

**SO** : gestisce le componenti HW del calcolatore  
figuratevi quindi da 

l'oggetto	processi
memoria	attuale
dischi	
I/O	

la memoria è vista  
recorderà i dati

In modo simile e analogo

Dall'utente verso l'utente: gestisci del software  
**SO**.

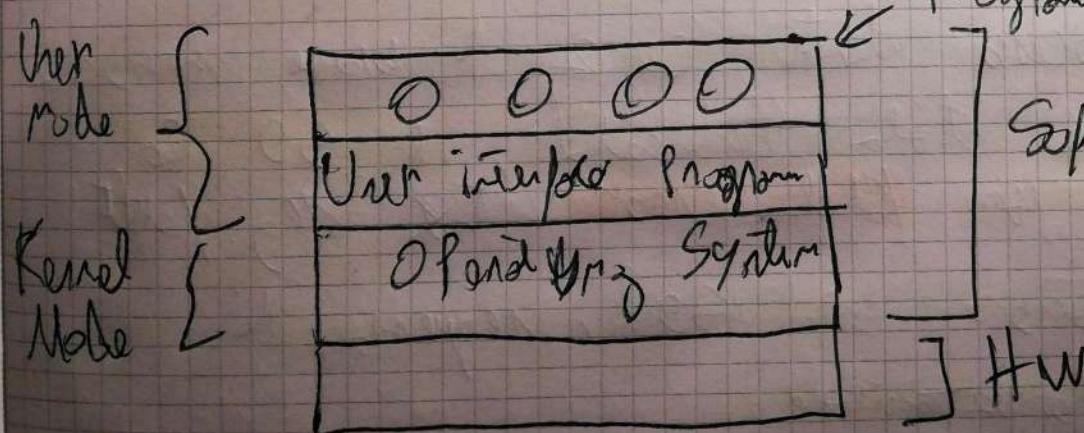
Un SO è un software che può interagire  
direttamente con l'HW e fa da  
intermediario tra HW e user offrendo una  
serie di servizi utili con le applicazioni.  
Chi usa il SO? I processi

Processo: intenzione di esecuzione di un programma

User Interface: componente software che offre  
interfaccia per l'uso dei servizi offerti dal SO

Programma: istruzione

utente



Il software che viene è istruito

## Nobilità & no dell' HW

La CPU si trova direttamente in linea di ciclo.

Fetch - decode - execute.

L'esecuzione può già essere eseguita in linea  
nobilità. Cittadine due:

- Nobilità Kernel
- Nobilità User

Nella prima il software (quello ha tutti i privilegi) e può effettuare qualsiasi sys call senza restrizioni.

Nella seconda il software che sta sotto la CPU ha delle limitazioni (molte istruzioni potrebbero essere delicate)

Ogni processo ha un'area di memoria assegnata ed non intercede le aree degli altri processi tranne in cui la comunicazione tra processi.

Quando un processo tenta di uscire dalla sua area di memoria che non gli è stata assegnata, il SO lo blocca con un **bad error**: **Ex Reg. fault**

## KERNEL: come nel SO, request di istruzione forzata.

Quando è caricato in memoria l'OS non è attivo continuamente e amministra tutti i processi in linea.

In generale ~~tutti~~ il SO è eseguito in modalità Kernel e i processi user in modalità user, però alcune componenti del SO potrebbero lavorare in modalità user. Si fa regolarmente perché queste componenti non ~~hanno~~ ~~hanno~~ necessariamente di tutto i privilegi quindi per funzionare si lavora in questa modalità.

Un solo utente che esegue un SO ha concedere i minimi privilegi necessari per ogni processo.

N.B.: Utente interagisce con OS che usa il Kernel e amministra tutti i processi, mentre l'HW

### Macchina virtuale

Si può vedere il SO come un'astrazione dell'HW. I servizi offerti sono astrattivi.

L'astrazione è mai chiara per gestire le componenti, presenti in molti semplici qualcosa di già esistente (molto costoso)

L'HW esiste, ma è complessa essere, usiamo il SO per offrire una bella interfaccia per programmatori (questo è un'astrazione).

### Esempio astrazione: FILE

Un dispositivo meccanico è composto da una sequenza di blocchi in cui possono essere scritte o lette informazioni.

Ma da un punto di vista logico, i file non esistono solo astrazioni.

Dall' estensione del File ci si può ricavare a quale del File risponderà.

L' HW mom è cosa in un file, se solo si sapeva e legge (in modo meccanico)

**Attribuzione:** Servizio offerto dal SO implementato tramite software il quale si appoggia ad un insieme composto software o Hardware.

## SO come gestore di risorse

Un moderno SO gestisce più risorse in concorrenza fra loro.

Necessita di associazione arbitrata e controllata

Risorse (via HW da SW)

La gestione è fatta con MULTIPLEXING

Il multiplexing può essere di due tipi

- SPAZIALE
- TEMPORALE

Risorse condivise temporalmente

E' lo a turno per works

Il SO si occupa di gestire gestione, Ex: Stargante

Risorse condivise spazialmente

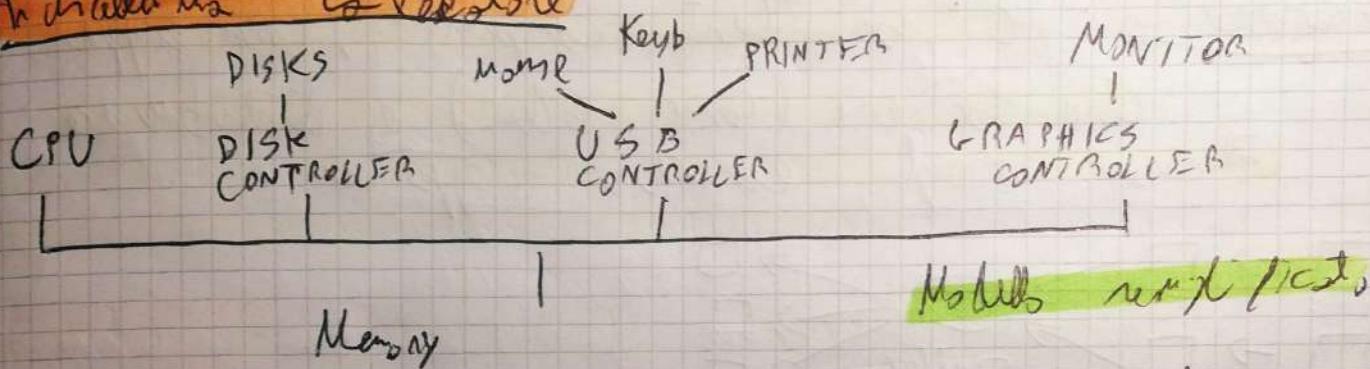
Ex: Disco fisso → Allocazione di una "zona" sul disco e gestione traccia di cui ha quote bloccate

Centrale Logica CPU: dei processi fanno a turno  
per poterla usare

Altro esempio esecuzione: Oggi ~~CPU~~ Processore "freel" come no la CPU di base interamente dedicata, ma in realtà non è così, oppure se ha opzioni di timer.

• Storia SO

• Architettura Calcolatore



Componenti legati direttamente alla memoria sono le periferiche

Processione

Basta un codice: Fetch - Decod - execute

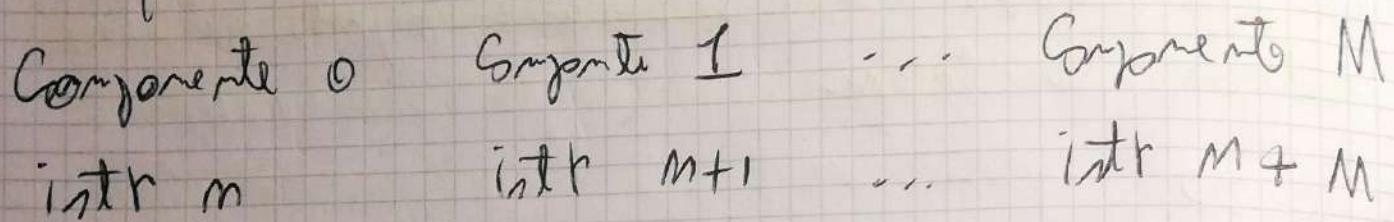
Ecco spieghi i registri: bit di corso del programma

• Program Counter = ha inizio con il primo indirizzo  
• Stack pointer = punta allo zero dello stack, che  
ha il frame di ogni procedura non  
finalizzata

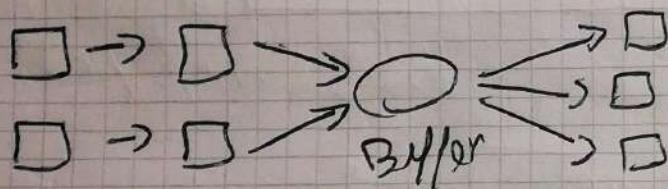
• Program Status Word = iniziale bit flag

Esempio per i progettatori già avviate per controllo delle sequenze delle istruzioni

• Pipeline: elaborazione in varie sezioni, ogni sezione fa una cosa in una linea parallela



• Superpipeline: ci sono tante componenti, un buffer che contiene (come cache) e distribuisce istruzioni negli esecutori opportuni

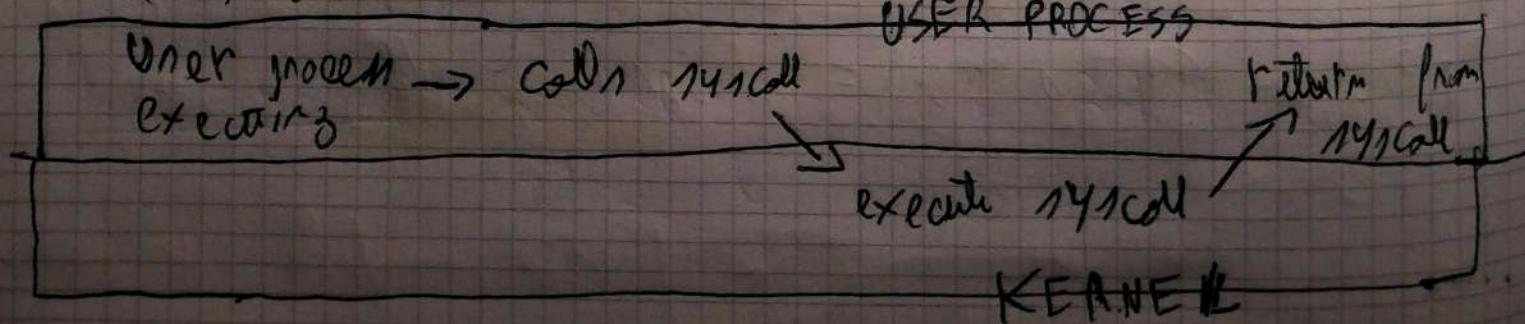


## Necessità di esecuzione

I processi stante hanno bisogno di aiuto del SO in Kernel mode in certe istruzioni, le "necessità" si ricava quando la chiamata a interruzione, che entra nel Kernel e richiede il SO.

L'istruzione che switcha da User mode a Kernel è detta TBAP.

Quando il lavoro del SO è finito si torna a user mode



L'instruzione TRAP pone in Kernel Mode, per un motivo  
a una routine nel Kernel.

Lo TRAP è una call speciale, in alcun sintomi  
ha un parametro per specificare cosa deve fare

Dentro organizzazione è comunque ricca perché  
"pre-programmata", la syscall indica quanto  
routine pre-programmate dal progettista del SO

Molti trap sono ottimizzati dall'HW

Ex: tentativo di divisione per 0 → TRAP → SO prende  
in mano la situazione.

A volte nessuno di Internet.

Il Kernel a volte li fornisce per lavorare intenzionalmente.

## CPU con multithreading ("Vero 2 CPU")

Si tiene all'interno della CPU lo stato di  
due thread, per ottimizzazione.

Per cosa può essere utile?

La CPU a volte aspetta varie entrate informazioni  
dalla memoria, invece di non fare nulla  
commette null'altro thread e lavora su di esso.  
"Ammultano" il tempo morto.

"E' come se avessi 2 CPU" Il SO ne tiene  
comunque conto.

## Multiplazione: Abbiamo tanti CHIP-CPU

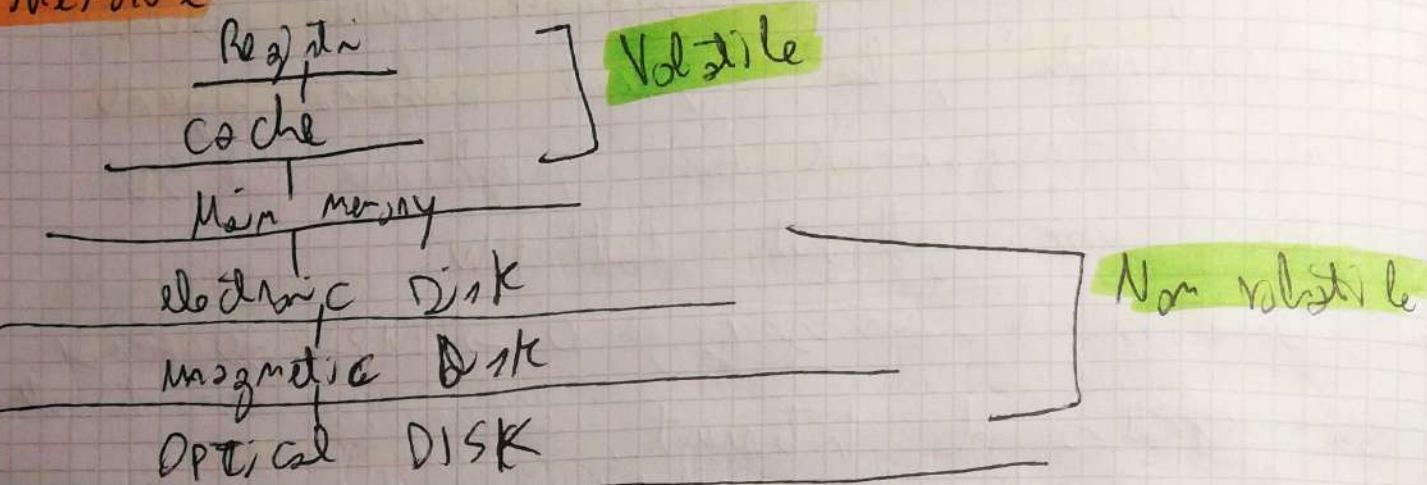
Vantaggi: più efficienza

Multi-core: la CHIP-CPU è diviso in vari CORE  
ognuno opera in grado di fare elaborazione

Doti infinitamente dagli altri CORE

**GPU**: processore core migliaia di micro logic core  
utilizzati per eseguire tante piccole operazioni  
Ex: 2 CPU, ognuna 2 thread  $\rightarrow$  i "rabbis" 5 CPU

## Memorie



Register: veloci come la CPU, accelerano non comporta ritardi, l'accesso da potuto dai programmi

Cache: grande molto dall'HW, i suoi cicli sono da qualche accesso riduce.

E' organizzato a livelli

L1: all'interno della CPU, 16 KB

L2: già grande (già lenta)

Alcuni sistemi mettono una L2 grande per ogni

CPU/Core, opp altri sistemi mettono una L2 grande comune.

In alcuni sistemi c'è presente la L3

Main memory: generalmente della RAM  
Centinaia di megabyte

Trovano ormai sempre anche la ROM (non volatile) usata in alcuni pc per l'avvio o altro.

Disk: Per mantenere dati in memoria non volatile servono i disk, che sono le memorie più grande e dense.

## Dispositivi I/O

Infiltrano due componenti:

- Controller; gestisce interfaccia per l'usr, invia al driver che controlla fisicamente il dispositivo.
- Dispositivo stesso; l'interfaccia è elementare ma complicata da usare

Ex: Disco SATA

C'è bisogno di un software che maneggi ogni controller e i suoi driver

Il driver interagisce col dispositivo tramite il controller, lo fa con le porte I/O

Il driver deve essere inserito nel SO, sarà già in modalità Kernel

L'interazione avviene così:

- Si usa int IN/OUT ①
- Si fa una richiesta in memoria
- ① Driver eseguito in Kernel Mode, se ② riguarda

E può gestire Input Output in 3 modi diversi

1) La CPU segue il bitset che ora I/O  
e si calcola a un ciclo ogni e contiene in  
cui si interroga continuamente il dispositivo per  
vedere se l'IO è finito.

Fatto l'I/O, se c'è bisogno il bitset mette  
i biti nel punto opportuno e finisce l'esecuzione  
il metodo è detto busy waiting. Troppo noioso.

2) Usano un interruttore che sposta l'interruttore.  
Il controller invia una notifica alla CPU che  
mentre esegue qualcosa fa altro.

La notifica serve a regolare varie cose,  
es: se ora i biti sono pronti e la CPU  
può tornare a lavorare col dispositivo.

E basta usare anche quelli da i biti

~~return false~~ gestione degli interrupt che controlla  
semplicemente la cosa degli interrupt

3) DMA : direct memory access

Già una componente hardware che il chip DMA  
che serve a controllare il flusso di bit tra  
RAM e dispositivo senza usare in modo costante  
la CPU.

All'inizio del process communge la CPU impostare le  
direttive sul "come lavorare"

## Catalogo SO

- Per smartphone / server:  
Molti SO sono  
multi-utente  
per gestire molti processi I/O,  
che operano su server.
- Per PC;  
supporta multi-programmazione
- Per tablet / smartphone;  
Android / iOS, hanno le app.
- Per sistemi integrati (embedded);  
hardware, software unitario, in ROM
- Real time; devono creare efficienza in gestione di eventi, con modi per valori in banca dati.  
Timing assicurabile

## STRUTTURA A SO

### MOLTIOSO

#### 1) Senza supporto f/w.

Non c'è ambizionamento Kernel e User mode.  
Da molti problemi, non è più considerato  
multi-programmato, nessuna interattività.

#### 2) Supporto HW per mode User - Kernel

Chico Kernel su tutto tutto

Ogni componente ha un'interfaccia  
e può essere gestibile

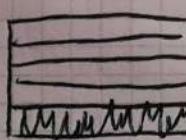
## Struttura a livelli / strati

Visione verso gerarchia a livelli  
 Ogni livello è controllato in basse a qualsiasi  
degli suoi poteri per i suoi servizi  
offerti. Autonomia in ciascuno a una serie d'  
autonomie.

Strato 0 : HW

Strato sopra : SW utilizzati in utile strati  
 Questo strato offre maggiore protezione

"Chi usa il servizio offerto dal livello sovrastante  
 non avrà preoccupazioni di come funziona"



SW  
HW

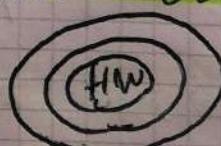
E' una buona strategia per la  
continuazione del primo livello  
concreto.

## Variante di arredi concentrici (MULTICS)

La separazione del HW ora è formalizzata

Questa variante garantisce struttura a strati soltanto  
 a funz.-time base in cosa è bene.

Ma non invece nella gerarchia livelli semplici la  
 strutturazione è solo geografica, qui abbiano  
 una affidabilità verso le maglie del primo  
livello frico



Questo non ha 3 layer-alla scatola 3 ridicole.

Il potere che ci riesce molte ridicole a  
costruire completi problem di gestione

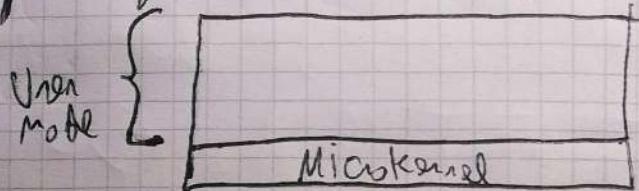
## Microkernel

Si usa un Kernel che i occupa la scheduling, memoria e IPC. Questi componenti che normalmente starebbero nel SO stanno "fuori". Tutto il resto è gestito da moduli o Userspace. La comunicazione fra servizi è fatta per messaggi, che permettono le comunicazioni anche se Kernel.

Queste strutture permette un miglior design e miglior stabilità ~~dato che~~ dato che abbiano tra loro i vari componenti.

Vantaggi: Se un modulo è buggato, molto probabilmente non renderà tamme gli altri moduli o al kernel, per questo è utile la modifica.

## Strutture e moduli

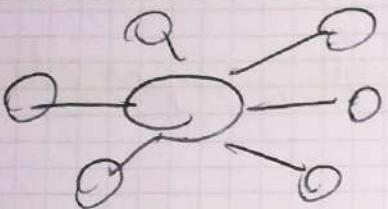


Si applica programmazione OOP al Kernel.

I vari moduli implementano un protocollo specifico. Il kernel principale ha funzionalità già integrate e i moduli sono considerati binari.

Abbiamo un design getto, efficiente in cui qualsiasi modulo può già invocare un altro tramite la struttura degli User Space i messaggi.

La struttura ha un nucleo con "stanno" i vari moduli.



Mobilità Kernel: molti dati  
necessari e dati mobili.  
User: mappa di posizioni  
utente.

Il Kernel NON fa da triste nella comunicazione  
tra moduli.

## Macchine Virtuali

L'elaborazione l'utente ha potuto alla  
creazione delle macchine virtuali, **Virtualizzazione**.

Scopo: usi le già SO, insieme servizi, quelle  
che fanno **simulazioni** e generate altri un SO  
vincisi il concetto di **molti spazi a parità**

Le macchine virtuali sono macchine estese,  
sono una cosa dell'HW che la macchina reale  
di sopra.

La macchina virtuale funziona proprio un SO

NB: La macchina virtuale in un qualche modo,  
direttamente o indirettamente ha un HW reale.

Infatti viene a crearsi in certi casi **Overhead** fatto che deve usare la reale CPU

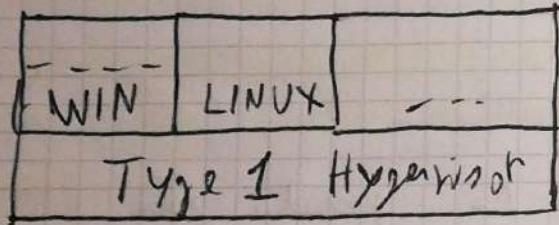
Concetto di Paravirtualizzazione: è una variante già  
affidabile, il SO è adattato per funzionare sulla  
macchina virtuale.

E risparmia Over Head però ci sono già vincoli  
per l'omogeneità tra macchina finca e  
SO sulla virtuale.

Il tutto è gestito dall' Hypervisor , esponente software

### Type 1

Software che già finisce nello HW su cui  
puoi subito installare poi un SO (oggi)



### Type 2

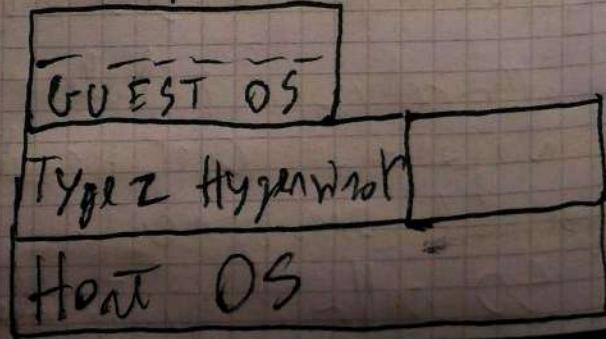
E' un processo che gira su un SO host in User mode.

Alcune sue componenti hanno comunque bisogno del Kernel , per questo i richieste di presenza ai moduli che generano ciò

Quindi :

Il software Hypervisor gira sempre in User mode e a volte deve fare cose che richiedono Kernel Mode , il che è generato da moduli aggiuntivi .

Oggi nei calcolatori moderni ci sono bell supporto HW per gestire questo problema in maniera efficiente



## Processo

Intervalli di esecuzione di un programma

Cosa ha un proprio stato, si eseguo le istruzioni e lo stato evolve.

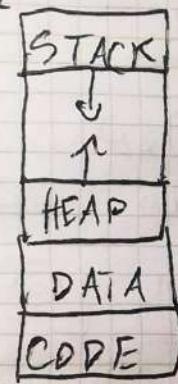
A un processo associamo:

- codice
- dati
- stack

Sposto degli indirizzi  
(quantizazione into cells  
RAM riservate all'utente)

Spazio di  
memorizzazione

Max



Coda regista CPU

File aperti (ancora il  
file pointer)

Trasmette queste  
informazioni  
nella Tabella  
di processi

alarm globale

Processi imponenti

Stack e Heap sono le aree di memoria che crescono, lo stack è memoria statica mentre l'Heap è dinamica.

Crescendo gli indirizzi tocconi

DATA = dati già contenuti all'intante 0, è una area di memoria statica

Il File pointer è una testina che si muove dentro al file

## Tabelle dei Processi

Ci sono Tabelle dei tutti i processi in esecuzione.  
Qui sono contenute le informazioni in file seguenti:  
stato, identificativo e processi impostati.

Lo immaginiamo come una Tabella lineare di processi, a ogni processo è associato il Process Control Block che è la sua "ID".

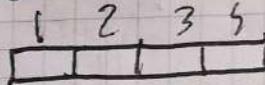
La Tabella è suddivisa in memoria dinamica.

Dentro al record del processo trovo le informazioni in maniera diretta o puntatore alle informazioni.

In pratica il PCB è un punto di accesso a tutto quello che c'è nel processo.

Potremo considerare come identificativo del processo l'indice della Tabella.

L'identificativo è il PID



I vostri mi sono molti programmati per cercare lo pseudo-paralleloismo, dunque si implementa quello che è il concetto di CPU virtuale.

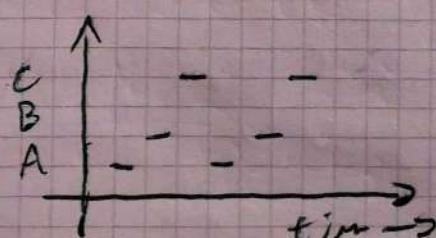
Invece di eseguire i processi in maniera sequenziale facciamo a "turno".

Ma se il processo A e B usano contemporaneamente la CPU come si gestiscono i registri?

Nel PCB c'è appunto la lista dei registri.

E se Seguenti: A → B → C → D ...

Pseudo-Parallelismo



Inglese mentito il concetto di CPU virtuale, ogni processo vede la CPU come se gli fosse interamente dedicata.

• Process switch: Cambiano processo a cui si dedica la CPU

## Creatiōne Processo

Come si crea?

Nella fase di boot/avvio del sistema

Da parte di un altro processo/utente

## Metodi di creazione

• Sboggiamento padre: fork e exec (UNIX)

• Creazione attorno al programma: CreateProcess (WIN)

↳ Processo base: INIT, si occupa della gestione di tutti i processi, è sempre attivo.

Ess è l'antenato di tutti i processi

(In questo albero se un processo è creato è aggiunto come figlio di chi lo ha creato)

E' cre l'albero di processi con radice l'INIT

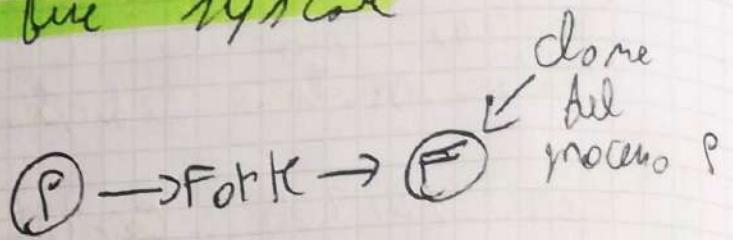
Se termina un processo che aveva figli il processo INIT li eredita.

Perché abbiamo l'albero?

Perché alcuni processi sono strettamente legati al padre, se il padre muore anche loro devono morire

Fork e exec: sono due syscall

Crea nuovo processo,  
essa è richiamata da  
un altro processo, non  
ha parametri  
Avviene una biforcazione



La Fork torna 0 nel  
contenuto del figlio e il  
PID nel contenuto del padre  
per gestire la domenione

if (fork() == 0)

FIGLIO

else

PADRE

In questo modo gestiamo cosa  
deve fare il figlio e cosa il  
padre.

EXEC

è una chiamata a sistema con dei  
parametri exec(c, arg1, arg2 ...)

La exec inizia il comando e ha treli  
argomenti.

La exec rivela l'ambiente di esecuzione/parametro  
di indirizzamento e gli piazza il C

L'exec usa lo spazio di indirizzamento per  
piazzare il codice chiamato C, preso dal DSO,  
e lo esegue con i vari parametri

La exec permette di eseguire un nuovo programma.  
Si può usare la Fork e non la exec facendo  
attenzione alla comunicazione tra processi

La CreateProcess di WIN è già composta da  
una, che ha molti parametri.

## Terminazione del Processo

- Uscita normale : exit (UNIX), EXIT Proc (WIN)

Il processo regrada al SO che vuole auto terminarsi.

Exit status : è un valore che se è 0 regola che l'uscita non ha avuto problemi, se è >0 regola delle anomalie.

### VOLONTARIA

- Uscita in errore

### VOLONTARIA

Il codice che fa terminare giochi se che non più esiste sviluppi

- errore critico

### IN VOLONTARIO

Legge di anomalie di germei, istruzioni illegali o ricerche, divisione per 0.

Alcuni errori critici sono gestibili, altri no.

Il processo potrebbe richiedere di avviare una routine per gestire il problema.

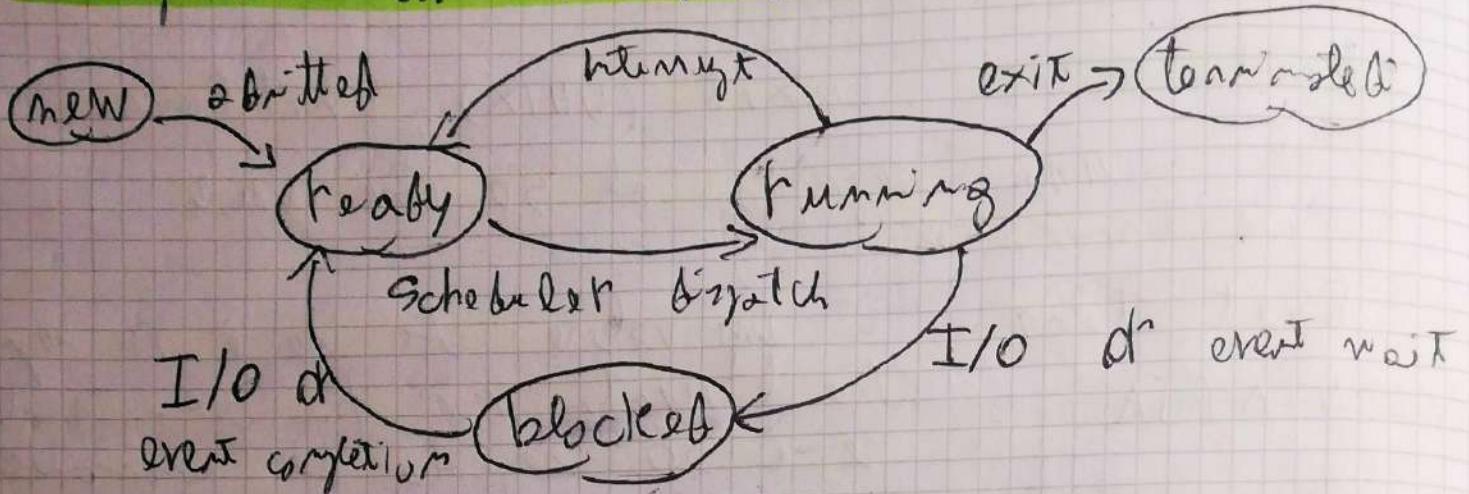
Se molti sono gestibili, questi errori causano chiuse interruzioni del processo

- terminazione da un altro processo

### IN VOLONTARIO

metodi : kill (UNIX), terminate process (WIN)

Il processo come un automa ha uno stato



- **RUNNING** : CPU effettivamente in uso per il processo
- **Ready** : processo eseguibile ma attualmente in sospensione
- **Blocked** : bloccato finché non intrinseca un evento utente

### Transizioni

**RUNNING → Blocked** : il processo si blocca in attesa di input

**RUNNING → Ready** : lo scheduler prende un altro processo

**Ready → Running** : lo scheduler prende questo processo

**Blocked → Ready** : input è ora disponibile

**Schedulatore** : componenti / algoritmo per la gestione dei processi

Dato che ci sono tanti processi in stato Ready creiamo una lista di tali processi

## Tavolla dei processi

Abbiamo in ogni record il PCB che contiene  
informazioni sul processo

- state
- number
- program context
- registers
- memory limits
- open files
- ...

Structure Tables

Process				
0	1	2	3	...
Schedalet				

Gestione interrupt per passaggio di Processo:

- 1) Salva nello stack il PC e la PSW che sono nello stack attuale.
- 2) Si carica nel vettore degli interrupt l'indirizzo della procedura associata.
- 3) Salva i registri e impila un nuovo stack.
- 4) Si esegue la procedura di servizio dell'interrupt.
- 5) Quando finisce si definisce un nuovo come stato processo prosegui.
- 6) Si aggiorna del PCB lo stato del processo.
- 7) Si riprende il processo.

Come struttura fisica per l'organizzazione  
memoria dei processi

### CODE

Un processo può stare in uno dei seguenti stati:

Coda dei processi READY: sottoinsieme delle tabelle dei processi

### Coda dei Disponibili

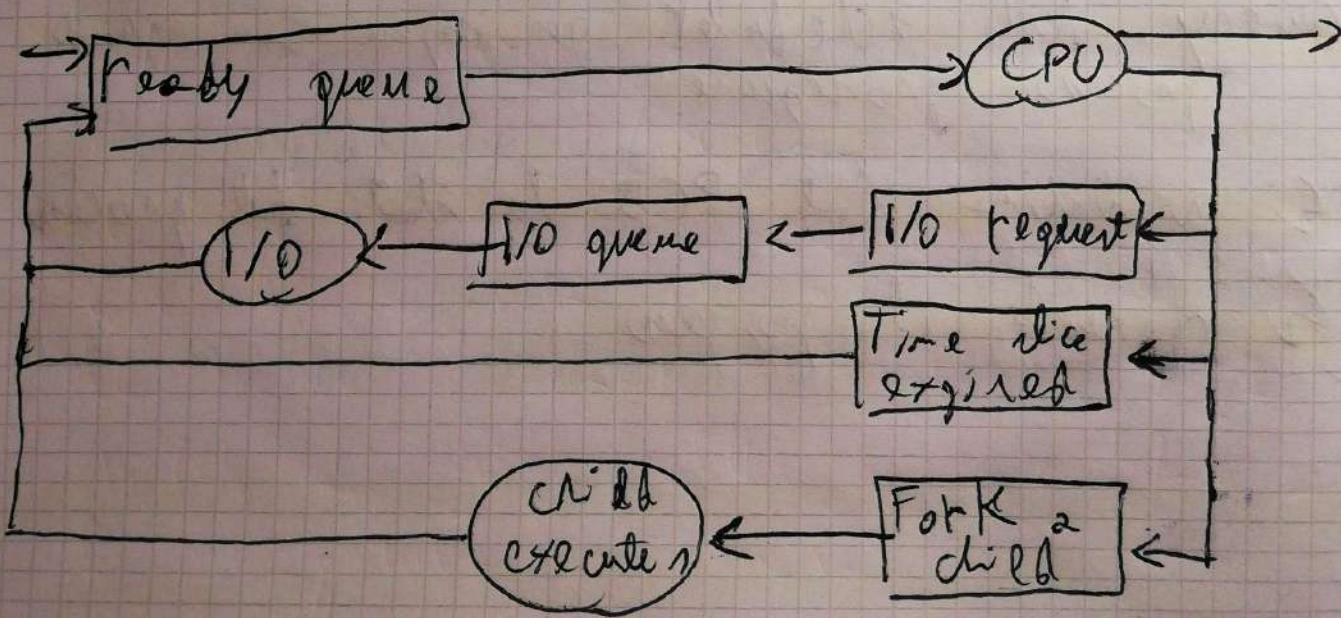
Coda verso la quale procedente struttura dati  
tabelle dei programmi, è implementata con doppio  
LINKING

Coda con dei processi bloccati che devono essere  
messi in disponibili

Abbiamo una coda per ogni controller

Quindi il processo fornito è già predisposto  
nella coda del controller, verrà già eseguito  
quando dovrà essere messo nella ready queue.

ACCODAMENTO (processi in cui avvengono gli  
incrementi e estrazioni dalla  
coda)



## Thread

Evoluzione del modello di processo

Facciamo combiniada e più flussi di esecuzione  
di stessi spazi di indirizzi.

Facendo così diversi flussi in un solo processo

P1 P2 P3  
① ② ③

P      record nico  
      ④ ⑤ ⑥

Quindi ogni processo ha  
molti CPU virtuali

Qui ho già CPU virtuali  
per un solo processo

Puoi essere utile aprire molti file nel vari flussi  
senza uscire di stessa tab.

Thread: flussi di esecuzione. Nel 1° modello  
ne abbiamo di più, nel 10° modello  
abbiamo già processi, ognuno con un suo  
thread.

Un thread può creare un altro, ma non c'è  
affidabilità, sono fratelli

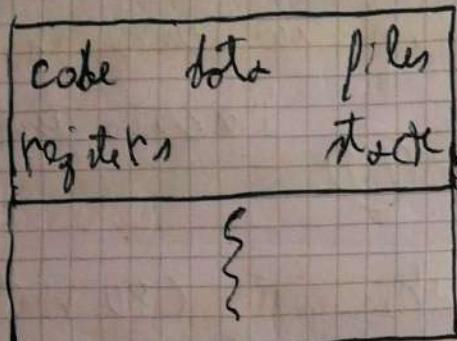
Vantaggio: se un thread fa una chiamata  
bloccante, il SO non blocca tutto il  
processo, come nel 1° modello ma solo  
quel thread. (liberalmente)

Il modello a Thread cerca un modo  
per avvicinare ancora di più il parallelismo

## Thread è Compatibile

- PC, Register, Stack, State
- (il resto è tutto condiviso)

2° modello



1° modello

Code	Data	Files
Reg.	Reg.	Reg.
Stack	Stack	Stack
{	{	{

Ogni thread ha il suo PC per le operazioni  
delle sue varie routine.

E applica il concetto degli stati anche ai thread  
quindi si viene usato scheduling, con un cambio  
di contesto più veloce.

Il problema dello scheduling  
è rendere lo switch sul

thread, non più sul processo

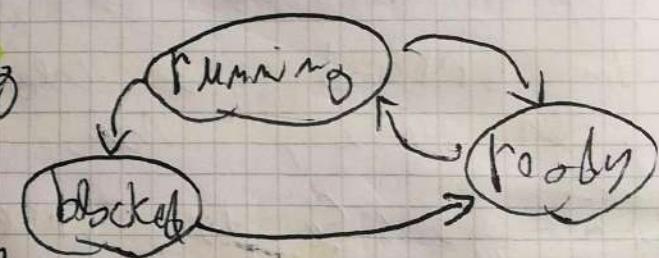
Poi se abbiamo switchare su thread,

l'efficienza per cambiare rapidamente re i

due thread sono fratelli o no.

Dato che se si fa lo switch tra fratelli non  
c'è lo swap della memoria, solo la gestione

dei register e stack, invece se si fa lo  
switch tra non fratelli si deve fare anche  
la mappatura in memoria.



Poco molto shadowing in certi momenti si preferisce lo switch verso fratelli.

Thread: "process leggero"

### Ospozione tipiche

thread-create: un thread nasce in altro

thread-exit: il thread dismano termina

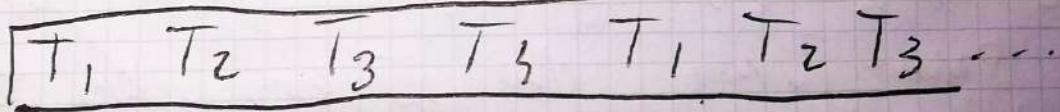
thread-join: un thread si incarica di una fine di un altro thread

thread-yield: il thread dismano libera volontariamente la CPU.

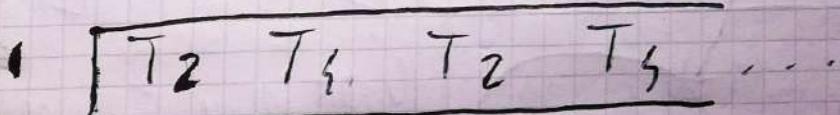
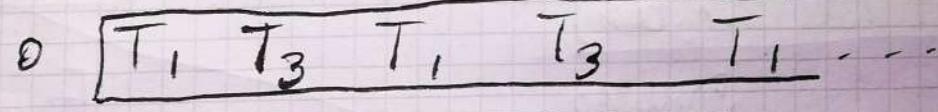
### Programmazione multicores

I thread permettono maggiore realibilità con sse che permettono multithreading e con sistemi con architettura multicore.

Single-core



Multi-core 0  
(2)



Nel single-core: esecuzione interleave

Nel multicore: paralleloismo puro

Il multithreading (o "genetizzazione") solo se i due thread concorrono sulla CPU sono fratelli, poiché concorrono lo stesso la memoria

## Applicazione basata sul multi core comportamento

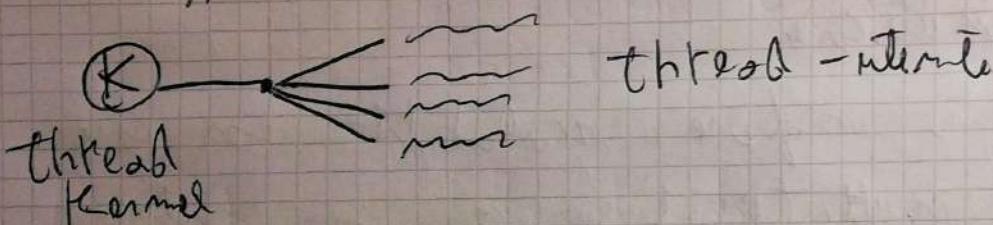
delle core in più da ottenzione

- Separare i task
- suddividere i dati
- test e debugging (già compilato)
- bilanciamento (by task)
- dipendenze dei dati

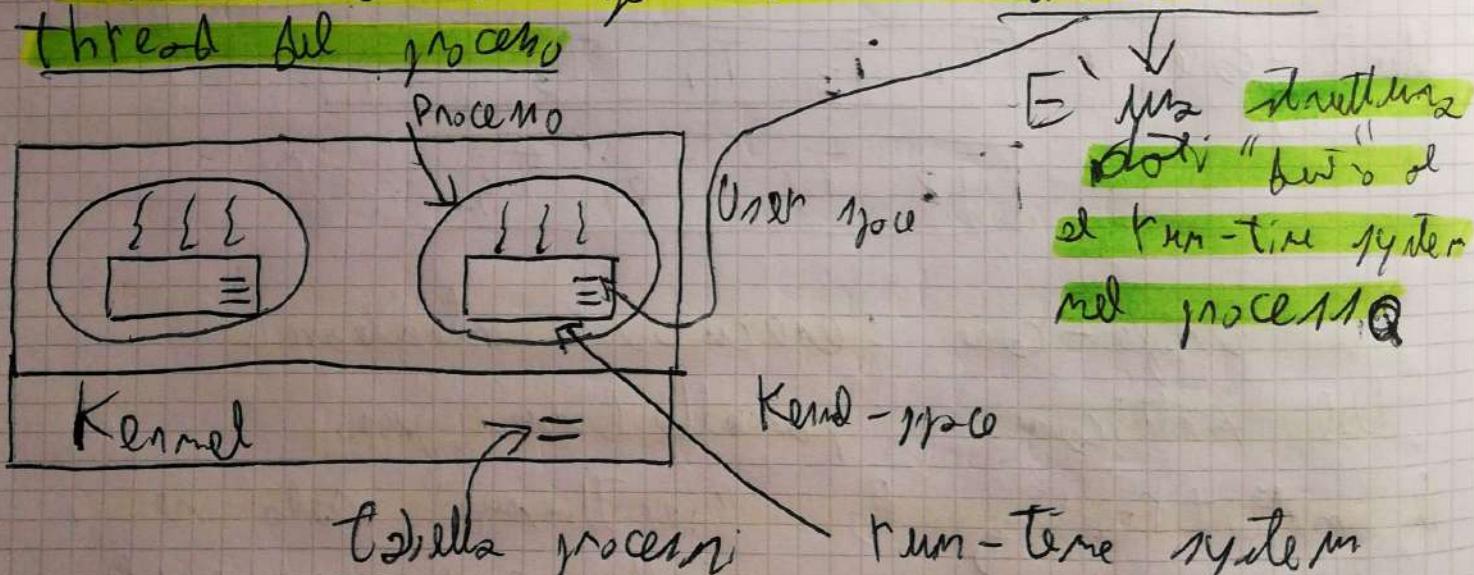
## Thread a livello utente

### • 1 o molti

- può essere utile se il Kernel non supporta i thread



Ci sono librerie che implementano switch run-time che gestiscono la tabella di thread all processo



Come avviene lo switch di Thread se il Kernel non ne ha nulla? Su base spontanea, il thread prima o poi dovrà lasciare la CPU.

Pro: Dispatching non richiede trap = context switch trap  
Schedulering personalizzabile

Contro: Chiamate bloccanti | Si blocca tutto il processore,  
possibilità di non ritorno della CPU | fino che il Kernel non fa  
della relazione in thread

Selezione gestione  
page-fault Mo chiamata a sistema

Selezione per vedere se una fault genererà un  
blocco

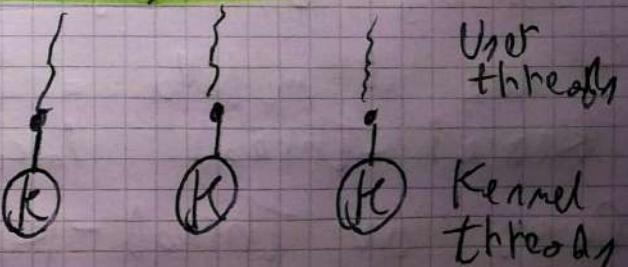
Page fault | se il programma fa una chiamata o  
solto sotto un'istruzione non im-  
magine (grazie) il SO prende  
l'istruzione e le sue vicine dal  
Disco.

Se un thread causa un page-fault il Kernel  
blocca tutto il processo e procede a quello  
traferimento I/O nonostante ci saranno anche  
altre thread eseguibili

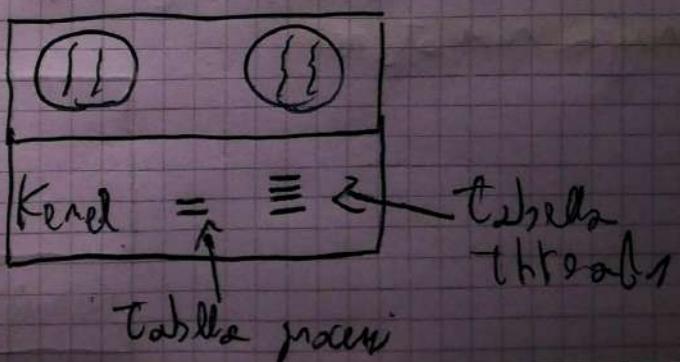
## Threads e Diretta Kernel

Questa non è mai la  
thread-yield

- modelli 1 a 1
- richiede supporto  
specifico del Kernel



Abbiamo una sola  
tabella per thread  
nel Kernel

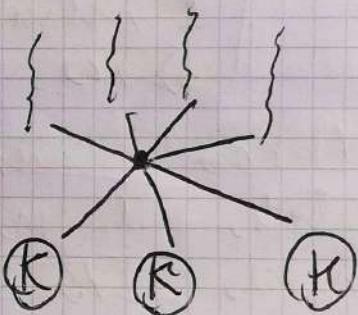


Pro: un thread se ha una chia parla  
blockante non impatta gli altri.

- Contro:
- Creazione e distruzione threads più costose
  - Il numero di threads diversi limitato, è possibile non necessario ricrea
  - Context-switch più lento, richiede la swap

In realtà lo switch se ci tra modelli è più lento, è più veloce se i thread non sono prestati.  
(ci intende rigetto al modello thread a livello utente)

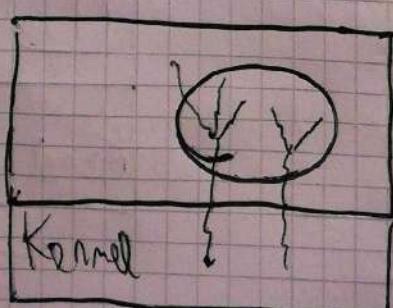
## Threads modello ibrido



- molti a molti
- Prende i vantaggi degli altri due modelli

E riguarda risorse del sistema ibrido un numero limitato di thread Kernel.

E usa il modello a livello Kernel, per sé ogni Kernel-thread fa un gestore per thread utente. Ognuno di thread-kernel è associato a 1 o più thread utente, l'esecuzione è decisa dal programma utente.



## Oggi

- Tutti i SO prevedono i thread a livello Kernel, ci sono poi librerie che permettono di programmazione di user i thread utente
  - Ci sono anche le librerie di accesso ai Thread (e pressoché dal web modo)
- Pthreads & Posix (con specifiche per il particolare sistema)
  - Threads WIN32
  - Thread in Java

## Comunicazione tra processi

Esistono varie tecniche

Più venire (i messaggi) ne esiste un collegamento

I/O tra processi (risorse file)

• La comunicazione è detta Inter Process Communication (IPC)

Di solito ci sono 3 canali separati nella comunicazione: Target - Origin - Error

Type: strumento di comunicazione bidirezionale, può buffer, è adattante limitata poiché non costituisce una comunicazione come flusso di dati

Problemi della comunicazione:

- come sincronizzare dati
- accavallamento di operazioni
- sincronizzare le operazioni

## Tecniche di memoria concorrente

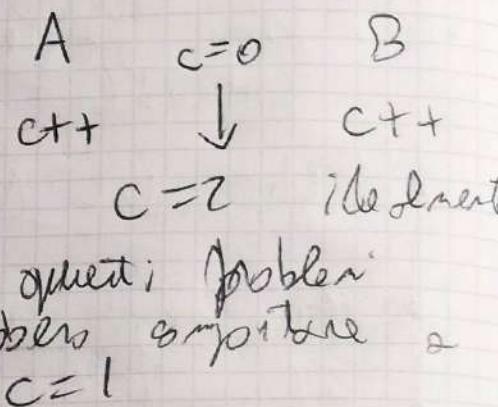
Due processi (o più) contribuiscono alla struttura dati, non possono creare come problema perché le **CORSE CRITICHE** (legate al parallelismo)

Esempio: risparmio sincrono

Bisogna realizzare (bene)

le **sezioni critiche** nei processi

mi fatti combinarsi.



Le corse critiche possono verificarsi anche nel codice del Kernel, fatto che anche qui ci sono flussi paralleli

Soluzione: mutua esclusione nell'accesso alle **fatti combinarsi**

Dobbiamo introdurre le **SEZIONI CRITICHE**.

- Ecco ora restituire alle strutture dati in combinatorie.

- In un processo poniamo identificare pezzi di codice che agiscono su dati combinarsi con un altro processo.

All'inizio e alla fine della sezione critica ci sono dei meccanismi che evitano che un altro processo "ponga entro", ciò viene intonato finché il 1° processo non esce dalla sezione critica.

Quindi se A è in sezione critica, B può proseguire e fatto che non entri in una zona critica

Così l'importante è che non entrambi siano in zone critiche contemporaneamente, e nello stesso istante dati.

## Buona soluzione ha le 5 caratteristiche

- mutua esclusione nell'accesso alle zone critiche
- Non fanno assunzioni sulla velocità di esecuzione e/o numero di CPU
- Un process che è fuori da una zona critica non può bloccare un altro
- Nessun process può aspettare all'infinito il permesso di entrare in zona critica

N.B.: Se A e B condividono due strutture dati.  
la mutua esclusione non è detto esista,  
la versione critica è relativa a una certa struttura dati.

## REALIZZARE LA MUTUA ESCLUSIONE

Più idea: Distributore di interrupt (può creare lista in K-mode)

E' molto semplice ma causa la monodistribuzione della CPU.

Svantaggio: se ti sbilanciamo gli interrupt, li sbilanciamo per TUTTI

A volte lo fa il Kernel quando vuole essere sicuro che nessuno lo interrompa

Distributore di interrupt non funziona come soluzione nei sistemi multiprocessori.

Condizione: non è una vera e propria soluzione

## 2<sup>a</sup>: Variabile di lock (soluzione software)

Il processo controlla una variabile critica con uno (o più) processi, la variabile è detta lock.  
Se è 0 il processo lo setta a 1 e entra nella regione critica (grado ecc lo si porta a 0).

Se è 1 aspetta se atteso a 0.

Siamo facendo busy waiting.

Problema: la variabile è comune, il processo fetch-store può dare problemi di corre critiche

## 3<sup>a</sup>: Alternanza turn (soluzione software)

```
[int N=2 int turn=0]
```

```
function enter-region (int process)
```

```
while (turn != process) do nothing
```

Si replica se turn corrisponde al process, se non lo è, lo blocca

```
function leave-region (int process)
```

```
turn = 1 - process
```

Dai così i due meccanismi per entrare nella regione critica.

Con queste due procedure facciamo fare ai processi una alternanza turn.

E' una soluzione che fa busy waiting  
(un lock che non serve meccanismo è detto  
spin lock)

La soluzione è generalizzabile a N processi

Sono implicati turni rigidi e si viola la 3a  
combinazione in quanto i processi potranno  
bloccare altri anche se non sono in  
zona critica.

Un'ottimizzazione è la Soluzione di Peterson  
che risolve il problema lasciando l'alternanza stretta

[int N=2

int turn

int interested[N] → inizializza le celle a false

function enter\_region (int process)

other = 1 - process

interested[process] = true

turn = process

while (interested[other] = true AND turn = process) do  
nothing

function leave\_region (int process)

interested[process] = false

Analisi della situazione di Peterson nei 3 problemi contestati:

1) Processo A va "fa 3b", interessed [other] è falso quindi non vi blocca nel loop

2) Sono dentro A, dovrebbe scattare la routine di B ma vi blocca nel loop perché  
• interessed [other] è vero  
• tutti<sub>m</sub> = process è vero

Quando A finisce interessed [other] diventa falso  
quindi B può proseguire

3) A e B cercano di entrare nella zona critica contemporaneamente

C'è una contesa, entrambi cercano di impostare tutti<sub>m</sub>

Per entrambi interessed [other] sarà vero, però tutti<sub>m</sub> avrà uno dei due valori.

Quindi per un processo tutti<sub>m</sub> = process sarà vero per l'altro processo sarà falso.

Quando la "vincente" farà la leave-region, per l'altro processo (che era bloccato) la interessed [other] sarà falso e quindi potrà proseguire.

- C'è anche busy waiting
- Ci può appaltizzarsi a N processi
- Può dare problemi nei sistemi multi-procenzione per il Notching degli accessi alla memoria centrale.

Alcune architetture implementano delle "barriere" che permettono al programmatore di evitare il Nondio di slice in certe zone.

Soluzione con TSL e XCHG (avendo busy waiting)

Molte architetture mettono a disposizione la TSL (test and set lock)

Sintesi: TSL registro, lock

Era è un'operazione atomica, blocca il bus di interr.

L'effetto della TSL è  
bloccare il bus  
garantisce atomicità.

1) MOVE registro, lock  
2) MOVE lock, 1

Funzioni:

[enter\_region:

TSL registro, lock  
CMP registro, #0  
JNE enter\_region  
RET

] leave\_region:

MOVE lock, #0  
RET

La lock viene a cogne se già c'era un blocco  
e non c'era il blocco lo fa il processo stesso.

In ogni corso lock sarà 1 alla fine, controllando  
il register vediamo se era bloccato (lo stava uscendo prima), se lo era (quindi era 1) ritorna alla  
etichetta enter\_region con lo JNE (è un loop)

Se non c'era il blocco fa RET e prosegue il processo  
(che ha bloccato la funzione mettendo lock a 1)

La XCHG è molto simile alla TSL

Le soluzioni viste sono basate sullo SPIN lock  
Lo spin lock causa:  
- tempo di risveglio  
- Problema dell'invocazione  
- Si rivolge

Sono H e L due processi  
riguardanti ad altri e hanno priorità e via  
di fatto stanno l'uno sopra l'altro.  
H è bloccato per un I/O (non finito o una  
sezione critica)

L entra in una sezione critica, finisce l'I/O e non  
è torna pronto.

L viene interrotto durante quella sezione critica  
perché H ha maggiore priorità ma H sta  
pronta ad entrare in sezione critica e  
non può più far colpa della invisibile & fastidiosa (=)  
interruzione da L.

Si è creato uno stesso

### Soluzione

Si fa al processo la possibilità di bloccarsi  
(in modo passivo, viene rimesso dai proc. passivi)  
e più di rivedersi al momento abatto.

Strumenti: sleep, wakeup. Sono due syscall  
Se un processo fa sleep si addormenta  
Se fa wakeup in un altro processo lo risveglia

## Problema del Produzione - Consumazione

Abbiorvi due processi produzione  
processi consumo  $\Rightarrow$  & informazione

variabile count comune, nessa a 0

### function producer()

while(true) do

item = produce\_item()

if (count=N) sleep()  $\leftarrow$  inserisci\_items(item)

Count = Count + 1

if (Count=1) wakeup(consumer)  $\leftarrow$  buffer ~~vuoto~~  
con un item, megliano cons.

### function consumer()

while(true) do

if (Count=0) sleep()  $\leftarrow$  buffer vuoto, bloccare

item = remove\_item()

Count = Count - 1

if (Count=N-1) wakeup(producer)  $\leftarrow$  c'è una libera,  
commune-item(item)  
meglio prob.

Facciamo questa soluz. perché non aggiornare  
quanto velocemente i processi.

Si avranno in parte limiti di limiti e  
faccendo tornare e rivedere i processi

Problema: I due processi potrebbero bloccarsi

Il SO blocca consumer poco prima di fare sleep.  
Il producer fa wakeup ma consumer ha non  
torna forward

Il producer continua a ciclare e il buffer non viene e va in sleep.  
Nel frattempo il SO ha noncesso la CPU al consumer che era già sleep (dopo che producer ha fatto wakeup)

Sono entrate in sleep. Sono in stallo per sleep di un consumo critico

Situazione inversa: il SO blocca producer fino alla fine della sleep. Consumer fa wakeup sul producer (non dormiente)

Consumer cida e il buffer si riempie, va in sleep. Nel frattempo il SO ha noncesso la CPU al producer che è già sleep.

Sono entrate in sleep.

Soluzione: bit-flag ~~se il flag è impostato~~ a impostare a 1 se il flag è impostato a 0 e mettere a 0 il flag a seguito di processo NON dormiente

Come funziona il bit?

Se il bit è 1 la sleep non farà dormire il processo

Il problema sta nel fatto che

if (count=0) sleep()

NON fa niente

if (count=N1) sleep()

Il bit di attesa ha un problema: conserva solo una regista (e ciò può già consentire che mandino wakeup, non è più possibile) il consumer che manda wakeup, non

## Semafori

Generalizzazione del modello sleep e wake up

• È una variabile binaria S

• Si definiscono le operazioni down, up (o wait, signal)

È un decremento e incremento S

Down per non più decrementare S se è 0,  
diventa bloccante.

(Un processo se prova a decrementare S se è 0  
è bloccato)

La down non può fare il decremento se S è 0  
ma non finisce se non riesce a farlo.

La up non è mai bloccante

L'incremento potrebbe svegliare un certo processo

Gli argomenti di up e down sono i processi  
ma semafori.

Dà un punto di vista logico circa la cosa dei  
processi bloccati dal ~~processo~~ S semaforo S.

Le operazioni down e up sono atomiche

Non c'è più corse critiche sul semaforo

Le garanzie sono date disponibilità degli interrupt  
e la macchina è meno CPU o meno la  
TSL / XCHG. ← si usa un lock che "protegge" S

C'è sempre un po' di spin lock ma è meno  
presente rispetto allo spin lock tra processi

L'incrementazione avviene senza busy waiting,  
mentre la lista dei processi bloccati

## Esempio

- A prova di fermentazione S che è + 0, va in stallo nella coda
  - B stesso caso
  - C fa la uj, B si blocca
- $$B \xrightarrow{++} 1 \rightarrow B \text{ si blocca} \rightarrow S \xrightarrow{-} 0$$

## Il semaforo ha vari utilizzi

Semaforo Mutex: per la mutua esclusione

Resettare i processi di non entrare tutti e due in sezione critica.

La S si trova tra 0 e 1

All'inizio della sezione critica:  $down(S)$

Allo fine:  $up(S)$

In questo modo garantisce la mutua esclusione

Semaforo contatore per le risorse per memorizzazione

Conteggio: una coda tipologia di risorse

Esempio: numero di item nel buffer

Quando il buffer si riempie si potrebbe bloccare un altro processo

## Risolvere Produzione-Consumazione con semafori

Dati concorrenti

Int N=100

semaphone mutex = 1

semaphone empty = N

semaphone full = 0  $\rightarrow$  Conteggio delle risorse

function producer()

while(true) do

item = produce\_item()

down (empty)

down (mutex)

insert\_item(item)

down (mutex)

up (mutex)

up (full)

bloccante se  
empty è 0

Ci assicuriamo  
di lavorare solo  
nella critica

function consumer()

while(true) do

down (full)

down (mutex)

item = remove\_item()

up (mutex)

up (empty)

consumere\_item(item)

bloccante se full = 0

Se  $N=1$   
mutex è  
iniziale.

Ci assicuriamo mutex esclusione nelle zone  
critiche: insert\_item() e remove\_item()

Blocciamo Producer se le celle vuote (empty) sono 0,  
bloccano Consumer se le full sono 0

L'ordine tra down e up è fondamentale.

down e up sul mutex devono coincidere solo  
cioè de è regione critica

Il procedimento funziona anche con più di due  
processi

Nei thread intere (1 o molti) che fanno riferimenti  
a un solo processo N implementano mutex con TSL

### mutex\_lock:

```
TSL REGISTER, MUTEX  
CMP REGISTRA, #0  
JZ E OK  
CALL thread_yield  
JMP mutex_lock
```

OK: RET

### mutex\_unlock:

```
MOVE MUTEX, #0  
RET
```

E' tutto implementato in modi di user

Non ha nemmeno fatto busy waiting (unisci la  
yield con lavoriamo su altri thread)

E' molto efficiente dato che non lo lock né lo  
unlock richiedono il Kernel

Queste due procedure sono ricavate dagli estremi  
della zona critica nel riferimento a mutex.

### FUTEX

I mutex nella user-space sono inefficienti per  
lo spin lock già come hanno dimostrato.

Ci sono situazioni in cui è meglio fare spin lock  
classico altrimenti in cui è meglio usare in Kernel

Per questo nome Futex = fast user space mutex  
che entra in syscall because se è troppo niente  
necessario

### Il Futex ha due parti

- A livello Kernel
- Libreria utente

Le prime code di thread bloccati

che usa un variabile lock

la causa per la reason critica è  
in mutexnable con la TSL (niente)

L'unico motivo per cui richiamiamo il Kernel è  
nei libri che bloccano il processo/thread

L'unico caso in cui si richiama il Kernel è  
la contesta di sezione critica fra i thread.

## Monitor

Tipo struttura con variabili e procedure, garantisce mutua esclusione per il suo accesso.

I monitor funzionano per i thread.

Dentro al monitor ci sono le funzioni che in combinazione

il monitor garantisce che un thread alla volta possa eseguirsi.

Questo viene garantito da meccanismo di sincronizzazione.

- codi di attesa interno
- variabili di condizione

K  $\xrightarrow{\text{wait}} \text{operazioni sulle variabili}$   $\xrightarrow{\text{signal}} \text{condizione.}$

### Variabili speciali

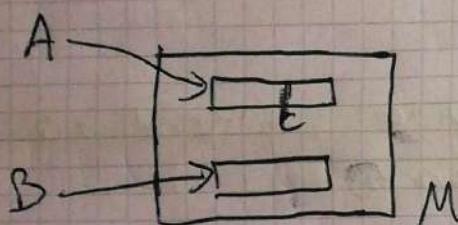
Per ogni variabile c'è una sola o più attese per i thread che si sono leggermente bloccati.

Le variabili e le operazioni sono molto semplici, fare signal su un thread non bloccato non fa problemi, e gli intrecciamenti (race cond.) non si riscontrano.

Le operazioni attese sono gestite all'interno dei metodi. Manca strumenti come semaphore mutex,

## Problema

Hyp: A e B lavorano con un monitor M  
A entra in M e fa wait in C



Ors B può entrare, B richiede le combinazioni per avviare A.  
Fa signal in C

Ma così ci sono sia A che B in M.

Per risolvere il problema basta sapere che il signal non fa subito il risveglio.

Le signal già avviate rimanono

- **Hoare (tecnica)**: Vale come "Signal and Wait" così risveglia A e va a dormire

- **Mes (Java)**: "Signal & continue" il risveglio viene effettivamente solo alla fine del lavoro di B.

Sintesi: Fa riavviare i Thread  
Non nei momenti giusti

- **Composizio**: "Signal & return"

E forza il programmatore a fare la signal alla fine del thread

In questo modo si evita il blocco tra la chiamata signal e la sua effettiva realizzazione

NB: Se faccio signal su un monitor dormiente  
non c'è problema, non c'è yet il monitor a "memorizzare" le segnali

produzione - consumo coi monitor

monitor pc-monitor  
cond non full, empty  
int count = 0

function insert(item)

if count = N then wait(full)

insert-item(item)

count ++

if count = 1 then signal(empty)

function remove()

if count = 0 then wait(empty)

removed = remove-item()

count --

if count = N-1 then signal(full)

function producer()  
while(true)  
item = produce-item()  
PC-monitor.insert(item)

function consumer()

while(true)

item = pc-monitor.remove()

consume(item)

Dato che insert e remove hanno mutua esclusione, non abbiamo problemi legati alle corse critiche.

Sembra meno oggi tra i processi

tra thread ci invia lo stesso i messaggi, negli memoria condivisa, scambiano messaggi i processi.

Si usano le primitive | send(dest, msg)  
receive( src, msg )

Se receive potrebbe essere bloccato se il destinante non dà messaggio

Il problema dello scambio di messaggi è estremamente  
anche in C++ ci sono già implementazioni con le  
library MPI

In realtà anche la std:: no dinamico bloccante  
re lo scambio è tra i buffer & i buffer  
è falso.

Metodi inizializzazione MPI: Brutto o mailbox

Brutto → invia msg a Lca quindi v'è  
Mailbox → invia msg nella cassetta | critica stessa  
dei dati

Problema: lo scambio di messaggi è meno efficiente  
rispetto alle memorie condivise.

### Problema - Consumazione con messaggi

function producer()

while (true)

item = produce-item()

build-msg(M, item)

send(consumer, msg)

function consumer()

while (true)

receive(producer, msg)

item = extract-msg(msg)

consume(item)

### Problema dei 5 filosofi

Descrivere il problema dell'accesso ad un  
numero limitato di risorse da parte di  
processi in esecuzione

Abbiamo 5 filosofi, 5 piatti e 5 forchette

① ② ③ ④ ⑤

Un filosofo per mangiare deve avere  
entrambe le forchette accanto

Dunque abbiamo ottime le tigiane e  
i processi (filosofi) che le vogliono usare

### Soluzione 1

int  $N = 5$

function fibonaccer(int i)

CRNK()

take-fork(i)

take-fork((i+1) mod N)

CRNK()

put-fork(i)

put-fork((i+1) mod N)

I problemi sono bloccanti

Problema: Se ogni fibo  
mette in fork e da  
ci si blocca

### Soluzione 2

Come in Sol 1 però la take-fork non è  
bloccante.

Se ha preso una forchetta e l'altra è  
bloccata, libera quella che ha preso e ignora.

C'è la remota possibilità che riceva uno stop  
attivo.

### Soluzione 3

Facciamo come in Sol 2 un altro tentativo  
aspettando però un tempo random

### Soluzione 4

Usare un mutex sul tavolo →

Permetteva a un  
sol fibo di alla volta  
di mangiare

Soluzione basata su  
semafori Sidle

Soluzione basata su  
monitors Sidle

## Sistemi basati sui tempi

Introduciamo 3 stati : THINKING, HUNGRY, EATING

- Thinking : pensa ovviamente
- Hungry : ha fame, non ha le forcille
- Eating : ha le forcille

Vogliamo che un filosofo faccia questo percorso



Da Hungry a Eating  
passerà quando avrà  
il "pernello"

Viamo un **ritratto continuo** stato [N] che permette di esprimere lo stato  
di ogni filosofo

Viamo un **tempono matrice** per la matrice  
esclusione in quale è quel tempo, smo  
un filosofo, conservati nel vettore S[N].

Questi tempi generano al filosofo di abbandonare  
se non vengono a prendere le forcille e  
di essere svegliati poi dai vicini che avranno  
finito di mangiare

## Funzioni (solte in idle)

- left(i) : da vicino left di i
- right(i)
- test(i) : Se i è fine non sono E  
allora l'asse a E (e gli facciamo  
v) sul tempo)
- takeforks(i) : lo stuo pensa a H e facciamo la test.  
le tre operazioni smo facciamo tra  
down e v) su mateix.
- Alla fine facciamo down sul tempo

- gut-fork(i): lo stesso punto a T e facciamo test sui due vicini. Le 2 og. sono no collegate tra down e up nel mateix
- philosopher(i): basandosi sul filosofo prima parola, poi prende le fork (a prova), mangia e pone le forchette.

### Cose in stile

Vogliamo rendere la test bloccante in un qualche modo, ma lo stile sta tra down e up mateix. Bloccare pernici tutto.

Facciamo un trucchetto; Se non ci sono le condizioni perché i punti a E non viene fatta la Uf (SIC!) e dunque la down (non dalla versione critica) in ~~take~~forks diventa bloccante. Lo stesso sarà la fonte di un altro filosofo. Allora la gut-fork (che fa le test in vicini)

```
int N=5; int THINKING=0  
int HUNGRY=1; int EATING=2  
int state[N]  
semaphore mutex=1  
semaphore s[N]={0,...,0}
```

```
function philosopher(int i)  
  while (true) do  
    think()  
    take_forks(i)  
    eat()  
    put_forks(i)
```

```
function left(int i) = i-1 mod N  
function right(int i) = i+1 mod N
```

```
function test(int i)  
  if state[i]=HUNGRY and state[left(i)]!=EATING and state[right(i)]!=EATING  
    state[i]=EATING  
    up(s[i])
```

```
function take_forks(int i)  
  down(mutex)  
  state[i]=HUNGRY  
  test(i)  
  up(mutex)  
  down(s[i])
```

```
function put_forks(int i)  
  down(mutex)  
  state[i]=THINKING  
  test(left(i))  
  test(right(i))  
  up(mutex)
```

## Soluzione basata sui Monitors

Le take e put diventano punti sul monitor  
acquisendo multa esclusione.

Uso un vettore di variabili di condizione, una  
per libri, visto che abbondanti e vagabondi.

Le ghiacciofet tiene la maglia, misura ancora il  
vettore. Vede ma il multex ed se no, il vettore  
condizione è sufficiente.

La test(i) fa la stessa cosa ma al posto di up(sic)  
faciamo signal(sic[i])

La take\_forks fa passare lo stato a H, lo misura  
e controlla se lo stato è E, se non lo fa  
waits su di uno.

La put\_forks passa in T e fa la test su 2  
vicini.  
Codice in slide

```
int N=5; int THINKING=0; int HUNGRY=1; int EATING=2
```

```
monitor dp_monitor
```

```
int state[N]
```

```
condition self[N]
```

```
function take_forks(int i)
state[i] = HUNGRY
test(i)
if state[i] != EATING
    wait(self[i])
```

```
function put_forks(int i)
state[i] = THINKING;
test(left(i));
test(right(i));
```

```
function test(int i)
if ( state[left(i)] != EATING and state[i] = HUNGRY
and state[right(i)] != EATING )
    state[i] = EATING
    signal(self[i])
```

```
function philosopher(int i)
while (true) do
    think()
    dp_monitor.take_forks(i)
    eat()
    dp_monitor.put_forks(i)
```

## Problemi bei lettori e scrittori

I due processi agiscono su un DB.

Il DB è da bloccare nei momenti opportuni.  
Ci può essere tanti lettori contemporaneamente ma  
basta anche un solo scrittore per rendere i dati  
inconsistenti.

Bisogna gestire in qualche modo l'accesso al DB.

- Tanti lettori vanno bene, basta un solo scrittore per avere problemi.

### Soluzione con semaphore

function reader()

```

1   while (true)
2     down (mutex)
3     rc++
4     if (rc=1) down (db)
5     up (mutex)
6     read - db()
7     down (mutex)
8     rc--
9     if (rc=0) up (db)
10    up (mutex)
  
```

semaphore mutex = 1  
" " db = 1  
int rc = 0

function writer()

```

while (true)
think()
down (db)
write - db()
up (db)
  
```

La riga 4 serve per chiudere la porta di altri  
scrittori mentre la riga 9 serve per aprire

Il writer con down(db) e up(db) chiude e  
apre la porta a tutti.

Il mutex serve per gestire la variabile continua  
rc.

Svantaggio: la scrittura in coda potrebbe aspettare  
un tempo imeterminato per l'accesso.

E' una soluzione sbilanciata a favore dei lettori.

Il lettore deve avere una down (ab) dentro a una  
sezione critica non fa comunque senso.  
Se avessero due abbiano uno startore, entro un  
lettore che non ha niente problema il down (mutex) non  
è bloccato né da un down (ab).

Il prossimo lettore è bloccato sul down (mutex),  
in questo modo garantiamo inizialmente la esclusione.

### Soluzione con monitor

monitor rw\_monitor

int rc=0; boolean busy\_on\_write=false  
condition read, write

function start\_read()

```
if (busy_on_write) wait(read)  
rc++  
signal (read)
```

function end\_read()

rc--

```
if (rc=0) signal (write)
```

function start\_write()

~~busy\_on\_write = false~~

```
if (rc>0 OR busy_on_write)  
wait (write)
```

busy\_on\_write = true

function end\_write()

busy\_on\_write = false

```
if (in_queue (read))  
signal (read)
```

else

signal (write)

read()

while (true)

rw. start\_read()

read\_ab()

rw. end\_read()

write()

while (true)

rw. start\_write()

write\_ab()

rw. end\_write()

La variabile busy-on-write ci serve a bloccare  
processi che vogliono entrare mentre i due elementi  
sono scritti sul DB.

Nella funzione read se i lettori trovano il write (ne c'è)

Nella stessa read abbiamo un signal (read) per  
avvertire in questo modo i lettori che i dati  
sono bloccati.

La start-write controlla se non ci sono altri o  
lettori, nel caso si mette in attesa.

La end-write impone la variabile a false e  
controlla se nella coda c'è un lettore e lo  
è negativo, se sì nega un altro scrittore.

Questa soluzione prebilige ancora i lettori

## Soluzione 2

Moltiplichiamo la start-read: b' il Giventà  
così if(busy-on-write OR In-queue(write)) wait(read).

Ora la wait(read) avviene anche se c'è in  
coda un writer.

Così evitiamo che il lettore lo sovralichi.  
La soluzione ancora silenziosa per i lettori è  
consistere nella procedura end-write() che  
"preferisce" i lettori.

```
monitor rw_monitor
int rc = 0; boolean busy_on_write = false
condition read,write

function start_read()
    if (busy_on_write OR in_queue(write)) wait(read)
    rc = rc+1
    signal(read)

function end_read()
    rc = rc-1
    if (rc = 0) signal(write)

function start_write()
    if (rc > 0 OR busy_on_write) wait(write)
    busy_on_write = true

function end_write()
    busy_on_write = false
    if (in_queue(read))
        signal(read)
    else
        signal(write)
```

```
function reader()
    while true do
        rw_monitor.start_read()
        read_database()
        rw_monitor.end_read()
        use_data_read()
```

```
function writer()
    while true do
        think_up_data()
        rw_monitor.start_write()
        write_database()
        rw_monitor.end_write()
```

```
graph TD
    subgraph Monitor [Monitor]
        startRead[function start_read()]
        endRead[function end_read()]
        startWrite[function start_write()]
        endWrite[function end_write()]
    end

    subgraph Reader [Reader]
        readerFunc(function reader())
        readDatabase[read_database()]
        endReadFunc(function end_read())
        useData[use_data_read()]
    end

    subgraph Writer [Writer]
        writerFunc(function writer())
        thinkUpData[think_up_data()]
        startWriteFunc(function start_write())
        writeDatabase[write_database()]
        endWriteFunc(function end_write())
    end

    startRead --> readerFunc
    endRead --> endReadFunc
    startWrite --> writerFunc
    endWrite --> endWriteFunc
```

### Soluzione 3

Uscita alla Z ha multiplexing la anal-write .  
• if (in\_node (write)) signal (write) else signal (read)  
Questa soluzione contrariamente è molto  
simplificata e favorisce ~~il setto~~ degli errori

```

monitor rw_monitor
int rc = 0; boolean busy_on_write = false
condition read,write

function start_read()
    if (busy_on_write OR in_queue(write)) wait(read)
    rc = rc+1
    signal(read)

function end_read()
    rc = rc-1
    if (rc = 0) signal(write)

function start_write()
    if (rc > 0 OR busy_on_write) wait(write)
    busy_on_write = true

function end_write()
    busy_on_write = false
    if (in_queue(write)) signal(write)
    else signal(read)

```

```

function reader()
    while true do
        rw_monitor.start_read()
        read_database()
        rw_monitor.end_read()
        use_data_read()
    
```

```

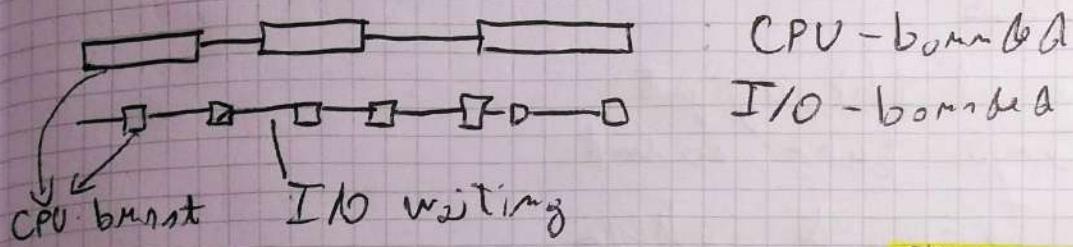
function writer()
    while true do
        think_up_data()
        rw_monitor.start_write()
        write_database()
        rw_monitor.end_write()
    
```

## Scheduling

Nei processori multiprogrammati ci sono appena tanti processi che i "congestioni" la CPU  
la parte del SO che i occupa della scelta del prossimo processo è lo scheduler attraverso  
l'algoritmo di scheduling.

Distinguono i processi in

CPU-bound (uso giù CPU)  
I/O-bound (uso giù I/O)



Si tende a favorire i processi I/O-bound perché se si facesse il contrario la CPU venirebbe monopolizzata. Preferire l'I/O-bound permette di far lavorare bene entrambi i reparti: CPU e I/O.

Quando si usano lo scheduler?

- terminazione (e creazione) processi
- diverse bloccante e arrivo interrupt
- interrupt periodici (generano non monopolizzazione CPU)
- interni non-preemptive (senza prelazione)
- interni preemptive (con prelazione)

Coi interni preemptive si evita che un processo monopolizzi la CPU.

Lo scheduler collabora col dispatcher che implementa la transizione tra il processo che finisce e il processo scelto dallo scheduler.

L'intervallo di dispatching: intervallo temporale tra inizio e fine lavoro di dispatching

## Obiettivi di un algoritmo di scheduling

Abbiano 3 tipi di ambienti e di conseguenze 3 tipi di algoritmi.

- **BATCH**
- **INTERATTIVI**
- **REAL-TIME**

### Obiettivi comuni

- eguale nell'assegnazione delle CPU
- bilanciamento nell'uso delle risorse

### Obiettivi dei batch

- maximizzare il throughput
- minimizzare tempo turnaround
- minimizzare tempo di attesa

### Obiettivi dei sistemi interattivi

- minimizzare il tempo di risposta

### Obiettivi nei sistemi real-time

- rispetto alle deadline
- prevedibilità

## Scheduling nei Batch

(un batch lo vediamo come un insieme di job)

- **First-Come First-Served FCFS**  
egisce per ordine di arrivo

E' un sistema Non-preemptive che usa una coda FIFO

- **Shortest Job First SJF**  
egisce per brevità

E' un sistema Non-preemptive e presupone di conoscere il tempo impiegato per ogni lavoro a priori.

P	T
P <sub>1</sub>	25
P <sub>2</sub>	3
P <sub>3</sub>	3

$t_{ma} = (0 + 25 + 25) / 3 = 17$   
 $t_{mc} = (25 + 25 + 30) / 3 = 27$

E' ottimale solo se i lavori da svolgere sono tutti subito disponibili (tempo di attesa nullo)

### Shortest Remaining Time Next

Utile al SJF ma è preemptive per risolvere il problema legato ai tempi d'attesa non nulli.

Il principio di brenta è applicato in questo momento sul "Remaining time".

Esempio: Eseguiamo che job S è arrivato a 2 (inizio 3).  
Arriva B che vale 2.  
A viene bisognoso, è eseguito B e poi viene A

### Scheduling nei sistemi interattivi ROUND ROBIN

Veniamo a preemptire la FCFS, la gestione è applicata basandosi su un quanto di tempo.  
Time slice: tempo massimo che il job può usare la CPU prima che venga riassegnato.

Se la sol. è  $B \rightarrow F-D-G-A$  e finisce il time slice di B  $\rightarrow F-D-G-A-B$

Conseguenza: con n processi e un quanto di q ms. Ogni processo avrà diritto a  $\frac{1}{n}$  di CPU e attende al più ( $M-1/q$ )ms

Quando un processo va in coda a un evento I/O è tutto della sol. il quanto torna rimasto nella coda, se in testa o alla fine è a disposizione del progettista.

Metterlo in testa sembrerebbe di ridurre subordinato a favore dei processi che fanno molto I/O

Valori tipici  $q = 20-50$  ms

## Scheduling a priorità

Regole di base: Si ammira il processo con più alta priorità da CPU.

Ammira un'unità detta numerica che indica le priorità di un processo.

### Ammiramento priorità

- Può essere binario o statico
- Tende a favorire I/O bounded

Un processo CPU-bounded tende ad usare sempre gli interi il suo  $q$  mentre l'I/O-bounded tende ad usare "parzialmente".

Criamo un algoritmo dove la priorità  $p$  è calcolata come  $\frac{1}{q}$  dove  $q$  è la frazione di uso della CPU da parte del processo. Se  $q = 50 \text{ ms}$  lo ha usato per 25 ms ha  $p = 2$  ( $P = \frac{1}{2}$ ,  $\frac{1}{12} = 2$ )

- SJF può essere visto come un banale algoritmo per priorità dove la  $p$  è decisa in base alla brevità.

Nei primi giri dell'algoritmo ovviamente non ci sono dei valori di  $p$ , i processi entrano decisamente in default e dopo i pochi istanti la situazione si stabilizza.

E' un intervallo dove si potrebbe scegliere di fare o non fare prelazione.

Ese: Esegue A che ha la  $p_{max}$ , entra B che ha priorità ancora più alta.

Sospendo A per eseguire B o conviene ormai finire A?

E' a discarica del maghiere

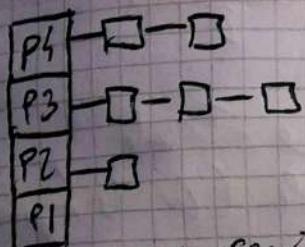
Se appliciamo subito la regola di base i processi a bassa priorità non resteranno mai su CPU, ovvero **l'eliminazione di starvation**.

Potremo risolvere la starvation con l'**aging**. Un processo aumenta la sua priorità man mano che rimane ad aspettare nella coda.

(Il processo vecchio chiede di essere eseguito)

Così permettiamo ai processi a bassa priorità di **far uscire ogni tanto la CPU**. E' una tecnica **non calcolistica**.

**Scheduling a code multipla (clasi di priorità)**



E' un insieme di code, quella più in alto ha i processi a priorità superiore.

P4 giunta a sua volta contenente processi con priorità 4.

Come scelgo un processo?

La scelta si divide in 2 fasi

• Quale coda prendere?

La coda non vuota già in alto

• Quale elemento di questa coda prendo?

La tecnica è **selezionabile**.  
Esempio: Applicare Round-Robin

Andiamo in contro prima di uno scheduling verticale e poi uno orizzontale.

In alto stanno per lo più processi I/O-bound e in basso i CPU-bound.

In base alla natura dei processi della coda alta e più scegliere meglio grande scheduling orizzontale.

Ex: Round Robin nella j*n* alta  
FCFS nella j*n* bassa.

Nel livello intermedio è molto meglio Round Robin  
ma con un quanto che cresce man mano  
che scendiamo.

~~Questo interno~~ Quando Risolve lo scheduling  
verticale?

- Se c'è un nuovo candidato che sta in mezzo  
alla j*n* alta
- Se la coda è vuota

Questo sistema può risolvere in modo elegante  
il problema della starvation.

Applichiamo a ogni coda una gerace attuale  
la CPU si dedica per quelle percentuali (ai temps)  
a quella coda.

P1	P2	P3	P4	%
5	15	30	50	

Su 10 record si  
dedica 5 su P1, 3 su P3  
ecc...

E' una soluzione migliore dell'aging

Come si ariegnano le priorità nelle code  
multiple?

- Si usa un criterio a feedback

Si osserva come si comporta un processo,  
determinando il suo ~~completamento~~ inserimento  
nella coda opportuna.

Se tempeza essere prioritario il q di lo  
spettiamo oggi, se tempeza ab works tutto  
lo spettiamo sotto.

## Shortest process Next SPN

Applicazione di SJF ai ritardi intatti

Necessità di un modo per calcolare la durata del prossimo burst di CPU.

Quindi appliciamo l'SJF basandoci sui burst

Cos'è possibile conoscere il burst prossimo di un processo?

Calcolando una stima basata sui burst precedenti

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

S: stima

T: tempo effettivo

$$\alpha = \frac{1}{2} \text{ (esempio)}$$

Questa formula fa peso maggiore alle stime e tempi "nuovi" nella trascrizione

La nuova stima dipende dalla precedente stima e dal precedente tempo effettivo

Se  $\alpha$  è prossimo a 1?

Do giusto peso a T

Vale il ricavamento.

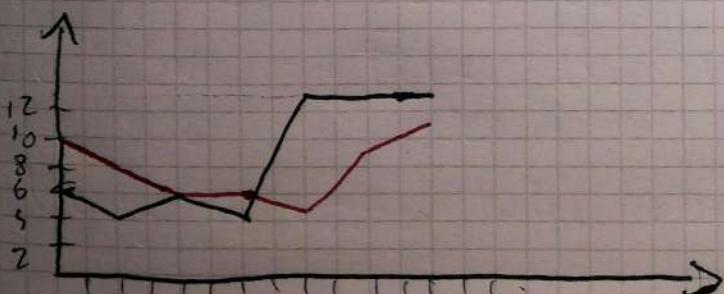
La migliore strategia ha nell'avere  $\alpha$  né troppo alto né troppo basso.

La stima tende ad avvicinarsi al tempo reale, graficamente parlando

$T_n$  6 5 6 5 13 13 13

$S_n$  10 → 8 6 6 5 9 11

Traente la formula e i ho calcolato



## Altri scheduling nei sistemi interattivi

- **Graziano:** E' possibile una percentuale di uso delle CPU e viene fatto rispettare. Per l'algoritmo viene tenuto conto del rapporto tra la CPU usata e quella. Se è  $\leq 0,5$  l'algoritmo fa "riscuotere" per ripetere passo 1. Se è  $\geq 2,0$  l'algoritmo fa saltare per portare verso 1. Soli un numero che l'algoritmo come la ritazionne si utilizza.
- **Lottizzazione:** E' come magazzinare biglietti con estrazione random. Per assegnare le risorse si fa, appunto, l'estrazione. Qui si intravede un bisogno di priorità. Infatti se vogliamo che senza eleggono uno un processo ad alto priorità basta analogamente già biglietti. I biglietti si "comprano". Permette di avere processi cooperanti ovvero, se un processo ne vuole può considerare priorità biglietto a un altro processo.
- **Fair-Share**: Si cerca l'equità tra gli utenti. Se finiamo qui nei processi "globalmente" non saremmo qui negli utenti. Idea: fare che bello girare negli utenti e poi nei processi.

## Scheduling dei thread

• **thread waiting** (il SO non ha nulla)

E' un algoritmo non preemptive e lo scheduling è sezionalezzabile.

• **thread del Kernel**:

E' deve rispettare le considerazioni: tutti i thread disponibili o restringere anche l'appartenenza a dei processi.  
Infatti switchare su un thread non protetto implica riprogrammazione MMU (unità gestione memoria) e a volte movimento cache della CPU.  
In certe situazioni è meglio provare di fare lo switch su un protetto.

## Scheduling nei sistemi multiprocessore

Possiamo fare:

• **multi elaborazione simmetrica**

Un bel processor fa da Master Server ovvero fa da amministratore per la gestione delle risorse. Le altre CPU lavorano sotto sua direttrice. Se ho più CPU il master server rischia di creare l'effetto "collo di bottiglia" e si riducono le prestazioni.

• **multi elaborazione asimmetrica**

Le CPU solo tutte uguali fra i loro, la CPU applica l'algoritmo di scheduling nel momento in cui non ha i requisiti thread.

L'algoritmo può pensare o fa una coda implicita dei processi pronti oppure potrebbe essere una coda personale per ogni processo.

Coda manifestata: Permette di non dover fare

indirizzamento verso una certa coda ma essa è una struttura dati comune quindi soggetta a race conditions.  
E' un problema risolvibile al costo che i processori obiettino per la metà esclusione.

Coda ripartita: abbiano sempre all'indirizzamento verso una certa coda per entrambi di dovere ai problemi legati alla struttura dati comune.

### Politiche di Scheduling

le politiche variano in base alla priorità o sulla preferenza dei processi.

Prioritazione: mantenere un thread in esecuzione mentre altri processori per sfruttare al meglio la cache.

La priorità deve essere debole: diverso il SO prova a mantenere questa caratteristica ma la migrazione sarebbe impossibile.

Forte: il thread è rimasto a tutti gli effetti quel processo.

La prioritazione crea dunque l'esclusione.

Processo Thread - Processore

### Bilanciamento del Carico

Si vuole distribuire il lavoro in modo omogeneo nei processori.

E' necessario questo meccanismo se i lavori sono divisi nei processori.

Può avvenire tramite

Migrazione Egitata (PUSH)

Spostamento (PULL)

**Push**: un thread dedicato a occuparsi di far uscire i thread per creare equilibrio.

**Pull**: un processore inattivo ne "scambia" uno in attesa su un'altra coda

Linux adotta un approccio ibrido

### Schedulat Windows 8

Usa un algoritmo preemptive basato su thread e classi di priorità.

E' possibile gestire le priorità

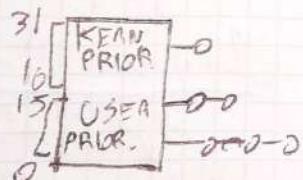
SetPriorityClass: per processi  
SetThreadPriority: per thread

Controlliamo una tabella per le

priorità date e relative che influiscono la priorità di base

Abbiano 32 priorità: da 0 a 31

Lo schedulatore guarda solo ai thread.



Si applica un algoritmo di selezione per priorità

in coda

All'interno della classe coda abbiamo poi un algoritmo di scheduling orizzontale Round-Robin

La tabella delle classi di cui ciascuna ha uno molto basso del tempo delle altre priorità, quindi si dà una priorità definita per i thread real time.

Le priorità sono dinamiche (tranne per RT)

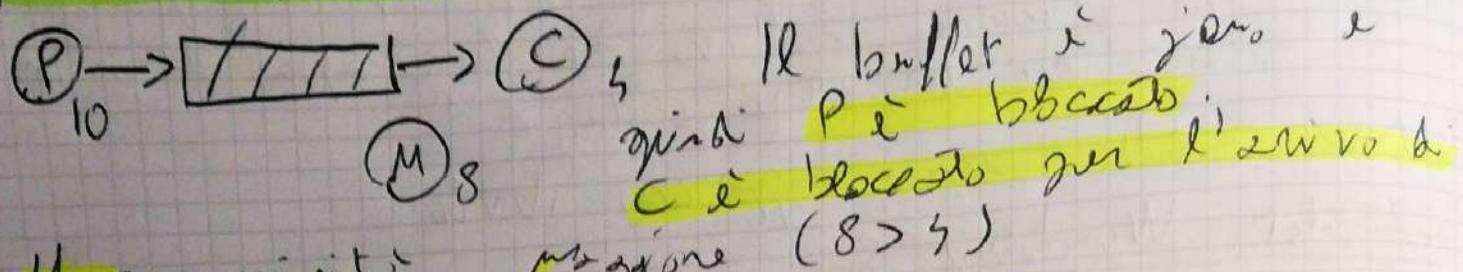
Ovvero a ogni ~~process~~ thread si applica la priorità di base e poi una delle dinamiche

- Mo del quanto

- Rigresso da evento I/O

- Si potrebbe aumentare il quanto in certe situazioni senza toccare la priorità

Questo sistema potrebbe subire in caso di  
problem di inversione di priorità



$M$  con priorità maggiore  
 $M$  è bloccato e non potrà da  $P$ .  
E' risolvibile facilmente con la tecnicha  
Autoblock

Come Agendo su  $\mu$  "moto", individua le operazioni  
tra processi che risolvono problemi  
in modo opportuno e blocca le priorità.

Close di Priorità 0: si rivede un thread  
speciale legato solo quando non c'è  
altro thread  
E' un task che non è necessario legarlo vero,  
e occupa della registrazione.

## TASK in LINUX

Giungiamo a un'unica entità che sintetizza  
concrete le processi e thread, permette di  
implementare semplici

o una task struct (nella PCB), che  
e contiene info sul task.

E' usata la syscall clone(func, stack, sharing flags, obj)  
che restituisce un PID

E' un modo unico per fare fork e thread\_create,  
infatti queste 3 chiamate sono coincidere sotto  
cette condizioni (detto dai flags)

E' usato PID per retrocompatibilità  
TID: task identifier  
Thread dello stesso processo  
PID uguali  
NB: 2 task hanno TID diversi  
ma potranno avere

## Schedulare O(1) a Linux

Sostituito poi da CFS

Distinguiamo 3 classi di thread

le priorità sono da 0 a 139

O è la più alta, 139 la più bassa.

RT FIFO e Round Robin vanno da 0 a 199 mentre il time sharing da 100 a 189

Le variazioni di priorità possono essere

- statiche: una sys call nice che continua da -20 a +19

se non intervengono ulteriori modifiche questi rimangono

- dinamiche: da -5 a +5 per I/O bounded CPU bounded

## Struttura algoritmo

Per ogni core abbiamo una funzione

ci metteremo poi all'interno della funzione

la coda active e la coda expired.

La active e la expired sono gruppi di code multiple, con una gestione simile allo scheduling a code multiple (c'è boppio scheduling)

In ogni coda ci poi applicato il Round Robin

Nell'esecuzione subiamo a porre il task a priorità maggiore e rendere finché non si vuole la active per poi switchare nella expired

Quando un task finisce il suo quanto, è spedito nella expired.

Il Round Robin funziona che se un task ha I/O boppo  $\frac{1}{3}$  del suo q alla fine dell'I/O il task è rimesso alla fine della coda ma con  $\frac{2}{3}$  del q

Se un processo avrà fabbici X ms ma non finire dopo questi X è spedito nella expired.

(non preempitive)

RT FIFO

(preempitive)  
Round Robin

time sharing  
(preempitive)

I quanti sono lunghi (800 ms) per priorità alta e corti (5 ms) per priorità bassa.

I decoloramenti e innalzamenti di priorità avvergono subito quando il processo passa dalla o viene eseguita.

N.B. attivo è eseguito e alternarsi.

L'algoritmo è  $O(1)$ .

Quando un processo è bloccato è tolto dalla struttura momentaneamente.

L'algoritmo è funzionante per multi processori.

Struttura RUN queue

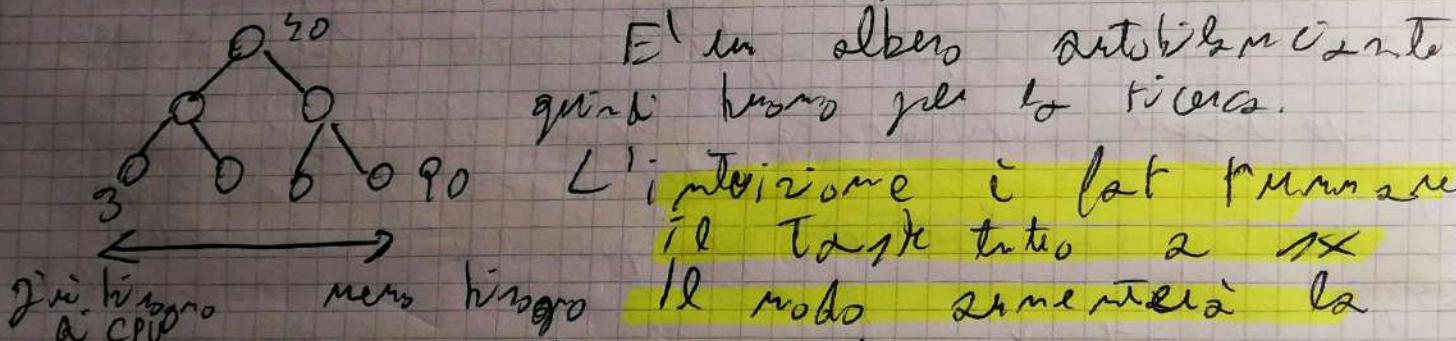
FLAGS CPU ...	ACTIVE	EXPIRED	PO PISR	PO PISR
			↑	↑

## Scheduler CFS

Crea lo scheduler garantito, cosa che non fa  $O(1)$ .

È uno scheratore una variabile detta Virtual Run-Time dove il tempo che ha la CPU ha fatto.

Usiamo un albero ramo nero che contiene i task e una come chiavi le rispettive virtual run-time.



E' un albero automaticamente quando hanno diritto di ricerca.

L'interazione è fatta funzionare il Task tutto a su a lungo verso lungo il modo aumenterà la chiave e l'albero si bilancerà.

Il bilanciamento è fatto periodicamente per non appesantire l'algoritmo, quindi prende il task che non può correre in po e poi si fa il bilanciamento. I timer ricevono di nuovo e dipendono anche dal numero di nodi.

Garantire la base (sistema di rete con server) è mappa nei desktop.  
Saranno: il virtual run-time e un altro modo.

I numeri crescono continuamente, prima o poi avranno un overflow.

Per evitare facciamo un "reset", ovvero il task col key minimo diventa 0, se un altro modo ha già più alto di 30 diventa infatti 30.

I DS rimangono uguali.

Quando arriva un task nuovo potrebbe essere meno che minimo 0 come massimo.

Come è implementato

Conseguenza: Un I/O-bound tende a ritardare a +x nell'albero per una matita, un CPU-bound al contrario.

Come è implementato il concetto di priorità?

Così fanno le decadenze: il virtual run-time dei processi a bassa priorità saranno già velocemente.

Il motivo è perché per i processi ad alta priorità è già grande per i processi a bassa priorità. Con questo sistema elegante evitiamo starvation e aging.

La rideistribuzione può avvenire anche per gruppi, così che non farà male in dentro che fra i gruppi.

Esempio: Nell'albero ci sono gli utenti; ogni utente "contiene" i processi propri che a loro volta relazionano con altri utenti con un albero.

## Gestione Memoria

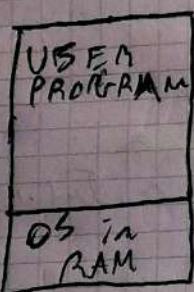
La memoria è divisa per gerarchia

La RAM è il livello più basso che è utilizzabile dalla CPU direttamente

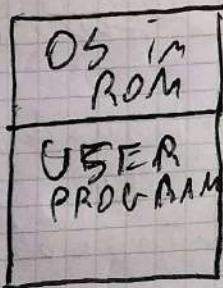
Nei sistemi moderni i processori sono sintesi nelle gerarchie della memoria tramite le astrazioni che diventano via via più complesse.

I primi modelli erano senza astrazione  
I programmi lavoravano direttamente agli indirizzi fissi il che rendeva molto complicato fare multiprogrammazione

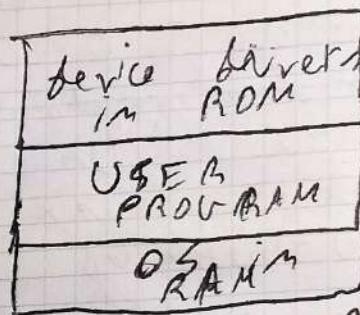
Ecco tre vari modelli:



OS nello spazio inferiore della RAM



OS in cima nella ROM



device drivers in ROM e il resto in RAM

## Multiprogrammazione senza astrazione

Montiamo già processi in memoria partizionando la RAM

E' presentato un problema risolvibile con la

Rilocazione

Compilazione

Rilocazione ← Statica :

E' fatta una "traduzione" degli indirizzi logici in quelli fissi per evitare collisioni con lettura e scrittura.

La traduzione avviene semplicemente tenendo conto delle celle a cui inizia P.

Svantaggio: il boot viene rallentato.

Se per un qualche motivo un codice genera un indirizzo fuori dalla porzione abbiamos pensare a un meccanismo di protezione della memoria.

- **lock & key**: Questo meccanismo fa da interfaccia tra CPU che genera indirizzi e memoria che deve essere letta / scritta.  
Per ogni blocco abbiamo un numero che è una chiave.

La CPU avrà una chiave dentro a un registro speciale. L'MMU controlla se la chiave della CPU coincide con la chiave del blocco e in caso contrario nega l'accesso.

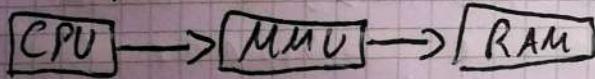
Se ho un process che scrive già blocchi, questi blocchi hanno la stessa chiave.

Nel context switch la CPU cambia la sua chiave.

## SPAZIO DEGLI INDIRIZZI

E' un'estrazione da memoria che già avere fatto da un process. (run-time)

Faremo una tabacazione dinamica. Mentre Registro base e Registro limite.

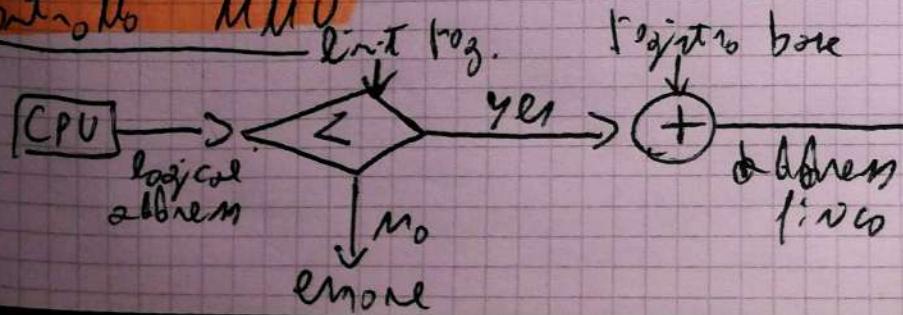


La CPU genera gli indirizzi logici, tratti da MMU usando i due registri.

In sistemi meno evoluti non c'è l'MMU e fare intermedio.

- Registro base: indirizzo fisico inizio spazio
- Registro limite: "aggiustato" porzione.

## Controllo MMU



Registro base

indir. fisico



Context switch:

MMU mappa i due registri

## Swapping

La multi programmazione è limitata dalla dimensione della RAM.

Come soluzione adottiamo lo swapping:

Se non c'è spazio e voglio inserire un nuovo programma; un altro programma lo sposta su disco e inserisco quello nuovo per l'esecuzione. A un certo punto quello su disco lo swapperemo con uno in RAM.

Permettiamo un giri elevato parallelismo.  
Il sistema deve essere già sofisticato se vogliamo bene il processo da swappare (ad esempio un processo non interattivo).

Lo swapping è fatto dalla Magger che è la scheda del terminal.

- bus terminal: alta frequenza (sostituisce process)
- porto terminale: media frequenza

Bisogna fare attenzione a non appiattire programma strettamente in etica a I/O.

Anche non inciderà a scrittura / connessione della memoria (swap i due programmi e avremo la scrittura nel programma sbagliato).

↑  
Questi problemi sono legati

I/O gestimenti brughi.

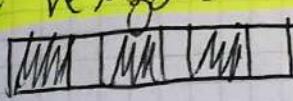
Come elocare?

- bin fine: semplice ma crea sprechi / è limitante
- bin dinamico: bin variabile che può crescere.

Se ho due processi zia clint e uno vuole già spazio forni o mettere P1 su disco e fornisce P2 o mettere P2 su disco o sposta un po' P2 (non è giusto). Poi essere costretto come problema. Altre forme di spreco sono legate alla frammentazione

## La frammentazione può essere:

- interna: spazio legato a spazio (eccessivo) fatto per una ridetta (allocare 100 mezzi ma 70 bastano necessari)
- esterna: spazio fatto di fuori dalle partizioni si vengono a creare buchi nella memoria tra processi



Gran problema: ho 3 buchi da 10 mezzi e devo mettere un processo da 30 mezzi.

Lo spazio ci serve ma non serve

Possibile soluzione: memory compaction

Compattione: buchi in un unico buco. Ha un conto notevole

## Gestione dell'allocation

La memoria la amministra per blocchi.

Stiamo facendo una approssimazione per facilitare la gestione. Conseguenza: Spazio

Più è grande il blocco più è alto statisticamente lo spazio. Lo spazio è al più un blocco di memoria (l'ultimo), almeno  $\Omega$  è statisticamente  $\frac{1}{2}$  di blocchi

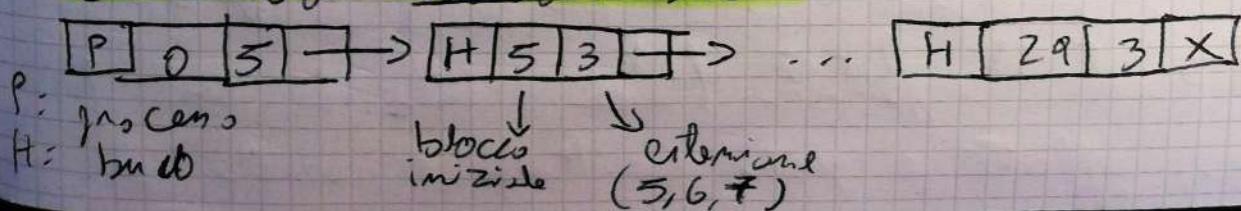
Non facciamo blocchi comunque troppo piccoli perché dovremmo usare struttura dati per tenere traccia (se ci sono eccessivi blocchi la struttura è lenta)

## 1a Struttura: BIT MAP

Tener traccia dello stato di allocazione della memoria. La bit map è di bin fine, essa tiene sotto conto dell'informazione blocco allocato / NON allocato

## 2a Struttura LISTE

Criamo una lista che contiene i blocchi liberi o i ~~blocchi~~ blocchi



La lista è ordinata per indirizzo del 10 blocchi e la sommatoria linkata quindi la gestione è facile e efficiente.

Il doppio linking permette una più facile coalescenza ovvero far dare blocchi liberi contigui (bere tutti di vicini diventano un solo bus)

La coalescenza fibra la lista e permette di avere buchi grossi invece di tanti piccoli buchi

La coalescenza non è disponibile perché:

- Quando un processo deve muoversi e deve togliersi da quella lista ci sono facilmente con un gestore nel PCB
- il doppio linking la facilita.

Ricerca di una sequenza libera / buco

- first fit: ricerca del 10 blocchi sufficientemente grande.

N.B.: se lunga della lista trovo la frammentazione multiplazionaria

- next fit: applica la ricerca come first fit però si ricorda dove si è fermato alla precedente ricerca e ripete da lì

- best fit: scelgo il blocco sufficientemente grande ma si obbliga la richiesta ma che sia il più piccolo possibile.

E' lento dato che dobbiamo cercare (quasi sempre)

l'intera lista, ma spedito (dove molti piccoli buchi).

- Worst fit: sceglie il blocco peggiore (il più grosso della lista)

Sperimentalmente si è visto che il migliore algoritmo è il first fit

Ottimizzazione: Creare liste separate  
voci degli indirizzi ordinata per indirizzo  
• Ho due liste separate ↗  
blocki liberi Ordinata per dim.  
L'efficienza sono ottime (rispetto per gli algoritmi  
di ricerca) ma la soluzione diventa difficile

## MEMORIA VIRTUALE

L'allocazione contigua da opere limitazioni implementative  
il concetto di memoria virtuale è affumicato  
l'allocazione NON contigua della memoria fisica  
per evitare questo

Lo spazio di indirizzamento diventa virtuale, ovvero  
concretamente i vari pezzi di pagine sono contigui  
ma il loro stato non devono stare per essere vicini  
nella memoria fisica. Ho addirittura la possibilità  
di avere pagine in RAM e pagine in Disco  
contemporaneamente

Se ho tante pagine ricade la struttura dati che le  
gestisce è troppo grossa, e se ho poche pagine  
quale la struttura è piccola ma la modellazione  
dello spazio in pagine viene meno e c'è frammentazione (int)  
La pagina è un taglio di binaria fissa preselezione  
dal SO e l'HW è progettato per gestire

Soltanto (e altri pochi SO) usano pagine di binaria variabile

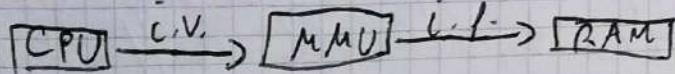
Anche la memoria fisica è suddivisa in pezzi  
detti frame, le cui dimensioni coincidono alle binarie  
delle pagine.

Il fatto che le pagine contigui nello spazio non lo  
siano nel spazio in memoria risolve il problema  
della frammentazione esterna.

E' implementato in modo implicito la protezione  
di memoria. Un P1 non ha modo di operare  
indirizzi di un P2

## Paginazione

E' gestita dall'MMU. Si occupa di fare la traduzione indirizzo virtuale - indirizzo fisico



L'MMU dato un i.v. fa la traduzione verso un i.f. che sta o in RAM o in disco (page fault)

**Page-fault:** Richiesta di una page NON in RAM  
Nel caso di page-fault poi si esegue un procedimento per riportare la pagina in RAM (2 volte anche facendo swap con una pagina già presente)

- Nell'HW ci sono il bit di presenza per sapere se una pagina è presente o meno
- Lo spazio di memoria virtuale è sempre maggiore dello spazio fisico di RAM.

Come avviene la traduzione i.v.  $\rightarrow$  i.f.?

- MMU ha come input un numero che c'è l'indirizzo di una word (nella virtuale)
- Dato questo i.v. ne invieremo la pagina che lo contiene

$$\text{i.v. / size} = \leftarrow \# \text{P.V.}$$

offset: Dimensione word presa e da word della pagina

Avevamo il #P.V. e il bit per sapere se sta in RAM o no (page fault)

Se siamo nel 1o caso

- #frame • 1 byte con questo oggetto otengo l'indirizzo fisico della 1a word della frame
- Il #frame rappresenta inoltre da cosa dipende l'indirizzo offset e questo risultato è chiamato offsetto

Il procedimento rende i bit da questa strategia ma  
è ottimizzato infatti la tabella delle pagine e le  
valenze del 2.

Se dunque C.V. ha size =  $2^x$  o shift a dx  
C.V.  $\rightarrow$  Volte è la stessa cosa ma le  
prestazioni sono migliori.

Dunque che l'immagine è  $1 \times 2^x$  è #P.V., quello che  
dice a dx è l'offset

i.v. #P.V offset

Ora faccio il moltiplicare per la size ma basta lo  
shift a  $1 \times$  di  $x$  volte.

Ora devo sommare l'offset ma se faccio  
per OR (bit a bit) offset ottengo lo stesso  
risultato. (Perchè c'è il risultato all'og. precedente)

## Tabelle delle pagine

Ne abbiamo 1 per processo, ~~indica la sua dimensione come  $2^m$~~

- dim spazio virtuale:  $2^m$   $\Rightarrow$  Numero di pagine nella Tabella  $2^{m-n}$
- dim ~~tabella~~ pagina:  $2^n$

offset: n bit meno significativi dell'indirizzo

- per ogni indirizzo free:  $2^t$   $\rightarrow$  abbiamo  $2^{t-n}$  frame

Ng  $M > t$  SEMPRE

## Esercizio 1 RAM = ?

$$VM = 5 \text{ MB} = 2^{22} \text{ byte}$$

c.v. è a  $2^2$  bit

$$N.PV = 2^{m-n}$$

$\exists 2^{13}$  voci nella tabella pagine

# frame è a 8 bit

$$2^{13} = \frac{2^{22}}{2^M}$$

$$2^M = \frac{2^{22}}{2^{13}} \rightarrow M = 9$$

Il numero di frame è  $2^8$   
RAM =  $2^8 \cdot 2^9 = 128 \text{ KB}$ . ( $2^m$  page e frame coincidono)

Potremmo ragionare così

$\hookrightarrow MB = 2^{22}$  byte  $\rightarrow$  i.v.

13	9
#log	offset

22 bit

$$8+9=17$$

i.p.

↓	8 bit	9
#frame		offset

? bit

~~Il i.p. è 2<sup>17</sup> bit~~

Dunque...  $2^{17}$  byte = 128 KB

## Esercizio 2

VM = 1 GB

#PV a 22 bit

#i.p. a 20 bit

Quante frame?

1 GB =  $2^{30}$  byte

$30 \rightarrow m$

i.v.

22	8
#PV	offset

Ho  $2^{12}$  frame

20 bit	12	8
#frame		offset

## Voci di tabella

La tabella tiene anche altre informazioni oltre al numero di frame e il bit di presenza.

• campo protezione: specifica cosa è consentito fare sulla pagina (lettura / scrittura e esecuzione)

• bit di modifica (dirty bit): è un flag che specifica se la pagina è stata modificata. Per questo bit va in controllo alla pulizia delle pagine. Ogni incaricazione le due cose in RAM e binario (se c'è uno).

• bit di referenzamento: è 0 se la pagina non è stata letta, 1 altrimenti.

E' previsto periodicamente l'azzeraamento, è detto le questo bit per gli algoritmi di sostituzione delle righe. E' detto dell'HW e si utilizza del SO per

- bit per disponibilità il meccanismo di uno cache viene disabilitato per le pagine che fanno mappatura per le porte di I/O

- bit di validità / allocazione

I SO tiene traccia quindi delle pagine slave o meno, genera errori fatal nel caso si voglia accedere a pagine col bit disabilitato.

### Tabelle di frame

Il SO tiene traccia dello stato di ogni frame tramite questo struttura dati.

Tiene tutte info per ogni frame

- Se è libero / occupato
- Occupato da chi?

La tabella è consultata in due momenti:

- creazione nuovo processo (per creare la task)
- processo chiede di estrarre nuove pagine

### Progettare una tabella delle pagine

Bisogna badare a  $\begin{cases} \text{Velocità} \\ \text{Dimensione} \end{cases}$

#### 1a strategia

Avere un numero sufficiente di registri in cui mettere l'intera tabella. Se la tabella è grande è infatti belli.

#### 2a caricare interamente in memoria la tabella

e usare il PTBR.

Il PTBR è un registro speciale che punta alla tabella. Egli informa l'MMU su dove sta la tabella. Il contest switch diventa molto veloce e semplice: basta modificare PTBR

Strategia: Per prelevare un dato dev'essere due accessi in memoria (1 accesso per consultare la tabella e 1 per consultare la word desiderata)  
Il throughput di accesso alla memoria è mezza

Ottimizziamo la soluzione con la TLB

La TLB è un insieme HW che riporta come una sorta di cache relativa alle tabella delle pagine e verifica (se c'è hit) la traduzione

È ma sta dentro all'MMU, ma un numero ridotto di registri per tenere le rispettive voci

- #P.V.      bit validità della voce in TLB      Codice protezione
- dirty bit      • #frame

Se la TLB contiene le info (TLB hit),  
viamo la TLB per ottenere #frame in modo istantaneo e dovrà fare un solo accesso in memoria.

A titolo di esempio c'è TLB miss che ci porta ad usare la page Table (2° accesso)

La TLB evita il 2° accesso in memoria molte volte

Un TLB miss provoca anche la riabilitazione della relativa voce nella TLB

La ricerca nella TLB è molto veloce essendo parallelizzata in HW

Anche qui ci sono algoritmi per ricoprire/mappare pagine nella TLB

Nel caso di context switch poniamo di fare multamente (flush) ovvero il bit di validità diventa 0 in tutte le voci oppure:

- Usare ASID come campo in più nella voce come chiave con #PV ed evito multamento

ASID: address-space id è un id della page  
address associato alla pagina  
come offset intero c'è la possibilità di avere  
Voci vincolante

Voci ~~della pagina~~ della TLB "privilegiate" ovvero che  
non vengono buttate via dalla TLB durante gli  
algoritmi di mapping

- La TLB è una sorta di cache per le pagine  
della Tabella

### Effective time access EAT

tempo accesso memoria: 100 ms Energia  
tempo accesso TLB : 20 ms

Se ne deduce il tempo effettivo di accesso sarà

• 120 ms per TLB hit

• 220 ms per TLB miss

Averemo TLB hit ratio di 80% (percentuale successo)

$$t_{\text{m. a.}} = 0.8 \cdot 120 \text{ ms} + 0.2 \cdot 220 \text{ ms} = 150 \text{ ms}$$

### In generale

tempo accesso memoria =  $\alpha$

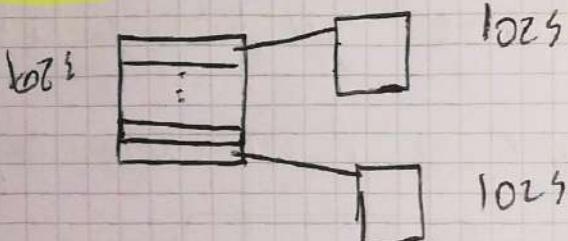
tempo accesso TLB =  $\beta$

TLB hit ratio =  $\varepsilon$

$$\text{EAT} = \varepsilon(\alpha + \beta) + (1 - \varepsilon)(2\alpha + \beta)$$

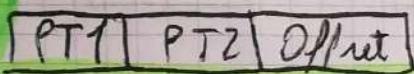
## Problema di m tabella

Abbiamo una tabella già o non sparsa che vuole essere in contigui, bisogna trovare la soluzione.  
Non manteniamo l'intera tabella in memoria, creiamo le tabelle multi livello che quindi fanno paginazione gerarchica.



Creiamo la page table 1 che materialmente è un gruppo di tabelle (le page table 2)

Un indirizzo sarà di questa forma



I bit in base a come saremo distribuiti saranno

usati per la traduzione.

Vantaggio: È una struttura dinamica, le porzioni possono essere propagiate nella memoria invece che stare tutte contigue. Inoltre le tabelle 2 non è detto che siano tutte in memoria. Dunque non abbiamo vincoli sulla contiguità e si risparmia memoria (se ci sono 1024 gruppi non è detto non abbiano tutti)

Svantaggio: La struttura a 2 livelli potrebbe cominciare ad invadere 3 accessi alla memoria. La TLB è applicabile anche qui e assume valore ancora più elevato (la risparmio 2 accessi).

La grazie a ciò può aumentare gli accessi ancora di un livello (rendendo estensione)

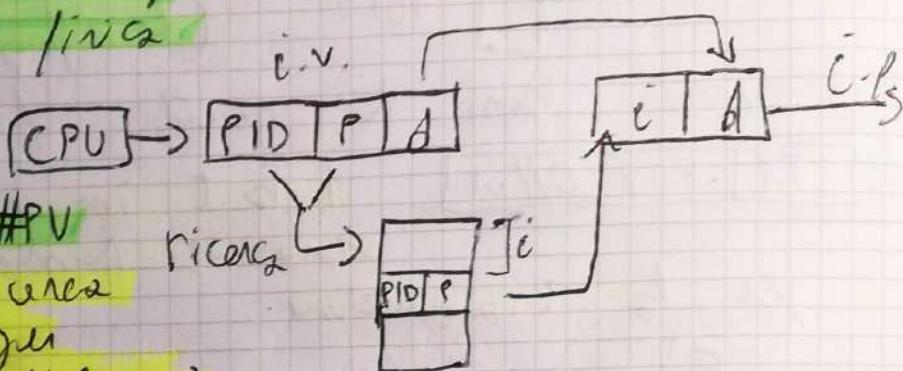
Il bilancio tra PT1 e PT2 può essere diversamente progettato. Potrei usare un tot di bit per PT1 e un altro tot per PT2

Sistema alternativo al multilivello

Tabelle delle pagine invertite

Ce n'è 1 unica, non 1 per ogni processo  
Abbiano una voce per frame fisso, Ogni voce ha  
• id processo • pagina virtuale  
Il numero di frame non serve perché sarebbe  
l'indice delle tavole  
Le tavole si compone come se fosse una rappresentazione  
della memoria fisica

Trobazione



Tramite PID e #PV

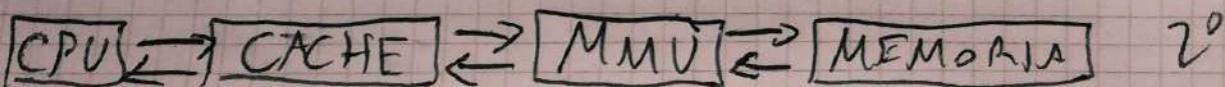
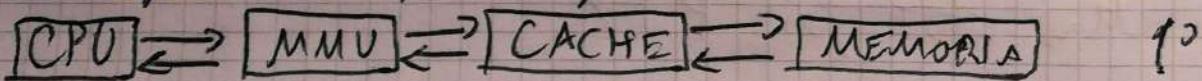
è fatta la ricerca  
nella Tavola per  
trovare i (#frame)

che meno con l'offset ci da l'i.p.  
Se la ricerca è senza successo c'è un page fault  
La ricerca può essere lenta, si usa quindi il  
comando di Hashing su PID e #PV, in più  
per velocizzare ancora di più usiamo le TLB  
Scriviamo sullora le tavole chiavi (per nome)?

Sì, dato che la tavola invertita fa  
riferimento allo state della pagina in RAM, queste  
tavole ci servono per gestire i page fault.

CACHE MEMORIA VS MEMORIA VIRTUALE

La cache della memoria può essere solo  
o prima o dopo la MMU



Nel 1° caso diciamo che i bontà degli indirizzi  
fini è appunto i nuovi quelli.  
Il cacheing è poco efficace e l'MMU farebbe  
di colpo di bottiglia.

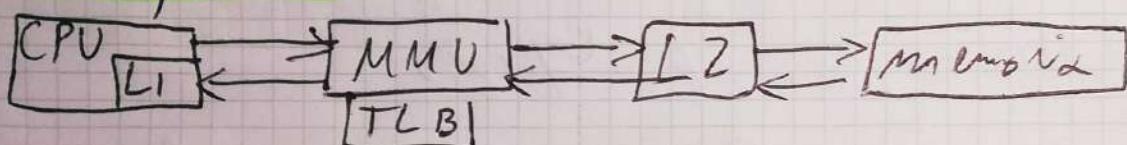
Permette però di non fare azzeraamenti in caso  
di context-switch.

Nel 2° caso diciamo che i bontà degli indirizzi  
virtuali.

L'efficacia nel cacheing è maggiore

Sono però costretti ad usare gli ASID per riconoscere  
gli azzeraamenti. Se muo una L2, scala male.

Nella pratica



## Algoritmi sostituzione pagine

In caso di pagine fault e assenza di frame liberi  
dobbiamo fare una sostituzione e bisogna  
seguire tramite appunto un algoritmo per  
pagina vittima.

La scelta è fatta con una gestione simile a  
quelle ~~per la~~ cache ponendo l'obiettivo di  
evitare pagine fault in futuro il più possibile.

## La soluzione ottimale OPT

E' una soluzione ottimale ma solo teorica poiché  
non implementabile.

Per implementarla dovremmo considerare i più  
nuovi pagine fault.

Nell'algoritmo seguiremo la pagina che viene  
mossa nel futuro più presto.

Ci permette di avere un tempo di pagina

## NRU : Not Recently Used

Uiamo statistiche provenienti dal bit b referenzamento e bit d di modifica che sono gestiti da HW e azzzerati a dovere dal SO intuitivamente vuol dire che la pagina non usata è recente.

L'algoritmo distingue 4 classi di pagine

- 0: non referenzato, non modificato
- 1: non referenzato, modificato
- 2: referenzato, non modificato
- 3: referenzato, modificato

L'algoritmo segue 2 passi: prima seleziona la classe non vista di classe già bassa per poi successivamente selezionare una pagina della classe in qualche altro modo.

L'algoritmo potrebbe lavorare in modo globale oppure guardare solo le pagine del process.

La selezione di una pagina in una classe è

- a caso
- da trovata

oppure tramite altri algoritmi.

## FIFO

• rimuove la ~~pagina~~ già vecchia

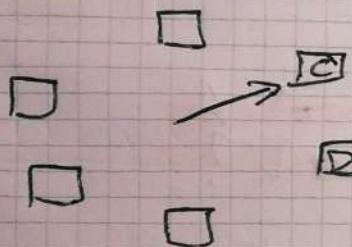
• potrebbe non essere un'idea molto buona dato che una pagina potrebbe infatti essere vecchia ma allo stesso tempo essere molto usata.

## Second Chance

Come FIFO, si butta via la pagina già recata che ha per il bit  $R$  a "non referenzato". Dunque controlliamo la testa della coda, se  $R=0$  si butta via per la sostituzione, se  $R=1$  la mettiamo in fondo alla coda e ripetiamo. Di solito quando abbiamo  $R=1$  gettiamo prima di mettere la pagina in fondo utile  $R=0$  per evitare di ripetere la pagina troppo volte.

## Algorithm clock

Implementazione della recerca che sia efficiente. Viene organizzata una coda circolare



Puntatore esterno che punta alla "testa"

Nel momento in cui C è battuta fuori, D diventa la testa.

D diventa la testa anche se C era stato esaminato e riportato.

## Algorithm LRU

Idea: Probabilmente le pagine già usate & recente lo saranno anche in futuro (masino), dunque rimoviamo le pagine meno usate & recente.

LRU: buona idea, ma è difficilmente implementabile nella sua forma rigorosa.

Si usa un rapporto HW: si tiene un contatore nella CPU e altri campi relativi nelle tavole delle pagine.

A questo punto l'LRU sarebbe implementabile per comodità dovendo fare troppi accessi in RAM.

Per evitare gli accessi eccessivi non fa un'approssimazione.  
Usando delle matrici di bit che tengono le informazioni sullo stato recente dello frame.

Se ho 4 frame la matrice sarà  $3 \times 3$ , con le righe 0, 1, 2 e 3.

Il numero di bit a 1 indica quanto è stata usata recentemente lo frame relativa a quella riga.

### Funzionamento:

Inizialmente la matrice è piena di zero.  
Quando una pagina viene letta, la riga corrispondente è negata a 1 e la colonna corrispondente è riempita a 0.

Quando devo leggere una pagina/frame da memoria in un istante chi farà?

2 approcci < frame con j=0  
frame con numero j=0 stesso  
(i j sono le pagine come un numero)

Nella matrice i 2 approcci sono equivalenti.

Ci sarà sempre una riga con 3 1 (tranne all'istante 0) e mai più meno degli 2 righe contemporaneamente con 3 1.

Il jex di una pagina è alleggerito quando un'altra pagina viene letta.

La matrice potrebbe essere fatta all'MMV, ma è proporzionale al numero di frame.

La matrice potrebbe avere un timestamp/contatore delle pagine ed evitare di accedere I/O in RAM.

## NFU : ~~non faulted user~~

LNU è impraticabile, allora un'approssimazione  
è lo software ~~counter~~.

Trovare un counter in ogni voce della tabella delle pagine, periodicamente il valore di R deve di essere azzeroato e sommato al counter.

Quando n deve fare meno swap prendiamo la pagina col counter già zero.

L'informazione è ovviamente approssimativa.

Problema: potrebbe privilegiare pagine usate molto in passato ma poco recente.

Aumenterebbe il page fault.

## AGING

Res risolve a NFU, ma risolve il problema.

Il generatore al ciclo di clock ha shift 2  $\times 6^1$   
il bit tutto a 1x diventa R.

Il generatore diventa un storia: se è 2 8 bit  
abbiamo informazioni sugli ultimi 8 usi.

La scelta è base sul valore del counter e non fa glianti falso, ma l'uso a 1x  
ha un peso maggiore.

L'idea è sempre rimuovere la pagina col counter  
già zero.

L'informazione rimane comunque approssimativa perché  
l'I ci indica che la pagina è stata usata  
ma non quanto recentemente.

## Confronto Prestazioni

Aspettativa: già che ho meno fault ho

metrika: numero di fault finendo un frame  
di frame in RAM

RAM: 3 frame

Sequenza 10 pagine:

7	0	1	2	0	3	0	5	2	3
0	3	2	1	2	0	1	7	0	1

**OPT**: algoritmo in cui mappano a priori di memoria le pagine usate nel futuro più lontano

9 fault

**FIFO**: si nega la pagina, nella RAM, già letta.

15 fault  
Il FIFO presenta l'anomalia di Belady ovvero aumentando i frame disponibili potrebbero aumentare i fault

L'anomalia attacca tutti i fault gli algoritmi imparentati con il FIFO prima anche recente chiama è clock

Anche NRU potrebbe andare in contro all'algoritmo se gli sono tutte le pagine sono nella stessa classe e la gestione interna delle classi è FIFO

**LRU**: rimanere le pagine meno usate o recente (NRU è A(ING sono approssimazioni)

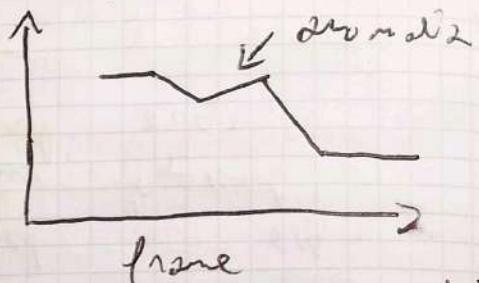
12 fault.

Non soffre dell'anomalia visto che gode di proprietà di inclusione: l'insieme delle pagine caricate avendo  $m$  frame è indoss nell'insieme che si avrebbe con  $m+1$  frame all'intante  $t$

$$B_t(m) \subseteq B_t(m+1) \quad \forall t, m$$

**NFU, A(ING**: godono delle proprietà ensemble iniziali  $\subseteq LRU$ .

Nella pratica si usa il clock con un aggiornamento che riguarda il bit di mobilità oppure l'aging



## Allocazione dei frame

Rete devant page : organizzazione per richiesta pagina.  
Un processo guarda cosa è senza pagine dunque all'inizio C'è un solo frame non pagine. Poi l'arrivo di dati come file, l'arrivo di dati, l'arrivo di dati, ecc.  
Avanti, frame sengono a un processo?

- Minimo strutturale: si sceglie il minimo di pagine in cui sono contenute le strutture con varie specifiche che servono al processo per la esecuzione. Il minimo dipende dall'architettura. Sotto questo avendo l'esecuzione è impossibile.
- Massimo: massima libertà

## Strategie allocazione

- Eguali: Ogni processo ha un RAM dedicato. Può avere problemi, chi non ha bisogno di quelle pagine che ha troppe, chi invece ha bisogno di più pagine solle.
  - Proporzionale: diamo RAM in base alla quantità del processo (valore che varia). Supponiamo di avere il processo i di dimensione  $S_i$ . Diammo  $z_i = S_i / S \times m$  frame.
  - M1: frame disponibili per i processi.
  - $S$ : somma di tutti i frame utili;  $S = \sum z_i$ . L'allocazione dunque è proporzionale alla taglia  $S_i$  e dipende dal livello di multi-programmazione.  $S_i$  varia  $\rightarrow$  anche  $z_i$ . Il istante tratta i processi senza bloccare per quelle quantità.
- Allocazione per priorità: dare più frame se il processo ha priorità alta.

Allocazione locale: la vittima fa scegliere è una pagina dello stesso processo che deve inserire una pagina.

Allocazione globale: le pagine vengono tutte dalla stessa memoria.

Quella che è una scelta pratica è la globale.

Cos'è succede se ci sono pochi frame per un processo?

- Sotto il minimo disponibile: l'esecuzione non è possibile, si sospende il processo e c'è la swaping in blocco.
- poco spazio: thrashing

Il thrashing consiste nell'avere il minimo tempo (o poco spazio) di frame zero ~~ma~~ troppo picchi il numero di page fault sarebbe elevatissimo.

Tutti i processi in thrashing: ritardi in esecuzione (eccessivo livello di multi programmazione)

Di solito quando il sistema non rende conto che c'è un processo in thrashing lo entra regolarmente frame.

Bisognerebbe trovare un modo per dare le frame necessarie per le necessità del processo.

località: inizio di località utili per un processo in un certo istante.

La località è legata alle necessità del processo in un certo istante.

L'idea potrebbe essere tracciare in qualche modo la località e dare un numero di frame minimo per soddisfare la quantità di località in un certo istante.

## Working Set

Ne manteniamo uno per processo.

Stavolta un  $\Delta$ , che indica il numero di ultime occorrenze in RAM, il working set è l'insieme delle pagine usate negli ultimi  $\Delta$  accessi.

Il working set si comporta come un'approssimazione dell'insieme delle località.

La scelta di  $\Delta$  è fondamentale

tropo: eccesso.

Il working set non ha cardinalità

poco: meno delle località

A cosa serve? A cogliere un numero più preciso e quanto più vicino da aggiungere a un processo. Permette dunque la generalizzazione del tracking sia individuale che globale.

$D = \sum WSS_i$  (size del  $WS_i$ ) Permette il calcolo globale di frame che servono

$D$  più grande della memoria disponibile  $\rightarrow$  tracking  
 $D$  più piccola  $\rightarrow$  situazione più stabile

Calcolo del WS nella pratica (strumenti: jettatura, approssimazione)

- interrupt periodo
- bit R
- log con la storia di R basata su  $\Delta$

Si potrebbe velocizzare l'arrivo di un programma con le tecniche del working set model.

Un programma all'arrivo tende sempre ad avere molti fault, ma avendo pagine in memoria ma soltanto far consumo di SO quali sono le pagine che vengono usate all'arrivo per velocizzarla, basterà

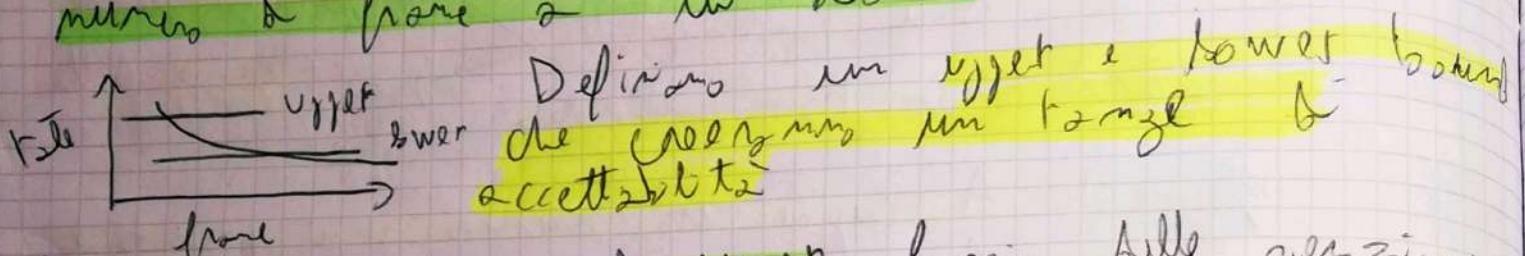
- Working set model: working set dell'arrivo di un programma
- preparazione: caricare pagine (quelle del WSM) senza prima della esecuzione del programma.

## Page Fault Frequency

Modello più diretto per approssimare il tracking

NB: secondo il modello teorico, a meno delle 2 parole di Belady, più frame ci sono e meno fault ci saranno.

Monitoriamo la PFF del processo in relazione al numero di frame e ai due debiti

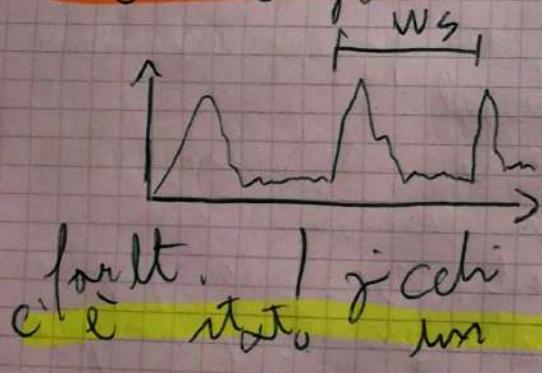


Se mi trovo sopra al upper faccio delle operazioni per sintetizzare il processo (è in tracking). Infatti il mio WS sarebbe già spinto nelle forme necessarie.

Se sto sotto al lower bound mi tendo contro che il processo sta venendo destruito. Un numero di frame già spinto dal WS, se ci fossero altri processi in tracking potrei pensare di fare da questo processo per destruirlo e farlo e gli troverò un equilibrio.

Sistema in vista classica: tutti sopra al upper

grafico che mostra il comportamento del rate nel tempo.



I cicli sono dovuti al raggruppamento di una hosta all'altra. All'inizio subito dopo la transizione aumentano i conflict. I cicli sempre ci fanno capire dove è stato un raggruppamento di località.

## Pulizia

Il meccanismo di gestione page fault deve essere più efficiente se ci sono frame liberi disponibili poiché non saremmo in grado di eseguire l'algoritmo di swapping.

Pagine fotonom: processo di sistema che si occupa di tenere, quando può, una pool di frame liberi. Si occupa di ~~fare~~ anticipare la scelta della eventuale pagina da buttare.

• Seleziona, libra e in caso pulisce la pagina. Sostanzialmente attiva anticipando l'esecuzione dell'algoritmo di scelta in momenti in cui la CPU non è particolarmente occupata.

Caso sfondando: il fotonom ha scelto una pagina che sta per essere usata.

In realtà è abbastanza gestibile senza I/O grazie al rigurgito dalla pool dei frame liberi. La riguttatura è comune sia in Linux che Windows.

## Dimensione della pagina

Bisogna trovare un buon compromesso per la scelta della dimensione della pagina.

Pagina grande:

- tabella delle pagine più piccola
- ridotto trasferimento I/O (avremo più contiguità in disco delle Word)
- minimizza i page fault

Pagina piccola:

- minore frammentazione interna
- WS definito medio: l'approssimazione diventa più precisa (meno i page fault)

Ha una certa relazione con la dimensione del blocco in disco (di solito è un multiplo).

## Pagine condivise

I processi possono condividere in vari modi alcune pagine (es: le pagine del cache)

Bisogna fare attenzione alle pagine di

- solo lettura : condivisione semplice

• lettura/scrittura: si fa IPC con memoria condivisa  
Il modello delle pagine condivise è implementabile  
sulle tabelle Caché che sono multilivello in modo  
semplice ed efficiente

## Dificoltà:

• Cache: Se abbiamo una cache basata su  
indirizzi virtuali potremmo avere problemi di  
aliasing (la stessa linea pica potrebbe avere  
nomi differenti nella memoria virtuale dei processi che  
la condiviscono)

Il problema nasce anche grazie agli ASID: Usando  
gli ASID potrei avere linee di cache diverse  
che fanno allo stesso riferimento alla stessa Word pica

Soluzioni:

- disattivare coherency per pagine condivise
- fare ricerca in cache basata su indir. e tag pica

Nella cache ogni linea assume come chiave  
(ASID, C.V.) Si fa la ricerca sulla cache e  
si trovano dei candidati (col tag pico), nel mentre  
l'MMU fa ricerca sullo TLB e vede se c'è una  
corrispondenza

Se c'è corrispondenza tra i candidati con l'i.f.  
dato in output dalla TLB si trovano dei duplicati

Dunque la CI e l'MMU collaborano in parallelo  
e trovano i duplicati

## + tabella invertita delle pagine

No single one: dovremo fare un'elaborazione della tabella in corso & contesto switch o page fault. Il SO sa tutte le etichette che ha una pagina, però riesce a risolvere i problemi & algoritmo di swap in corso rivede incontro all'evento per cui cerca un C.V. nella tabella e non c'è più in realtà il link a chi lo riferisce ed (com'è in altro C.V.) il SO risolve il problema con l'elaborazione della tabella (cambiare l'etichetta). La struttura è moltiplicata per avere un ultimo C.V. insomma e così evitiamo l'I/O.

No interi multi one: la cosa diventa problematicamente ingestibile. L'algoritmo non è gestibile efficientemente.

L'unico "soluzione" sarebbe avere delle tecniche tabella con corrispondenza multi-a-uno.

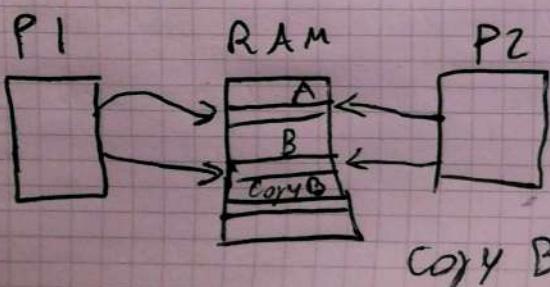
## Copy-on-write

E' una possibile ottimizzazione per la gestione della condivisione di pagine.

Idea: condividere in lettura/scrittura una pagina (com'è dei dati in corso) anche senza la esecuto richieste dei processi.

Se uno dei due processi tenta di scrivere la pagina condivisa, essa è copiata in un altro frame (com'è moltiplicata).

Il secondo processo finisce con la vecchia pagina che non ha subito moltipliche.



P1 scrive in B, allora si crea copy B, e questo punto la moltiplica verà la propria pagina in copy B. P1 continuerà in copy B e P2 in B.

Sostanzialmente questo meccanismo riguarda  
RAM (almeno temporaneamente)

Nelle protz le frame cache hanno il livello  
di contenuto da P1 e P2, il SO nel caso in  
posta richiede di scattare se ne accorgono e lancia  
il meccanismo copy-on-write  
Dopo il meccanismo P1 scrive i primi WR in  
Copy B e P2 scrive i primi WR in B.

## Ottimizzazione zero-fall-on-Bernard

ibex base: vengono ellocate le nuove pagine direttamente nel disco

Ei potrebbero, dopo una richiesta della pagina /frame, fare spazio frame al processo.

Ei consegnano pagina "vusto" ovvero con byte null.

Il gesso di memoria che diamo sarà avuto un precedente proprietario, dunque il gesso è "speso".  
Data questa proprietà si deve gestire

- **scorrimento** (per non violare l'istruzione del "virtuale")
- **ricchezza** (E.g.: se la password di un galosso era in quel gesso?)

Trasformare una pagina in vuoto ha un costo notevole.

Il SO gestisce il tutto con delle diverse tecniche

- **pool di pagine vuote** (rossi)
- **copy-on-write** in una read only static zero pool

In queste tecniche una frame libra viene sempre tenuta vuota e usata come stampa per qualsiasi operazione pagina VM.

E' una pagina comune da tutti i processi.

Sostanzialmente il process crea molte pagine che

Fanno riferimento alla pagina vuota.  
Quando provo a scaricare scritto la copy da write  
Inoltrandosi una parte libera della RAM non ci è  
stata coi altri le ROSEG.  
Ovviamente se apprendo ci sono frame libere, allora  
scritto lo mapping.  
L'allocazione funziona nel caso ci sono frame libere  
Avendo già risolto.

Le altre tecniche consiste nel tenere frame vuoti.  
Frame liberi vuoti. Si utilizzano frame nei  
momenti in cui non sta venendo fatto altro lavoro  
e si sposta il meccanismo di DMA.  
E' un motivo spesso in WINDOWS 8 e un  
processo con priorità 0 (minima).

## Librerie condivise e file magazzinati

Più processi temono ad usare funzioni di codice  
che stanno nelle librerie.

Come si "toca" il codice delle librerie?

- **Linking statico:** si include il codice (quello del  
renne) in fase di linking (pre compilazione)

- **Linking binarico:** collegamento e caricamento a  
è il più diffuso. Funzione

Sono diversamente i risultati più possibili il linking

A funzione viene caricata la libreria in memoria  
Le librerie avrebbero ovviamente girato in sola  
lettura.

Se una libreria è caricata in RAM, è vista da  
tutti i processi in condivisione (non è di caricata  
in RAM tante volte)

Ogni processo ha priorità alle routine della libreria

- Si risparmia lavoro

- Sviluppo e aggiornamento semplificati.

## File mappati

Lo mappato mappa le posizioni  
e consente comunicazione tra processi

- modello di interazione con i file (è un modello alternativo al classico I/O)

Idea: mappa il contenuto del file nello spazio di indirizzamento (nella RAM) poi potrà anche su memoria contigua

Dal punto di vista logico viene presa una porzione dello spazio di indirizzamento e fatto coincidere col file. Mappata ≠ caricamento

Il caricamento è rimandato.

Afferra il processo provo a fare una fetch if SO provvede col caricamento in RAM del blocco del file che il processo ha provato a usare.

Con questa tecnica stiamo rimandando il più possibile l'I/O

La base è applicare anche in cache, se il processo provo a modificare il contenuto all'inizio fa modifica ovverà in RAM e poi swerà in seguito la sincronizzazione tra la cache in RAM e la cache in disco.

Con questa tecnica si possono gestire file enormi di fatto uscendo molti pezzi

In questo modello c'è la possibilità di combinare il blocco del file può essere meno in cache (come combina)

Il concetto di mappato gestire estremamente:

- libere combinare (pezzi di due mappati in memoria)
- caricamento cache eseguibile
- caricamento dati statici

Se questi campi non prevedono zettabyte, quei dati potrebbero combiniarsi le pagine relative

## Allocazione memoria per il Kernel

Come si gestisce la memoria di lavoro del SO?

Problema dell'uso e della gestione = se il SO implementa le astrazioni, già a sua volta userà le astrazioni?

Sì e no

La paginazione non può essere fatta per garantisce contatto sul SO: Troppo Over Head, non può avere un rapporto se lo implementando il supporto.

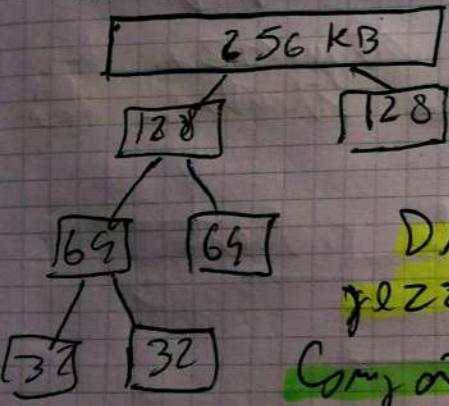
Nella pratica il SO moderno ha un approccio misto ovvero che prevede memoria contigua con qualche accorgimento per ottimizzazione.

Mentre nell'allocare i processi esistenti viene la paginazione che ci porta alla frammentazione interna, nell'allocare processi del Kernel puntiamo a frammentazione interna minima.

### La paginazione

Buddy System: ottiene uno spazio contiguo e massimo per richieste del SO. Di solito è un pezzo con una dimensione pari a una potenza di 2.

Questo spazio èusto per le richieste che arrivano dal SO stesso, ogni richiesta ha una sua taglia



Se arriva una richiesta di 25 KB, seppure dallo stesso di 2 sufficientemente grande è 32.

Dunque ripartiamo lo spazio per scegli il pezzo già ricavato dalla richiesta.

Consente spazi (framme interne) ma è semplice e permette la coalescenza delle rotture quando libere da "unire".

E' possibile creare anche frammentazione esterna.

## Slab allocator 2<sup>a</sup> soluzione

• no preco  
• efficiente

Permette di gestire tramite pezzi di RAM le richieste del SO.

Una richiesta del SO si può analizzare o non e individuare che tipo di struttura dati è opportuna.

Dalla richiesta poi che importa è la dimensione della richiesta e individuare le tavolino le strutture dati utili per il SO in base alle sue richieste.

Possiamo immaginare ad esempio che il mio SO mi 3 tipi di struttura: 1KB, 3KB, 12KB

Per ognuna di queste dim è creata una cache (è una struttura dati che contiene slab)

Slab: sequenza di frame (contigui)

Come può uno slab appartenere a una cache?

- framme contigue
- La mia dim è un multiblock della dimensione di due della cache e multiblock della dim della pagina

Uno slab può essere visto come una sequenza di libri a contenere ogni libro struttura dati relativa alla cache in uso.

Lo slab è usato per soddisfare le richieste del Kernel (che hanno dim fisse).

Come si usa lo slab?

Se mi individua uno libro e poi si usa un taglio di quelli slab già la richiesta

- Non c'è frammentazione

La gestione dello slab prevede lo stato vuoto / pieno / partiale

La gestione è dinamica: così cache può aumentare o diminuire gli slab disponibili

Lo slab allocato è oppribile nel Kernel in quanto  
soprattutto a priori le taglie delle somme richieste dal  
SO. ~~E~~ E' impossibile per i processi utente

## Segmentazione

Come vede l'utente la memoria?

- stessa dimensione di byte
- collezione di oggetti con dimensione variabile

## Alternativa: segmentazione

- viene sconsigliata la logica visione dell'utente
- si crea un segmento per ogni oggetto, programma, stack libreria.

Segmentiamo la memoria per questi oggetti

Ogni segmento contiene dei dati: ognigenere, relativi allo stesso "contesto"

Ogni segmento ha dimensione variabile.

Dunque la memoria è vista come una collezione di segmenti dove ogni segmento è una collezione di dati ognigenere

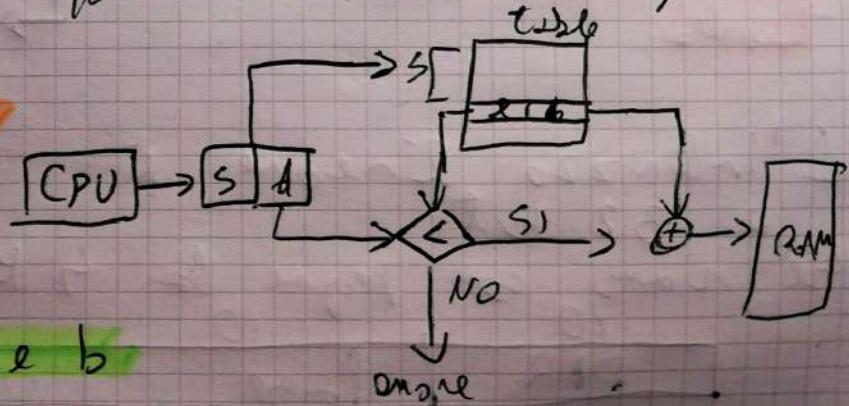
Una word è adesso individuata da una coppia di indirizzi, detta indirizzo bidimensionale

- indirizzo bidimensionale: (#segmento, offset)
- tabella segmenti: per la traduzione in c.p.

Permette di separare le informazioni in modo logico e fisico

Vediamo uno schema per la traduzione e controllo dell'indirizzo bidimensionale in fisico

Ogni segmento ha #S, l e b.



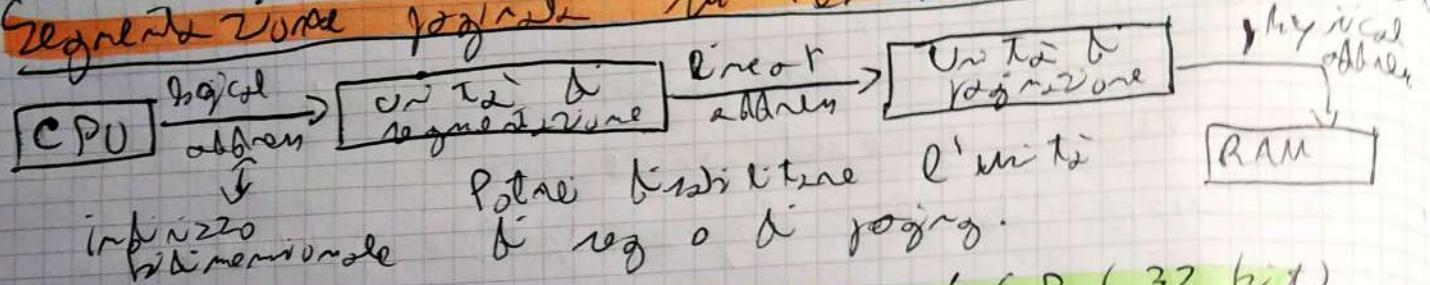
Lo Tabelle si occupa delle segmenti tra loro

# segmento → (base, limit)

Problema: frammentazione esterna

Soluzione: interno ibrido Paginazione + Segmentazione

"Segmentazione paginata" su Pentium Intel 32 bit



Ci sono circa 16000 segmenti a 4 GB (32 bit)

Uiamo due tavole dei segmenti (8000 ciascuna)

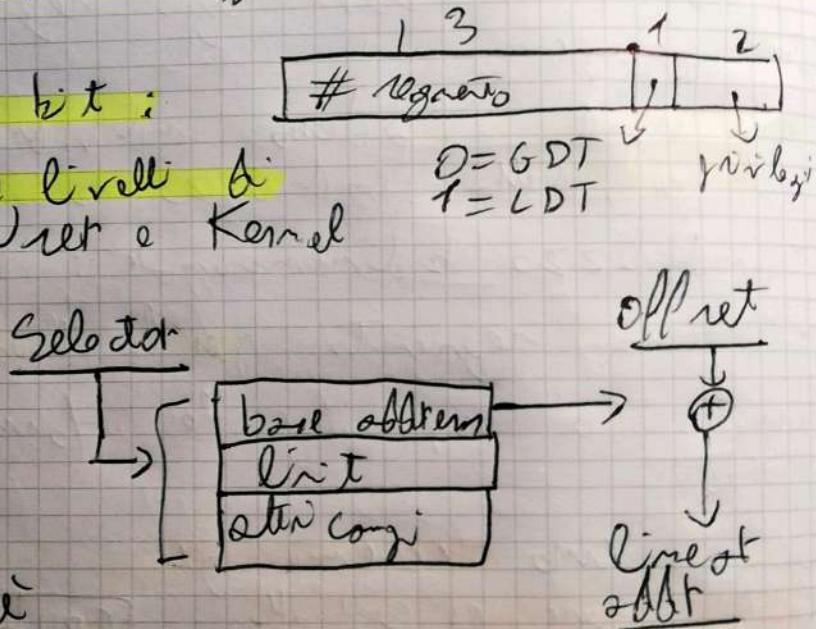
- LDT: segmenti per i programmi (una per ogni processo)
- GDT: segmenti del SO (globale)

Uiamo 6 registri per la gestione dei segmenti, sono 32 bit; solitamente perché servono appunto a selezionare segmenti specifici

I selectori sono a 16 bit:

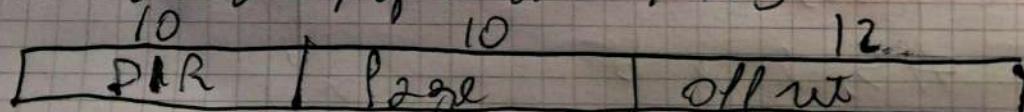
Pentium supporta 3 livelli di privilegi, non solo User e Kernel mode

Come ottenere il linear address della tabella?



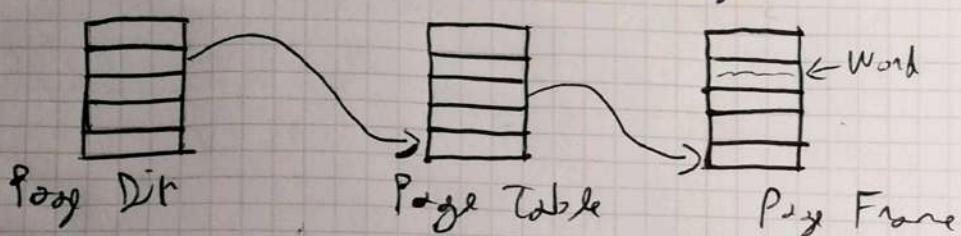
L'indirizzo lineare è a 32 bit.

Se ram con pagina da 5 KB



Nella gestione si usa una Tabella multilivello dove:

- PT1 = DIR
- PT2 = Page

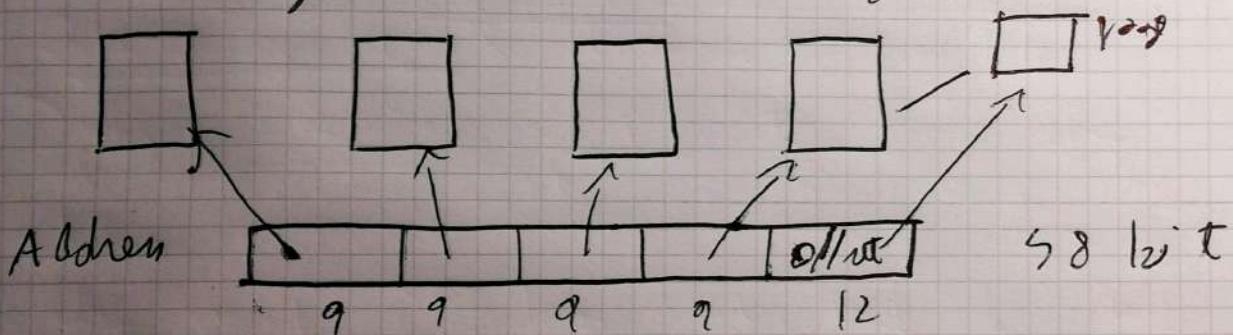


E' una Tabella a 2 livelli

In questo v'interessa la swaping solo sulle tabella di 2° livello

Usa la TLB

Nelle architetture x86-64 abbiamo una Tabella a 2 livelli e i limiti di segmentazione. Qui gli indirizzi virtuali effettivamente generati sono 58 bit (porta nulla o ignorata). Per la gestione dei livelli della Tabella



La segmentazione è limitata a massimo solo 2 livelli di protezione

Limitata all'8Gb nel SO

La memoria in LINUX è gestita così

- copy on write e read only static zero page
- file mappati (come le librerie condivise)
- allocazione dinamica Head usando brk() (syscall)
- algorithmi sostituzione pagine: i blocchi tra block e NRU
- slab allocation per la memoria riservata al Kernel

# FILE SYSTEM

E' un'organizzazione che permette di gestire bene l'organizzazione e memorizzazione dei file in modo non volitivo ma preciso (certe volte anche grande quantità di dati) mentre basta anche solo suddivisione dei file per i processi.

In sistemi: è un insieme di suddivisione in uno spazio di memoria e gestisce: organizzazione, combinazione e accesso.

Un file system è costituito da metodi di gestione e implementazione.

## Dati di:

- nome/dattura (come nominare i file)
- tipi di file (estensione) su Unix trovano i magic byte (2 byte iniziali identificano il tipo di file)
- tipi di accesso (casuale, sequenziale)
- metadati (attributi dei file)
- operazioni sui file (read, write)
- Accesso condiviso ai file (n. mano: lock)
  - Shared: più processi accedono al file (ne devono solo leggere)
  - Exclusive: un solo processo si accede (voleva scrivere)

## Mandatory vs Advisory

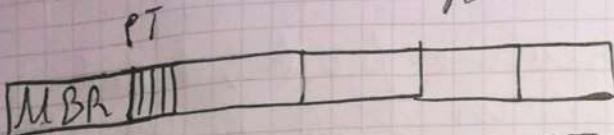
↓  
Mecanismi  
lock obbligatori

↓  
Mecanismi lock suggeriti  
(responsabilità del programmatore)

- Abitano strutture dati per la gestione dei file (es: tabelle file agenti)  
Ogni struttura è o globale o per processo

## Struttura di un file system

Standard storico: genera il Master Boot Record, suddiviso in partizioni (ognuna poi partizionata a sua volta)



La prima partizione contiene MBR che indica all'arrivo

della macchina è PT: una struttura dati molto semplice per le partizioni.

In ogni partizione troviamo blocco di boot e superblocco. Il blocco di boot potrebbe fare arrivare il SO nella partizione (sotto di sottira di MBR). Se non è presente un SO il blocco di boot diventa inutile ma è comunque presente.

Il superblocco (nella il 2°) contiene meta-info (es: tipo di file system, info ecc.)

Un'abitato partizione è il cluster: unità minima allocabile per un file

Il SO potrebbe decidere di non usare come cluster i blocchi ma ad esempio 5 blocchi

A questo punto sarebbe già possibile poi il resto della partizione può essere gestito in altri modi

Elementi riconosciuti nella partizione:

- Root bit → dicit quando per l'intrazione i file
- files and directories
- i-node: tengono traccia delle info dei file

## Partizione

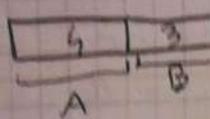
Boot block	Superblock	FREE SPACE management	i-nodes	Root bit	File area
------------	------------	-----------------------	---------	----------	-----------

Lay out moderno: GPT definito da standard EFI  
Permette di partizionare qualsiasi dispositivo in più parti

## Implementazione dei file

NB: i problemi di memorizzazione RAM sono quasi del tutto risolti.

- **Allocazione contigua**: i blocchi del file sono messi tutti contigui



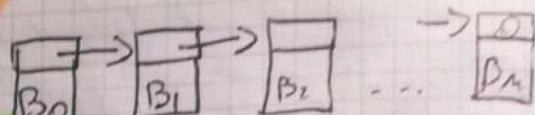
Vantaggi: gestione facile, i blocchi si trovano da un blocco a un altro nel gabinetto da un blocco a un altro.

Vengono meno (non contigui)

L'allocazione contigua sarebbe l'ideale ma non è possibile sempre (può imporre o rientrare) la tendenza è quella di allocare contigamente finché è possibile per poi ricorrere a tecniche di allocazione non contigua.

- **Allocazione con liste collegate**

File F divide in modi (block/cluster)



Vantaggio: si bypassano i problemi relativi alla allocazione contigua (frammentazione esterna). La frammentazione continua ad esistere (nell'ultimo blocco di uscita di byte per il joint next non cresce di saltare mentre che influenzano le prestazioni).

I disavallamenti sono dovuti al fatto che i dati veri e propri non sono più sotto una dimensione potenza di 2.

L'accesso sequenziale funziona abbastanza bene ma rimane difficile e inefficiente l'accesso casuale.

Mai fidiamo queste tecniche per entrare un accesso casuale lento.

## • Allocazione con liste collegate in una tabella o allocazione di file

FAT (file allocation table): è una tabella struttura del file system che indica l'indirizzo fisico della tabella che permette l'accesso casuale. Di solito avremo una lista per ~~ogni~~ file ora abbiamo una tabella FAT e back. La voce del blocco s ha il numero del next blocco per la lettura del file (-1 se è l'ultimo). Nella bit c'è anche l'indicazione per cogne se è il blocco iniziale o no. Lo standard prevede due tabelle per motivi di tenuta ai guasti (le tabelle sono uguali). Potrei mettere un valore fittizio (non -1) per cogne se un blocco non è coinvolto in nessun file. La FAT sarebbe responsabile per l'accesso casuale (sarebbe garantito portare la FAT in memoria in RAM).

Ponendo la FAT in RAM si evita se ho a che fare con dischi enormi. Si risolve molto semplicemente magariando l'impostazione di FAT.

## • Allocazione con i-nodes

i-node: struttura dati (pixels) che contiene info sull'oggetto / file

ID del file / oggetto. Contiene metadati fondamentali per la gestione del file.

L'i-node riguarda un singolo file, lo portiamo in RAM solo se il file sta in RAM.

Sulla RAM stanno solo gli i-nodes che servono.

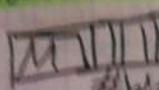
L'i-node numero identifica l'i-node.

L'i-node contiene tutti i metadati del file (tranne il nome che è nella bit).

La directory si immagina come una tabella dove ogni record è un i-node. Non nel file system FAT la voce è già pronta e non ha tutti i metadati del file.

L'i-node deve tener traccia in qualche modo dei vari pezzi del file nel disco.

L'i-node è formato da metadati e dati effettivi di block.

 Ogni voce contiene info su un blocco.  
Abbattuto se ho 10 voci, sono generate con un i-node un file di max 10 blocchi.

Se il file è grande? Estendo l'i-node

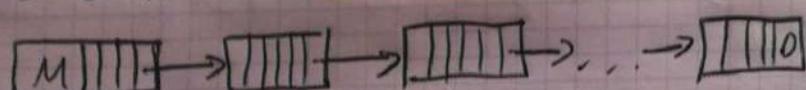
### Strategy:

- Usa l'ultima voce già presente a un blocco nel file che contiene delle voci che puntano ad altri blocchi (abbiamo fatto una estensione)
- L'estensione può avvenire più volte.  
L'ultima voce è detta Address of block of pointer.

 Ultima voce vista (se necessario) a un altro blocco di voci già puntare blocchi pieni.

Così estendiamo la lista iniziale se necessario

E ora la lista collegata



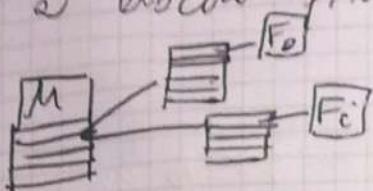
Accesso sequenziale ottimo  
Accesso casuale inefficiente

Potrei in realtà fare una struttura ad albero  
della multi-level

Ogni voce dell'albero guarda a una lista di voci che puntano a blocchi pieni.

E' un albero dove la radice è l'i-node e le voci

girava ai miei figli (le sevizie di voi che giravate  
si bloccò fino)



L'albero se il file è piccolo non  
è completo

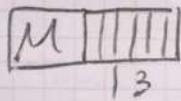
L'accesso diretto funziona più facile  
(giro era impossibile, no solo  $F_1$  doveva già farla  
rotolare da lato)

Problema: comporre 1 libro extra nel Giro  
La fibra più è quasi identica a quella delle  
tabelle delle pagine.

Potrei perdersi anche qui livelli (in overhead)  
Problema: file piccoli venrebbero gestiti meglio  
con questa tecnica (troppo spazio)

Che approccio si usa? **IBRIDO**

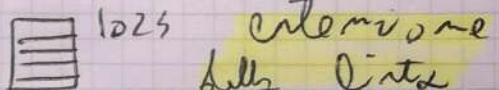
C-mode Flode



Usa le prime 10 voci  
come blocchi finiti  
(girova a blocchi finiti)

Se devo gestire file che usano giri di 10 blocchi  
le ultime 3 voci mi aiutano.

La terza linea giunta a



Ogni voce dell'ultima linea giunta a  
blocchi finiti.

1024 estensione  
della linea  
(bin di verde da file  
e blocchi)

Diciamo che la terza linea è gestita file  
di dimensione "mezza".

Se devo accedere a  $F_{10-1023}$ : overhead di 1

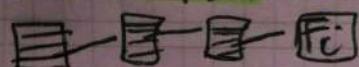
E non basta ancora usare la quarta linea che  
giunta a un albero a 2 livelli



Overhead (per accedere qui) = 2

Grazie al file gestibile è abbastanza spazio

L'ultima voce (e ovviamente) giunta a un albero a 3 livelli



Poco spazio file enormi. Overhead = 3

Supponendo che un blocco / chunk sia 5KB

File "piccolo":  $10 \times 5 \text{ KB}$

File "medio":  $(10 + 1025) \times 5 \text{ KB}$

File "medio-grande":  $(10 + 1025 + 1025^2) \times 5 \text{ KB}$

File "grande":  $(10 + 1025 + 1025^2 + 1025^3) \times 5 \text{ KB}$

L'overhead in realtà di già gestire con una qualche ottimizzazione.

(Pensavo: Overhead iniziale e fases I/O del contenuto  
vengono in RAM così se devo usare quel  
contenuto ho evitato di fare l'overhead già  
volte)

## Implementazione delle directory

Sono tipi speciali di file che hanno il compito di incapsulare altri oggetti, il loro contenuto è sostanzialmente un elenco di file.

Il contenuto di un file è un file dell'elenco di file del file system usato.

Nel caso di FAT: tutti i metadati sono memorizzati nel record insieme a nome-file e numero del 1o blocco.

Nel caso di C-mode: si memorizzano nome/file e C-number

Per rappresentare la via di una dir. ci sono diverse strategie

Ogni strategia prevede però necessariamente una sorta di struttura di dati e una variabile quella a sua volta rappresenta il nome della file

## 1<sup>a</sup> strategia

Per ogni voce ~~solitamente~~ max

l'entry del file. L'entry del file contiene:

- entry length

- attributi

- nome (cattivo)

cattivo var e punti

cattivo di terminazione

padding per allineamento/contendere  
di word

file 1 entry length			
file 1 attr.			
a	b	c	d
e	x		
file 2 entry length			
file 2 attr.			
a	c	b	d
x			

attributi

indice lung di tutta  
la entry

estettrare la  
terminazione

Compte spazi; uscire  
di fram. interna.

## 2<sup>a</sup> strategia

Pointet to file 1's name
File 1 attr
Pointet to file 2's name
File 2 attr
:
a   b   c   d
e   x   f   g
h   i   l   j

I cattivi padding non  
sono previsti  
ma non sono aggiornati  
in uso spazio fatto Heap  
Nella parte iniziale abbiamo il  
pointet al nome e gli  
attributi. La parte alla  
fine ha un file  
Heap. Se questo spot non  
ha problemi a finire dati

che sono a fini fini

La concatenazione dell'heap viene rimpicciolito  
tutto n misso meno dati (non cattivo)

## Ricerca di un file con un certo nome (it name o the phone)

Nei file system moderni i dati sono predisposti in cache Host che garantiscono ricerche efficienti. I meccanismi di ricerca sono favolti da un supporto detto Cache del file che velocizza l'accesso. Questa cache è implementata su un set di 50 spaziando la RAM.

## Combinazione di file in file system

Se già utili vogliamo usare un file e mi ritrovo in un file system basato in c-mode possiamo usare gli Hard Link (Conservano file).

Gli Hard Link sono dei riferimenti agli ~~file~~ c-mode che vogliamo mettere in una directory.

Tra i metadati dell'i-mode vi è il Contatore di Hard Link utile per il file system.

Metti in fase di rimozione della dir:

- la voce nulla dir è cancellata
- il contatore viene decrementato
- Se è uguale a 0: tutti i blocchi dell'i-mode vengono liberati

Con questi meccanismi

possiamo verificare le anomalie di accounting.

Ovvio: C è il proprietario di un file, il file appartiene a B e poi C cancella la sua voce relativa all'i-mode del file.

B sta puntando a un file B chi non è proprietario e C è l'unico a puntarci (anomaly).

Cioè il file monterà come file correttamente  
connesso dal proprietario l'orsa juntas da B.

Un'altra strategia prevede l'uso di soft-link,  
ci sono dei vari e propri oggetti detti soft-link  
che hanno come contenuto il path name dell'oggetto  
riferito.

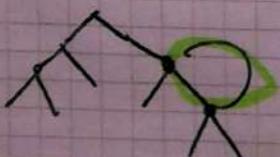
Avere un soft link dunque significa avere il  
file a cui si giunta.

Perché con un soft link andrà a creare un  
nuovo oggetto che ha il suo i-node & il suo path  
contento il che appena dice il meccanismo  
è l'unico modo per fare il riferimento (a differenza  
di oggetti al di fuori del file system nell'ambito  
di soft link possono riferirsi a qualunque tipo  
di oggetto).

In cosa non è possibile con gli Hard Link  
dato che nello i-node del file metto il numero  
dell'i-node dell'oggetto da riferire.

Dunque file-system differenti possono avere  
oggetti diversi ma con i-number uguali, però.  
L'Hard Link è utile solo nel contesto file system  
partizione.

Gli Hard Link potenzialmente potrebbero colpire alle  
directory ma la stessa sia altro file dir  
verrebbe meno nel file system ed esso diventerebbe  
un caso. Vedo che sarebbe problemi alla  
maniera delle certe kemberly infinite.



N.B.: Differenze soft link e hard link

### • Soft Link (Link simbolici)

Gerico file. Sono all'interno di solo il seth per il file. Appena si crea il necessario (altrimenti) se si cancella il file target il link diventa incommutabile

### • Hard Link

Veri e propri riferimenti agli i-node

E' un file contenente l'elenco i-node

Possiamo lavorare solo sulla partizione (i soft link no)

Non si possono usare gli Hard Link tra due

L'Hard Link crea una nuova "pista" per accedere al file

Gli Hard Link sono per definizione riferimenti allo stesso file/i-node del file

Moltiplicando un file da Hard Link lo modifica in tutte le quanti gli Hard Link, anche l'originale

Ultima cosa: Ho un file con un i-node e creo un Hard link riferito ad esso.

Cancello il file delle BT originali, in teoria il file è ancora più accessibile quando l'Hard Link (il record) appena creato.

Se ci sia un file System basato in FAT è possibile duplicare la lista con i riferimenti ai blocchi per fare sovrapposizione → problemi di spazio

## Versione blocchi liberi

Tra i contatti del file system c'è quello di tenere traccia dello stato di allocazione dei blocchi nel disco.

Bit map: invilene di bit, ognuno rappresentante un blocco (ci dà l'info "libero" / "non libero")

La struttura è chiamata free blocks (dove si indicano le caselle vuote).

La struttura è usata per trovare blocchi liberi da usare per i nostri file.

Essa è memorizzata in disco e può contenere solo la posizione della nostra tabella nel RAM oppure pagine intere.

Lista concatenata: rappresentiamo una lista di blocchi liberi. Viene usato un proprio tipo di struttura che rappresenta l'informazione voluta usando i blocchi non liberi.

Per i blocchi liberi contigui ci usano poi dei contatori.

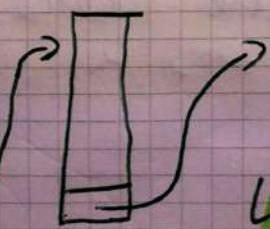
Nominalmente rendono vantaggiose il liste, vengono usate le BITMAP.

Le liste rappresentano SOLO i blocchi liberi (i numeri).

Struttura lista: ho 1KB di dimensioni di blocco e numeri a 32 bit.

Ogni nodo allora ha 256 voci ( $(1024 \cdot 8)/32$ ) che usano 255 per contenere numeri di blocchi (di cui) in disco e una per puntare prossimo nodo.

10
20
30
...
8
...



Se il File system è importante molto la lista ordinamente in accrescere.

La lista vera è implementata tramite blocchi liberi.

La gestione delle liste è già racchiusa  
L'elenco dei numeri non è ordinato (all'inizio si)  
ma poi l'ordine a fini di allocare l'indirizzo  
viene perso

Oggetto N fa un'allocazione o l'indirizzo si lavora  
sulla lista delle liste

La ricerca di contiguità nella lista non è buona,  
è inefficiente

Il controllo di blocchi liberi è protetto dalle memorizzazioni  
ma non tanto i blocchi liberi non informano  
sulle regole di blocchi liberi contigui.

### Controlli di consistenza

Effettuati dal file system o da utility fette  
a mano, invocati o periodicamente o dopo un crash  
di sistema. In seguito ai Crash la RAM perde il  
suo contenuto ma il file system non è robusto  
e le operazioni sui blocchi devono riconoscere

Queste operazioni non sono atomiche e il SO  
al pianto non è in grado di cogne dove  
è avvenuta l'interruzione.

Dopo il pianto dunque è possibile che il file  
system non sia coerente

Ese: potrebbero essere blocchi registrati come occupati  
ma in realtà liberi facendo l'operazione è  
stata interrotta e mette.

Per questi motivi vengono attivate le utility  
di controllo di consistenza che cercano inconcordanze  
all'interno del sistema e provano a correggerle

Contiene un meccanismo di controllo consistenza

Contiene 2 vitori: uno che indica il numero di volte che il blocco è usato e uno che indica il numero di volte che il blocco è citato come libero

Dopo aver creato i vitori (memoria file/c-mode e struttura dati che tiene traccia dei blocchi liberi)

dovrebbero contenere tutti 0 oppure 0 e 1. In realtà non possono garantire viene demandate

- [0, 0] 1°: ~~lib~~ nella tabella dei blocchi "in uso"  
2°: ~~lib~~ nella tabella dei blocchi liberi

• [0, 0]: blocco non usato che liberi? Fai la risposta: se no aggiungi il blocco alla lista dei liberi (il 2° sarà 1)

• [0, 1]: il blocco compare liber 2 volte (vi è una voce nella lista).

Basta togliere la lista

• [1, 0]: stesso blocco nello stesso file/c-mode  
Se cancelliamo da uno dei 2 si presenta [1, 1] (caso d'errore), se cancelliamo da entrambi si presenta [0, 2]

La soluzione è: allacciare un blocco libero, copiargli il contenuto del blocco in questione (quello che c'è da 2 volte) in uno dei due c-mode

(~~e cancellare quelli~~)

La struttura del file system viene resa consistente

(i due c-mode puntano a blocchi diversi ma con uguale contenuto)

Lo Journalling di controllo dei blocchi immutabili  
non può applicare allo Journalling di controllo dei cambiamenti  
degli i-nodo nelle biteset.

## Journaling

Molti file system moderni impiegano queste tecniche  
che consiste nell'effettuare una meta operazione  
(che è un insieme di micro-operazioni) e di  
prevedere una transazione fatti persistente sul JOURNAL  
in cui vi si tiene una descrizione delle  
operazioni che sta per effettuare.

Non vengono scritte tutte le info ma solo quello  
che tocca i metadati per motivi di spazio.

Dopo che è stata eseguita l'operazione i log nel  
Journal sono cancellati e si fa ritorno alla  
prima meta operazione.

Idea: registrare le operazioni salvate sul log (che è  
salvato sul Disk) in caso di crash.

Il Journaling ferde i metadati salvati e offre  
un reboot veloce.

Il meccanismo è applicabile solo alle operazioni  
salvate sul log sono interattivi ovvero rigettibili  
senza fare damage.

Salvare permanentemente le operazioni sul log può  
creare gravi over Head.

## Cache del Disco (buffer cache)

La cache del disco è implementata via SW e  
utilizza la RAM per "graze".

Ha le stesse 5 operazioni di I/O del  
disco quindi possibile.

Idee: memorizzare blocchi già precedentemente usati

Ni sistemi UNIX qualiasi blocco di RAM libero è  
usato nella cache del disco, bensìche la cache non  
ha memoria fino.

Tutti i frame in questione sono inseriti in una  
lista appositamente indicata, così si può applicare  
come algoritmo di gestione l'LRU

Il costo della lettura su lista non sarà mai  
presente quanto un I/O

Per prestazioni già elevate si usa una tabella Hash  
che mappa i nodi della lista (non si usa una  
lista circolare)

Dunque ho una struttura ibrida che permette  
accesso veloce e LRU

La cache funziona anche in scrittura, infatti  
ritorna (in chi può) lo stesso nel disco  
del blocco.

Infatti ti turbano 10 scritture in DAS per poi  
effettuare tutto di seguito già in la provocando  
materiali miglioramenti di prestazioni (il costo  
di tifprimento continua della testina)

Il blocco sparsa (non intende da memorizzare) venrebbe scritto in disco solo quando scritto dall'LRU.

Questo già dove problemi in caso di cache, il blocco non sarà verrebbe ad essere memorizzato con i logici in RAM (anzi però)

L'integrità è più importante dell'efficienza per questo l'LRU usato è un po' modificato.

Se c'è bisogno di una sincronizzazione ti guarda i metadati ma non viene ritardata ma effettuata subito.

Se la memorizzazione non ti guarda metadati vi è un po' di ritardo ma che è controllato.

Viene meno un limite al ritardo

### Optimizzazione Cache

- **Read-ahead**: Il SO quando carica un file in cache un blocco in carica anche i blocchi contigui.

Porta enormi vantaggi se il file è memorizzato in maniera sequenziale.

Questo è particolarmente importante, se avranno blocchi solisti in varie zone non contigue. Il meccanismo potrebbe in cache blocchi che non c'entrano nulla.

- **Free-behind**: L'idea è che in un processo sta leggendo un blocco. Difficilmente i blocchi precedenti saranno usati e quindi si liberano (abbiano già posto in RAM).

Queste due tecniche sono state usate insieme.

Antae toccherà più pezzi (nella soluzione linea)

Il SO gestisce anche come riunire i dati sul disco  
nel modo più efficiente possibile.

Ese: gli i-mode avendo pre-allocati sono messi in  
una posizione ben precisa quindi potremo disporre  
in modo contiguo nella traccia più esterna del  
disco. A ogni i-mode corrisponderanno poi  
dai blocchi che contengono il file.

Sappiamo che prima è letto l'i-mode e poi il  
file, mettere distanze non ha senso

\* Per rendere più efficiente la lettura di file conviene  
mettere gli i-mode in tracce diverse e disporre  
il file ~~in modo regolare sulla traccia dell'i-mode~~  
~~corrispondente al file allo stesso cilindro~~

In questo modo il braccio meccanico con la testina  
è嘛 il meno possibile nella lettura di un file

Dunque la distanza linea tra blocchi e i-mode corrisponde  
al tempo per I/O per riportare fisicamente la testina

Un esempio sarebbe mettere gli i-mode invece  
che sulla traccia più esterna sulla traccia intermedia  
(il tempo medio di i-mode  $\rightarrow$  blocchi è dimezza)

Nella realtà: \* gli i-mode sono suddivisi  
in gruppi, il gruppo è rappresentato dal cilindro  
In un cilindro ci mette degli i-mode e blocchi  
corrispondenti così le probabilità per il riavvicinamento  
della testina migliori.

Se il file in un certo cilindro ha dei blocchi il  
SO cerca di uscire blocchi dello stesso cilindro,  
se non sono disponibili prende blocchi di altri  
cilindri e si scommette

Idealmente il file è quindi sommerso tutto nello stesso cilindro.

Se non si può usare lo stesso cilindro (necessario)  
uso blocchi di cilindri controllati

Abbiamo creato una sorta di new contiguous

per strategia

contiguous



(l'assegnamento prende in considerazione gli ammortatori nella traccia centrale)

Degradazione: process che monitora blocchi per ottenere la situazione ideale quando ci viene incontro.

## Esercizi esami (file system)

1) i-node con metablo, 10 blocchi dati, 3 ad offset  
(1 blocks indirizzo, 2 e 3)

Blocos di 5KB  
numero di blocos: 32 bit  $\rightarrow$  Voci ?  $\frac{5 \cdot 1024 \cdot 8}{32} = 1024$

In quale blocco viene ; 0 byte di offset XXX ?

Sì fa XXX (in bit !?)  $\rightarrow$  5,36 (ad esempio)

attraversone per ecceso

5° bloco

NB: il 1° bloco è al 10  
alla estensione.

Dunque se per esempio tale divisione si mette 12 e nel 2° bloco alla 1a estensione c'è 565

La risposta è 55.  
 Dunque nella tabella prima: punto 10  
 nella tabella seconda: 11 - 1035 (ogni)  
 La 1^a voce della tabella terza è 10350.  
 (in realtà la tabella terza giunta a  
 1025 tabella con numeri di blocchi)  
 La 1^a di queste va da 1035 a 2058, e  
 neanche da 2060 a 3085

## 2) FAT

index	next
1	5
2	3
3	15
4	5
5	10
6	12
7	1
8	2
9	3
10	-1

Controlla facile

name	10	size
file1.txt	?	?
file2.txt	?	?
file3.txt	?	?

$$7 \rightarrow 1 \rightarrow 5 \rightarrow 10$$

Dim del File? (blocks = 5 KB)

Connesso tra  $(5 \times 5\text{KB}) + 1 = 26\text{KB}$

In quale blocco si trova è localizzabile

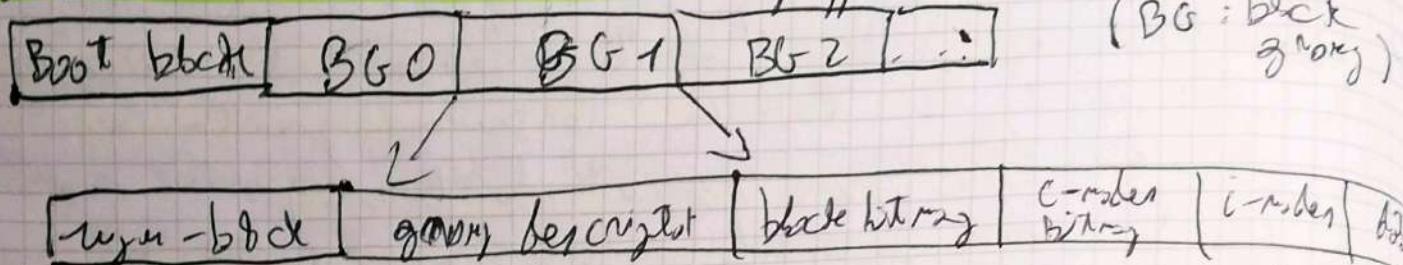
L'offset 10100 dà tra pippo.txt?

$$\frac{10100}{5096} = 2, 3 \approx 3$$

Nel 3^o blocco di pippo  
 Dunque 3 - 12 blocchi  
 E' il 12^o carattere C

# File system di Linux

Bordo in i-mode e in gruppi di blocchi.

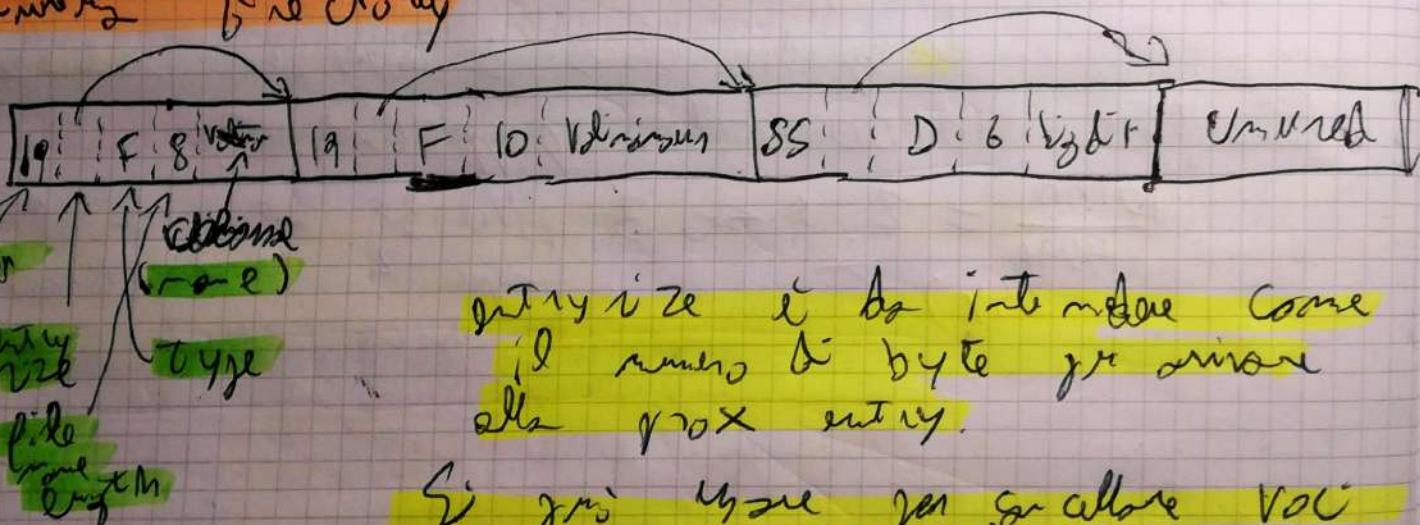


Nel singolo gruppo ha una sua struttura

- Super-block ha anche info sul gruppo stesso
- Unico degli i-nodes presenti in i-nodes gruppi, le gestisce con bitmap
- i block di dati li gestisce pure con bitmap

Bordo ha un file F in genere i suoi blocchi si cercano nei primi nello stesso BG per efficienza  
Se F non è presente integralmente nello stesso BU, var blocc. si cercano nei BU di intersc.

## Structure Directory



entry size è la dimensione come il numero di byte per arrivare alla prossima entry.

E' possibile usare per scaricare VOC (rendendo Unread).

Queste strutture sono chiamate framme interne (buchi interni) ma possono essere facilmente ricomposte queste direttive (-f /nurber anche la cache del Nics)

I primi quattro punti sono ovviamente gravi e questi dovrebbero  
essere evitati (fatto 255 char)

Con ext 3 si dirizza a usare il journaling  
(nella interno alla partizione si per la partizione  
principale)

Con ext 4:

• introdotto extent per avere i-mode e long  
variable (la voce contiene degli extent che  
contengono o seg di blocchi)

Così creano già contiguità.

Anche nelle versioni precedenti n. 4 sono tecniche  
~~per~~ per creare già contiguità possibile in  
partizione le stripe e  
a un file serve un  
lo contigui così i max  
in modo contiguo)

• allocazione multi-blocco: il processo può richiedere  
l'allocazione di tutti blocchi, quindi non  
extent

• allocazione (mette time out per evitare tick enorri)  
ritornata: lavora la allocazione di extent  
Quando il processo richiede 5 blocchi (ad esempio)  
il SO li prende in carico il contenuto e  
rimanda l'allocazione dei 5 blocchi

Può essere ~~errato~~ giusto (rimozione di  
allocazioni e scrittura)

Pericolo sono i holes: può accadere  
richieste (es: 5 file da 5 blocchi) per  
tutti insieme come un unico extent (16 blocchi)  
Perché rischia? se c'è crash durante questo  
procedimento

• Un altro con che oggi oggi è **ReFS** (Refrigerator online)

**BTRFS**: "B-tree FS" o "butter FS"

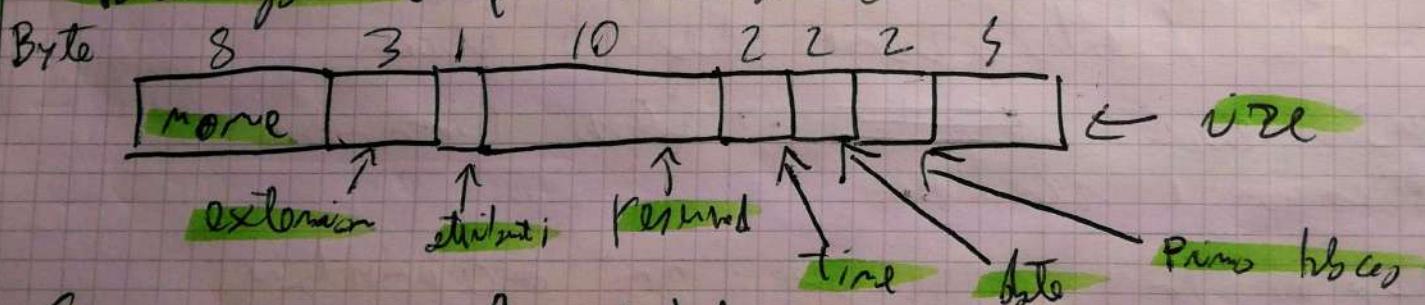
Un altro tipo di file system usato in Linux

Ajrage caratteristi che si prevedono FS Linux

- chiamazione di ogni file (e i parametri sono spesso copy-on-write nei file)
- sottovolumi
- maghiot di sottovolumi (con copy on write)
- non ricevere dati in maghiot
- API basata in B-tree
- Defragmentazione, aumento e riduzione online di volumi
- check sum per dati e metadati (per rilevare errori)
- compressione trasparente file
- meccanismo tipo RAID nel volume

**File System MS-DOS (FAT)** (non ridevo)

Voci per il file nella FAT:



FAT-12 (12 bit)    FAT-16    FAT-32

altrettanto: exFAT (64)

## File system NTFS (non tradotto)

- Nome con Windows NT: è avanzato e complesso
- Volume: una o più partizioni o di cui
- cluster
- Master file table composta da record
- file: collezione di attributi var  
    { metadati o flusso (dati dati)  
    { persistenti o non residenti  
    { file può richiedere più record
- API basata su B+Tree (ricerche efficienti)
- Block liberi gestiti con bit map
- hard-link e soft-link e montaggio altri FS
- journalizing
- compressione e crittografia file
- copy shadow di volumi (modo copy-on-write)

## Scheduling del Dico

- minimizzare throughput
- minimizzare tempo medio di accesso

Il SO raggiunge questo obiettivo facendo degli assegnamenti.

Ei vedono gruppi di richieste condivise sul Dico.  
Le richieste si mettono nello stato delle richieste pendenti  
che è nel controller del Dico.

Il SO abba varie politiche di selezione delle richieste per prendere la prossima da eseguire.

La sua già superficie da fare dovrebbe soddisfare le richieste nell'ordine in cui arrivano ma se il SO sceglie in modo opportuno potrebbe raggiungere gli obiettivi (o almeno in modo approssimativo).

Selezionare in modo opportuno sarà come conseguire:

- ridurre tempo di piavimento (seek time)
- ridurre latenza di lettura

Ottimizzando il seek-time (il seek-time è la parte già aperta)

Le richieste in ordine di arrivo e # di cilindri

98 183 37 122 15 125 65 67

per iniziale tenere: cilindro 53

Cilindri: rappresentazione delle tracce

Tracce rosse: non opposte, formano  
logamente un cilindro

Dunque le due richieste hanno # cilindri uguale il seek time è 0

Ei potrebbe immaginare che organizzino



le richieste in gruppi, dove le richieste di un gruppo formano lo stesso # cilindri.  
Colgono seek-time per la lista di grida.

53. 98 183 37 122 15 125 65 67  
inizio

FCFS: first come first served

seek time totale = 690 trace jucorre

53 - 98 + 198 - 183 + 183 - 37 + ... 165 - 67

Nb. FCFS è semplice, ma non inefficiente

SSTF: shortest seek time first (consiglio per es: ottimizzazione delle richieste)

Nella fl. diversi nella coda, si sposta nel giro vicino, salvo la ricerca e riconcilia.

Dal 53 valo a 65, poi 67 e così via...

53 → 65 → 67 → 37 → 15 → 98 → 122 → 125 → 183

seek time totale: 236 trace jucorre.

NB: i kerne re le richieste sono nate a tempo 0, se arrivano a tempo t sono di corrispondere in molti atti.

Il ritardo non è ego, crea l'arrivo di richieste (ovvero richieste inviate da clienti che sono già partiti).

NB      0      ...      199      trace estrema

## Scheda di funzione SCAN

(dopo l'ultimo  
della funzione)

La testina ha un solo di funzione

$$0 \leftarrow \dots \rightarrow 199$$

Si continua funzione della testina non dopo aver  
tracciato ~~o~~ un estremo

Dunque all'inizio effettua una posizione iniziale e  
un passo della testina.

(53,  $\leftarrow$ ) subito a 37, poi a 15, dopo va  
ancora verso ~~15~~ e raggiunge 0.  
Raggiunto 0 il passo si trasforma in dx.

Dunque la sequenza del sovrapposito è:

53 37 15 0 65 67 98 122 125 183  
 $\uparrow$   
inizio

Fare una rimozione uniforme

Garantisce la stessa manovra per ogni traccia, ma  
è più lente di modo

Dunque il worst case per una ricchezza  $t$  di avere  
una latenza di max equivalente al passo della  
lateralità del disco (se il binario è lungo 200  
tracce una ricchezza segnala max 400 tracce percorso)

## Variante C-SCAN : rimozione circolare

Considerare le posizioni 0 e 199 come collegate  
 $199 \rightarrow 0$

Se arriva a 199 posiziona la testina al cilindro  
0 (ignorando i ricchiamenti intermedii)

Con le liste di giri 2mo modo questo avviene

53 65 67 98 122 125 183 199 0 15 37

Il C-S CAN rende gli stadi più imballo con  
ritardori in cui arrivano a tempo coincidente formazione  
regalo

Grazie a un tempo medio di attesa più uniforme

Il C-S CAN ~~raggi~~ non ha c'è allo stesso, se ci  
sono canali è meglio il SCAN monodirezionale

### Ottimizzazioni SCAN

SCAN → LOOK

C-S CAN → C-LOOK

Ottimizzazione consiste nell'estate di orbale agli  
estremi del disco se non è necessario

Nello SCAN si gira invece di fare  $15 \rightarrow 0$  e  
~~poi~~  $0 \rightarrow 65$ , farsi invece  $15 \rightarrow 65$

(ovvero si ricorda con # gli indirizzi n eseguita  
come se fosse l'eterno 1x, ricevuta per il dx)

Nel C-SCAN si gira invece di fare  $183 \rightarrow 199$ ,  
 $199 \rightarrow 0$  e  $0 \rightarrow 15$  farsi direttamente

$183 \rightarrow 15 \rightarrow 37$

(ovvero 183 e 15 n eseguita come gli estremi)

Altri metodi:  
• C-LOOK per allo canali  
• LOOK o SSTF per bassi canali

### Esercizio

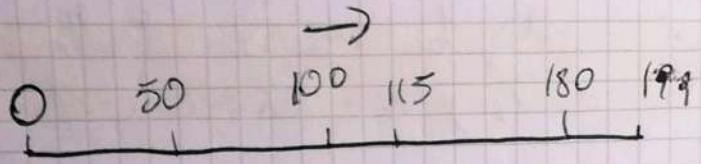
200 tracce (0...199) - velocità seek: 1 ms

a t=0 siamo alla traccia 100

Inoltre 50 115 180, a t=70 arriva traccia  
per 150

a t=130 per 90

Calcolare tempo seek con le  
viste la politica look  
(verso inizio  $\rightarrow$ )



$t=0$  coda: 30, 115, 180

$t=70$  150  
 $t=130$  90

Dopo 15 ms è arrivato a 115

$t=15$  coda 50, 180

Poi sollecita 180 al  $t=80$

$t=80$  coda 50, 150 (~~150 non è stata sollecitata perché (150 è arrivato nel frattempo)~~)

Allo stesso istante (fino a 150)

$t=110$  (80 + (180 - 150)) coda 50

Vado ora verso 50

$t=210$  (110 + (50 - 50)) coda: 90 (e' arrivato nel frattempo)

Vado verso 90 (ho arrivato lì)

$t=250$  (210 + (90 - 50))

Seek totale = 250 ms

Sistemi RAID (oltre 20 anni usati per proteggere i dati in più di un posto more per resistenza agli errori)

Abbocca la ragionevole pretesori già oltre, basati sullo 1° fattore il parallelo

E se non ce n'è uno già di cui interamente (almeno 2)

Attribuzione: creare un unico volume logico che in realtà è implementato su N dischi fisici

L'attribuzione consiste dunque nell'unire già molti dischi fisici per apparire come un solo logico volume logico

N.B.: mettendo in corso una rotazione degli estremi

**RAID:** Redundant Array of Inexpensive Disks  
o sono più in grado di creare un ottimo

Poi in realtà la cosa non era del tutto vero e quindi è cominciato l'acronimo: Redundant Array of Independent Disks

Infatti i dischi problemi lavorano in modo parallelo e indipendente

Questo modellino permette di utilizzare i dati relativi ad una unità logica (file) su due dischi.  
Si dice questa tecnica striping.

La suddivisione è trasparente all'utente

Il modello si può implementare via HW (e il SO non ha né scorge rispetto) o via SW (lavora applicato alla CPU)

**Tecnica striping:** tecnica di base per creare parallelismo

L'idea è partizionare il file in pezzi/linee  
che vengono scritti in parallelo sul disco

Ogni stripe è memorizzato su diversi dischi. Se ho

5 stripe e 5 dischi: 1<sup>o</sup> stripe su 1<sup>o</sup> disco,  
2<sup>o</sup> stripe su 2<sup>o</sup> disco, 3<sup>o</sup> stripe su 3<sup>o</sup> disco,  
4<sup>o</sup> stripe su 4<sup>o</sup> disco e 5<sup>o</sup> stripe su 5<sup>o</sup> disco.

Applicazione usata da Robert Baldwin per il salvataggio degli stripe.

Se poi voglio leggere l'intero file ci metterebbero un quanto del tempo rispetto a un solo disco.

Esempio: transfer rate di 1 MB/s

Pur leggendo un file da 5 MB con il singolo disco ci impiegherei 5 secondi, così se faccio RAID 5 records

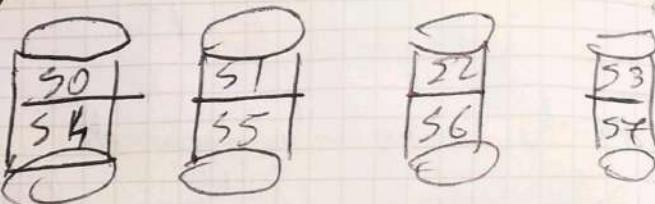
~~1~~ 1 5 dischi lavorano insieme in parallelo.

Il lavoro fatto sul file viene scritto sul Volume logico (insieme di byte)

Ese: Volume logico di 5 TB  $\rightarrow$  5 volumi fisici da 1 TB

### RAID 0 (striping in N disk)

- striping con Round Robin (non è ragionevole nel file ma sul volume logico)



Volume logico					
S0	S1	S2	S3	S4	S5...

Le richieste al S0 di leggere info sul Volume

logico vengono rispondute al leggere di tutte sul Volume fisico

• Caso ottimo: stripe richiesti alternati sui diversi fisici (speed up  $\times 5$ ) (es: 8 richieste  $\rightarrow$  2 stripes in ogni disco)

• Caso peggiore: tutti gli stripe richiesti stessi sullo stesso fisico (non vale nulla)

• Caso medio: è già vicino al Caso peggiore (Dato che  $\frac{1}{5}$  delle spedite)

Il RAID 0 non aggiunge nessuna robustezza al sistema dei quattro hard disk perché il volume logico ha la stessa probabilità di fallimento.

Molti preferiscono l'uso di due zone = 10 dischi utili, ma ne ha solo 8. Le 2 degli 8 si trovano entro i 4 dischi gli altri 2.

Proteggere dai guasti? Aggiungere robustezza

## RAID 1 (mirroring)

- al massimo ragione anche degli errori (mirroring)
- si crea la copia speculare del volume logico

S0	S1	S2	S3	S4	S5	S6	S7	S8
S5	S5	S6	S7	S3	S3	S6	S7	S4

Ho 8 dischi fissi frangere un volume logico in 8 nuclei fissi duplicati (il volume logico (max 5 TB))

- Questa tecnica di mirroring è più utile anche senza striping
- migrazione fault tolerance (non è perfetta)
- Ho un alto overhead di storage (mettere una striscia su un solo volume mettendo un simbolo diverso)

Le gestioni sul cosa di ricaricare i vari stripes sono costantemente

- Cosa migliore: ~~file~~ <sup>triste</sup> isolati here sui 8 dischi (e quindi anche sulla copia)
- Cosa peggiore: ~~file~~ <sup>triste</sup> concentrati su un singolo disco (quindi anche sulla sua copia)

$$\text{bytes up} = \times 2$$

- Cosa medio: bytes up tra 2 e 8

Per le ~~lettture~~ <sup>lettture</sup> vale appunto il binomio null'overhead di storage

- ~~Cosa pessima~~: ~~bytes up / 3~~ (~~è comunque l'isolamento~~)

NB: RAID n garsi fare anche con 2 dischi

RAID: probabilmente no un troppi dischi

## Raid 2 Mixing a Circuito di bit con ECC

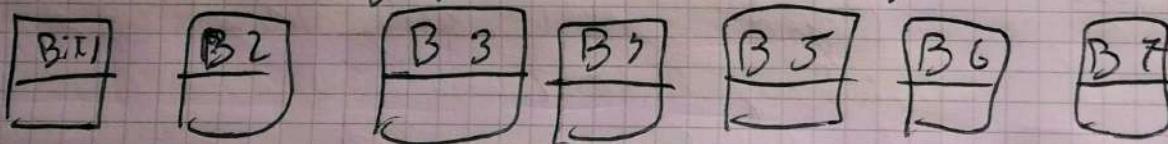
Si usc la tecnica del bit mixing e si applicano le tecniche di generazione e rilevazione di errori.

Il RAID 2 lavora ai byte significa un codice di correzione per carri

- Volume logico di 7 dischi fisici
- Parola di 5 bit  $\rightarrow$  7 bit per Hamming (3 bit ribaditi)
- I 7 bit sono di dimensioni di 7 blocchi
- Da un punto di vista logico ci sono circa 5TB  
ma: esistono 7 invece di 8.
- Ha un'ottima fault tolleranza e ha buone prestazioni per leggere e scrivere righe

Problema: serve sincronizzare perfettamente i blocchi  
cioè se anche solo 1 disco fisico fa un fallo, la lettura dell'intero volume  
è fallimentata.

Potrei? puo memorizzare i bit, per leggerne <sup>una parola</sup> byte  
mi serve usare tutti i 7 dischi  
mentre prima le parole le mettevamo  
tutte raggruppate sulle righe



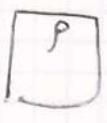
Per leggere la parola B1 B2 B3 B4 usiamo tutte e  
7 le righe (3 ribadimenti)

Questo problema viene risolto facendo mixing a livello  
di bit insieme che a livello di byte / blochi.  
(maggiorità delle ~~caso~~ si bit cells word)

(In raid 0 e 1 negli stage invece maggiore i  
block)

## RAID 3

storing a byte of bit in a word  
bit di parità  
Ese: 5 bytes fino + 1 da bit di parità



parità

Se i gradi di blocchi sono > 1 e quindi oltre a queste informazioni serve il bit di parità per bloccare il valore di  $B_5$  nonostante il bit sia in totale. Dunque fanno si operi tutto quanto RAID 2 ma con 2 dischi in meno.

Anche qui siamo che non a livello di bit ci sono i problemi di sincronizzazione tra bit

Dunque: stiving a livello di bit  $\rightarrow$  inefficiente  
Non notate RAID 2 e 3 non si usano

RAID 5 stiving a livello di blocchi con XOR  
sull'ultimo disco

- faccio stinge a blocchi
- ci sono blocchi extra  
dei bit XOR degli stinge

$$\text{Quindi } P_0-3 = S_0 \otimes S_1 \otimes S_2 \otimes S_3$$



volume logico

L'ultimo stinge è la  
ridondanza per la fault tolerance (ne abbiamo 3 e  
non 4 perché  $S_3$  è ricavabile)

- Non è bastato un bit quindi non meccanica di  
incisiva

Prestazioni lettura: uguali a RAID 0

Caso niente:  $x_3$   
caso regione:  $x_1$

Prestazioni scrittura: gran perdita inefficienza, moltiplicare  
con stinge poiché dobbiamo ricalcolare il blocco  
di ridondanza facendo tutti gli XOR

Ottimizzazione:  $P_0-3$  si può calcolare come  $P_0-3 \otimes S_3 \otimes S_3$   
ne è  $S_3$  che calcola (in  $S_3$ )

Dunque l'aggiornamento di  $P_{0-3}$  è in genere fatto da un solo ribaditore non chiede tutti gli altri

Saranno i vecchi blocchi di ribadimento ( $P_{0-3}$ )

$$\text{vecchio blocco di ribadimento } (S_3) \quad P_{0-3} = P_{0-3} \otimes S_3 \otimes S_3$$

$$\text{a nuovo blocco dati } (\overline{S_3})$$

Dunque con questa tecnica l'aggiornamento non è così inefficiente

NB: l'ultimo blocco (quello di ribadimento) è il più grande / usato blocco che è sincronizzato ogni 200ms

In pratica l'ultimo blocco si usa una

Risolviamo questo problema con **RAIDS**

- Utilizzando diversi blocchi con informazioni ribadimenti distribuite

Funziona come RAID 5 ma il file di contenzione di info ribadimenti è distribuito a tutti i dischi

$S_0$	$S_1$	$S_2$	$S_3$	$P_{0-3}$
$S_4$	$S_5$	$S_6$	$P_3-F$	$S_7$
$S_8$	$S_9$	$P_8-II$	$S_{10}$	$S_{11}$
$S_{12}$	$P_{12-B}$	$S_{13}$	$S_{14}$	$S_{15}$
$P_{16-I}$	$S_{16}$	$S_{17}$	$S_{18}$	$S_{19}$

In termini di feiabilità a quanti è identico a RAID 5

I blocchi sono in media sparsi sul tempo  
modo quindi non ci sono blocchi che si usano  
in modo più veloce

Permette una maggiore anche delle letture rispetto  
a RAID 5 uno speed up  $\times 5$

Per la scrittura i fatti le stesse considerazioni di  
RAID 5 e bisogna non più di tanto inefficiente

## Memoire Flash e file system

Dispositivi basati su flash (NAND o NOR) formidabili  
in modo diverso dai dischi elettronici.

- Un blocco può dunque essere riscritto bene senza cancellare.
- Il numero di cancellazione (e quindi scrittura) di un blocco è l'indirizzo (per una questione proprio finita, subendo oltre il blocco in quota)
- Il singolo blocco è composto da pagine; cioè le operazioni di lettura / scrittura si eseguono in pagina. PERO se devo cancellare / scrivere una pagina dovrei cancellare l'intero blocco, dunque si evita la cancellazione (finché non).

A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>

Dopo modificare A<sub>2</sub> → Ā<sub>2</sub>.

Se poi vado a sostituire A<sub>2</sub> con Ā<sub>2</sub> dovrei cancellare tutto il blocco e rimettere anche A<sub>1</sub> e A<sub>3</sub>.

Dunque il controller trova una nuova cellula vuota e ci mette Ā<sub>2</sub> e nega la vecchia A<sub>2</sub> come "non valida".

Il SO riconosce se ne accorge, lo (ri)protegge e fa il controller.

Se il blocco è pieno e molte pagine sono "non valide"?

E se cancella il blocco e poco prima lo ha riscritto in un altro blocco (sia solo le valide). In questo caso ho reagito alla pagina pagando una cancellazione.

Problema: cancella file → SO lo sa ma il controller no, quindi il Garbage collector continua a considerare valida le pagine del multiblock file.

Garbage collector: parte che cerca blocco alla fine

Nelle pagine viste ecc.

Sostanzialmente il controller non memorizza informazioni sul SO quindi cancellazione di file continua a fare garbage collection su file.

Infatti oggi si usa la TRIM che funziona molto bene perché comunica al controller i blocchi cancellati.

Così ottimo miglioramento nel tempo del file

- Garbage collection → blocco Nove (non usano pagine relative a file cancellati)
- Garbage collection → permette di rigenerare pagine con TRIM

Dunque il SO deve adeguarsi a questo tipo di file (mentre il meccanismo di TRIM)

Di solito non ci sono file system dedicati a queste memorie

• Flash Friendly File system è nato in tentativo di preferire l'abegamento da parte del SO

NB: file system visto precedentemente conserva senza lavorare il fatto che ogni blocco abbia un limite fino a cancellazioni impossibili.