APPUNTI DI SISTEMI OPERATIVI v 1.0.3

AUTORI: Niccolò Sghinolfi, Fabio Savioli AA 2023-2024

PREMESSA

Questa appunti sono stati creati in preparazione all'esame di Sistemi Operativi del docente Sangiorgi Davide per il corso di laurea "Informatica per il management" presso Unibo ed è il frutto dell'incrocio degli appunti personali presi a lezione di Niccolò Sghinolfi e Fabio Savioli integrati con le slide del corso, altri appunti di anni precedenti e risorse online (per citarne una, wikipedia).
Nascono originariamente come appunti personali e abbiamo deciso di condividerli per un nostro moto di orgoglio personale, **NON SONO QUINDI MATERIALE UFFICIALE**

QUESTI APPUNTI NON VOGLIONO SOSTITUIRSI IN ALCUN MODO ALLE RISORSE DIDATTICHE E ALLE LEZIONI TENUTE IN AULA.

Lo scopo di questi appunti è meramente di supporto allo studio riassumendo i concetti principali e per questo possono essere affetti da imprecisioni o potrebbero risultare in alcuni punti non esaustivi.

Il lettore è stato avvisato.

Chiunque voglia integrare con i suoi appunti è libero di contattarci (sarete elencati come collaboratori, non siamo freebooter!).
Siete liberi di far circolare questi appunti ricordando però di dare credito agli autori originali.

Detto questo vi auguro una buona lettura - Niccolo.

Ma prima ecco qualche memino

PREMESSA 1
- PER ACCEDERE AL FILE SU DRIVE (Condivisione) 5
- Collaboratori: 5

CONCETTI GENERALI PER SISTEMI OPERATIVI 6
- SYSTEM CALL 6
- INTERRUPT 6
- DIRECT MEMORY ACCESS (DMA) 6
- PROCESSO 6
  - STATI DEI PROCESSI 7
  - PASSAGGIO TRA STATI 7
- THREAD 7
- PRINCIPIO DI LOCALITA' 8
- MEMORIA CACHE 8
- CACHING 8
- HARDWARE PROTECTION 8
- STRUTTURA MICROKERNEL 9
- MULTIPROGRAMMAZIONE 9
- PROCESS CONTROL BLOCK (PCB) 10
- CONTEXT SWITCH 10
- CONTROLLER 10

SCHEDULING CPU 11
- SCHEDULING TIME 11
- CPU BURST e I/O BURST 11
- CRITERI OTTIMIZZAZIONE 11
- RISCHI MULTIPROGRAMMAZIONE 12
  - STARVATION 12
  - DEADLOCK 12
- ALGORITMI SCHEDULING 12
  - SHORT JOB FIRST (SJF) 12
  - FIRST COME FIRST SERVED (FCFS) 12
  - ROUND ROBIN (RR) 12
  - PRIORITY 13
    - PREEMPTIVE 13
    - NOT PREEMPTIVE 13
- MULTILEVEL QUEUE SCHEDULING 13
- MULTIPLE-PROCESSOR SCHEDULING 14
  - Homogenous Processor 14
  - Asymmetric Multiprocessing 14
  - Symmetric Multiprocessing 14
  - Processor Affinity 14
  - Process Migration 14

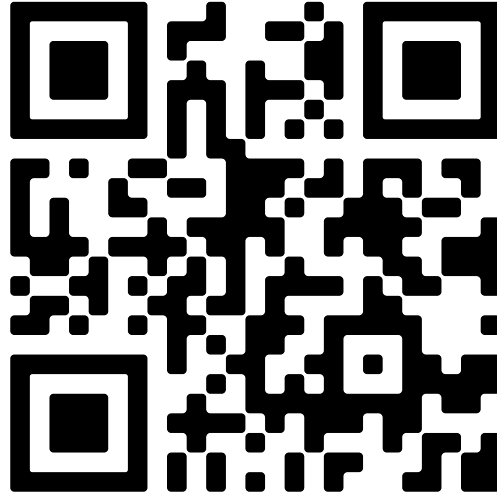
CONCORRENZA **14**
- Cos'è la Concorrenza? 14
- MUTUA ESCLUSIONE (MUTEX) 16
- RACE CONDITION 16
- SEMAFORI 16
  - TIPO DI SEMAFORO: SPINLOCK 16
  - JAVA: SYNCHRONIZED 16
  - BUFFER 16
- SEZIONE CRITICA 17
  - Algoritmo 1: 17
  - Algoritmo 2: 18
  - Algoritmo 3 (Algoritmo di Peterson): 19
- PROBLEMI TIPICI 20
  - PRODUTTORI vs CONSUMATORI 20
  - LETTORI vs SCRITTORI 21

MEMORIA CENTRALE **23**
- MEMORY MANAGEMENT 23
- SWAPPING: 23
  - FRAMMENTAZIONE 24
- ALLOCAZIONE CONTIGUA (politica più semplice) 24
- ALLOCAZIONE DINAMICA 25
- Problema dell’Allocazione Dinamica: 25
- PAGINAZIONE (PAGING) 25
  - MMU(Memory Management Unit): 26
  - PCB (Program Counter Block) 26
  - PAGE TABLE 26
  - TRANSLATION LOCATE BUFFER (TLB) 27
  - VANTAGGI PAGE TABLE 27
  - LIVELLI PAGE TABLE 27
- Problema doppio accesso alla RAM 29
- X ESERCIZI 29
- INVERTED PAGE TABLE 30

MEMORIA VIRTUALE **31**
- Demand Paging 31
- Page Fault 32
- Page Replacement (Decisione pagina vittima) 33
- ALGORITMI PAGE REPLACEMENT 33
  - FIFO Algorithm 33
  - Optimal Algorithm 33
  - Least Recently Used Algorithm (LRU) 33
  - Approssimazione LRU Algorithm (2nd Chance) 34
- ALGORITMI DI FRAME ALLOCATION 34

FIXED ALLOCATION.......... 34
PRIORITY ALLOCATION.......... 35
TRASHING (Problema della Memoria Virtuale).......... 35
FILE SYSTEMS.......... 36
TIPI DI ACCESSO.......... 36
SEQUENZIALE.......... 36
DIRETTO.......... 37
PROTEZIONE.......... 37
ALLOCAZIONE SU FILE.......... 37
ALLOCAZIONE CONTIGUA.......... 37
ALLOCAZIONE CONCATENATA (LINKED).......... 38
ALLOCAZIONE INDICIZZATA (INDEXED).......... 39
PROBLEMA JOURNALING.......... 40
FREE SPACE MANAGEMENT.......... 40
CRITTOGRAFIA.......... 41

PER ACCEDERE AL FILE SU DRIVE (Condivisione)



Collaboratori:

CONCETTI GENERALI PER SISTEMI OPERATIVI

SYSTEM CALL

Chiamate dai livelli superiori (livello utente o applicativo) per richiedere a livello kernel/hardware un servizio preciso.

INTERRUPT

Segnale asincrono che serve per indicare il “bisogno di attenzione” da parte di una periferica I/O.
Una volta si usava il Pollin(Testing) → ogni tot di tempo si fa il check del disco per capire se l’istruzione è terminata.
Bisogna settare un bit della CPU, per capire se l’istruzione è terminata controlla questo bit e in caso SO interviene.
Quando arriva una interrupt, SO cerca di capire che tipo di interrupt sia arrivata interrogando:
INTERRUPT VECTOR: Tabella che contiene una entry per tutti i tipi di interrupt → Possono essere di tipo HW o SW.
Negli attacchi informatici l’interrupt vector è uno dei bersagli principali. Esempio: processo che esegue un’istruzione che comporta l’accesso ad una parte di RAM che non gli compete → potrebbe voler rubare dati
Device Status Table: Contiene una entry per ognuna delle periferiche I/O, permette di capire al SO da dove arriva l’interrupt

DIRECT MEMORY ACCESS (DMA)

Il DMA è un meccanismo che permette alle periferiche di I/O (e altri sottoinsiemi) di accedere direttamente alla memoria in lettura/scrittura senza passare dalla CPU. Quando la periferica I/O finisce di lavorare, anziché creare una interrupt il controller DMU, che ha avuto l’ok dalla cpu di operare direttamente sul bus di fatto “scavalcando” la CPU e quindi senza bisogno di interrogarla, comincerà ad operare sugli indirizzi di memoria interessati. Questo crea un problema di concorrenza tra DMA e SO ma senza gravi conseguenze.

PROGRAMMA

Insieme di istruzioni indipendenti. Un programma può essere formato da più thread, ha accesso alla memoria. è una entità statica che non cambia nel tempo (es il codice che scriviamo sugli IDE)

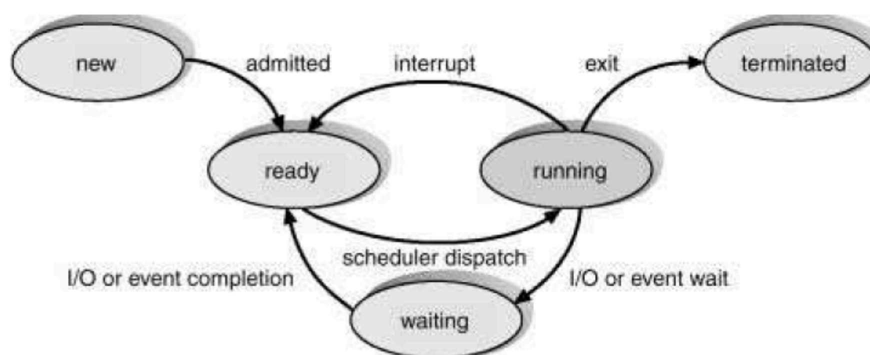
PROCESSO

Entità attiva che opera anche sulla memoria. ha bisogno di risorse come dispositivi I/O, sfrutta lo scheduling cpu, memoria e file per concludere delle task. I processi fanno uso di risorse come il PC (Program counter), ovvero un registro della CPU che tiene in memoria le

prossime istruzioni da eseguire e il punto in cui siamo arrivati in quel processo in caso di context switch

STATI DEI PROCESSI

- **new**: creazione del processo a comando utente o all'accensione della macchina
- **ready**: processo creato e pronto per essere messo in CPU ed eseguire
- **running**: Processo attualmente in corso sulla CPU
- **waiting**: processo attualmente in attesa di eventi (come terminazione di un'altro processo o attesa di I/O
- **terminated**: esecuzione terminata. può terminare volontariamente o forzatamente



PASSAGGIO TRA STATI

- **running-ready**: il processo o ha terminato il suo quanto di tempo o è stato sollevato da un interrupt. E' stato quindi sollevato dalla CPU. Il processo sarà quindi pronto a runnare non appena sarà caricato nuovamente in CPU
- **running-waiting:** il processo ha interrogato un dispositivo I/O e sta aspettando risposta. lo stato waiting si solleva solamente quando ha ottenuto la risposta
- **running- terminated:** il processo o ha terminato l'esecuzione, o ha creato un errore o è stato forzatamente terminato dall'utente (kill)

THREAD

Astrazione interna dei processi. condividono risorse, codice e memoria. servono per gestire i flussi di dati all'interno di un programma. il passaggio da thread a thread non svuota la memoria cache

- Thread più leggeri rispetto ai processi
- Thread più veloci nei context switch perchè sono all'interno dello stesso processo

VANTAGGI:
Efficienza (Concorrenza): sfruttare architetture moderne (con più CPU) e gestire molte richieste alla volta
Se un Thread si blocca, un altro lo recupera senza che si blocchi tutto.
SVANTAGGI:

SINCRONIZZAZIONE/CORRETTEZZA. un alto numero di thread aumenta empiricamente la complessità del sistema. se non sono delineati bene i compiti di ciascun thread si possono incorrere in problemi come la lettura “sporca” di un file (vedi SCRITTORI vs LETTORI)

PRINCIPIO DI LOCALITA’

Le informazioni/ i dati che un certo processo usa in un certo momento sono molto simili a quelli che userà nell’immediato futuro o che ha usato nell’immediato passato.

- Il principio è molto forte e ci permette di usare la cache in modo efficiente

MEMORIA CACHE

Estremamente veloci ma piccoli, usata per memorizzare le informazioni che sono usate più frequentemente dalla CPU. Quando la cpu interroga la cache può causare due eventi:
Cache HIT: Quando CPU trova subito informazione nella cache
Cache MISS: contrario → tempo di recupero più lento
La cache porta degli svantaggi:

- Cambio del processo: quando cambio il processo in esecuzione le info nella cache possono NON essere significative
- Inconsistenza: il dato in cache può essere usato da più programmi contemporaneamente, quindi rischio di operare su dei dati obsoleti o di leggere della memoria “sporca”

Tuttavia i vantaggi di avere dati in cache sono comunque maggiori degli svantaggi

CACHING

Idea utilizzata in più ambiti, permette di tenere in cache i dati “hot” ovvero i dati usati più frequentemente quindi senza dover interrogare memorie primarie e secondarie si ha direttamente accesso ai dati.

HARDWARE PROTECTION

Caratteristiche della macchina per venire incontro alle esigenze del SO → modifiche a livello hardware del computer per garantire al dispositivo una maggiore sicurezza.
Dual mode operation: La condivisione delle risorse di sistema richiede che il SO garantisca che un programma errato non possa causare l'esecuzione errata di altri programmi.
User Mode: esecuzione eseguita dall’utente
Monitor/System Mode: esecuzione eseguita dal SO
Per questo viene adibito un bit all’interno del processore che serve a differenziare il caso in cui ci sia un’istruzione da utente(User Mode) o da SO (Monitor/System Mode).
Quando c’è un interrupt o un errore l’HW fa uno switch da user a monitor mode
Nei vecchi computer non c’era questo meccanismo ed era molto semplice modificare parti di Sistema Operativo da utente (Hacker)
I/O Protection: Tutte le Istruzioni I/O sono privilegiate.
Deve garantire che un programma utente non possa mai ottenere il controllo del computer in

modalità monitor (ovvero, un programma utente che, come parte della sua esecuzione, memorizza un nuovo indirizzo nell'interrupt vector).
Memory protection:
Deve fornire protezione della memoria almeno per l'interrupt vector e per le interrupt service routine. Per avere protezione della memoria, vengono aggiunti due registri che determinano l'intervallo di indirizzi a cui un programma può accedere:
Base Register: indica l'indirizzo di cella di memoria di "job2"
Limit Register: indica la dimensione di "job2"
La memoria all'esterno del limite è protetta.
Quando la CPU esegue job2 controlla che l'indirizzo sia contenuto nella cella di memoria allocata → L'HW dà una grossa mano altrimenti il SO avrebbe da fare ogni volta 2 controlli.
Memory Management Unit(MMU): Contiene tutta la circuiteria hw per contenere questi registri che effettuano i controlli.
CPU Protection:
Viene aggiunto un **Timer**: interrompe il computer dopo un periodo prestabilito per garantire che il sistema operativo mantenga il controllo.
– Il timer viene decrementato a ogni battito dell'orologio.
– Quando il timer raggiunge il valore 0, si verifica un'interruzione.
• Timer Utilizzato anche per calcolare l'ora corrente.

STRUTTURA MICROKERNEL

Una volta si utilizzava un unico "blocco" per costruire SO → MS-DOS
Il primo passo è stato inserire i sistemi modulari (dividere il SO in più moduli)
Si è pensato di dividere il SO in modo GERARCHICO → Layered Approach
Si arriva oggi ad una Struttura MicroKernel che consiste nel dividere il SO in due grandi parti:

A layered design was first used in THE operating system. Its six layers are as follows:

layer 5:	user programs
layer 4:	buffering for input and output
layer 3:	operator-console device driver
layer 2:	memory management
layer 1:	CPU scheduling
layer 0:	hardware

1- Kernel: processi di base e gestione della memoria, alcune funzionalità di comunicazione, lo scambio di messaggi (questa è la caratteristica più importante di un microkernel)
2- I principali servizi del sistema operativo che vengono aggiunti come moduli che interagiscono tra loro utilizzando la funzionalità di comunicazione del microkernel.
VANTAGGI:
– estensione – portabilità
– affidabilità (se un servizio fallisce, il resto del SO rimane intatto)
– modificabilità (per modificare un servizio è sufficiente modificare un solo modulo

MULTIPROGRAMMAZIONE

consente ad una macchina di eseguire più attività contemporaneamente
PRO: maggiore efficienza di spazi e tempi. Maggiore produttività
CONTO: maggiore complessità del sistema, maggiori risorse richieste. runnare contemporaneamente più processi aumenta il consumo di energia e risorse

PROCESS CONTROL BLOCK (PCB)

Struttura tabellare di dati di un processo che contiene le informazioni essenziali per la gestione del processo come: Process ID e Process Date, informazioni sulla CPU SCHEDULING, registri principali CPU come Program Counter.
Ogni processo ha un PCB che permette di ritornare al punto in cui si era lasciato in caso di context switch.

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

CONTEXT SWITCH

Toglie processo dalla ram e ce ne mette un altro. il cambio può avvenire per terminazione del programma (o thread) o per fine quanto di tempo (vedi RR)

CONTROLLER

Sistema HW o SW che indirizza i flussi di dati tra due entità differenti come thread e programmi. un esempio sono microchip che gestiscono le periferiche di I/O
Per ogni dispositivo, gestisce le interrupt

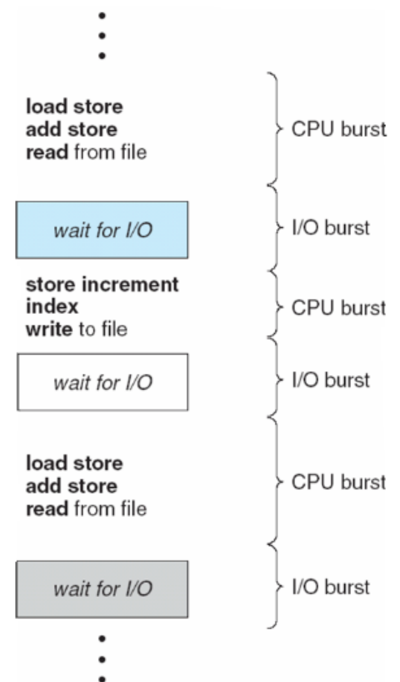
SCHEDULING CPU

In una nocciolina: La scheduling della CPU si occupa quando e che processi /thread mettere in cpu.

SCHEDULING TIME

Lo scheduling avviene su tre tempi diversi:

- LONG TIME: determina quali processi far partire e in quale ordine. viene interrogato poco spesso
- MEDIUM TIME(SWAPPER): opera quando la memoria centrale è satura, trasferisce momentaneamente processi dalla RAM alla memoria secondaria (es. deve essere caricato un processo ad alta priorità, quindi verrà messo in ready il processo running). viene interrogato ogni tanto
- SHORT TIME: serve per dare l'impressione che ci sia un processo intero per ogni processo. In realtà i processi non vengono eseguiti tutti interi ma un pezzo alla volta. viene interrogato spessissimo



CPU BURST e I/O BURST

La CPU Burst sono i periodi di breve durata in cui la CPU opera su dati o manda segnali alle periferiche I/O. Mentre le I/O burst sono i periodi di tempo generalmente lunghi in cui operano le periferiche I/O e la CPU rimane in attesa di un Interrupt inviato alla fine dell'utilizzo della periferica I/O

CRITERI OTTIMIZZAZIONE

- **CPU utilisation**: quanto viene utilizzata la cpu durante i periodi di CPU Burst
- **Response time:** quanto tempo passa tra l'input di un comando e il suo avvio. Ossia il tempo da READY a RUNNING
- **Waiting Time:** ovvero la penalità subita dalla multiprogrammazione. Indica il tempo di un programma in READY mentre aspetta di passare a RUNNING. E' il metodo di controllo dell'efficacia della multiprogrammazione (Tempo di fine – tempo di arrivo – tempo di esecuzione)
- **Turnaround time:** tempo necessario per eseguire un determinato programma (Tempo fine – tempo inizio oppure waiting time + execution (CPU burst) time)
- **Throughput:** numero maggiore di processi per unità di tempo

RISCHI MULTIPROGRAMMAZIONE

STARVATION

quando un processo non può ottenere risorse HW o SW per essere eseguito. Alcuni algoritmi di scheduling risolvono il problema con meccanismi di AGING, ovvero quando un processo è abbastanza vecchio viene "fatto passare" per permettere così la sua esecuzione

DEADLOCK

situazione molto grave che blocca tutta la macchina o uno o più processi. in questo caso una rimozione forzata o un kill da parte dell'utente è spesso richiesto. Un caso di deadlock sono due processi che per andare avanti con l'esecuzione aspettano la fine dell'altro. In questo caso runnano all'infinito causando un deadlock del sistema

ALGORITMI SCHEDULING

Gli algoritmi principali di gestione dei processi sulla CPU.

SHORT JOB FIRST (SJF)

Vengono fatti runnare prima i processi con un tempo di BURST minore. Può causare problemi di starvation
PRO: Ottimale per il waiting time medio
CONTRO: Non implementabile in forma pura (non si può conoscere il Bt con esattezza prima dell'esecuzione)

FIRST COME FIRST SERVED (FCFS)

Come la tipologia gestionale FIFO il processo che arriva prima viene eseguito per primo
PRO: Semplice, facilmente implementabile, no Starvation.
CONTRO: Waiting time potenzialmente alto

ROUND ROBIN (RR)

Algoritmo di scheduling basato sul quanto di tempo(TIME-SHARING). ogni processo può rimanere in CPU per un certo quanto di tempo. Al termine di esso può venire sostituito e fatto partire un altro processo utilizzando un CONTEXT SWITCH. Il quanto di tempo non può essere né troppo grande (vanificherebbe l'utilizzo prima del quanto) né troppo piccolo (ci sarebbero troppi content switch rendendo tutto l'algoritmo troppo costoso. I processi del RR in waiting vengono "messi in pila". Se arriva un altro processo questo viene messo in cima alla pila e partirà prima degli altri che rimarranno in attesa per un altro quanto di tempo.
PRO: Migliore response time di SJF (è reattivo!), no starvation, CPU equamente condivisa
CONTRO: Maggior turnaround di SJF

PER GLI ESERCIZI: quando arriva un programma a fine RR, va in CODA DOPO IL PROGRAMMA CHE HA FINITO DI RUNNARE! non va a inizio stack

Example: RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

- Typically, higher average turnaround than SJF, but better *response*.

PRIORITY

Vengono fatti runnare prima i processi con priorità maggiore. Questo algoritmo ha un problema di Starvation: i programmi a priorità più bassa potrebbero non runnare mai. per questo viene implementato un sistema di aging in cui viene aumentata la priorità ai sistemi in attesa da molto tempo
CONTRO: Processi con bassa priorità potrebbero rimanere molto (o per sempre) in attesa di essere eseguiti (Starvation). Soluzione -> Aging: con l'incremento del tempo, si aumenta progressivamente la priorità del processo

Tutti gli algoritmi seguono una tra le seguenti tipologie di operazione:

PREEMPTIVE

Il SO toglie il processo dalla CPU o quando finisce il quanto di tempo o quando arriva un processo a priorità maggiore

NOT PREEMPTIVE

Il processo in corso sulla CPU non viene mai tolto finché questo non finisce di operare

MULTILEVEL QUEUE SCHEDULING

La multiprogrammazione avviene anche verticalmente, ovvero avendo processi in background e foreground. Ogni livello deve gestire la sua scheduling indipendentemente dagli altri livelli e può usare algoritmi diversi (es background usa FCFS mentre foreground usa RR). lo stesso programma può avere dei processi in background e foreground. il SO può decidere di dare priorità sulla CPU a processi e thread in BG rispetto a processi in FG o viceversa.

MULTIPLE-PROCESSOR SCHEDULING

Lo scheduling diventa più difficile con più cpu

Homogenous Processor

in un sistema a più processori serve che siano uguali

Asymmetric Multiprocessing

un solo processore alla volta accede alle strutture dati del sistema.

Symmetric Multiprocessing

Ogni processore si auto-schedula

Processor Affinity

Siccome per ogni processore ci possono essere più più programmi nella relativa cache, ha più senso che quando un processore finisce la sua scheduling questo rimanga idle anziché fare un content switch per spostare i processi da una cache di un processore busy alla cache di un processore idle.

Process Migration

Situazione molto frequente negli SO Moderni.
Due liste ready, una per processore. Ce ne potrebbe essere una piena ed una vuota, quindi si effettua una migrazione → Se ne occupa un modulo del SO che ogni tanto controlla
Si chiama anche LOAD BALANCING (Bilanciamento di calcolo)

CONCORRENZA

Cos'è la Concorrenza?

Tema centrale nella progettazione dei S.O. riguarda la gestione di processi multipli
Multiprogramming: più processi su un solo processore - parallelismo apparente
Multiprocessing: più processi su una macchina con più processori - parallelismo reale
Distributed processing: più processi su un insieme di computer distribuiti e indipendenti - parallelismo reale
Esecuzione concorrente: due programmi si dicono in esecuzione concorrente se vengono eseguiti in parallelo (con parallelismo reale o apparente)
Concorrenza: è l'insieme di notazioni per descrivere l'esecuzione concorrente di due o più programmi - è l'insieme di tecniche per risolvere i problemi associati all'esecuzione

concorrente, quali comunicazione e sincronizzazione.
Differenza Multiprocessing e multiprogramming:
In un singolo processore:
- processi multipli sono "alternati nel tempo" per dare l'impressione di avere un multiprocessore
- ad ogni istante, al massimo un processo è in esecuzione
- si parla di *interleaving* (disporre i dati in maniera non contigua per migliorare le prestazioni di rilevazione errori)

In un sistema multiprocessore:
- più processi vengono eseguiti simultaneamente su processori diversi
- i processi sono "alternati nello spazio"
- si parla di *overlapping*

A prima vista si potrebbe pensare che queste differenze comportino problemi distinti: in un caso l'esecuzione è simultanea, nell'altro caso la simultaneità è solo simulata.
In realtà presentano gli stessi problemi che si possono riassumere nel seguente:
non è possibile predire la velocità relativa dei processi e quando un processo verrà eseguito.

ESEMPIO IMPORTANTE PER ESAME(Slide 8 Concorrenza):

In C:
```
void modifica(int valore) {
  totale = totale + valore
}
```
In Assembly:
```
.text
modifica:
  lw $t0, totale
  add $t0, $t0, $a0
  sw $t0, totale
```
- Supponiamo che:
  - Esista un processo P1 che esegue `modifica(+10)`
  - Esista un processo P2 che esegue `modifica(-10)`
  - P1 e P2 siano in esecuzione concorrente
  - `totale` sia una variabile condivisa tra i due processi, con valore iniziale 100
- Alla fine, `totale` dovrebbe essere uguale a 100. Giusto?

Scenario 1: multiprogramming (corretto)

P1	`lw $t0, totale`	totale=100, \$t0=100, \$a0=10
P1	`add $t0, $t0, $a0`	totale=100, \$t0=110, \$a0=10
P1	`sw $t0, totale`	totale=110, \$t0=110, \$a0=10
S.O.	`interrupt`	
S.O.	`salvataggio registri P1`	
S.O.	`ripristino registri P2`	totale=110, \$t0=? , \$a0=-10
P2	`lw $t0, totale`	totale=110, \$t0=110, \$a0=-10
P2	`add $t0, $t0, $a0`	totale=110, \$t0=100, \$a0=-10
P2	`sw $t0, totale`	totale=100, \$t0=100, \$a0=-10

Scenario 2: multiprogramming (errato)

P1	`lw $t0, totale`	totale=100, \$t0=100, \$a0=10
S.O.	`interrupt`	
S.O.	`salvataggio registri P1`	
S.O.	`ripristino registri P2`	totale=100, \$t0=? , \$a0=-10
P2	`lw $t0, totale`	totale=100, \$t0=100, \$a0=-10
P2	`add $t0, $t0, $a0`	totale=100, \$t0= 90, \$a0=-10
P2	`sw $t0, totale`	totale= 90, \$t0= 90, \$a0=-10
S.O.	`interrupt`	
S.O.	`salvataggio registri P2`	
S.O.	`ripristino registri P1`	totale= 90, \$t0=100, \$a0=10
P1	`add $t0, $t0, $a0`	totale= 90, \$t0=110, \$a0=10
P1	`sw $t0, totale`	totale=110, \$t0=110, \$a0=10

Scenario 3: multiprocessing (errato)

- Il due processi vengono eseguiti simultaneamente da due processori distinti

Processo P1:	Processo P2:
`lw $t0, totale`	
	`lw $t0, totale`
`add $t0, $t0, $a0`	`add $t0, $t0, $a0`
`sw $t0, totale`	
	`sw $t0, totale`

- Nota:
  - i due processi hanno insiemi di registri distinti
  - l'accesso alla memoria su `totale` non può essere simultaneo

Alcune considerazioni:
- Non vi è sostanziale differenza tra i problemi relativi a multiprogramming e multiprocessing
- Ai fini dei ragionamenti sui programmi concorrenti si ipotizza che sia presente un "processore ideale" per ogni processo
- I problemi derivano dal fatto che:

1- non è possibile predire gli istanti temporali in cui vengono eseguite le istruzioni
2- i due processi accedono ad una o più risorse condivise

MUTUA ESCLUSIONE (MUTEX)

La mutua esclusione è un processo di sincronizzazione tra processi che impedisce ad altri processi di ottenere dati in memoria o ad altre risorse in sezione critica.

RACE CONDITION

Più processi "corrono" per usare la cpu o una risorsa condivisa. Questo evento si può ottenere quando non c'è un "ordine di entrata" vincolato ad esempio tramite buffer. le risorse in sezione critica si dice "essere affette da race condition"

SEMAFORI

Entità "arbitro" esterne da invocare che vincolano l'accesso a sezioni critiche. Sono strutture messe a disposizione dal SO che prevedono variabili che possono solo

- incrementare (V(s)) → quando incremento vuol dire che sono uscito dalla sezione critica
- decrementare (P(s)) → quando decremento ottengo l'accesso alla sezione critica

Non è così facile come sembra, le operazioni di incremento e decremento sembrano atomiche ma non lo sono, quindi il SO durante queste operazioni interviene per garantire l'atomicità.
Per evitare race condition all'ingresso dei semafori vengono implementati dei buffer "di attesa".
Un semaforo può essere binario (cioè può assumere solamente valori 1,0. di conseguenza solamente 1 processo può entrare in sezione critica) o innario (cioè più assumere valori n,0. di conseguenza in sezione critica possono esserci al più n-1 processi)

TIPO DI SEMAFORO: SPINLOCK

Semaforo più semplice. Utilizza una tecnica di BUSY WAITING per i programmi in attesa. Ovvero i programmi attendono attivamente (B.WAITING) mentre ogni quanto di tempo una "sveglia" controlla che si sia liberata la sezione critica. Questo semaforo ha però una criticità: ovvero lo spreco di costi computazionali per tutti i processi in attesa

JAVA: SYNCHRONIZED

Struttura Java che ha già implementato i semafori (non dobbiamo implementarli noi ^^)

BUFFER

Area di memoria usata per tenere momentaneamente memorizzati i programmi in attesa di entrare in sezione critica

SEZIONE CRITICA

Sezione critica: Parte di un processo che accede a strutture dati condivise. Rappresenta un potenziale **punto di interferenza**. Quando ci sono variabili condivise bisogna assicurarsi che ci lavori un processo alla volta.
Soluzioni:
1- Mutual Exclusion:
2- Progress: Se nessun processo esegue nella sezione critica e alcuni vorrebbero eseguire non si può farli aspettare all'infinito
3- Bounded Waiting: Processo P ha uso esclusivo della variabile condivisa e la usa continuamente

Algoritmo 1:

Algorithm 1

```
public class Algorithm_1 extends MutualExclusion {
   public Algorithm_1() {
         turn = TURN_0;
   }
   public void enteringCriticalSection(int t) {
         while (turn != t)
                Thread.yield();
   }
   public void leavingCriticalSection(int t) {
         turn = 1 - t;
   }
   private volatile int turn;
}
```

Applied Operating System Concepts 7.11 Silberschatz, Galvin, and Gagne ©1999

Algoritmo 2:

Algorithm 2

```
public class Algorithm_2 extends MutualExclusion {
  public Algorithm_2() {
        flag[0] = false;
        flag[1] = false;
  }
  public void enteringCriticalSection(int t) {
        // see next slide
  }
  public void leavingCriticalSection(int t) {
        flag[t] = false;
  }

  private volatile boolean[] flag = new boolean[2];
}
```

Applied Operating System Concepts 7.12 Silberschatz, Galvin, and Gagne ©1999

Algorithm 2 – enteringCriticalSection()

```
public void enteringCriticalSection(int t) {
  int other = 1 - t;
  flag[t] = true;
        while (flag[other] == true)
              Thread.yield();
  }
```

Algoritmo 3 (Algoritmo di Peterson):

Algorithm 3

```
public class Algorithm_3 extends MutualExclusion {
  public Algorithm_3() {
          flag[0] = false;
          flag[1] = false;
          turn = TURN_0;
  }
  public void enteringCriticalSection(int t) {/* see next slides */ }
  public void leavingCriticalSection(int t) {{/* see next slides */ }
  private volatile int turn;
  private volatile boolean[] flag = new boolean[2];
}
```

Algorithm 3 – enteringCriticalSection()

```
public void enteringCriticalSection(int t) {
  int other = 1 - t;
  flag[t] = true;
  turn = other;

  while ( (flag[other] == true) && (turn == other) )
          Thread.yield();
  }
```

Algo. 3 – leavingingCriticalSection()

```
public void leavingCriticalSection(int t) {
          flag[t] = false;
}
```

PROBLEMI TIPICI

PRODUTTORI vs CONSUMATORI

Produttore: produce dati
Consumatore: consuma dati prodotti dal produttore
Vogliamo che il consumatore consumi tutti i dati prodotti

Producer

```
while (true) {

        /*  produce an item and put in nextProduced  */
         while (count == BUFFER_SIZE)
                     ; // do nothing
                buffer [in] = nextProduced;
                in = (in + 1) % BUFFER_SIZE;
                count++;
}
```

Consumer

```
while (true)  {
       while (count == 0)
              ; // do nothing
              nextConsumed =  buffer[out];
               out = (out + 1) % BUFFER_SIZE;
               count--;

                  /*  consume the item in nextConsumed
}
```

An execution leading to an incorrect value of count

Statement ++count can become, in assembly:

$$register_1 = count$$
$$register_1 = register_1 + 1$$
$$count = register_1$$

Similarly for --count:

$$register_2 = count$$
$$register_2 = register_2 - 1$$
$$count = register_2$$

The above instructions can be interleaved in a concurrent execution of ++count and --count, for instance thus:

(initial value of count is 5)

$$register_1 = count$$
$$register_1 = register_1 + 1$$
$$register_2 = count$$
$$register_2 = register_2 - 1$$
$$count = register_1$$
$$count = register_2$$

(final – and incorrect – value of count is 4)

Viene introdotto un BOUNDED BUFFER.
Questo bounded buffer tiene conto di quante celle sono piene e vuote all'interno del buffer.
PROBLEMA:
1- Count variabile condivisa, ci sono più operazioni che causano una race condition
2- Può avvenire un context switch pericoloso (Saperli riconoscere all'esame)
3- Non vogliamo che nel buffer ci siano produttori E consumatori

E' importante quindi sincronizzare l'accesso del count dai due thread (produttore e consumatore) per impedire che ci siano produttori e consumatori (usare un mutex)

SOLUZIONI COI SEMAFORI

Solution for Producer –

```
do{

//produce an item

wait(empty);
wait(mutex);

//place in buffer

signal(mutex);
signal(full);

}while(true)
```

Solution for Consumer –

```
do{

wait(full);
wait(mutex);

// consume item from buffer

signal(mutex);
signal(empty);


}while(true)
```

LETTORI vs SCRITTORI

Lettore: legge il file → non apportando modifiche possono esserci un numero n sul file di reader
Scrittore: modifica il file → HA BISOGNO DI ACCESSO ESCLUSIVO AL FILE

Già dal tipo di attori si capisce che ci deve essere una mutua esclusione tra lettori e scrittori. Infatti non voglio che ci siano lettori mentre uno scrittore sta scrivendo/modificando la pagina, altrimenti rischio di leggere un file “sporco”.
C’è quindi bisogno anche qua di un mutex in entrata del buffer.
Quando un writer entra nella sezione critica tutti i reader devono uscire.
Quando entra un reader questo tiene “aperta la porta” agli altri reader (come detto prima non c’è un limite ai reader in quanto non modificano il file in sezione critica)
Ho quindi bisogno di un READER COUNT per sapere quanti reader ci sono in sezione critica

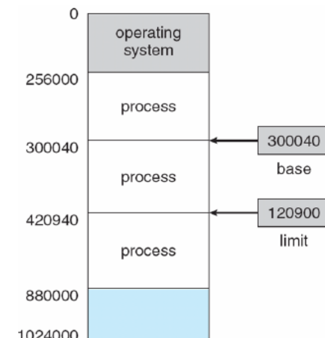
Writer Process	**Reader Process**

wait(*wrt*);
. . .
writing is performed
. . .
signal(*wrt*);
	wait(*mutex*);
	readcount := *readcount* + 1;
	if *readcount* = 1 **then** *wait*(*wrt*);
	signal(*mutex*);
	. . .
	reading is performed
	. . .
	wait(*mutex*);
	readcount := *readcount* - 1;
	if *readcount* = 0 **then** *signal*(*wrt*);
	signal(*mutex*);

mutex and *wrt* are each initialized to 1, and *readcount* is initialized to 0.

MEMORIA CENTRALE

MEMORY MANAGEMENT



Un programma deve essere portato in Memoria Centrale per essere eseguito, per farlo ci sono due metodi principali: segmentazione e paginazione.
RAM e registri sono gli unici accessibili dalla CPU direttamente.
La Cache si trova tra RAM e registri della CPU.
Per il corretto funzionamento è necessario implementare una corretta protezione.
Registro Base/Limit: Primo e ultimo indirizzo di cella di memoria per delimitare lo spazio utilizzato.
Logical Address: Prodotto dalla CPU(Virtual Address)
Physical Address: Indirizzo effettivo della cella di memoria.
Gli indirizzi logici e fisici sono gli stessi negli schemi di associazione degli indirizzi in compile-time e load-time. *Differiscono* nello schema di associazione degli indirizzi in execution-time.

L'associazione di *Indirizzi di istruzioni e dati* ad un *Indirizzo di Memoria* può avvenire in 3 fasi:
Compile Time: Se la posizione di memoria è nota a priori, è possibile generare un *absolute code*; deve ricompilare il codice se la posizione di partenza cambia.
Load Time: Deve generare codice rilocabile se la posizione della memoria non è nota in fase di compilazione.
Execution Time: Il collegamento(Binding) viene ritardato fino al momento dell'esecuzione se il processo può essere spostato durante la sua esecuzione da un segmento di memoria a un altro. Necessita di supporto hardware per le mappe degli indirizzi (ad esempio, registri di base e limite)

Chi ci consente di associare gli indirizzi è la MMU (vedi più avanti)

Dynamic Linking NON INTERESSA PER ESAME(?)

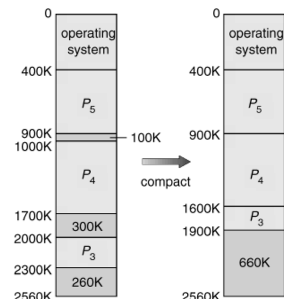
SWAPPING:

- Un processo può essere temporaneamente spostato fuori memoria in un archivio di backup e quindi riportato in memoria per continuare l'esecuzione
- *Backing Store:* disco veloce sufficientemente grande da contenere copie di tutte le immagini della memoria per tutti gli utenti; deve fornire accesso diretto a queste immagini di memoria
- *Roll out / Roll in:* variante di swapping utilizzata per algoritmi di pianificazione basati sulla **priorità**; il processo con priorità più bassa viene scambiato in modo che il processo con priorità più alta possa essere caricato ed eseguito

- *La maggior parte del tempo di swap è il tempo di trasferimento*; il tempo di trasferimento totale è direttamente proporzionale alla quantità di memoria scambiata
- Versioni modificate dello swapping si trovano su molti sistemi (ad esempio UNIX, Linux e Windows)
- Il sistema mantiene una ready queue di processi pronti per l'esecuzione che hanno immagini di memoria sul disco

FRAMMENTAZIONE

Prima cosa da nominare se si parla di problemi di allocazione contigua.
Dopo un po' di processi, ci si trova con tanti piccoli Hole.
External Fragmentation: Lo spazio di memoria totale per soddisfare una request esiste, ma non è contiguo
Internal Fragmentation: lo spazio allocato potrebbe essere più grande di quella richiesta, questa differenza di dimensione è la memoria interna di una partizione, ma non utilizzata.

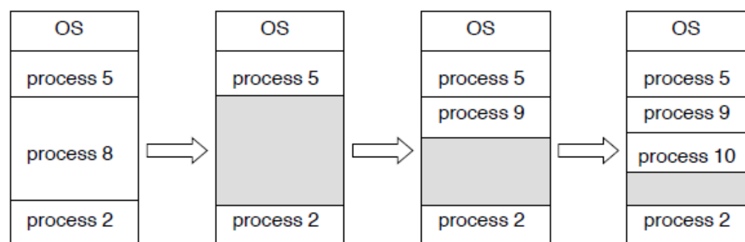


Si riduce la External Fragmentation tramite **Compattazione**:
- Si spostano i processi per lasciare un unico spazio libero
- Operazione pesante! Viene utilizzata tutta la RAM
- **N.B.** E' possibile solo se la riallocazione è **Dinamica** ed eseguita in **Execution Time**

Il SO mantiene informazioni su:
1)*Allocated Partitions*
2)*Free partitions(Hole)*

ALLOCAZIONE CONTIGUA (politica più semplice)

Schema in cui i processi vengono portati in RAM in blocchi contigui di memoria(non divisibili), ovvero in indirizzi di memoria uno di seguito all'altro



La memoria principale è solitamente suddivisa in due partizioni:
- SO residente, solitamente mantenuto in memoria ridotta con interrupt vector.
- Processi utente quindi mantenuti in memoria elevata

Relocation Register: sono utilizzati per proteggere i processi utente gli uni dagli altri e per evitare che modifichino codice e dati del sistema operativo.
- **Base Register:** contiene il valore dell'indirizzo fisico più piccolo

- **Limit Register:** contiene un intervallo di indirizzi logici - ciascun indirizzo logico deve essere inferiore al limit register
- MMU mappa l'indirizzo logico in modo dinamico

SOFFRE DI FRAMMENTAZIONE ESTERNA
Quando un processo finisce di eseguire crea un **Hole** nel blocco di memoria e il SO può decidere di allocare un altro processo in quello spazio.

ALLOCAZIONE DINAMICA

Viene usata quando non sappiamo a priori quanta memoria bisogna allocare. Utilizza i metodi di allocazione worst-best-first fit. SOFFRE DI FRAMMENTAZIONE INTERNA

Problema dell'Allocazione Dinamica:

Come soddisfare una richiesta di size n da una lista di free holes?:
First-Fit: assegna il primo Hole sufficientemente grande
Best-fit: Assegnare Hole più piccolo ma che sia sufficientemente grande; è necessario cercare nell'intero elenco, a meno che non sia ordinato per dimensione → Produce il foro rimanente più piccolo
Worst-Fit: assegnare Hole più grande; deve anche cercare nell'intero elenco. Produce il largest leftover hole

First-fit e best-fit sono migliori del worst-fit in termini di velocità e utilizzo dello spazio di archiviazione.
Best-fit potrebbe creare molti buchi piccoli non più utilizzabili.
First-Fit considerata opzione *MIGLIORE*

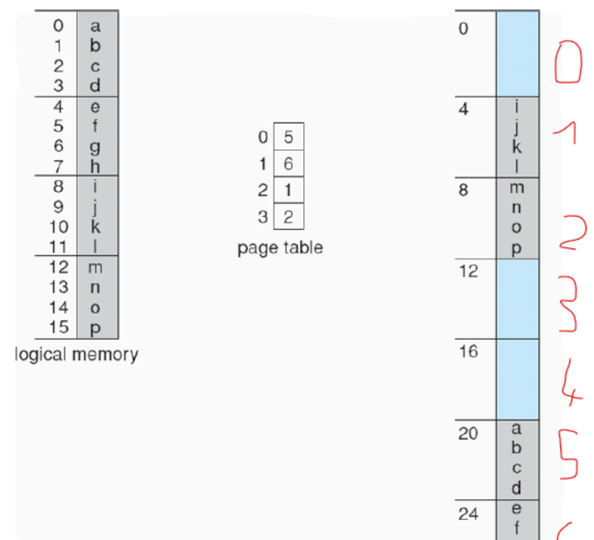
-

PAGINAZIONE (PAGING)

RICORDA: processo diviso in blocchi dette pagine, memoria divisa in blocchi detti frame

L'idea della paginazione è di permettere lo spezzettamento dei blocchi dei processi:

- Lo spazio degli indirizzi logici di un processo può essere non contiguo; al processo viene allocata memoria fisica ogni volta che questa è disponibile
- Si divide la RAM in blocchi di dimensioni fisse chiamati **Frame- entità fisica**



- Si divide la memoria logica in blocchi della stessa dimensione chiamati **Pages - entità virtuale**
- **IMPORTANTE: TAGLIA FRAME == TAGLIA PAGE**
- *Per eseguire un programma di dimensioni m pagine, è necessario trovare n frame liberi e caricare il programma*
- Configurare una **Page Table** per tradurre gli indirizzi logici in fisici
- *L'idea è di permettere a Pagine di varie taglie di essere posizionate su Frame non necessariamente contigui →*

MMU(Memory Management Unit):

- Dispositivo hardware che ci consente di fare il mapping e di associare l'indirizzo virtuale a quello fisico in execution-time
- Nello schema MMU, il valore nel registro di rilocazione viene aggiunto a ogni indirizzo generato da un processo utente nel momento in cui viene inviato alla memoria
- Il programma utente si occupa di indirizzi logici; non vede mai i reali indirizzi fisici

PCB (Program Counter Block)

- Tiene traccia del contesto
- Serve a ripristinare e salvare il contesto se il programma in esecuzione viene interrotto o fatto ripartire
- Usato dalla CPU per determinare il prossimo processo da eseguire seguendo la schedulazione
- Fa parte della MMU

PAGE TABLE

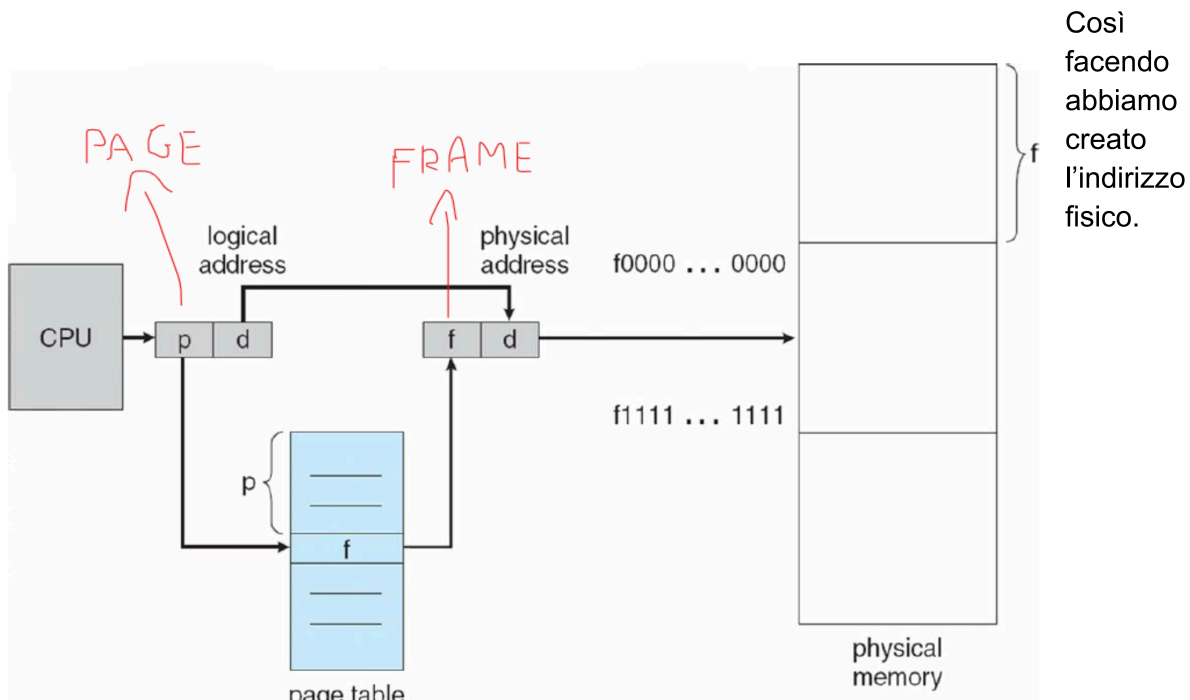
La page table è una tabella all'interno della RAM che la MMU utilizza per trasformare gli indirizzi logici in indirizzi fisici.
LA PAGE TABLE SI DIVIDE IN ENTRY. LE ENTRY PUNTANO I FRAME
GRANDEZZA FRAME == GRANDEZZA PAGE

Siccome la CPU produce degli indirizzi logici, questi bisogna trasformarli in indirizzi fisici prima di metterli in memoria fisica (hdd ad esempio).

L'indirizzo generato dalla CPU è suddiviso in:

- **Page Number (p)**: utilizzato come indice nella Page Table che contiene l'indirizzo base di ciascuna pagina nella memoria fisica
- **Page Offset(d):** combinato con l'indirizzo di base per definire l'indirizzo di memoria fisica che viene inviato all'unità di memoria

Per farlo l'idea è di passare in RAM, leggere la PAGE TABLE, ottenere l'indirizzo del FRAME corrispondente alla PAGE e aggiungere l'OFFSET.



Così facendo abbiamo creato l'indirizzo fisico.

PROBLEMA
La page table ha grandezze considerevoli, non sarebbe quindi sufficiente andare a leggere la page table ogni volta. Per questo usando il PRINCIPIO DI LOCALITA' si va a creare la TLB

TRANSLATION LOCATE BUFFER (TLB)

TLB ovvero una memoria cache particolare che carica solamente gli indirizzi PAGE interessati seguendo il principio di località. Essendo molto piccola è molto più veloce da accedervi e leggerla, risparmiando così tempo prezioso accedendo in RAM.

Tuttavia non contenendo tutta la PAGE TABLE, bisogna scendere a patti che ogni tanto ci sarà un miss. In quel caso bisogna accedere alla memoria centrale.
Tuttavia la HIT-RATIO della TLB è molto alto (circa 80~90%)

VANTAGGI PAGE TABLE

Un vantaggio della page table sono le pagine condivise. Infatti se il codice è rientrante può essere condiviso. questo porta alla creazione di page table e TLB relativamente grandi che però vengono scritte solamente una volta in una memoria condivisa.

LIVELLI PAGE TABLE

Siccome in ambienti condivisi posso avere tabelle molto grandi verranno spezzettate e caricate in memoria un pezzo alla volta.
Ad esempio se avessimo indirizzi a 64 bit avremmo 52 bit dedicati al frame, quindi una page table con 2^52 entrate. Fortunatamente il principio di località è molto forte quindi posso pensare che parte della memoria dedicata alla page table sia inutile al momento (ovvero inutilizzata). Quindi spezzo la page table in blocchi e quelli "lontani" dal principio di località non li carico

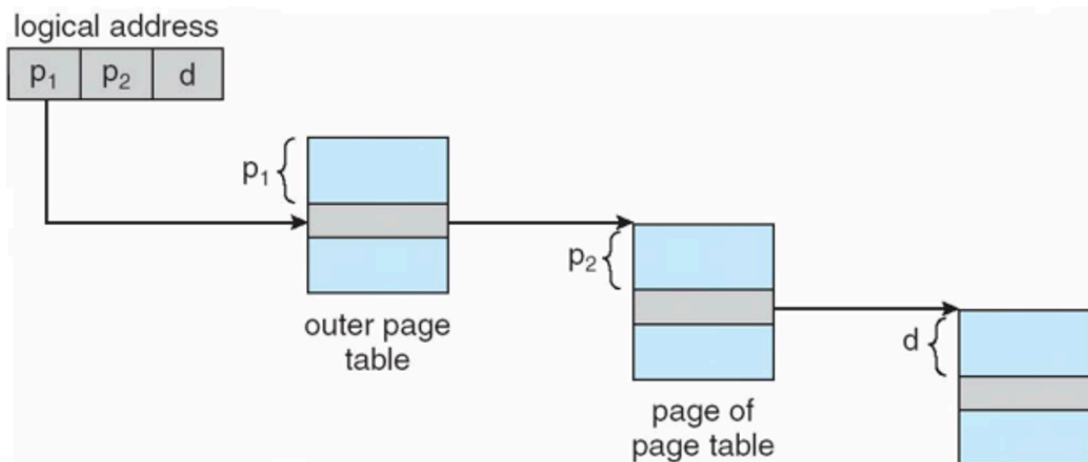
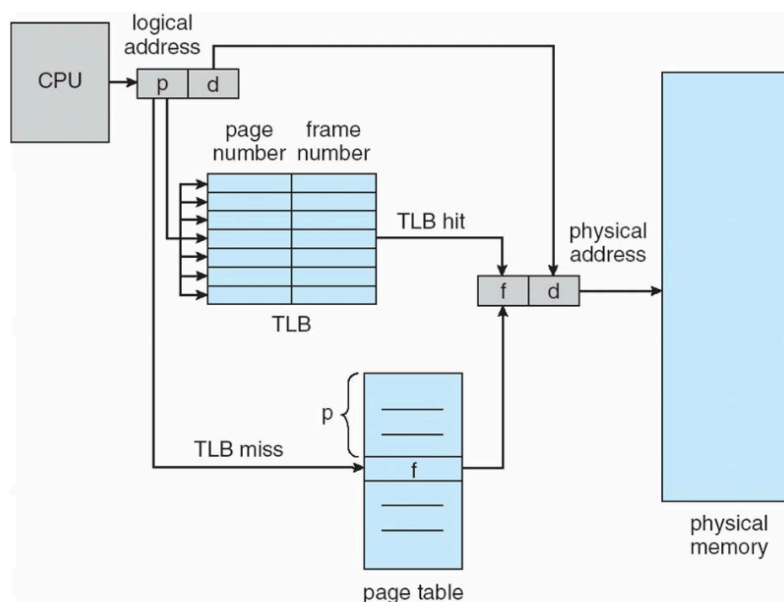


Figura 2. Esempio di traduzione di indirizzi in una page table a due livelli.



Come calcoliamo quindi il tempo di accesso alla memoria?

Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

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

Problema doppio accesso alla RAM

- TLB
- Effective access time
- Memory protection(valid/invalid bit)

X ESERCIZI

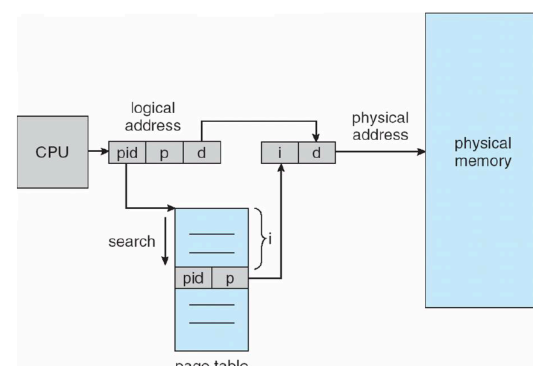
numero entry PT == numero pages
1 MB == 2^20 B
1B == 1024 b
ind. fisico = frame + offset
se n== bit offset → 2^ n B è taglia frame == taglia page

INVERTED PAGE TABLE

Nei processi moderni la dimensione della Page Table è molto grande.
L'idea è di invertire un po' gli schemi:

- Un'unica P.T. per tutta la RAM → non c'è page table per ogni processo
- La ricerca in una tabella così grande diventa Improponibile
- Usando la *Memoria Associativa* riusciamo a mitigare questo problema di ricerca → ma non è sufficiente perché se fallisce la ricerca diventa troppo pesante.

Inverted Page Table Architecture



Si usa quindi **L'HASHING:**

HASHTABLE: implementazione di una hash function, una funzione, che nel caso della P.T. prende come argomento la coppia: **1) pid (identificativo processo)**
2) p (numero pagina processo)
e restituisce la entry i che mi interessa → *hash(pid,p) = i*

- Spesso dà problemi: è difficile trovare una hash function efficiente e che sia iniettiva (*dà risultati diversi con input diversi*)

Iniettiva: hash(pid,p) = i , hash(pid',p') = i'

La funzione hash non è perfettamente iniettiva, ci potrebbero essere conflitti e casi in cui due input diversi danno lo stesso risultato.

- ***La cosa importante da sapere è che esistono tecniche combinate alla Memoria Associativa per rendere efficiente il problema della ricerca***
- Ora è rara da vedere, è stata un po' abbandonata
- Lo schema della paginazione condivisa non si adatta bene alla Inverted Page Table

MEMORIA VIRTUALE

Raffinamento dello schema di paginazione ma non è parte della paginazione stessa.
Si può implementare lo schema della paginazione in memoria centrale senza avere la virtuale.
Computer che gestiscono sistemi critici in cui il tempo di risposta è importante, non implementano la Memoria Virtuale → Comporta dei tempi di Overhead

- **Vantaggio**: poter eseguire un processo avendo a disposizione e caricato in memoria centrale solamente un **sottoinsieme** delle sue pagine, gli spazi di indirizzo logico possono essere **più grandi degli spazi di indirizzo fisico**
- **Possiamo eseguire applicativi che normalmente hanno una taglia superiore a quella della Memoria Centrale**

Abbiamo un aumento del livello di Multiprogrammazione, potendo tenere solo parte dei processi per poterli eseguire, si può pensare di far andare molti più processi.

Demand Paging

Idea di portare una pagina in memoria solo quando è necessaria.
Se una pagina è necessaria si controlla il **Bit di validità nella Page Table** per vedere se è già in memoria o se bisogna portarla.

- With each page table entry a valid–invalid bit is associated (v ⇒ in-memory, i ⇒ not-in-memory)
- Initially valid–invalid bit is set to i on all entries
- Example of a page table snapshot:

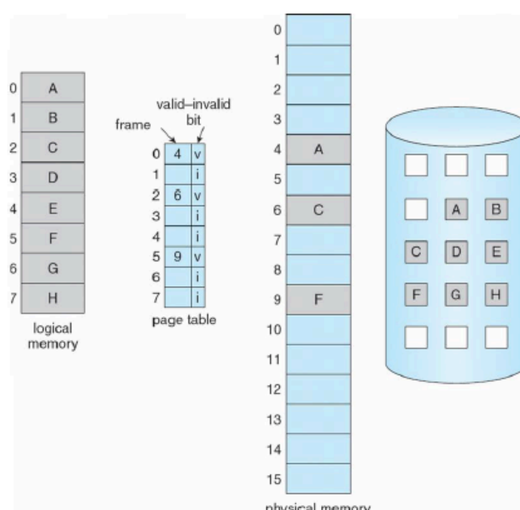
Frame #	valid-invalid bit
	v
	v
	v
	v
	i
....	
	i
	i

page table

- During address translation, if valid–invalid bit in page table entry is I ⇒ page fault

VALID / INVALID BIT

Page Table When Some Pages Are Not in Main Memory



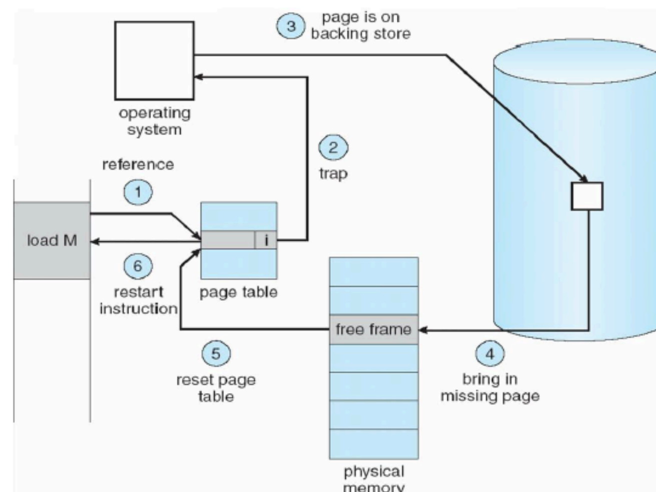
SWAP IN: dalla memoria virtuale alla centrale
SWAP OUT: dalla memoria centrale alla virtuale

Page Fault

Generato da un processo che cerca di accedere ad una pagina che in quel momento non è presente in Memoria Centrale.

Schema funzionamento Page Fault →

Steps in Handling a Page Fault



1) Reference: indirizzo logico che fa riferimento alla pagina assente nella P.T., con bit di validità **invalido**
2) Trap: Interrupt il SO che deve gestire
3) Individua la pagina e la recupera dalla virtuale(facilmente tramite il PCB)
4) Portare in memoria centrale, trovando un frame libero(se non si trova bisogna decidere **Pagina Vittima**)
5) Reset Page Table
6) Restart Instruction: Problema, da dove lo faccio ripartire? Potrei aver cambiato dei dati o tolto alcuni → sarebbe difficoltoso mantenere una "fotografia" della situazione precedente. Soluzione: Tenere traccia solo del cambiamento dei dati

PAGE FAULT RATE → 0<= p <= 1

- Se p = 0 → no page faults
- se p = 1 → ogni reference è un page fault

- Effective Access Time (EAT)

EAT = (1 – *p*) x memory access
+ *p* (page fault overhead
+ swap page out
+ swap page in
+ restart overhead
)

(+PESANTE)

N.B. Il page fault deve essere molto raro altrimenti non ha senso usare la memoria virtuale

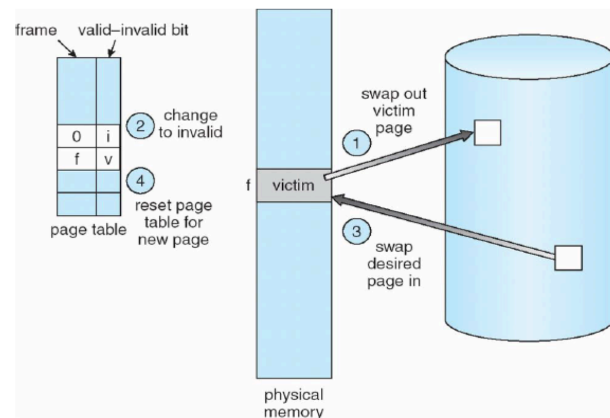
- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p x 200 + p x 8,000,000
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!

Page Replacement (Decisione pagina vittima)

Prevenire la sovra allocazione della memoria modificando la routine del servizio page-fault per includere il Page Replacement.

- **Utilizza il bit di modifica** (*dirty*) per ridurre il sovraccarico dei trasferimenti di pagina→ solo le pagine modificate vengono scritte su disco.
- Il Page Replacement completa la separazione tra memoria logica e memoria fisica → è possibile fornire una memoria virtuale di grandi dimensioni su una più piccola memoria fisica.

Page Replacement



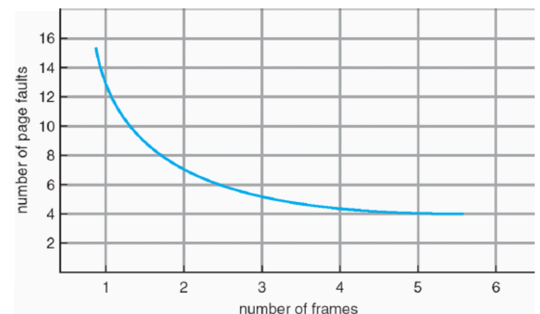
BASIC PAGE REPLACEMENT

1- Trovare la posizione della pagina desiderata su Memoria Secondaria
2- Trovare un frame libero (se c'è lo si usa, se non c'è → Algoritmo Page Replacement)
3- Mettere la pagina desiderata nel nuovo frame libero, aggiornare Page Table e Frame Table
4- Fare ripartire il processo

ALGORITMI PAGE REPLACEMENT

Si vuole il più basso page-fault rate. Esiste un numero di frame ottimale per avere meno Page Fault possibili.

Graph of Page Faults Versus The Number of Frames



FIFO Algorithm

Considero l'ordine in cui sono state caricate le pagine in Memoria Centrale.
Scelgo come Pagina Vittima quella caricata meno recentemente

Optimal Algorithm

Algoritmo che sceglie come vittima la pagina che **sarà usata più in là nel tempo → Si guarda al futuro!!**
Problema: non conosciamo il futuro
(Analogia con SJF nello Scheduling CPU)

Least Recently Used Algorithm (LRU)

Si utilizza il principio di località dei processi → si guarda all'immediato passato

- Pagina vittima: quella usata meno recentemente

L'implementazione è fattibile, bisogna valutare eventuali tempi di Overhead:
Counter Implementation:

- Si associa ad ogni pagina un counter che mi dice quando è stata usata l'ultima volta e si utilizza questo per capire la pagina usata meno recentemente.

Problema: Ogni volta che usiamo una pagina bisogna aggiornare il counter e ogni volta bisogna confrontare i valori del counter→ diventa pesante anche se comunque il page replacement è un evento insolito

Volendo evitare i Counter:
- Tenere uno stack/lista che rappresenta l'ordine temporale di uso delle pagine
- Devo poter percorrere la lista nei 2 sensi per modificare anche gli elementi in mezzo
- E' meno efficiente perchè non so esattamente quando è stata usata l'ultima volta la pagina (so solo quale è stata usata per ultima)
- La lista ha un costo bassissimo → potrebbe costare il mantenimento di questa lista

Conclusione: sapere che in ognuno di questi momenti, devo fare operazione software per gestire i puntatori → diventa pesante

Approssimazione LRU Algorithm (2nd Chance)

COSA SI FA IN PRATICA:
Ci porta alle politiche implementate nei sistemi moderni

1) **Politica reference bit:** si associa alle nostre pagine un **reference bit** che indica se la pagina è stata usata nell'ultimo intervallo di tempo → se non è stata usata si può selezionare come pagina vittima. (si potrebbe migliorare con altri due bit 1- intervallo attuale 2- intervallo precedente)

2) **Politica Second chance**(Implementabile ma è più usato quello prima)
   L'idea è di unire il reference bit con il FIFO
   Abbiamo un solo bit:
   - Controllo il bit, se è 1 dò alla pagina una seconda chance mettendo a 0 e passando oltre finché non trovo uno 0 → Se fossero tutti 1 farei il giro tornando alla prima
   - Ho una probabilità molto alta di lasciare in memoria una pagina usata frequentemente

ALGORITMI DI FRAME ALLOCATION

Capire quanti frame allocare ad ogni processo per capire quanto si può caricare in memoria centrale per farlo andare.
Capire quanto deve essere grande il sottoinsieme (in RAM) che rappresenta tutto il processo.
Esiste un numero di pagine MINIMO per ogni processo
I due maggiori schemi di Allocazione:

FIXED ALLOCATION

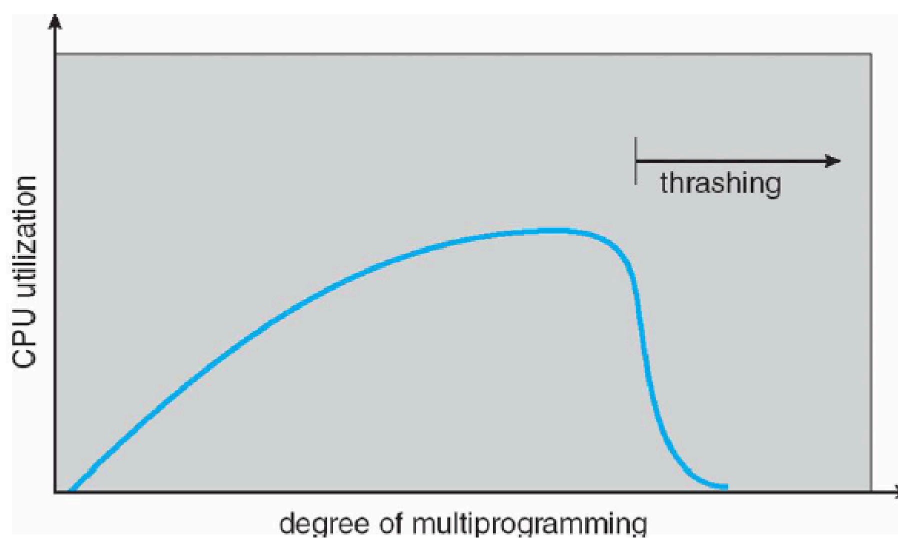
Equal Allocation: diamo a tutti i processi lo stesso numero di frame
Proportional Allocation: Allocare i frame in maniera proporzionale alla taglia del processo

PRIORITY ALLOCATION

Usare una Allocazione proporzionale usando la **priorità** al posto della **taglia.**
L’algoritmo di Allocation interagisce con quello di Page Replacement.
Global Replacement: Un processo sceglie il frame da sostituire dalla lista di TUTTI i frame
Local Replacement: Ogni processo sceglie solo dal proprio set di frame allocati

TRASHING (Problema della Memoria Virtuale)

Si arriva al Trashing quando il SO esagera con il livello di multiprogrammazione → Ci saranno molti più Page Fault

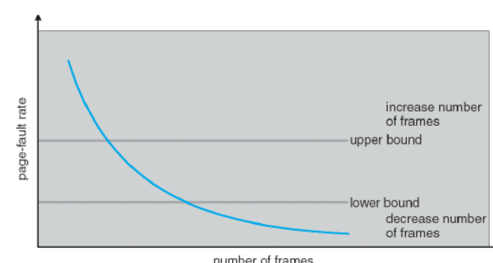


Cosa Succede?
1- CPU viene usata poco
2- SO pensa ci sia bisogno di incrementare la multiprogrammazione
3- Un altro processo viene aggiunto
4- Bisogna trovare frame liberi
5- Togliere frame ai processi in RAM
6- Questi processi stanno già producendo Page Fault → ne produrranno di più

Come fare allora?
SO può tenere traccia del Working set (Insieme delle pagine di cui ha bisogno → principio di località) dei singoli processi.
Si può cercare di monitorare il numero di Page Fault che un processo produce nel tempo.
Obiettivo: fornire ad ogni processo un insieme di frame che corrisponde, più o meno, alla **Numerosità** del Working Set in quel momento.
Upper Bound: Aumentare il numero di frame
Lower Bound: diminuire il numero di frame

Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

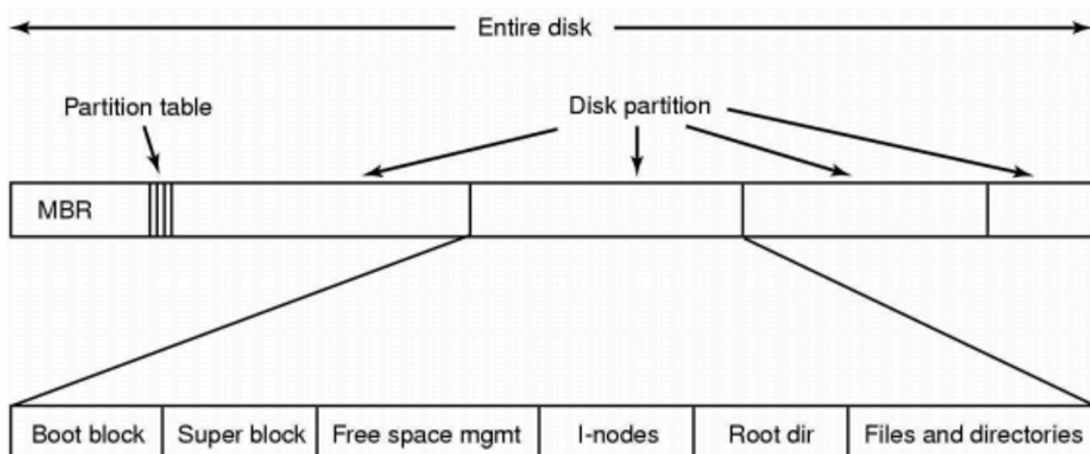


C'è un po' di Overhead per monitorare i page fault ma sempre meglio del Trashing!

FILE SYSTEMS

quando si parla di file system si pensa alla struttura ad albero delle directory.
Infatti attraverso delle system call posso non solo manipolare il file ma anche la sua posizione (ovvero la directory).
I file system si trovano in memoria secondaria e sono una serie in continuo aggiornamento di file. E' necessario dividere la memoria in partizioni, per un accesso contingentato a file importanti, come:

- MBR
- Boot Black
- Partition table
- Root-Dir
- Space Management



TIPI DI ACCESSO

quando un processo lavora su di un file ne modifica il RECORD LOGICO: ovvero un blocco di dati relativi ad un blocco logico la cui suddivisione segue una relazione logica. Il record logico contiene gli attributi (ovvero data e ora di creazione, data e ora di u
Vi sono quindi due tipi di accedere ad un file

SEQUENZIALE

SO fornisce accesso ai file in maniera arbitraria. Metodo utilizzato per l'accesso in memoria secondaria ha tempi di accesso alti rispetto ad altri metodi. Il tipo di accesso è "entro e vado". siccome i blocchi di memoria sono uno di seguito all'altro entro dal primo e seguo in successione.

DIRETTO

L'accesso diretto permette di avere tempi di ingresso minori rispetto al sequenziale ma a costo di una complessità maggiore. Il file è quindi un insieme numerato non ordinato di record logici a cui è possibile accedervi direttamente sapendo il numero del record da interrogare

PROTEZIONE

La protezione nei file system è di relativa importanza. Infatti trattando memorie volatili come la memoria principale devo garantire la FLESSIBILITÀ e GARANZIA. ovvero devo specificare diritti diversi a diverse tipologie di utenti come: owner, groups, public
Ho quindi 2 livelli di flessibilità su 3 bit: R(read) W (write) X(execute)

			RWX
a) owner access	7	⇒	1 1 1
			RWX
b) groups access	6	⇒	1 1 0
			RWX
c) public access	1	⇒	0 0 1

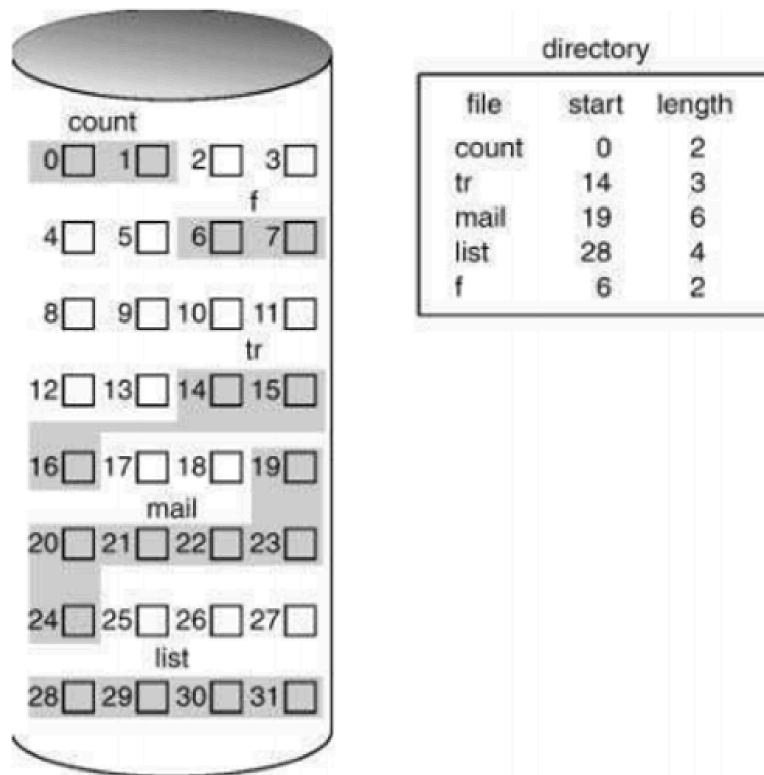
ALLOCAZIONE SU FILE

ALLOCAZIONE CONTIGUA

Blocchi allocato uni di seguito all'altro (in maniera contigua) se un file ha bisogno di essere memorizzato in più blocchi. Il SO terrà traccia di quale directory mi trovo e della directory del file e della lunghezza del file (quindi quanti blocchi occupa)
Soluzione più semplice ma con diversi problemi:

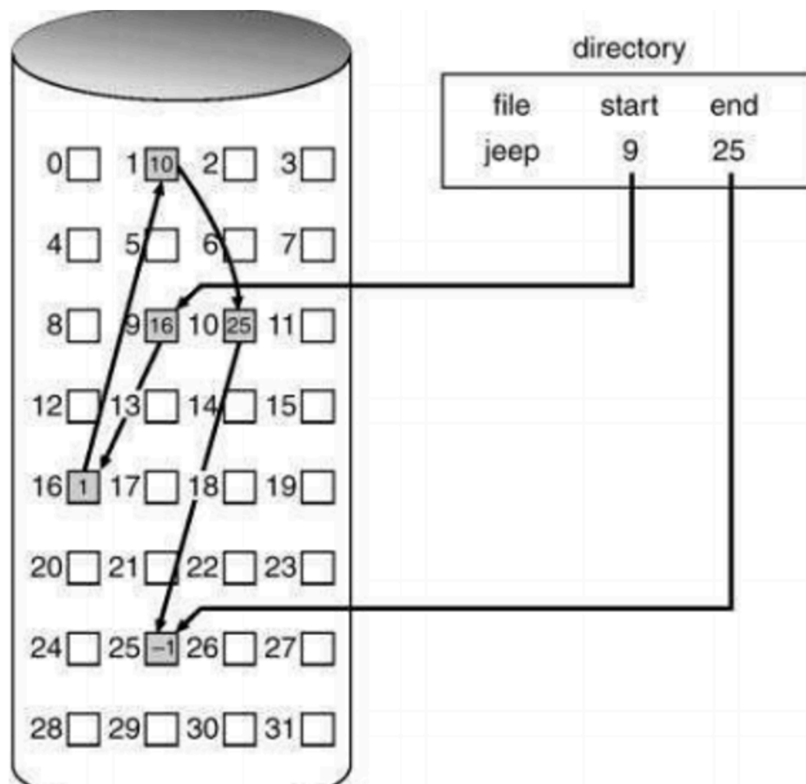
- frammentazione esterna
- necessita compattazione in caso di cancellazione, ovvero sposto i blocchi successivi
- Soggetto a variabilità del file, ovvero se il file cresce devo spostare tutti i blocchi successivi

Accesso al blocco: n/m= k(intero) dove n= record logico che mi interessa m=grandezza blocchi k= blocco che mi interessa



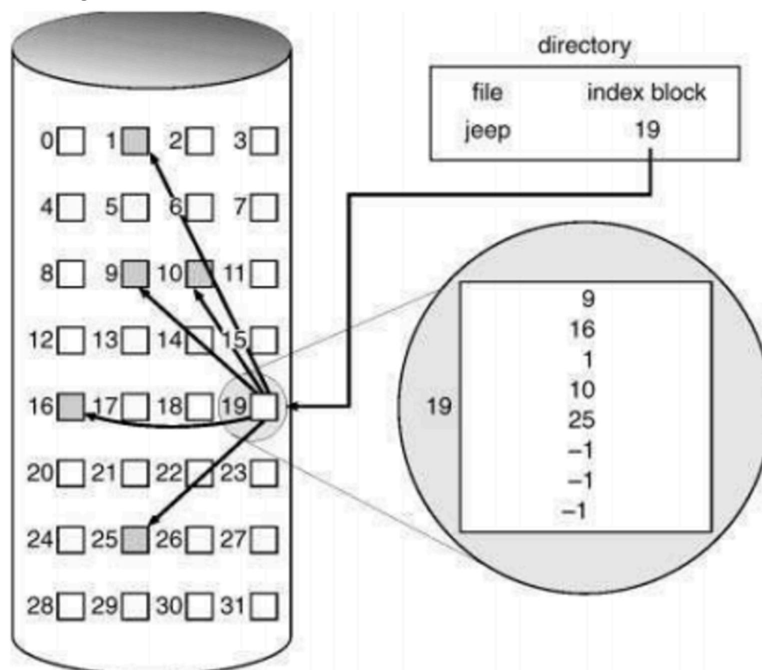
ALLOCAZIONE CONCATENATA (LINKED)

Allocazione su blocchi non contigui ma collegati (lista)
collegamento singolo / double
All’interno del blocco abbiamo un (-1) che indica la fine del file.
Presenza un metodo di accesso sequenziale, ovvero entro nel primo elemento della lista e scorro fino in fondo (ogni scorrimento corrisponde ad 1 entrata)
Accesso ai record logici banale ma poco efficiente: più il blocco interessato è lontano più ingressi in memoria devo fare
Non ho il problema della frammentazione esterna



ALLOCAZIONE INDICIZZATA (INDEXED)

Come la memoria centrale ho una tabella che mi indica dove sono i blocchi in memoria. Dal momento che ottengo la tabella faccio un accesso diretto al blocco interessato



PROBLEMA JOURNALING

Anche detto problema della eliminazione.
HOW TO ELIMINARE:

1. Eliminare file F dalla directory
2. Eliminare riferimenti ad F della table content
3. Liberare blocchi contenuti da F

Sembra facile ma sono una serie di operazioni non atomiche.
Siccome si parla di memorie si parla di volatilità più o meno alta: nel caso della memoria centrale ad alta volatilità il rischio crash è elevato.
Cosa succede in caso di crash? Rischiamo di aver eseguito i punti 1 e 2 ma non cancellare de facto la memoria relativa ad F e non abbiamo più modo alcuno di recuperare quelle informazioni!

HOW TO (ACTUALLY) ELIMINARE:

3. Liberare blocchi contenuti da F
1. Eliminare file F dalla directory
2. Eliminare riferimenti ad F della table content

Si capisce quindi che una parte delle operazioni non atomiche devono essere di controllo, ovvero servono per assicurarsi che le operazioni siano effettuate nonostante i failure

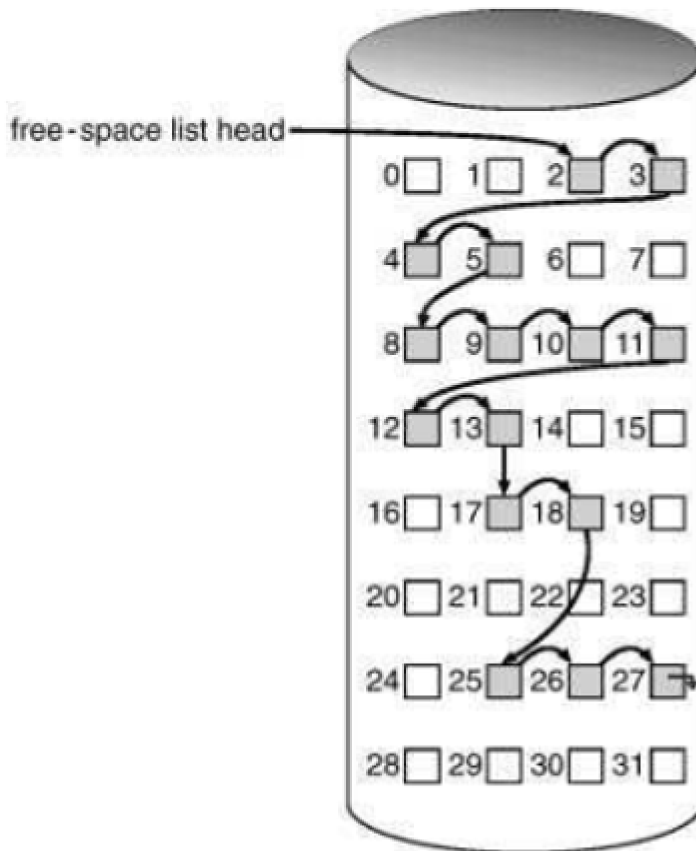
FREE SPACE MANAGEMENT

esistono 2 metodi per gestire i bit liberi:

1. **BIT VECTOR:** vettore fatto da tanti elementi quanti i blocchi in memoria secondaria. ogni bit del vettore può assumere i valori 0 oppure 1 e indicano rispettivamente se quel blocco è occupato oppure libero

$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

2. **LINKED LIST:** ogni elemento della lista è un blocco libero. Questa implementazione è in teoria meno efficiente ma lo scorrimento della lista (quindi l'accesso in memoria) è raro. Di fatti generalmente il sistema si limita ad usare il primo blocco della lista



CRITTOGRAFIA

- Confidenzialità/Autenticazione/Integrità/ Accessibilità e disponibilità
- Crittografia a Chiave Simmetrica
- Crittografia a Chiave Pubblica
- Algoritmo RSA e funzionamento
- Firma Digitale (VANTAGGI/SVANTAGGI)

L'argomento di crittografia è breve e tutto teorico. Ci siamo limitati ad indicare gli argomenti richiesti in sede d'esame. Sono giusto cenni e ben spiegati nelle slide del prof.