchi liberi* idonee ad essere allocati, permettendo quindi al *SO* di perseguire la contiguità nell'allocare i blocchi ai file.

***liste concatenate***

I *file system* basati su *i-node* devono invece avere una struttura dati dedicata a tener traccia dei blocchi liberi.

Gli *i-node* sono preallocati, quindi esistono in quantità limitata e sono riciclabili, precisamente esiste una bitmap che mi dice quali sono gli *i-node* liberi e quali quelli allocati e quando un *i-node* è allocato questo viene citato tra i meta-dati delle directory.

L'idea è di mantenere una lista dinamica di blocchi liberi che verrà mantenuta sul disco e che essa stessa userà i blocchi liberi come nodi della lista. La lista inizialmente occuperà quindi l'intero disco e la sua dimensione andrà diminuendo man mano che il disco si riempie. Generalmente in *memoria centrale* viene mantenuto un solo nodo della lista, ragion per cui è difficile garantire la memorizzazione contigua.

E' anche possibile strutturare la lista per memorizzare sequenze di blocchi liberi, salvandovi all'interno una coppia di valori, ovvero il *numero di blocco libero* e la lunghezza della sequenza di *blocchi liberi* che parte da quel *numero di blocco*.

**Controlli di consistenza**

Nell'ambito dei *file system* esiste una problematica di particolare rilevanza, ovvero il mantenere le sue strutture dati di base coerenti ed indenni rispetto a situazioni estreme, quali crash del sistema o il suo arresto repentino.

Il problema è legato al fatto che alcune strutture dati, per ragioni di efficienza, vengono mantenute in *memoria centrale* e solo periodicamente sincronizzate sul disco. In generale molte operazioni di base sul file system coinvolgono più di una struttura dati e nell'applicazione di una qualsiasi operazioni può succedere che a seguito di un crash queste vengano interrotte a metà, non permettendo la sincronizzazione e portando a fenomeni di inconsistenza e alla potenzialmente corruzione dei dati stessi. Questo problema viene gestito tramite strategie per limitare i danni e recuperare la consistenza delle strutture dati al costo di controlli di ridondanza piuttosto dispendiosi. Spesso i sistemi prevedono di attivare delle utility a seguito di crash, o in generale a seguito di un ***unmount*** scorretto del file system, verificabile tramite un *flag* di *mounting*.

Black number 
7 
8 
9101112131415 
0 
o 
o 
20 
0 1 1 0 [ğ 1 Blocks in üşe 
1 00 1 0 0 Free 

**prima strategia:**

Una prima verifica da effettuare è la coerenza tra lo stato di allocazione dei singoli blocchi e ciò che invece risulta scritto nelle strutture dati. Un blocco può infatti essere libero o allocato ad un *i-node* in uso e per controllare questa coerenza si può effettuare una scansione prevedendo due liste, una dei *blocchi liberi* ed una dei *blocchi in uso*.

Partendo quindi da due liste vuote si riempie quella dei blocchi allocati andando a scansionare le strutture dati che servono per allocare gli *i-node* allocati.

Le utility alla fine della scansione e alla rilevazioni di errori affrontano delle scelte per la loro correzione.

Ad esempio nel primo caso dell'immagine risulta un blocco né allocato né libero, in questo caso la correzione consiste nel segnalare il blocco nella struttura dati dei blocchi liberi.

Nel secondo caso di incoerenza invece un blocco risulta essere presente due volte nella lista dei *blocchi liberi*, si gestisce andando a togliere una delle due voci dalla lista dei *blocchi liberi*.

Nel terzo caso di incoerenza risulta invece che due file distinti X e Y (ma può anche trattarsi di un medesimo file) usino un blocco comune per memorizzare una porzione di essi. La risoluzione in questo caso sarà più complicata perché bisognerà andare a capire a chi appartiene realmente quel blocco comune ma una scelta che può fare l'utility è quella di cercare un blocco libero, duplicare in questo blocco quello comune e separare i due flussi. Ma il problema permane perché potenzialmente ci sarà stato un danno al file (o ad entrambi) che l'utility non gestire.

**Journaling:**

E' un'idea che può essere aggiunta a qualunque *file system*, anche datato, e permette di aggiungere un grado di resistenza delle strutture dati di base dei *file system* di fronte ad arresti inaspettati del sistema.

La strategia del *journaling* consiste nel mantenere un file *log* sul disco (o *journal*) che tiene traccia della serie di micro operazioni che stanno per essere effettuate sul momento sul *file system* ancor prima di farle e cancellarle subito dal journal solo dopo averle eseguite tutte ed aver sincronizzato i dati sul disco.

In generale, in caso di crash, per ragioni di efficienza sia di importanza si dà più importanza al mantenimento della coerenza dei meta-dati di un file piuttosto che al dato stesso, questo perché sarebbe più dannoso riuscire a scrivere i dati e poi magari non potervi accedere perché i suoi meta-dati risultano corrotti.

In seguito ad un crash avvenuto durante l'esecuzione delle operazioni ed in presenza del *journal* sostanzialmente vengono ripetute tutte le operazioni presenti sul *log* prima del crash, ma affinché tutto funzioni bisogna garantire che le operazioni citate godano della proprietà di ***idempotenza***, ovvero che il loro reiterare non porti a sua volta a creare incoerenze.

Le operazioni segnate nel *journal* sono più che altro relative ai *meta-dati* ed i vari controlli di consistenza si limiteranno alle sole strutture dati coinvolte nel crash.

Immagine che contiene testo

Descrizione generata automaticamente**Cache del disco (buffer cache)**

E' una porzione di memoria centrale usata per collegare una memoria più veloce -la memoria centrale- con una memoria più lenta -ovvero la memoria secondaria, disco-.

Questa cache viene implementata via software, *il SO* utilizzerà una porzione di *RAM* per mantenere alcuni dei blocchi recentemente letti dal disco e inoltre, ritardando la sincronizzazione, fungerà anche come supporto in fase di scrittura per la sincronizzazione tra dati in memoria centrale e gli stessi in memoria secondaria, portando a dei vantaggi prestazionali.

Idealmente uno stesso blocco potrebbe essere nuovamente sovrascritto da una seconda operazione di scrittura ma, trovandosi ancora nella *cache del disco* in memoria centrale verrà generato un *cache-hit* risparmiando la lettura sul disco.

Oppure nei casi di operazioni su blocchi contigui, ritardando la scrittura sui *blocchi del disco* potrei permettere al *SO* di accorpare queste operazioni istruendo il *controller* del disco a scrivere con un'unica operazione e quindi un unico movimento della testina risparmiando in termini di *sick-time* e *latenza di rotazione*.

L'implementazione avviene mantenendo un insieme di blocchi, a cui vengono riservati frame liberi della memoria centrale, in una lista doppiamente linkata -in generale tutta quella memoria centrale non usata dai processi viene usata come *buffer cache*-, implementando il meccanismo di *cache-hit* e *cache-miss* e di persistenza al suo interno dei blocchi più importanti con ordinamento [*Least Recently Used* (*LRU*)](onenote:#Sistemi%20Operativi&section-id={4AEF8A0B-D08D-440D-9800-40CC449CE770}&page-id={9050FF9B-4B2F-4BE4-818C-09B990453640}&object-id={5BF8B80C-EEDE-0A47-02F6-0E1DC50C2B9F}&65&base-path=https://d.docs.live.net/adbb3d02623eb279/Documenti/Università/Materie%20da%20Dare/Sistemi%20Operativi.one), ovvero portando in cima al *buffer cache* i blocchi che man mano generano *cache-hit* e scartando gli ultimi in caso venisse richiesto un blocco libero da un processo.

La ricerca lineare per un *cache-hit* di un contenuto richiesto avviene sovrapponendo a questa lista una tabella hash che permette di ricercarvi all'interno coppia di chiavi (disco, #blocco).

Ma ritardare di troppo la sincronizzazione delle operazioni aumenta il rischio di perdita di coerenza e integrità dei dati se dovesse accadere un crash, pertanto si implementa una variante modificata che tende a sincronizzare senza ritardo le operazioni sui *meta-dati* e ogni certo lasso di tempo invece le operazioni di scrittura dei dati. Si parlerà di scrittura **sincrona** e **asincrona**.

Nell'ottica dei dischi meccanici spesso il *controller* del disco, alla richiesta di lettura di un singolo blocco questo ne leggerà anche i successivi inserendoli nella *buffer cache*, pagando un costo di trasferimento più alto ma scommettendo che quei blocchi in più non richiesti possano tornare utili nel prossimo futuro e nel caso pagando un solo costo di *posizionamento* e di *latenza* (**read-behind**). Un'altra tecnica consiste invece di rimuovere subito un blocco che è stato appena scritto sul disco dalla *cache del disco* con la speranza che un processo che abbia appena scritto quei dati non abbia bisogno subito dopo di rileggerli (**free-behind**).

Immagine che contiene testo, dispositivo

Descrizione generata automaticamente

**Tecniche per migliorare le prestazioni**

Quando il *SO* sceglie dove fisicamente posizionare le varie strutture dati del proprio *file system* deve tenere in conto che questa posizione può avere risvolti in termini di prestazioni.

L'idea ottimale sarebbe di posizionare gli *i-node* predeterminati in gruppi di *i-node* alternatamente in ogni cilindro del disco, permettendo l'uso della restante traccia. Quando il *SO* dovrà scegliere dei blocchi per salvare un file allora, quando possibile, sceglierà un blocco che si trova sulla stessa traccia del suo *i-node* o su uno vicino.

# Scheduling del disco

*Il SO* può intervenire dal punto di vista delle prestazioni anche sul modo in cui le operazioni vengono svolte dal *controller del disco*. Il *sistema operativo* crea infatti una ***coda di richieste pendenti*** che man mano inoltra al controller e che quando svolte verranno eliminate dalla coda.

Nel caso dei dischi elettromeccanici si può pensare di gestire questa coda in modo da ottimizzare le operazioni cercando di minimizzare il *sick-time* -ovvero il tempo di spostamento del *blocco delle testine* da un *cilindro* ad un altro.

Una richiesta pendente riguarderà uno specifico settore, ovvero uno spicchietto di un cilindro di una traccia, allora conoscendo il cilindro coinvolto si può gestire la coda delle richieste:

* **First Come First served**, metodo più semplice ed equo che gestisce la coda secondo l'ordine di arrivo delle richieste ma inefficiente perché non tiene conto del numero di spostamenti della testina.
* **Shortest Seek Time First**, sceglie nella coda la richiesta attualmente più vicina alla testina, buone prestazioni ma non equo in quanto si possono presentare fenomeni di **starvation** per cui delle richieste vengano rimandate perché continuano ad arrivare nuove richieste per la zona più vicina alla testina.
* **Scan** (o dell'ascensore), l'algoritmo ha uno stato interno -un verso-, verranno soddisfatte le richieste secondo il verso via via fino a raggiungere un cilindro estremo per poi cambiare verso ogni volta. Prestazioni decenti, ma fornisce una garanzia sul tempo massimo di attesa per ogni richiesta.
* **Scansione Circolare** (***scan-c***), una volta arrivato al cilindro estremo sposta la testina direttamente all'estremo opposto. Garantisce un tempo di attesa medio più uniforme.
* **Look** e **c-look**: uguale ai due **scan** ma senza toccare gli estremi.

# Immagine che contiene testo, elettronico Descrizione generata automaticamente Sistemi RAID

In questi sistemi i *SO* sfruttano l'esistenza di più dischi allo scopo di migliorare le prestazioni e creare un sistema resistente ai danni (***fault tolerance***) aggiungendo ridondanza nel volume dei dati per ottenere queste proprietà che di norma i dischi non offrono.

La tecnica di base è quella dello *striping*: per cui si memorizza il volume logico -dato dalla somma dei volumi dei singoli dischi- spezzettandolo in *strip* ciascuno in un disco. In questo modo alla richiesta ad esempio di un file si può sfruttare il fatto che questo sia memorizzato in pezzetti su ciascun disco per fornire *n* pezzetti in parallelo.

#### *come funziona questa tecnica?*

La si applica in vari modi

##### RAID 0 (*striping*)

Immagine che contiene testo, contenitore, vetro

Descrizione generata automaticamenteIl volume logico si dividerà in stripe e memorizzati in modalità *round-robin*. Un grande gruppo di richieste sul volume logico tenderà a spalmarsi sui dischi sottostanti, quindi buonissime prestazioni ma molto più vulnerabile a guasti logici.

##### RAID 1 (*mirroring*)

Immagine che contiene testo

Descrizione generata automaticamenteIntroduciamo ridondanza per garantire *fault tolerance*. Al caro costo di usare il doppio dei dischi avremo una copia per ogni disco di dati. Migliora ancora maggiormente le prestazione, garantisce *fault tolerance* per un disco ma richiede il doppio dei dischi per la stessa quantità di dati.

##### RAID 2 (*striping* a livello di bit con *ECC*)

Aggiunge dei bit di ridondanza alla dimensione standard della *word*. Questa ridondanza permette di rilevare, tramite ***codice di Hamming***, un singolo bit di errore nella *word*. Il volume logico sarà diviso in un numero di dischi sottostanti pari al numero di bit che compone una word ed andrà a salvare un bit di ogni word in ciascun disco. Buone prestazioni e *fault tolerance* che permette il guasto di un disco ma serve mantenere la perfetta sincronizzazione nella rotazione dei dischi.

##### RAID 3 (*striping* a livello di bit con *bit di parità*)

Il bit di parità aggiunge un ulteriore bit alla word al fine di rendere pari la word stessa.

Memorizzo quindi ogni bit + il bit di parità, ciascuno in ogni disco sottostante il volume logico. In questa configurazione l'ultimo disco conterrà i bit di parità. Se si guastasse un disco riuscirei a ricostruire la word piazzando il disco di parità al posto del disco guasto. Serve mantenere la perfetta sincronizzazione nella rotazione dei dischi.

##### RAID 4 (*striping* a livello di blocchi con *XOR* sull'ultimo disco)

Suddivisione del volume logico in stripe e memorizzati in modalità *round-robin*, ma applicando l'operazione di XOR tra gli *strip* delle righe di ogni disco e memorizzando il risultato in un disco di ridondanza otterrò lo stesso risultato del bit di parità ma con il vantaggio di lavorare in *strip* e non *bit-per-bit*.

Immagine che contiene testo, contenitore, vetro

Descrizione generata automaticamenteAbbiamo un miglioramento prestazionale pari al *RAID 0* ma la scrittura di uno *strip* ci costringe a ricalcolare e riscrivere il disco di parità andando a stressare ed usurare molto l'ultimo disco. Non serve sincronizzazione.

##### RAID 5 (*striping* a livello di blocchi ma con informazioni di parità distribuite)

Equivalente al *RAID 4* ma che garantisce un uso ed una usura più omogenea per ogni disco andando a spalmare gli *strip* di ridondanza *XOR* su tutti i dischi.

# Memorie flash e Dischi Elettronici:

Sono dei banchi di memoria non volatili che garantiscono una certa uniformità di velocità di accesso, che sarà proporzionale alla quantità di dati da leggere/scrivere. La contiguità di memorizzazione non è necessaria, non essendoci parti meccaniche in movimento.

Questi dispositivi, attualmente, hanno dei limiti fisici sul numero di cancellazioni effettuabili su un blocco. Possono essere visti come divisi in blocchi ciascuno composto da più pagine, le operazioni di lettura e scrittura possono essere effettuate sull'unità minima ovvero sulla singola pagina mentre le operazioni di cancellazione possono essere fatte esclusivamente sull'intero blocco.

#### *Se una pagina di un blocco contiene una porzione di un file che un processo ha sovrascritto cosa succede?*

Non posso sovrascrivere fisicamente perché dovrei cancellare l'intero blocco e riscrivere i dati buoni solo per cancellare un dato vecchio e ogni blocco ha un numero di cancellazioni massime…

L'idea è quella che il *controller* scriverà in una pagina ancora libera il dato aggiornato e contrassegnerà quello vecchio come obsoleto tramite un sistema di rimappatura delle pagine. Il *controller* si occuperà di ripulire e ricompattare i blocchi che contengono dati obsoleti ripulendo l'intero blocco che li contiene grazie anche all'operazione di TRIM con cui il *SO* informerà il *controller del disco* quando un file non è più utile a livello di *file system*.