Sistemi Operativi

Daniel Biasiotto

May 31, 2022

CONTENTS

- Teacher: Daniele Gunetti (daniele.gunetti@unito.it)
- PDF Version

1 GENERALITÁ

2 obiettivi di un OS:

- utente: rendere il sistema semplice
- macchina: rendere il sistema efficente e sicuro
- fornisce strumenti per uso corretto e semplice da usare
- Alloca risorse in maniera conveniente
- Controlla l'esecuzione dei processi per evitare pericoli

1.1 Confine OS - Software

1.1.1 GUI all'inizio non parte del OS

Con Windows viene integrata

1.1.2 Comandi Shell

La shell non e' parte OS in Unix

1.2 Kernel

E' il cuore del PC gestisce

- programmi in esecuzione

- memoria principale
- memoria secondaria

1.3 Gestione Eventi

Il Sistema Operativo e' `Event Driven`

1.3.1 L'OS generalmente non sta utilizzando le risorse

- le gestisce e le fa usare ai programmi
- puo' essere chiamato in causa dai programmi
- puo' controllare che tutto sia in ordine

1.3.2 Interruzioni

Evento –> OS gestisce l'evento prendendo il controllo della macchina

- una volta gestito e' restituito il controllo ad uno dei programmi che stavano girando prima dell'evento

1.3.3 Interrupt

natura hardware

- segnali che richiedono intervento OS

1.3.4 Eccezioni

natura software

- causate dal' programma in esecuzione divise in

TRAP

- Malfunzionamenti del programma in esecuzione
  - tentata divisione per 0
  - tentato accesso ad area protetta

SYSTEM CALL

- Richiesta di un servizio da parte del OS
  - richiesta di eseguire un operazione su un file

1.3.5 CPU segue dei passi predefiniti a livello hardware

- Salva lo stato della computazione
  - `PC` e altri valori della CPU in appositi registri
  - permette il riavvio del programma al punto di gestione evento
- in PC si scrive l'indirizzo in RAM della porzione di codice del OS che gestisce l'evento verificatosi
  - in `BootStrap`
    - OS carica in RAM
      - `vettore delle interruzioni`
      - `n` puntatori che indicano l'inizio del codice di gestione eventi
      - Codice gestione evento `n`
- return from event
  - ultima istruzione di ogni procedura di gestione
  - riprende l'esecuzione del programma precedente

1.4 Struttura della Memoria

1.4.1 Uniform Memory Access - UMA

Sistema multi-processore in cui questi si affacciano su un'unica memoria principale, con eguale tempo di accesso.

1.4.2 Principale - RAM

La memoria indirizzata alle istruzione eseguite

- qui risiedono dati e codice dei programmi

1.4.3 Secondaria - di Massa

Risiedono qui permanentemente dati e programmi

1.4.4 La gerarchia di memoria

- le memoria sono sempre piu' veloci ma di costo maggiore
- da fisse diventano volatili
- la componentistica per bit occupa piu' spazio
  - da condensatori passiamo ai Flip-Flop nei registri

La gerarchia di memorie e' una gerarchie di `Cache` di una rispetto alla precedente

- la RAM fa da cache per l'HD
  - le istruzioni di un programma sono copiate da HD a RAM per essere eseguite
- la Cache fa da cache per la RAM
- I Registri fanno da cache per la RAM

1.5 Struttura di I/O

CPU connessa a dispositivi di I/O

- connessi da BUS

Ogni dispositivo e' controllato da un `controller` hardware

- ogni controller e' un piccolo processore
  - con
    - registri
    - memoria interna
      - `buffer`
      - dove il controller trasferisce i dati del dispositivo
- OS interagisce con il controller
  - attraverso software:
    - `driver`
  - specifica nei registri del controller le operazioni da compiere
    - il controller eseguira' quello che gli e' specificato
      - utilizzando il suo buffer
    - una volta completato invia `interrupt` al driver
      - OS riprende il controllo
      - preleva dal buffer

Questa gestione e' adeguata solo per piccole quantita' di dati

- inefficiente per moli maggiori
- per inviare interi blocchi di dati dal controller al RAM

  - `DMA` Direct Memory Access
    * canale diretto tra dispositivo e RAM
    * OS tramite driver istruisce il controller, poi non viene coinvolto
      · prendi blocco numero n su HD e trasferisci in RAM a partire dalla locazione di indirizzo xxxx

1.6 Multitasking & Time-sharing

1.6.1 Multitasking

Mantenere in memoria principale piu' programmi insieme ai dati di questi in modo da aumentare la `produttivita'`

- quando un programma si ferma temporaneamente (per eseguire operazioni di I/O) l'OS ha gia' in RAM un secondo programma a cui assegnare la CPU
  - `job`

1.6.2 Timesharing

- interattivita'
- sistemi multi utente In caso la CPU non abbia tempo di idle durante l'esecuzione dei programmi
  - Il tempo di CPU sara' distribuito tra gli utenti e i loro programmi
    * da' l'impressione di simultaneita' (solamente apparente)

1.7 Modalita' di Funzionamento

1.7.1 Doppia Modalita'

- Specificata da un `bit di modalita'`
- Esistono istruzioni protette che sono eseguibili solo in modalita' di sistema (quindi dall'OS)
  - i programmi utente usano le `system call` per operazioni che richiedono l'esecuzione di istruzioni privilegiate
    * in realta' provocano eccezioni

    * l'OS gestisce (kernel mode) e poi restituisce il controllo all'utente

  – realizzate attraverso eccezioni che cambiano il bit di modalita'

NORMALE

SISTEMA | KERNEL | MONITOR | SUPERVISOR

1.7.2 Timer

`for(;;)i++;` ciclo che non termina mai

- Per questi casi e' disponibile in CPU un Timer, dopo un certo tempo inizializzato dal OS viene inviato un `interrupt`
  – utilizzato anche in caso di Time Sharing (quanto di tempo concesso ai processi)
  – il timer e' gestito con istruzioni provilegiate
    * per evitare usi impropri malevoli

1.7.3 Protezione della Memoria

Evita la sovrascrittura delle aree di memoria di programmi in RAM da parte di altri programmi in esecuzione

- Soprattutto le aree dedicate all'OS
- Due registri in CPU
  – base
  – limite Ogni indirizzo generato dal programma in esecuzione viene confrontato con i valori contenuti nei registri
    * se non contenuto viene generata una `Trap`

1.8 Strutture dei Sistemi Operativi

Livelli di complessita' e di accesso

- alcuni sono invisibili agli utenti

1.8.1 interfaccia col sistema operativo

non fa parte del kernel, ma e' fornito insieme all'OS

- interpreti di comandi | shell Unix
  - comandi == eseguibili
- GUI - interfaccia grafica
  - prima diffusione commerciale - 1984 Macintosh

1.8.2 programmi/servizi di sistema

non fanno parte del kernel, ma forniti insieme all'OS rendono piu' semplice l'uso del sistema

- editor
- compilatori
- assemblatori
- debugger
- interpreti
- IDE
- browser
- gestori di email

1.8.3 chiamate di sistema

processo == programma in "esecuzione"

- un processo deve compiere una operazione privilegiata
  - System Call
- le system call sono la vera interfaccia tra processi e OS
  - procedure inserite in programmi scritti in linguaggi di alto livello
  - sembrano normali subroutine ma l'esecuzione e' portata avanti direttamente dal'OS
- esempi:

  - open() `restituisce file descriptor`
  - write()
  - close()
  - fork()
- API
  - Application Programming Interface
  - strato intermedio tra applicazioni e system call
    - semplificano l'uso e la portabilita'
  - Api Windows / Api POSIX
  - esempi:
    - fopen() `restituisce file pointer`
    - fprintf()
    - fclose()

1.8.4 gestione dei processi/memoria primaria/memoria secondaria

- `Processi concorrenti`
  - Competono per
    1. CPU
    2. spazio in memoria
    3. dispositivi INPUT/OUTPUT
- Gestione dei processi
  - Creazione | fork()
  - Sospensione e Riavvio
  - Sincronizzazione
  - Comunicazione
- Gestione Memoria Primaria

  - un programma in esecuzione e' caricato in memoria primaria (vedi Memoria Virtuale)
  - Time-Sharing
    * tenere traccia delle aree di RAM utilizzate e da che processo
    * distribuzione della RAM tra i processi
    * gestione dinamica della RAM
- Gestione Memoria Secondaria | File System
  - informazioni del sistema contenute in un `file`
  - file organizzati in una struttura gerarchica
    * `File System`
  - strumenti del OS
    * creazione
    * cancellazione
    * gestione file e directory
    * memorizzazione efficiente

1.8.5 protezione e sicurezza

Ogni processo deve essere protetto dalle attivita' improprie degli altri processi

- non deve essere possibile impadronirsi di una risorsa in modo esclusivo
- non devono essere accessibili aree di memoria assegnate ad altri processi

Nessun utente puo' accedere a file di altri utenti
`Macchine Virtuali`

- ogni utente usa la VM indipendentemente dell'hardware
- l'utente ha l'illusione di avere una CPU, un File System
  - nella realta' le risorse sono condivise

1.9 Problemi

1. tener traccia di tutti i programmi attivi nel sistema
    - stanno usando la CPU
    - richiedono l'uso della CPU
        - `processi thread`
2. CPU libera: a quale programma in RAM assegnare la CPU
3. interazione tra programmi senza danneggiarsi
    - evitare stallo `deadlock`
    - problemi di `sincronizzazione`
4. gestione della RAM
    - traccia delle aree di memoria occupate e da che programma
        - `memoria centrale`
        - `memoria virtuale`
5. gestione del File System
    - memoria di massa
        - gestire in modo efficiente e
    - fornire un'interfaccia
    - implementare il file system

1.10 NB

- Single/Multi-Core
    - '90 CPU == singolo
        * un unico programma poteva utilizzare piu' CPU
        * sistema multiprocessore
            · tutti i processori condividono un'unica memoria principale
            · UMA

  - 2000
    * l'aumento delle prestazioni rallenta sensibilmente
    * processori costituiti da 2 processori affiancati sullo stesso Die
      · 2 Core
      · Processore Dual-Core
    * piccoli sistemi UMA
      · tutti i core possono indirizzare la stessa memoria principale
      · si condivide anche un livello di cache (L3) solitamente

- Non esiste una grande differenza tra OS per single-core o multicore
  - in questo corso si presume che esiste un'unica unita' di calcolo
- OS di Rete e OS distribuiti

2 GESTIONE PROCESSI

Componente del OS: `CPU Scheduler`

- Sceglie processi in coda di ready
- si attiva ogni 50/100 secondi
  - crea `overhead`

2.1 Processi

Unita' di lavoro del OS

- il primo ruolo del OS e' amministrare i processi
  - creazione
  - cancellazione
  - scheduling dei processi

- sincronizzazione e comunicazione

  Un processo non e' solamente un programma in esecuzione

  * Struttura in Memoria `immagine del processo`
    · Codice
    · dati
    · stack
    · heap

  Un programma puo' definire piu' processi

  * un programma puo' contenere codice per generare piu' processi
  * piu' processi possono condividere lo stesso codice

  Fondamentalmente:

  * processo: entita' `attiva`
  * programma: entita' `statica`

  Un processo nasce sempre a partire da un'altro processo attraverso una opportuna `System Call`

  * tranne il primo: all'accensione #systemd#

2.1.1 Stati di un processo

L'OS sposta il processo tra vari stati attraverso cui esso evolve

- `New`
  - Va assegnato `Process Control Block` e `spazio in memoria` necessario per il codice e i dati
  - questo e' gestito con interrupt, l'OS deve controllare subito perche' non conosce la natura dell'interrupt e non puo' lasciare finire un processo in Running
- `Ready` (to run)
  - Scheduler Dispatch - componente del OS che sceglie e lancia il processo
  - Ci sono livelli di priorita' per i processi

        * un processo a bassa priorita' potrebbe rimanere in attesa del suo turno per sempre

- `Running`
    - L'OS gira nel tempo tra un processo e l'altro, altrimenti sta in attesa (sleeps)
    - Se piu' processi: dopo un determinato tempo l'OS prende il controllo inviando un `interrupt`
        * il tempo di esecuzione puo' essere interrotto da interrupt ma verra' poi restituito subito dopo al processo in questione
- `Waiting`
    - Il processo puo' aver richiesto una operazione di I/O (con una System Call)
        * queste operazioni sono sotto il controllo del OS, quindi sara' questo a interrompere il Waiting una volta completate
- `Terminated`
    - Il processo termina
    - L'OS riprende il controllo per ripulire la memoria dall'area occupata dal processo ora terminato

diagramma DI TRANSIZIONE DEGLI STATI DI UN PROCESSO

- Rimuovere l'arco interrupt
    - da' il diagramma di un OS multitasking ma non time-sharing

2.1.2 *Process Control Block - PCB*

- Process ID
- Stato
- Contenuto dei registri della CPU una volta sospeso il processo
- Indirizzi RAM aree dati e codice
- File in uso

- Informazioni Scheduling

E' il PCB del processo che viene inserito in coda di ready dopo che l'OS ha recuperato il codice e caricato in RAM

```
int main(){ //NB: ad un programma possono corrispondere
 ↪  piu' di un processo
    pid_t pid, childpid;
    pid = fork(); //genera un nuovo processo copiando
     ↪  codice e dati del padre,
                //nel PID indica gli indirizzi che lo
                 ↪  riguardano
                //nella cella di memoria del PID
                 ↪  padre scrive il PID del figlio
                //nella cella di memoria del PID
                 ↪  figlio scrive 0
    printf("questa la stampano padre e figlio"); //sia
     ↪  padre che figlio riprendono dopo il fork
    if(pid == 0){
        printf("processo figlio");
        execlp("/bin/ls", "ls", NULL); //specifica il
         ↪  codice da eseguire, NB non ritorna
    }
    else{ //eseguito dal padre in quanto in pid
     ↪  contiene un numero maggiore di 0
        printf("sono il padre, aspetto il figlio");
        childpid = wait(NULL); //Waiting Queue, i due
         ↪  processi si sincronizzano
                              //a processo figlio
                               ↪  terminato viene
                               ↪  scritto il PID figlio
                              //a questo punto il
                               ↪  padre viene
                               ↪  reintrodotto nella
                               ↪  Ready Queue
        printf("il processo figlio e' terminato");
        exit(0);
    }
} //System Call: fork(), execlp(), wait(), exit()
```

Il codice e' copiato solo concettualmente, le aree dati sono realmente duplicate

- System Calls utili
  - getpid()

        * restituisce il process Id del processo chiamante

    - getppid()

        * restituisce il process Id del parent del processo chiamante

2.1.3 *Operazioni su processi*

CREAZIONE

- ogni OS possiede almeno una `System Call` di creazione

    - tutti i processi nascono da altri processi `con l'eccezione` di quello all'accensione del Sistema

- nel sistema si forma un `albero di processi`

    - Il Creatore e' detto Padre - `parent`

    - Il Creato e' detto Figlio - `child`

      Nel Creare un albero l'OS riferisce i processi con un `PID` (Process ID) ovvero un identificatore

        * Comando:

            · ps - process status

#+SOURCE-START

Scelte ingenieristiche

Moderni OS implementato tutte queste combinazioni nelle loro System Call

1. Avvio

    a) Processo padre continua concorrentemente al figlio - ready queue

    b) Processo padre di ferma attendendo l'esecuzione del figlio - waiting queue

2. Esecuzione

    a) Fornire al figlio copia del codice padre

    b) Nuovo programma al figlio

UCCISIONE

- kill / TerminateProcess(Win)
  - secondo PID
  - puo' avvenire se `TRAP`
  - se il processo utilizza troppe risorse
  - se il processo padre muore (non in Windows o Unix)

2.1.4 Comunicazione tra processi

INDIPENDENTI Non si influenzano l'uno con l'altro

COOPERANTI si influenzano l'un l'altro

- si scambiano informazioni
- portano avanti una elaborazione suddivisa tra i processi

Per permettere cio' l'OS deve mettere a disposizione meccanismi appositi

Inter-Process Communication `IPC`

L'OS mette a disposizione System Call volte all'implementazione di:

- memoria condivisa
  - sovrascritto il divieto della memoria dell'altro processo
  - Scelte implementative
    - dimensione variabile?
    - che processi hanno diritto di uso?
    - un processo
- scambio di messaggi
  - coda di messaggi
    - gestita dal OS
      - Scelte implementative
      - coda usata da piu' di due processi?
      - limite alla dimensione della coda?

- ricevente se non ci sono messaggi? sospensione?
- trasmittente se la coda e' piena? sospensione?

Esempi di System Call

* msgget()
* send(message, line, PID)
* receive(message, line, PID)

1. Pipe
2. Client-server
   a) Socket
   b) Remote Procedure Call `RPC`
3. Produttore - Consumatore processo `produttore`, produce informazioni utilizzate da un processo `consumatore`
   - informazioni poste in un `buffer`

     `produttore` - Compilatore ~ produce codice oggetto `consumatore` - Assemblatore ~ consuma codice oggetto

```
#define SIZE 10
typedef struct {...} item;
item buffer [SIZE];
int in = 0, out = 0;
```

in: prossimo item libero out: primo item pieno buffer vuoto: in=out buffer pieno: in+1 mod SIZE = out –il buffer e' utilizzato in modo circolare NB: Il buffer pieno usera' `SIZE-1 posizioni`

```
item nextp;
repeat
while (in == out) // empty buffer
    do no_op;
nextp = buffer[out];
out = out+1 mod SIZE;
<consuma l'item in nextp>
until false;
```

```
item nextp;
repeat
<produci nuovo item in nextp>
while(in+1 mod SIZE == out) // full buffer
    do no_op;
buffer[in] = nextp;
in = in+1 mod SIZE;
until false;
```

2.2 Thread

syscall: UNIX: clone(); Windows: CreateThread()

Se due processi (con spazi di indirizzamendo separati) devono lavorare sugli stessi dati sará necessario:

- area di memoria condivisa
- utilizzo di messaggi per lo scambio dei dati
- dati in un file acceduto a turno
  - comunque passaggio attraverso memoria secondaria piú lenta

Inoltre al context switch é introdotto un overhead

- disattivazione aree codice e dati dell'uscente
- attivazione aree codice e dati dell'entrante

Cache CPU

- contengono dati dell'uscente
- entrante genererá subito molti miss di cache

Per questo nascono i thread

- `peer thread`
  - "processi" che condividono lo spazio di indirizzamento
    - codice e dati

Un processo semplice `HWP` é contraddistindo da un unico thread di computazione

2.2.1 Peer Thread

Un processo Multi-Thread o Multi-Threaded é composto da piú thread di computazione detti `peer thread`

- un processo di questo tipo é anche detto `task`
- ad ogni `peer thread`
  - corrisponde un suo thread computazionale
    - registri CPU
    - stack privato
    - Program Counter
  - informazioni condivise, spazio di indirizzamento
    - codice
    - dati
      - unica copia delle strutture dati del codice
    - file

2.2.2 Context Switch

Avviene normalmente in modo che ciascun thread possa eseguire

- vengono cambiate solo
  - PC
  - Registri CPU
  - Stack
- Molto piú veloce
  - non vanno attivate diverse Page-Table
  - miss cache minori
    - i dati utilizzati dai peer thread é probabile siano gli stessi

2.2.3 Scheduling Thread

LIVELLO USER

LIVELLO KERNEL OS mantiene strutture dati per gestire sia i normali processi che tutti i peer thread di un task Quando un thread si blocca volontariamente o meno, il OS assegna la CPU:

- a un altro peer-thread dello stesso task
- a uno dei peer-thread di un altro task
- ad un altro processo

2.2.4 *Vantaggi*

- efficienza (Solaris)
  - un LWP richiede 30 volte meno tempo per essere creato rispetto un HWP
  - context switch 5 volte piú veloce
- condivisione di dati e risorse
- architettura multi-core
  - ancora emeglio architetture multi-threaded

2.2.5 *Architetture Multi-Core*

Molto adatte a gestire multiplici thread

- con due core
  - core1 puó gestire solo peer-thread di un task
  - core2 puó gestire solo peer-thread di un altro task

I context switch saranno solamente tra peer-thread con massima efficienza I core sono sempre in costante tentativo di bilanciamento in un sistema reale

- dipendentemente da
  - numero di core
  - numero di task
    - numero di peer-thread
  - numero di processi

CPU/CORE MULTITHREADED Attraverso una pipe-line é possibile eseguire fino a 4 o 5 istruzione del programma in esecuzione `multiple issue`

- Pipeline
  - IF
  - ID
  - EX
  - MEM
  - WB
- Architettura Superscalare
  - ogni core deve essere dotati di piú ALU e Unitá Floating Point

Questo non é sempre possibile

- le istruzioni devono essere indipendenti tra di loro
  - non devono avere bisogno del risultato di un'istruzione precedente

Invece le istruzioni di due peer-thread distinti saranno probabilmente indipendenti

- esecuzione in parallelo di istruzioni appartenenti a thread diversi
  - `Simultaneous Multi-Threading`
    * aumento di produttivitá della CPU

2.3 Scheduling

Presupponendo un sistema Single-core L'OS fa credere ai processi di avere tutta la CPU per loro

- Process Switch/Context Switch
  - L'`unico` PC viene aggiornato con i valori relativi al processo Running

    NB: Diagramma di Gantt

2.3.1 Context Switch

Passaggio da un processo in esecuzione all'altro `Commutazione` della CPU tra i processi

- OS prende il controllo CPU ~ questo e' tecnicamente pure un Context Switch
- Salva lo stato della computazione del processo uscente in `PCB`
- Scrive in PC e nei registri CPU i valori PCB del processo entrante

  Questa operazione richiede tempo: `overhead` di sistema (sovraccarico)

2.3.2 Code di Scheduling

OS gestisce varie code di processi

- una lista di `PCB`
  - mantenuta in una delle aree di memoria riservate al OS
- Coda di Ready ~ Ready Queue ~ `RQ`
  - coicide con lo stato Ready nel `diagramma`
- n Code di Waiting
  - Code dei dispositivi `Device Queues`
    - piu' processi possono essere in coda per l'accesso ad un dispositivo
    - ogni dispositivo ne ha una
  - Code di Eventi `Waiting Queues`

DIAGRAMMA DI ACCODAMENTO riformulazioe del diagramma di transizione prendendo in considerazione le code

2.3.3 Implementazione

Tecniche per massimizzare la produttivita' della CPU

- `Multitasking`
- `Time Sharing` Per cio' devono essere definite delle regole dal progettista

  I processi vivono fasi di `CPU-burst` e `I/O-burst` I processi possono essere

  - CPU-bound
    * un compilatore x es
  - I/O-bound
    * un browser
    * un editor

SCHEDULER anche Short Term Scheduler decide quale processo in coda di ready sara' eseguito quando:

1. il processo in esecuzione passa volontariamente in stato di waiting
2. il processo in esecuzione termina
3. il processo in esecuzione viene `obbligato` a passare allo stato di ready
   - questo con un timer hardware - `vettore delle interruzioni`
4. un processo P_x entra in coda di ready arrivando da un coda di wait oppure e' stato appena lanciato
   a) l'OS interviene per gestire il `PCB` di P_x spostandolo in coda di ready
   b) se P_x e' piu' importante del processo in esecuzione

per 1. 2. e' sufficiente un OS multitasking

DISPATCHER

- implementa il 2.2.2
- passa in user mode
- ripristina il PC della CPU alla corretta locazione

Tempo impiegato per queste operazioni detto `Dispatch Latency`

SENZA DIRITTO DI PRELAZIONE `non-preempive scheduling` Casi 1. e 2.

- I processi non posso interrompere l'esecuzione di altri processi

Implementazione piu' snella utilizzata per OS specifici

CON DIRITTO DI PRELAZIONE `preemptive scheduling` Casi 1. 2. 3. e 4.

- I processi non possono eseguire a tempo indeterminato
- I processi possono avere priorita' diverse

Implementazione utilizzata per OS general purpuse
Se una `System Call` chiamata dal processore in esecuzione viene `interrotta dal vettore di interrupt`?

- la prima istruzione della System Call puo' essere un'istruzione che
  - `disattiva gli interrupt`
- ultima istruzione
  - `riabilitazione degli interrupt`

Criteri

Obiettivi:

- massimizzare `uso CPU`
- massimizzare il `Throughput`
  - ovvero la produttivita'
- minimizzare il `tempo di risposta`
  - importante per i processi interattivi
- minimizzare il `Turnaround time`
  - tempo medio di completamento di un processo
    * da quando entra per la prima volta in coda di ready fino a quando non termina l'esecuzione in stato running
      · per semplificare non si considera la creazione e la terminazione del processo
- minimizzare il `Waiting time`
  - somma del tempo passato dal processo in `coda di Ready`

`Turnaround Time` = WaitingT + RunningT

Algoritmi

Considerando in questo corso processi con un `unico burst di CPU` e `nessun burst di I/O`

Un Algoritmo tanto é migliore quanto le sue prestazioni di avvicinano da `SJF` allontanandosi da `FCFS`

- Starvation #def#
  - il processo non viene mai scelto in quanto mai di prioritá
    - Aging
      - il processo aumenta di prioritá con il tempo passato in RQ

1. First Come, First Served `FCFS`
   - Normale coda FIFO
     - PCB inserito in fondo alla coda
     - CPU libera assegnata al primo PCB alla testa
     - Non-preemptive - non adatto a sistemi real-time
     - Tempo di attesa elevato
       - non implementa time-sharing
     - Effetto convoglio ~ accodamento job piu' corti
   - Osservazioni
     - sfavorisce i processi brevi
     - non implementa sistemi time-sharing
     - Peggiore degli Algoritmi ragionevoli
2. Shortest Job First `SJF` ~ Shortest Next CPU Burst
   - Esamina la durata del prossimo burst di CPU dei processi in RQ
   - assegna la CPU al processo con burst minimo
   - Puó essere `preemptive` o `non-preemptive`

     a) Preemptive - `Shortest Remaining Time First` `SRTF`

        - se in RQ é presente un processo il cui `CPU-burst é minore del tempo di esecuzione rimanente` al processo Running, ha la priorita' il nuovo processo e viene interrotto quello in stato Running
            * Ipotesi solamente teorica

    - É dimostrabile che SJF é `ottimale`
        - spostando un processo breve prima di uno lungo
            * si migliora l'attesa del processo breve piú di quanto di peggiori l'attesa del processo lungo
                · quindi diminuisce anche il Turnaroud-time medio

    - `MA`
        - la durata del prossimo burst di CPU non é nota
            * SJF non é implementabile

3. Priority scheduling `PS` calcolo della prioritá:
    - interna al sistema
        - sulla base di ogni processo
    - esterna al sistema
        - sulla base del utente Puó essere `preemptive` o `non-preemptive`

Round Robin

`RR` L'algoritmo di implementazione del time-sharing, la RQ e' utilizzata come una coda circolare

- ogni processo ha un `quanto di tempo` implementato da un timer hardware che invia un interrupt allo scadere del tempo
    - entro il suo tempo il processo non lascia la CPU se non per wait
    - alla fine del suo tempo il processo é interrotto
- il prossimo processo ad andare in esecuzione sará il primo in RQ

Con n processi in coda di ready e il quanto di tempo q ogni processo riceve $1/n$ del tempo della CPU e nessun processo aspetta piú di $(n-1)q$ unitá di tempo

- Turnaround medio peggiore di SJF
  - ovviamente
- Tempo di risposta medio migliore di SJF

Prestazioni dipendenti da q:

- $q \to \infty$
  - RR == FCFS
- $q \to 0$
  - aumenta l'illusione di `parallelismo`
  - aumenta il numero di `context switch`
    - e quindi l'overhead
- regola empirica per max turnaround
  - 80% dei CPU burst $< q$

Multilevel Queue

`MQ` Code multiple

- foreground – RR
  - interagiscono con l'utente
- background – FCFS
  - non interagiscono
- batch
  - la loro esecuzione puó essere differita

Si puó suddividere la RQ in piú code

- gestire ogni coda con un algoritmo ottimale
- Scelta:

  - prioritá fissa
    * possibile starvation
    * time slice
      · quanti di tempo maggiori per foreground, minori per background e batch

1. Multilevel Feedback Queue `MFQS` Code multilivello con retroazione
   - I processi possono essere promossi a code a piu' alta priorità o retrocessi
   - assegnamento a coda dinamico
     - i processi sono spostati dal OS per
       * adattarsi alla lunghezza del CPU burst
       * gestire ogni coda con lo scheduling adatto rispetto al comportamento mostrato
   - Es
     - se il processo esaurisce il quanto assegnato dalla prima coda RR, sara' spostato alla coda RR successiva con un quanto maggiore
     - se il processo esaurisce i quanti delle code RR sará spostato in una coda FCFS

Multielaborazione Simmetrica

`SME`

- scheduler per ogni core
  - code condivise
    * sincronizzazione
- code private ai core ~ preferita dagli OS moderni
  - necessario un sistema di bilanciamento tra le RQ dei core
    * difficoltá dovute a cache a piú livelli

- dati e istruzioni di un processo sono man mano indirizzati e copiati nei vari livelli di cache
- se spostato su un'altro core le informazioni vanno recuperate in quanto contenute in cache private di un altro core
- OS possono relegare un processo particolare ad un unico core per questo motivo

2.3.4 Esempi di Scheduling

SOLARIS

- Scheduling a code multiple con retroazioine
  1. real time
  2. sistema
  3. interattiva
     - 60 livelli di prioritá - 50-59
  4. timesharing
     - 60 livelli di prioritá - 0-40

Di norma i processi nascono nella classe timesharing I processi seguono prioritá formattate cosí:

Priority	Quantum[m/s]	New priority (exhausted quantum)	New priority (unexhau
0	200	0	
...	...	...	
59	20	49	

I processi possono essere promossi o meno in base al quanto che hanno sfruttato

- maggiore é la prioritá maggiore é la probabilitá che verrá scelto per l'esecuzione al prossimo ciclo ma minore sará il quanto a lui assegnato dal OS

Processi di sistema e real time hanno prioritá fissa, maggiore di interattiva e time sharing

- lo scheduler calcola la prioritá globale di un processo
  - prioritá == si usa RR
  - algoritmo preemptive

WINDOWS Prioritá con retroazione e prelazione

- 32 livelli
  - real time - 16-31
  - altri - 1-15

Lo scheduler sceglie il processo a prioritá piú alta

- se il processo va in wait
  - viene alzata la sua prioritá
    - dipendentemente dalla tipologia del wait
      - se é atteso un dato dal disco l'aumento é minore
- in caso di prioritá uguale é utilizzato il RR
  - se il quanto viene esaurito la sua prioritá é abbassata
    - limite 1

Favorisce i processi che interagiscono con mouse e tastiera Inoltre distingue tra background e foreground

- il processo foreground ottiene 3 volte l'aumento del quanto di tempo che gli altri processi

LINUX Completly Fair Scheduler `CFS` Cerca di distribuire a tutti i processi equamente il tempo di CPU

Ad ogni context switch lo scheduler calcola il quanto tempo che spetta ad un processo P in modo che tutti i processi abbiano avuto la stessa quantitá di tempo di CPU

- P.vruntime = $P.expected_{runtime}$ - $P.due_{cputime}$
  - CPU data al processo con P.vruntime piú basso
    - CPU-use minore
- i processi ready-to-run sono nodi di un albero di ricerca bilanciato: `red-black tree` o R-B tree
  - permette operazioni molto efficienti
    - O(logx)
  - i nodi sono inseriti con la chiave del P.vruntime
    - il nodo piú a sinistra sará quello scelto dallo scheduler

2.4 Sincronizzazione

I processi possono cooperare, perció dovranno condividere dei dati

- é necessario evitare la creazione di `dati inconsistenti`

Devono sincronizzarsi `Problema`

- mentre P1 elabora dati che verranno usati da P2 viene rimosso dall'esecuzione
  - P2 non dovrá lavorare sui dati incompleti lasciati da P1
- esempio
  - produttore - consumatore
    - utilizzata variabile condivisa buffer/counter (buffer circolare)
      - se produttore esegue counter++ 'mentre' consumatore esegue counter–
      - questo puó verificarsi perché quella eseguita non é una operazione `atomica`, non utilizzano una sola istruzione ISA a livello di architettura

La sincronizzazione é un problema solamente se si effettuano scritture su memoria condivisa

- le operazioni da sincronizzare devo concludersi completamente e non essere interrotte dallo scheduler per passare al processo sincronizzato dati consistenti

Va sviluppato un protocollo usato dai processi che vanno ad usare variabili condivise Il codice sará strutturato in questo modo: `entry section`

- richiesta di entrare nella sezione critica `sezione critica exit section`
- segnalazione di uscita dalla sezione critica

Una soluzione al problema avrá queste proprietá

- `Mutua Esclusivitá`
  - mai ci saranno conflitti di accesso
- `Progresso`

  - se la sezione critica non sta venendo eseguita allora un processo in futura ne avrá accesso
  - questo garantisce l'assenza di `deadlock`

- `Attesa Limitata`
  - qualsiasi processo che richiede di accedere alla sua sezione critica non soffrirá di `starvation`
  - evitare attese infinite

Una soluzione corretta deve permettere ai processi di computare indipendentemente dalla loro velocitá

- non deve dipendere dallo scheduling del sistema

2.4.1 Sezione Critica

Zona del codice di manipolazione delle variabili condivise, non deve `intrecciarsi` con altre sezioni critiche

- se un processo P_i sta eseguendo una sua sezione critica allora altri processi P_j non possono eseguire la propria
- L'esecuzione della sezione critica di un P_i é mutualmente esclusiva con l'esecuzione delle sezioni critiche di altri P_j
  - anche se interrotto dalla scheduler nessun altro processo manipolante

NEL SISTEMA OPERATIVO

- accesso contemporaneo alla tabella dei file aperti
- uso contemporaneo della fork
  - devono avere diversi PID

In un sistema operativo il problema é risolto con una scelta

- `kernel con diritto di prelazione`
  - un processo in kernel-mode puó essere interrotto da un altro processo
  - migliore per un sistema per applicazioni real-time
    * minore tempo di risposta

- `kernel senza diritto di prelazione`
  - in kernel-mode un processo non puó essere interrotto
    - inaccettabile in sistemi real-time
  - implementazione semplice: disattivazione degli interrupt
  - un solo processo alla volta puó accedere alle strutture dati dei kernel
    - accesso in modo esclusivo al codice della System Call

Soluzione:

- istruzioni macchina particolari
  - TestAndSet(v)

```
boolean TestAndSet(boolean *lockvar){
    boolean tempvar = *lockvar;
    *lockvar = true;
    return tempvar;
}
```

Poi usata cosí

```
boolean lock = false; // shared var
do{
    while(TestAndSet(&lock)); // while senza corpo
    //sezione critica            qui la variabile di
    ↪  lock == true
    lock = false;             // quando l'altro
    ↪  processo eseguirá il ciclo passerá il test
} while(true);
```

- In questo modo se un altro processo che testa lock resterá nel while in quanto while(&lock) == while(true)
- l'$_{\text{Attesa}}$ Limitata_ non é garantita
  - un processo potrebbe uscire dalla sezione critica e rientrarci nello stesso quanto di tempo
  - un meccanismo di aging non serva in quanto i processi entrano in esecuzione solamente che non riescono ad eseguire

  - puó essere implementata con una versione piú complessa

- Busy Waiting:
  - il processo che tenta di accedere ad un lock fa busy-waiting
    * in quanto cicla in base ad una variabile che é modificabile solo da un altro processo
      · con un RR:
      · con N processi lo spreco di tempo di CPU sará $N - 1$ quanti di tempo
    * risolvibile con la disattivazione degli interrupt
      · perdita di controllo per un tempo arbitrario del OS
      · ci si deve fidare che il processo riabiliterá gli interrupt

- Swap($v1$,$v2$)

Queste sono istruzioni macchina e quindi atomiche, non saranno mai interrotte a metá da un context switch I passi sono:

- il processo tenta di accedere al lock
- esegue la sezione critica
- restituisce il lock

NB La mutua esclusione in sistemi multi-core é piú complessa

Semafori

Dijkstra - 1965 Semaforo S: variabile strutturata operabile tramite operazioni atomiche:

- wait(S) ALIAS: P, down

```
while S <= 0 do no-op;
S = S-1;
```

- signal(S) ALIAS: V, up

```
S = S+1;
```

S é detta variabile semaforica, come se fosse un oggetto condiviso da tutti i processi per la sincronizzazione

La variabile la chiameremo mutex (mutual exclusion)

```
P {
    do{
        wait(mutex);
        // sezione critica
        signal(mutex);
    } while(true);
}
```

`sync`

```
sync = 0;
P1{
    S1;
    signal(sync);
}
P2{
    wait(sync);
    S2;
}
```

Questo tipo di semafori soffre ancora di busywaiting, sono chiamati spinlock Soluzione implementata utilizzando System Call

- lista di semafori memorizzata nelle aree dati del kernel
- System Call
  - `sleep()` ALIAS: block()
    - toglie il processo dall'esecuzione
      - non viene inserito nella Ready Queue
  - `wakeup()`
    - rimette il processo in Ready Queue
- implementazione

```c
typedef struct{
    int valore; // se > 0 indica sezione critica libera
    struct process *waiting_list;
}semaforo;

wait(semaforo *S){
    S->valore--;
    if S->valore < 0 {
            // aggiunto processo a S in waiting_list
            sleep(); // il processo si é addormentato
             ↪  sul semaforo
    }
}

signal(semaforo *S) {
    S->valore++
    if S -> valore <= 0 {
            // togli un processo P da S -> waiting_list
            wakeup(P); // risvegliato P, va in Ready
             ↪  Queue
    }
}
```

- NB
  - wait e signal sono esse stesse sezioni critiche perché usano le stesse aree dati
    - risolvibile con una interruzione di interrupt o con busywaiting perché queste sono System Call e molto brevi
      - interruzione degli interrupt in multiprocessori non ovvio: sono disattivati solo su un particolare core
  - $|mutex|$ = numero di processi addormentati
    - una S < 0 indica (in valore assoluto) il numero di processi addormentati su quel semaforo
      - se mutex = 1 allora P_1 entra e $mutex = 0$, context switch
      - un P_2 testa mutex, $mutex = -1$, P_2 si addormenta

  - Utilizzabile un valore di semafori > 1 allora una risorsa é utilizzabile da piu' P contemporaneamente

I semafori se utilizzati non correttamente possono provocare deadlock e starvation

ESEMPI Problemi di sincronizzazione risolti utilizzando semafori

Produttori e Consumatori

- buffer circolare[SIZE]
  - memoria condivisa da tutti i produttori e tutti i consumatori
- semafori
  - full
  - empty
  - mutex
- in
- out

```
while(true){
    produciItemInNextp();
    wait(empty);
    wait(mutex);
        buffer[in] = nextp;
        in = in++ mod SIZE;
    signal(mutex);
    signal(full);
}
```

```
while(true){
    wait(full);
    wait(mutex); // in caso di piú consumatori e piú
    ↪ item nel buffer
        nextc = buffer[out];
        out = out++ mod SIZE;
    signal(mutex);
    signal(empty);
```

```
    consumaItemInNextc();
}
```

Lettori e Scrittori

Condivisione di un file tra molti processi

- alcuni processi richiedono la sola lettura
  - possono essere paralleli
- alcuni richiedono la scrittura
  - richiede la mutua esclusione di tutti i processi

1. Readers First Variabili:
   - condivise
     - semaforo mutex = 1
     - semaforo scrivi = 1
     - int numlettori = 0

```
wait(scrivi);
scriviFile();
signal(scrivi);
```

```
wait(mutex);
numlettori++;
if numlettori == 1
    wait(scrivi);
signal(mutex);
leggiFile();
wait(mutex);
numlettori--;
if numlettori == 0
    signal(scrivi);
signal(mutex);
```

E' garantita l'assenza di Deadlock e Starvation?

- no

    - uno scrittore addormentato su scrivi dovra aspettare la terminazione di tutti i lettori, se continuano ad aggiungersi lettori ci sara' un Deadlock

2. Writers First
3. Fair

Cinque Filosofi

- 1 tavolo circolare
    - 5 posti
    - 5 piatti
    - 5 bacchette condivise
        * 2 necessarie per mangiare

Ogni risorsa e' associata ad un semafori in un array

```
do{
    wait(bacchetta[i]) // context switch qui causa
    ↪  Deadlock
                       // Attesa Circolare
    wait(bacchetta[i+1 mod 5]);
        mangia();
    signal(bacchetta[i]);
    signal(bacchetta[i+1 mod 5]);
        pensa();
}while(true);
```

2.5 Deadlock

Programma A aspetta informazione dal Programma B che aspetta... Il deadlock non é affrontato dagli OS, deve essere gestito dagli utenti

- se uno dei due processi cede il passo risolviamo la deadlock ma non la `starvation`

Modello del Sistema

- Tipi di risorse R (per esempio cicli di CPU)
    - ognuna formata da istanze indistinguibili tra loro
- Processi P

  - hanno bisogno di alcune istanze di R

In una situazione di `attesa circolare` le risorse possono rimanere bloccate, quindi questo é un problema di tutto il sistema L'OS potrebbe implementare delle soluzioni con adeguate rappresentazione del grafo di assegnazione delle risorse

- rappresentazione di ogni istante di
  - risorse assegnate ad ogni processo
  - risorse attese da ogni processo
- se si verifica un ciclo in questo grafo é chiara la situazione di deadlock e allora viene risolta
  - causa un sottoutilizzo delle risorse (poiché non evita i deadlock di per se)
- oppure si potrebbe evitare i deadlock verificando prima di concedere una risorsa che questa non porti ad una attesa circolare
  - troppo dispendioso dal punto di vista della computazione per l'OS

3 GESTIONE MEMORIA

3.1 Centrale

Bisogna decidere come spartire lo spazio di memorizzazione tra i processi attivi

- l'immagine di un processo inattivo nei prossimi cicli di CPU puo' essere spostato su hard disk
- quando un processo rientra in RAM occupera' spazio prima occupato
  - Questo e' lo `swap`
    * o avvicendamento dei processi

Obiettivo massimizzare il numero di processi in Memoria Principale per aumentare la multiprogrammazione

3.1.1 Binding

Associazione degli indirizzi

- ad ogni variabile di un programma va associato un indirizzo che ne contiene il valore
- alle istruzioni di salto va associato l'indirizzo di salto in caso questo avvenga

In fase di Compilazione

- generato codice assoluto o statico
- il compilatore deve conoscere l'indirizzo della cella a partire dalla quale verra' caricato il programma per poter portare a termine il binding
- se il processo e' spostato in memoria secondaria
  - dovrá essere messo allo stesso indirizzo
  - o ricompilato ad un nuovo indirizzo

In fase di caricamento in RAM

- generato codice staticamente rilocabile
- il compilatore associa indirizzi relativi all'inizio del programma (indirizzo 0)
- indirizzi assoluti generati in fase di caricamento
- se il processo e' spostato in memoria secondaria
  - piu' efficiente in quanto é in fase di caricamento che vengono risolti i riferimenti

In fase di esecuzione aka `binding dinamico degli indirizzi`

- generato codice dinamicamente rilocabile
- il codice utilizza sempre e solo indirizzi relativi
  - questi sono risolti solo al momento dell'esecuzione dell'istruzione in particolare
- necessita un supporto hardware per non perdere efficienza
  - registro di rilocazione

        * indirizzo di partenza in cui e' caricato il programma in esecuzione

    - MMU - Memory Management Unit

        * risolve gli indirizzi relativi in assoluti

- cosi non ci sono complicazioni nel spostare un processo da un'area all'altra

LIBRERIE 2 tipi:

- `Statiche`

    - Associata dal compilatore o dal loader e collegata al programma in memoria

        * anche se la subroutine non e' utilizzata viene memorizzata in memoria principale

    - Ogni programma dovra' avere una copia del codice della libreria in quanto direttamente associati

    - provoca duplicazione del codice in memoria

- `Dinamiche`

    - caricate a runtime

        * solo dopo una specifica invocazione in corso di esecuzione il Sistema Operativo interrompe e carica in RAM il necessario prima di ridare il controllo al programma

    - diversi programmi condividono la stessa porzione di codice in RAM se chiamano la stessa libreria

        * viene caricata una sola volta eliminando la duplicazione di codice

    - una nuova versione della libreria e' automaticamente caricata dal programma, non ci sara' bisogno di ricompilare i moduli per compilare la nuova libreria

3.1.2 Spazi degli indirizzi

tag: Memorie

Ogni indirizzo di un programma in un sistema allocato dinamicamente sará sempre compreso tra 0 e un max

- Questo spazio e' chiamato spazio degli indirizzi o spazio di indirizzamento logico
- Gli indirizzi sono definiti `logici` o `virtuali`
  - questi sono convertiti in indirizzi `fisici` dal **registro di rilocazione**
    * somma del registro e del registro logico a livello hardware
    * indicano una determinata cella in RAM
- Analogamente c'é uno spazio di indirizzamento fisico
  - da $r + 0$ a $r + max$

`NB`: il numero di bit per la memorizzazione degli indirizzi logici puo' essere diverso da quello per la memorizzazione degli indirizzi fisici

- allora lo spazio degli indirizzi logici sara' piu' piccolo in quella architettura
  - un programma avra' un limite di grandezza e memorizzazione

Questo é il caso piú frequente, infatti in caso di indirizzi fisici a 64 bit, questi sono troppi in casi normali:

- sono indirizzabili 2^{40}B ovvero 1TB con un indirizzamento di 40 bit

Solitamente vale questa relazione: $|RAM|_{effettiva} < |RAM|_{max} << |PhisSpace| < |VirtSpace|$

- Questo é possibile grazie la memoria virtuale

3.1.3 Tecniche di Gestione della memoria

SWAPPING Salvataggio in memoria secondaria di un `immagine` del processo non in esecuzione (`swap out`) e ricaricarla successivamente (`swap in`) prima di dargli la cpu

- `area di swap`
  - area di harddisk ad uso esclusivo del OS
- l'operazione di `swap in` posiziona il processo in una diversa area di MP
  - viene aggiornato il `registro di rilocazione`
- grande `overhead` causato dallo spostamento su disco
  - tecnica abbandonata
    * ora sostituita dalla memoria virtuale
      · e' spostato solo una parte del programma

ALLOCAZIONE CONTIGUA A PARTIZIONI MULTIPLE FISSE NB: tecnica utilizzata dal IBM OS/360 Memoria Principale suddivisa in 2 partizioni

- OS
  - stessa area del vettore delle interruzioni
- Processi Utente
  - occupata solo da un processo nei casi piu' semplici
  - `registro limite` protegge la memoria primaria riservata al OS
  - un registro di rilocazione permettera' la risoluzione del indirizzo fisico
  - Le partizioni sono di dimensione fissa
    * non necessariamente uguali
    * ogni processo puo' accedere solo alla sua porzione
      · registri di rilocazione aggiornati ad ogni context switch

· registro limite aggiornato con la dimensione della partizione

Limiti

- Questa tecnica limita il grado di `multiprogrammazione` al numero di partizioni previste
- Inoltre si verifica `frammentazione`
  - interna perche' nessun processo occupera' esattamente la partizione assegnata
  - esterna perche' le frammentazioni interne si sommano per uno spreco globale

ALLOCAZIONE CONTIGUA A PARTIZIONI MULTIPLE VARIABILE Partizioni misurate sulla grandezza dei processi

- Questo crea buchi di RAM sempre piu' piccoli e numerosi tra i processi durante l'evoluzione dell'esecuzione
  - sara' sempre piu' difficile utilizzare lo spazio in quanto troppo frammentato

Scelta della partizione

- First Fit
  - utilizzata prima partizione abbastanza grande
- Best Fit
  - utilizza piu' piccola partizione abbastanza grande
- Worst Fit
  - utilizza la partizione piu' grande

Limiti

- Frammentazione esterna aumenta con il tempo
- Frammentazione interna in quanto costa troppo tenere traccia dei buchi piccoli tra i processi e questi rimarranno nascosti

Soluzione

- Rilocazione dei processi in maniera contigua

  - quindi sara' necessaria una implementazione dinamicamente rilocabile
  - verso il basso o l'alto
- Compattamento
  - creazione di un unica area libera di memoria
  - la compattazione puo' richiedere molto tempo e rende il sistema inutilizzabile

PAGINAZIONE

- Vedi IA-32

Area di memoria allocata da un processo suddivisa in pezzi non contigui

- `Frame` o `Pagine Fisiche`
  - pezzi di dimensione fissa in cui e' divisa la Memoria Principale aka spazio di indirizzamento fisico (potenze di 2)
    * a differenza dalla `segmentazione`
- `Pagine`
  - pezzi di dimensione identica ai frame in cui e' suddiviso lo spazio di indirizzamento logico

L'OS carica x pagine cercando x frame liberi, il cui ordine e posizione non e' importante
Architettura di Paginazione

- `Page-Table`
  - array con cui tiene traccia degli indici di pagine e frame
- Traduzione Indirizzi Logici
  - questo implementa la `traduzione` tra indirizzi paginati logici a indirizzi paginati fisici
    * pagina `p` indice della tabella per ottenere il frame `f` che lo contiene
      · nella entry `p` si trova l'indirizzo di partenza del frame puntato

        * offset d (displacement) utilizzato a partire dall'indirizzo fisico del frame f

            · questo e' sommato all'indirizzo puntato da p nella tabella delle pagine per ricavare l'indirizzo fisico

- Elenco dei frame liberi
    - aggiornato ogni volta che e' necessario

Indirizzi Logici reimplementati

- non piu' lineari
- nuova implementazione - pero' sotto alcune condizioni coincide con l'implementazione lineare
    - coppia di valori `(page, offset)`
        * numero della pagina da indirizzare
        * offset rispetto all'inizio della pagina

L'hardware `impone` alcune dimensioni

- bit indirizzo logico - m
- dimensione del frame - 2^n
    - n bit di offset
    - $m - n$ bit per indirizzare le pagine
- Spazio di Indirizzamento Logico
    - $2^{m-n} \times 2^n$

In questo caso l'OS deve adeguarsi al hardware cui e' posto, cosi' facendo la sequenza lineare i valori degli indirizzi fisici e' interpretata come coppia di valori

- bit piu' significativi come numero del frame
- bit meno significativi come offset d

Questo e' implementato in modo piu' semplice utilizzando come grandezze di indirizzamento potenze di 2

- in questo modo:

  - non sara' necessario memorizzare l'indirizzo di partenza del frame ma solo il suo numero
  - non sara' necessario operare una somma tra indirizzo e offset ma solamente una concatenazione (piu' veloce e semplice a livello hardware)

Vantaggi

- `Protezione` dello spazio di indirizzamento
  - Un processo puó solo indirizzare i frame contenuti nella sua tabella delle pagine
    * quei frame contengono le pagine che appartengono al processo stesso perché é l'OS a costruire la tabella
- evita `frammentazione esterna`
  - quella `interna` rimane tipicamente nell'ultima pagina di un processo
    * mediamente mezzo frame é sprecato per ogni processo
- é una forma di `rilocazione dinamica`
  - ad ogni pagina corrisponde un diverso valore del registro di rilocazione
    * ogni frame indirizzato dalla pagina (indirizzo logico) é di fatto/agisce come un registro di rilocazione
      · ma in questo caso non c'é piú un controllo sulla validitá del offset
      · ovvero che l'indirizzamento non esca dai limiti del processo

Dimensioni Le pagine storicamente sono aumentate di dimensioni col tempo

- > la dimensione
  - > la frammentazione interna
  - < la lunghezza della tabella delle pagine

Svantaggi

- page-table
  - per ogni processo attivo
- frame-table
  - frame liberi
  - frame occupati
    - da che pagina
    - da quale processo
- ad ogni context switch
  - OS
    - attiva table del processo assegnato
    - disattiva il processo uscito
- ogni accesso alla Memoria Principale passa attraverso la paginazione
  - la traduzione da indirizzi logici a fisici deve essere efficiente
    - se la tabella delle pagine é piccola puó essere contenuta in registri della CPU
      - soluzione utilizzata dal PDP-11 ma ora le dimensioni non lo permettono
      - 8 registri per la PT
      - indirizzi da 16 bit
  - essendo la tabella contenuta in MP (in una zona riservata dal OS)
    - in questo modo per ogni indirizzo logico richiesto dal processore il `numero di accessi alla MP raddoppia`

TLB

Una tecnica di `caching` della PT mediante memoria associativa messa a disposizione della CPU

- Memoria Associativa
  - `translation look-aside buffer` `TLB`
    - una tabella di entry
      - ogni entry ha
      - chiave - pagina
      - valore - frame
    - per ogni chiave questa é cercata in parallelo
      - questo permette una grande efficienza
    - Solo parte della PT é caricata in TLB
      - la ricerca puó fallire `miss` o meno `hit`
      - in caso di miss si va ad utilizzare la PT e fare un doppio accesso alla memoria principale
      - questa é fatta partire insieme alla ricerca in TLB, se quest'ultima ha successo viene semplicemente annullata
      - la coppia mancante viene caricata in TLB
      - se piena si sovrascrive least recently used
    - al context switch viene svuotato
  - `page-table base register` `PTBR`
    - al context switch basta modificare questo registro per attivare la PT del processo mandato in esecuzione

1. i7 Due livelli di cache di indirizzi TLB
   - L1 - a sua volta divisa in due cache
     - instruction-address
       - 128 entry

  - data-address
    - 64 entry
  - costo virtualmente nullo
- L2
  - 512 entry
  - costo di 6 cicli

In caso di miss il costo sará piú di 100 cicli giá solo per la ricostruzione dell'indirizzo

Pagine condivise

La paginazione facilita la condivisione di codice in quanto questo non cambia durante l'esecuzione e tutti i processi possono leggerlo in maniera sicura

- la pagina condivisa puó essere usata per contenere una libreria dinamica

PAGINAZIONE A PIU' LIVELLI Paginazione Gerarchica

- le page-tables possono raggiungere grandi dimensioni
  - la soluzione puó essere paginare anche le tabelle delle pagine
    - PT interna richiede una PT esterna
      - quest'ultima indica dove si trovano le pagine in cui é divisa la PT interna
      - La PT esterna richiede a sua volta di essere paginata per le grandi dimensioni nel caso reale
      - Questo perfino in architetture da 32 bit (di indirizzamento logico)
      - SPARC
      - 3 livelli
      - Motorola 68030
      - 4 livelli

            - Uno schema a 4 livelli non basta per architetture a 64 bit
            - UltraSPARC
            - sarebbero richiesti 7 livelli
            - overhead altissimo in caso di miss nel `TLB`
            - in quanto si dovrebbe seguire la catena delle tabelle delle pagine per ricostruire l'indirizzo fisico

Tabella delle Pagine Invertita `IPT`

Adottata in alcune architetture a 64 bit

- 1 tabella per tutto il sistema
  - indice di ogni entry corrisponde al numero di un frame nella memoria principale
    - il corrispettivo di una normale PT
  - Ogni entry é una coppia
    - `<process-id, page-number>`
  - Ogni indirizzo logico generato dalla CPU é una tripla
    - `<process-id, page-number, offset>`
  - Per generare l'indirizzo fisico
    - si `ricerca` nella IPT `<PID, page-number>`
      - questo in linea di principio impone un enorme overhead
      - vorrebbe dire centinaia o migliaia di accessi a MP
      - risolvibile con una `memoria associativa`
      - il controllo é fatto in parallelo
    - l'indice i a cui si é trovata la coppia e offset d generano
      - <i, d>
- si riduce lo spazio occupato a discapito del tempo di traduzione

3.2 Virtuale

Tecniche che permettono di eseguire processi in cui codice e/o dati non sono completamente caricati in Memoria Primaria

- esempi
  - codice per trattazione di errore potrebbe non essere mai utilizzata in una data esecuzione
  - array liste e tabelle spesso dichiarate di dimensioni maggiori di quelle effettivamente utilizzata
  - librerie dinamiche caricate in RAM solo se e solo quando sono effettivamente usate

3.2.1 Vantaggi

- possibile eseguire programmi piú grandi in memoria principale
- possibile l'esecuzione contemporanea di processi che occupano piú spazio della MP disponibile
- maggiore multiprogrammazione e throughput della CPU
- velocitá maggiore di prima esecuzione
  - non é necessario caricare completamente i processi in memoria principale

3.2.2 Svantaggi

- aumento traffico tra RAM e HD
- esecuzione rallentata
- prestazioni complessive possono degradare
  - `thrashing`

3.2.3 Tecniche

PAGINAZIONE SU RICHIESTA `Demand Paging` Portare una pagina in Memoria solo al momento del primo indirizzamento di una locazione della pagina stessa

- un'indirizzazione su una pagina che non si trova in Memoria
  - si ha `Page Fault`

        * l'OS interviene con il `Pager` per recuperare la pagina su Memoria Secondaria

            1. il processo é messo in uno stato `waiting for page`
            2. intanto lo scheduler sostituisce il processo con qualcun'altro
            3. una volta fatto aggiorna il bit di validitá

- bit di validitá
    - ad ogni entry della page table indica se si trova in Memoria Principale
        * un bit di validitá o scatta la trap `page fault`

Pure Demand Paging

Un processo é eseguito senza nessuna delle sua pagine in Memoria Principale

- la prima istruzione indirizzata da PC (inizializzato da OS)
    - `page fault`

Demand Paging

Almeno qualche pagina é caricata in Memoria Principale

Supporto Hardware

Mentre la paginazione puó essere aggiunta in qualsiasi sistema la memoria virtuale necessita un hardware specifico

- le istruzioni devono essere rieseguibili dopo page fault

Oppure

- CPU deve controllare bit di validitá di tutti gli operandi prima di eseguire l'istruzione

Prestazioni

ma Tempo di accesso MP se dato é presente p probabilitá di page-fault
eat Effective Time Access t Tempo gestione del page-fault
$eat = [(1 - p) \times ma] + [p \times t]$
t consiste grossolanamente al tempo di recupero della pagina dalla Memoria di Massa

- in quanto nell'ordine dei ms invece che µs

Perció per non degradare troppo le prestazioni il numero di page-fault deve essere molto basso

- questo puó anche essere fatto aumentando di dimensioni le pagine
  - pagine grandi danno meno page-fault in media

Considerazioni sulle pagine

Pagine piccole implicano

- PT piú grandi
- meno frammentazione interna
- peggiori prestazioni nell'uso dell'HD perché seek e latenza del disco sono costanti
- maggiori page-fault in media

AREA DI SWAP

Memorizzazione Pagine

Utilizzo di meccanismi piú semplici ed efficienti di quelli utilizzati per il filesystem

- non vengono utilizzati file
  - questo per evitare l'uso dei `file descriptor`
- sono utilizzati blocchi piú grandi rispetto i normali file
- copia dell'eseguibile di qualsiasi processo nell'area di swap alla sua esecuzione
  - tempo di avvio aumenta
  - swap piú grandi
  - migliora tempo di gestione page-fault
    - il recupero delle pagine in swap é piú efficiente che non la normale Memoria Secondaria
      - questo a causa del file system

Liberare Spazio

L'idea della memoria virtuale é proprio di

- eseguire un processo piú grande della memoria primaria
- esecuzione contemporanea di processi che assieme occupano pú spazio di quello disponibile in RAM

All'avvenire di page-fault se tutti i frame sono occupati

- OS libera un frame rimuovendo la `pagina vittima`
  - se contiene dati modificati e/o fa parte dello stack o della heap di un processo
    * va salvata nello swap, in modo da essere recuperabile dal processo
    * in questo caso il tempo di gestione raddoppia
    * `dirty bit`
      · utilizzato per individuare le entry della PT la cui pagina relativa é stata modificata da quando é entrata in Memoria Principale
  - se contiene codice
    * non deve essere salvata, c'é nel file system
      · se erano giá copiate nello swap potranno essere recuperate piú velocemente
    * in questo caso il tempo di gestione migliora

Problemi

1. Scelta delle pagine vittima

   a) Algoritmo di sostituzione delle pagine Ottimalmente scelta una pagina che non provocherá page-fault nel futuro
      - altrimenti sará uno spreco di tempo e lavoro

      Per test si utilizzano le sequenze di riferimenti durante l'esecuzione
      - si ignorano le ripetizioni

Un maggior numero di frame causerá meno page-fault

La sostituzione delle pagine sará `locale`

i. FIFO Pagina vittima quella che é da piú tempo in memoria principale

   - inizializzazione del processo
     - buon candidato vittima
   - inizializzazione di una variabile utilizzata per tutto il processo
     - pessima candidata vittima

   Soffre della `Anomalia di Belady`

   - piú frame possono aumentare i page-fault

ii. OPT - MIN Optimal / Minimal Pagina vittima quella che sará usata piú in lá nel tempo

   - ovviamente `non implementabile`
   - utilizzato come confronto per altri algoritmi

iii. LRU Least Recently Used Pagina vittima quella che non é stata usata per piú tempo

   - si avvicina piú a OPT che FIFO
   - difficilmente implementabile in modo efficiente
     - dovrebbe avere un supporto hardware non disponibile normalmente
       * sarebbe necessario un timer per tener traccia

   Approssimazione `Reference Bit`

   - bit associato ad ogni entry della PT
     - quando una pagina é indirizzata il bit é settato ad 1

iv. Algoritmo della Seconda Chance Soffre dell'`Anomalia di Belady` Utilizza i reference bit per approssimare LRU

- Utilizza una coda FIFO circolare
  - la pagina in coda:
    * se ha reference bit 1 gli viene data una seconda chance
      · azzera il bit e prosegue la coda
  - nel caso peggiore equivale a FIFO

v. Algoritmo della Seconda Chance Migliorato Utilizza sia

- dirty bit
- reference bit

4 classi (r,d)

- (0,0) ottima per essere sostituita
- (1,0)
- (0,1) va salvata in memoria secondaria
- (1,1) peggiore candidata

vi. Tecniche aggiuntive Uso del pool di frame liberi, non assegnati normalmente a nuovi processi

- pagine con dirty bit spostate qui prima di essere salvate
  - a tempo perso, quando l'OS é abbastanza libero

2. Allocazione dei frame I `frame` vanno `distribuiti` tra i processi in modo

- `Uniforme`
  - stesso numero per tutti

- `Proporzionale`
  - in base alla loro dimensione
- `Prioritá`

Da quale gruppo di pagine scegliere la vittima?

- `Allocazione Globale` - Unix
  - escluse le pagine dell'OS
  - turnaround di un processo fortemente influenzato dai processi con cui convive
    * molto dipedente dall'esecuzione
- `Allocazione Locale` - Windows
  - troppe pagine ad un processo possono peggiorare il throughput
    * in quanto gli altri processi causeranno page-fault causando l'intervento dell'OS

L'allocazione `globale` porta un `throughput maggiore` sperimentalmente in sistemi time-sharing

3. Thrashing Questo avviene quando il grado di multiprogrammazione diventa troppo alto Un `page-fault` porta alla rimozione di una pagina di un altro processo
   - questo causerá successivi page-fault da altri processi

   Si innesca un `circolo vizioso`
   - processi passano il loro tempo in waiting
   - OS passa il suo tempo a gestire page-fault

   L'`utilizzo della CPU crolla`
   - questo puó ingannare gli utenti che vedendo un basso utilizzo di CPU
     - questo puó causare ancora piú problemi di thrashing
   - `Prevenzione del fenomeno`

  - `Gestione della frequenza dei page fault`
    * Si stabilisce una soglia da non eccedere con i page-fault
    * si puó anche stabilire che se la frequenza sia troppo bassa é possibile aumentare il grado di multiprogrammazione
  - `Sostituzione Locale` puó mitigare il problema
    * in questo modo i processi non andranno a prendersi frame a vicenda
  - `Quantitá sufficiente di memoria principale`

3.2.4 Struttura dei Programmi

Queste possono aumentare il numero di page fault Per esempio:

- array, allocati per righe
  - se acceduti per colonne si rischia di aumentare i page-fault
    * le hash-table forniscono per questo prestazioni pessime con la memoria virtuale

3.2.5 Windows 10

`Demand paging with Clustering` alla creazione di un processo

- insieme di lavoro minimo
  - min di pag (50)
- insieme di lavoro massimo
  - max di pagine che SO allocherá in RAM a quel processo (345)

Inoltre

- lista frame liberi
  - associato il numero minimo di frame presenti nella lista
- sostituzione locale delle pagine

Se si raggiunge il limite minimo di frame liberi in RAM

- si libera spazio
  - vengono rimosse le pagine in eccesso dei processi che hanno ecceduto l'insieme di lavoro massimo
    * algoritmo delle seconda chance (reference bit) per Intel

3.2.6 Solaris

`Demand paging` parametro `lostfree` associato alla lista dei frame liberi

- 1/64 del numero di frame del sistema
- ogni 1/4 di secondo OS controlla che lostfree non sia maggiore del numero di frame liberi
  - se si: processo `pageout`
    * variante dell'algoritmo della seconda chance
      1. scandisce le pagine in RAM azzerandone bit di riferimento
      2. riscandisce e le pagine con il bit di riferimento a 0 vengono considerate riutilizzabili
         a) dirty bit a 1 vengono prima salvate prima di essere riutilizzate
         b) se un processo fa riferimento ad una pagina riutilizzabile in attesa di essere salvata questa viene riassegnata a quel processo
    * il tempo tra due pageout puó variare in base a parametri dell'OS
      · ma sempre nell'ordine di qualche secondo
    * se il pageout non riesce a mantenere la quantitá di frame ad un livello accettabile é possibile si stia verificando il `thrashing`
      · OS puó rimuovere tutte le pagine di un processo
      · scegliendo tra i processi inattivi da piú tempo

4 GESTIONE MEMORIA DI MASSA

4.1 Rigidi

Contrapposto al floppy-disk

- dischi

    - traccie circolari
        * settori
            · unitá minima di memorizzazione di informazione
            · solitamente 4096 Byte
        * cilindri
            · insieme delle traccie su diversi piatti
- bracci
    - testine rd/wr
        * sfiorano i dischi in rotazione
    - un braccio per ogni piatto
- `Seek time`
    - tempo impiegato dalla testina per spostarsi nella zona del settore
- `Latenza posizionale`
    - tempo che impiega un settore a trovarsi sotto la testina

4.1.1 Mappatura degli indirizzi

Array di Blocchi logici (settori) da 4096 Byte

- settore 0 il `primo della traccia piú esterna del primo disco` della pila

Mappatura complicata dalla diversa lunghezza delle tracce e dai difetti di fabbricazione OS deve ottimizzare le prestazioni

- latenza rotazionale non influenzabile dal OS
    - mediamente 1/2 del tempo di una rotazione completa
- `seek time minimizzabile` riordinando le richieste
    - minimizzando il movimento delle testine

4.1.2 Scheduling dei dischi rigidi

Interessa solo la traccia su cui si trova il settore da leggere

- si puó semplificare in quanto tutte le testine e tutti i dischi si muovono assieme

FCFS

C-SCAN La testine si muove da un estremo all'altro del piatto

- raggiunta l'estremitá del piatto torna all'inizio senza servire nessuna richiesta

La sequenza delle tracce viene trattata come una `lista circolare`

4.1.3 Formattazione

- `Basso livello`
  - formattazione fisica
  - associa indice ad ogni settore
  - prevede lo spazio per il codice di errore
  - scelta la dimensione dei blocchi fisici
    - 512 / 4096 Byte
- `Alto livello`
  - formattazione logica
  - sottoposta dal OS
  - crea lista dei blocchi liberi
    - e una directory iniziale
  - riservate le aree utilizzate dal OS
    - `boot block`
      - codice per fare partire il sistema operativo
      - codice eseguito dal codice in ROM
    - area contenente `attribuiti` dei file

        - **index-node** in Unix
        - **Master File Table** in Windows

    * area di `swap`

        - puó essere un file molto grande
        - in windows
        - pagefile.sys
        - puó essere una partizione specifica
        - non viene trattata allo stesso modo dal File System
        - viene ottimizzata per una maggiore velocitá

4.2 RAID

`Redundant Array of Inexpensive Disks` **Reduntant Array of Independent Disks** vs **Single Large Expensive Disk**

- `RAID/SLED`
- simile alla contrapposizione `RISC/CISC`
    - **Reduced Instructions Set Computer** vs **Complex Instruction Set Computer**

Aumenta la `velocitá` e la `affidabilitá` di un sistema con diversi piú harddisk

- il sistema vede l'array come un normale SLED
    - parallelizzando parte delle operazioni migliora le prestazioni
    - duplicando l'informazione su piú dischi per poterli recuperare in caso di guasto
    - non sono necessari cambiamenti a livello di Sistema Operativo

4.2.1 Livello 0

Non c'é duplicazione dei dati, non é davvero un RAID

- il RAID mappa diverse strips (`striping`)
  - ogni k blocchi consecutivi
  - suddivide gli `strip tra i dischi` seconda la formula
    - numero di strip MOD numero di dischi
  - una lettura di strip consecutivi puó essere eseguita dal controller in parallelo sui vari dischi
  - l'affidabilitá in un RAID di livello o é minore di quella di un semplice SLED

4.2.2 Livello 1

Ogni disco é usato come disco di `mirroring`

- migliora la affidabilitá
  - in caso di guasto il Sistema puó continuare a funzionare
    - non sono persi dati

4.2.3 Livello 01

Utilizza `stripping` e `mirroring`

- piú affidabile ed efficiente
  - piú costosa

4.2.4 Livello 10

Prima fatto il `mirroring` e poi lo `stripping`

4.2.5 Livello 4

Permette di gestire il recovery con meno dischi `Striping a livello di blocchi`

- calcola strip di paritá per permettere il recovery
  - strip di paritá piazzati nello stesso strip di un'altro disco
    - detto disco di paritá
      - aggiornato ad ogni modifica con il ricalcolo dello strip di paritá
      - utilizzato in media 4 volte di piú degli altri dischi

4.2.6 Livello 5

Strip di `paritá distribuiti` in tutti i dischi

- per distribuire il lavoro su tutti i dischi

4.2.7 Livello 6

`Multipla paritá`

- ma é molto raro che si rompano due dischi contemporaneamente

4.2.8 Stringa di Paritá

$s_1 \veebar s_2 \veebar s_3 =$ paritá $s_1 = s_2 \veebar s_3 \veebar$paritá

4.3 SSD

`Flash Memories` o `Solid State Disk` Anche se non sono dischi Piú veloci di un disco rigido

- blocchi o pagine
  - 2/16KB
- riscrivibili fino a ad un limite di circa 100000 volte
  - ogni volta che una pagina viene riscritta va prima cancellata
    - **flashing**
    - questo rende la scrittura molto piú lenta che la lettura
- non necessita scheduling in quanto di `accesso diretto`
  - al contrario dell'hard disk

4.4 File System

Nasce nella seconda parte degli anni sessanta Fornisce meccanismi di memorizzazione e accesso ai dati e applicativi Consiste in

4.4.1 File

- unitá logica di informazione permanentemente memorizzata su un supporto di memoria secondaria, dotata di:
  - **nome**
  - **tipo**
  - posizione **fisica**
  - posizione **logica** nel File System
    - pathname *non é esplicitamente memorizzato da nessuna parte*
      - tranne che in casi particolari
  - **attributi**
    - dimensioni
    - diritti di accesso
    - tempo di crezione
    - proprietario
- informazioni contenute:
  - **dati**
    - numerici
    - testuali
  - **programmi**
    - sorgenti
    - linkabili
    - eseguibili
  - **documenti**
    - multimediali

Operazioni su File

- creazione

- scrittura / lettura
- riposizionamento all'interno di un file
- rimozione
- troncamento di un file
  - cancella i dati memorizzati tranne che i sui attributi
- rinominare
- spostare

Metodi di accesso *questo quando non esistevano mezzi di memorizzazione ad accesso diretto*

- `sequenziale`
- `diretto`

4.4.2 *Directory*

Un File System puó essere molto grande

- fondamentale che i `tempi di accesso` ai singoli file `non` crescano `linearmente`
  - con il numero di file
  - con lo spazio occupato

Una directory permette di risalire a tutti gli attributi di un file che contiene Permette operazioni di

- ricerca
- creazione e cancellazione
- visualizzazione
- cambiamento
- spostamento

La perdita dei dati della directory comporta la perdita dell'accesso ai file contenuti

Una directory é un **file particolare**

- non modificabile a piacimento
  - `OS garantisce l'integritá` della struttura del FS

- in `ms-dos`
  - una lista
    - nome / attributi
    - entry da 32 byte
      - 8: file name
      - 3: estensione
      - ...
- `Unix`
  - lista
    - nome / puntatore agli attributi
      - puntata una struttura interna
- `NTFS` - New Technology File System
  - albero di ricerca bilanciato
    - ogni **foglia** corrisponde ad un **file**
      - e una file reference ad una struttura interna
      - `alcuni attributi` sono immediatamente accessibili
    - il tempo di ricerca é lo stesso indipendentemente dalla posizione nell'albero

STRUTTURA

- directory `unica`
- directory a `due livelli` *nasce il concetto di pathname*
  - una directory per ogni utente
    - contenuta in un'unica directory
- directory con `struttura ad albero`
  - una `Root:` **/** in Unix e **C:**\ in Windows

        * home directory dell'utente *nasce il concetto di pathname relativo e assoluto*

            · ogni utente puó navigare nella working directory

- directory con struttura a `grafo aciclico`

    – permette di condividere file o directory con nomi diversi

        * link

            · diversi OS hanno diverse implementazioni dei link

- directory con struttura a `grafo generale`

    – una directory puó contenere il nome di una directory padre o antenata

        * pericoloso in quanto puó creare cicli all'interno del FS

La struttura procede in questo modo: Programmi Applicativi –> Fyle System Logico –> Modulo organizzazione dei file –> –> File System di base –> Controllo I/O –> Dispositivi Passaggio dalla rappresentazione `esterna` alla rappresentazione `interna`

Accesso rapido ai file

L'accesso utilizzando il `pathname` é estremamente `inefficiente`

- richiede piú accessi alla memoria secondaria

L'OS mette a disposizione l'operazione *open* che restituisce un **descrittore**

- open file table

    – in **RAM**, contiene le informazioni relative al file aperto

    – le modifiche sono prima fatte sulla copia del file in MP

Protezione dei file

Implementazioni perfette ma `non realizzabili` con

- `lista d'accesso`
- `capability list`

Implementazione in Unix

- classi di utenti
- protezioni
  - in lettura
  - in scrittura
  - in esecuzione

4.4.3 Interfaccia

4.4.4 Realizzazione

L'OS vede l'HD come un array di entry (sequenza di 256 / 4096 byte)

- l'`accesso` avviene comunicando al `controller` il numero del blocco e le operazioni
  - lettura e scrittura avvengono sempre in unitá di blocchi

Per ogni File il Sistema mantiene una struttura interna che memorizza tutti gli attributi del file

- per gestire le operazioni compiute sul file

ALLOCAZIONE I File che superano la dimensione di un blocco vengono allocati con 3 metodi

allocazione contigua - buono per l'area di swap

- necessario memorizzare solamente
  - blocco di `partenza`
  - `numero di blocchi` utilizzati
  - `grandezza` del file
- **possibile accesso diretto**

    - piú efficiente di un accesso sequenziale
        * sapendo il numero di byte da leggere
            · semplice calcolare in che blocco lo si puó trovare
- Problemi
    - necessari **blocchi liberi adiacenti**
    - necessaria una **strategia di scelta del buco libero** (first/best/-worst fit)
    - disco soggetto a **frammentazione esterna**
        * sará necessaria **ricompattazione** periodica del disco
    - se un file aumenta di dimensioni
        * **riallocarlo**
            · molto costoso
        * **sovraddimensionarlo**
            · forte frammentazione interna
            · comunque presente nell'ultimo blocco che non sará mai completamente occupato
            · il problema si ripresenterá

allocazione concatenata

- ogni `blocco contiene un puntatore al successivo`
- info aggiuntive
    - `numero di blocchi`
    - `grandezza` del file
- numero negativo all'`ultimo blocco per segnalare la fine` del file
- Vantaggi
    - non c'é **frammentazione esterna**
    - non c'é **necessitá di ricompattazione**

- Svantaggi
  - ogni blocco byte utilizzati per **memorizzazione dei puntatori**
  - **accesso diretto non possibile**
    * sempre necessario seguire la catena di puntatori
      · estremamente `inefficiente`
  - inaffidabile
    * la perdita di un blocco impica la `rottura della catena`
      · risolvibile
      · con una doppia catena
      · con la memorizzazione del file di riferimento e del numero del blocco in ogni blocco
  - necessario tenere un elenco dei blocchi liberi
- Soluzioni
  - `utilizzo di cluster` e non singoli blocchi
    * tempi di accesso minori
      · meno riposizionamenti della testina di lettura
      · meno spazio sprecato per i puntatori
      · maggiore frammentazione interna
      · mediamente sará sprecata una maggiore quantitá dell'ultima parte del cluster

1. FAT Variante di allocazione concatenata molto efficiente `File Allocation Table` Area (array) all'$_{\text{inizio}}$ della Memoria Principale_ in cui l'indice di ogni entry corrisponde ad un blocco e ne contiene il numero del successivo
   - l'ultimo blocco J di un file é segnalato:
     - alla J-esima entry contiene un marker di fine file
   - entry o corrispondono a blocchi liberi

   Svantaggi
   - Occupa **spazio in MP**
   - se la FAT é persa non c'é modo di accedere i dati
     - **va periodicamente salvata** il Memoria Secondaria

allocazione indicizzata

Utilizzato un `blocco indice poi portato in memoria principale` per indicare la posizione dei blocchi dati del file

- Vantaggi
  - non sono necessari blocchi contigui
  - non si crea frammentazione esterna
    - l'acceso diretto é efficiente una volta portato in RAM il blocco indice
      - facile calcolare quale blocco dell'elenco contiene il byte desiderato
- Svantaggi
  - anche in caso di **file piccoli**
    - sempre utilizzato un **intero blocco indice**
      - molto andrá sprecato
  - Necessario mantenere un `elenco dei blocchi liberi`
- Problemi
  - Blocco indice non sufficiente per contenere i numeri di tutti blocchi dati

1. Schema Concatenato `Ultima entry del precedente blocco indice` punta al successivo blocco indice

a) NTFS `New Technology File System`

- ogni file descritto da un elemento
  - simile all'i-node
  - sempre di `dimensione fissa e numerati consecutivamente`
  - contiene tutti gli attributi del file
  - se il file é molto piccolo é possibile contenerne i dati direttamente nel elemento
    - molto efficiente

        - o puntatori a cluster del disco che contengono dati del file
            * variante di schema concatenato
    - `Master File Table all'inizio del disco` gestito da OS
        - **numero di elemento** associato al **nome del file** nella directory corrispondente

    Necessario tenere traccia di tutti i blocchi liberi

    - `MFT`

2. Schema a piú livelli Blocco `indice esterno` che punta a blocchi `indice interni`

    a) I-Node Struttura interna Unix gestita dall'OS memorizzata in Memoria Secondaria (zona riservata)

    - ad ogni file ne viene associato uno
        - `index-node` contiene
            * **attributi**
            * **elenco dei blocchi**
        - scritto affianco al nome simbolico di un file nella tabella di una directory che lo contiene

    Ogni i-Node memorizza

    - 10 puntatori `diretti` a blocchi dati
    - 1 puntatore `single indirect`
        - punta a un blocco indice che punta a blocchi dati
    - 1 puntatore `double indirect`
        - punta a un blocco indice contenente puntatori a blocchi indice che contengono puntatori a blocchi dati
    - 1 puntatore `triple indirect`

      - punta a un blocco indice contenente puntatori a blocchi indice che contengono puntatori a blocchi indice che contengono puntatori a blocchi dati

Necessario tenere traccia di tutti i blocchi liberi

- `superblocco`
  - a partire dal blocco 1

Elenco dei Blocchi Liberi

- `MFT / Superblocco`
- `Vettori di bit` in Mac/OS
  - ogni bit indica se il blocco corrispondente é occupato o libero
    * necessario un numero di bit uguali al numero di blocchi del disco
    * necessario questa sia sempre presente in MP
- `Lista Concatenata`
  - lenta ma senza sprechi di spazio
  - possibile utilizzare la FAT per una maggiore efficienza
- `Raggruppamento`
  - in un blocco piú puntatori a blocchi liberi
    * ultimo puntatore punta ad un altro blocco di blocchi liberi
- `Conteggio`
  - mantenere il numero di un blocco e quanti blocchi consecutivi liberi lo seguono
    * simile alla lista concatenata con meno entry

EFFICIENZA | PRESTAZIONI Necessari degli accorgimenti da parte del Sistema Operativo

- `caching` in Memoria Principale dei `file` (e `attributi`) utilizzati frequentemente / recentemente
  - soprattutto **file aperti**
  - **modifiche solo in MP**
    * OS deve assicurarsi di mantenere la consistenza dei file

5 LABORATORIO

5.1 C

5.2 Unix

5.3 Progetto