

# Sistemi Operativi Modulo 1 - Appunti

Axel

December 13, 2024

## Contents

<b>1 I Processi</b>	<b>5</b>
1.1 Requisiti di OS . . . . .	5
1.1.1 Cos'è un processo . . . . .	5
1.1.2 Processo in esecuzione . . . . .	5
1.1.3 Fasi di un processo . . . . .	5
1.1.4 Elementi di un processo . . . . .	6
1.2 Process Control Block . . . . .	6
1.3 Traccia di un processo . . . . .	6
1.4 Eseuzione di un processo . . . . .	6
1.5 Modello dei Processi a 2 stati . . . . .	6
1.6 Creazione di un processo . . . . .	7
1.7 Terminazione di un processo . . . . .	7
1.8 Modello dei processi a 5 stati . . . . .	7
1.9 Processi Sospesi . . . . .	8
1.10 Strutture di controllo del OS . . . . .	9
1.10.1 Memory Table . . . . .	9
1.10.2 Tabelle per l'I/O . . . . .	10
1.10.3 File Table . . . . .	10
1.10.4 Process Table . . . . .	10
1.11 Attributi di un processo . . . . .	10
1.11.1 Come si Identifica un processo . . . . .	10
1.11.2 Stato del processore . . . . .	11
1.11.3 Control Block del Processo . . . . .	11
1.12 Processi e Memoria virtuale . . . . .	11
1.13 Modalita di esecuzione . . . . .	11
1.13.1 Kernel Mode . . . . .	12
1.13.2 Da UserMode a KernelMode . . . . .	12
1.14 Creazione del processo . . . . .	13
1.15 Switching tra processi . . . . .	13
1.15.1 Quando effettuare lo switching . . . . .	13
1.15.2 Passaggi per lo switching . . . . .	13
1.15.3 Il Sistema operativo è un processo . . . . .	13
1.15.4 Esempio in Linux . . . . .	14
1.15.5 Stati in Unix . . . . .	14
1.15.6 Transizione tra processi . . . . .	15
1.16 Immagine del processo in Unix . . . . .	15

<b>2 I Thread</b>	<b>16</b>
2.1 Perché introdurre i thread . . . . .	16
2.2 ULT e KLT . . . . .	16
2.3 Processi e Thread in Linux . . . . .	17
2.4 Gli stati dei processi in Linux . . . . .	18
2.5 Segnali ed interrupt in Linux . . . . .	18
<b>3 Scheduling</b>	<b>18</b>
3.1 Tipi di scheduling . . . . .	19
3.2 Long-term scheduling . . . . .	19
3.3 Medium-term scheduling . . . . .	20
3.4 short-term scheduling . . . . .	20
3.4.1 Criteri Utente . . . . .	21
3.4.2 Criteri di sistema . . . . .	21
3.5 Turnaround time . . . . .	21
3.6 Tempo di risposta . . . . .	21
3.7 Deadline . . . . .	21
3.8 Throughput . . . . .	21
3.9 Utilizzo del processore . . . . .	21
3.10 Bilanciamento delle risorse . . . . .	22
3.11 Fairness e Priorità . . . . .	22
3.12 Prioritá e Starvation . . . . .	22
3.13 Politiche di scheduling . . . . .	22
3.14 Selection Function . . . . .	23
3.15 Decision Mode . . . . .	23
3.15.1 Pre-emptive - Non pre-emptive . . . . .	23
3.16 ESEMPIO . . . . .	23
3.17 First Come First Served . . . . .	23
3.18 Round Robin . . . . .	24
3.19 Shortest Process Next . . . . .	26
3.20 Shortest Remaining Time . . . . .	29
3.21 Highest Response Ratio Next . . . . .	29
3.22 Scheduling in Unix . . . . .	30
3.23 Architetture Multicore . . . . .	31
3.24 Scheduling in Linux . . . . .	32
<b>4 Gestione della Memoria</b>	<b>34</b>
4.1 Requisiti . . . . .	34
4.1.1 Rilocazione . . . . .	34
4.1.2 Protezione . . . . .	37
4.1.3 Condivisione . . . . .	37
4.1.4 Organizzazione Logica . . . . .	38
4.1.5 Organizzazione Fisica . . . . .	38
4.2 Partizionamento . . . . .	38
4.2.1 Partizionamento Fisso Uniforme . . . . .	38
4.2.2 Partizionamento Fisso Variabile . . . . .	38
4.2.3 Partizionamento Dinamico . . . . .	39
4.2.4 Next Fit . . . . .	42
4.2.5 Buddy System . . . . .	42
4.3 Paginazione . . . . .	44
4.4 Segmentazione . . . . .	46
4.5 Indirizzi Logici . . . . .	46

4.6 Memoria Virtuale . . . . .	48
4.6.1 Trashing . . . . .	49
4.6.2 Supporto Hardware . . . . .	49
4.6.3 Translation Lookaside Buffer . . . . .	51
4.6.4 Dimensione delle pagine . . . . .	54
4.7 Segmentazione . . . . .	55
4.8 Segmentazione e Paginazione . . . . .	56
4.8.1 Protezione . . . . .	58
4.9 Decisioni . . . . .	58
4.9.1 Fetch Policy . . . . .	59
4.9.2 Posizionamento . . . . .	59
4.9.3 Sostituzione . . . . .	59
4.9.4 Gestione del resident set . . . . .	60
4.9.5 Politica di Pulitura . . . . .	60
4.9.6 Controllo del Carico . . . . .	60
4.10 Algoritmi di Sostituzione . . . . .	62
4.11 Gestione della Memoria in Linux . . . . .	65
<b>5 La Gestione I/O</b>	<b>67</b>
5.1 Categoria di dispositivi . . . . .	67
5.2 Dispositivi leggibili dall'utente . . . . .	67
5.3 Dispositivi leggibili dalla macchina . . . . .	67
5.4 Dispositivi di rete/comunicazione . . . . .	67
5.5 Funzionamento(Semplificato) dei dispositivi . . . . .	68
5.6 Differenze tra dispositivi di I/O . . . . .	68
5.7 Tecniche per effettuare l'I/O . . . . .	70
5.8 Obiettivo per SO : Efficienza . . . . .	71
5.9 Obiettivo per SO : Generalità . . . . .	71
5.10 Buffering I/O . . . . .	75
5.11 HDD vs SSD . . . . .	76
5.11.1 Gestione della Cache . . . . .	81
5.12 RAID . . . . .	82
<b>6 File System</b>	<b>86</b>
6.1 I File . . . . .	86
6.2 Obiettivi per il File Management Systems . . . . .	88
6.2.1 Directory . . . . .	89
6.2.2 Nomi . . . . .	90
6.3 Directory di Lavoro . . . . .	90
6.4 Gestione Della Memoria Secondaria . . . . .	91
6.4.1 File Allocation . . . . .	91
6.5 Gestione dello Spazio Libero . . . . .	94
6.6 Volumi . . . . .	95
6.7 Dati e Metadati . . . . .	95
6.8 Gestione dei File in Unix . . . . .	95
6.9 Gestione dei File in Windows . . . . .	98
6.10 Linux . . . . .	99

<b>7 Gestione della concorrenza</b>	<b>100</b>
7.1 Concetti di base . . . . .	100
7.2 Concorrenza : Difficolta	101
7.3 Race Condition . . . . .	102
7.4 Per ciò che Riguarda il SO . . . . .	102
7.5 I processi e la competizione per le risorse . . . . .	103
7.6 Mutua Esclusione . . . . .	104
7.7 Semafori . . . . .	109
7.7.1 semafori Deboli e Forti . . . . .	110
7.8 Produttore-Consumatore . . . . .	112
7.9 Produttore consumatore con buffer circolare . . . . .	116
7.10 Mutua Esclusione: Soluzioni software . . . . .	117
7.11 Comunicazione Esplicita . . . . .	120
7.11.1 Sincronizzazione . . . . .	120
7.12 Indirizzamento . . . . .	121
7.13 Struttura dei messaggi . . . . .	122
7.14 Usare i messaggi per risolvere la mutua esclusione . . . . .	123
7.15 Producer-Consumer con Message Passing . . . . .	123
7.16 Problema dei lettori e scrittori . . . . .	124

## 1 I Processi

### 1.1 Requisiti di OS

Il Compito fondamentale di un sistema operativo é la gestione dei processi, ovvero delle diverse computazione che si vuol eseguire su un sistema computerizzato. Il sistema operativo deve:

- permettere l'esecuzione alternata di più processi (multitasking)
- assegnare le risorse del sistema ai processi, e decidere se dare la CPU a un processo o meno
- permettere ai processi di scambiarsi informazioni
- permettere la sincronizzazione tra processi (es. semafori)

#### 1.1.1 Cos'è un processo

Un processo è un programma in esecuzione, ovvero un'istanza di un programma in esecuzione su un computer, un certo programma puó essere eseguito più volte, e ogni esecuzione è un processo diverso. L'entitá che puó essere assegnata a un processore è il processo. Un' unitá di attività caratterizzata dall' esecuzione di una sequenza di istruzioni, da uno stato corrente, e da un insieme associato di risorse di sistema. un processo è composto da:

- Codice
- insieme di dati
- un numero di attributi che definiscono lo stato del processo

#### 1.1.2 Processo in esecuzione

processo in esecuzione vuol dire un utente ha richiesto l'esecuzione di un programma, che ancora non é terminato, ciò non significa che il processo sia in esecuzione sulla CPU, decidere se mandare in esecuzione un processo su un processore é compito del sistema operativo. Dietro ad ogni processo c'é un programma:

- nei sistemi operativi moderni, é solitamente memorizzato su archivio di massa
- possono fare eccezione i processi creati dal sistema operativo stesso
- solo eseguendo un programma si crea un processo
- eseguendo un programma più volte si creano più processi

#### 1.1.3 Fasi di un processo

Un processo si compone di 3 macro fasi:

- Creazione
- Esecuzione
- Terminazione

La terminazione di un processo può:

- essere prevista dal programma
- essere non prevista, ad esempio per un errore, in questo caso il sistema operativo lancia una interruzione che puó terminare il processo.

#### 1.1.4 Elementi di un processo

Finché il processo é in esecuzione, il sistema operativo deve tener traccia di:

- Identificatore del processo
- Stato del processo (Running,...)
- Priorità del processo
- Hardware Context: valore corrente dei registri del processore (include program counter)
- puntatori alla memori (definisce l'immagine del processo)
- informazioni sullo stato dell'input/output
- informazioni di accounting (quale utente lo esegue)

### 1.2 Process Control Block

Il sistema operativo mantiene un Process Control Block (PCB) per ogni processo, che contiene tutte le informazioni necessarie, che sono contenute nella zona di memoria riservata al kernel, solo il sistema operativo può accedere a queste informazioni. consente al sistema operativo di gestire piú processi contemporaneamente, contiene sufficiente informazioni per permettere al sistema operativo di sospendere un processo e riprenderlo in un secondo momento.

### 1.3 Traccia di un processo

Il comportamento di un particolare processo é determinato dalla sequenza di istruzioni che esegue, e dallo stato del processo, questa sequenza é detta trace, il **Dispatcher** é un piccolo programma che sospende un processo per farne andare in esecuzione un altro

### 1.4 Eseuzione di un processo

Considerando 3 processi, Ogni processo viene eseguito senza interruzioni fino al termine, ma in veritá il dispatcher sospende un processo per farne eseguire un altro, il tempo di esecuzione di un processo é diviso in piccoli intervalli di tempo, detti **time slice** es :

- Parte il processo A
- dopo un certo tempo il dispatcher sospende il processo A e fa partire il processo B
- dopo un certo tempo il dispatcher sospende il processo B e fa partire il processo C
- dopo un certo tempo il dispatcher sospende il processo C e fa partire il processo A
- e cosí via
- il dispatcher é un programma che si occupa di fare questo

### 1.5 Modello dei Processi a 2 stati

Un processo può essere in uno di due stati:

- Running: il processo é in esecuzione sulla CPU
- Not Running: il processo é in attesa di essere eseguito

esistono anche 2 stati nascosti ovvero: entrante e uscente, in ogni caso é il dispatcher che si occupa di fare il cambio di stato tra Running e not Running Dal punto di vista dell'implementazione esiste una coda di processi pronti, il dispatcher prende il primo processo dalla coda e lo esegue sulla CPU

## 1.6 Creazione di un processo

In ogni istante ci sono  $n \geq 1$  processi in esecuzione, come minimo c'è un interfaccia GUI,...ecc Se l'utente dà un comando quasi sempre si crea un processo, la creazione di un processo avviene con il **Process Spawning**: un processo crea un altro processo, il processo che crea il processo è detto **parent process**, il processo creato è detto **child process**, in questa maniera si crea una gerarchia di processi, il processo padre può creare più processi figli, e un processo figlio può creare altri processi figli.

## 1.7 Terminazione di un processo

Un processo può terminare in 2 modi:

- Normale completamento: istruzione macchina HALT, che genera un'interruzione per OS, in linguaggi di alto livello l'istruzione HALT è invocata da una system call come exit inserita dai compilatori
- Kill: Il sistema operativo può terminare un processo in modo forzato, per errori come:
  - Memoria non disponibile
  - Errore di protezione
  - Errore fatale a livello di istruzione(Divisione per 0)
  - operazione di I/O fallita

oppure l'utente può terminare un processo con un comando

si passa quindi da  $n \geq 2$  processi a  $n-1$  processi C'è sempre un processo master che non può essere terminato, il processo master è il primo processo che viene eseguito

## 1.8 Modello dei processi a 5 stati

il modello a 2 stati è troppo semplice, il modello a 5 stati è più realistico, un processo può essere in uno di 5 stati:

- New: il processo è stato creato ma non è ancora in esecuzione
- Ready: il processo è pronto per essere eseguito
- Running: il processo è in esecuzione sulla CPU
- Blocked: il processo è in attesa di un evento (es. I/O)
- Terminated: il processo è terminato

le transizioni tra gli stati sono:

- New → Ready: il processo è stato creato e pronto per essere eseguito
- Ready → Running: il processo è in esecuzione sulla CPU
- Running → Ready: il processo è stato sospeso
- Running → Blocked: il processo è in attesa di un evento
- Blocked → Ready: il processo è pronto per essere eseguito
- Running → Terminated: il processo è terminato

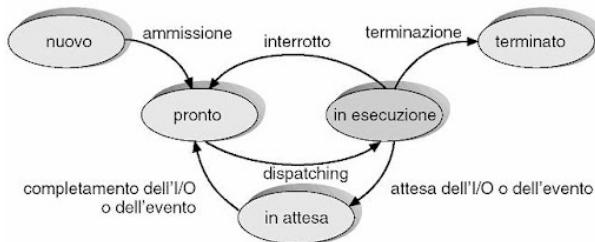


Figure 1: Modello dei processi a 5 stati

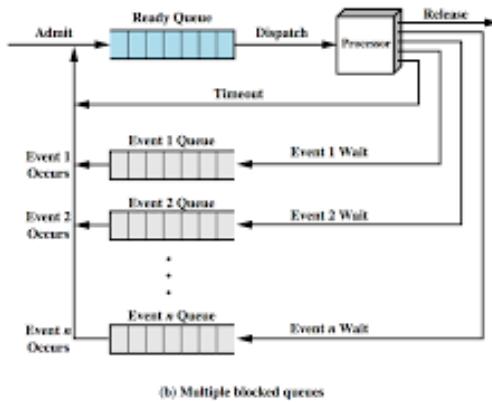


Figure 2: Diagramma delle 2 code di processi

Cosa c' é dietro a blocked ? il processo é in attesa di un evento, es. I/O, il processo é sospeso e il sistema operativo si occupa di far partire un altro processo, quando l'evento é completato il processo torna in ready. il dispatcher per cui non metterá mai un processo blocked in esecuzione. Con 5 stati abbiamo bisogno quindi 2 code di processi:

- Ready Queue: contiene i processi pronti per essere eseguiti
- Blocked Queue: contiene i processi in attesa di un evento

I sistemi operativi hanno piú di una coda di eventi che contiene gli eventi che devono essere completati, quando un evento é completato il processo torna in ready

### 1.9 Processi Sospesi

Esiste la possibilità di avere dei processi sospesi, questo é dovuto quando molti processi sono in attesa di un evento, quindi fino a che sono bloccati non possono essere eseguiti, quindi vengono spostati dalla RAM al disco, in questo modo si libera spazio in RAM, quando l'evento é completato il processo viene spostato dalla memoria secondaria alla RAM, questo cambiamento aggiunge 2 stati ai 5 precedenti:

- Ready Suspended: il processo é pronto per essere eseguito ma é sospeso
- Blocked Suspended: il processo é in attesa di un evento ma é sospeso

I motivi per sospendere un processo sono:

- Swapping: il processo é spostato dalla RAM al disco
- Interno al OS : Il sistema sospetta che il processo stia causando problemi

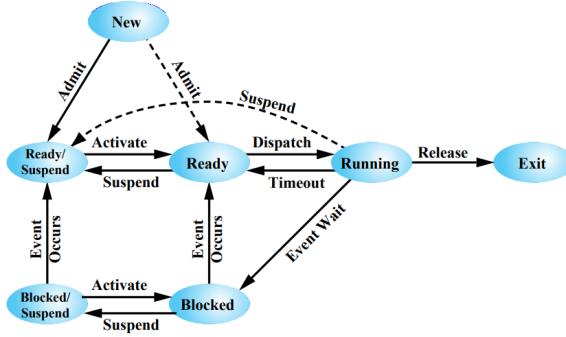


Figure 3: Modello dei processi a 7 stati

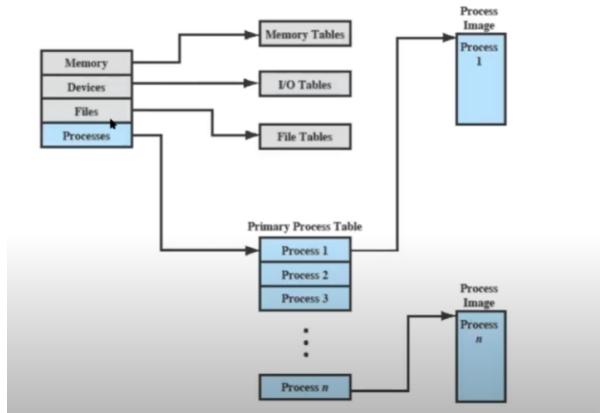


Figure 4: Strutture di controllo del OS

- Periodicità: il processo viene eseguito solo periodicamente
- Richiesta del processo padre : il processo padre può richiedere di sospendere un processo figlio

## 1.10 Strutture di controllo del OS

Il sistema operativo è l'entità che si occupa di gestire le risorse da parte dei processi, per fare ciò esso deve conoscere lo stato di ogni processo, per compiere questa operazione il sistema operativo crea 1 o più tabelle per ciascuna delle risorse esistono tabelle per gestire la memoria, i file, i dispositivi di I/O e i processi tutte queste tabelle si trovano nella zona di memoria riservata al kernel

### 1.10.1 Memory Table

Le tabelle di memoria sono usate per gestire sia la memoria principale che quella secondaria(quella secondaria serve per la memoria virtuale), le tabelle di memoria contengono informazioni su:

- allocazione di memoria principale da parte dei processi
- allocazione di memoria secondaria da parte dei processi
- attributi di protezione per l'accesso a zone di memoria
- informazioni per gestire la memoria virtuale

### 1.10.2 Tabelle per l'I/O

Le tabelle per l'I/O contengono informazioni su:

- se il dispositivo é disponibile o gi assegnato
- lo stato dell'operazione di I/O
- la locazione in memoria principale usata come sorgente o destinazione dell'operazione di I/O

### 1.10.3 File Table

Le tabelle per i file contengono informazioni su:

- Esistenza di un file
- Locazioni in memoria secondaria
- Stato corrente
- Altri attributi

### 1.10.4 Process Table

Per gestire i processi il sistema operativo usa una tabella di processi, che contiene informazioni su:

- Stato del processo
- identificatore del processo
- locazione in memoria

Blocco di controllo dell'processo (PCB) é un blocco di informazioni che contiene tutte le informazioni necessarie per gestire un processo

Si dice **process image** l'insieme di programma sorgente,dati,stack delle chiamate e PCB, eseguire un'istruzione cambia l'immagine , unica possibile eccezione é l'istruzione di jump all'istruzione stessa.

## 1.11 Attributi di un processo

Le informazioni in ciascun bloocco di controllo del processo possono essere raccolte in 3 gruppi:

- Identificazione
- Stato
- Controllo

### 1.11.1 Come si Identifica un processo

ad ogni processo é assegnato un identificatore unico, che é un intero positivo, Molte tabelle del OS usano PID per realizzare collegamenti tra le varie tabelle.

### 1.11.2 Stato del processore

Da non confondere con lo stato del processo, é dato dai contenuti dei registri del processore stesso:

- registri visibili all'utente
- registri di controllo e di stato
- puntatori allo stack
- PSW(Program Status Word) : la program status word contiene informazioni sullo stato del processo

### 1.11.3 Control Block del Processo

Per Ricapitolare il PCB contiene:

- Contiene informazioni di cui l'OS ha bisogno per coordinare processi attivi
- Identificatori : PID, PPID (parent process ID) oppure dell'utente che lo ha eseguito
- Informazioni sullo stato del processore : registri utente, program counter, stack pointer e registri di stato
- Informazioni per il controllo del processo : priorità, stato del processo, informazioni di scheduling, l'evento d'attender per essere ready
- Supporto per strutture dati: puntatori ad altri processi, per fare code di processi o liste di processi
- Comunicazione tra processi: informazioni per la comunicazione tra processi flag, segnali, messaggi
- Permessi Speciali: permessi speciali per l'accesso a risorse
- Gestione della memoria: puntatori ad aree di memoria
- Gestione delle Risorse : file aperti, ecc.

## 1.12 Processi e Memoria virtuale

Se ci sono n processi attivi, ci sono n PCB, e sono conservati nella ram conservati nella zona del kernel, tutto il resto é conservato nella memoria virtuale, una parte della memoria virtuale puó essere condivisa tra i processi mentre normalmente un processo non puó accedere alla memoria di un altro processo é peró possibile condividere la memoria se chi ha scritto il programma lo ha previsto, es. condividere una variabile tra 2 processi. Tutto questo fa sì che il control block sia una delle strutture dati piú importanti del sistema operativo perché definisce lo stato dell'OS stesso, Richiede inoltre Protezione, una funzione scritta male potrebbe danneggiare il blocco.

## 1.13 Modalita di esecuzione

La maggior parte dei processori supporta almeno due modalità di esecuzione:

- Modalitá utente: molte operazioni non sono messe a disposizione
- Modalitá kernel : pieno controllo del processore ad esempio si possono eseguire istruzione macchina

### 1.13.1 Kernel Mode

Il kernel mode è la modalità di esecuzione in cui il sistema operativo ha il controllo completo del processore, si possono gestire i processi (tramite PCB)

- creazione e terminazione di processi
- pianificazione di lungo,messo e corto termine(scheduling e dispatching)
- avvicendamento dei processi(swapping)
- sincronizzazione e comunicazione tra processi

si può gestire la memoria principale:

- allocazione di spazio
- gestione della memoria virtuale

si può gestire l'I/O:

- gestione dei buffer e delle cache per l'I/O
- assegnazione risorse I/O ai processi

Funzioni Supporto: Gestione interrupt/eccezioni,accounting,monitoraggio

### 1.13.2 Da UserMode a KernelMode

Esite quindi la necessità di passare da user mode a kernel mode, esempio: un processo vuole fare una operazione di I/O, deve chiedere il permesso al sistema operativo, di passare in kernel mode, per poi tornare in user mode. Ogni processo inizia sempre in user mode, se viene fatta una richiesta che necessita della kernel mode come una system call il processo passa in kernel mode, il sistema operativo esegue la system call e poi torna in user mode, l'ultima istruzione dell'interrupt handler è una istruzione di ritorno che fa tornare il processo in user mode. esistono quindi 3 casi in cui si passa in kernel mode:

- Codice eseguito per conto dello stesso processo interrotto, che lo ha esplicitamente voluto
- Codice eseguito per conto dello stesso processo interrotto, che non lo ha esplicitamente voluto
- Codice eseguito per conto di un altro processo

Esempio di system call sui pentium:

1. prepara gli argomenti della chiamata nei registri, tra tali argomenti c'è un numero che identifica la system call
2. esegue l'istruzione int 0x80, che appunto solleva un interrupt(in realtà una eccezione)

Anche creare un nuovo processo è una system call : l'handler di questa system call verrà ovviamente eseguito in modalità kernel, può quindi modificare la lista dei PCB

### 1.14 Creazione del processo

per creare un processo il sistema operativo deve:

- Assegnagli un PID Unico
- Allocargli spazio in memoria principale
- Inizializzare il PCB
- Inserire il processo nella giusta coda
- Creare o espandere strutture dati

### 1.15 Switching tra processi

Lo switching tra processi é il passaggio da un processo all'altro, ovver per qualche motivo l'attuale processo non deve piú usare il processore che concesso ad un altro processo.

#### 1.15.1 Quando effettuare lo switching

Lo switching tra processi puó avvenire per diversi motivi:

- interruzione : Cause esterna all'esecuzione dell'istruzione corrente Uso Reazione a un evento esterno asincrono, include i quanti di tempo per lo scheduler
- Eccezione: Associata all'esecuzione dell'istruzione, gestione di un errore sincrono
- Chiamata al OS : Richiesta esplicita, chiamata a funzione di sistema

#### 1.15.2 Passaggi per lo switching

1. Salvare il contesto del programma (registri e PC) salvati nel PCB
2. Aggiornare il process control block, attualmente in running
3. Spsotare il PCB nella coda appropriata : ready,blocked;ready/suspended
4. Scelgiere un nuovo processo da eseguire
5. Aggiornare il process control block del processo selezionato
6. Aggiornare le strutture dati per la gestione della memoria
7. Ripristinare il contesto del processo selezionato

#### 1.15.3 Il Sistema operativo é un processo

Il sistema operativo é un insieme di programmi, ed é eseguito dal processore come ogni altro programma, semplicemente lascia che altri programmi vadano in esecuzione, per poi riprendere il controllo tramite interrupt, quindi se é un processo lui stesso deve essere gestito... esistono quinndi 3 modi per gestire il sistema operativo:

- kernel separato : il sistema operativo é un processo separato
- kernel unico : il sistema operativo é un processo come gli altri
- Sistemi ibridi : il sistema operativo é un processo separato, ma esiste un kernel unico

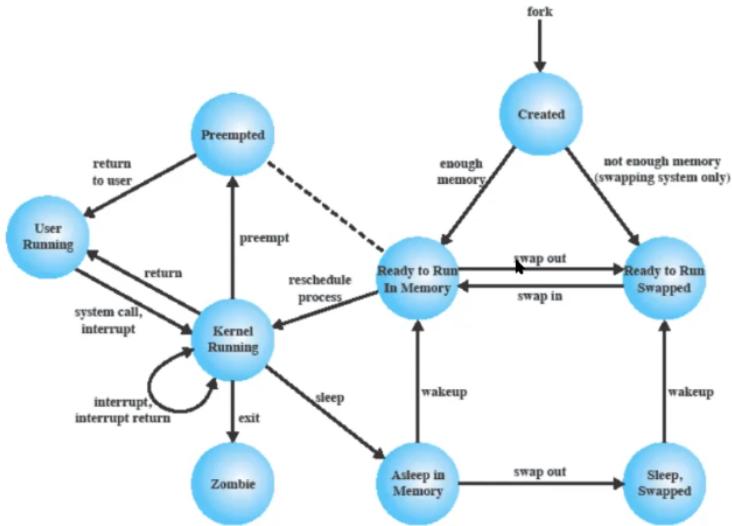


Figure 5: Diagramma stati Unix

Kernel non é un processo il kernel é eseguito al di fuori dei processi, il concetto di processo si applica solo ai processi utente, il kernel é eseguito Esecuzione all'interno dei processi utenti Il SO viene eseguito nel contesto di un processo utente, non c'è bisogno di un process switch per eseguire una funzione del sistema operativo, sole del mode switch, Comunque , stack delle chiamate separato, process switch solo, eventualmente, per passare il controllo ad un altro processo Sistema Basato su processi il sistema operativo é un insieme di processi, ogni processo é un modulo del sistema operativo, ogni processo é un modulo del sistema operativo, ogni processo partecipa alla competizione per il processore, lo switch però non é un processo

#### 1.15.4 Esempio in Linux

In linux ci sono però anche dei processi di sistema, che partecipano alla normale competizione per il processore, senza essere invocati esplicitamente, sono operazioni tipiche del sistema operativo, es. gestione della memoria, gestione dei dispositivi di I/O

#### 1.15.5 Stati in Unix

Con l'istruzione fork() si crea un processo figlio, una volta che il processo é stato creato, abbiamo 2 possibili stati Run in memory oppure Run in swapped(Memoria Virtuale) i due stati sono intercambiabili ovvero posso spostare il processo dalla memoria principale alla memoria virtuale e viceversa anche lo stato di sospensione puó avvenire in memoria o in memoria virtuale. Quando il processo in running si trova in user mode , quando vengono effettuate delle system call il processo passa in kernel mode, quando siamo in kernel mode c'è la possibilità di effettuare la preemption, ovvero sospendere il processo per farne eseguire un altro, il processo puó essere sospeso per un interrupt , nello stato di preemption si prende in considerazione l'idea di non continuare l'esecuzione, quando un processo finisce va nello stato zombie (Tipico dei sistemi Unix), quello che succede é che ci si aspetta che il processo padre sopravviva al figlio e fino a che il processo figlio non comunica al padre che ha terminato, il processo figlio rimane nello stato zombie, l' unica cosa che rimane é il PCB, quando il processo padre comunica al sistema operativo che il processo figlio é terminato, il processo figlio.

### 1.15.6 Transizione tra processi

non é interrompibile quando siamo in kernel mode quindi non va bene per sistemi real time

## 1.16 Immagine del processo in Unix

Un insieme di strutture dati forniscono al sistema operativo le informazioni necessarie per gestire un processo

- User Area : contiene il codice, i dati e lo stack del processo
  1. **Codice**: linguaggio macchina
  2. **Dati**: variabili globali e locali
  3. **Stack**: contiene le informazioni necessarie per gestire le chiamate a funzione
  4. **Memoria Condivisa**: usata per la comunicazione tra processi (deve essere esplicitamente richiesta)
- Registro : contiene i registri del processore
  1. **Program Counter**: contiene l'indirizzo dell'istruzione corrente
  2. **Stack Pointer**: contiene l'indirizzo della cima dello stack
  3. **Registri Generali**: contengono i dati temporanei
- sistema : contiene le informazioni necessarie per la gestione del processo a livello di memoria
  1. **Process Table Entry**: puntatore alla tabella di tutti i processi, dove individua quello corrente
  2. **U Area**: informazioni per il controllo dell'processo, informazioni addizionali per quando il kernel viene eseguito da questo processo
  3. **Per process region table**: definisce il mapping tra indirizzi logici e fisici (Page Table), inoltre indica se per questo processo tali indirizzi sono in lettura, scrittura o esecuzione
  4. **Kernel Stack**: Stack delle chiamate, separato da quello utente, usato per le funzioni da eseguire in modalità sistema (kernel mode)

Process Status	Current State
Pointers	puntatori alle zone del processo
Process Size	quanto é grande l' immagine del processo (il PCB ha una grandezza predefinita)
User Identifier	Identificatore dell'utente che ha lanciato il processo, c'è una differenza tra <b>Real User ID</b> e <b>effective user ID</b>
Process ID	Identificatore del processo
Event Descriptor	Motivazioni per il quale il processo é bloccato
Priority	Prioritá del processo
Signal State	Stato dei segnali
Timers	Include il tempo di esecuzione del processo, l'utilizzo delle risorse del kernel, timer usati per mandare segnali di allarme ad un processo
P <sub>Links</sub>	Puntatori per le code di processi(LinkedList),
Memory Status	Indicazioni se la memoria é in memoria principale o secondaria, indica anche se il processo se puó essere spostato

Table 1: Process Table entry

## 2 I Thread

Ci sono alcune applicazioni che richiedono la gestione in parallelo, ovvero quando viene programmata un'applicazione viene suddivisa in più parti, quindi l'applicazione rimane un singolo processo, ma che al suo interno esegue più computazioni in parallelo, quindi possiamo dire che un processo può essere composto da più thread. In questo caso è compito del programmatore occuparsi della sincronizzazione tra i thread, il sistema operativo si occupa di gestire i processi, mentre il programmatore si occupa di gestire i thread. Diversi thread condividono molte risorse, del processo, non condividono lo stack delle chiamate ed anche il processore, condividono invece memoria(Stack Esclusi), files, dispositivi di I/O, ecc. Teoricamente si può dire che il processo incorpori gestione delle risorse e scheduling

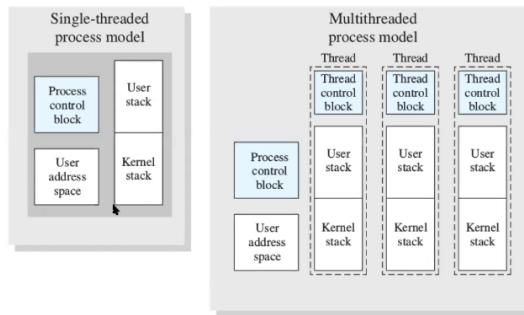


Figure 6: Thread

Se sono su un sistema operativo Single Threaded, il sistema operativo non supporta il multithreading, ho l'immagine del processo ed il PCB, se il sistema operativo supporta il multithreading, ho l'immagine del processo, il PCB e il TCB(Thread Control Block), dove Il TCB gestisce solo la parte dello scheduling

### 2.1 Perché introdurre i thread

Creare un thread è semplice/efficiente, la creazione la terminazione, fare lo switching e farli comunicare, quindi ogni processo viene creato con un thread, dopo il programmatore può creare altri thread con il comando `spawn()`, esistono chiamate di sistema per bloccare un thread, sbloccare un thread, terminare un thread.

### 2.2 ULT e KLT

Esistono 2 tipi di thread:

- **ULT(User Level Thread):** A livello di sistema operativo i thread non esistono, opportune librerie si occupano di gestire il thread
- **KLT(Kernel Level Thread):** Il sistema operativo supporta i thread, quindi il sistema operativo è a conoscenza dei thread
-

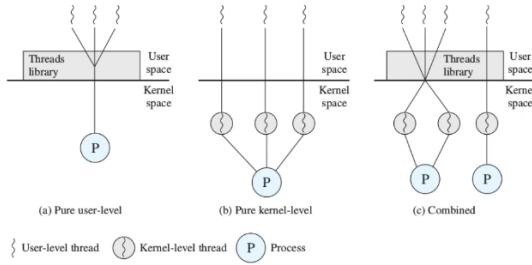


Figure 7: ULT e KLT

perché usare ULT:

- Sono più veloci da creare e gestire (Non serve fare il mode switching)
- Si può avere una politica di scheduling per ogni processo
- Permettono di usare i thread anche sui sistemi operativi che non li offrono nativamente

perché NON usare ULT:

- Se un thread si deve bloccare, si bloccano tutti i thread del processo, a meno che il blocco non sia chiamato dalla chiamata di block, al contrario con i KLT solo il thread che si blocca viene bloccato
- Se ci sono più processori o più core, i thread non possono essere eseguiti in parallelo perché il sistema operativo non è a conoscenza dei thread

### 2.3 Processi e Thread in Linux

I thread sono spesso associati al termine Light Weight Processes o LWP. Questo termine risale ai tempi in cui Linux supportava i thread solo a livello utente. Ciò significa che anche un'applicazione multithread era vista dal kernel come un singolo processo. Questo creava grandi sfide per la libreria che gestiva questi thread a livello utente, poiché doveva garantire che l'esecuzione di un thread non fosse ostacolata se un altro thread emetteva una chiamata bloccante.

Successivamente l'implementazione è cambiata, e ai processi sono stati collegati singoli thread, in modo che fosse il kernel a gestirli. Tuttavia, come discusso in precedenza, il kernel Linux non li vede come thread: ogni thread è trattato come un processo all'interno del kernel. Questi processi sono conosciuti come processi leggeri o light weight processes.

La principale differenza tra un processo leggero (LWP) e un processo normale è che gli LWP condividono lo stesso spazio di indirizzamento e altre risorse come i file aperti. Poiché alcune risorse sono condivise, questi processi vengono considerati più "leggeri" rispetto agli altri processi normali, da cui il nome di processi leggeri.

Quindi, in sostanza, possiamo dire che thread e processi leggeri sono la stessa cosa. È solo che thread è un termine usato a livello utente, mentre light weight process è un termine utilizzato a livello di kernel.

Nel kernel, ogni thread ha il proprio ID, chiamato PID (anche se forse avrebbe più senso chiamarlo TID), e ha anche un TGID (Thread Group ID) che corrisponde al PID del thread che ha avviato l'intero processo.

Semplificando, quando viene creato un nuovo processo, appare come un thread in cui sia il PID che il TGID sono lo stesso numero.

Quando un thread avvia un altro thread, il thread avviato ottiene il proprio PID (in modo che lo scheduler possa programmarlo indipendentemente), ma eredita il TGID dal thread che lo ha creato.

In questo modo, il kernel può programmare i thread indipendentemente dal processo a cui appartengono, mentre i processi (gli ID del gruppo di thread, o TGID) vengono mostrati agli utenti.

Per ogni thread, esiste quindi un PCB per ogni thread, questo crea un piccolo overhead perché esiste una duplicazione di alcune informazioni (puntatori)

## 2.4 Gli stati dei processi in Linux

Linux ha 5 stati per i processi, Linux non distingue tra ready e running, divide però in due stati blocked,

- Task Running: il processo è in esecuzione sulla CPU
- Blocked:
  1. Task Interruptible: il processo è in attesa di un evento, ma può essere interrotto
  2. Task Uninterruptible: il processo è in attesa di un evento, ma non può essere interrotto
  3. Task Stopped: il processo è stato fermato
  4. Task Traced: il processo è tracciato
- EXIT Zombie: il processo è terminato, ma il processo padre non ha ancora comunicato al sistema operativo

## 2.5 Segnali ed interrupt in Linux

Non bisogna confondere i segnali con gli interrupt, I segnali possono essere inviati da un processo ad un altro processo, quello che succede che il campo del PCB viene aggiornato con il segnale che è stato inviato, quando il processo viene schedulato, il kernel controlla se ci sono segnali da gestire se si esegue la funzione di gestione del segnale, alcuni signal handlers possono essere sovrascritti dal programmatore, alcuni segnali hanno l'handler non sovrascrivibile, in ogni caso sono eseguiti in user mode, mentre gli interrupt sono eseguiti in kernel mode.

## 3 Scheduling

Un sistema operativo deve allocare risorse tra i processi, tra le risorse quella più ovvia è il processore, l'uso del processore è detto scheduling, la metodologia che gestisce l'uso del processore è detta politica di scheduling, lo scopo dello scheduling è assegnare ad ogni processore qualcosa da eseguire, bisogna quindi che lo scheduling sia il più efficiente possibile, tenendo conto di vari aspetti:

- tempo di risposta (Tendo a minimizzare)
- throughput (Tendo a massimizzare)
- efficienza del processore (Tendo a massimizzare)

altri obiettivi dello scheduling è non fare favoritismi tra i processi, tuttavia nei moderni sistemi operativi esiste la priorità dei processi, quindi alcuni processi vengono privilegiati rispetto altri, inoltre lo scheduling deve evitare lo starvation, ovvero che un processo non venga mai eseguito, non lasciare mai il processore inattivo, avere un overhead basso (il tempo per fare lo scheduling deve essere il più basso possibile),

### 3.1 Tipi di scheduling

- Long-term scheduling: decide quali processi devono essere caricati in memoria
- Medium-term scheduling: decide quali processi devono essere spostati dalla memoria principale alla memoria secondaria
- Short-term scheduling: decide quale processo deve essere eseguito sulla CPU
- I/O scheduling: decide quale processo deve essere eseguito sul dispositivo di I/O

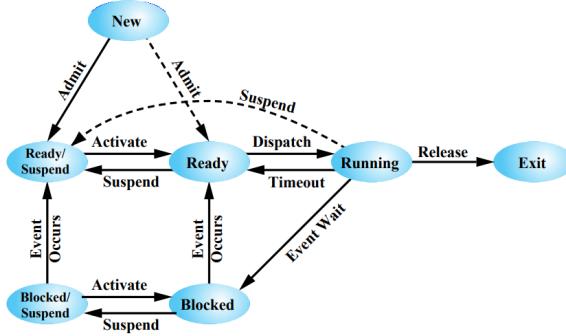


Figure 8: Tipi di scheduling

abbiamo visto che si di ready che di blocked ci sono le versioni suspended (sono in memoria secondaria), molte delle transizioni sono dovute allo scheduler, quello che decide se un processo appena creato Ready o Ready Suspended é il long-term scheduler, il medium-term decide tra le versioni suspended e non suspended, il short-term scheduler decide se un processo é Ready o Running, il I/O scheduler decide se un processo é Blocked o Ready

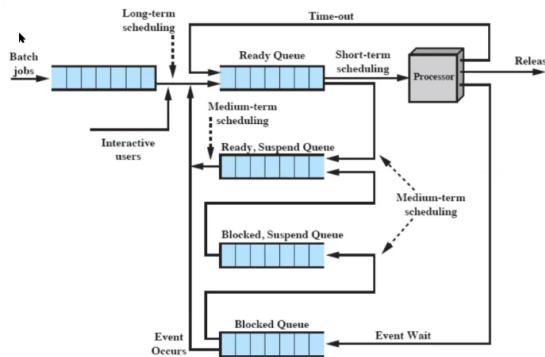


Figure 9: Tipi di scheduling

lo scheduling fa una distinzione tra processi interattivi e processi batch, i processi in batch hanno una coda, mentre quelli interattivi cercano direttamente di essere eseguiti, Il long-term come detto sopra decide se i processi sono messi in ready o ready suspended, il medium term scheduler decide se un processo é messo in ready o ready suspended oppure blocked o blocked suspended e viceversa, il short term scheduler decide se un processo é messo in running o blocked

### 3.2 Long-term scheduling

Decide quali programmi sono ammessi nel sistema, spesso segue una logica FIFO (First In First Out), spesso é un FIFO "Corretto", tenendo conto di criteri come la priorità, il tempo di esecuzione, ecc.

Inoltre controlla il grado di multiprogrammazione, ovvero il numero di processi che possono essere eseguiti contemporaneamente, il grado di multiprogrammazione é il numero di processi che possono essere eseguiti contemporaneamente,

Piú processi ci sono, piú é piccola la percentuale di tempo per cui ogni processo viene eseguito.

Le strategie di scheduling sono:

- i lavori batch vengono accodati e il LTS li prende man mano che il processore é libero
- I lavori interattivi vengono ammessi fino a saturazione del sistema
- Se si sa quali processi sono I/O bound e quali sono CPU bound, si possono fare scelte migliori facendo un giusto mix tra due tipi
- Se si sa quali processi fanno richieste a quali dispositivi, di I/O, fare in modo di bilanciare le richieste

Il Long-term scheduler puó essere chiamato anche quando non ci sono nuovi processi, ad esempio quando termina un processo oppure quando un processo é idle da troppo tempo.

### 3.3 Medium-term scheduling

Il medium-term scheduler decide se bisogna fare degli aggiustamenti tra RAM e Disco, il motivo principale é quello di gestire la multiprogrammazione, se terminano 10 processi ad esempio, il medium term scheduler decide quali processi ready presenti in memoria secondaria devono essere spostati in memoria principale, il medium term scheduler é chiamato anche

### 3.4 short-term scheduling

Il short-term scheduler é chiamato anche dispatcher, decide quale processo deve essere eseguito sulla CPU, ed é quello eseguito piú frequentemente. viene invocato in seguito a:

- interruzioni di clock
- interruzioni di I/O
- chiama di sistema
- segnali

Lo scopo dello short-term scheduler é quello di allocare tempo di esecuzione tra i processori, per ottimizzare il comportamento dell'intero sistema, per valutare quindi una politica di scheduling bisogna valutare:

- Utente: tempo di risposta, tempo di esecuzione
- Sistema: throughput, efficienza del processore

L' altra categoria é quella se sono criteri se sono criteri prestazionali (quantitativi) oppure quelli non prestazionali (qualitativi) che sono piú difficili da misurare.

### 3.4.1 Criteri Utente

Prestazionali :

- **Turnaround time:** tempo tra la creazione e la terminazione di un processo
- **Tempo di risposta:** tempo tra la creazione e la prima risposta
- **Deadline:** tempo entro il quale un processo deve essere completato

Non Prestazionali:

- **Predictability:** quanto è prevedibile il tempo di esecuzione

### 3.4.2 Criteri di sistema

Prestazionali:

- **Throughput:** numero di processi completati in un certo intervallo di tempo
- **Efficienza del processore:** percentuale di tempo in cui il processore è utilizzato

Non Prestazionali:

- **Equità:** tutti i processi devono avere la stessa possibilità di essere eseguiti
- **Enforces priorities:** i processi con priorità più alta devono essere eseguiti prima
- **Balancing resources:** bilanciare l'uso delle risorse

## 3.5 Turnaround time

Il turnaround time è il tempo tra la creazione e la terminazione di un processo, comprende i vari tempi di attesa (I/O, CPU, ecc.) si usa spesso per processi non interattivi.

## 3.6 Tempo di risposta

Il tempo di risposta è il tempo tra la creazione e la prima risposta, è importante per i processi interattivi (es. un utente che clicca su un bottone) lo scheduler in questo caso ha un duplice obiettivo per lo scheduler: minimizzare il tempo di risposta medio e massimizzare il numero di utenti che hanno un risposta veloce.

## 3.7 Deadline

La deadline è il tempo entro il quale un processo deve essere completato, è importante per i processi real-time, un buon dispatcher deve massimizzare il numero di scadenze rispettate, per quanto riguarda invece la Predictability se lancio tante volte lo stesso processo, il tempo di esecuzione deve essere sempre lo stesso, altrimenti il dispatcher non è prevedibile.

## 3.8 Throughput

Il throughput è il numero di processi completati in un certo intervallo di tempo, ovviamente l'obiettivo è massimizzare il throughput, è una misura di quanto lavoro viene effettuato.

## 3.9 Utilizzo del processore

L' utilizzo del processore è la percentuale di tempo in cui il processore è utilizzato, ovviamente l' obiettivo è massimizzare l' utilizzo del processore, quindi il processore non deve mai essere inattivo, questo è un criterio molto costoso condiviso tra più utenti.

### 3.10 Bilanciamento delle risorse

Lo scheduler deve bilanciare l'uso delle risorse, ad esempio se un processo è CPU bound, non ha senso assegnargli un tempo di I/O, quindi lo scheduler deve bilanciare l'uso delle risorse, quindi processi che utilizzano meno le risorse attualmente più usate devono essere favoriti.

### 3.11 Fairness e Priorità

La fairness è il concetto che tutti i processi devono avere la stessa possibilità di essere eseguiti, per cui non sono presenti favoritismi, a meno che non ci siano priorità, in questo caso i processi con priorità più alta devono essere eseguiti prima, inoltre questo causerà la creazione di code di processi, quindi lo scheduler deve essere in grado di gestire le code di processi.

### 3.12 Priorità e Starvation

La priorità ha un problema, ovvero che può indurre starvation, esempio: se un processo ha una bassa priorità, potrebbe non essere mai eseguito, quindi lo scheduler deve evitare lo starvation, ovvero che un processo non venga mai eseguito, per evitare lo starvation si può usare la politica di aging, ovvero aumentare la priorità di un processo che non viene mai eseguito.

### 3.13 Politiche di scheduling

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
<b>Selection function</b>	$\max[w]$	constant	$\min[s]$	$\min[s - e]$	$\max\left(\frac{w + s}{s}\right)$	(see text)
<b>Decision mode</b>	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
<b>Throughput</b>	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
<b>Response time</b>	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
<b>Overhead</b>	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
<b>Effect on processes</b>	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
<b>Starvation</b>	No	No	Possible	Possible	No	Possible

Figure 10: Politiche di scheduling

Raramente si usa una sola politica di scheduling, si usano piú politiche di scheduling, inoltre non si utilizzano gli algoritmi di scheduling implementate, ma revisioni di esse.

### 3.14 Selection Function

La selection function é quella che sceglie effettivamente quale processo deve essere eseguito, la scelta viene fatta in base a vari criteri:

- $w$  = tempo di attesa
- $e$  = tempo trascorso in esecuzione
- $s$  = tempo totale richiesto (stimato), incluso quello già servito ( $e$ )

### 3.15 Decision Mode

Specifica in quali istanti di tempo la funzione di selezione viene invocata, ci sono 2 possibili modi:

- Non pre-emptive: la funzione di selezione viene invocata solo quando il processo in esecuzione termina
- Pre-emptive: la funzione di selezione viene invocata ad intervalli regolari

#### 3.15.1 Pre-emptive - Non pre-emptive

Non pre-emptive: se un processo é in esecuzione, allora arriva o fino a terminazione o fino ad un I/O, non gli tolgo il processore

Pre-emptive: Il sistema operativo puó interrompere un processo in esecuzione per eseguire un altro processo, in questo caso il processo passera da running a ready, la preemption di un processo puó avvenire o pre l'arrivo di nuovi processi (appena creati) o per un interrupt di I/O, oppure per interrupt di clock, quest'ultimo é periodico per evitare che alcuni processi monopolizzino il processore.

### 3.16 ESEMPIO

Scenario Comune :

Processo	Tempo di arrivo	Tempo di esecuzione
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

### 3.17 First Come First Served

Tutti i processi sono aggiunti alla coda ready, é non pre-emptive, quando il processo ha finito di essere eseguito, si passa al processo che ha aspettato di piú nella coda.

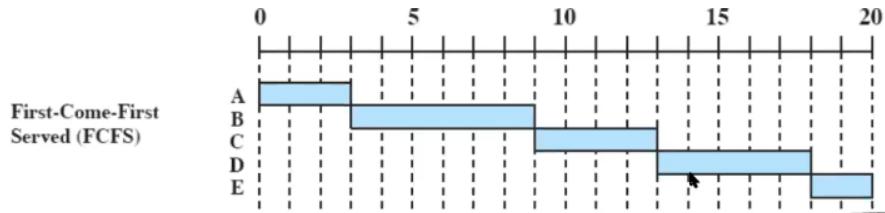


Figure 11: First Come First Served

L' algoritmo é molto semplice, é anche molto equo, dal punto di vista del sistema operativo, un processo corto deve aspettare che altri processi terminino, favorisce i processi CPU bound, per cui degenera perché un processo monopolizza il processore.

### 3.18 Round Robin

Processo	Tempo di arrivo	Tempo di esecuzione
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Il round robin usa la preemption, basandosi su un clock, Talvolta chiamato time slicing, perché ogni processo ha una fetta di tempo, é un algoritmo di scheduling per processi interattivi.

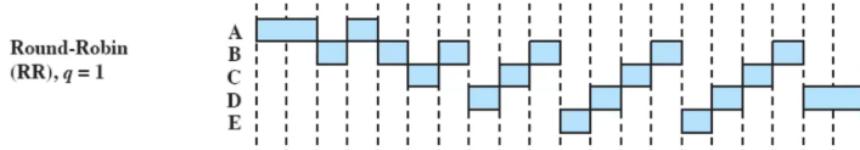


Figure 12: Round Robin

in questo caso il processo E finisce prima rispetto all'algoritmo FCFS, nel caso ci sia una coda si utilizza una politica FIFO, quando un processo finisce il tempo di esecuzione viene riaggiunto alla coda

**Quanto di tempo** é un intervallo di tempo che viene assegnato ad ogni processo e rappresenta il tempo per il quale il processo ha accesso al processore, il quanto di tempo deve essere non troppo piú grande del tipico tempo di interazione di un processo.

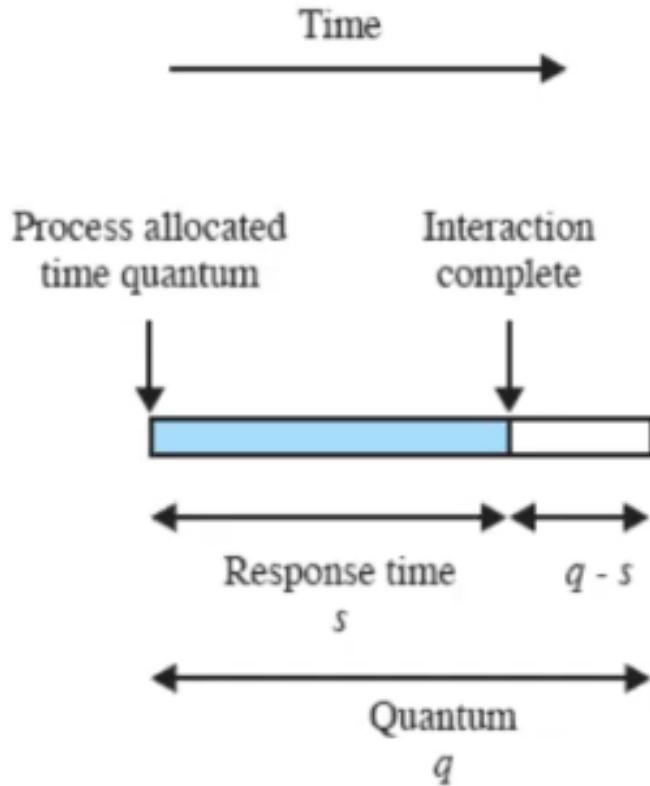


Figure 13: Quanto di tempo piú grande del tempo di interazione

Se invece uno sceglie il quanto di tempo piú piccolo, il processo che va in esecuzione avrebbe bisogno di piú tempo del quanto, prima della risposta gli viene tolto il processore, questo significa che il tempo di risposta é piú lungo.

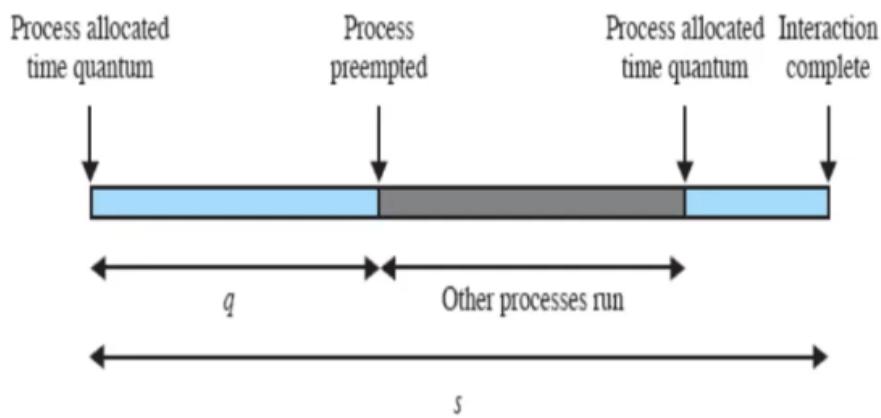


Figure 14: Quanto di tempo piú piccolo del tempo di interazione

Nel momento in cui si sceglie di usare il round robin, si fa una analisi statistica di quantitá di tempo di esecuzione necessaria per i processi per assegnare il quanto di tempo, invece se assegno un quanto di tempo troppo grande, allora il round robin diventa un FCFS.

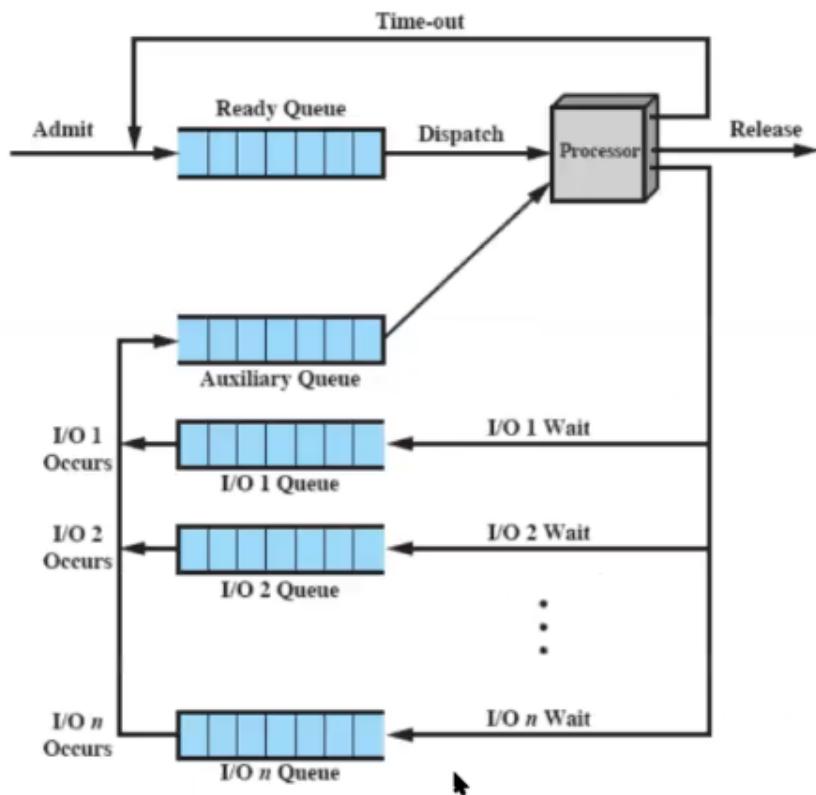


Figure 15: Round Robin Virtuale

### CPU bound vs I/O bound

C'è un problema con il round robin, per quanto riguarda i processi CPU bound contro i processi I/O bound, anche con il round robin anche i processi CPU bound vengono favoriti, perché i processi CPU bound utilizzano tutto il quanto di tempo, mentre i processi I/O bound non utilizzano tutto il quanto di tempo in caso di richiesta bloccante, dal punto di vista dell'equità non va bene, è stata proposta una soluzione **Round Robin Virtuale** che funziona come il round robin, ma se un processo fa una richiesta bloccante, allora il processo non va in una coda dei ready dopo aver completato la richiesta di I/O come succede normalmente, invece con il round robin virtuale esiste una coda ausiliaria, che accoda i processi che sono stati blocked, dopo di che il dispatcher sceglie prima la coda ausiliaria e poi la coda dei ready. Da notare che scegliendo dalla coda ausiliaria, rimando i processi in esecuzione solo per il quanto di tempo che gli rimaneva

### 3.19 Shortest Process Next

Il Shortest Process Next è un algoritmo non pre-emptive, per implementarlo è necessario sapere quanto tempo di esecuzione il processo richiede, la logica per scegliere il prossimo processo è quello col tempo di esecuzione più breve, questo fa sì che i processi corti scavalchino i processi più lunghi.

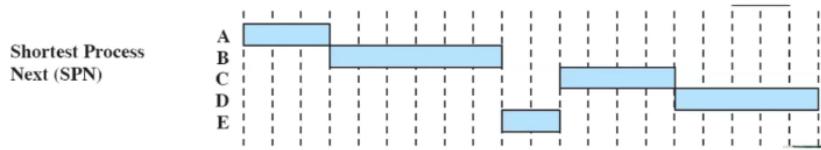


Figure 16: Shortest Process Next

Il problema é quello di dover dare una stima del tempo di esecuzione dei processi, ma anche assumendo di avere una buona stima, dal punto di vista dei criteri utente, la predictability di processi lunghi é ridotta, questo addirittura potrebbe creare starvation, perché i processi corti vengono sempre eseguiti mentre i processi lunghi potrebbero rimanere in attesa per sempre, se il tempo stimato é sbagliato il sistema operativo potrebbe abortire il processo.

### Stimare il tempo di esecuzione

Per stimare il tempo di esecuzione, ci sono alcuni processi che vengono eseguiti piú volte, quindi per questo tipo di processi si puó guardare il passato per prevedere il futuro, ad esempio facendo una media dei tempi di esecuzione passati,

$$S_{n+1} = \frac{1}{n} \sum_{i=1}^n T_i \quad (1)$$

per fare questo significa che il dispatcher deve tenere traccia del tempo di esecuzione di ogni processo, questo potrebbe richiedere molta memoria, l'altro modo é quello di ricordarmi solo l'ultimo tempo di esecuzione e l'ultima stime con :

$$S_{n+1} = \frac{1}{n} T_n + \frac{n-1}{n} S_n \quad (2)$$

questa si puo' generalizzare con un parametro  $\alpha$ , dove  $\alpha$  é un valore tra 0 e 1 otteniamo:

$$S_{n+1} = \alpha T_n + (1 - \alpha) S_n \quad (3)$$

formula media esponenziale:

$$S_{n+1} = \alpha T_n + \dots + (1 - \alpha)^i T_{n-i} + \dots + (1 - \alpha)^n S_1 \quad (4)$$

Piú  $\alpha$  é vicino a 1, piú sparisce velocemente il passato, questo serve a capire che con un buon  $\alpha$  si possono fare delle ottime previsioni.

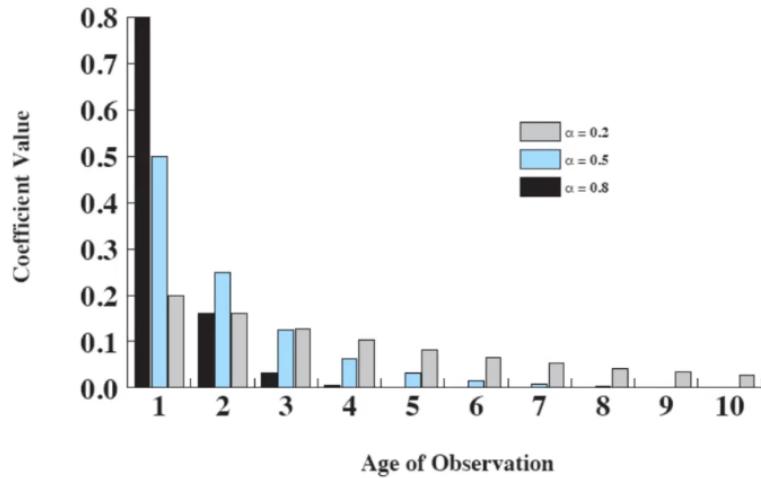


Figure 17: Analisi di Alpha

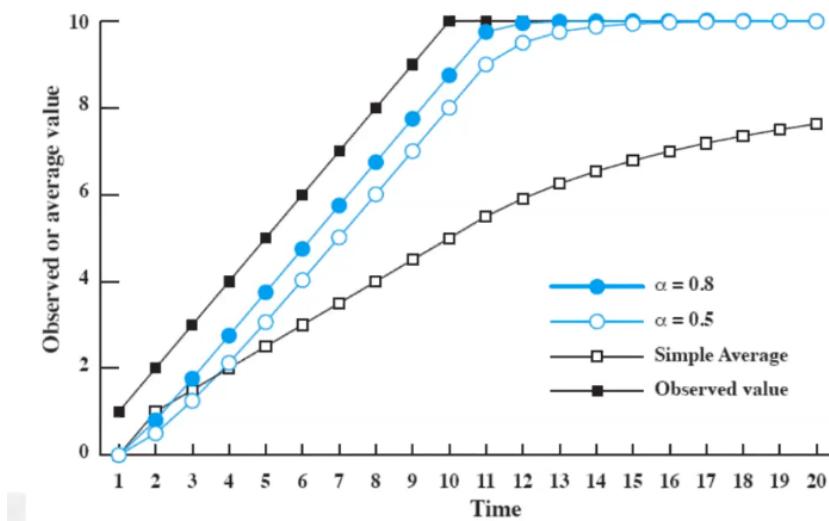
**Esempio**

Figure 18: Esempio 1

Abbiamo fissato un processo e diciamo che la sua prima istanza cresce e poi si stabilizza, se io facessi semplicemente la media, quello che la media predirebbe sarebbe un valore lontano dall'obiettivo, invece se uso degli  $\alpha$  fissi, siamo molto più vicini alla curva reale.

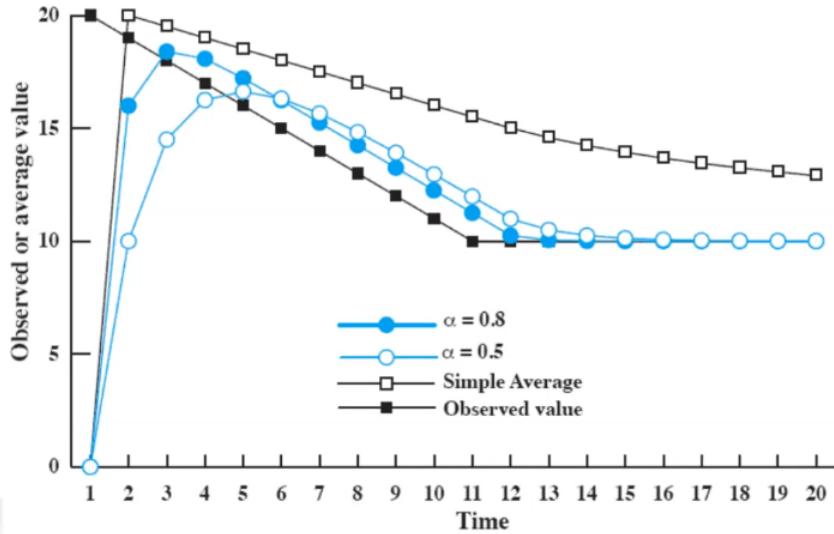


Figure 19: Esempio 2

Praticamente dopo un certo tempo, i valori vecchi vengono dimenticati, specialmente per  $\alpha$  grandi

### 3.20 Shortest Remaining Time

Lo Shortest Remaining Time é una versione pre-emptive dello Shortest Process Next, é pre-emptive sulla base che arrivi un nuovo processo, quindi io stimo il tempo rimanente per l'esecuzione, se il tempo rimanente di un processo in coda é minore del tempo di esecuzione del processo in esecuzione, allora il processo in esecuzione viene pre-empted.

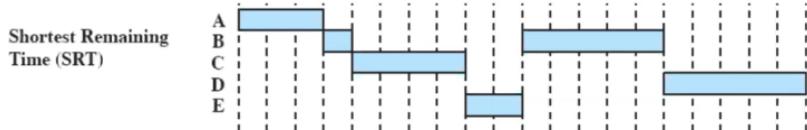


Figure 20: Shortest Remaining Time

I processi lunghi comunque possono soffrire di starvation, perché i processi corti vengono sempre eseguiti, anche in questo caso é necessario sapere il tempo di esecuzione.

### 3.21 Highest Response Ratio Next

L' algoritmo Highest Response Ratio Next é un algoritmo non pre-emptive, é una versione migliorata dello Shortest Process Next, che risolve il problema dello starvation, é basato sul tempo di attesa, il tempo di esecuzione e il tempo totale richiesto, é un compromesso tra quanto tempo sto aspettando e quanto tempo ci metto ad eseguire.

L'algoritmo massimizza il seguente rapporto:

$$\frac{w + s}{s} = \frac{\text{tempotrascorsoinattesa} + \text{tempototalerichiesto}}{\text{tempototalerichiesto}} \quad (5)$$



Figure 21: Highest Response Ratio Next

\*Diagramma Riassuntivo

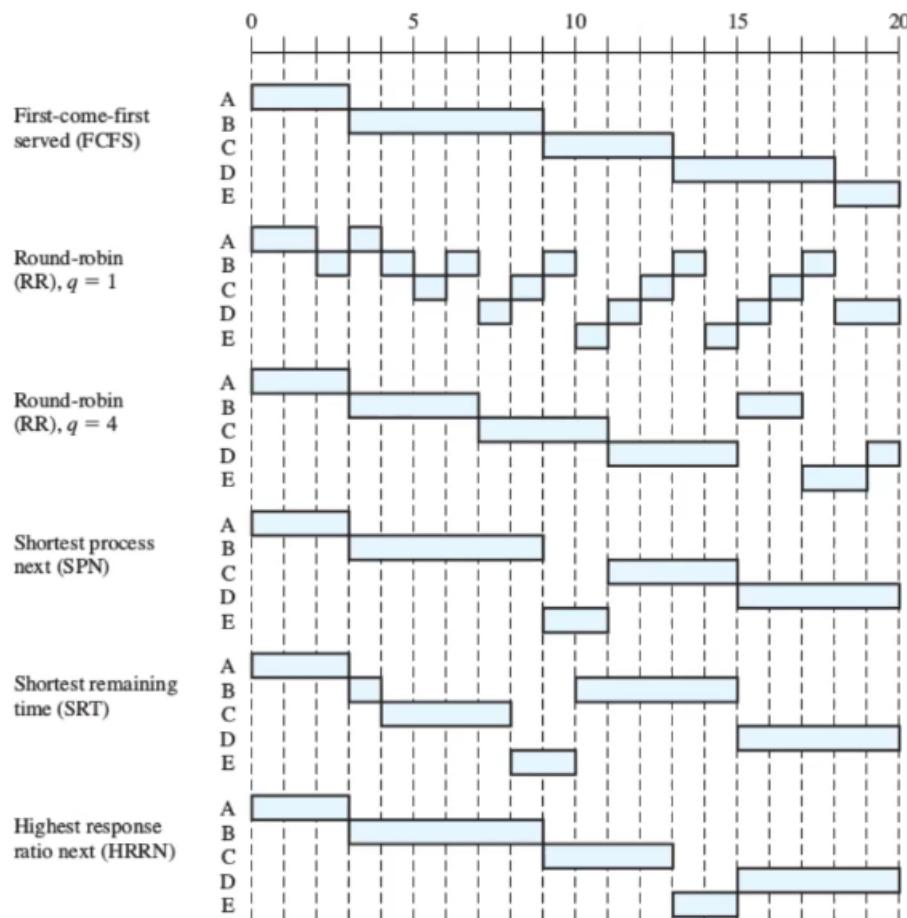


Figure 22: Diagramma Riassuntivo

### 3.22 Scheduling in Unix

Unix combina priorità e round robin, un processo quindi resta in esecuzione per al massimo un secondo, a meno che non termini o si blocchi, esistono diverse code a seconda della priorità; all'interno di ciascuna coda viene eseguito il round robin. Le priorità vengono ricalcolate ogni secondo, più un processo resta in esecuzione, minore sarà la sua priorità quando viene rimesso in coda(feedback). Le priorità iniziali vengono assegnate in base al tipo di processo :

- swaper (alta)
- controllo di un dispositivo di I/O a blocchi
- gestione di file
- controllo di un dispositivo di I/O a caratteri

$$Base_A = Base_B = Base_C = 60, nice_A = nice_B = nice_C = 0$$

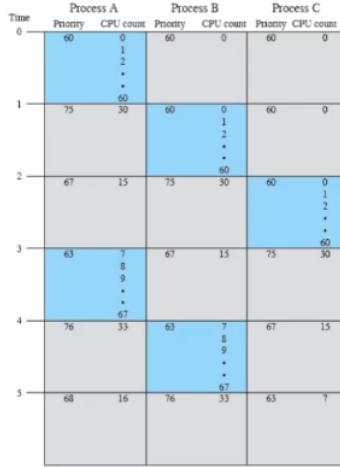


Figure 23: Esempio Scheduling Unix

- processi utente (basso)

la formula per lo scheduling é

$$[H]CPU_j(i) = \frac{CPU_J(i-1)}{2} \quad (6)$$

che viene poi utilizzata nella formula per il calcolo della prioritá

$$[H]P_j(i) = Base_{Priority_j} + CPU_j(i) + Nice_j \quad (7)$$

- **J** é un indice che indica i processi **ready**
- **Base\_Priority\_j** é la prioritá iniziale del processo (da 0 a 4)
- **Nice\_j** Ricordano che il valore della prioritá piú é alto piú la prioritá é bassa, l'idea dietro di  $nice_j$  è la capacità di un processo di auto-declassarsi, è usata nei processi di sistema. **CPU\_j(i)** È una misura di quanto tempo il processore ha passato nel tempo  $i$  eseguendo il processo  $j$ .

### Esempio

Supponiamo che ci siano 3 processi, A, B e C, con il nice = 0 e la base priority = 60, e che il processo A sia in esecuzione quindi si comincia da 60 e poi bisogna aggiungere  $CPU_j$  che inizialmente è 0, che non è  $\frac{30}{2}$  (applicare la formula), finito il tempo di esecuzione, si calcola la nuova priorità ( $P_j$ ) e si rimette in coda, dopodiché  $\frac{60}{2} = 15(8)$

$$P_j = 60 + \frac{CPU_j}{2} + 0 = 75 \quad (9)$$

questa operazione viene fatta per tutti i processi ogni secondo. Il Round robin é virtuale, quindi se un processo é bloccato, non viene messo in coda, ma viene messo in una coda ausiliaria, e viene eseguito prima di quelli in coda quando torna disponibile.

### 3.23 Architetture Multicore

Ci sono diversi modi per avere piú modi per avere piú core:

- **Cluster** : ogni processore ha la propria RAM

- **Processori specializzati:** un processore per le operazioni di I/O, un processore per le operazioni di calcolo
- **Multi-processore:** Condividono la stessa RAM, un solo sistema operativo controlla tutto

Noi ci concentreremo sui multi-processori, nel caso di un sistema mono-processore, ho n processi e decido quale di essi va in esecuzione, nel caso di un sistema multi-processore, ho m processi e m processori, quindi devo decidere se devo fare un Assegnamento Statico o Dinamico.

### Assegnamento Statico

Con l'assegnamento statico, ad ogni processo viene assegnato un processore, per tutta la sua durata andrà in esecuzione su quel processore, si può anche usare uno scheduler per ogni processore, i vantaggi sono che è facile da implementare, ma il problema è che un processore potrebbe rimanere idle.

### Assegnamento Dinamico

Per migliorare lo svantaggio dello statico, un processo, nel corso della sua vita, potrà essere eseguito su diversi processori, il sistema operativo potrebbe essere sempre eseguito su un processore fisso, questa cosa è semplice da realizzare mentre solo i processi utenti possono essere assegnati a processori diversi, un'altra scelta è quella di eseguire il sistema operativo su tutti i processori ma questo causa più overhead.

### 3.24 Scheduling in Linux

Linux cerca la velocità di esecuzione, tramite semplicità, per questo non esiste long-term e medium-term scheduler (non ha senso perché non esistono processi suspended), Un embrione del long-term c'è perché quando creo un processo il sistema potrebbe essere già saturo.

### Come Funziona

Ci sono le runqueue (Code dei Ready), e le wait queues (code dei blocked), Le wait queues sono condivise tra i processori, mentre le runqueue sono separate, ogni processore ha la sua runqueue. Essenzialmente lo scheduling è derivato da quello di UNIX, quindi è pre-emptive, a priorità dinamica, ma con importanti correzioni :

1. essere veloce, ed operare quasi in O(1)
2. servire in modo appropriato i processi real-time

Linux istruisce l'hardware di mandare un timer interrupt ogni 1ms:

- più lungo abbiamo problemi per applicazioni real-time
- più corto arrivano troppi interrupt, per cui abbiamo tanto tempo speso in Kernel Mode e quindi meno tempo per i processi utenti

perciò il quanto di tempo per ciascun processo è un multiplo di 1ms.

Linux considera tre tipi di processi:

- **Real-time** : hanno priorità fisse, e vengono eseguiti prima di tutti gli altri
- **Interattivi** : hanno priorità dinamiche, e vengono eseguiti dopo i real-time
- **Batch** : hanno priorità dinamiche, e vengono eseguiti dopo gli interattivi

## Interattivi

Non appena si agisce sul mouse o sulla tastier, é importante dare lor la CPU in 150ms al massimo

## Batch

Lo scheduler puó decidere di penalizzare i processi batch, perché non sono interattivi, quindi non c'è bisogno di dare un feedback immediato all'utente.

## Real-time

Gli unici riconosciuti come tali da Linux: Il loro codice sorgente usa la system call **sched\_setscheduler**, per gli altri usa un'euristica, esempi di sistemi real-time sono riproduttori di audio e video, controllori, ... ma normalmente sono usati dai KLT.

## Classi di scheduling

Linux ha 3 classi di scheduling:

- **SCHED\_FIFO** e **SCHED\_RR** : fanno riferimento ai processi real-time
- **SCHED\_OTHER** : fanno riferimento a tutti gli altri processi

Prima si eseguono quelli che sono in SCHED\_FIFO e SCHED\_RR, poi quelli in SCHED\_OTHER, le prime due classi hanno un livello di priorità che va da 1 a 99, mentre la terza classe ha un livello di priorità che va da 100 a 139, quindi ci sono 140 runqueue per ogni cpu, si passa dal livello n al ilvello al livello n+1 solo se o non ci sono processi in n, o nessun processo in n é in RUNNING.

La preemption puó essere dovuta a:

- si é esaurito il quanto di tempo
- un altro processo passa da blocked a RUNNING, questo succede quando c'è un processo interattivo bisogna cercare di eseguirlo il prima possibile

Molto spesso, il processo che é appena diventato eseguibile sará quello eseguito dal processore, questo é dovuto al fatto che il processo é stato bloccato per un I/O, quindi é probabile che sia un processo interattivo.

## Regole generali

Un processo SCHED\_FIFO viene non solo preempted, ma anche rimesso in coda solo se:

- si blocca per I/O
- un processo passa da uno degli stati blocked a RUNNING, ed ha prioritá maggiore

Un processo SCHED\_RR viene preempted per i motivi dello SCHED\_FIFO, ma anche se il quanto di tempo é scaduto (RR = RoundRobin).

I processi real-time hanno una prioritá fissa, e non possono essere preempted da processi con prioritá dinamica, invece i processi SCHED\_OTHER si con un meccanismo simile a quello di UNIX, inoltre per sistemi multiprocessore esiste una routine per distribuire il carico.

## 4 Gestione della Memoria

La memoria é oggi a basso costo, e con trend in diminuzione, questo fa sì che le applicazioni usino sempre piú memoria, se ogni processo dovesse gestire la propria memoria, ogni processo userebbe semplicemente tutta la memoria disponibile, questo porterebbe all'assenza della multiprogrammazione che é un aspetto essenziale per il corretto funzionamento del sistema operativo, si potrebbe imporre dei limiti di memoria a ciascun processo, diventa però difficile per un programmatore scrivere un processo che rispetti tali limiti, quindi ogni sistema operativo deve avere un sistema di gestione della memoria cercando di dare l'illusione ai processi di avere tutta la memoria, la soluzione é quella di usare il disco come buffer per memoria, questa gestione di I/O é ovviamente piú lenta del processore, per cui il SO deve pianificare lo swap

### 4.1 Requisiti

- Rilocazione : importante che ci sia aiuto hardware, aiuto, non gestione diretta per cui sistema operativo e hardware collaborano
- Protezione : importante che ci sia un aiuto hardware
- Condivisione
- Organizzazione logica
- Organizzazione fisica

#### 4.1.1 Rilocazione

Il programmatore non sa e non deve sapere in quale zona della memoria il programma verrá caricato :

- potrebbe essere swappato su disco, e al ritorno in memoria principale potrebbe essere caricato in un'altra zona
- potrebbe anche non essere contiguo, oppure con altre pagine in RAM e altre in disco
- in questo contesto, si intende chi usa l'assembler o il compilatore

I riferimenti alla memoria devono essere tradotti nell'indirizzo fisico : preprocessing, run-time, se run-time occorre supporto hardware

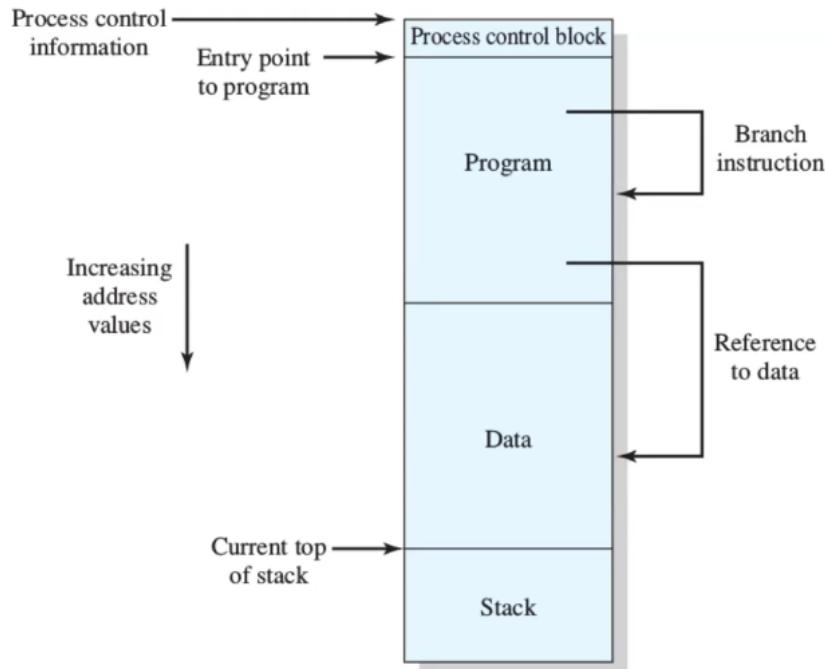


Figure 24: Rilocazione

un processo ha una zona con il programma in linguaggio macchina, e una zona con i dati, e una zona con lo stack, la sua parte iniziale è il PCB, gli indirizzi che possiamo avere sono indirizzi di salto oppure referenze a variabili, tutti questi indirizzi devono essere ricalcolati.

### Indirizzi nei programmi

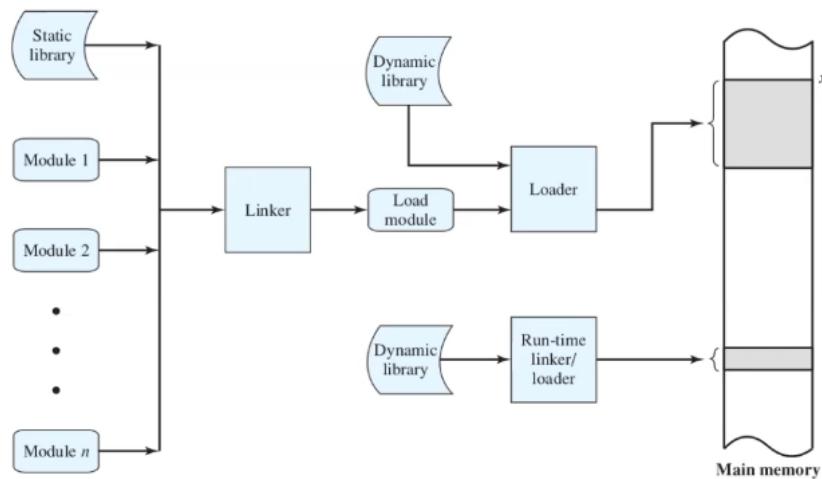


Figure 25: Indirizzi nei programmi

Per capire come avviene la rilocazione dobbiamo prima precisare... Un programma eseguibile viene prima scritto in moduli, uno di questi moduli ha il main, quindi ci sono tanti moduli scritti dal programmatore oppure librerie, ognuno di questi moduli viene compilato separatamente, ed per ogni modulo viene creato un file oggetto, tutto questo viene collegato attraverso il linker in modo da creare un file eseguibile (nell'esempio Load module), poi c'è il loader che carica il file eseguibile in memoria, nel fare questo ci potrebbe essere bisogno di alcune librerie dinamiche

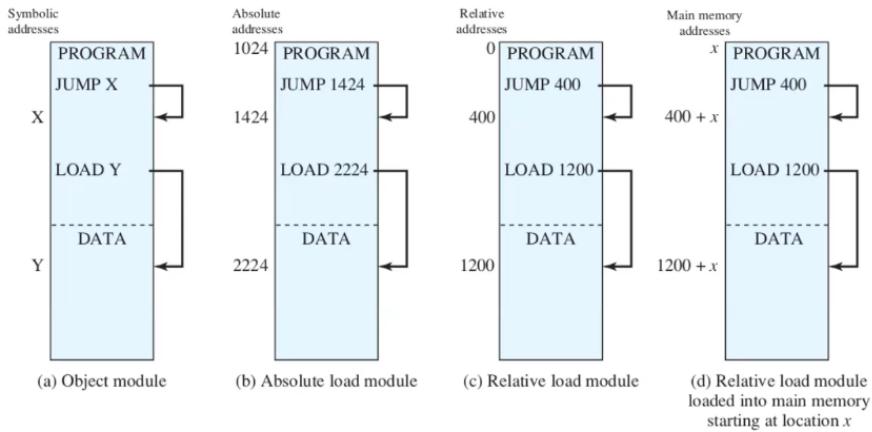


Figure 26: Effettivo in memoria

Il risultato é mostrato nell'immagine, ogni singolo modulo ha la sua parte di programma e la sua parte di dati, sostanzialmente il programma contiene soltanto indirizzi simbolici, quando però viene trasformato in un file eseguibile abbiamo 2 possibilità:

- Indirizzo Assoluto : Lui sa che deve cominciare a 1024, e se deve saltare a 1424, il loader deve caricare il programma all'indirizzo 1024 altrimenti non funziona.
- Indirizzo Relativo : Con gli indirizzi relativi, si può supporre di partire da 0, e nel caso di un salto scrivo solo l'indirizzo rispetto all'inizio del programma.

### Tipi di Indirizzi

- **Indirizzi Logici** : vengono usati dal programmatore, sono indirizzi simbolici, non sono reali, sono rilocati
- **Indirizzi Fisici** : sono gli indirizzi reali, sono quelli che vengono usati dal processore
- **Indirizzi Relativi** : il riferimento é espresso come un spiazzamento rispetto ad un punto di riferimento.

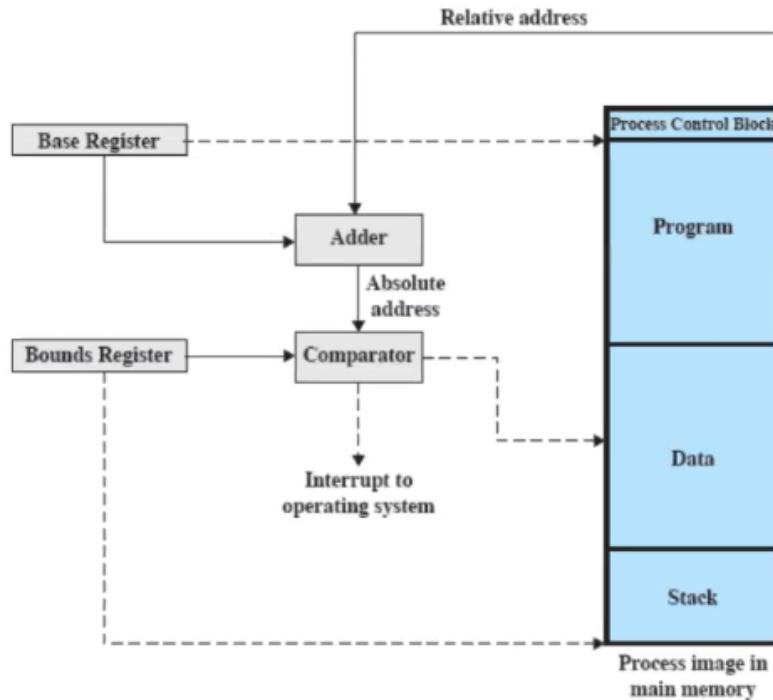


Figure 27: Tipi di Indirizzi

Nel caso di indirizzi relativi, l'hardware della macchina sa che se per esempio abbiamo un salto a 100, allora l'hardware sa che deve sommare 100 all'indirizzo base (Es. 6000), quindi l'indirizzo fisico sarà 6100, inoltre c'è una fase di controllo per rimanere nei limiti di memoria, e fondamentale che ogni volta che il sistema operativo carica il processo il sistema operativo deve preoccuparsi di mettere l'indirizzo corretto nel base register.

I registri usati sono:

- Base Register : contiene l'indirizzo base del processo.
- Limit Register : contiene l'indirizzo di fine del processo.

I valori per questi registri vengono settati nel momento in cui il processo viene posizionato in memoria, mantenuti nel PCB del processo, fa parte del passo 6 del process switch e non vanno semplicemente modificati occorre proprio modificarli.

#### 4.1.2 Protezione

I processi non devono poter accedere alla locazione di memoria di memoria di un'altro processo, a meno che non sia stato esplicitamente condiviso, A causa della rilocazione non può essere fatto a tempo di compilazione, pertanto serve un supporto hardware.

#### 4.1.3 Condivisione

La condivisione deve essere possibile, permettere a piú processi di accedere alla stessa locazione di memoria, solo se é effettivamente utile allo scopo perseguito, c'è anche casi in cui é il sistema operativo in maniera trasparente, il caso tipico é quando si eseguono piú processi eseguendo lo stesso codice sorgente, quindi lo metto in RAM una volta sola.

#### 4.1.4 Organizzazione Logica

A livello hardware, la memoria é organizzata in modo lineare, A livello software , i programmi sono scritti in moduli, per cui il SO deve offrire tali caratteristiche, facendo da ponte tra la prima visuale (moduli) e la seconda (lineare).

#### 4.1.5 Organizzazione Fisica

L' organizzazione fisica é quella che si occupa del flusso di dati tra RAM e la memoria secondari, questa non é una cosa lasciata al programmatore, se per esempio io scrivo un programma che necessitá di 1GB di ram ma il sistema operativo me ne assegna 500MB, una volta il programmatore doveva usare l'overlay per suddividere il programma in pezzi e gestire lo swap tra ram e disco in maniera manuale, oggi il sistema operativo si occupa di tutto ciò.

### 4.2 Partizionamento

Uno dei primi metodi per gestire la memoria é il partizionamento, la memoria viene divisa in partizioni, esso puó essere di diversi tipi:

- **Partizionamento Fisso** : la memoria é divisa in partizioni di dimensione fissa.
- **Partizionamento Dinamico** : la memoria é divisa in partizioni di dimensione variabile.
- **Paginazione Semplice** : la memoria é divisa in pagine di dimensione fissa.
- **Segmentazione Semplice** :
- **Paginazione con memoria virtuale** :
- **Segmentazione con memoria virtuale** :

#### 4.2.1 Partizionamento Fisso Uniforme

Quando accendo il sistema operativo, tra le cose che vengono fatte il SO divide la memoria in partizioni di dimensione fissa, 1 é riservata al kernel, le altre sono per i processi, l'idea é quella di mettere al loro interno i processi che peró non possono superare la partizione assegnata all'inizio, chiaramente il SO puó decidere se sospendere e quindi spostare il processo sul disco, in questo caso era il programmatore a dover essere sicuro di non sforare la partizione assegnata.

#### Problemi

un programma potrebbe non entrare in una partizione, questo porta anche ad un uso inefficiente della memoria, perché porta al fenomeno della frammentazione interna.

#### 4.2.2 Partizionamento Fisso Variabile

Nel partizionamento variabile comunque le partizioni vengono assegnate una sola volta, ma la dimensione delle partizioni é variabile, questo permette di mettere i processi piú leggeri in partizioni piú piccole.

#### Algoritmo di Posizionamento

Da momento in cui ho partizioni di dimensioni variabili, mi devo preoccupare di dove mettere i processi, una scelta é quella di avere una coda per partizione , oppure ho una unica coda e mano a mano assegno alla partizione che spreca meno spazio, se uso la coda unica posso fare delle

ottimizzazioni nel senso che piuttosto che non far eseguire un processo lo carico in memoria, anche se spreco un po' di memoria.

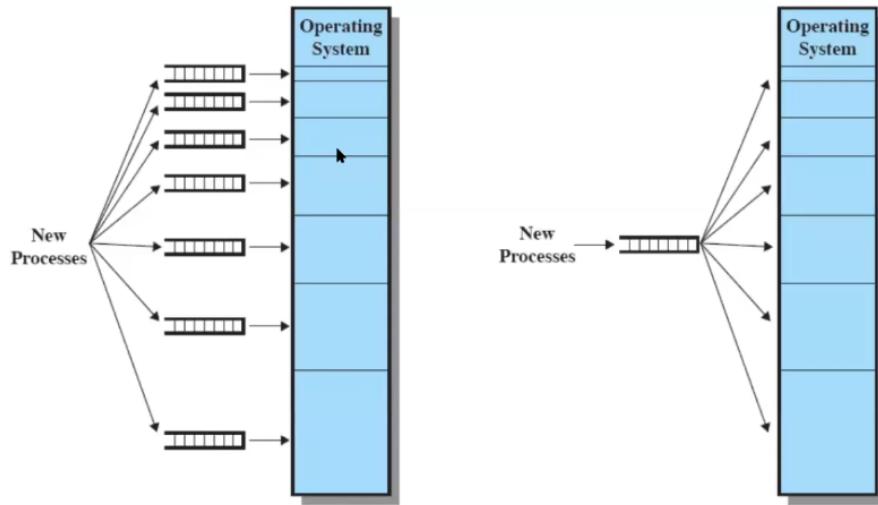


Figure 28: Partizionamento Variabile

### Problemi Irrisolti

C'è un numero massimo di processi in memoria principale dettato dal fatto che il numero di processi non può superare il numero di partizione, inoltre la gestione della memoria risulta comunque inefficiente se ho tanti processi piccoli.

#### 4.2.3 Partizionamento Dinamico

Le partizioni variano sia in misura che in quantità, la dimensione delle partizione varia in base alla dimensione del processo.

### Esempio



Figure 29: Partizionamento Dinamico

Supponiamo che arrivino in sequenza tre processi, p1=20MB, p2=14MB, p3=18MB, stiamo assumendo che chiaramente stiamo usando indirizzi relativi, in una memoria da 56M resta un blocco da 4MB, se arriva un processo da 5MB, il sistema operativo deve fare una scelta, supponiamo che il processo da 5MB sia il processo p4 e sia più importante di p2



Figure 30: Partizionamento Dinamico

quello che succede é che si lascia uno spazio vuoto di 6MB, p2 chiaramente viene copiato sul disco in attesa che venga richiamato, ora però vogliamo far eseguire p2 che é piú importante di p1 copiamo p1 sul disco e carichiamo p2 in memoria, ora abbiamo un ulteriore spazio vuoto di 6MB

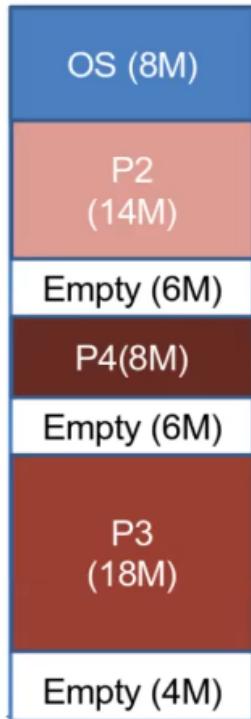


Figure 31: Partizionamento Dinamico

si nota che ci sono 16MB di spazio vuoto, se arriva un processo da 10MB, non lo posso eseguire perché la memoria non é contigua.

### Problemi

La frammentazione esterna é un problema, che però si puó risolvere con la compattazione.

se ho piú blocchi liberi, ed arriva un processo che potrebbe entrare in uno di questi blocchi, il sistema operativo utilizza un algoritmo per scegliere il blocco in cui mettere il processo, l'algoritmo puó essere:

- First Fit : metto il processo nel primo blocco che trovo
- Best Fit : metto il processo nel blocco piú piccolo che trovo
- Worst Fit : metto il processo nel blocco piú grande che trovo

### Best Fit

l'algoritmo Best Fit ad una prima valutazione potrebbe sembrare il migliore, ma in realtà é il peggiore, perché lascia tanti piccoli blocchi liberi, che non possono essere usati.

### First Fit

1. scorre la memoria dall'inizio; il primo blocco con abbastanza spazio viene usato

2. é molto veloce
3. tende a riempire solo la prima parte della memoria
4. A conti fatti era il migliore

#### 4.2.4 Next Fit

Next Fit é una variante di First Fit, la differenza é che Next Fit ricorda dove ha finito l'ultima volta, e riparte dall'ultima appena assegnata per evitare che solo la prima parte della memoria venga usata, assegna piú spesso l'ultimo blocco di memoria.

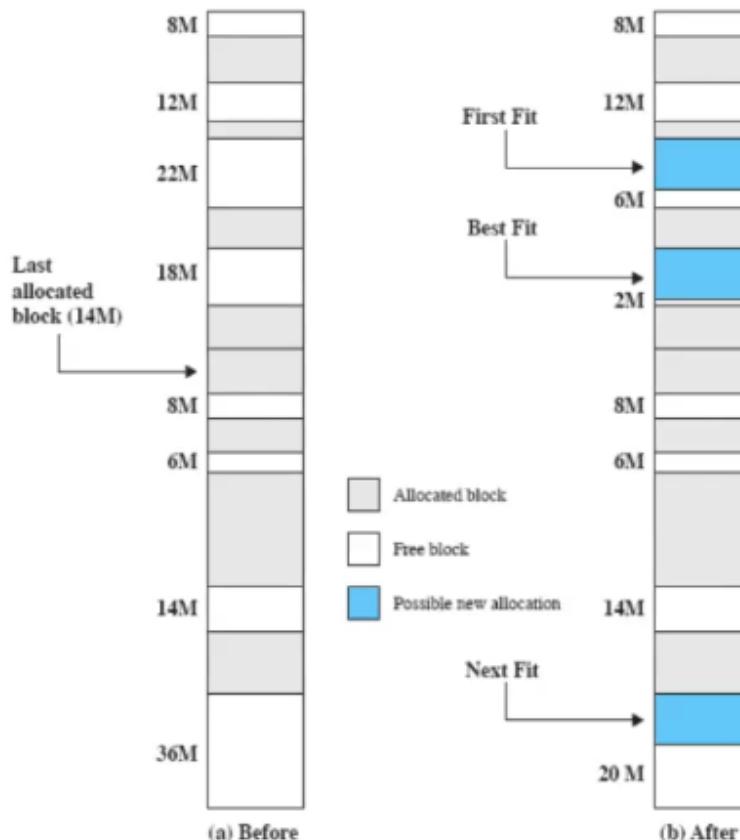


Figure 32: Confronto tra algoritmi

#### 4.2.5 Buddy System

Il Buddy system é ancora partizionamento, in questo caso é un compromesso tra partizionamento fisso e dinamico, é ancora usato nei sistemi operativi moderni, esempio : supponiamo che  $2^U$  la dimensione di memoria ad disposizione per l'utente e che sia la dimensione di un processo, quello che  $1) <= s < 2^x(10)$  Una delle 2 porzioni é usata per il processo, L' invece serve per dare un lower bound, ovvero non si potranno creare partizioni troppo piccole, quando il processo finisce, se il buddy é libero, si uniscono.

Esempio: supponiamo di avere un blocco da 1 MB, e che arrivi un processo da 100KB, quindi andiamo a dividere il blocco per 2, ed osserviamo di avere 2 blocchi da 512KB, il blocco é ancora troppo grande, ne prendiamo uno e lo dividiamo per 2, ottenendo 2 blocchi da 256KB, il blocco é ancora troppo grande, ne prendiamo uno e lo dividiamo per 2, ottenendo 2 blocchi da 128KB,

il blocco adesso è della dimensione corretta perché se divido ancora per 2 i 2 blocchi risultanti saranno troppo piccoli per ospitare il processo, ci sarà quindi una frammentazione interna di 28KB, supponiamo ora che arrivi un processo da 240KB, vediamo che dalla divisione di prima è avanzato un blocco da 256KB e quindi quella partizione viene usata, ora supponiamo che arrivi un processo da 64KB, dividiamo il blocco da 128KB e selezioniamo uno dei due blocchi da 64KB, per ricreare le partizioni quando un processo termina, l'idea è quella di riaccoppiare i blocchi con i propri buddy, quindi se un blocco da 64KB termina, si unisce con il blocco da 64KB, e tutti e due devono derivare dallo stesso blocco da 128KB e così via fino a riformare il blocco da 1MB.

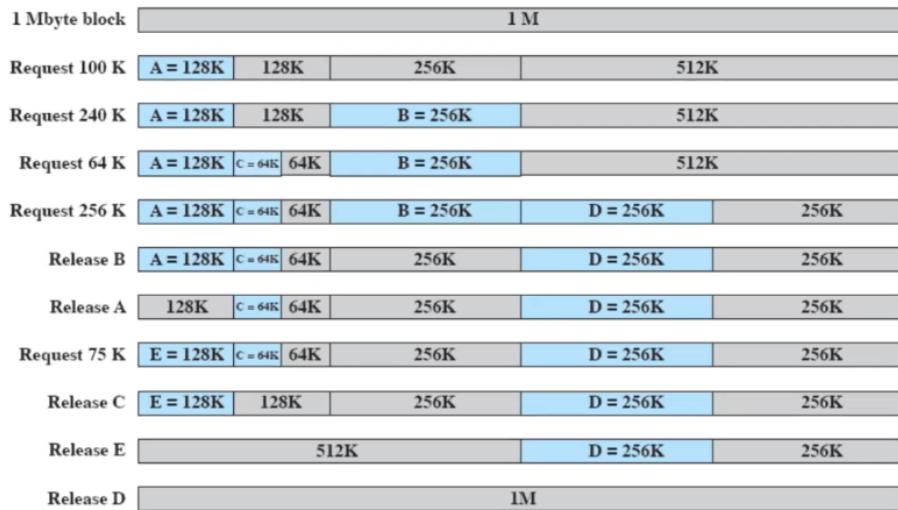


Figure 33: Buddy System

Da notare che il buddy system si presta bene per essere rappresentato come un albero binario, quindi per migliorare la ricerca si implementava la ricerca tramite binary search tree, in questo modo si evita di scorrere tutta la memoria.

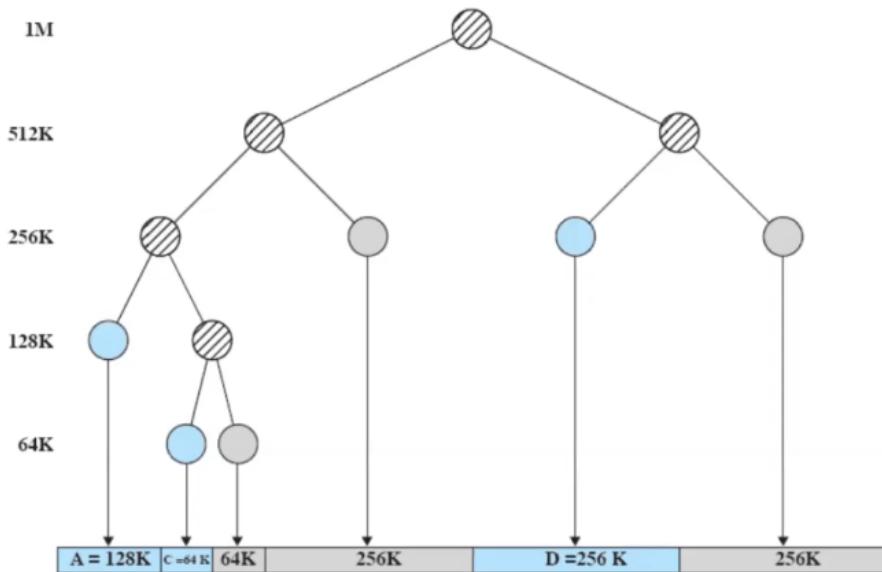


Figure 34: Buddy System

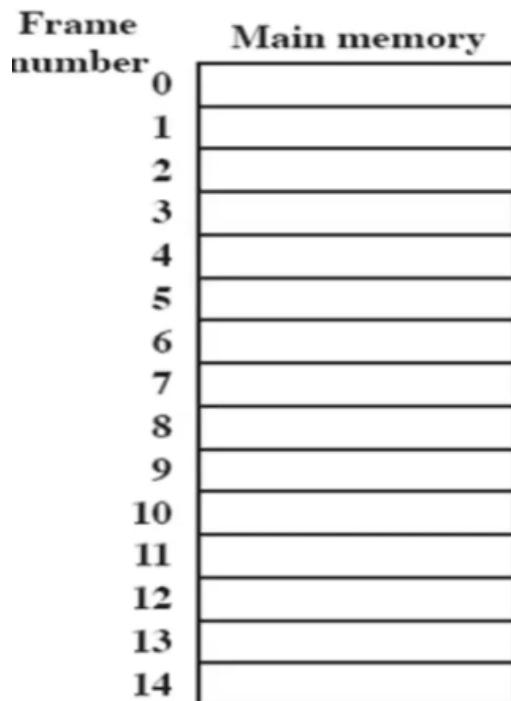
### 4.3 Paginazione

#### Paginazione Semplice

Sia la memoria che i processi vengono divisi in pezzi di dimensione fissa e piccola (1KB), questo si fa sia con la RAM che con i processi, se un processo è grande 1MB viene diviso in 1024 pezzi (1KB), questi pezzi sono chiamati pagine per i processi, mentre i pezzetti di memoria sono chiamati frame, La cosa essenziale che ogni pagina per essere utilizzata deve essere collacata in un frame, la cosa interessante che pagine contigue non devono essere collocate in frame contigui, questo permette di evitare la frammentazione interna, tutto il processo è trasparente al programmatore, a questo punto serve che i sistemi operativi mantengano una tabella che mappi le pagine ai frame, questa tabella è chiamata Page Table, questo punto però bisogna correggere gli indirizzi, per cui c'è bisogno di un supporto hardware.

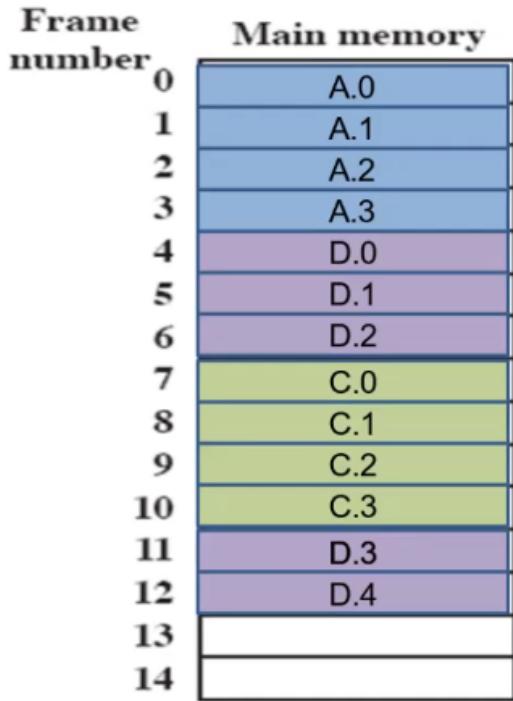
#### Esempio

Supponiamo che arrivi un processo **A** che richiede 4 frame, poi **B** da 3 frame, e poi **C** da 4 frame, poi il processo **B** termina, quello che succederebbe se fossimo in partizionamento dinamico ed arrivasse un processo da 5 dovrei eseguire la compattazione, in questo caso invece posso usare i 3 frame che erano stati assegnati a **B** per il processo **D**, ed i restanti 2 frame li posso accodare a C.



Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	

Frame number	Main memory
0	A.0
1	A.1
2	A.2
3	A.3
4	B.0
5	B.1
6	B.2
7	C.0
8	C.1
9	C.2
10	C.3
11	
12	
13	
14	



Le tabelle risultanati sono:

0 0 1 1 2 2 3 3 Process A page table	0 — 1 — 2 — Process B page table	0 7 1 8 2 9 3 10 Process C page table	0 4 1 5 2 6 3 11 4 12 Process D page table	13 14 Free frame list
--	---	---	---	-----------------------------

Figure 35: Tabelle delle pagine risultanti

#### 4.4 Segmentazione

##### segmentazione Semplice

La differenza tra segmentazione e paginazione è che la segmentazione divide i processi in segmenti di dimensione variabile, mentre la paginazione divide i processi in pagine di dimensione fissa, la differenza è che il programmatore a dover dividere il processo in segmenti (Sorgenti, Dati, ), dichiarando quali segmenti ci sono e quale la loro dimensione a caricarli in memoria ed a risolvere gli indirizzi è il sistema operativo, sempre con l'aiuto dell'hardware.

#### 4.5 Indirizzi Logici

Dobbiamo quindi considerare una rivisitazione degli indirizzi logici, con gli indirizzi relativi ad esempio, il programmatore sa che il suo programma inizia a 0, e che se deve saltare a 100, deve scrivere 100 poi è il sistema operativo che aggiunge l'offset necessari per andare all'istruzione corretta, Supponiamo quindi di avere un processo diviso in 3 pagine (notare che è presente la frammentazione interna), la prima cosa da fare quindi con un indirizzo è capire in quale pagina si trova dopo di che ho un offset rispetto all'inizio della pagina, devo andare ad usare la tabella delle pagine per capire dove si trova l'inizio del frame e sommare l'offset, il risultato è l'indirizzo

fisico, in maniera analoga funziona anche per la segmentazione, da tenere presente che i segmenti hanno dimensione variabile.

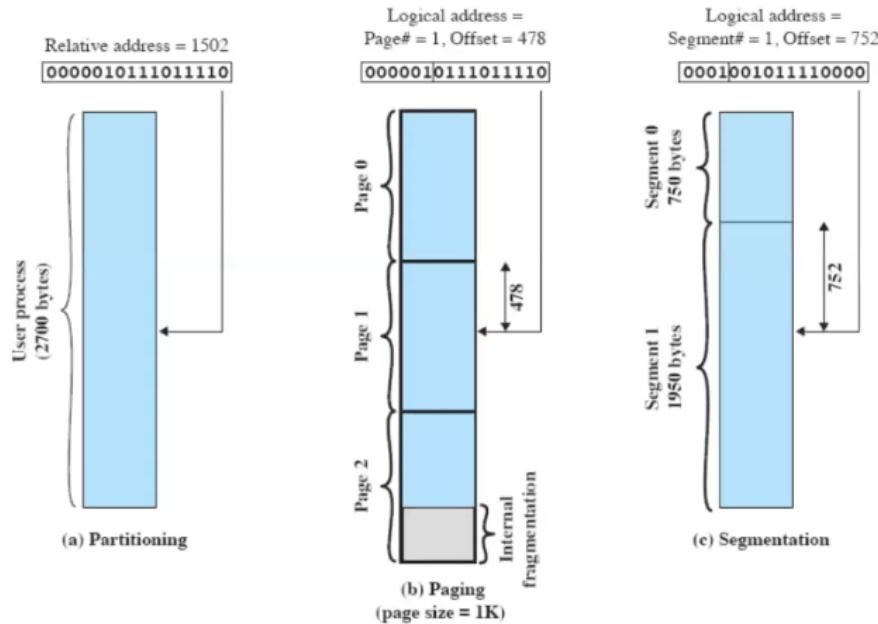


Figure 36: Indirizzi Logici

### Paginazione Esempio

Supponiamo di essere in una istruzione hardware e questa istruzione hardware ha un indirizzo di 16 bit logico, quello che devo fare è ricavare l'indirizzo fisico, le dimensioni delle pagine sono sempre un potenza di 2 allora lascio i primi 10 bit che sono usati per l'offset, mentre i 6 bit più significativi sono usati per capire in quale pagina si trova l'indirizzo questo è vero perché abbiamo preso pagine di dimensione  $2^{10}$  quindi per generalizzare se la dimensione della pagina è  $2^x$  allora gli  $x$  bit meno significativi sono usati per l'offset, mentre i bit più significativi sono usati per capire in quale pagina si trova l'indirizzo, quindi una volta trovata la pagina, vado a sostituire il contenuto della tabella all'interno dei bit che prima erano riservati alla tabella, in sostanza i bit più significativi all'inizio sono usati per contenere l'indirizzo per la tabella delle pagine che contiene i bit che servono per trovare l'indirizzo fisico.

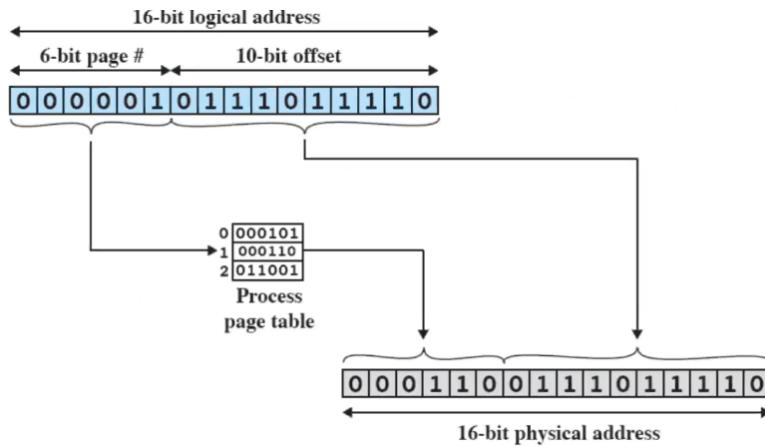


Figure 37:

### Segmentazione Esempio

Se voglio tradurre da indirizzo logico a indirizzo fisico, anche in questo caso la lunghezza massima dei segmenti é decisa dal sistema operativo, in questo caso siccome ogni segmento ha una dimensione varabile devo andare a recuperare la posizione del segmento nella tabella dei segmenti, e sommare i bit di offset, inoltre nella tabella é contenuta anche la lunghezza del segmento.

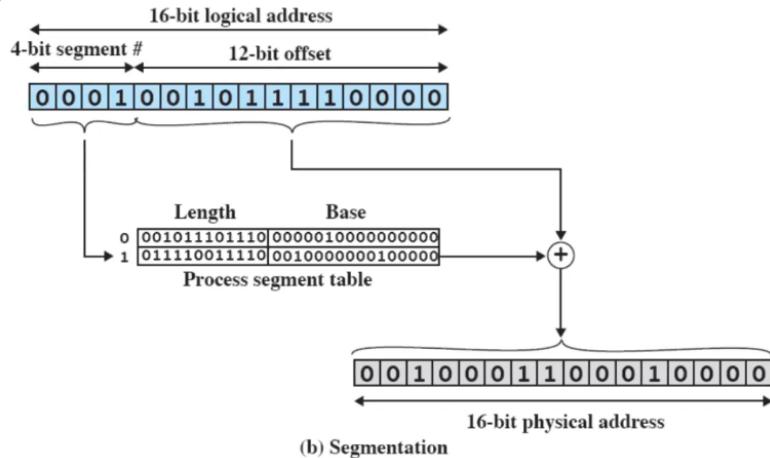


Figure 38:

### 4.6 Memoria Virtuale

I riferimenti alla memoria quindi sono indirizzi logici che devono essere tradotti in indirizzi fisici, con la paginazione e la segmentazione un processo puó essere diviso in piú parti é puó trovarsi in zone differenti della memoria, l'idea é che effettivamente non c'è la necessitá che tutto il processo sia in memoria principale, perché quello che serve effettivamente é che la parte del processo che é in esecuzione sia in memoria principale, tutto il resto puó essere sul disco, quindi il sistema operativo mette in memoria solo la parte del processo che é in esecuzione questo insieme di dati in memoria principale si chiama **resident set**, se la pagina peró non si trova in memoria principale, si verifica un **page fault**, il sistema operativo quindi chiama un interrupt che si occupa di andare a prendere la pagina mancante e metterla in memoria principale, fino a che la pagina non é in memoria principale il processo é blocked, quando effettivamente la pagina é effettivamente in memoria il processo viene sbloccato e puó continuare l'esecuzione, **quando il processo verrá eseguito bisognerá ri-eseguire l'istruzione che ha causato il page fault**.

#### conseguenze

- Ci possono essere molti processi in memoria principale perché basta una pagina in memoria principale
- Diventa molto probabile che ci sia un processo ready diminuendo l'idle del processore
- Posso eseguire un programma piú grande della memoria principale

#### Terminologia

La Memoria virtuale é uno schema di allocazione di memoria, in cui la memoria secondaria puó essere usata come se fosse memoria principale.

- Gli indirizzi usati nei programmi e quelli usati dal sistema sono diversi
- C'è una fase di traduzione automatica dai primi indirizzi (logici) ai secondi (fisici)
- La dimensione della memoria virtuale é limitata dallo schema di indirizzamento, oltre che dalla dimensione della memoria secondaria
- la dimensione della memoria principale, non influisce sulla dimensione della memoria virtuale
- **Indirizzo Virtuale** : indirizzo logico
- **Spazio degli indirizzi virtuali** : la quantitá di memoria virtuale assegnata ad un processo
- **Spazio degli indirizzi**: la quantitá di memoria assegnata ad un processo
- **Indirizzo Reale** : indirizzo fisico

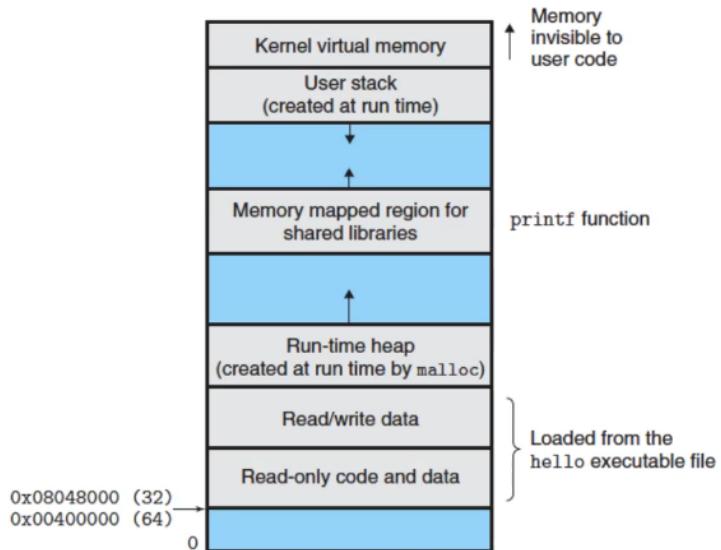


Figure 39: come un processo vede la memoria

#### 4.6.1 Trashing

Il trashing é quando il sistema operativo perde piú tempo a fare swap tra memoria principale e secondaria per caricare le pagine che a fare effettivamente il lavoro, per evitarlo il sistema operativo cerca di indovinare quali pezzi di processo saranno usati con minore o maggiore probabilitá, nel futuro prossimo, questo tentativo di divinazione avviene sulla base della storia recente, sfruttando il principio di localitá (i riferimenti tendono ad essere vicini), questo vale sia per i dati che per le istruzioni.

#### 4.6.2 Supporto Hardware

Paginazione e segmentazione devono essere supportate dall'hardware, in particolare la traduzione degli indirizzi, mentre il sistema operativo si occupa di muovere le pagine/segmenti tra memoria principale e secondaria

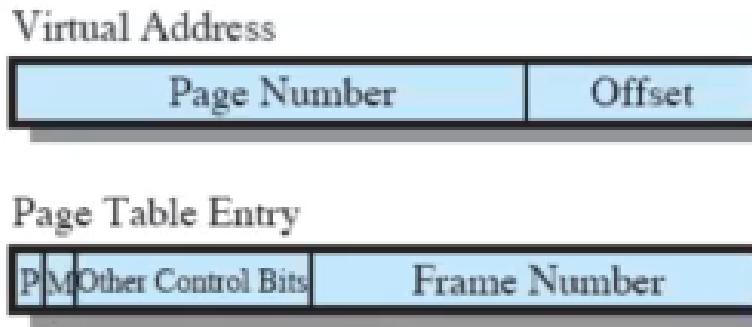


Figure 40: Page Table Entry

### Paginazione

Ogni processo ha una sua tabella delle pagine, il control block di un processo punta a tale tabella, ed ogni entry di questa tabella contiene:

- Il numero di frame in memoria principale
- NON c'è il numero di pagina, è direttamente usato per indicizzare la tabella
- un bit per indicare se è in memoria principale o meno (Bit di presenza)
- un bit per indicare se è stato modificato in seguito all'ultima volta che è stata caricata in memoria (Bit di modifica)

Per quanto riguarda invece l'hardware per la traduzione degli indirizzi, la somma non è una semplice somma: il numero di pagina va moltiplicato per il numero di bytes di ogni singola entry della tabella delle pagine, dopo di che si può fare la somma.

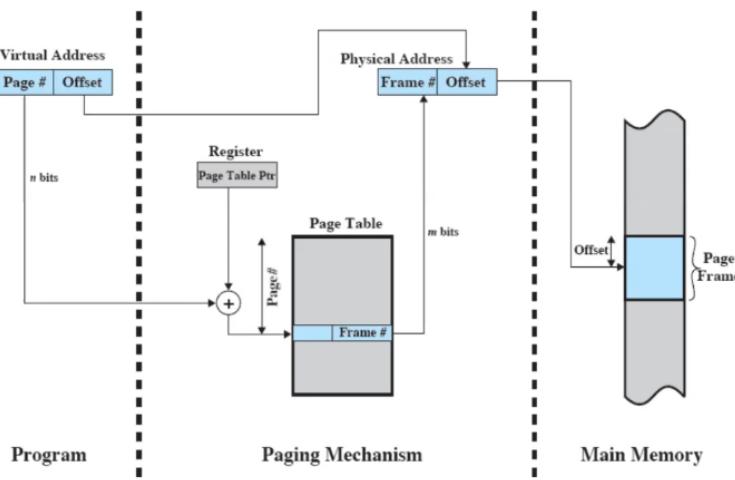


Figure 41: Hardware Paginazione

Il sistema operativo deve :

- caricare a partire da un certo indirizzo / tabella delle pagine del processo
- caricare il valore di I in un opportuno registro dipendente dall'hardware
- questo va fatto per ogni process switch quindi fa sempre parte del passo 6

### Tabelle delle pagine

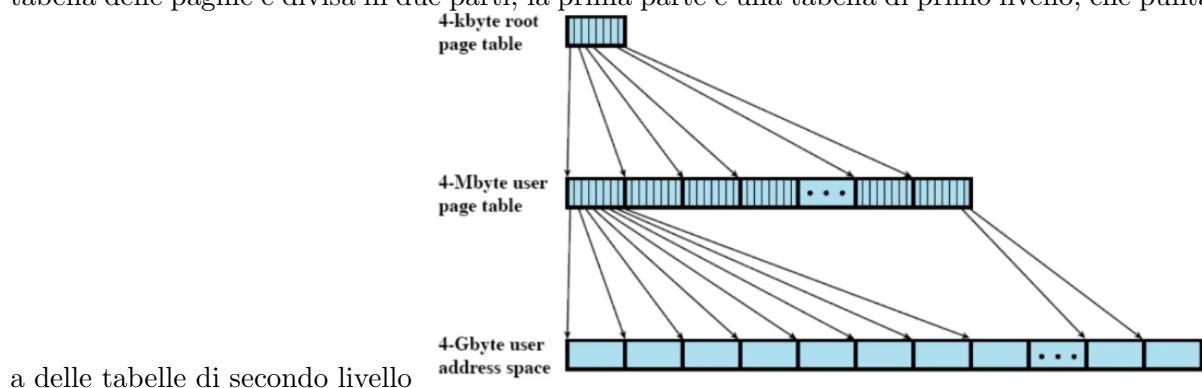
Uno dei problemi é un overhead, perché le tabelle potrebbero contenere molti elementi, quando un processo é in esecuzione , viene assicurato che almeno una parte della sua tabella sia in memoria principale.

Esempio:

- Abbiamo 8GB di spazio virtuale, 1KB per pagina,  $2^{23}$  entries per ogni tabella delle pagine, ovvero per ogni processo
- Con max 4GB di RAM, abbiamo 4 byte
- 4 byte per ogni entry  $2^{23} = 32MB$  per ogni processo quindi con 1 RAM di 1GB, bastano 20 processi per occupare più

### Tabella delle pagine a 2 livelli

Per risolvere il problema dell'overhead, si puó usare una tabella delle pagine a 2 livelli, in cui la tabella delle pagine é divisa in due parti, la prima parte é una tabella di primo livello, che punta



a delle tabelle di secondo livello

Tabella delle pagine a 2 livelli In questo caso l'indirizzo virtuale é diviso in 3 parti, la prima parte é usata per indicizzare la tabella di primo livello, la seconda parte é usata per indicizzare la tabella di secondo livello, e la terza parte é usata per indicizzare la pagina, in questo modo si riduce l'overhead.

#### 4.6.3 Translation Lookaside Buffer

IL TLB (memoria temporanea per la traduzione futura), ogni riferimento alla memoria virtuale puó generare due accessi alla memoria, si usa un cache veloce per gli elementi delle tabelle delle pagine.

#### come funziona

Dato un indirizzo virtuale, il processore esamina il TLB, se l'indirizzo é presente, il TLB restituisce l'indirizzo altrimenti si prende la normale tabella delle pagine del processo, se la pagina risulta in memoria principale, si aggiorna il TLB, se la pagina non é in memoria principale, si genera un page fault e si carica in memoria ed infine viene aggiornato il TLB usando un algoritmo di sostituzione (LRU tipicamente).

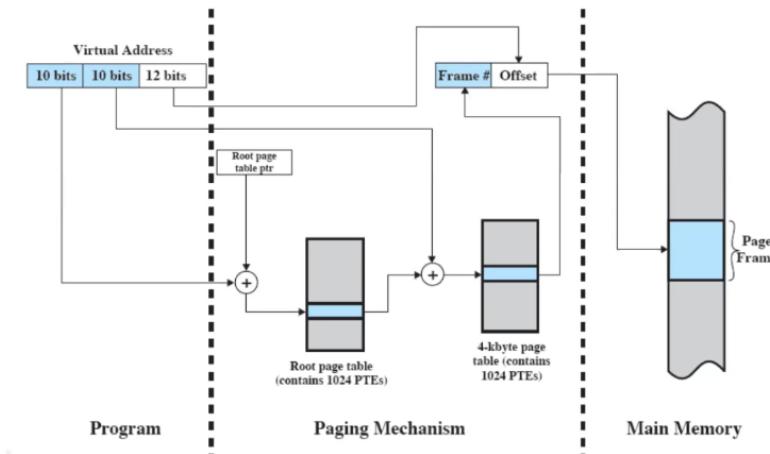


Figure 42: Hardware Tabella delle pagine a 2 livelli

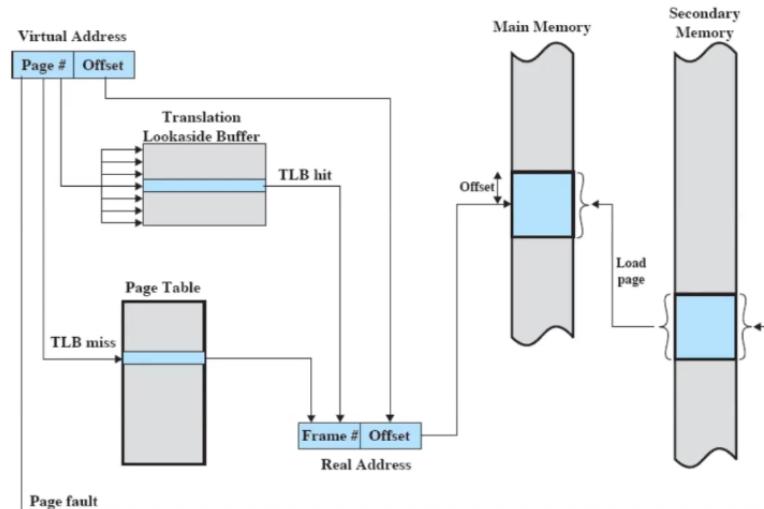


Figure 43: TLB

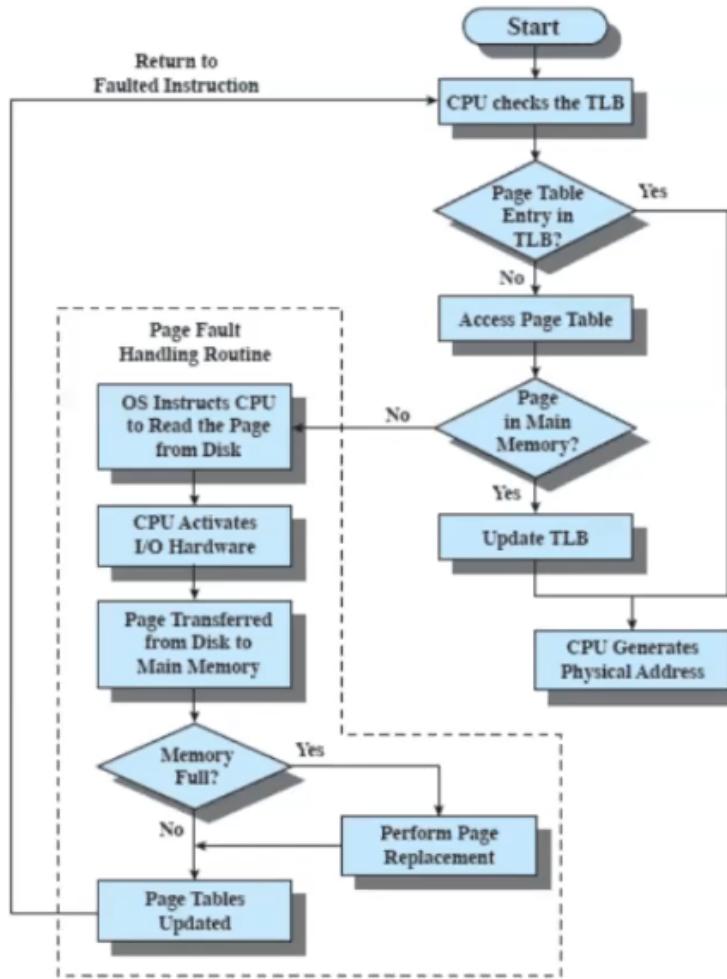


Figure 44: TLB

Il sistema operativo deve poter resettare il TLB, perché il TLB contiene informazioni di un processo, e se il processo cambia, il TLB deve essere resettato, alcuni processori permettono di avere il PID nel TLB, in modo da invalidare solo alcune parti del TLB, è comunque necessario anche senza TLB dire al processore dove é la nuova tabella delle pagine.

### Mapping Associativo

La tabella delle pagine ha tutte le entry, il TLB contiene solo alcune entry, quindi il numero della pagina non può essere usato direttamente come indice per il TLB (possibile nella tabella delle pagine), Il SO inoltre può interrogare più elementi del TLB contemporaneamente per capire se c'è o no un hit **con il supporto hardware**, un altro problema é che bisogna fare in modo che il TLB contenga solo pagine in RAM, perché se ci fosse un page fault dopo un hit del TLB potremmo non accorgercene, quindi ogni volta che si swappa una pagina bisogna anche resettare o parzialmente il TLB.

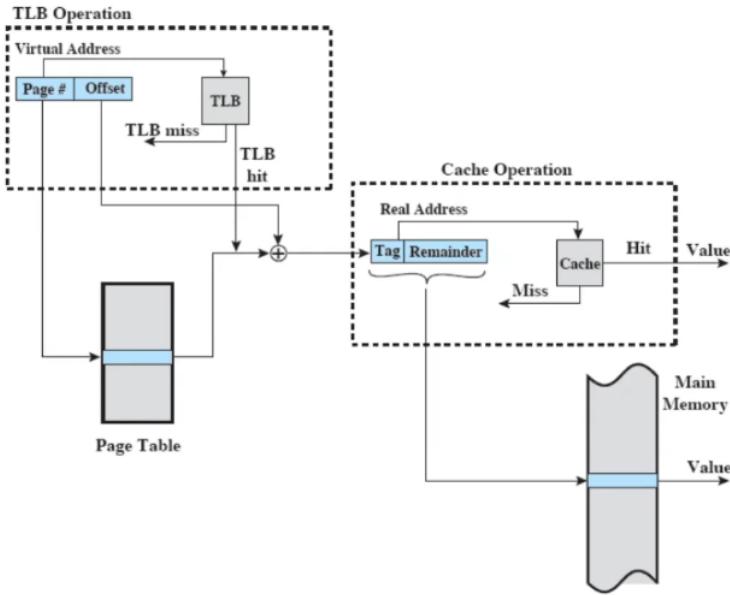
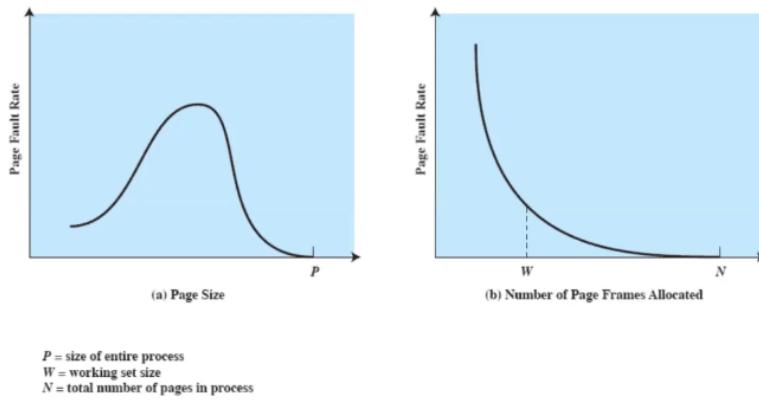


Figure 45: Mapping Associativo

#### 4.6.4 Dimensione delle pagine

Piú piccola é la pagina, minore é la frammentazione interna, ma maggiore é l'overhead, perché ci sono piú entry, la memoria secondaria é ottimizzata per trasferire grossi blocchi di dati, quindi avere le pagine ragionevolmente grandi non sarebbe male, quindi piú piccola é una pagina, maggiore il numero di pagine in RAM, E in tutte queste pagine, i riferimenti saranno vicini: in accordo con la località, i page fault saranno piú rari.

#### Pagefault vs Dimensione delle pagine



Con pagine grandi, pochi fault di pagina, ma poca multiprogrammazione!

Figure 46: Pagefault vs Dimensione delle pagine

Nelle moderne architetture HW possono supportare pagine anche fino ad 1GB, il sistema operativo ne sceglie 1 Es. Linux sugli x86 usa 4KB e le dimensioni più grandi sono usati in sistemi

operativi di grandi architetture cluster ma anche per i sistemi operativi stessi (kernel Mode)

#### 4.7 Segmentazione

La segmentazione permette al programmatore di vedere la memoria come un insieme di spazi di indirizzi, la loro dimensione può quindi variare, questo è utile perché semplifica la gestione delle strutture dati che crescono (Es. stack delle chiamate), permette di modificare e ricompilare i programmi in modo indipendente, permette di condividere e di proteggere i dati in modo più semplice : In un segmento ci sono dati condivisi (Tra processi) e invece in un altro segmento ci sono dati privati.

#### Organizzazione

Ogni processo ha una sua tabella dei segmenti (Il PCB punta a questa tabella), ogni entry contiene:

- l'indirizzo base del segmento
- la lunghezza del segmento
- un bit per indicare se il segmento è in memoria principale
- un bit per indicare se il segmento è stato modificato in seguito all'ultima volta che è stato caricato in memoria

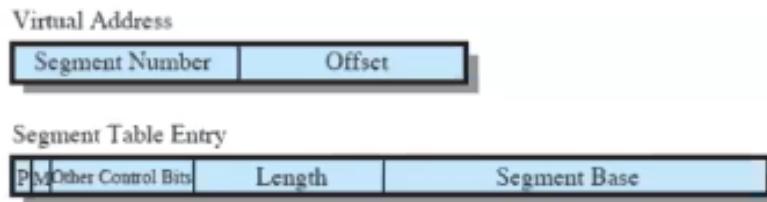


Figure 47: Segment Table Entry

#### Traduzione Indirizzi

Nella somma per trovare l'indirizzo fisico, è necessario moltiplicare l'indirizzo del segmento per la dimensione di ogni entry della tabella dei segmenti, dopo di che si può fare la somma.

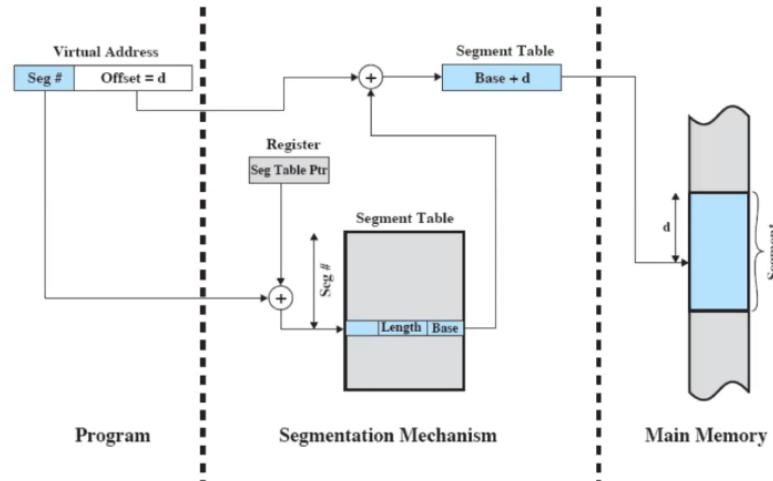


Figure 48: Traduzione Indirizzi

#### 4.8 Segmentazione e Paginazione

La paginazione é trasparente al programmatore, nel senso che il programmatore non ne é a conoscenza, mentre la segmentazione invece é implementata dal programmatore (se programma in assembly), tipicamente (Pentium) paginazione e segmentazione sono usate insieme, in questo caso il programmatore vede la memoria come un insieme di segmenti.

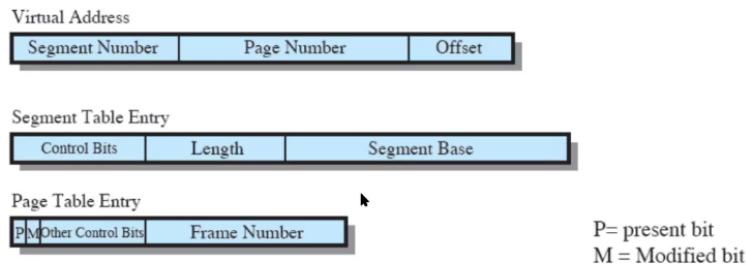


Figure 49: Segmentazione e Paginazione

### Traduzione di segmenti e pagine

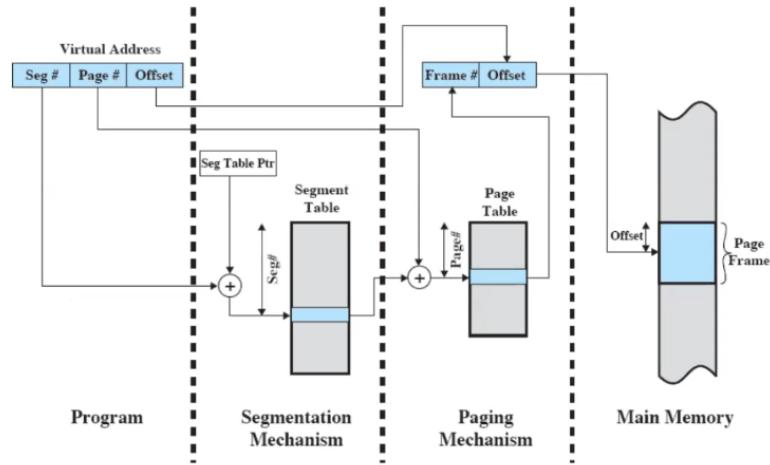


Figure 50: Traduzione Segmenti e Pagine

Combinando segmento e pagine, prima calcolo l'indirizzo del segmento, e poi calcolo l'indirizzo della pagina quindi nella Tabella dei segmenti trovo l'indirizzo della tabella delle pagine.

#### 4.8.1 Protezione

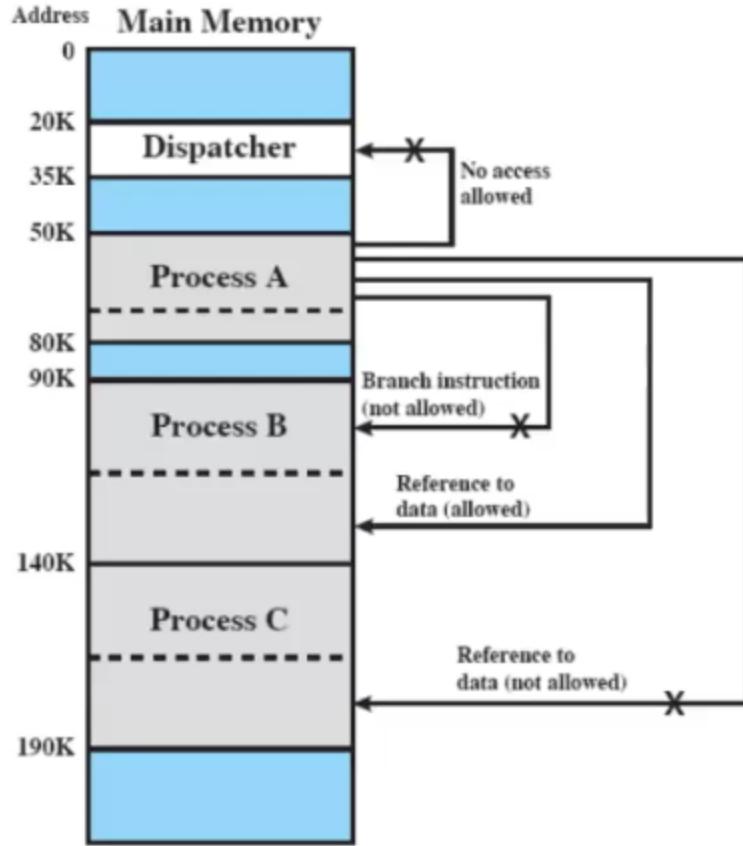


Figure 51: Tre Processi

Con la segmentazione implementare la condivisione ed la protezione è più semplice, perché basta mettere i segmenti condivisi in una zona di memoria condivisa, e mettere i segmenti privati in una zona di memoria privata, inoltre siccome ogni segmento ha una base ed una lunghezza, è facile controllare che i riferimenti siano contenuti ne giusto intervallo, invece per la condivisione basta dire che uno stesso segmento è condiviso tra più processi. Nell'immagine sopra, si vede come ogni processo ha una parte condivisa e una parte privata, la parte in alto è privata mentre in basso è condivisa, spetta al programmatore tramite opportune system call quali parti dei processi sono condivise

#### 4.9 Decisioni

- Usare o no la memoria virtuale ?
- Usare solo la paginazione? Linux usa sempre la paginazione
- Usare solo la segmentazione? Linux usa la segmentazione solo quando è costretto
- Usare entrambe?
- Che algoritmi usare per gestire i vari aspetti della gestione della memoria?

Esistono diversi aspetti da definire per la progettazione di un sistema operativo:

- **politica di prelievo**

- **Politica di posizionamento**
- **Politica di Sostituzione**
- **Gestione del residen set**
- **Politica di pulitura**
- **Controllo del carico**

Tutto questo deve essere effettuato cercando di minimizzare il numero dei page fault; non esiste quindi una politica sempre vincente.

#### 4.9.1 Fetch Policy

La fetch policy decide quando una pagina deve essere caricata in memoria:

- **Demand Fetch** : la pagina viene caricata solo quando c'è un page fault
- **Prepaging** : si carica in memoria anche le pagine che potrebbero essere usate in futuro

##### **Demand Fetch**

Con il paging on demand, abbiamo molti page fault all'inizio, siccome ogni pagina viene caricata solo quando serve.

##### **Prepaging**

Con il prepaging, si carica in memoria anche le pagine che potrebbero essere usate in futuro, in modo da evitare page fault, il problema è che si potrebbero caricare in memoria pagine che non verranno mai usate. Si cerca quindi di sfruttare il principio di località, caricando in memoria pagine vicine a quelle che sono state richieste.

#### 4.9.2 Posizionamento

La politica di posizionamento decide dove mettere le pagine in memoria, quando c'è almeno un frame libero, salto la prima parte riservata al kernel e metto la pagina al primo frame libero che trovo, grazie alla traduzione degli indirizzi possiamo mettere le pagine in memoria in modo non contiguo, ogni volta che viene caricata in memoria la tabella delle pagine viene aggiornata, quindi questo si può fare solo quando c'è un frame libero, se invece tutti i frame sono occupati, bisogna avere una politica di sostituzione.

#### 4.9.3 Sostituzione

La politica di sostituzione decide quale pagina sostituire quando tutti i frame sono occupati, il problema è che non si può sapere a priori quale pagina verrà usata, essenzialmente ci sono due problemi quando ho deciso quale frame sostituire:

- prendere la pagina che verrà usata e sostituirla nella tabella delle pagine e mettere il bit di presenza a 1
- per la pagina che ho appena sostituito devo aggiornare la tabella delle pagine e mettere il bit di presenza a 0

Gli algoritmi sono pensati per minimizzare la probabilità che appena ho sostituito una pagina, debba subito essere ricaricata.

#### 4.9.4 Gestione del resident set

Il resident set é la parte del processo che é in memoria principale, la gestione del resident set ha due problemi:

- decidere per ogni processo quanti frame vanno assegnati
- Quando sis rimpiazza un frame, bisogna scegliere solo tra i frame del solo processo corrente o tra tutti i frame presenti in memoria (Replacement Scope)

Esistono 2 tecniche per risolvere il problema:

- **Fixed Allocation** : assegno un numero fisso di frame ad ogni processo e non lo modifco fino al termine di esso
- **Variable Allocation** : assegno un numero variabile di frame ad ogni processo

Per quello che riguarda il Replacement Scope, si puó scegliere di rimpiazzare solo i frame del processo corrente o tutti i frame presenti in memoria, la scelta dipende dal fatto che si voglia privilegiare la localitá o meno, da notare che se scelgo la locazione fissata, il Replacement Scope é automaticamente limitato al processo corrente.

#### Frame Bloccati

Un frame é bloccato se non puó essere rimpiazzato, ad esempio perché contiene una parte del sistema operativo, o perché contiene una parte di un dispositivo di I/O, o perché contiene una parte di un processo che é in esecuzione.

#### 4.9.5 Politica di Pulitura

Se un frame é stato modificato, va riportata la modifica anche sulla pagina corrispondente, il problema é : quando ?

- non appena avviene la modifica
- quando il frame viene rimpiazzato

Tipicamente si fa una via di mezzo, intrecciata con il page buffering, solitamente si raccolgono un po'di richieste di frame da modificare e si fanno tutte insieme.

#### 4.9.6 Controllo del Carico

Esistono 3 tipi di scheduler:

- **Short Term Scheduler** : riferito ai processi ready.
- **Medium Term Scheduler** : riferito ai processi in memoria principale
- **Long Term Scheduler** : riferito ai processi in memoria secondaria

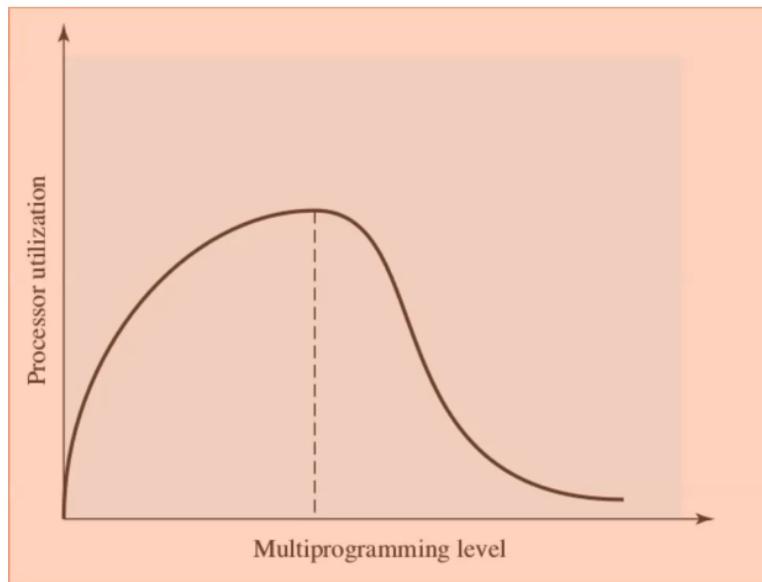


Figure 52: Controllo del carico

L'idea del controllo del carico é quella di mantenere il grado piú alto possibile di multiprogrammazione, quindi mantenere quanti piú processi possibile in memoria principale, ma non troppo perché altrimenti si rischia il thrashing, ovvero sono presenti tanti page fault, si vede dal grafico che con un buon livello di multiprogrammazione si ha un buon tempo di utilizzo del processore, mentre con un livello troppo alto si ha un tempo di utilizzo del processore inferiore.

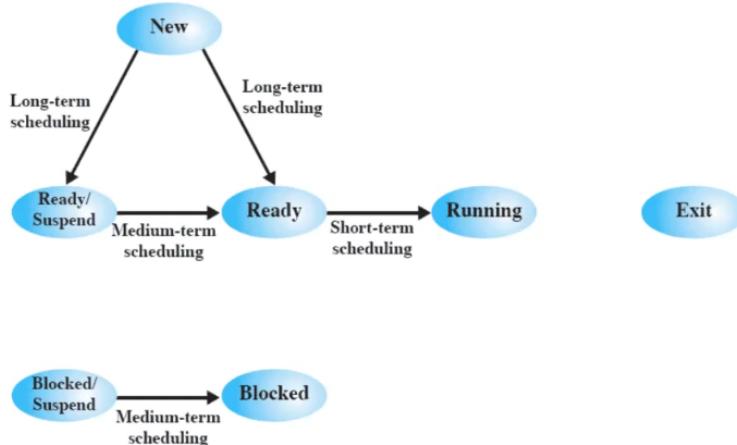


Figure 53: Stati dei processi e Medium Term Scheduler

Un processo quindi si dice **suspended** quando il suo resident set é 0, mentre si dice **Run** quando il suo resident set é maggiore di 0, il Medium Term Scheduler si occupa di decidere quali processi sospendere e quali far ripartire, inoltre si occupa di decidere quali processi caricare in memoria principale

### Controllo del carico

Controllo del carico vuol dire che il sistema operativo ha due cose che puó fare:

- Prendo un processo che é suspended e lo metto in memoria principale rendendolo ready
- Prendo un processo che é ready e lo sospendo mettendolo in memoria secondaria

Se devo "svegliare" dei processi da disco → *ram, lodevo fare con processi che sono o che saranno ready, al contrario*

L'idea è quella di effettuare un monitoraggio dei processi, ogni tanto c'è un processo del SO che controlla i processi e decide quali sospendere e quali far ripartire, ad esempio misurare il tempo medio per due fault di pagina e confrontarlo con il tempo medio della gestione di un fault se queste cose sono molto vicine significa che stiamo facendo thrashing, la politica di monitoraggio viene chiamata ogni tot page fault, questo viene messo all'interno della politica di rimpiazzamento

### Come scegliere un processo da sospendere

- processo con minore priorità
- processo che ha causato l'ultimo page fault
- ultimo processo attivato
- working set più piccolo
- il processo che ha il più numero di pagine
- processo con il più alto tempo rimanente di esecuzione(se disponibile)

### 4.10 Algoritmi di Sostituzione

Immaginiamo di avere la RAM piena e per un dato processo conosco i frame che gli sono stati assegnati. gli algoritmi che possiamo usare sono:

- **Optimal** : sostituisce la pagina che non verrà usata per il più lungo periodo di tempo
- **FIFO** : sostituisce la pagina che è stata caricata per prima
- **LRU** : sostituisce la pagina che non è stata usata per il più lungo periodo di tempo
- **Clock**

### Esempio comune

Siamo in una memoria con 3 frame, e abbiamo 5 pagine, e la sequenza di riferimenti è la seguente:

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

### Optimal

L'algoritmo ottimale sostituisce la pagina che non verrà usata per il più lungo periodo di tempo, quindi per ogni pagina che devo sostituire, guardo tutte le pagine che verranno usate in futuro e sostituisco quella che verrà usata più tardi, il problema è che non posso sapere a priori quali pagine verranno usate in futuro, quindi l'algoritmo ottimale è un algoritmo teorico, perché non è implementabile.

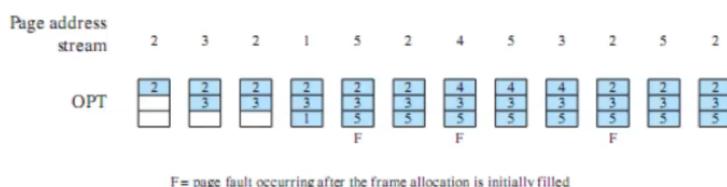


Figure 54: Optimal

come si nota dalla figura, abbiamo solo 3 pagefault,dopo che la memoria é piena, si puó notare quando si vuole caricare 5 vado a sostituire 1, perché 1 non verrá piú usato, e cosí via.

### LRU

LRU (Least Recently Used) sostituisce la pagina che non é stata usata per il piú lungo periodo di tempo, Basandosi sul principio di localitá, dovrebbe essere la pagina che ha meno probabilitá di essere ri-usata al riferimento successivo, il motivo per cui si é restii ad adottare LRU é che é difficile da implementare, perché ad ogni frame bisogna associare un timestamp, e poi scorrere tutti i tempi e scegliere il minore, la Cache lo fa in hardware, ma in software é un overhead considerevole.

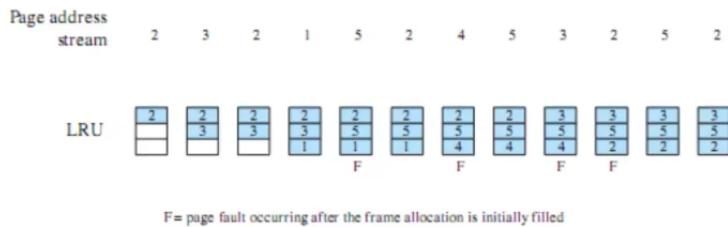


Figure 55: LRU

Nell'esempio con l'algoritmo ottimale avevamo sostituito 1, invece con LRU sostituiamo 3, perché 3 é stato usato prima di 1 e di 2, quindi é la pagina che ha meno probabilitá di essere riutilizzata, anche se la scelta ottimale sarebbe stata 1, il totale dei page fault nella sequenza é comunque 4 che é un ottimo risultato.

### FIFO

FIFO (First In First Out) sostituisce la pagina che é stata caricata per prima, é un algoritmo molto semplice, in pratica i frame vengono trattati come una coda circolare, da questa coda, le pagine vengono rimosse a turno (Round Robin) ,il vantaggio é che é molto semplice da implementare, in pratica vengono rimpiazzate le pagine che sono state in memoria per piú tempo.

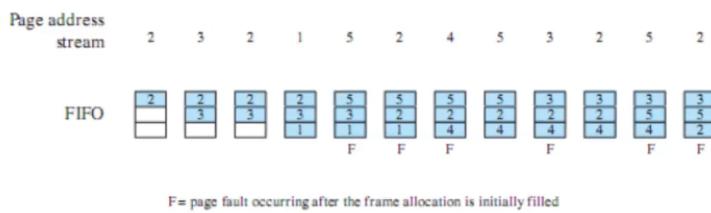


Figure 56: FIFO

Vediamo come in questo caso il risultato sono 6 page fault, perché l'algoritmo non si accorge che la pagina 2 e 5 sono molto richieste.

### Clock

L'algoritmo Clock é un compromesso tra LRU e FIFO, esiste quindi un "use bit" per ogni frame, che indica se la pagina caricata nel frame é stata riferita, il bit quindi viene messo a 1 quando la pagina viene caricata in memoria principale, e poi rimesso ad 1 per ogni accesso al suo interno,

quando é necessario sostituire una pagina, si scorre la coda circolare con il FIFO e si controlla il bit, se il bit é 0, la pagina viene sostituita, altrimenti il bit viene messo a 0 e si passa al prossimo frame.

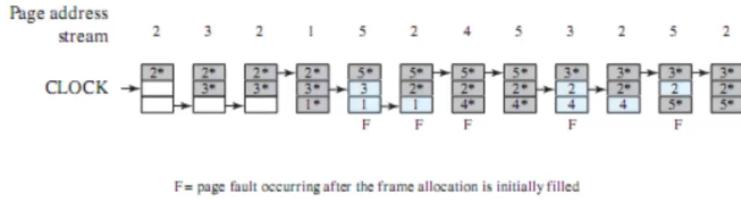


Figure 57: Clock

Riempiamo la memoria con 3 pagine e ci troviamo con 3 frame occupati con tutti i bit ad 1, quando arriva la pagina 5 a questo punto scorro tutti i frame e nella prima passata metto tutti bit a 0, alla seconda passata trovo come primo frame con bit 0 il frame contenente la pagina 2, quindi sostituisco la pagina 2 con la pagina 5, dopo arriva la chiamata alla pagina 2, e quindi vado a sostituire la pagina 3 perché é la prima con bit 0 e cosí via. Il numero di page fault é 5.

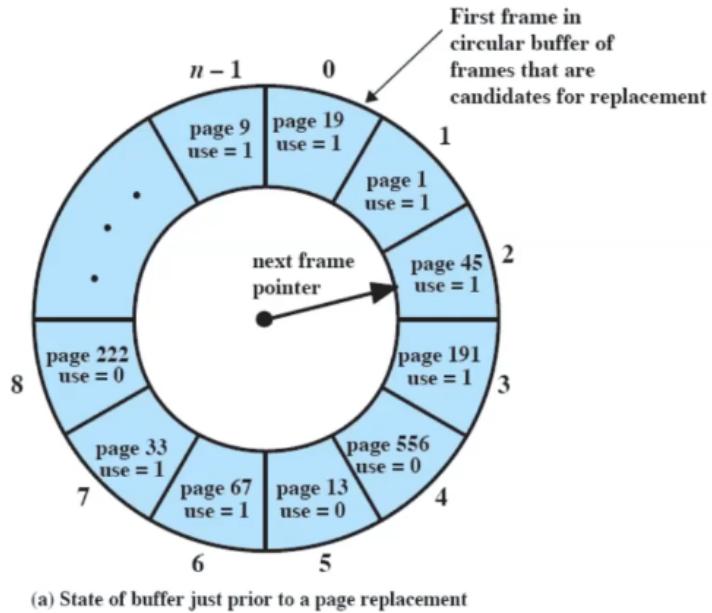


Figure 58: Clock

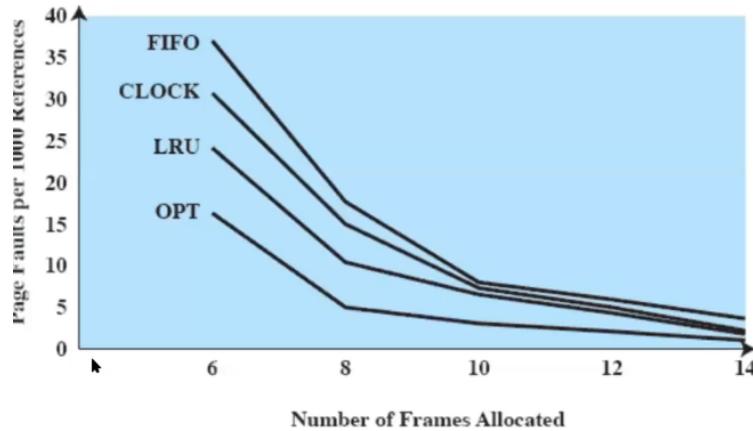


Figure 59: Clock

### Buffering delle pagine

Un'altra tecnica per ridurre il numero di page fault è il buffering delle pagine, è nata come una modifica dell'algoritmo FIFO, serve per ridurre il numero di page fault, in pratica riduco la memoria al processo, e la parte rimanente la uso come buffer(cache), in questo modo se una pagina viene rimpiazzata, non viene scritta su disco, ma viene messa nel buffer, in questo modo se la pagina viene richiesta di nuovo, non devo andare a leggerla dal disco, ma la prendo dal buffer.

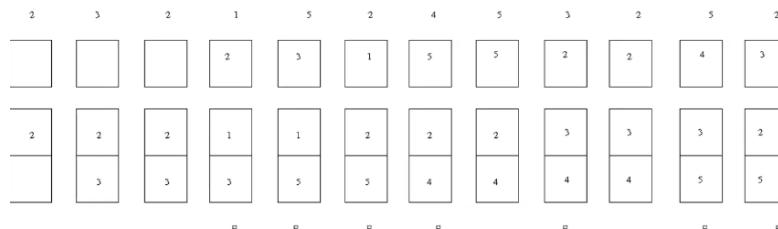


Figure 60: Buffering delle pagine

Ci sono solo 2 frame per il processo bastano le prime 2 pagine, quando arriva 1 sostituisco 2 (FIFO), ma non butto via 2, ma lo metto nel buffer, quando arriva 5 sostituisco 3, ma non butto via 3, ma lo metto nel buffer e così via, l'efficienza del buffering si apprezza solo con un numero ragionevole di frame.

### 4.11 Gestione della Memoria in Linux

Linux fa una distinzione netta tra richieste di memoria da parte del kernel e di processi utente, il kernel si fida di se stesso, mentre per i processi utente ci sono tanti controlli che uccidono il processo se fa qualcosa di sbagliato.

#### Gestione della memoria Kernel

Il kernel potrebbe avere bisogno di memoria sia piccole che grandi contemporaneamente, quindi il kernel può usare sia la parte riservata che anche quella dei processi utente, se la richiesta è piccola, il kernel fa in modo di avere già pronte alcune pagine in memoria, questa tecnica si chiama **Slab Allocator**, se la richiesta è grande, il kernel fa in modo più pagine contigue in frame contigui, questo per il DMA per esempio, ricordiamo che il dispositivo di I/O non sa

niente di paginazione, quindi se devo fare un trasferimento di dati tra memoria e dispositivo, devo avere pagine contigue, per fare questo il kernel usa la **Buddy Allocator** che è una versione modificata del buddy system.

### Gestione della memoria utente

- **Fetch Policy** : Linux usa la paginazione on demand
- **Placement Policy** : Il primo frame libero
- **Replacement Policy** : TBA
- **Gestione del Resident Set** : politica dinamica con replacement scope globale (Rientra anche la cache del disco)
- **Pulitura** : viene ritardata il più possibile : page cache piena, troppe pagine sporche, richiesta esplicita
- **Controllo del Carico** : assente

### Replacement Policy Linux

Linux usa un algoritmo dell'orologio corretto ma il kernel più recente usa LRU, a differenza dell'algoritmo dell'orologio originale quello corretto ha due flag in ogni entry della page tabel, PG\_referenced e PG\_active, sulla base di PG<sub>active</sub>, *duelistedi pages sono mantenute dal kernel : attive e inattive*.

kswapd è il demone che si occupa di scorrere le pagine inattive PG\_referenced è settato quando la pagina viene richiesta, dopo di che, delle due úna : o arriva prima kswapd (Rimpiazza la pagina) o un altro riferimento , nel secondo caso (pagina riferita più volte in poco tempo) pagina settata attiva ,nel primo PG\_referenced è resettato, solo le pagine inattive possono essere rimpiazzate.

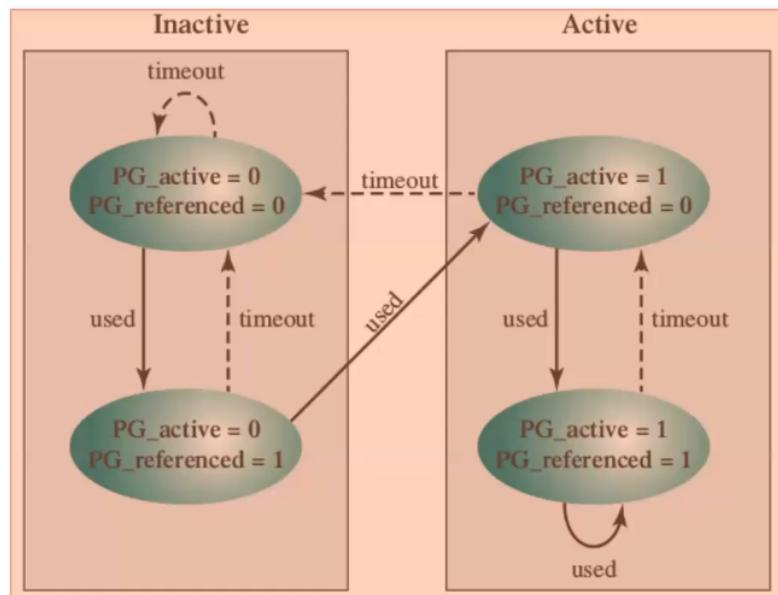


Figure 61: Replacement Policy Linux

## 5 La Gestione I/O

### 5.1 Categoria di dispositivi

Area assai problematica per la progettazione di sistemi operativi perché:

- I dispositivi sono molto diversi tra loro
- Tante applicazioni che ne fanno uso
- Difficili scrivere un SO che supporti tutti i dispositivi

le categorie di dispositivi sono:

- leggibili dall'utente
- leggibili dalla macchina
- dispositivi di rete/comunicazione

### 5.2 Dispositivi leggibili dall'utente

I dispositivi per la comunicazione diretta con l'utente sono:

- stampanti
- terminali: ovvero monitor + tastiera
- joystick

### 5.3 Dispositivi leggibili dalla macchina

I dispositivi leggibili dalla macchina sono quelli che soltanto usando l'elettronica si possono leggere, come:

- dischi
- chiavi USB
- Sensori

### 5.4 Dispositivi di rete/comunicazione

I dispositivi di rete/comunicazione sono:

- schede di rete
- modem
- WiFi

## 5.5 Funzionamento(Semplificato) dei dispositivi input

Un dispositivo di input prevede di essere interrogato sul valore di una certa grandezza fisica al suo interno:

- tastiera: codice Unicode del tasto premuto
- mouse: Coordinate dell'ultimo spostamento effettuato
- disco: valore dei bit che si trovano in una certa posizione al suo interno (questo se è usato come input)

Se un processo fa una syscall read su un dispositivo di input vuole conoscere questo dato, per poterlo ovviamente elaborare il processo che gestisce un edito, una volta saputo quale tasto è stato premuto sulla tastiera, può fare l'echo del carattere corrispondente.

## output

Un dispositivo di output prevede di poter cambiare il valore di una certa grandezza fisica al suo interno:

- monitor: Valore RGB di tutti i suoi pixel, oppure quelli che cambiano rispetto alla situazione immediatamente precedente
- stampante: PDF o PS di un file da stampare
- disco: valore dei bit che devono sovrascrivere quelli che si trovano in una certa posizione al suo interno

Un processo che effettua una syscall write su un dispositivo del genere vuole cambiare qualcosa, spesso l'effetto è direttamente visibile all'utente, in altre occasioni l'effetto è visibile solo usando altre funzionalità di lettura.

Ci sono quindi, minimalmente, due syscall read e write tra i loro argomenti c'è un identificativo del dispositivo da leggere o scrivere per esempio, Unix e Linux hanno i file descriptor. Se c'è una system call read o write viene sollevata una eccezione di sistema, il kernel prende il controllo e si occupa di fare la lettura o la scrittura, mentre il processo è blocked e passa ad altro, la parte del kernel che gestisce un particolare dispositivo di I/O è chiamata driver, spesso il trasferimento si fa con il DMA(Direct Memory Access), il driver scrive direttamente in memoria, senza passare per la CPU. A trasferimento completato il driver solleva un'interruzione, il kernel prende il controllo e risveglia il processo che aveva fatto la syscall, l'operazione potrebbe anche fallire, (bad block su disco...), potrebbe anche essere necessario fare ulteriori trasferimenti, per esempio dalla RAM dedicata al DMA a quella del processo.

## 5.6 Differenze tra dispositivi di I/O

I dispositivi di I/O possono differire sotto molti aspetti:

- data rate (frequenza di accettazione/emissione di dati)
- applicazione
- difficoltà di controllo (Tastiera vs disco)
- unità di trasferimento dati (caratteri vs. blocchi)
- rappresentazione dei dati
- condizioni di errore

## Data Rate

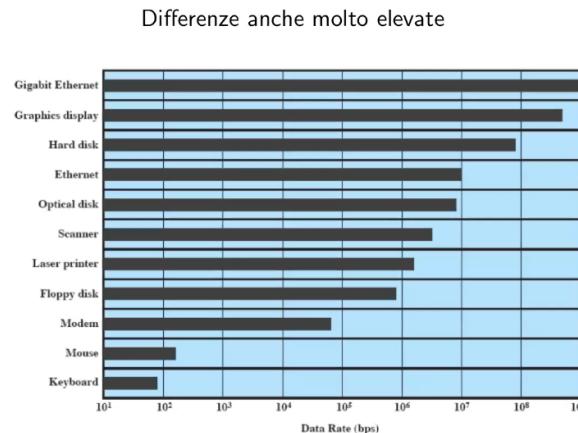


Figure 62: Data Rate

I data rate possono variare di molto, ci sono dispositivi che sono molto lenti, ed altri molto veloci, vediamo per esempio che il dispositivo gigabit ethernet può trasferire  $10^9$  bits secondo.

## Applicazioni

I dischi sono usati per memorizzare files, richiedono un software per la gestione dei file, i dischi sono usati anche per la memoria virtuale, per la quale serve altro software apposito (nonché hardware), Un terminale usato da un amministratore di sistema dovrebbe avere una priorità più alta.

## Complessitá di controllo

Una tastiera o un mouse richiedono un controllo molto semplice, mentre una stampante è più complicata, ma non troppo, alle stampanti moderne basta ricevere un PDF o PS la traduzione da PDF ad azioni della stampante è ovviamente complessa, il disco è molto più complesso, richiede sia un controllo software che hardware

## Unitá di trasferimento dati

I dati possono essere trasferiti in unitá di trasferimento diversi:

- Blocchi di byte di lunghezza fissa (dischi, chiavi USB, CD)
- Come un flusso (Stream) di byte, o equivalentemente di caratteri (Qualsiasi cosa non sia memoria secondaria)

## Rappresentazione dei dati

I dati sono rappresentati secondo codifiche diverse, una vecchia tastiera ad esempio potrebbe rappresentare i suoi dati in ASCII, una moderna invece utilizza UNICODE, inoltre possono anche essere diversi i controlli di parità.

## Condizioni di errore

La natura degli errori varia di molto da dispositivo a dispositivo, ad esempio nel modo in cui gli errori vengono notificati, sulle loro conseguenze (fatali/ignorabili) su come possono essere gestiti.

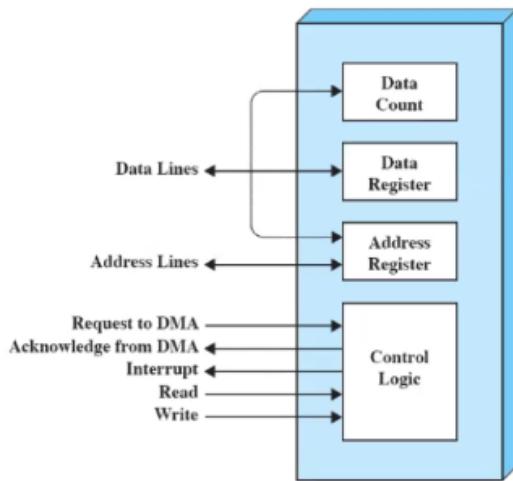


Figure 64: DMA

### 5.7 Tecniche per effettuare l'I/O

- Programmato
- Guidato dagli interrupt
- Accesso diretto in memoria (DMA)

	Senza Interruzioni	Con Interruzioni
Passando per la CPU	I/O programmato	I/O guidato dalle interruzioni
Direttam. in memoria		DMA

Figure 63: I/O

Per fare input output ci sono 4 modalità con o senza interruzioni, con o senza CPU, fare I/O senza interruzioni significa fare I/O Programmato, invece usando le interruzioni si può fare I/O usando comunque la CPU, oppure evitare di usare la CPU e delegare il DMA di fare il trasferimento.

### DMA

Il DMA contiene una serie di registri che si occupano di contenere i dati da trasferire, poi c'è la control logic che si occupa di gestire il trasferimento, in pratica si interfaccia con il sistema operativo.

### Evoluzione della funzionalità di I/O

1. il processore controlla il dispositivo, il trasferimento è fatto dalla CPU
2. Viene aggiunto un modulo di I/O, direttamente sul dispositivo (I/O programmato) ma senza interrupt ma il processore non si deve occupare di alcuni dettagli del dispositivo stesso
3. Modulo o controllore di I/O con interrupt, il processore viene avvisato quando il trasferimento è completato

4. DMA dispositivi comunicano direttamente con la memoria, il processore non é coinvolto
5. Il modulo I/O diventa un processore separato, il processore "principale" comanda al processore di I/O di eseguire un certo programma di I/O in memoria principale
6. Processore per l'I/O ha una sua memoria dedicata usata per la comunicazione con terminali interattivi

### 5.8 Obiettivo per SO : Efficienza

La maggior parte dei dispositivi di I/O sono molto lenti rispetto alla memoria principale, grazie alla multiprogrammazione, alcuni processi potrebbero essere in attesa del completamento di un'operazione di I/O mentre altri processi sono in esecuzione, Ma l'I/O potrebbe comunque non tenere il passo con il processore

- quindi il numero di processi ready si riduce fino a diventare 0
- quindi si potrebbe pensare che sia sufficiente portare altri processi ready ma sospesi in memoria principale (Medium-Term Scheduler) , ma anche questa é una operazione di I/O.

si rende quindi necessario cercare soluzioni software dedicate, a livello di SO, per l'I/O, in particolare per il disco.

### 5.9 Obiettivo per SO : Generalitá

Per semplicità e per evitare errori, sarebbe bene gestire i dispositivi di I/O in modo uniforme, per esempio esiste un'unica systemcall read che prende il giusto argomento per sapere quale dispositivo leggere, occorre nascondere la maggior parte dei dettagli dei dispositivi di I/O nelle procedure di basso livello, Progettare le funzioni di I/O in modo modulare e gerarchico(anche se é difficile farlo in modo generale), le funzionalità da offrire sono:

- read
- write
- lock
- unlock
- open
- close

### Progettazione Gerarchica

Per mantenere la generalità, si fa uso della progettazione gerarchica, in pratica ci sono dei livelli, che sono usati all'interno del codice del SO, l'idea è che ogni livello si basa su quello che fa il livello sotto-stante, e quello sottostante quindi si occupa di fornire servizi a quello soprastante,ogni livello contiene funzionalità che sono simili per complessità, tempi di esecuzione per l'I/O, ci sono 3 macro tipi. \*Dispositivo Locale



Figure 65: Dispositivo Locale

Cose che vengono attaccate direttamente al computer, il livello piú alto é un processo un-tente, infondo invece troviamo l'hardware, in mezzo invece troviamo quello che deve fare il sistema operativo, vicino all'hardware troviamo operazioni di scheduling e controllo ad esempio tanti processi che cercano di scrivere sul monitor, quindi il sistema operativo deve essere in grado di gestire queste richieste ed ordinarle in modo da non creare conflitti, piú in su troviamo Device I/O e Logical I/O servono a fornire una interfaccia semplice all'utente, ad esempio logical I/O prende delle richieste di alto livello ad esempio **open** e **close** mentre Device I/O si occupa di trasformare le richieste logiche in comandi.

### Dispositivo di comunicazione



Figure 66: Dispositivo di comunicazione

IL dispositivo di comunicazione è praticamente lo stesso del dispositivo locale, solo che è presente un blocco di communication architecture

## File System



Figure 67: File System

Quello che differisce il file system é che il logical I/O é che sono presenti piú blocchi tra lo user e l'hardware:

- Directory Management permette di definire tutte le operazioni che hanno a che fare con i file creare file ecc
- File System é la struttura logica che permette di aprire chiudere leggere scrivere ecc
- Physical Organization é la struttura che permette di allocare e di-allocare spazio sul disco quindi il sistema operativo interroga il disco per capire dove c'é spazio libero

### 5.10 Buffering I/O

Per cercare di rendere piú efficiente il sistema di I/O, é conveniente fare buffering, ovvero tenere in memoria i dati per completare una certa operazione, essenzialmente l'idea é che noi sappiamo che le pagine devono essere spostate dalla memoria principale a disco, e quindi il semplice fatto di usare la memoria virtuale ha come effetto collaterale gestire I/O, c'è un problema con il DMA, perché un processo puó richiedere un I/O su una certa zona di memoria (DMA), ma viene swappato subito, Il problema é quindi che ci ritroviamo con 2 richieste discordanti, perché per completare I/O il processo deve essere in memoria, invece se il sistema operativo swappa il processo questo ha come effetto che bisogna fare essenzialmente l'opposto, per riassumere **Ci troviamo in una situazione dove la richiesta I/O é in conflitto con la gestione dei processi in memoria** questo problema é risolvibile con il frame blocking, semplicemente le pagine che vengono accedute dal DMA non possono venire swappate, tuttavia se inizio a fare troppo frame blocking, limito il numero dei processi che posso eseguire, portando a minore multiprogrammazione, Una possibilità é quella di fare il buffering in memoria, quindi fare I/O on demand, ovvero fare I/O solo quando viene richiesto, oppure fare buffering in memoria, ovvero fare I/O in anticipo, ovvero fare I/O prima che venga richiesto.

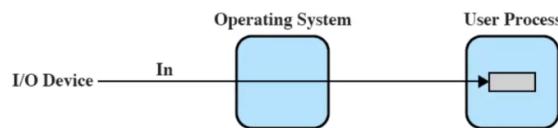


Figure 68: senza buffer

Senza buffer il sistema operativo dice al dispositivo di I/O di scrivere, e come si vede il DMA viene istruito di scrivere direttamente nel processo.

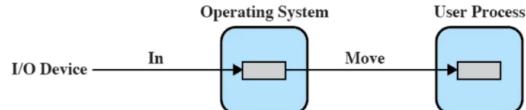


Figure 69: Con buffer

Con il buffer non operiamo piú direttamente sul processo, ma sul buffer, ovvero una memoria intermedia del sistema operativo, dove solo in un secondo momento il processo legge i dati, in questo modo si fa un trasferimento da RAM a RAM.

#### buffer singolo orientato ai blocchi

I trasferimenti di input sono fatti al buffer in system memory, e il blocco viene mandato nello spazio utente quando necessario, il prossimo blocco viene comunque letto nel buffer, l'input quindi viene anticipato, i dati, solitamente, vengono acceduti sequenzialmente: c' é buona probabilitá che servirá, e sarà già stato letto, invece l'output viene posticipato, per questo serve la system call flush... potrebbe succedere che debuggando un programma, il programma crasha, quello che si fa quindi é fare delle richieste di printf per vedere se effettivamente il programma passa da quel punto, se non vedo una certa printf, allora il programma crasha prima, ma questa cosa potrebbe essere non vera a causa del buffering, per questo si usa la system call flush per forzare il buffer a scrivere

### buffer singolo orientato agli stream

I terminali tipicamente hanno a che fare con linee, in questo caso la bufferizzazione riguarda le linee, quindi si bufferizza una linea intera di input o di output un byte alla volta per i device in cui un singolo carattere premuto va gestito

### Buffer Doppio

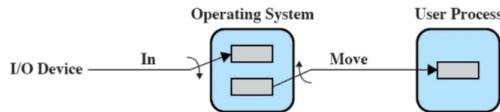


Figure 70: Buffer Doppio

Esiste il problema che nel buffer singolo la zona di memoria dedicata é limitata, quindi esiste la possibilità che il buffer si riempia, per ovviare questo problema si puó adottare il buffer doppio.

### Buffer Multiplo



Figure 71: Buffer Multiplo

In generale si possono avere piú buffer, in pratica un certo dispositivo di I/O ha un puntatore che ha un certo indirizzo con il buffer da usare, poi il puntatore viene incrementato, e si passa al buffer successivo, in questo modo quando ho finito di usare l'ultimo riparto dal primo, sicuramente il primo sarà libero, progettare questa cosa non é semplicissima infatti si tratta di un problema che si chiama produttore consumatore

### pro e contro del buffering

IL buffering riesce a mantenere il processore non idle anche quando ci sono tante richieste di I/O, quindi ha il compito di smussare i picchi di richieste di I/O, soprattutto é utile quando ci sono molti e diversi dispositivi da gestire, generalmente la zona di memoria é statica mentre alcuni SO tipo linux possono decidere di allargarla ai processi.

## 5.11 HDD vs SSD

Abbiamo visto che per gestire l'input output i sistemi operativi devono essere efficienti, ci sono alcune tecniche che devono essere pensate per alcuni dispositivi in particolare, un di quelli dove c' é stato piú focus sono quelli dell' archiviazione di massa, quindi per aprire un file devo cercare di rispondere nella maniera piú veloce possibile.

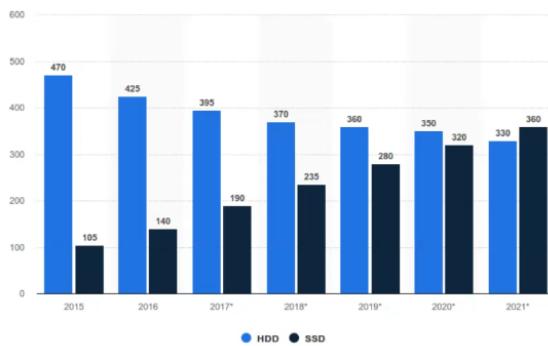


Figure 72: HDD vs SSD

## HDD

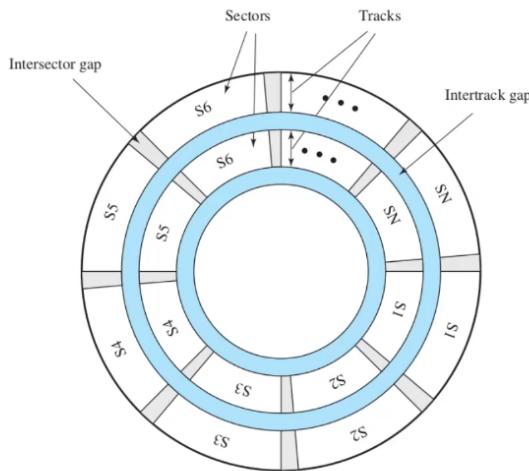


Figure 73: HDD

Una traccia é un intero percorso (corona circolare), un settore é una parte di una traccia, quindi possiamo vedere il disco composto da diverse circonferenze che sono allora volta divise in settori, esistono quindi delle zone di demarcazione per dividere le tracce e i settori, quindi i dati sono sulle tracce essenzialmente se devo leggere/scrivere devo sapere su quale traccia e settore si trovano, per leggere scrivere é presente un meccanismo é presente una testina che puó essere una sola o piú testine, esse si spostano/sono posizionate su una traccia , una volta che sono posizionato sulla traccia se non sono nel giusto settore, il disco deve girare fino a che non si trova nel giusto settore se i dati sono tanti possono essere su piú settori o addirittura su piú tracce, un settore puó tenere 512 byte

## prestazioni dell'HDD

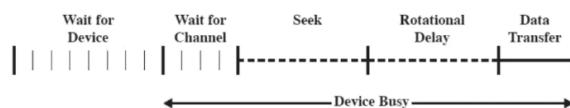


Figure 74: Prestazioni HDD

Possiamo suddividere il tempo di accesso come nell'immagine :

1. Wait for device aspetto che il disco sia pronto e che abbia finito l'operazione precedente
  2. wait for channel aspetto che il canale sia pronto
  3. Seek tempo per muovere la testina fino a posizionare la testina sulla traccia giusta
  4. Rotational delay tempo per far girare il disco fino a che il settore giusto non si trova sotto la testina
  5. Data Transfer tempo per trasferire i dati il disco ruota con la testina che legge, se mi devo spostare si ripartirà da seek
- Il tempo di accesso è composto dalla somma di seek e rotational delay
  - Tempo di trasferimento tempo necessario per trasferire i dati che scorrono sotto la testina

### Politiche di scheduling del disco

Come è successo per la gestione della ram nel caso in cui siamo su un disco con testine mobili sono stati pensati diversi algoritmi per rendere più efficiente la lettura/scrittura.

#### Esempio generale

- all'inizio la testina si trova sulla traccia numero 100
- ci sono 200 tracce
- vengono richieste le tracce in ordine : 55,58,39,18,90,160,150,38,184
- considereremo solo il seek time, che è il parametro più importante per le prestazioni
- confronteremo con il random scheduling che è il peggiore

#### FIFO

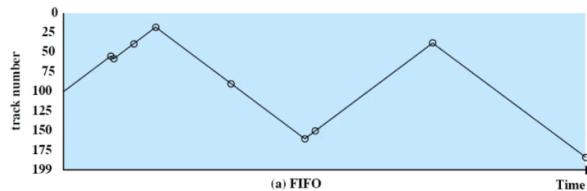


Figure 75: FIFO

Le richieste sono servite in modo sequenziale in questo modo la testina si muove molto, in questo modo i processi sono serviti in modo equo, se ci sono molti processi in esecuzione, le prestazioni sono simili allo scheduling random

#### Priorità

Esiste la possibilità di affidarsi alla priorità dei processi, ovvero lo scopo non è più quello di migliorare le prestazioni del disco, ma cerco di soddisfare prima i processi più importanti, ad esempio i processi interattivi devono essere serviti prima per garantire l'esperienza utente, i processi lunghi potrebbero aspettare troppo e non va bene per i DBMS

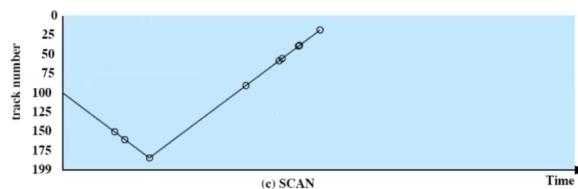


Figure 77: SCAN

## LIFO

Il lifo (si riferisce all'utente) é ottimo per DBMS con transazioni (una sequenza di iterazioni che non puó essere interrotta), il dispositivo é dato all'utente piú recente, altri utenti quindi posso soffrire di starvation (Continuamente c'è qualcuno che resta fermo) se siamo in un ambiente con transazioni, se un certo utente ha bisogno di un certo dato, é meglio che questo utente sia servito prima, grafico non possibile perché per ogni richiesta bisogna sapere quale utente l'ha fatta.

## Minimo tempo di servizio (SSTF)

Non vado nell'ordine in cui sono state fatte le richieste, ma le analizzo e cerco di fare in modo di andare alla traccia piú vicina, in questo modo la testina si muove di meno, quando si esegue questo algoritmo si hanno un buon numero di richieste, é possibile che ci sia una starvation, perché se arrivano molte richieste vicine (in termini di traccia), quelle lontane potrebbero non essere servite.

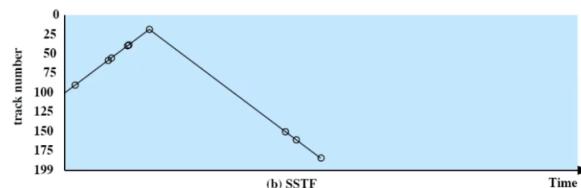


Figure 76: Minimo tempo di servizio

## SCAN

Per risolvere il problema del minimo tempo di servizio, si puó usare lo SCAN, ovvero le richieste vengono servite seguendo un solo verso, quando si arriva alla fine si inverte la direzione, in questo modo si evita la starvation, un problema é che favorisce le richieste vicine ai bordi, perché esse possono essere servite sia in un veroso che in un'altro, potrebbe anche favorire anche le richieste appena arrivate se queste si trovano sulla stessa traiettoria

## C-Scan

Il C-Scan é una variante dello SCAN, in pratica c'è una delle due direzioni della testina che non accetta richieste, in questo modo il problema delle richieste vicine ai bordi viene risolto.

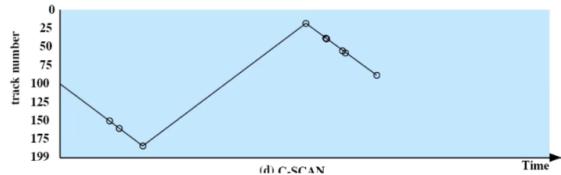


Figure 78: C-SCAN

### F-Scan

Il F-Scan è una variante dello SCAN, in pratica se arrivano nuove richieste le metto in una coda a parte, ad esempi: ho una coda F (front) e R (rear), l'idea è che scan serve le richieste dentro F, e se arrivano nuove riechieste le aggiungo a R, quando F è vuota, allora scambio F con R.

### N-step-SCAN

L' N-step-SCAN è una variante dello SCAN, in pratica è la generalizzazione dell'F-SCAN, se N è alto, le prestazioni sono quelle di SCAN, Con N = 1 si preferisce comunque usare FIFO.

### Confronto Prestazioni

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
55	45	90	10	150	50	150	50
58	3	38	32	160	10	160	10
39	19	55	3	134	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length		Average seek length		Average seek length		Average seek length	
55.3		27.5		27.8		35.8	

Figure 79: Confronto Prestazioni

Algoritmo	Descrizione	Note
<b>Selezione a cura del richiedente</b>		
RSS	Scheduling random	Solo per simulazioni e confronti
FIFO	A coda semplice	Il più equo
LIFO	Ultimo utente	A sorpresa, può andare bene
PRI	Priorità del processo	Se la priorità è importante
<b>Selezione sulla base del dato richiesto</b>		
SSTF	Tempo di servizio più corto	Alto uso del disco, piccole code
SCAN	Avanti ed indietro sul disco	Miglior distribuzione del servizio
C-SCAN	Avanti, con ritorno veloce	Minore variabilità di servizio
N-step-SCAN	SCAN su N richieste	Garanzia del servizio
FSCAN	N-step-SCAN con due code	Tiene conto del carico

Figure 80: Confronto Prestazioni

Cache del disco La cache del disco anche chiamata page cache è un altro buffer dedicato al disco, in particolare per alcuni settori, essenzialmente è una cache quindi contiene una copia di alcuni settori del disco, essendo una cache ha sempre lo stesso obiettivo, ovvero prima di andare sul disco si controlla se il dato è presente in cache oppure no, questa cache è da notare che è situata nella RAM, da notare che questa è una cache software non hardware, non bisogna confonderla con la cache hardware presente nei dischi, nei dischi moderni quando viene fatta una richiesta i dischi vanno a vedere la stessa prima di muoversi con la testina.

### 5.11.1 Gestione della Cache

#### Usato meno recentemente (LRU)

Se occorre rimpiazzare qualche settore nella cache piena, si prende quello nella cache da piú tempo senza referenze. La cache viene puntata da uno "stack" di puntatori, quello riferito piú di frequente é quello che sta in cima allo stack, ogni volta che trovo un settore nella cache, metto in cima il puntatore facendo una operazione di scorrimento, se devo scegliere quello usato meno di recente, prendo quello in fondo allo stack, in realtá LRU non funziona bene.

#### Least Frequently Used (LFU)

Piuttosto che usare LRU si preferisce usare LFU, ovvero si tiene traccia di quante volte un settore é stato usato, quando si deve rimpiazzare un settore, si prende quello usato meno di frequente, ovviamente per implementarlo serve un contatore per ogni settore che conta quante volte ho fatto l'accesso, l'idea alla base é meno vieni usato meno servi, tuttavia se implementato in maniera semplice potrebbero esserci effetti collaterali, per esempio puó esistere un settore al quale accedo molte volte di fila, e poi non lo uso piú, in questo caso il settore non verrebbe mai rimpiazzato.

#### Ibrido LRU-LFU

Per migliorare la situazione si fa un ibrido tra LRU e LFU, quindi si ha uno stack di puntatori, però in realtá questo stack é spezzato in 2 parti con una parte che é nuova ed una che é vecchia, quindi l'idea è che ogni volta ho un riferimento che é nella cache l'incrementamento avviene soltanto se sono nella parte vecchia.

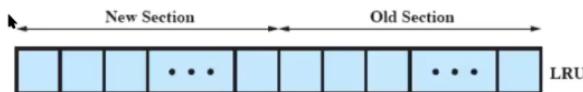


Figure 81: Ibrido LRU-LFU

Quello rappresentato sopra é lo stack di puntatori quindi punta alla cache, come si vede c'è una parte nuova e una vecchia, se quello che sta a sinistra é il most recently used, quello che sta a destra é il least recently used quello che dovrei prendere se usassi LRU standard, quindi se io ho un processo che richiede un settore, che é nella sezione nuova lo sposto in cima alla sezione ma non incremento il contatore, lo incremento solo se si trovava nella sezione vecchia, se io ho un miss, carico il settore nuovo con il contatore con valore 1, quindi il vantaggio che se io mi riferisco spesso ad un settore, nuovo siccome il contatore non viene modificato, posso evitare che nonostante le numerose chiamate in un breve periodo di tempo, il settore rimanga in memoria perché ha un counter con un valore alto, dalla parte nuova alla parte vecchia per scorrimento, invece dalla parte vecchia alla parte nuova per riferimenti, é da notare che c'è ancora un problema se non c'è presto un riferimento ad un blocco, questo potrebbe essere rimpiazzato, perché i riferimenti a lui non arrivano, abbastanza velocemente, per esempio io mi riferisco al blocco **A** ed ha il contatore 1, arrivano altri riferimenti a **B** e **C** puó succedere che il blocco **A** venga rimpiazzato, perché i suoi riferimenti tardano ad arrivare e per questo motivo lui scorre verso la parte vecchia fino ad essere sostituito.

#### Sostituzione Basata su frequenza 3 Segmenti

Per ovviare al problema dello scorrimento, invece di avere 2 parti, si hanno 3 parti, in questo modo i contatori vengono incrementati sia nella parte di mezzo che in quella vecchia, per contro

si puó rimpiazzare solo nella parte vecchia, per il resto é uguale a prima, anche se non arriva presto il riferimento la probabilitá di essere sostituito é minore.

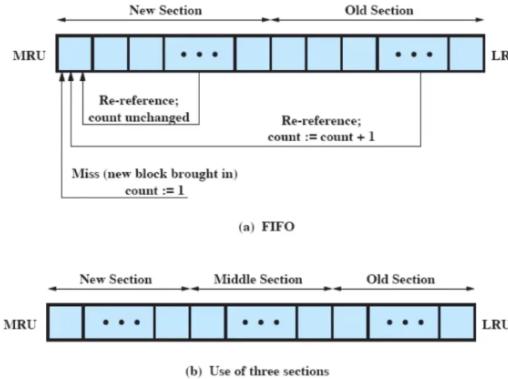


Figure 82: Sostituzione Basata su frequenza 3 Segmenti

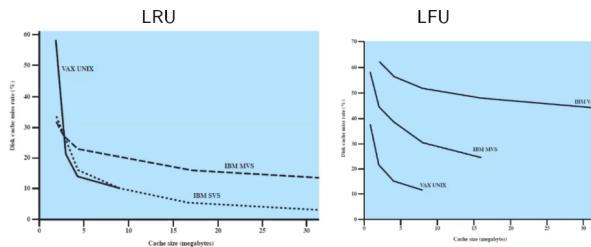


Figure 83: Risultati Esperimenti Cache Disco

LRU é meglio perché sono studi diversi: se la sequenza é la stessa, conviene usare LFU.

## 5.12 RAID

Un altra cosa importante quando si parla di I/O é il RAID, che significa Redundant Array of Independent Disks, ovvero un insieme di dischi indipendenti che lavorano insieme, e ci sono diversi modi per farli lavorare insieme, é comunque possibile trattarli separatamente per esempio, Windows li mostrerebbe esplicitamente come dischi diversi, con etichette diverse , mentre Linux, si potrebbe dire che alcune directory sono in un disco, altre in un altro, quindi si specifica all'inizio, oppure si possono considerare diversi dischi fisici come un unico disco logico.

### Dischi Multipli

Possibilitá ovvia : Linux LVM (Logical Volume Manager), alcuni files/directory sono memorizzati su un disco, altri su un altro, ci pensa una parte del kernel, LVM appunto, l'utente puó non occuparsi di decidere dove salvare i file, con la tecnica del mount di directory diversi, potrebbe succedere che una directory cresca fio a riempire il relativo disco, mentre l'altra resta vuota, LVM é fatto apposta per tenere conto di questo problema.

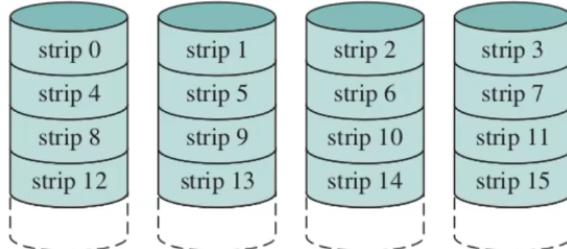
### RAID

L' LVM va bene per pochi dischi, ed in generale se non si é interessati alla ridondanza, questa parte di ridondanza LVM non la gestisce, per questo esiste il RAID, il RAID era stato pensato

per la ridondanza ma con esso si riescono anche a velocizzare alcune operazioni, esistono device composti da piú dischi fisici gestiti da un RAID direttamente a livello di dispositivo.

### Dischi RAID : gerarchia

#### RAID 0

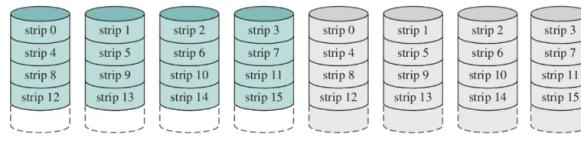


(a) RAID 0 (nonredundant)

Figure 84: Dischi RAID

Nell'raid 0 i dati sono distribuiti per maggiore efficienza nell'accesso e non c'è ridondanza, perché puó essere parallelo, i dischi sono divisi in strip, ogni strip contiene un certo numero di settori, un insieme di strips sui vari dischi (una riga) si chiama stripe.

#### RAID 1

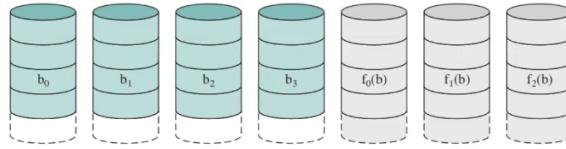


(b) RAID 1 (mirrored)

Figure 85: Dischi RAID

Il RAID 1 é come il RAID 0 ma duplicando ogni dato, fisicamente si hanno  $2N$ , ma la capacità di memorizzazione é  $N$ , in pratica si scrive su entrambi i dischi, se si rompe un disco, sono sicuro di recuperarlo, se se ne rompono 2 bisogna vedere quali si sono rotti

#### RAID 2

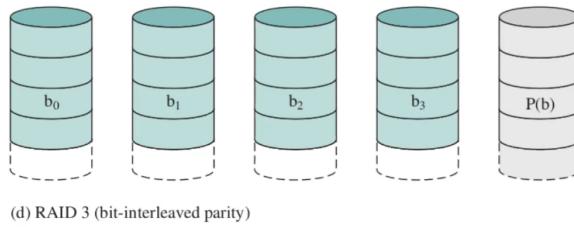


(c) RAID 2 (redundancy through Hamming code)

Figure 86: Dischi RAID

Nell'RAID 2 non si usa la copia 1:1 ma si usa la paritá, l'idea quindi é quella che invece di replicare l'intera informazione utilizzo un opportuno codice di paritá, che serve a correggere i dati, l'overhead é logaritmico

### RAID 3

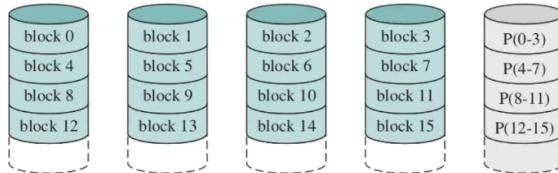


(d) RAID 3 (bit-interleaved parity)

Figure 87: Dischi RAID

Nell'RAID 3 abbiamo un solo disco di overhead, in pratica per ogni bit il disco in piú memorizza la paritá, comunque resta possibile il recupero dei dati, quando fallisce un unico disco oppure il disco che contiene la paritá.

### RAID 4

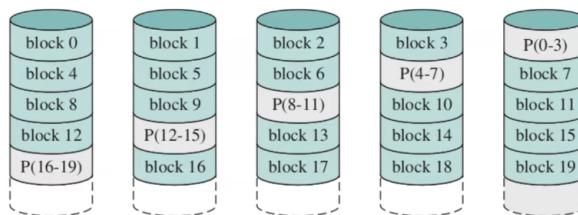


(e) RAID 4 (block-level parity)

Figure 88: Dischi RAID

Il raid 4 é simile al raid 3, ma invece di funzionare di bit in bit, funziona a livello di blocco.

### RAID 5



(f) RAID 5 (block-level distributed parity)

Figure 89: Dischi RAID

Nel RAID 5 la paritá é distribuita su tutti i dischi, in pratica non si ha un disco di paritá, ma la paritá é distribuita su tutti i dischi, in pratica si ha un disco di paritá virtuale, in questo modo si ha un miglioramento delle prestazioni, permettendo comunque l'accesso parallelo.

## RAID 6

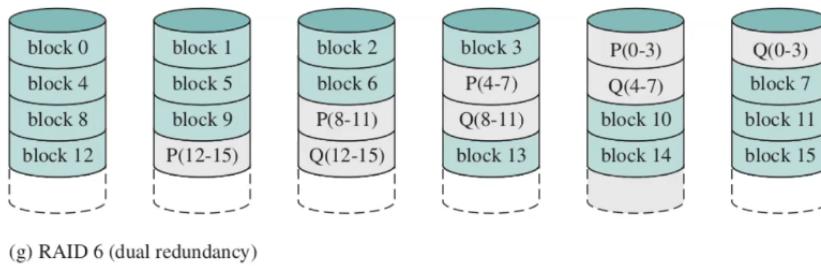


Figure 90: Dischi RAID

Il RAID 6 è simile al RAID 5, ma si hanno 2 dischi di parità virtuali, in pratica si ha una ridondanza maggiore, ed è anche possibile recuperare i dati se falliscono 2 dischi, per le operazioni di lettura con il RAID 5 esiste una somiglianza con il RAID 6 mentre per le operazioni di scrittura il RAID 6 è più lento.

## Riassunto

**Table 11.4** RAID Levels

Category	Level	Description	Disks Required	Data Availability	Large I/O Data Transfer Capacity	Small I/O Request Rate
Striping	0	Nonredundant	$N$	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

Note:  $N$ , number of data disks;  $m$ , proportional to  $\log N$ .

Figure 91: Confronto di tutti i RAID

è da notare che se faccio una operazione sul RAID, tutti i dischi effettuano in sincrono, un'operazione su un raid è una operazione su un sottoinsieme dei suoi dischi, il vantaggio di avere un RAID diverso da 0 è possibile recuperare i dati se fallisce un disco, smallI/O Request rate descrive il modo come i dischi gestiscono piccole richieste di I/O

## I/O Linux

Linux soprattutto per quanto riguarda la gestione della page cache, la page cache è unica e combatte nella memoria destinata agli utenti per tutti i trasferimenti compresi quelli dovuti alla gestione della memoria virtuale, questo porta 2 vantaggi : condensare le scritture sul disco

siano esse dovute a I/O o alla gestione della memoria virtuale, inoltre serve anche per sfruttare la località dei riferimenti, si scrive sul disco o quando è rimasta poca memoria: una parte della page cache è ridestinata ad uso diretto dei processi, quando l'età delle pagine "sporche" va sopra una certa soglia, Niente politica separate di replacement: è la stessa utilizzata per la sostituzione delle pagine (algoritmo dell'orologio).

## 6 File System

Il file system è una parte del sistema operativo che si occupa di organizzare i file all'interno dei dischi

### 6.1 I File

Sono l'elemento principale per la maggior parte delle applicazioni, molto spesso, l'input di un'applicazione è un file; quasi altrettanto spesso, l'output è un file, i file sopravvivono ai processi, il file system è una delle parti del sistema operativo che sono più importanti per l'utente, le propietà che il SO deve garantire sono :

- i file devono esistere a lungo termine
- condivisibilità con altri processi (tramite nome simbolici)
- strutturabilità (directory gerarchiche)

### Gestione dei file

I file sono gestiti da un insieme di programmi e librerie di utilità, tali programmi sono gestiti in kernel mode, le librerie vengono invocate come system call, ed hanno a che fare soprattutto con la memoria secondaria, inoltre in linux è possibile gestire porzioni di RAM come se fossero file, forniscano un'astrazione sotto forma di operazioni tipiche per ogni file vengono mantenuti degli attributi tipo nome permessi ...

### Operazioni sui file

- Creazione (con annessa scelta del nome)
- Cancellazione
- Apertura
- Lettura
- Scrittura
- Chiusura

le operazioni di lettura e scrittura sono possibili solo se il file è stato aperto

### Terminologia

- Campo (field)
- Record
- File
- Data Base

**Campi**

- dati di base
- contengono valori singoli
- caratterizzati da lunghezza e tipo di dato (o demarcazioni)
- esempio tipico: ASCII

**Record**

- Insieme di campi
- ognuno trattato come un'unità

**File**

- Hanno un nome
- Sono insiemi di record correlati in ogni file ogni record è un solo campo con un byte
- Ogni file è trattato come unità con nome proprio
- possono implementare meccanismi di controllo di accesso

**DataBase**

- Sono un'opportuna collezione di file
- sono gestiti dai DBMS che sono processi di un sistema operativo
- Nei system operativi moderni non è più necessario gestire File attraverso i data base

## Sistemi per la gestione di file



I file management Systems forniscono servizi agli utenti e alle applicazioni per l'uso di file e definiscono anche il modo in cui i file sono usati, sollevando i programmatore dal dover scrivere codice per gestire i file

### 6.2 Obiettivi per il File Management Systems

- Rispondere alle necessita degli utenti riguardo alla gestione dei dati
- Garantire che i dati nei file siano validi
- Ottimizzare le prestazioni : sia dal punto di vista di throughput che tempo di risposta

- Fornire supporto per diversi tipi di memoria secondari
- minimizzare i dati per o distrutti
- Fornire un insieme di interfacce standard per i processi utente
- Fornire supporto per l'I/O effettuato da piú utenti in contemporaneamente

## Requisiti

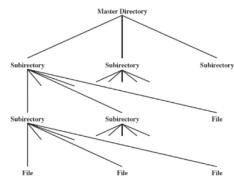
1. Ogni utente deve essere in grado di creare, cancellare, leggere, scrivere e modificare un file
2. Ogni utente deve poter accedere, in modo controllato, ai file di un altro utente
3. Ogni utente deve poter leggere e modificare i permessi di accesso ai propri file
4. Ogni utente deve poter ristrutturare i propri file in modo attinente al problema affrontato
5. Ogni utente deve poter muovere dati da un file ad un altro
6. Ogni utente deve poter mantenere una copia di backup dei propri file (in caso di danno)
7. Ogni utente deve poter accedere ai propri file tramite nomi simbolici

### 6.2.1 Directory

Le directory contengono informazioni sui file (attributi, posizione, proprietario) Una directory è essa stessa un file (speciale) e fornisce il mapping tra i nomi dei file e i file stessi, un altro modo per chiamare gli attributi di un file è **metadati**, le operazioni che sono possibili sulle directory sono:

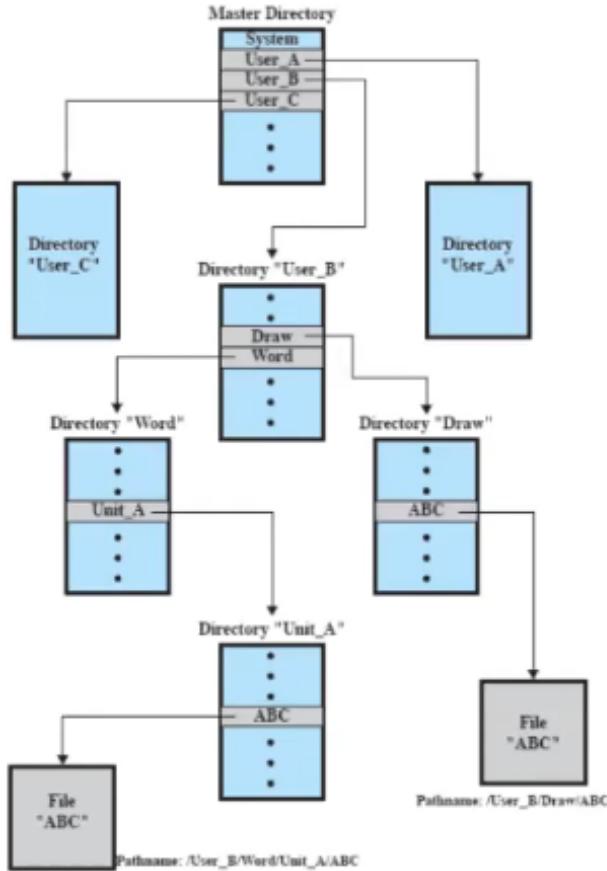
- Ricerca
- Creazione
- Cancellazione
- Lista del contenuto
- Modifica della directory

Le informazioni di base sono: il nome del file che deve essere unico per ogni directory, Tipo di file, Organizzazione del file, il volume che indica il dispositivo su cui il file è memorizzato, indirizzo di partenza del file, dimensione attuale, dimensione allocata, il proprietario (può dare i permessi di accesso), informazioni sull'accesso (potrebbe contenere username e password per ogni utente autorizzato), le Azionimesse, poi ci sono anche i metadati sull'uso: Data di creazione, Identità del creatore, Data dell'ultimo accesso in lettura, Data dell'ultimo accesso in scrittura, Identità dell'ultimo lettore, Identità dell'ultimo scrittore, Data dell'ultimo backup, Uso attuale (Lock, azione, corrente, ...). Struttura semplice per le Directory. Inizialmente le directory era una lista di file, che rappresentava tutti i file presenti nel dispositivo. Schema a due livelli. Una directory per ogni utente, più una (master) che le contiene, la master contiene anche l'indirizzo e le informazioni per il controllo dell'accesso. Schema ad albero. Nello schema ad albero, è presente una directory master che contiene le directory utente, ed ogni directory utente contiene a sua volta delle directory utente, questo schema è molto simile a quello a due livelli, ma permette una maggiore organizzazione.



### 6.2.2 Nomi

Gli utenti devono potersi riferire ad un file usando solo il suo nome, i nomi devono essere unici, ma un utente può non avere accesso a tutti i file, quello che succede siccome è presente una struttura ad albero c'è una directory di base (root) se devo accedere ad una specifica directory devo specificare il percorso, che permette agli utenti di trovare i file, questo permette di avere file con nomi uguali su directory diverse.



### 6.3 Directory di Lavoro

A questo punto ogni volta che devo accedere ad un file dovrei specificare tutto il percorso dalla root directory fino al file, per ovviare a questo problema è stata introdotta la directory di lavoro, quindi ogni processo ha una directory di lavoro quindi chi ha bisogno di accedere ad un file, deve specificare solo il percorso relativo alla directory di lavoro.

## 6.4 Gestione Della Memoria Secondaria

Il SO é responsabile dell'assegnamento di blocchi a file, ci sono due problemi correlati:

- decidere su quali byte del disco memorizzare i file
- mantenere traccia di dove ho memorizzato i file

Come si vede un problema é correllato all'altro, esistono tanti modi per farlo ogni modo definisce un nuovo file system, i file si allocano in porzioni o blocchi, l'unità minima é il settore del disco, ogni porzione o blocco é una sequenza contigua di settori

### 6.4.1 File Allocation

ci sono diversi problemi d'affrontare:

- pre-allocatione vs allocazione dinamica
- Porzioni di dimensione fissa o dinamice e quanto grandi
- Metodi di allocazione : contiguo, concatenato, indicizzato
- Gestione della file allocation Table

#### pre-allocatione vs allocazione dinamica

La pre-allocatione vuol dire che: sto creando un file, sul disco alloco immediatamente la dimensione massima che il file mi richiederá, nei file system che fanno uso di pre-allocatione, il file system deve sapere in anticipo la dimensione massima, si intuisce che sistemi operativi moderni Linux non fanno uso di pre-allocatione, in alcuni casi é stimabile la dimensione massima di un file(Compilazione di un programma) in altri casi alcune applicazioni tendono a sovrastimare e quindi si spreca spazio, per questo motivo l'allocazione dinamica é preferibile, quindi si comincia con una dimensione minima che é concessa e mano a mano che aggiungo informazioni aggiusto la dimensione, per fare ciò si usano due system call: **append** e **truncate**

#### Porzioni di dimensione fissa o dinamice e quanto grandi

Abbiamo due possibilità agli estremi:

- Si alloca una porzione che é larga a sufficienza per l'intero file, questa soluzione é molto efficiente perché alloco in maniera contigua
- Si alloca un blocco alla volta, questa soluzione é molto efficiente per un sistema operativo che deve gestire molti file, ciascun blocco é una sequenza di n settori contigui

Per le prestazioni di accesso, é meglio per l'utente avere blocchi di grandi dimensioni, dal punto di vista del sistema operativo potrebbe essere meno efficiente, fare porzioni molto piccole significa che la struttura dati che devo mantenere per tenere traccia dei blocchi é molto grande, al contempo si facilita la frammentazione.

Alla fine abbiamo 2 possibilità:

- Porzioni Grandi di dimensione **variabile** : ogni singola allocazione é contigua, tabella di allocazione contenuta, complicata la gestione dello spazio libero: servono algoritmi ad hoc
- Porzioni fisse e piccole : tipicamente 1 blocco 1 porzione, molto meno contiguo del precedente, spazio libero (Guardare la tabella dei bit).

Da notare che ci sono alcune combinazioni naturali, per esempio se sono in un sistema che fa pre-allocazioni, allora faccio le porzioni grandi, essenzialmente per ogni file mi serve sapere solo dove inizia e dove finisce e quindi non ho bisogno della tabella di allocazione, ogni file è quindi in un'unica porzione, e come per il porzionamento della RAM posso usare : best fit, first fit, next fit, ma essendoci troppe variabili in questo caso non esiste un algoritmo migliore di un'altro, questo sistema è inefficiente per lo spazio libero infatti necessita una periodica compattazione e compattare il disco è molto più oneroso che compattare la RAM perché è un I/O.

## Metodi di allocazione

	Contiguous	Chained	Indexed	
<b>Preallocation?</b>	Necessary	Possible	Possible	
<b>Fixed or variable size portions?</b>	Variable	Fixed blocks	Fixed blocks	Variable
<b>Portion size</b>	Large	Small	Small	Medium
<b>Allocation frequency</b>	Once	Low to high	High	Low
<b>Time to allocate</b>	Medium	Long	Short	Medium
<b>File allocation table size</b>	One entry	One entry	Large	Medium

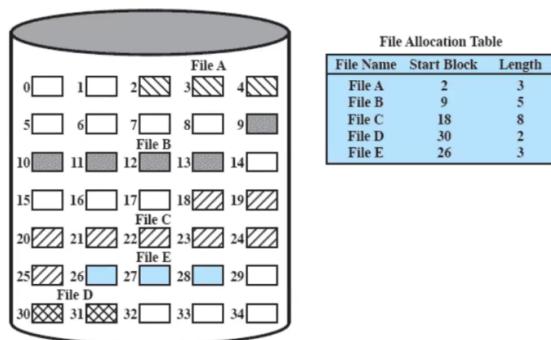
Nella realtà si utilizzano questi tre metodi per allocare spazio:

- Contiguo
- Concatenato
- Indicizzato

Ci sono per ciascuno di questi metodi ci sono delle caratteristiche, ad esempio se voglio fare allocazione contigua la pre-allocazione è obbligatoria, Time to allocate indica l'overhead per il sistema operativo.

### Contiguo

Per allocare un file contiguo, significa che io conosco già a priori la memoria che questo file richiederà, quindi è necessaria la pre-allocazione, se un file prova a crescere oltre la dimensione massima, il sistema operativo si rifiuta, è necessaria una sola entry nella tabella di allocazione dei file, inoltre sarà presente frammentazione esterna.



Come si vede dall'immagine sopra i file sono tutti allegati in maniera contigua, e la FAT ha bisogno di poche informazioni per tenere traccia di dove inizia e finisce il file, è presente lo stesso problema presente nella ram ad esempio se arriva un file che richiede 8 settori contigui siamo costretti a fare compattazione, compattando si cambia la FAT.

### Concatenata

Abbiamo un'allocazione di un blocco alla volta, il blocco è costituito in 2 parti, una parte che contiene i file e una piccola parte che contiene il puntatore al prossimo blocco, con questo modo non abbiamo più frammentazione esterna quella interna invece è trascurabile, questo modo va bene se dobbiamo accedere ad un file sequenzialmente, con il consolidamento serve per mettere i blocchi il più vicino possibile per migliorare le prestazioni.

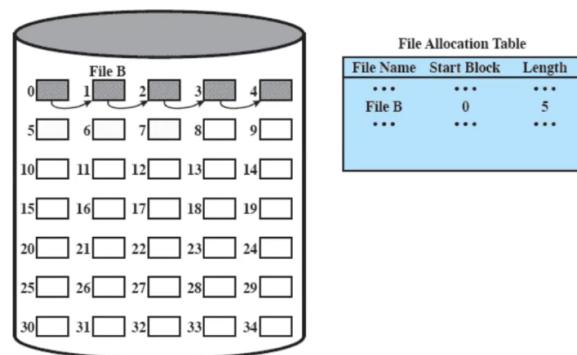
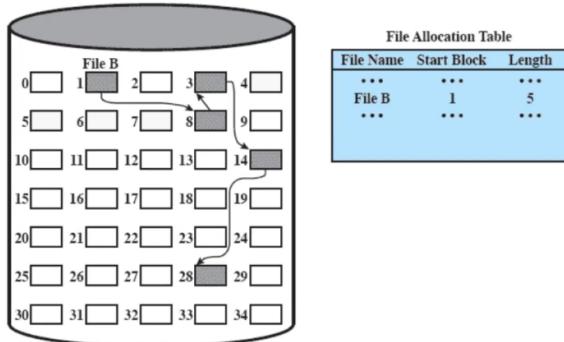
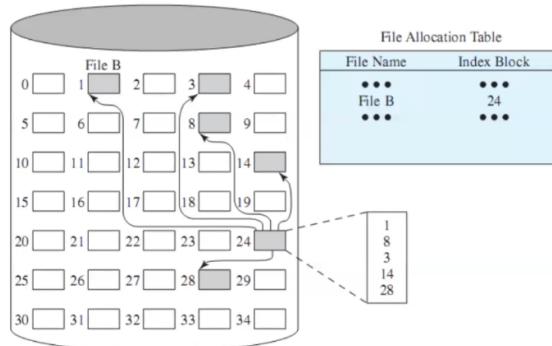


Figure 92: Consolidamento

### Indicizzata

Quello che si utilizza nei file system moderni è l'allocazione indicizzata, sostanzialmente fa una via di mezzo tra una allocazione contigua e una concatenata, l'idea è che ci sono dei blocchi che contengono i dati ed altri che hanno dei puntatori.



La File allocation table si limita a dire quale é il blocco indice di questo file, nel nostro caso il blocco 24 dove troviamo tutti i puntatori ai blocchi che contengono i dati per quel determinato file.

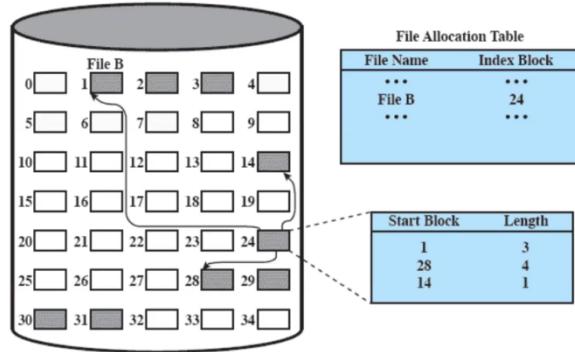


Figure 93: Allocazione indicizzata variabile

Nel caso in cui la lunghezza delle porzioni sia variabile il blocco indice deve contenere anche la lunghezza del singolo tratto di blocchi contigui.

L'allocazione indicizzata puó essere fatta sia con blocchi di lunghezza fissa che con blocchi di lunghezza variabile, se si usa la lunghezza fissa non abbiamo frammentazione esterna, ma se usiamo la lunghezza variabile abbiamo prestazioni migliori con rischi di frammentazione esterna.

## 6.5 Gestione dello Spazio Libero

Per Gestire lo spazio libero, serve una tabella di allocazione di disco, oltre a quella di allocazione per i file ogni volta che si alloca o cancella un file si deve aggiornare la tabella di allocazione, ci sono due metodi per tenere traccia dello spazio:

- BitMap
- Lista di blocchi liberi

### BitMap

Faccio un vettore con un bit per ogni blocco su disco, va bene per tutti gli schemi visti finora, questo modo minimizza lo spazio richiesto, ma il problema é che se il disco é molto pieno, la ricerca di uno spazio libero diventa molto lenta, si puó risolvere facendo delle tabella riassuntive della tabella di bit, in modo da avere un'idea di dove cercare.

### Porzioni libere concatenate

Le porzioni libere concatenate sono una soluzione senza overhead di spazio, uso lo stesso sistema della allocazione concatenata solo per i blocchi liberi, se c'é molta frammentazione esterna, le porzioni sono tutte da un blocco e la lista diventa molto lunga, occorre leggere un blocco libero per sapere qual'é il prossimo, é dispendioso cancellare i file molto frammentati.

### Lista dei blocchi liberi

É presente una zona di memoria che va riservata, e ci metto una lista di indici (numeri di blocchi che sono liberi), potrebbe sembrare poco intelligente, in realtà l'overhead puó essere molto basso, supponiamo che per ogni blocco bastano 4 bytes, se mi servono 4 bytes per l'indirizzo ma i blocchi sono da 512 byte, richiede meno dell'1% di spazio su disco, per avere parti della lista in memoria principale si puó:

1. organizzare la lista come pile, e tenere solo la parte alta
2. pop per allocare spazio libero, push per deallocare spazio occupato
3. quando la parte in memoria principale finisce, si prende una nuova parte da disco
4. Funziona anche come coda

## 6.6 Volumi

Un volume é un disco "logico" che sarebbe una partizione di un disco, oppure piú dischi assieme LVM, un insieme di settori in memoria secondaria, che possono essere usati dal SO o dalle applicazioni, i settori di un volume non necessariamente devono essere contigui.

## 6.7 Dati e Metadati

### Consistenza

I Dati sono il contenuto del file, mentre i metadati sono tutto il resto, i metadati devono essere tutti sul disco, per efficienza dati e metadati possono essere in memoria principale quando si apre un file, c' é un problema di consistenza quando il file é su RAM ovvero che risulta molto oneroso mantenere i metadati consistenti in tutte e due le memorie quindi l'aggiornamento si fa solo quando il disco é poco usato e ci sono tanti aggiornamenti tutti insieme, ad oggi con il **Journaling** si é risolto questo problema, anziché fare le modifiche on demand, quello che si fa é raccogliere in un file la lista di tutti i cambiamenti che si devono fare e poi si fanno tutti insieme, in questo modo si evita di fare tanti accessi al disco, se c' é un crash basta scrivere un bit all'inizio del disco, che dice se il sistema é stato correttamente spento o no, se non é stato spento correttamente si esegue un programma per il ripristino del disco, con il journaling questo ripristino é piú facile perché basta consultare il log.

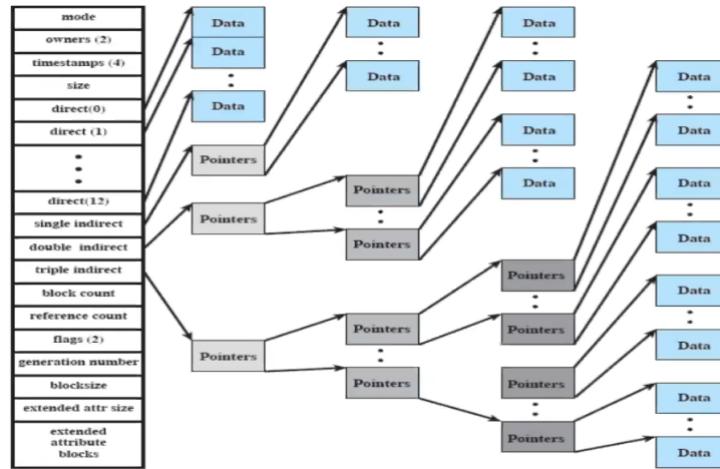
## 6.8 Gestione dei File in Unix

In Unix ci sono 6 tipi di file:

- normale
- directory
- speciale (mappano su nomi di file i dispositivi di I/O)
- named pipe (Per far comunicare due processi)
- hard link (Collegamenti, nome di file alternativo)
- symbolic link (Collegamenti simbolici)

### Inode

Inode é un insieme di informazioni che riguardano il file(directory,file word ...) e sono memorizzate in un blocco di memoria, ogni file ha un inode



In Unix si usa il concetto di Inode, che é un tipo particolare di allocazione indicizzata, Inode significa index node, perché i blocchi stessi sono usati come indici per trovare i blocchi che contengono i dati, un certo Inode puó essere associato a piú file (hard link), oltre al fatto che contiene informazioni, c'é anche un numero che é associato ad ogni Inode per rendere piú veloce la ricerca, l'idea é che ogni volta che creo un file creo un Inode e gli assegno un numero, quando cancello il file il numero puó essere riutilizzato, alcuni Inode sono su disco, altri sono in memoria principale, quelli in RAM sono quelli che sono aperti da processi in esecuzione, tutti gli Inode sono in un'area dedicata del disco.

I dati che sono presenti in un Inode sono:

- Tipo e modo di accesso del file
- Identificatore del proprietario
- **Tempo di creazioone e di ultimo accesso (lettura o scrittura)**
- Flag utente e flag per il kernel
- numero sequenziale di generazione del file
- Dimensione delle informazioni aggiuntive
- altri attributi (Controllo di accesso e altro)
- Dimensione
- Numero di blocchi, o numero i file (per le directory)
- Dimensione dei blocchi
- **Sequenze di Puntatori ai blocchi di dati**

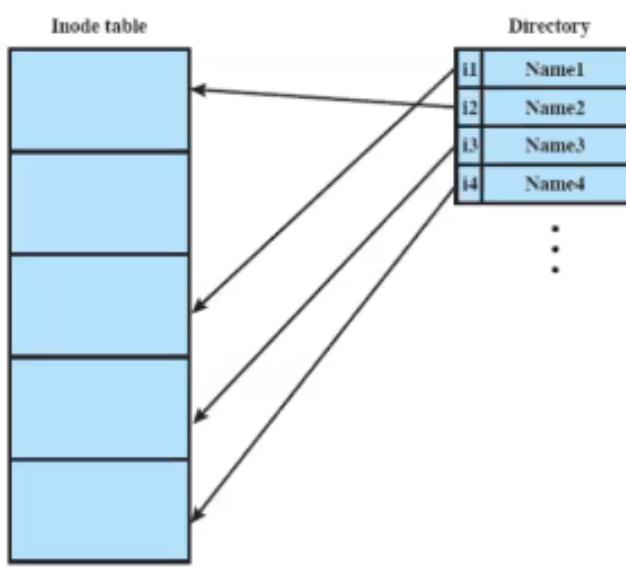
Se il file é piccolo, i dati sono puntati direttamente dall'Inode (direct nell'immagine dell'Inode), se il file é grande allora si passa agli altri puntatori, il concetto é quello di indicizzazione, quindi ho un indirizzo che punta ad una memoria dove sono presenti altri indirizzi (questo é fatto anche su piú livelli (Single, Double, Triple Indirect)), in questo modo non serve che i blocchi vengano contrassegnati se contengono dati o indirizzi, perché il sistema operativo sa che con il triplo indirizzamento, il file system deve attraversare 3 blocchi per trovare i dati.

- Fatta a blocchi
- Allocazione dinamica
  - quindi, blocchi potenzialmente non contigui
- L'indicizzazione tiene traccia dei blocchi dei file
  - parte dell'indice è memorizzata nell'inode
- L'inode ha anche alcuni puntatori diretti
  - e 3 puntatori indiretti

Level	Number of Blocks	Number of Bytes
Direct	12	48K
Single Indirect	512	2M
Double Indirect	$512 \times 512 = 256K$	1G
Triple Indirect	$512 \times 256K = 128M$	512G

## Inode e Directory

Le directory sono file che contengono una lista di coppie (nome di file, puntatore all'inode), alcuni di questi file potrebbero essere a loro volta directory, quindi è una struttura gerarchica, una directory può essere modificata solo dal SO, mentre può essere letta da ogni utente.



Nell'immagine abbiamo una tabella fatta da 2 colonne, 1 nome del file o della sotto-directory e nell'altra colonna l'inode.

Più file vengono messi dentro una directory, più file vengono messi dentro ad una directory diventa grande, per un file che rappresenta una directory generalmente bastano i direct, nei casi in cui non bastino i direct anche i file che rappresentano le directory potranno usare single, double e triple indiretti.

## Accesso ai File

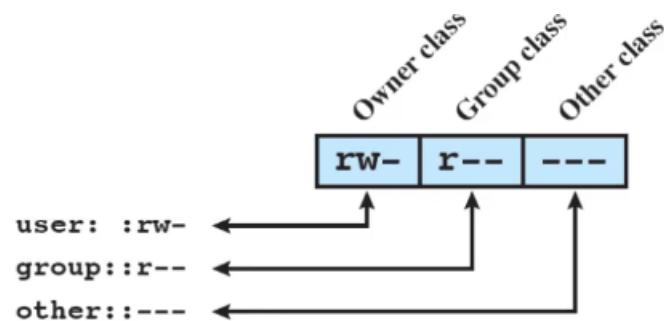
Sotto Linux ci sono gli utenti, dobbiamo essere abilitati come utente abilitato ad accedere al sistema, quindi quello che succede ogni processo viene eseguito da un qualche utente, quindi se un processo crea un file questo file avrà come proprietario l'utente che ha eseguito il processo, un'altra cosa che succede dentro linux è che ci sono i gruppi, quindi un utente può appartenere a più gruppi, l'idea è che per l'accesso ci interessano 3 tipi di permessi:

- Chi puó leggere il file
- Chi puó scrivere il file
- Chi puó eseguire il file

di queste terne di permessi ce ne sono 3:

- Proprietario
- Gruppo
- Altre classi

se c' un utente che cerca di leggere un certo file, guarda chi  il proprietario, se il proprietario coincide con l'utente che legge il file, se non coincide si guarda se l'utente appartiene al gruppo, se non appartiene al gruppo si guarda se  un altro.



(a) Traditional UNIX approach (minimal access control list)

## 6.9 Gestione dei File in Windows

Windows ha due tipi di file system:

- **FAT** allocazione concatenata, con blocchi (cluster) di dimensione fissa
- **NTFS** allocazione con bitmap(!), con blocchi di dimensione fissa

### FAT

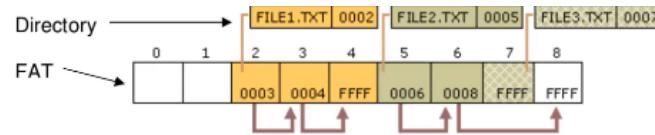
FAT andava bene per i vecchi dischi, ma viene ancora usata per le chiavette USB, Formattare significa preparare il disco in maniera d'accogliere le informazioni di un certo file system (Es. per l'allocazione inidicizzata avremmo tutto il disco libero e avremo un File Allocation Table a cui verr riservata un certo spazio)

Dimensione in settori	Area riservata		FAT		Root directory (solo FAT12/16)	Regione dati
	Settore di avvio	Informazioni FS (solo FAT12)	Riservati (opzionale)	FAT #1	FAT #2	
	Variable			(# FAT)*(settori per FAT)	32 * (# voci root) / bytes per settore	(# cluster) * (settori per cluster)

Nell'immagine  presente una parte con la **file allocation table** che contiene le informazioni su dove sono allocati i file, poi c' lo spazio per i dati, ed infine un area riservata, della file allocation table ci sono 2 copie, una  la copia principale e l'altra  la copia di backup, la file allocation table  organizzata come una lista questa colonna contiene un valore che pu essere a : 12 bit, 16 bit, 32 bit dipendentemente dalla versione di FAT, questa lista  lunga quanti sono i cluster del disco, ogni cluster pu essere di dimensione che va da 2 a 32 KB, se scelgo 32KB e il disco  32MB avr che la lunghezza della lista sar 1000, la dimensione si pu scegliere in fase di formattazione.

## File Allocation Table

Se la entry i-esima é 0 allora il cluster i-esimo é libero, se l' i-esima entry é diversa da 0 e non é un valore speciale allora indica qual'é il prossimo blocco di questo file, se per esempio nella riga 10 c'é scritto 5, vuol dire che il blocco 10 contiene i dati del file e i prossimi dati sono nel blocco 5, é abbastanza simile all'allocazione concatenata.



All'inizio della regione dati c'è il **root directory** piú o meno simile ad Inode indica nome del file e **primo blocco**, uno dei valori speciali é tutti 1, che indica che il file é finito.

Byte offset	Lunghezza (byte)	Descrizione
0x00	8	Sella dimensione. Questo settore verrà integrato e indicato che il resto dell'intera traccia (non eseguibile) se lo percorre. È possibile (in NTFS), che il valle è pari a due byte. Intercette (MAP) viene di seguito da un'intestazione MFT.
0x00	8	Nome CSD (metadati di spazio). Determina come il disco è stato formattato. MFT-DOS controlla questo campo per determinare quali altre parti del record avranno risorse valutate. I valori comuni sono: 16H, 512H, con due spazi intermedii, 4096H, 8K, 16K.
0x00	2	Byte per settore. Solitamente il valore è 512. È finito dal <b>B05 Parameter Block</b> .
0x00	1	Sette per cluster. I valori ammessi sono potenze di due da 1 a 128. Tuttavia, il numero totale di byte per cluster deve essere inferiore a 512K.
0x00	1	Numero di cluster. Il valore deve essere minore o uguale al numero di cluster. Per esempio, se si ha un cluster da 128 byte, il numero di cluster deve essere minore di 4096. Per esempio, se si ha un cluster da 512 byte, il numero di cluster deve essere minore di 8192.
0x10	1	Numero di blocchi di allineamento del file. Solitamente vale 1.
0x11	2	Numero massimo di voci nella directory root. Per il FAT12/16 questa voce deve essere multiplo della dimensione di un settore. Per il FAT32 vale 0.
0x13	2	Sette totali. Se vale 0 allora la fine è la fine. Se è maggiore di 0, il valore all'offset 0x20.
0x14	1	Flag per la ricerca del file. Se è 1, allora il file è protetto.
0x16	2	Sette per FAT (per FAT12/16).
0x18	2	Sette per Inode.
0x1A	2	Numero di Inode.
0x1C	4	Sette iniziali.
0x20	4	Totale settori. Se minore di 0x532 vale il contenuto all'offset 0x13.

Figure 94: Informazioni Area Riservata

FAT12	FAT16	FAT32	Descrizione
0x000	0x0000	0x00000000	Cluster libero
0x001	0x0001	0x00000001	Valore riservato
0x002-0xFFE	0x0002-0xFFFF	0x00000002-0xFFFFFFFF	Cluster dati
0xFF0-0xFF6	0xFFFF0-0xFFFF6	0xFFFFFFFF0-0xFFFFFFFF6	Valori riservati
0xFF7	0xFFFF7	0xFFFFFFFF7	Cluster danneggiato
0xFF8-0xFFFF	0xFFFF8-0xFFFF	0xFFFFFFFF8-0xFFFFFFFF	Ultimo cluster

Figure 95: Valori speciali

## Come sono fatte le directory

Tutti i metadati che in linux sono nell'Indoe in windows sono nella directory.

Byte offset	Lunghezza	Descrizione																											
0x00	8	Nome del file (riempito da spazi). Il primo byte se vale 0 significa che la voce è disponibile. Il valore 0x05 è riservato per la codifica kanji, il valore 0xE indica il valore '' o ''.																											
0x08	3	Estensione del file (riempita con spazi)																											
0x0B	1	Attributi del file. Segue il seguente schema:																											
		<table border="1"> <thead> <tr> <th>Bit</th> <th>Maschera</th> <th>Descrizione</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0x01</td> <td>Sola lettura</td> </tr> <tr> <td>1</td> <td>0x02</td> <td>Nascosto</td> </tr> <tr> <td>2</td> <td>0x04</td> <td>Di sistema</td> </tr> <tr> <td>3</td> <td>0x08</td> <td>Elenco volume</td> </tr> <tr> <td>4</td> <td>0x10</td> <td>Sottodirectory</td> </tr> <tr> <td>5</td> <td>0x20</td> <td>Archivio</td> </tr> <tr> <td>6</td> <td>0x40</td> <td>Device (solo per uso interno, in realtà non si trova sul disco)</td> </tr> <tr> <td>7</td> <td>0x80</td> <td>Inutilizzato</td> </tr> </tbody> </table>	Bit	Maschera	Descrizione	0	0x01	Sola lettura	1	0x02	Nascosto	2	0x04	Di sistema	3	0x08	Elenco volume	4	0x10	Sottodirectory	5	0x20	Archivio	6	0x40	Device (solo per uso interno, in realtà non si trova sul disco)	7	0x80	Inutilizzato
Bit	Maschera	Descrizione																											
0	0x01	Sola lettura																											
1	0x02	Nascosto																											
2	0x04	Di sistema																											
3	0x08	Elenco volume																											
4	0x10	Sottodirectory																											
5	0x20	Archivio																											
6	0x40	Device (solo per uso interno, in realtà non si trova sul disco)																											
7	0x80	Inutilizzato																											
0x0C	1	Riservato, generalmente vale 0 (tranne in Windows NT e successivi)																											
0x0D	1	Ora di creazione. Risoluzione: 10ms, valori da 0 a 199																											
0x0E	2	Ora di creazione in ore, minuti e secondi. Le ore sono codificate nei bit 15-11, i minuti nei bit 10-5, i secondi nei bit 4-0																											
0x10	2	Data di creazione. L'anno è codificato nei bit 15-9, il mese nei bit 8-5, i giorni nei bit 4-0. L'anno è calcolato a partire dal 1980.																											
0x12	2	Ultima data di accesso																											
0x14	2	EA-Index in OS2 e NT per FAT12/16, in FAT32 sono riportati i 2 byte alti del primo cluster																											
0x16	2	Ora ultima modifica																											
0x18	2	Data ultima modifica																											
0x1A	2	Puntatore al primo cluster in FAT12/16, in FAT32 sono riportati i 2 byte bassi del primo cluster																											
0x1C	4	Dimensione del file in byte																											

## 6.10 Linux

Linux nativamente supporta 3 tipi di file system:

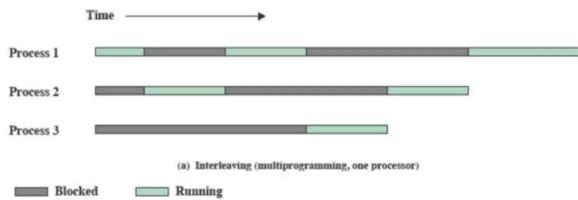


Figure 96: Multiprogrammazione: un solo processore, piú processi in memoria

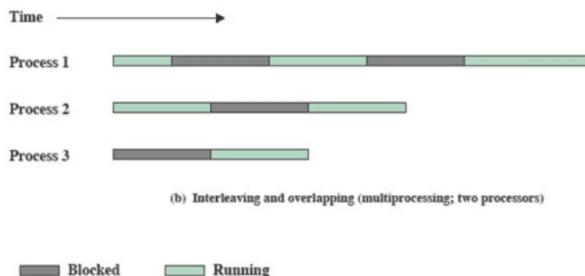


Figure 97: Multiprocessing: piú processori, piú processi in esecuzione

- **Ext2** : File system UNIX, allocazione indicizzata, con blocchi di dimensione fissa
- **Ext3** : Estensione di Ext2, con journaling
- **Ext4** : Estensione di Ext3, con journaling, allocazione indicizzata, con blocchi di dimensione variabile

Ha il pieno supporto per gli Inode, memorizzati nella parte iniziale come per esempio quelli di windows, come FAT; non è possibile memorizzare gli i-node sul dispositivo, vengono quindi creati on the fly, su una cache quando vengono aperti i file

## 7 Gestione della concorrenza

### 7.1 Concetti di base

Per i SO moderni, è essenziale supportare piú processi in esecuzione:

- **multiprogrammazione**: multiplo processi in memoria
- **multiprocessing**: multiplo processi in esecuzione
- **computazione distribuita**: multiplo processi in esecuzione su piú macchine

uno dei problemi principali è la **concorrenza**, ovvero gestire il modo con cui questi processi interagiscono tra di loro. Il problema della concorrenza si presenta in entrambi i casi, può succedere che ci siano applicazioni diverse che sono in esecuzione contemporaneamente, ci sono applicazioni che sono fatte per essere eseguite in parallelo, il sistema operativo stesso può avere piú processi in esecuzione, in tutti questi casi è necessario gestire la concorrenza.

#### Temini chiave

- **Mutua esclusione**: Ci sono 2 o piú processi che vogliono accedere alla stessa risorsa, però solo uno alla volta può accedere.
- **Sezione critica**: Parte del codice di un processo in cui viene effettuato un accesso a una risorsa condivisa.

- **Corsa critica(race condition):** È un caso in cui la mutua esclusione non è rispettata, due o più processi accedono alla stessa risorsa contemporaneamente.
  - **Operazione Atomica:** È una sequenza indivisibile di comandi, nessun altro processo può vedere uno stato intermedio della sequenza o interrompere la sequenza, durante una operazione atomica il dispatcher non può intervenire.
  - **Stallo (Deadlock):** Situazione in cui due o più processi sono in attesa di una risorsa che è posseduta da un altro processo, che a sua volta è in attesa di una risorsa posseduta da uno dei processi in attesa.
  - **Stallo attivo (livelock):** Situazione in cui due o più processi sono in attesa di una risorsa che è posseduta da un altro processo, che a sua volta è in attesa di una risorsa posseduta da uno dei processi in attesa.
  - **Morte per fame (Starvation):** Un processo, pur essendo ready, non viene mai scelto dallo scheduler.

Le ultime tre situazioni devono essere evitate, quindi o il sistema operativo se ne prende carico o è il programmatore che deve evitare questi problemi.

## 7.2 Concorrenza : Difficoltà

La difficoltà principale è che non si può fare nessuna assunzione sul comportamento dei processi, e neanche su come funzionerà lo scheduler, un'altra difficoltà si ha quando si hanno delle risorse condivise (Es. una stampante, oppure due thread che accedono alla stessa variabile globale), per cui è necessario gestire anche l'allocazione delle risorse condivise, per cui diventa impossibile fare una gestione ottima perché c'è il rischio di violare la mutua esclusione (Es. un processo potrebbe richiedere un I/O e poi essere rimesso in ready prima di usarlo: quell'I/O va considerato locked oppure no ?), diventa inoltre difficile tracciare gli errori di programmazione, spesso il manifestarsi di un errore dipende dallo scheduler e dagli altri processi presenti, rilanciare l'ultimo processo spesso non riproduce lo stesso errore (Es. è uscita una stampa di due documenti assieme, provo a rilanciare il processo dopo un errore ma la seconda volta va a buon fine).

```
/* chine chout sono due variabili globali */
void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

switch a P1, P1 esegue il putchar() e poi lo switch a P2, P2 esegue il putchar() e poi lo switch a P1, P1 esegue il putchar() e poi lo switch a P2, P2 esegue il putchar() e poi lo switch a P1, P1 esegue il putchar() e poi lo switch a P2, P2 esegue il putchar() e poi lo switch a P1, P1 esegue il putchar() e poi lo switch a P2, P2 esegue di nuovo getchar(), quando rientra P1 cono l'assegnazione di chout = chin, ma chin é stato sovrascritto da P2, quindi il programma non funziona correttamente.

```
Process P1          Process P2
.
chin = getchar();
.
chout = chin;
putchar(chout);
.
.
.
Process P2
.
chin = getchar();
.
chout = chin;
putchar(chout);
```

Figure 98: Esempio di echo con 2 processi su un processore

il problema é dovuto alla concorrenza, perché ci sono piú processi che accedono a memoria condivisa.

```
Process P1          Process P2
.
chin = getchar();
.
chout = chin;
putchar(chout);
.
.
.
Process P2
.
chin = getchar();
.
chout = chin;
putchar(chout);
```

Figure 99: Esempio di echo con 2 processi su due processori

il problema si presenta anche con piú processori, perché chin e chout sono variabili globali, quindi sono in memoria condivisa, quindi anche in questo caso il programma non funziona correttamente.

Per risolvere il problema si puó pensare che la funzione echo puó essere chiamata da un processo alla volta, o meglio la funzione echo puó essere chiamata da tanti processi, ma puó essere eseguita da un solo processo alla volta, facendo diventare echo atomica, però c'é bisogno che qualcuno indichi che questa funzione é atomica.

### 7.3 Race Condition

Una corsa critica si ha quando piú processi leggono e scrivono una variabile condivisa, e lo fanno in modo tale che il risultato finale dipende dall'ordine in cui vengono eseguiti i processi, in particolare il risultato puó dipendere dal processo che finisce per ultimo, un sezione critica é la parte di codice di un processo che puó portare ad una corsa critica, nell'esempio precedente la sezione critica é l'intera funzione.

### 7.4 Per ciò che Riguarda il SO

Il SO deve:

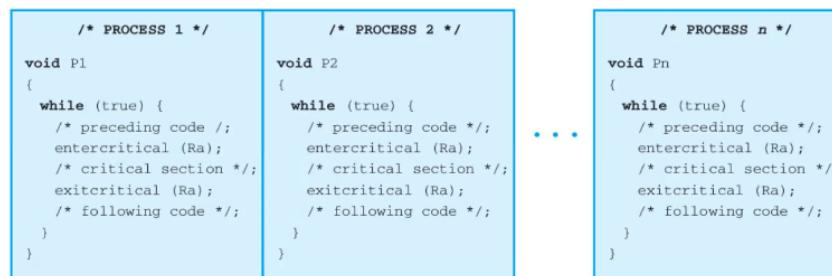
- Tenere traccia di vari processi

- allocare e deallofare risorse ( CPU, memoria, I/O,file)
- Proteggere dati e risorse dall'interferenza (non autorizzata) di altri processi
- redassicurare che processi ed output siano indipendenti dalla velocità di computazione (ovvero lo scheduling), indipendenti da intendere cum grano salis, rispetto alle specifiche di ciascun processo.

Comunicazione	Relazione	Influenza	Problemi di Controllo
Nessuna (ogni processo pensa di essere solo)	Competizione	Risultato di un processo indipendente dagli altri; Tempo di esecuzione di un processo dipendente dagli altri	Mutua esclusione; deadlock; starvation
Memoria condivisa (i processi sanno che c'è qualche altro processo)	Cooperazione	Risultato di un processo dipendente dall'informazione data da altri; Tempo di esecuzione di un processo dipendente dagli altri	Mutua esclusione; deadlock; starvation; coerenza dei dati
Primitive di comunicazione (i processi sanno anche i PID di alcuni altri processi)	Cooperazione	Risultato di un processo dipendente dall'informazione data da altri; Tempo di esecuzione di un processo dipendente dagli altri	Deadlock; starvation

## 7.5 I processi e la competizione per le risorse

Il problema è che quando i processi devono accedere ad una risorsa devono chiedere al sistema operativo, quindi la necessità principale è quella della mutua esclusione, ovvero solo un processo alla volta può accedere alla risorsa, però bisogna stare attenti che comunque non ci siano deadlock e starvation nella systemcall,



Ho una systemcall (Voglio usare il monitor), sostanzialmente la systemcall dovrebbe:

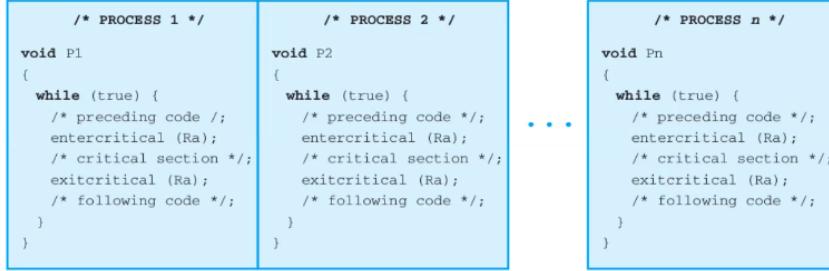
1. Entra nella sezione critica (Monitor)
2. Accede alla risorsa
3. Esce dalla sezione critica (Fa sì che altri processi possano accedere alla risorsa)

questo non è sempre possibile perché potrebbe essere necessario fare una richiesta di blocaggio, quindi ci sono tanti processi che vogliono accedere alla stessa risorsa, quindi la syscall fa una parte iniziale di controllo per vedere se posso concedere la risorsa, se ci sono più processi la syscall deve decidere chi accede alla risorsa, tutti i processi sono visti come dei cicli in cui continuamente fanno richieste a determinate risorse.

## 7.6 Mutua Esclusione

### Mutua Esclusione Processi Cooperanti

Nei processi cooperanti é il programmatore che si deve occupare della mutua esclusione, il programmatore deve fare in modo che i processi non accedano contemporaneamente alla stessa risorsa, tutto questo viene fatto tramite strumenti messi a disposizione dal sistema operativo.



### Starvation

Supponiamo che ci siano 2 processi A e B tutti e due richiedono accesso alla stampante, quello che puó succedere che A rilascia la stampante, ma il sistema operativo lascia A in esecuzione quindi A fa in tempo a fare un'altra richiesta alla stampante, e quindi il sistema operativo puó decidere di dare la stampante ad A, in questo modo il processo B puó andare in **Starvation**

### Deadlock

Diciamo che A richiede prima la stampante e poi il monitor, mentre B richiede prima monitor e poi stampante,( **il fatto che siamo indipendenti dallo scheduler puó essere visto come un diavolo che cerca in tutti i modi di bucare il funzionamento del sistema operativo o delle applicazioni**, quindi dobbiamo fare in modo di metterci nel caso peggiore di scheduling per validare un'applicazione), puó capitare che lo scheduler quindi faccia andare B in mezzo alle 2 richieste di A, quel tanto che basta per fargli richiedere il monitor, facendo in modo che B prende il monitor e prima che B possa prendere la stampante, faccio tornare A, e quando A chiede il monitor, il monitor é già preso da B, e quindi A va in attesa, e quindi B non puó prendere la stampante perché A ha la stampante, e quindi A e B restano bloccati per sempre, eppure tutto é avvenuto legalmente: la mutua esclusione é stata rispettata, nessuno ha violato la mutua esclusione, però abbiamo un deadlock.

### Requisiti per la mutua esclusione

- Solo un processo alla volta puó essere nella sezione critica per una risorsa
- Niente Deadlock e Starvation
- Nessuna assunzione su scheduling dei processi, né sul numero dei processi
- Un processo deve entrare subito nella sezione critica, se nessun altro processo usa la risorsa (se un solo processo che vuole la risorsa, deve entrare subito)
- Un processo che si trova nella sua sezione non-critica non deve bloccare altri processi che richiedono la risorsa (Non puó essere bloccato)
- Un processo che si trova nella sezione critica ne deve prima o poi uscire, piú in general ci vuole cooperazione, Es. non bisogna scrivere un processo che entra nella sua sezione critica senza chiamare entercritical().

### Mutua Esclusione for Dummies

```

int bolt = 0
void P(int i)
{
    while (true)
        bolt = 1;
    while (bolt == 1) /*do nothing*/;
        /* critical section */;

    bolt = 0;
    /* remainder section */;
}
parbegin(P(0),P(1),\ldots,P(n-1));

```

metto bolt a 1 per prendermi la critical section e aspetto che bolt diventi 0 per uscire dalla critical section, questa é una cosa pessima perché metto bolt a 1 e non rilascio mai la risorsa per il loop infinito, basta inoltre che lo scheduler faccia eseguire i 2 processi in interleaving perfetto, per avere un deadlock, quindi va bene la safety, ma ci vuole anche la liveness (almeno un processo ci deve entrare).

```

int bolt = 0
void P(int i)
{
    while (true)
        while (bolt == 1) /*do nothing*/;
        bolt = 1;
        /* critical section */;
        bolt = 0;
        /* remainder section */;
}
parbegin(P(0),P(1),\ldots,P(n-1));

```

quindi facciamo che quando entro nel secondo while, metto bolt a 1, se il dispatcher manda in esecuzione un'altro processo l'altro processo sarà in attesa a while (bolt == 1), fino a che il primo processo non mette bolt a 0, quindi il secondo processo può entrare nella sezione critica, il problema é che non basta che ci sia qualche scheduling, ma deve essere garantito che tutti gli scheduling possibili garantiscano l'unicità di un processo nella sezione critica.

### Scheduler e Livello Macchina

Il dispatcher può interrompere un processo in qualsiasi istante, vuol dire che potrebbe farlo anche prima che un'istruzione sia completata, per esempio nel secondo esempio se prendiamo in considerazione il secondo while del secondo esempio, a livello macchina dobbiamo fare un confronto tra bolt e 1, e sulla base del risultato del confronto, fare un salto oppure andrà all'istruzione successiva, supponiamo che il primo processo P(0) venga eseguito fino a bolt = 1 compreso, si potrebbe pensare che la mutua esclusione sia garantita, ma non é così, supponiamo che prima sia andato in esecuzione P(1), che é riuscito ad arrivare a metà del test di bolt ==1, ovvero carico bolt in un registro e lo confronto con 1, il problema é che il dispatcher potrebbe aver lasciato caricare il valore di bolt in un registro, poi prima di fare il test manda in esecuzione P(0) che arrivato a bolt = 1, se P(1) riprende il controllo, il fatto che bolt sia stato messo a 1 da P(0) non lo sa, perché bolt é in memoria, e quindi P(1) potrebbe entrare nella sezione critica, quindi non é garantita la mutua esclusione.

### Mutua Esclusione con Interruzioni Disabilitate

```
while (true)
{
/* prima sezione critica */
disable_interrupts();
/* sezione critica */
riabilita_interrupts();
/* rimanente */
}
```

Una prima soluzione per garantire la mutua esclusione é quella di disabilitare le interruzioni, l'idea quindi é che immediatamente prima di entrare nella sezione critica disabilito le interruzioni, in questo modo il dispatcher non puó interrompere il processo, dopo che ho finito la sezione critica riabilito le interruzioni.

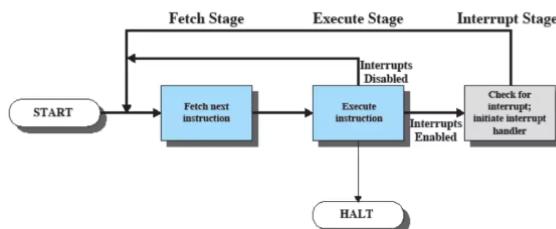


Figure 1.7 Instruction Cycle with Interrupts

Ci sono però dei problemi, il primo problema é che se do questa possibilità ai processi utenti é possibile che essi ne abusino e di conseguenza cala la multiprogrammazione, inoltre disabilitare le interruzioni funziona solo se abbiamo un solo processore, se abbiamo piú processori non funziona, attualmente la disabilitazione delle interruzioni si fa solo in kernel mode.

### Istruzioni speciali macchina

Esistono delle soluzioni che sono piú accettabili :

- **Compare\_and\_swap:**
- **exchange**

L'exchange prende due indirizzi in memoria e scambia i valori, il punto é che sono 3 micro istruzioni, e potrebbe succedere che il processo venga interrotto in qualsiasi punto, ma l'hardware garantisce che se un processo esegue questa istruzione, nessun altro processo puó eseguire questa istruzione, quindi é garantita la mutua esclusione, e quindi é una istruzione atomica, quindi é garantita la mutua esclusione.

```
void exchange(int register , int memory)
{
    int temp
    temp = memory;
    memory = register;
    register = temp;
}
```

Il compare\_and\_swap é una istruzione che prende 3 parametri, una locazione di memoria, 2 valori, se il valore in memoria é uguale al secondo argomento, allora cambio il valore in memoria con il terzo argomento, in ogni caso la funzione ritorna il valore vecchio.

```

int compare_and_swap(int word, int testval, int newval)
{
    int oldval;
    oldval = word;
    if (word == testval)
        word = newval;
    return oldval;
}

```

Subito prima della sezione critica faccio un compare\_and\_swap, che confronta il valore di ritorno della funzione con 1, al primo processo che arriva cambia il suo valore in 1 e ritorna 0 perché è il vecchio valore, supponiamo che a questo punto mando in esecuzione un altro processo, esso si troverà con bolt = 1, e quindi non entrerà nella sezione critica, quando il primo processo esce dalla sezione critica, mette bolt a 0, e quindi il secondo processo può entrare nella sezione critica.

```

/* program mutual exclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true) {
        while (compare_and_swap(bolt, 0, 1) == 1)
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ... ,P(n));
}

```

l'idea è che uso il fatto che bolt sia 1 per i processi successivi al primo per farli stare nel while, quindi non entrano nella sezione critica.

l'exchange si limita a scambiare i valori, per cui l'idea è che ho una variabile locale **keyi** (ogni processo ha il suo) quindi c'è uno scambio di valori tra **keyi** e **bolt**, se **keyi** è diverso da 0, allora il processo può entrare nella sezione critica, altrimenti deve aspettare.

```

/* program mutual exclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
    int keyi = 1;
    while (true) {
        do exchange (keyi, bolt)
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}

```

Figure 100: Sbagliato

l'errore dietro a questo codice è che il primo che entra va avanti, e rimette bolt a 0, supponiamo che non ci siano altri processi, quando ritorna per un'altra richiesta lui si blocca perché scambia 0 con 0.

I vantaggi di usare queste istruzioni sono:

- Applicabili a qualsiasi numero di processi, sia su un sistema ad un processore che su un sistema a più processori
- Semplici e quindi facili da verificare
- Possono essere usate per gestire sezioni critiche multiple

Gli svantaggi sono:

- Sono basati sul concetto di busy-waiting, il punto è che queste soluzioni devono continuamente ciclare fino a che non possono entrare nella sezione critica, dal punto di vista del sistema operativo lui non ha modo di distinguere un processo che sta facendo una istruzione speciale da uno che sta facendo un altro tipo di operazioni, quindi dal punto di vista del dispatcher questi processi sono ready e non possono essere esclusi, questa cosa è inevitabile, se siamo in un sistema operativo con dispatcher round robin viene eseguito fino a time-out
- Possibile Starvation: supponiamo 2 processi P(1) passa, va in esecuzione P(2) dopo di che il dispatcher dà la CPU a P(1) che va in esecuzione, e potrebbe riuscire a rientrare di nuovo nella sezione critica, e quindi P(2) va in starvation
- Possibile Deadlock: se abbiamo una priorità fissa tra i processi, es. P(1) con priorità bassa e P(2) con priorità alta, P(1) esce dalla sezione critica e P(2) entra, se P(1) rientra nella sezione critica, P(2) entra in sezione critica e P(1) non riesce più ad entrare.

## 7.7 Semafori

In molti caso per evitare il busy waiting si possono usare i semafori, che sono delle particolari strutture dati sulle quali è possibile fare 3 operazioni **Atomiche** garantite dal sistema operativo:

- Initialize
- decrement o semWait : può mettere il processo in blocked per cui niente CPU sprecata come con il busy waiting
- increment o semSignal : può mettere un processo blocked in ready

I semafori racchiudono al loro interno un contatore, che può essere decrementato o incrementato, la cosa interessante è che l'operazione di decremento che viene chiamata wait può mettere in blocked il processo che la chiama, questo evita quindi il busy waiting, per contro l'operazione di incremento può mettere in ready un processo che era blocked, stiamo quindi considerando un'altro tipo di blocked rispetto a quello che abbiamo visto con l'I/O, in questo caso il processo è blocked perché è in attesa di un semaforo, ovviamente essendo syscall, sono istruzioni che vengono eseguite in kernel mode e possono agire direttamente sui processi.

```

struct semaphore {
    int count;
    queueType queue;
};

void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignal(semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}

```

un semaforo quindi è composto da un intero e da una coda di processi, se facciamo una wait significa che decremento il contatore, se come risultato il contatore è negativo allora il processo che ha fatto questa chiamata va bloccato e messo in coda, invece una signal incrementa di 1 il contatore, se il valore è ancora negativo, significa che c'è qualcosa da sbloccare, per cui dalla coda vado a scegliere uno dei processi, lo tolgo e lo metto in ready.

```

struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};
void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}

```

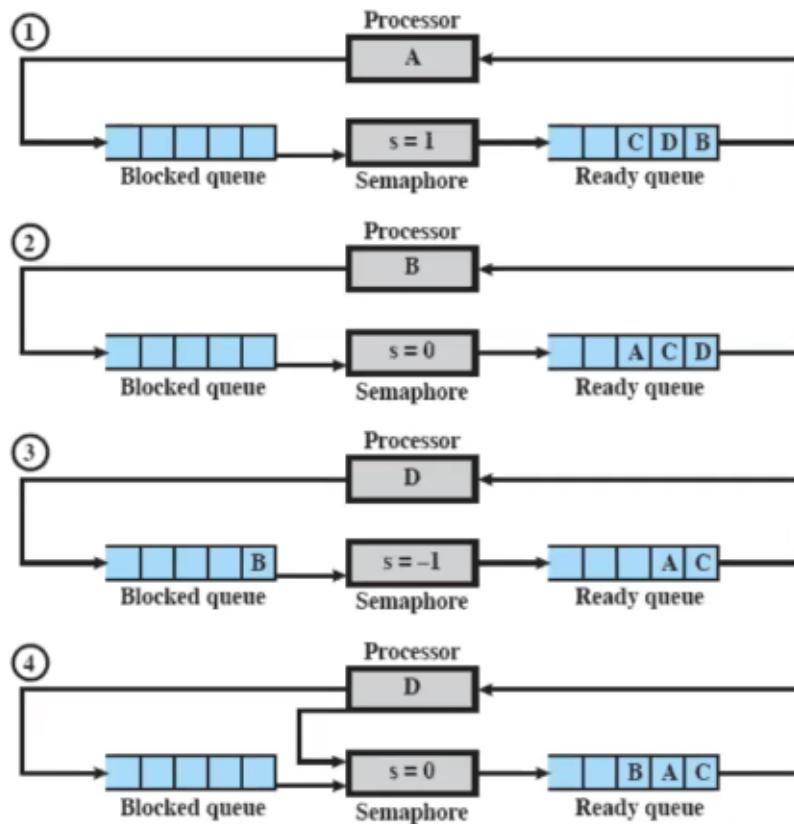
In un semaforo binario il valore non é piú un intero, ma un booleano, quindi il valore puó essere 0 o 1, l'idea però rimane la stessa, se il valore é 0 il processo va in blocked, se la coda é vuota allora metto il valore a 1, invece se ci sono processi in coda, prendo uno di questi processi e lo metto in ready.

<pre> semWait(s) {     <b>while</b> (compare_and_swap(s.flag, 0 , 1) == 1)         /* do nothing */;     s.count--;     <b>if</b> (s.count &lt; 0) {         /* place this process in s.queue*/;         /* block this process (must also set s.flag to 0) */;     }     s.flag = 0; }  semSignal(s) {     <b>while</b> (compare_and_swap(s.flag, 0 , 1) == 1)         /* do nothing */;     s.count++;     <b>if</b> (s.count &lt;= 0) {         /* remove a process P from s.queue */;         /* place process P on ready list */;     }     s.flag = 0; } </pre>	<pre> semWait(s) {     inhibit interrupts;     s.count--;     <b>if</b> (s.count &lt; 0) {         /* place this process in s.queue */;         /* block this process and allow inter- rupts */;     }     <b>else</b>         allow interrupts; }  semSignal(s) {     inhibit interrupts;     s.count++;     <b>if</b> (s.count &lt;= 0) {         /* remove a process P from s.queue */;         /* place process P on ready list */;     }     allow interrupts; } </pre>
--	--

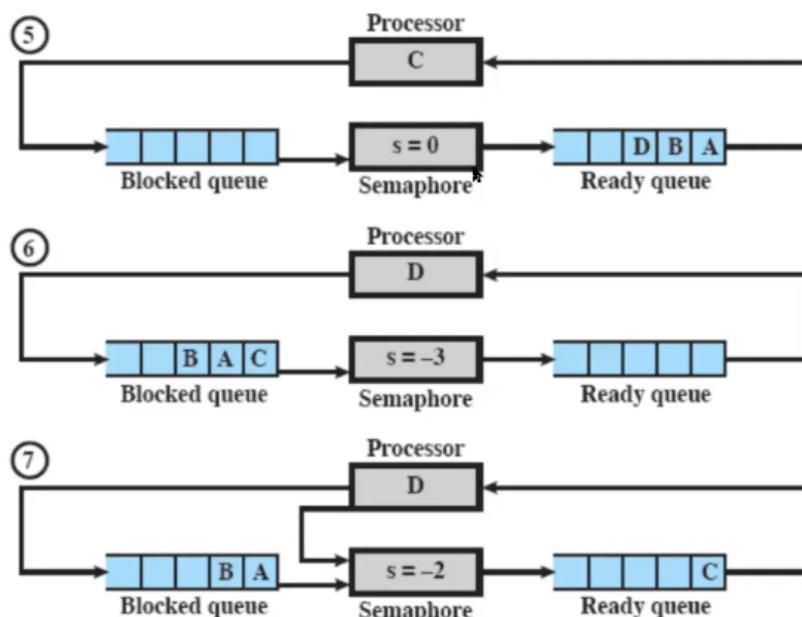
In questa maniera il busy waiting viene ridotto al minimo, se sono su un solo processore, disabilito le interruzioni.

### 7.7.1 semafori Deboli e Forti

Si parla di semafori deboli e di semafori forti, a seconda di come devo scegliere il processo da sbloccare, i semafori forti sono quelli che usano una coda FIFO, se la politica non viene specificata, allora si parla di weak semaphore, sappiamo quindi che un generico processo viene sbloccato, con i semafori forti, é possibile inoltre evitare la starvation.



1. Faccio una wait, il contatore va a 0 , ma il processo no va blocked perché il contatore è maggiore di 0
2. Faccio un'altra wait, il contatore va a -1, il processo va in blocked e va in esecuzione
3. D fa signal, il contatore va a 0, il processo B ritorna nella coda dei ready



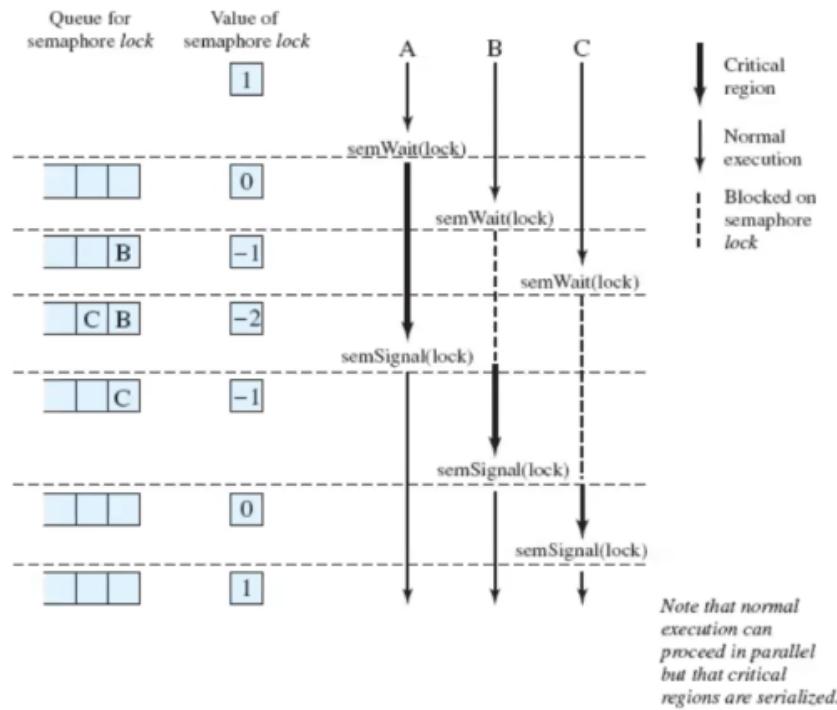
In questa seconda parte faccio 3 wait e mando in blocked i tre processi, poi faccio 3 signal, ed in ordine FIFO sblocco i processi in blocked.

```

/* program mutual exclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true) {
        semWait(s);
        /* critical section */
        semSignal(s);
        /* remainder */
    }
}
void main()
{
    parbegin (P(1), P(2), . . . , P(n));
}

```

A questo punto non c'è starvation, se siamo tra due processi non c'è in nessun caso, con più di due processi ci potrebbe essere solo con i semafori deboli, perché potrebbe succedere di sbloccare solo alcuni processi.



## 7.8 Produttore-Consumatore

Ci sono casi in cui siamo interessati ad avere processi che cooperano per risolvere un problema, un esempio è quello del produttore-consumatore. La situazione è la seguente:

- Un processo produttore che crea dati e li mette in un buffer
- Un consumatore che legge i dati dal buffer creati dal produttore
- al buffer può accedere un solo processo alla volta

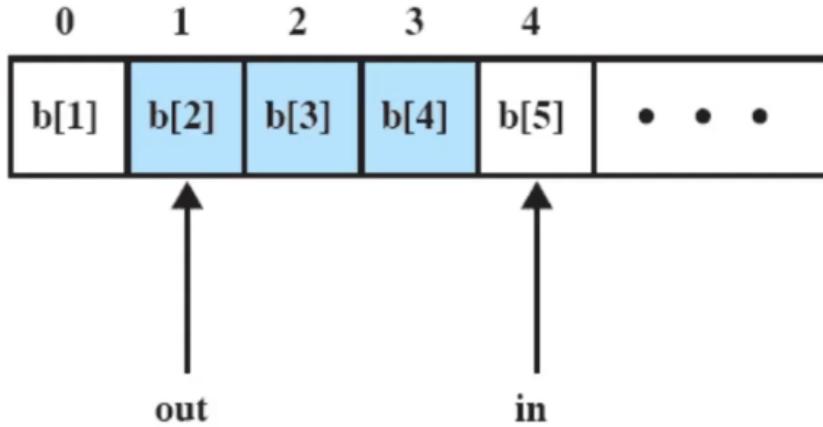
Il problema è quello di garantire la mutua esclusione nell'accesso, ma il vero problema che voglio risolvere è assicurare che i produttori non inseriscano i dati se il buffer è pieno, ed il consumatore non deve prendere dati se il buffer è vuoto, il progettista deve fare in modo che il produttore

aspetti se il buffer é pieno, e il consumatore aspetti se il buffer é vuoto, in realtà ci sono molte varianti di questo problema, essenzialmente potremmo vedere il buffer come una coda, e quindi il produttore mette i dati in coda e il consumatore li toglie dalla coda, ma noi considereremo il buffer come una entità ad accedere in mutua esclusione, supponiamo inizialmente che il buffer sia infinito allora:

```
while (true){
    /* produce an item v*/
    b [in] = v;
    in++;
}

while (true){
    while (in <= out) /* do nothing in attesa che out sia pi piccolo di i
    w = b [out];
    out++;
    /* consume the item w */
}
```

Da notare che la fase di produzione e di consumo sono da considerare operazioni locali quindi non c'è nessun problema se il produttore e il consumatore fanno queste operazioni in parallelo, v e w sono variabili locali, e almeno in deve essere globale.



### Soluzione al Problema

Una prima implementazione che potremmo pensare é quella riportata nell'immagine sopra che usa i semafori binari, nel nostro caso  $s=1$  e  $delay = 0$ , abbiamo un main con 2 processi, il produttore e il consumatore, essenzialmente c'è un controllo che ci sia la mutua esclusione sul buffer, questa cosa é controllata dal semaforo  $s$ , non deve succedere quindi che il consumatore cerchi di consumare con la coda vuota, oltre ai 2 semafori abbiamo una variabile condivisa  $n$  intera, durante la produzione incremento  $n$ , durante il consumo decremento  $n$ , quindi  $n$  rappresenta il numero di elementi nel buffer, l'idea é che il consumatore che va in esecuzione per primo si blocca perché  $delay$  é 0, quindi é necessario che il produttore vada in esecuzione per primo, a questo punto il produttore mette a 1 il semaforo  $delay$ , sbloccando quindi la possibilità che il consumatore possa andare in esecuzione, l'idea é **ho messo un nuovo elemento dopo che il buffere é vuoto allora sblocco il consumatore**, se dopo che ho consumato  $n$  diventa 0 allora mi devo bloccare, sembra sia una soluzione corretta, ma non lo é, infatti potrebbe succedere che a metá tra il consumo ed il controllo che  $n$  sia = 0, il produttore mette un nuovo elemento, e quindi il consumatore non si blocca, essenzialmente per risolvere il problema

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

Figure 101: Sbagliata

dopo il decremento di n mi salvo il suo valore in una variabile locale, e quindi se viene richiamato il produttore che modifica n, io mi ricordo il vecchio valore di n.

```

/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    int m; /* a local variable */
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}

```

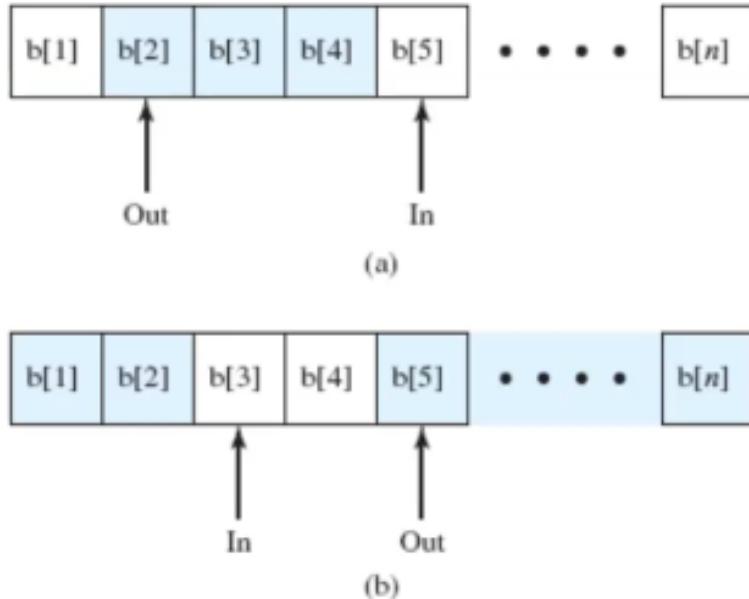
```

/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}

```

Qualsiasi cosa scritta con semafori binari si può scrivere con semafori generali, vale anche il viceversa.

### 7.9 Produttore consumatore con buffer circolare



Mettiamoci quindi nel caso in cui il buffer sia finito, quindi c'è una regione finita di memoria, che è dedicata al buffer, ed è gestita in maniera circolare, perché non si possono fare assunzioni su quanto sia veloce il produttore e il consumatore, potrebbe succedere che il dispatcher dia tempo al produttore di produrre 10 elementi, e il consumatore ne consumi 5, quindi questo si realizza con un buffer circolare, nell'immagine ci sono n slot nel buffer dove si possono mettere gli oggetti da consumare, l'idea è che esistono 2 puntatori in e out, in è il puntatore che indica dove mettere il prossimo elemento, out è il puntatore che indica dove prendere il prossimo elemento, c'è un problema, che è dovuto al fatto che se in e out sono uguali, il buffer può essere sia pieno che vuoto, se lo pensiamo in maniera circolare, se per esempio: se  $n=5$ , ed il consumatore non fa niente, ed il produttore mette 5 elementi, nel buffer se partivamo da uno stato vuoto,  $in = 0$  e  $out = 0$ , dopo che il produttore ha messo 5 elementi,  $in = 0$  e  $out = 0$ , quindi in e out sono uguali, ma il buffer è pieno, per questo motivo dobbiamo sacrificare uno slot, quindi il buffer avrà  $n-1$  slot, a questo punto  $in=out$  significa che il buffer è vuoto, invece è pieno se  $in + 1 \bmod n = out$ , con questo scenario le cose rimangono simili a prima, addesso c'è un controllo sul pieno e sul vuoto.

```

/*produce an item v */
while (true){
    while ((in + 1) % n == out) /* do nothing */;
    b[in] = v;
    in = (in + 1) % n;
}

while (true){
    while (in == out) /* do nothing */;
    v = b[out];
    out = (out + 1) % n;
    /* consume the item w */
}

```

come facciamo quindi a scrivere una funzione con i semafori per risolvere il problema del pro-

duttore consumatore con buffer circolare? La soluzione di prima chiaramente non funzionerebbe perché non c'è un controllo sul pieno e sul vuoto, per risolverlo quindi abbiamo bisogno di un terzo semaforo.

```

/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}

```

quindi partiamo con un terzo semaforo inizializzato alla lunghezza del buffer, il trucco è

1. Nel consumatore faccio subito una semWait su n così se sono il primo mi blocco
2. Nel produttore faccio subito una semWait su e, così se sono da solo e faccio subito size-OfBuffer iterazioni, mi blocco.
3. ad ogni Wait deve corrispondere una signal, in modo che il produttore possa svegliare il consumatore e viceversa.

questa è la soluzione classica di **Un produttore e un consumatore**.

### 7.10 Mutua Esclusione: Soluzioni software

Fino adesso abbiamo fatto riferimento a soluzioni atomiche, diamo uno sguardo a soluzioni, per risolvere problemi di mutua esclusione, che non sono atomiche, ovvero che non sono garantite dal sistema operativo.

Esempio ci sono due processi che vogliono fare la critical section uno per volta, cerchiamo di capire come possiamo garantire questa cosa, facendo solo assegnamento a variabili, facciamo un primo tentativo (questa cosa funziona solo per 2 processi),

```

/* PROCESS 0 */           /* PROCESS 1 */

.
.

while (turn != 0)         while (turn != 1)
    /* do nothing */;      /* do nothing */;
/* critical section*/;    /* critical section*/;
turn = 1;                  turn = 0;
.
.
```

Una prima soluzione potrebbe essere che ho una variabile condivisa tra i due processi, se turn è 0 P(0) si blocca, se turn è 1 P(1) si blocca, chiaramente adesso è tutta attesa attiva, è chiaro che una soluzione come questa risolve il problema della mutua esclusione, se un processo è da solo e vuole accedere alla sezione critica non lo può fare.

```

/* PROCESS 0 */           /* PROCESS 1 */

.
.

while (flag[1])         while (flag[0])
    /* do nothing */;      /* do nothing */;
flag[0] = true;           flag[1] = true;
/*critical section*/;     /* critical section*/;
flag[0] = false;          flag[1] = false;
.
.
```

Invece che usare turn, uso un vettore (2 variabili globali), il processo 0 legge la variabile dell'altro e modifica la propria, il processo 1 fa la stessa cosa, ovviamente questo non può andare bene, se entrambi sono a 0 potrebbe succedere che entrambi entrano nella sezione critica, quindi non è garantita la mutua esclusione.

```

/* PROCESS 0 */           /* PROCESS 1 */

.
.

flag[0] = true;           flag[1] = true;
while (flag[1])           while (flag[0])
    /* do nothing */;      /* do nothing */;
/* critical section*/;     /* critical section*/;
flag[0] = false;           flag[1] = false;
.
.
```

Metto il flag a 1, ad entrambi i processi, e poi faccio un while, il flag indica l'intento di iniziare la sezione critica, però se faccio di nuovo un interleave perfetto, potrebbe succedere che entrambi i processi entrino in deadlock

```

/* PROCESS 0 */           /* PROCESS 1 */

.

flag[0] = true;          flag[1] = true;
while (flag[1]) {         while (flag[0]) {
    flag[0] = false;      flag[1] = false;
    /*delay */;           /*delay */;
    flag[0] = true;       flag[1] = true;
}
/*critical section*/;     }
/* critical section*/;
flag[0] = false;          flag[1] = false;
.

```

Invece di non fare niente provo, a dichiararmi inizialmente come non interessato a entrare nella sezione critica, e poi metto il flag a 0, e dopo un delay metto il flag a 1, mentre sono in attesa, l'idea è che se introduco un ritardo random, allora con una buona probabilità succede che il processo uno fa in tempo a mettere il suo flag a true, ma il processo 0 non riesce a mettere a true il suo flag, quindi il processo 1 entra nella sezione critica, potrebbe effettivamente funzionare in un discorso con alta probabilità, però non è garantito, quindi non è una soluzione corretta.

### Algoritmo Di Dekker

<pre> p0: wants_to_enter[0] = true while wants_to_enter[1] {     if turn ≠ 0 {         wants_to_enter[0] = false         while turn ≠ 0 {             // busy wait         }         wants_to_enter[0] = true     } }  // critical section ... turn = 1 wants_to_enter[0] = false // remainder section </pre>	<pre> p1: wants_to_enter[1] = true while wants_to_enter[0] {     if turn ≠ 1 {         wants_to_enter[1] = false         while turn ≠ 1 {             // busy wait         }         wants_to_enter[1] = true     } }  // critical section ... turn = 0 wants_to_enter[1] = false // remainder section </pre>
---	---

Solo una variabile condivisa non basta, solo 2 variabili booleane non bastano, proviamo quindi a mettere insieme le due idee, succede che fin dall'inizio dichiaro di voler provare ad entrare nella sezione critica, l'idea è che c'è una variabile turn condivisa, se per caso non tocca a me rimetto a falso il fatto che voglio entrare e faccio una attesa attiva, fino a che turn continua ad essere l'altro, quando esco rimetto wants to enter a true, si può dimostrare matematicamente che questa soluzione funziona.

Questa soluzione vale solo per 2 processi, se ci sono più processi, non è così semplice, garantisce la non starvation, garantisce che non ci siano deadlock e non richiede nessun supporto dal sistema operativo e dall'hardware, ci sono alcune architetture che potrebbero riordinare le istruzioni, chiaramente se ciò succede l'algoritmo non funziona più.

### Algoritmo Di Peterson

```

boolean flag [2];
int turn;
void P0()                                Peterson's Algorithm
{
    while (true) {
        flag [0] = true;
        turn = 1;
        while (flag [1] && turn == 1) /* do nothing */;
        /* critical section */;
        flag [0] = false;
        /* remainder */;
    }
}
void P1()
{
    while (true) {
        flag [1] = true;
        turn = 0;
        while (flag [0] && turn == 0) /* do nothing */;
        /* critical section */;
        flag [1] = false;
        /* remainder */;
    }
}
void main()
{
    flag [0] = false;
    flag [1] = false;
    parbegin (P0, P1);
}

```

Anche qui il processo che vuole entrare segnala che vuole entrare, per quanto riguarda la variabile turn, viene acceduto in scrittura prima della critical section, il processo 0 dice di passare al processo 1 e viceversa, in questa maniera anche se facessimo un interleave perfetto, o P0 setta turn 1 o P1 setta turn 0, l'ultimo che fa l'assegnamento vince, alla fine l'unica cosa che serve dopo la critical section bisogna mettere a 0 il proprio flag.

Questa soluzione vale solo per 2 processi, anche se é piú semplice estenderlo ad N processi, é possibile la bounded waiting.

### 7.11 Comunicazione Esplicita

É possibile che un processo mandi un messaggio ad un altro processo, per fare message passing funziona sia con memoria condivisa che distribuita, puó essere anche usata per la sincronizzazione, le funzionalita di message passing sono:

- send (destinatario, messaggio da inviare)
- receive (mittente, messaggio (zona di memoria dove voglio mettere i valori))
- spesso c'é anche un test di ricezione
- notare che message é un input per la send ed un output per la receive

#### 7.11.1 Sincronizzazione

La comunicazione richiede anche una sincronizzazione, tra i processi, allora prima il mittente deve inviare e poi il ricevente deve ricevere, queste operazioni possono essere bloccanti ed operazioni che non bloccano, il test di ricezione non é mai bloccante.

### Send e receive bloccanti

Il processo che fa la send si blocca fino a che il processo che fa la receive non riceve il messaggio, quando il processo riceve il messaggio allora si sblocca il processo che ha fatto la send, tipicamente questo tipo di comunicazione si chiama rendez-vous.

### Send non bloccante

Con ricezione bloccante, il mittente continua (non si blocca) il destinatario rimane bloccato finché non ha ricevuto il messaggio, Con ricezione non bloccante, se il messaggio c'è viene ricevuto, altrimenti il destinatario continua, la receive non bloccante può settare un bit dentro il messaggio per dire se la ricezione è avvenuta oppure no, se la ricezione non è bloccante anche il mittente non è bloccante, sono operazioni atomiche, un solo processo per volta le esegue ed finché l'operazione non è completata, oppure il processo non viene bloccato.

## 7.12 Indirizzamento

Il mittente deve poter dire a quale processo mandare il messaggio(o quali processi), lo stesso vale per il destinatario, ci sono 2 modi per fare l'indirizzamento:

- Diretto: il mittente sa l'identità del destinatario
- Indiretto: il mittente sa l'identità di un intermediario che sa l'identità del destinatario.

### Indirizzamento Diretto

Nell'indirizzamento diretto la send ha come parametro il destinatario, per la receive invece può essere presente oppure no es:

```
send( destinatario , messaggio );  
receive( mittente , messaggio );
```

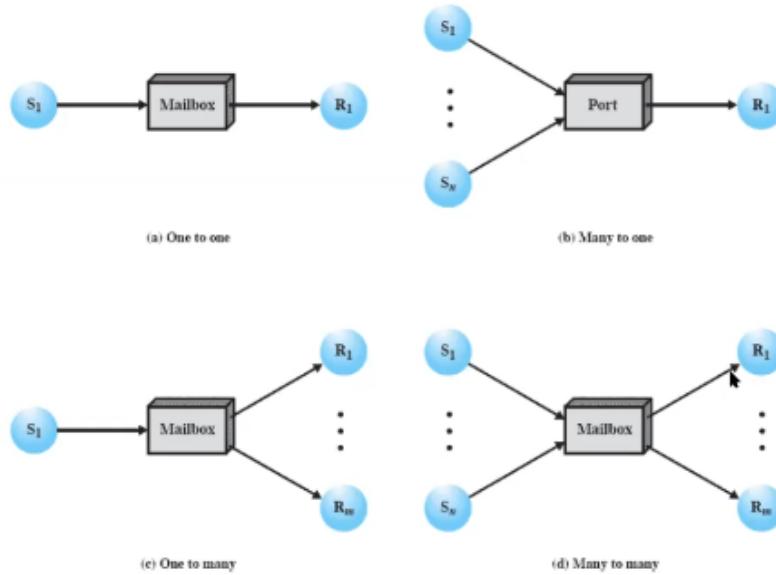
oppure

```
send( destinatario , messaggio );  
receive( null , messaggio );
```

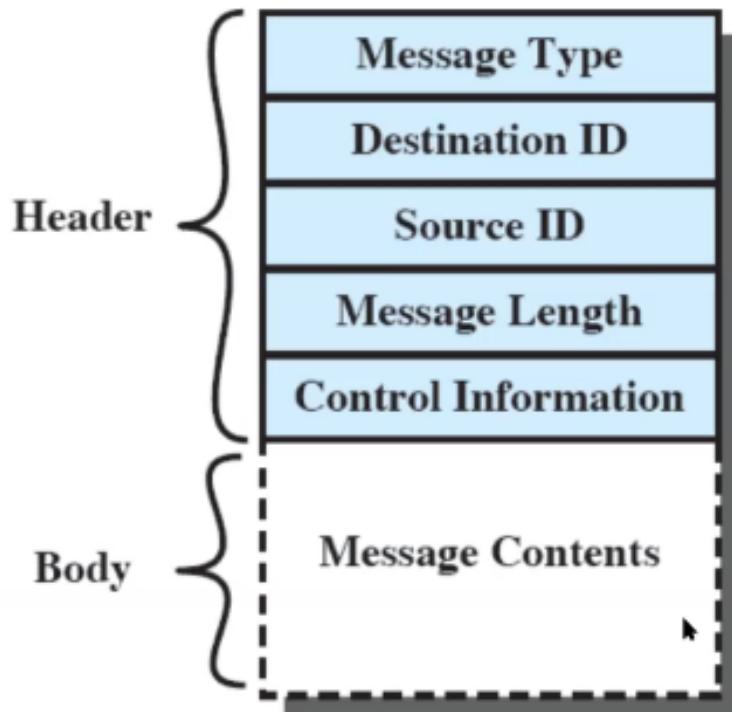
questo succede perché dentro al messaggio è presente anche il mittente, inoltre ogni processo ha una sua coda per i messaggi; una volta piena, solitamente il messaggio si perde o viene ritrasmesso.

### Indirizzamento Indiretto

Nell'indirizzamento indiretto i messaggi vengono mandati ad una mailbox (Zona di memoria condivisa) ed va esplicitamente creata, il mittente manda messaggi alla mailbox, da dove il destinatario li prende, redse la ricezione è bloccante e ci sono più processi in attesa su una ricezione, un solo processo viene sbloccato Ci sono quindi evidenti analogie con il problema del produttore/consumatore, in particolare, se la mailbox è piena allora anche nbsend si deve bloccare, se serve solo per le versioni bloccanti, e per comunicazioni x-to-one, la mailbox può avere dimensione 1.



### 7.13 Struttura dei messaggi



Di solito quando intendiamo un messaggio quello che intendiamo é il contenuto del messaggio(BODY), ma in realtá un messaggio é composto da:

- **Header:** contiene informazioni sul messaggio, come il mittente, il destinatario, la lunghezza del messaggio, il tipo di messaggio, control information.
- **Body:** contiene il contenuto del messaggio

le informazioni che si trovano nell'header sono chiamate meta-dati.

### 7.14 Usare i messaggi per risolvere la mutua esclusione

```

const message null = /* null message */;
mailbox box;
void P(int i) {
    message m;
    while (true){
        receive(box, m);
        /* sezione critica */
        nbsend(box, m);
        /* resto */
    }
}
void main() {
    box = create_mailbox();
    nb_send(box, null);
    parbegin(P(0), P(1));
}

```

Se io eseguo P senza inizializzazione non funziona, perché qualsiasi processo arriva si blocca sulla receive, questo funziona perché c'è la parte di inizializzazione nel main, prima di mandare gli n processi inizializzo la mailbox con un messaggio nullo, e mando in esecuzione i processi, se anche uno di questi processi arriva a ricevere il messaggio allora esegue la critical section, per poi rilasciare con un nbsend, se la send nel main la send è bloccante avremmo un problema perché i processi non andrebbero in esecuzione, invece all'interno di P potrei metterla bloccante, però non è efficiente

### 7.15 Producer-Consumer con Message Passing

```

const int capacity = /* dimensione del buffer */;
mailbox mayproduce, mayconsume;
const message null = /* null message */;
void main(){
    mayproduce = create_mailbox();
    mayconsume = create_mailbox();
    for (int i = 1; i <= capacity; i++)
        nb_send(mayproduce, null);
    parbegin(producer(), consumer());
}

void producer(){
    message m;
    while (true){
        receive(mayproduce, m);
        /* produce an item */
        nbsend(mayconsume, m);
    }
}

void consumer(){
    message m;
    while (true){

```

```
receive (mayconsume , m);
/* consume an item */
nbsend (mayproduce , m);
}
}
```

All'inizion mayconsume é vuota quindi il consumatore si blocca se va in esecuzione per primo, quindi il produttore puó produrre e riempire mayconsume. Le soluzioni con i messaggi possono tranquillamente risolvere la mutua esclusione, se usati bene evitano il deadlock, e se usati bene evitano la starvation (solo se le code di processi bloccati su una receive sono gestite in maniera "forte" ovvero i processi si sbloccano con FIFO).

### 7.16 Problema dei lettori e scrittori

Nel problema dei lettori e scrittori :

- Un area dati condivisa
- alcuni leggono ed altri scrivono

Quello che vogliamo ottenere che piú lettori possano leggere contemporaneamente, ma solo uno scrittore puó scrivere, se uno scrittore sta scrivendo allora nessun lettore puó leggere, la differenza con i produttori consumatori é che all'area di memoria si accede per intero.

**Soluzione con i semafori precedenza ai lettori**

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
void writer()
{
    while (true) {
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

In questa soluzione la precedenza é data ai lettori, c'è un intero condiviso, lo scrittore ha un semaforo di mutua esclusione, se per caso ci sono piú scrittori quelli rimangono in attesa sulla wait, per quanto riguarda i lettori, abbiamo il semaforo x per che incrementa i lettori, ho una condizione che dice **"Se sono primo blocco gli scrittori"**, dopo aver fatto la lettura decremento il numero di lettori, se sono l'ultimo lettore sblocca gli scrittori, il problema di questa soluzione é che gli scrittori potrebbero essere sottoposti a starvation.

### Soluzione con i semafori precedenza agli scrittori

```
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        semSignal (rsem);
        semSignal (z);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
```

```
void writer ()
{
    while (true) {
        semWait (y);
        writecount++;
        if (writecount == 1) semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount--;
        if (writecount == 0) semSignal (rsem);
        semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

In questa soluzione la precedenza è data agli scrittori, se c'è una sequenza di lettori che vogliono leggere, e un scrittore vuole scrivere, lascio finire i lettori che avevano già fatto richiesta, e poi faccio scrivere lo scrittore, è presente una variabile che indica il numero di scrittori, la prima if blocca i lettori, la seconda sblocca i lettori.

**Soluzione con i messaggi**

```
void reader(int i)
{
    while (true) {
        nbsend (readrequest, null);
        receive (controller_pid, null);
        READUNIT ();
        nbsend (finished, null);
    } }

void writer(int j)
{
    while (true) {
        nbsend (writerequest, null);
        receive (controller_pid, null);
        WRITEUNIT ();
        nbsend (finished, null);
    } }

void controller() {
    int count = MAX_READERS;
    while (true) {
        if (count > 0) {
            if (!empty (finished)) { /* da reader!
                receive (finished, msg);
                count++;
            }
            else if (!empty (writerequest)) {
                receive (writerequest, msg);
                writer_id = msg.sender;
                count = count - MAX_READERS;
            }
            else if (!empty (readrequest)) {
                receive (readrequest, msg);
                count--;
                nbsend (msg.sender, "OK");
            }
        }
    }
}
```

---

In questo tipo di soluzione, c'è un controller che gestisce le richieste.