Riassunto da: Enrico Mensa

Sistemi Operativi
Cenni generali

Capitolo 1

Indice degli argomenti

Il sistema operativo

Il sistema operativo è, sostanzialmente, un insieme di programmi che si frappone tra l'utente e l'hardware puro: è una sorta di mediatore. Il sistema operativo si occupa di tre differenti aspetti:

- **Ambiente di lavoro** (dal punto di vista dell'utente, esso offre una interfaccia grafica ed un luogo dove poter eseguire i propri programmi)
- **Gestore di risorse** (vengono gestite risorse software e hardware, come CPU, RAM, ecc.)
- **Funzione di controllo** (si occupa di verificare che non ci siano errori durante l'esecuzione dei programmi, inoltre, ad esempio, permette la gestione di utenti differenti con differenti livelli di accessibilità)

Per comprendere al meglio come può il sistema operativo adempiere a tutte queste mansioni è necessario fare una premessa storica.

L'evoluzione storica del sistema operativo

*** Il sistema Single-Stream Batch Processing** *(processione in sequenza di un singolo filamento di dati)*
Ai tempi i calcolatori erano in grado di leggere fogli di carta perforati. Su tali fogli venivano pertanto registrate sia le operazioni da eseguire che i dati necessari per eseguirle. Un pacchetto dati+programma viene chiamato **job**. La macchina esegue costantemente (fino a che ci sono job) il ciclo leggiJob --> esegui Job --> risultato.

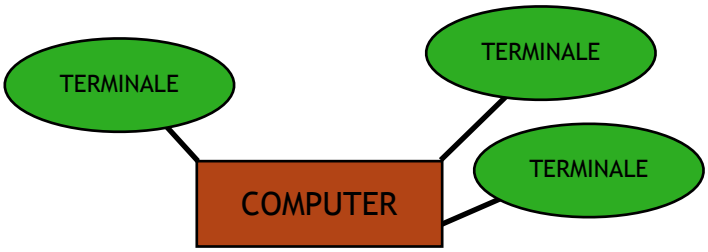
*** Il multitasking**
Un miglioramento rispetto al sistema precedente è certamente una migliore organizzazione dei dati. Se ad esempio noi avessimo tutti i dati su una memoria interna al computer, non sarebbe più necessario metterli nei job. L'esecuzione di un programma è ora differente: è infatti necessario attendere che i dati vengano recuperati dalla memoria per essere eseguiti (seguendo le direttive del job, oramai solo più contenente l'istruzione). Supponendo quindi di avere nel *job1* l'istruzione "somma x+y", avremmo bisogno prima di recuperare x e y dalla memoria, e questo occupa tempo.
La rivoluzione consta nel fatto che mentre x e y vengono prelevati dalla memoria, la CPU, che è libera, può tranquillamente iniziare ad eseguire il *job2*. Questo metodo viene detto **multitasking**, cioè multi-operazione.

In sostanza, ora, il sistema operativo vedrà i suoi job spezzettarsi ed alternarsi continuamente fra parti di esecuzione e parti di comunicazione con periferiche di input/output.
Chiaramente questa funzionalità complica notevolmente il sistema operativo, infatti ora è necessario:
- Accorgersi che sta avvenendo una lettura/scrittura in memoria
- Congelare un job
- Effettuare un context switch (cambiamento di contesto, cioè di job)
- Identificare ogni job (stack)

Per capire l'effettivo **miglioramento prestazionale** che questa modifica ha apportato, è necessario parlare dell'unità di misura chiamata "**turn around time**". Questa unità conta in sostanza il tempo in cui un job sta in esecuzione. Il TAT di un sistema operativo multitasking è decisamente inferiore rispetto a uno "classico". Diminuire il TAT significa diminuire i tempi morti della CPU.

Una ulteriore miglioria potrebbe essere ordinare diversamente i jobs a seconda di come occupano la CPU: questa operazione viene detta **scheduling della CPU**.

*** Utenti e terminali**
Supponendo di voler permettere l'accesso ad un singolo computer a più persone, dovremmo considerare parecchie migliorie.



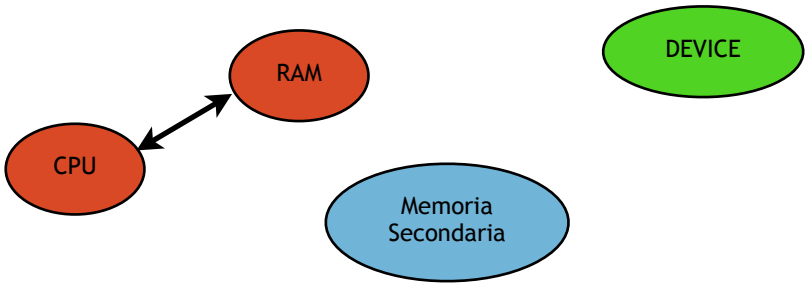
Prima di tutto, il computer deve condividere la propria CPU con tre utenti differenti, questo significa che ogni utente deve avere accesso alla CPU in maniera equa. Questo sistema viene detto **time-sharing**.
Inoltre, nasce anche l'idea di file, l'idea di directory e l'idea di protezione (accessibilità differenziata a seconda dell'utente).

*** Il personal computer**
Con la nascita del personal computer, c'è stata una involuzione anziché una evoluzione. Alcune delle potenzialità implementate precedente sono infatti (in questa applicazione) inutili. Ad esempio, la gestione time-sharing piuttosto che la protezione dei file è inutile se il computer è ideato per essere utilizzato da una persona soltanto.

Questo ci porta alla conclusione finale che: **non esiste un sistema operativo migliore dell'altro, ognuno è tagliato per il suo utilizzo.**

Il duro mestiere del sistema operativo

*** Dove è posizionato il sistema operativo?**



Il sistema operativo si trova totalmente nella memoria secondaria (ad esempio, un hard disk), ma durante la sua esecuzione una piccola parte viene immagazzinata sulla RAM (i processi correnti).

*** L'avvio del sistema operativo**
Quando la macchina viene avviata un piccolo chip che si trova sulla scheda madre, detto **BIOS** *(Basic I/O System)*, esegue la prima istruzione detta **INIT** che permette il riconoscimento delle varie periferiche del computer e quindi riconosce l'indirizzo di memoria (nella memoria secondaria) da cui parte il sistema operativo e ne esegue l'avvio. Questa fase viene detta **identificazione del sistema operativo**.
Questo procedimento prende anche il nome di **bootstrap**, cioè l'esecuzione di processi a catena (un processo ne chiama un altro, proprio come il BIOS che chiama il sistema operativo).
Dopodiché, il sistema operativo resta in attesa.

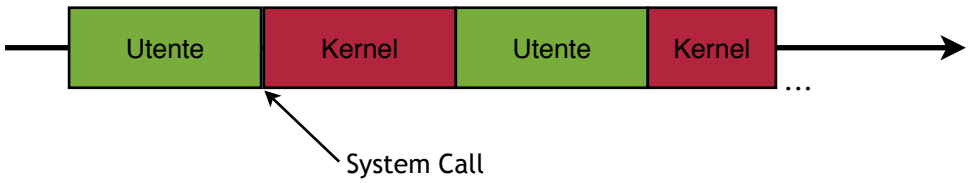
*** Sistema operativo: un programma guidato dagli eventi**
Il sistema operativo resta costantemente in attesa finché non c'è un evento, che può essere di tipo **fisico** (ad esempio è richiesta da qualche applicazione una lettura su disco), oppure **software** (ad esempio un puntatore errato cerca di scrivere in una zona della RAM riservata al sistema operativo stesso, o ad altri dati rilevanti).
Nel primo caso il sistema operativo si trova a dover gestire un **interrupt**, nel secondo una **trap**.

Il sistema operativo, quindi, si trova continuamente con una lista di eventi da dover gestire.
evento3 --> evento2 ---> evento1 --> dispatcher --> handler corretto
Una parte del sistema operativo detto **dispatcher** si occupa di gestire l'evento mandandolo all'**handler** corretto (cioè colui che poi gestisce effettivamente l'errore / richiesta).

La CPU è quindi continuamente interrotta, e pertanto avvengono molteplici switch context in un solo secondo.

*** La gerarchia del sistema operativo (dual mode)**
Chiaramente è necessario determinare una gerarchia tra le differenti richieste: potrebbe infatti essere pericoloso lasciare che un programma utente abbia a disposizione l'intero instruction set. Nasce così il **dual mode**: un bit fisico che a seconda del suo stato definisce la modalità attiva al momento.
Le due modalità sono dette rispettivamente **modalità kernel** (su cui lavora il sistema operativo) e **modalità user** (su cui invece lavorano gli utenti). Come standard possiamo pensare che con il bit a 0 si abbia la modalità kernel, viceversa la modalità user.

Pertanto, quando avviene un interrupt od una trap, viene effettuata una **system call**, tramite la quale viene effettuato il cambio di modalità, l'interrupt (o la trap) viene passato/a al dispatcher che capisce qual'è l'handler corretto il quale infine gestisce la richiesta (corrispondente quindi a quella system call).
Più precisamente la **system call** è quella funzione che modifica il bit, alternando quindi gli stati da utente (ISA limitato) a kernel (ISA completo). Le funzioni che vengono eseguite durante la fase kernel non sono scritte dall'utente, bensì dai programmatori del sistema operativo. Pertanto l'esecuzione di un programma utente ha un andamento di questo genere:



Le system call

Parlando in maniera più dettagliata delle system call, possiamo dire che ve ne sono di cinque tipi diversi. Esaminiamoli. Si ricorda che le system call vengono attivate tramite interrupt (hardware).

*** System call: processi**
Le system call riguardanti i processi sono svariate, esse mettono a disposizione le seguenti funzionalità:
- Creare un processo (quindi il controllo delle risorse)
- Terminare un processo (utilizzare al meglio le risorse occupate)
- Abort (viene interrotto bruscamente un processo e ne viene generata una immagine - dello stack e dell'heap ad esempio - utile per il debugging)
- Cambio di programma (un processo potrebbe essere gestito da più programmi)
- Fork (generazione di figli da processi padri)

*** System call: file**
Le system call riguardanti i file sono:
- Creazione del file (gestione delle memorie)
- Cancellazione di un file
- Accesso / scrittura / lettura / copia

*** System call: device**
Per parlare di queste system call è necessario prima di tutto dire che i devices sono governati da due entità differenti: il **controller** (hardware) ed il **driver** (software). Le system call quindi dovranno permettere la comunicazione tra le due parti, seguendo questi passi:
1) Il driver carica i registri del controller (ad esempio viene capito se è stata richiesta una scrittura piuttosto che una lettura)
2) Il controller esamina i registri e decodifica le operazioni da svolgere
3) Il controller effettua il trasferimento dei dati
4) Il controller avvisa il driver che l'operazione è conclusa
5) Il driver avvisa il sistema operativo che l'operazione è conclusa

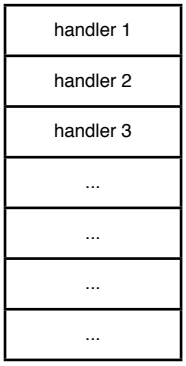
Le system call dei devices comprendono anche alcuni messaggi riguardanti l'utilizzo del device in maniera riservata.

* **System call: informazioni**
Il sistema operativo si fa carico di memorizzare importanti informazioni: per questo ci sono delle system call appropriate.
- Data ed ora
- Meta informazioni (un chiaro esempio è il comando ls -l, che restituisce tutte informazioni)

* **System call: comunicazione fra i processi**
I processi hanno bisogno di comunicare, ad esempio per poter lavorare insieme. Pertanto la comunicazione fra i processi va gestita da un elemento "superiore", quale è il sistema operativo. Vengono pertanto gestiti:
- Scambio di messaggi (vi sono due metodi, rappresentabili metaforicamente come la lavagna [uno o pochi scrivono e tutti leggono sulla stessa risorsa] e gli sms [necessaria una identificazione e un canale di trasmissione])
- Memoria condivisa

La gestione delle system call: handler

Come già detto, in seguito ad una system call, la richiesta viene inviata ad un **dispatcher** che poi ridireziona all'handler corretto per la system call in questione. Ma come sono organizzati gli **handler**?



Si ha un array di handler (detto "vettore di interruzioni") con dimensione prestabilita (dato che le system call sono in numero finito e conosciuto) che contiene tutti gli handler.
Quando il dispatcher deve "passare la palla" ad uno di essi, deve semplicemente fare un accesso in memoria alla giusta cella dell'array.

Per ottimizzare ancora di più questa struttura dati, l'indice **i** che scorre l'array è anche l'identificatore dell'evento.

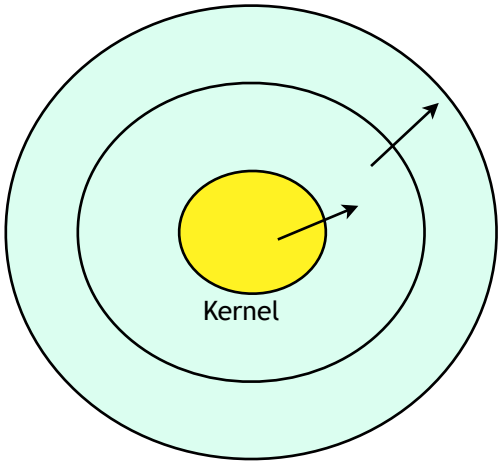
Pertanto vale l'equivalenza: identificatore evento = indice nel vettore delle interruzioni.

La struttura dei sistemi operativi

Vi sono principalmente tre maniere differenti in cui il sistema operativo potrebbe essere organizzato:
- Stratificazione
- A micro-kernel
- A moduli

Vediamoli nel dettaglio

*** I sistemi operativi stratificati**
Si parte dall'idea che vi sia un nucleo centrale in grado di eseguire alcune operazioni base, piuttosto semplici (poiché in diretto collegamento con l'hardware). Viene quindi costruito un livello superiore che astrae il nucleo ed aggiunge delle funzionalità (basandosi, però, sempre sulle vecchie funzioni del nucleo). Questa stratificazione si può fare molte volte, creando così una sorta di cipolla (che ci ricorda la struttura ad oggetti del linguaggio Java).

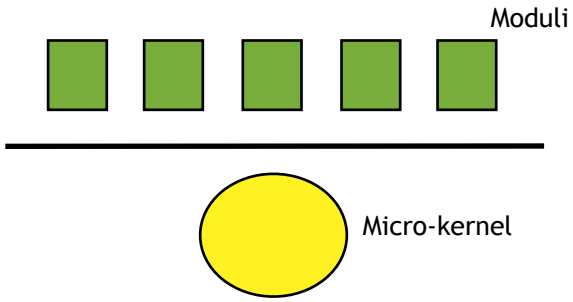


✓*Vantaggi:*
La stratificazione garantisce una divisione chiara delle funzionalità, rendendo quindi il debugging semplice (se sono sicuro che gli strati più a fondo funzionano bene, posso cercare solo negli strati superficiali).
✓*Svantaggi:*
Ogni funzione fa affidamento su funzioni basilari più interne e questo ovviamente comporta una grande quantità di record di attivazione sullo stack con conseguente diminuzione delle prestazioni.

Questo metodo è stato abbandonato nel corso degli anni, ma nell'ultimo periodo è stato rivalutato per i suoi utili risvolti in due applicazioni:
Macchine virtuali
Tali macchine stratificano un hardware "simulato" sopra al sistema operativo, permettendo così l'installazione di diversi SO.
Green computing
In grandi infrastrutture spesso si hanno diversi server chiusi in sale refrigerate. Col passare degli anni, le macchine divengono obsolete, pertanto si può optare per l'acquisto di un server di ultima generazione che è in grado di svolgere le mansioni di numerosi altri server, gestiti questa volta tramite macchine virtuali. Questo genera un risparmio di tipo ecologico: a pari prestazioni si consuma meno elettricità (sia per la refrigerazione che per la macchina stessa).

*** I sistemi operativi a micro-kernel**

L'idea di base di questo sistema è di ridurre le mansioni del nucleo centrale, limitandolo alla sola gestione dei processi, della RAM e della comunicazione. Tutte le altre funzionalità sono distribuite su altri moduli.



✓*Vantaggi:*

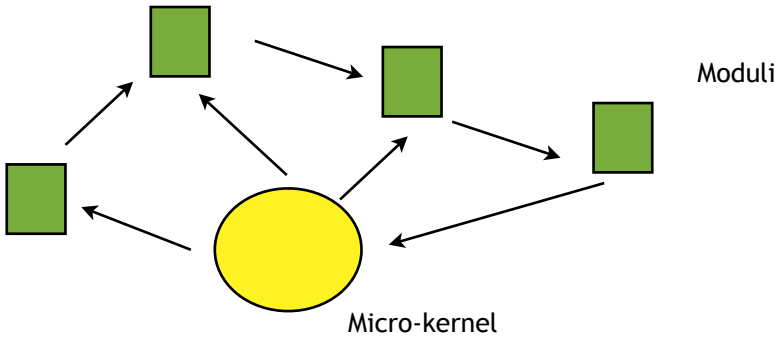
Essendo un sistema modulare, è possibile aggiungere e rimuovere facilmente parti più o meno utili.

✓*Svantaggi:*

Se il modulo due dovesse necessitare di una funzionalità gestita dal modulo due, dovrebbe passare per il micro-kernel ben due volte (e questo chiaramente influisce negativamente sulle prestazioni).

*** I sistemi operativi modulari**

Il sistema è molto simile a quello precedente, ma i moduli questa volta hanno la possibilità di comunicare direttamente fra loro.



Sistemi Operativi

I Processi

Capitolo 2

Indice degli argomenti

Informazioni sui processi

Un processo potrebbe essere riassunto nella definizione di “programma in esecuzione”.
Come è intuibile, di tale programma il sistema operativo deve conoscere diverse informazioni tra le quali:
1) ID (sarebbe il PID leggibile tramite il comando top)
2) Program Counter
3) Programma (qual’è il programma che utilizza tale processo) - detta “sezione-testo”
4) Stack/Heap di esecuzione - detta “sezione-dati”
5) Stato del processo (si veda sotto)
6) Copia dei registri della CPU
7) Informazioni sull’uso della CPU

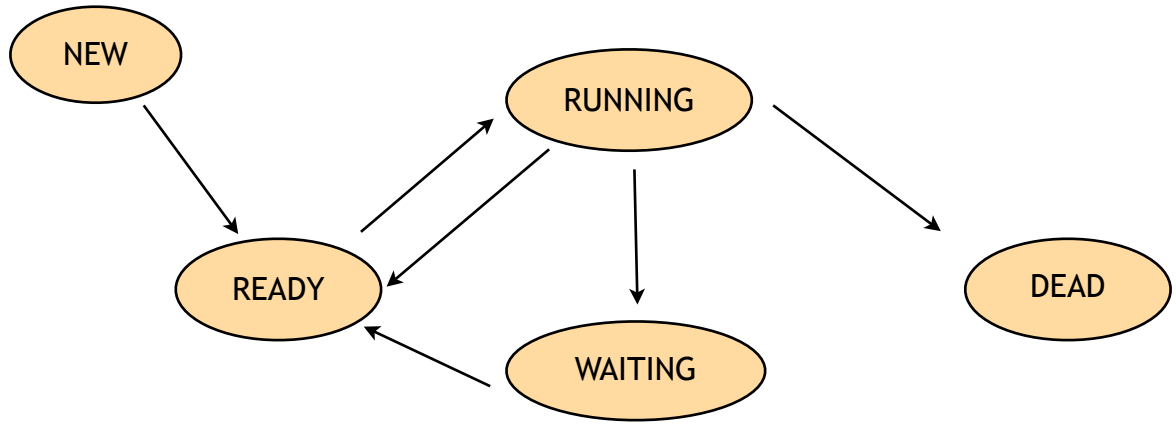
Tutte queste informazioni sono conservate nella PCB (Process Control Block): una parte della memoria. Quando avviene un context switch vengono caricate le informazioni del PCB del processo in corso.

Lo stato dei processi

Supponendo che tre programmi siano in esecuzione (ciascuno di essi coi suoi processi), l’illusione data ai tre programmi è quella di essere “simultaneamente” attivi (anche se avendo una sola CPU la cosa è impossibile). In realtà ciò che accade è l’**avvicendamento della CPU**.

I processi hanno, concettualmente, tre stadi differenti: esecuzione, attesa CPU ed attesa di letture/scritture (I/O). Avvengono continuamente **transizioni di stato**, ovviamente regolate dal sistema operativo.

Più precisamente gli stadi sono i seguenti:

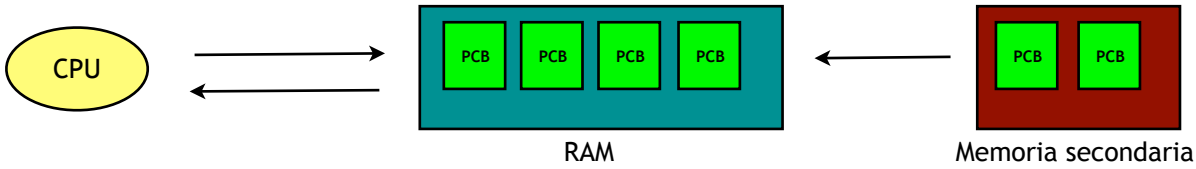


Esaminiamoli!
- **New:** il processo è appena stato creato.
- **Ready:** il processo è pronto ad essere eseguito dalla CPU. Questo stato potrebbe essere indotto dal sistema operativo, muovendo quindi il processo da running a ready in maniera forzata (probabilmente perché un processo più importante deve utilizzare la CPU).
- **Running:** il processo ha il controllo della CPU.
- **Waiting**: il processo sta attendendo di ricevere dati di I/O, quindi lascia “di sua volontà” la CPU per permetterne l’utilizzo ad altri processo.
- **Dead:** il processo è concluso.

Lo scheduling

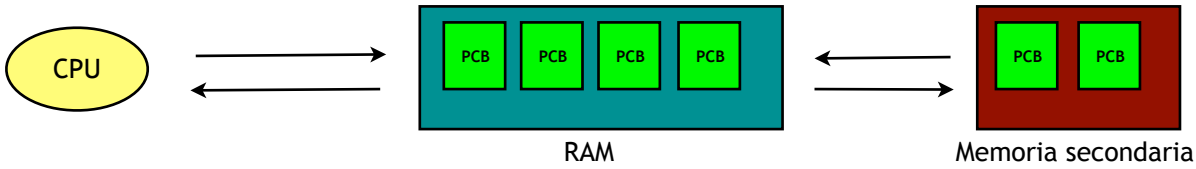
Abbiamo parlato di PCB, il contenitore delle informazioni di ogni processo. Avendo un sistema multi-tasking, è necessario poterne immagazzinare diversi: questo avviene nella RAM. Diversi PCB sono immagazzinati nella RAM contemporaneamente, uno soltanto dei quali è assegnato alla CPU.
Sorge pertanto spontanea la domanda: come funziona il cambio di processo? Secondo quali principi? Vediamolo.

Lo scheduling a lungo termine
Manteniamo dentro alla RAM un numero limitato di PCB, quelli in eccesso vengono invece immagazzinati in una memoria secondaria (più lenta della RAM).



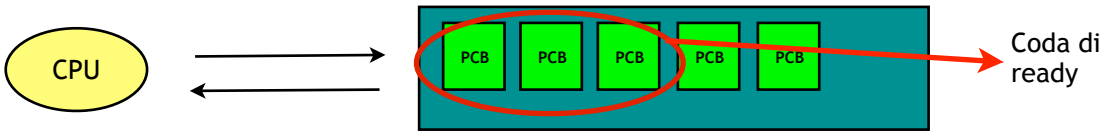
L'unico modo con cui un processo contenuto nella memoria secondaria possa passare alla RAM è che uno dei processi della RAM **termini**, quindi si "liberi un posto".
La scelta sul nuovo processo entrante è dettata dal fatto che si tende a creare un equilibrio tra processi detti **CPU-bound** (che usano molto la CPU, come potrebbe essere la risoluzione di una complessa equazione) e quelli **I/O-bound** (processi che usano principalmente I/O, come potrebbe essere la lettura di un file).

Lo scheduling a medio termine (swapping)



Questo scheduling è sostanzialmente una modifica del precedente, infatti viene introdotta la possibilità di far entrare in RAM processi non solo quando un posto è vuoto, bensì anche effettuando uno **scambio** tra RAM e memoria secondaria. Questo significa poter portare avanti più processi contemporaneamente (poiché quelli in memoria secondaria non sono "confinati"), ma anche rallentare in generale l'esecuzione dei processi in quanto un processo in memoria secondaria dovrà prima passare nella RAM per poter accedere alla CPU.

Lo scheduling a breve termine (scheduling della CPU)
Pensiamo a una soluzione totalmente differente: gli unici processi ad aver diritto all'accesso alla CPU sono quelli in ready. Viene quindi generata una "sotto-coda" dentro alla RAM, detta coda di ready.



Questa soluzione porta però alla luce molte problematiche, come ad esempio: quale processo merita la CPU? Per quanto tempo? Quando?
Sono pertanto necessarie delle **politiche di scheduling.**
In ogni caso, una volta determinato quale sia il prossimo processo viene interpellato il **dispatcher di context switch** che assegna il nuovo processo alla CPU.

Le politiche di scheduling (nello scheduling a breve termine)

Le politiche di scheduling sono semplicemente delle regole che guidano i processi: più precisamente quale tra questi "meriti" la CPU prima degli altri. Le politiche di scheduling sono quindi l'insieme delle **condizioni che fanno scattare lo scheduler** e dei **criteri di selezione**.
Enunciamo prima quali politiche tratteremo, quindi studieremo quali condizioni potrebbero essere utilizzate e poi entreremo nel dettaglio di ogni politica.

1) First-Come First-Sort (FCFS)
2) Shorter Job First (SJF)
3) Round Robin
4) Priorità (generale)
5) Code multilivello con/senza feedback

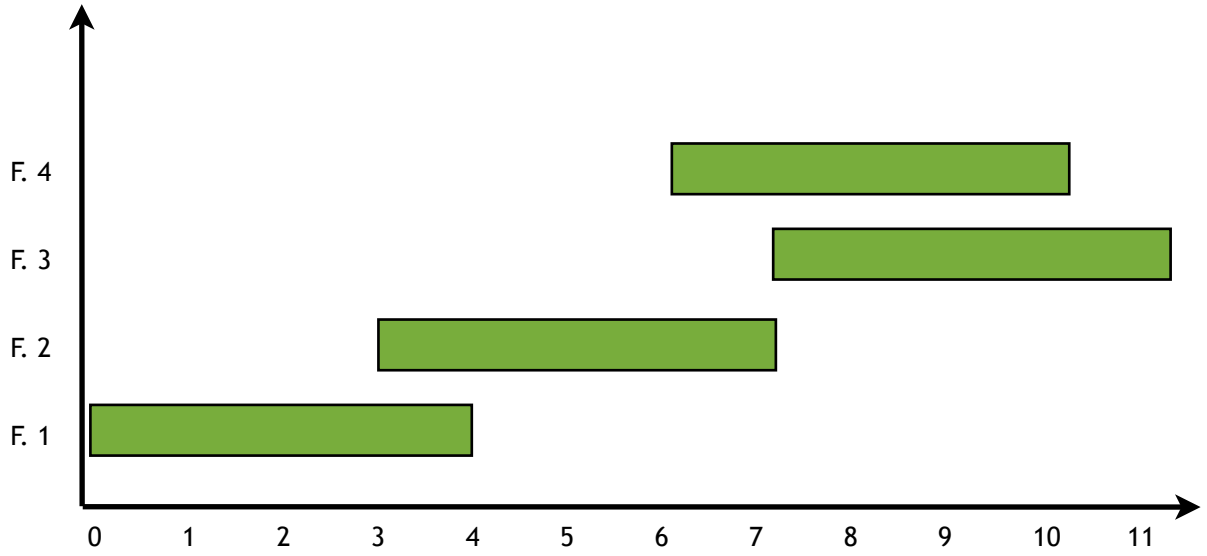
Passiamo ora ad alcuni concetti necessari per capire le successive spiegazioni.

La prelazione
Dicesi prelazione l'azione di privare un utilizzatore della sua risorsa nonostante esso voglia continuare ad adoperarla. Nel nostro caso, un processo sta utilizzando la CPU, vorrebbe continuare ma viene privato della risorsa (ovviamente per motivi di ottimizzazione). Dicesi **preemptive** una politica che utilizza prelazioni, mentre una politica che non ne fa uso viene detta **non-preemptive.**
La prelazione è possibile grazie ad un supporto hardware, chiamato timer.

Le unità di misura
Le unità misura utilizzate per considerare le prestazioni sono:
1) Throwput --> processi completati nell'unità di tempo
2) Turn around time --> tempo intercorso tra l'inizio di un processo e la sua fine
3) Tempo di attesa --> somma del tempo passato nello stato "ready" (tempi morti)
4) Tempo di risposta --> tempo intercorso tra l'invio di una query da parte dell'utente e la risposta del computer.

Principi secondo i quali deve avvenire un context switch
1) Il processo in running termina
2) Il processo in running va in waiting
3) Un processo in waiting diventa ready (ad esempio ci sono dati volatili da gestire, quindi merita attenzione). *Prelazione!*
4) Il processo running diventa ready (qualcosa ha "rubato" la CPU al processo). *Prelazione!*

Diagrammi di Gantt
I diagrammi di Gantt servono per descrivere l'evolversi dell'esecuzioni di determinate fasi nel tempo. Una ottima applicazione è la progettazione di un evento qualsiasi, diviso appunto in fasi.



Come possiamo vedere la fase 1 finisce all'istante 4, (ma la due inizia all'istante 3), e così via. Questo sistema è molto intuitivo.

Ben sapendo che la CPU può essere "usata" da un processo soltanto nello stesso istante di tempo, non vi saranno sovrapposizioni tra le "fasi" (una fase equivale al tempo in cui un processo è in stato di running) del diagramma di Gantt rappresentante l'andamento dei processi. Possiamo quindi rappresentare tutti i processi "in linea temporale" in questa maniera:

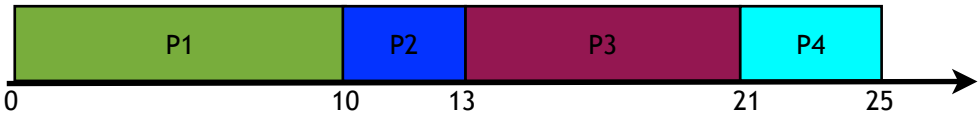
Si noti che il tempo in cui P1 è nello stato "running" viene detto **CPU-burst**.

Passiamo ora a esaminare le politiche di scheduling precedentemente elencate.

Politica First-Come First-Sort *(non-preemptive)*
La coda ready viene gestita in maniera FIFO (first-in, first-out) cioè in semplice ordine di arrivo (come in un supermercato). I processi vengono quindi eseguiti senza particolari priorità, uno dopo l'altro.
Per comprendere le prestazioni di questa politica, prendiamo una tabella "data", dove troviamo i tempi di CPU-Burst.

P	CPU-B
P1	10
P2	3
P3	8
P4	4

Diagramma di Gantt:

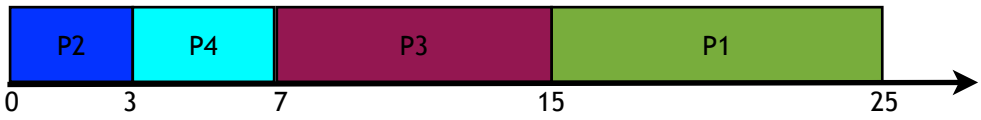


Calcolando il tempo medio di attesa (d'ora in poi TMA), si ottiene (0+10+13+21)/4 = **11**.

Politica Short Job First *(non-preemptive)*
Come si è visto, il TAM si basa sulla somma dei tempi di attesa. Ma allora se facessimo "passare" prima i processi più corti, forse potremmo ridurlo! Cioè l'ordine dei processi è in qualche maniera rilevante?
La risposta è sì. Vediamolo.
Modifichiamo la coda, ordinando i processi per CPU-B (dal più veloce al più lento, facendo passare prima il più veloce).

P	CPU-B
P1	10
P2	3
P3	8
P4	4

Diagramma di Gantt:



Il TMA questa volta è (0+3+7+15)/4 = **6,25**.

Incredibile! Il tempo è quasi dimezzato! Ricordiamo che a quel tempo va comunque sommato il tempo di computazione derivante dal processo di inserimento in maniera ordinata (a seconda del CPU-Burst) dei processi nella coda-ready. I principi su cui si basa la politica per stabilire i context switch sono i punti 1 e 2 elencati prima.

Ma chi calcola il tempo di CPU-Burst? Da dove viene?
Il tempo di CPU-Burst

Il tempo di CPU-Burst viene definito tramite delle operazioni "statistiche" che fa il sistema operativo. Abbiamo la formula:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\cdot\tau_n$$

Tale formula è chiaramente ricorsiva, e abbiamo che: τ_n è la previsione a livello ennesimo (precedente),

t_n è la durata del CPU-burst ennesimo e α è un numero compreso tra 0 e 1 (tendenzialmente esclusi).

In sostanza, i termini iniziali (più vecchi) perdono di significato man mano che si va avanti, mentre invece i termini più recenti sono quelli che valgono di più.

Politica Shortest Remaining Time First *(preemptive) - variante della politica SJF*
Viene introdotto un principio che richiede prelazione, cioè il passaggio da waiting a ready (il punto 3).
In questo modo viene sostanzialmente introdotto il context switch nel caso in cui il processo che è in running richieda ancora più tempo per essere eseguito rispetto ad un processo che è stato appena introdotto e che è evidentemente più veloce.
Cioè ad esempio se stiamo eseguendo un processo che dura 5 unità di tempo, e siamo alla seconda, se dovesse arrivare un processo che occupa una sola unità di tempo questo avrebbe la precedenza e avverrebbe quindi una prelazione (poiché 1 < 3).

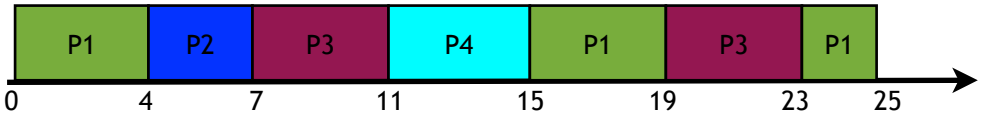
Il concetto di starvation
Tipico delle scheduling con priorità è il problema dello starvation: supponiamo che ci sia un processo che richiede 5 unità di tempo. Supponiamo ora per assurdo che arrivino continuamente processi più corti: il processo da 5 unità non verrebbe mai eseguito. Per impedire questo fenomeno viene introdotto il sistema di **ageing** (invecchiamento) che media la priorità dei processi tenendo anche conto del tempo in cui sono stati in coda.

Politica Round Robin (*preemptive*)
Questa politica è usata spesso in sistemi **time-sharing** (con più utenti da servire). Sostanzialmente la coda è organizzata come la politica FCFS, ma viene aggiunta una nuova particolarità: il quanto di tempo.
Il quanto di tempo è sostanzialmente una costante che viene impostata dal sistema operativo, la quale ha il ruolo di riassegnare la CPU una volta passate x unità di tempo.
Si crea pertanto una rotazione (ecco il motivo del nome).
In sostanza, se il processo in running ha un tempo di burst superiore al quanto di tempo, una volta passate le x unità di tempo definite dal quanto stesso il processo viene rimesso in coda-ready (prelazione).
Questo sistema risolve il problema dello starvation.
La scelta del quanto di tempo chiaramente è discriminatoria... con un quanto troppo grosso, il sistema Round Robin sostanzialmente non esiste e diventa un semplice FCFS. Con un quanto troppo piccolo invece il numero di context switch aumenterebbe incredibilmente, rallentando il sistema.

Vediamo ora dal lato prestazionale quali sono i vantaggi/svantaggi. Quanto di tempo = 4 unità.

P	CPU-B
P1	10
P2	3
P3	8
P4	4

Diagramma di Gantt:



Il TMA questa volta è più complesso da calcolare, pertanto calcoleremo a parte i singoli TA e poi ne faremo la media.
TA: P1=(0+11+4)=15; P2=4; P3=(7+8)=15; P4=11. TAM = (15+15+11+4)/4 = **11,25**.

Come si nota, le prestazioni non sono migliorate, anzi. Questo potrebbe sia dipendere dal quanto "poco appropriato", ma in ogni caso abbiamo avuto il vantaggio di "servire" i processi in maniera equilibrata.
Tramite questo sistema è un po' come se avessimo per ogni processo una CPU che lavora ad 1/n (dove n è il numero di processi in coda), poiché il processo n per essere eseguito, deve necessariamente attendere almeno n-1 quanti.

Politica di code multilivello con/senza feedback
Di questa politica è sufficiente sapere che vengono formate delle categorie a seconda della priorità che si vuole assegnare ai processi. Ogni categoria avrà una coda a se stante, con una sua politica indipendente dalle altre. L'accezione "feedback" significa che il processo può, durante la sua esistenza, cambiare categoria.

La creazione di un processo

Tutti i processi (tranne uno) vengono creati da altri processi. Data questa implementazione, i processi possono essere rappresentati in un albero.
Il primo processo a nascere è sempre l'**init**, il quale ha PID=1. Da esso si generano i processi del sistema operativo, ecc.

Fork
Appena un processo padre genera un processo figlio, viene generata una PCB che viene messa in RAM. Le due PCB sono del tutto uguali, anche se totalmente indipendenti ed autonome. Il fenomeno di creazione di un processo figlio da parte di un processo padre è detto **fork** (biforcamento). Nella maggior parte dei casi, un processo ne genera un altro per avere un "aiuto" nello svolgimento di una qualche mansione: per questo motivo appena un processo figlio nasce esegue lo stesso codice del padre.

Zombie
Quando un processo figlio termina, il padre ha "diritto" di avere accesso al suo PCB (per ricevere dei dati, magari), pertanto per un po' di tempo la PCB dei processi appena terminati rimane in RAM.

Wabbit
La creazione sconsiderata di processi figli (che a loro volta ne generano altri), può portare ad un riempimento della zona della RAM riservata ai PCB. Questo fenomeno, detto **wabbit**, impedirà al computer di eseguire qualsiasi nuovo processo (quindi rimarrà impallato).

La cooperazione dei processi
Come detto sopra, un processo figlio può essere utile al padre in qualità di collaboratore. Proviamo a vedere un esempio.

Dobbiamo eseguire una certa operazione, y=g(f(x)). Un programma monolitico (per un solo processo) apparirebbe in questa forma:

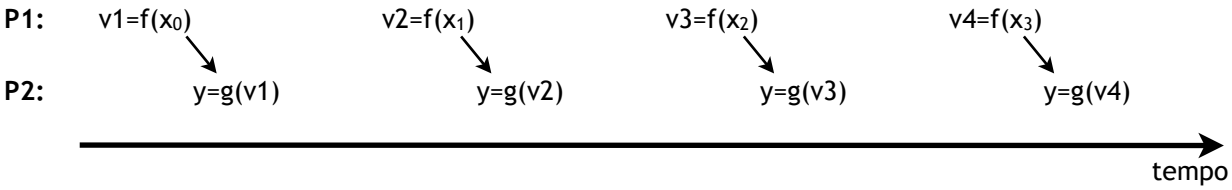
```
loop
      read(x)
      v1=f(x)
      y=g(v1)
      write(y)
end
```

Ma avendo la possibilità di utilizzare due processi cooperanti, potremmo distribuire le operazioni in questo modo:

P1 *(produttore)*	**P2** *(consumatore)*

`loop`
`  read(x)`
`  v1=f(x)`
 	
`  *invia i dati*`
 	
`end`
	`loop`
	`  *ricevi i dati*`
	
	`  y=g(v1)`
	`  write(y)`
	
	`end`

In questo modo, su una linea temporale, avremo:



Come si vede, gli eventi sono sfalsati ma anche incastrati, quindi c'è un risparmio di tempo. Questo risparmio è ottenibile recuperando il tempo delle letture/scritture.

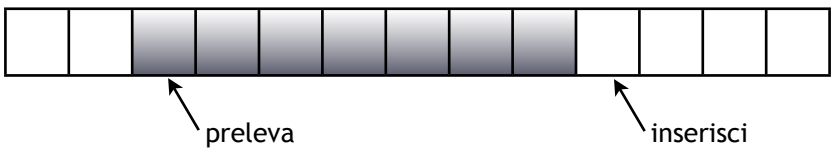
È oramai palese che i processi abbiano bisogno di **comunicare** per poter effettuare queste operazioni insieme, per poter in sostanza cooperare. Vediamo quindi come possono farlo.

La comunicazione tra processi

Vi sono due tipi di comunicazioni (anche se il secondo si basa in buona parte sul primo):
1) Comunicazione a memoria condivisa (*es: lavagna*)
2) Comunicazione a scambio di messaggi (*es: SMS*)

Comunicazione a memoria condivisa
Supponiamo di avere una zona della RAM particolare, alla quale hanno accesso più processi. I processi vedranno questa zona come una variabile. Data la particolarità di questa zona, sarà il sistema operativo a stanziarla.
Questa area di memoria è paragonabile a una struct del genere:

```
struct {
      elementi dato[D]; //buffer
      int inserisci, preleva; //indici
}
```



Questa struttura dati è un classico buffer, con due indici: preleva segna il primo dato da prelevare, mentre inserisci indica la prima posizione vuota del buffer.
A questo punto non resta che definire i due codici che dovranno essere riportati nei processi:

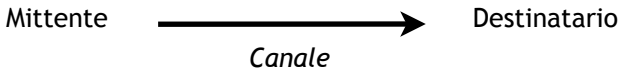
P1 *(produttore)*	**P2** *(consumatore)*

`if(!pieno(b)){`
`  b.dato[b.inserisci]=v1;`
`  b.inserisci=(b.inserisci+1)%D;`
`}`
	`if(!vuoto(b)){`
	`  v1=b.dato[b.preleva];`
	`  b.preleva=(b.preleva+1)%D;`
	`}`

Problemi generali di questo sistema
Questo sistema funziona molto bene, ma ci sono alcuni problemi che vanno considerati. Primo tra tutti è il fatto che una context-switch potrebbe interrompere P1 o P2 a metà del corpo dell'if, impedendo il corretto **aggiornamento** dei due **indici**, rendendo così il buffer inconsistente.

L'unica maniera di risolvere il problema, sarebbe ignorare ogni context-switch durante l'esecuzione di quelle due righe di codice... scopriremo in futuro come farlo.

Comunicazione a scambio di messaggi
Per realizzare questo tipo di comunicazione necessitiamo di un **canale** (mezzo di propagazione). Intuitivamente, questo sistema vede da parte del mittente (diciamo P1) una funzione del tipo **`send`**`(PDI, "text")`, e da parte del destinatario un'altra funzione del tipo **`receive`**`(PDI, var)`.



Ma in questo modo, i due processi dovrebbero conoscere esattamente il PDI dell'altro processo con cui vogliono avere una comunicazione, e questo dato non è così facile da reperire. Così viene introdotto anche qui una memoria condivisa, che è però soltanto una coda di messaggi. Tale coda ha un ID, pertanto le funzioni diverranno `send(ID_CODA, "text")` e `receive(ID_CODA, var)`. I due processi devono ora conoscere soltanto l'ID della coda, non più il PDI del processo.

Il mittente ed il ricevente possono avere approcci differenti: **sincrono** oppure **asincrono**. Nel primo caso, il mittente attenderà che il ricevente possa ricevere il suo messaggio, mentre nel caso asincrono semplicemente "deposita" il messaggio non curandosi di nulla.
Nel caso del destinatario, invece, il sincrono attende di ricevere un messaggio da parte del mittente, mentre nel caso asincrono il messaggio viene ricevuto senza preavviso.

Se entrambi hanno approccio sincrono, la comunicazione viene detta **rendez-vous**.

La coda di messaggi può essere configurata in tre modi differenti:
1) 0 spazi (il mittente ed il destinatario usano la coda semplicemente come punto di incontro)
2) N spazi (possono essere depositati fino a N messaggi)
3) Infiniti (quando la coda è piena, viene allocata altra memoria contigua per accodare altri messaggi)

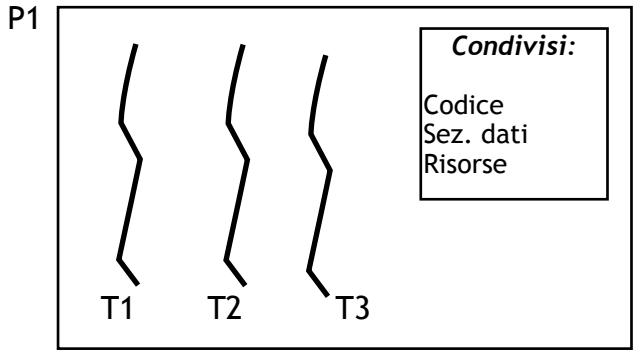
Esempi di applicazione di comunicazione a messaggi
I socket: tramite i socket, più pc possono collegarsi insieme ed inviarsi stream di bytes. Questo significa che su entrambe le macchine dovranno essere impostate delle codifiche per tradurre gli stream.

RPC (remote process control): Un processo P1 che si trova sulla macchina PC1, può utilizzare un processo P2 che si trova invece sulla macchina PC2. Tutto questo avviene grazie allo "stub", che indica su quale macchina è presente il P2 desiderato, e ovviamente alla connessione internet.

I thread

Una degli aspetti che per il momento non abbiamo considerato, è la presenza di parti ancora più piccole dei processi: i thread. Thread significa "filo" in inglese.

Sappiamo bene che un processo ha salvato nel suo PCB riferimenti ad un'area testo, un'area dati, un PC, uno stack, ecc.
I thread, (che vivono solamente all'interno di un processo), sono elementi non autonomi, e cioè che condividono parte delle loro informazioni con altri thread. Infatti ogni thread possiede solamente uno stack, un PC, un ID e i valori dei registri della CPU. Come è possibile?



Le parti mancanti del thread ci sono, ma sono condivise tra tutti i thread del processo.

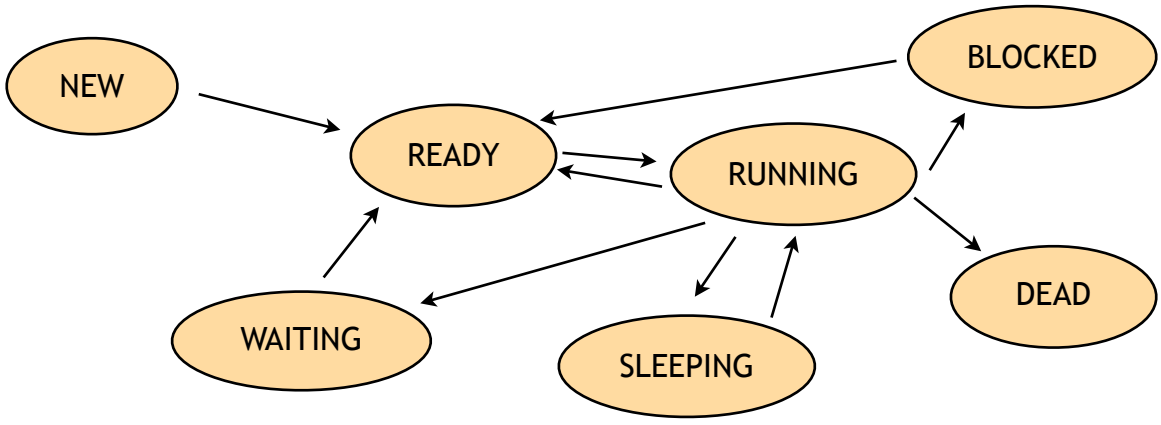
Questo tipo di implementazione porta dei vantaggi immediati: i context-switch tra thread sono molto più veloci poiché non viene sostituito l'intero PCB, ma solo una parte. Chiaramente utilizzare delle memorie condivise significa anche saperle utilizzare attentamente, quindi si aumenta la complessità del tutto.

Altro vantaggio dei thread è il fatto che non devono essere gestiti dal sistema operativo.

Nascita, vita e morte di un thread
Quando un processo nasce, esso ha di default un solo thread (dicesi **High-Weight Process**). Durante l'esistenza del processo è possibile che i thread aumentino di numero, oppure che resti solo il primo. Due thread possono eseguire l'operazione di **joint**, cioè sostanzialmente uno dei due attende l'altro per "morire" e fondersi con il primo.
I thread possono eseguire l'operazione di **exit** (muoiono tutti i thread, poiché il processo stesso termina). Se un processo viene forkato, allora il processo P' potrebbe avere stesso numero di thread, oppure anche essere vuoto.

In generale, esistono **metodi di sincronizzazione** tra thread (li vedremo in seguito) che permettono una esecuzione vantaggiosa.

il diagramma di stato del thread



Esaminiamoli!
- **New:** il thread è appena stato creato.
- **Ready:** il thread è pronto a prendere possesso della CPU.
- **Running:** il thread sta utilizzando la CPU.
- **Blocked**: equivalente del waiting dei processi, mentre il thread attende qualche dato I/O viene messo in questo stato
- **Waiting**: il thread è in attesa di un evento (tipicamente un evento di sync)
- **Sleeping**: il thread si "assopisce" per un tempo predefinito.
- **Dead:** il thread è morto.

Il rapporto tra il sistema operativo ed i thread

La prima implementazione dei thread della storia, era sostanzialmente una libreria del linguaggio di programmazione che si stava usando. In sostanza i thread erano visibili solamente a livello utente, il sistema operativo era invece in grado di trattare solamente processi.

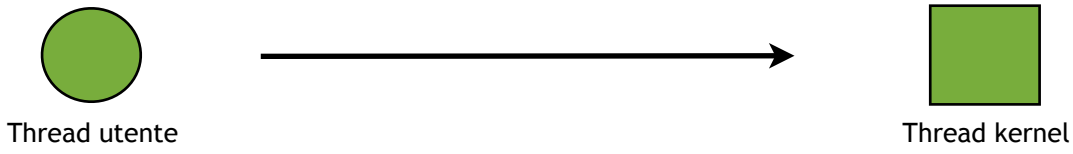
Questa soluzione, per quanto interessante (dare la gestione all'utente dei thread significa poter ottenere programmi ancora più performanti), non è una grande miglioria: infatti se uno dei thread di uno dei processi fosse in blocked, metterebbe in waiting tutti gli altri thread (quindi tutti gli altri processi a catena) rallentando comunque l'esecuzione.

È stata così pensata una soluzione più complessa ma certamente più performante: i **thread kernel**.

Dando la possibilità al sistema operativo di riconoscere (e manipolare) i thread, chiaramente si sono dovute implementare nuove strutture dati e nuove strutture di controllo, senza contare le modifiche da apportare allo scheduling... ma i context-switch sono stati resi così molto più veloci.

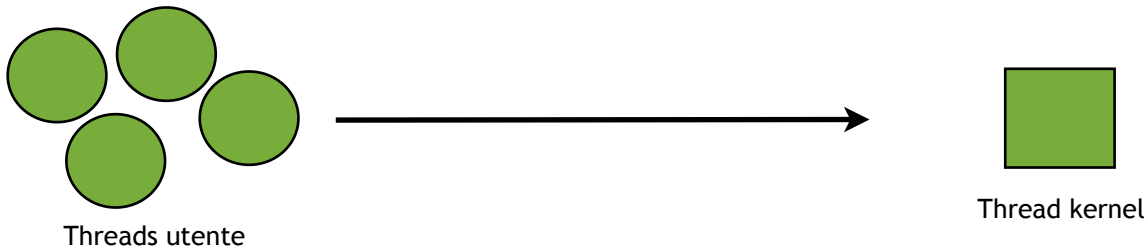
Vi sono tre diversi modi con cui i thread possono rapportarsi al sistema operativo:

Metodo uno a uno



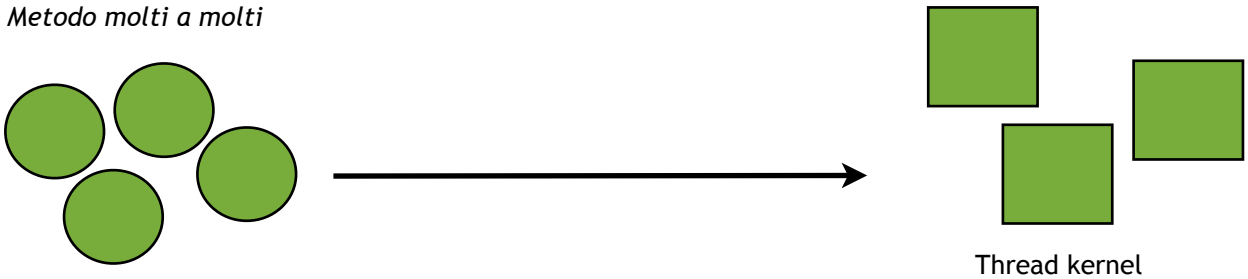
Non ci sono problemi di comprensione, il thread verrà eseguito dal sistema operativo così come l'utente l'ha concepito.

Metodo molti a uno



Uno scheduler imposterà le priorità dei thread utente, che vedranno il thread kernel come risorsa da "usare in cooperativa".

Metodo molti a molti



Con questo metodo diversi thread utenti si interfacciano con diversi thread kernel (solitamente in numero minore rispetto a quelli utente). Tale tipo di rapporto viene chiamato **Light-Weight Process**.

Anche qui ci sono due scheduler che impostano le priorità (rispettivamente **process contentation mode** e **system contentation mode**). Ma c'è da dire che quando uno dei thread utenti che ha assegnato uno dei thread kernel finisce per qualche motivo in block, il sistema operativo deve effettuare una **upcall** al processo utente che uno dei suoi thread è inattivo e quindi quel processo può riassegnare la LWP ad un altro thread.

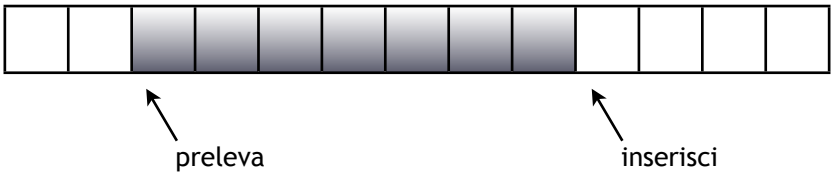
Sistemi Operativi
La sincronizzazione dei processi

Capitolo 3

Indice degli argomenti

Esecuzione concorrente asincrona

Veniamo ora alla risoluzione di uno dei problemi lasciati indietro poco tempo fa, quando si parlava di memorie condivise.
La situazione era questa:



P1 *(produttore)*	**P2** *(consumatore)*

`if(!pieno(b)){`
`  b.dato[b.inserisci]=v1;`
`  b.inserisci=(b.inserisci+1)%D;`
`}`
	`if(!vuoto(b)){`
	`  v1=b.dato[b.preleva];`
	`  b.preleva=(b.preleva+1)%D;`
	`}`

Vorremmo a tutti i costi impedire che queste due righe di codice si interrompano. Ricordiamo che questa parte di codice (cioè quella che comunica con la memoria esterna) viene detta **sezione critica**. La parte sopra alla sezione critica viene detta **sezione di ingresso** e quella dopo **sezione di uscita**.

Nella ricerca della soluzione cercheremo di tenere conto di tre aspetti indispensabili:
1) Mutua esclusione (solo un processo per volta può accedere alla memoria)
2) Progresso (se uno o più processi vogliono l'accesso alla risorsa, almeno uno di loro potrà ottenerlo)
3) Attesa limitata (non deve esistere lo starving)

Soluzioni software

Soluzione software 1 *(gestione con contesa fra due processi)*
Potremmo introdurre un codice nei due processi, che "tenga occupato" il processo che non ha accesso alla memoria di modo che non disturbi quello che invece sta operando su di essa. Abbiamo bisogno di una variabile "turno" globale che ci dica quale dei due processi ha diritto di esecuzione. Avremmo quindi:

P1 *(definito come i)*	**P2** *(definito come j)*

`while(turno==j) do nop;`
 	
`*sezione critica*`
 	
`turno = j;`
	`while(turno==i) do nop;`
	
	`*sezione critica*`
	
	`turno = i;`

Il codice è chiaramente intuibile: fintanto che il turno appartiene all'altro processo, non si esegue la sezione critica. Quindi, finito il ciclo, viene eseguita la sezione critica e poi viene ripassato il turno all'altro processo.

Questo metodo potrebbe sembrare perfetto, ma ha un grande punto debole. Supponiamo che P2 abbia il turno, quindi esegua la sua sezione critica, ripassi l'esecuzione al processo P1, e poi decida di smettere di usare la memoria condivisa (o comunque non la usa più). A questo punto, P1 avrà eseguito la sua sezione critica e ripasserà l'esecuzione a P2. Ma P2 non sta più eseguendo il codice di prima, quindi sostanzialmente non ridarà mai il turno a P1. Questo viola la regola numero due che ci siamo posti all'inizio.

Soluzione software 2 *(gestione con contesa fra due processi)*
Una soluzione alternativa per risolvere il problema al punto due, potrebbe essere quella di implementare un array di due elementi (chiamato flag) boolean. Ad esempio, flag[i] settata a true esprime il desiderio del processo P_i di eseguire la sua sezione critica. Avremo quindi un codice del genere:

P1 *(definito come i)*	**P2** *(definito come j)*

`flag[i]=true;`
`while(flag[j]) do no op;`
 	
`*sezione critica*`
 	
`flag[i]=false;`
	`flag[j]=true;`
	`while(flag[i]) do no op;`
	
	`*sezione critica*`
	
	`flag[j]=false;`

Questo sistema rispetta sia il punto uno che il punto due delle nostre specifiche. Putroppo però, se la sequenza eseguita fosse `flag[i]=true;` e poi in seguito ad una context switch, `flag[j]=true;` entrambi i processi aspetterebbero l'altro in eterno, contravvenendo quindi al punto tre della nostra specifica (i tempi di attesa).

Soluzione software 3 (algoritmo di Peterson) *(gestione con contesa fra due processi)*
Si è pensato di utilizzare entrambe le variabili viste prima, sia la flag che la variabile turno. Vediamone l'implementazione:

P1 *(definito come i)*	**P2** *(definito come j)*

`flag[i]=true;`
`turno=j;`
`while(flag[j] && turno==j) do no op;`
 	
`*sezione critica*`
 	
`flag[i]=false;`
	`flag[j]=true;`
	`turno=i;`
	`while(flag[i] && turno==i) do no op;`
	
	`*sezione critica*`
	
	`flag[i]=false;`

Anche se può sembrare contorto, l'assegnamento inverso del turno permette ai cicli di "ciclare" finché l'altro processo non imposti la sua flag a false, e quindi si sblocchi il while.
Questo metodo funziona con ogni combinazione di interlineamento tra istruzioni.

Questo metodo rispetta le nostre tre regole, ma ha comunque dei difetti: è solamente per due processi, ed inoltre prevede una attesa attiva (cioè l'altro processo cicla "a vuoto" mentre aspetta, si potrebbe investire meglio quel tempo!).

Soluzione software 4 (algoritmo di Lampart) *(gestione con contesa fra più processi)*
Estendendo semplicemente l'algoritmo di Peterson possiamo ottenere un risultato piuttosto complesso ma funzionante. Vediamo prima il codice e poi commentiamolo.

P1 *(definito come i)*	**P2** *(definito come j)*
00 choosing[i]=true; 01 ticket[i]=max_ticket+1; 02 choosing[i]=false; 03 for(j=0; j<N; j++) { 04 while(choosing[j]) no op; 05 while(ticket[j]!=0 && (ticket[j]<ticket[i] || (ticket[j]==ticket[i] && j<i))) no op; 06 } 07 *sezione critica* 08 ticket[i]=0;	00 choosing[j]=true; 01 ticket[j]=max_ticket+1; 02 choosing[j]=false; 03 for(k=0; k<N; k++) { 04 while(choosing[k]) no op; 05 while(ticket[k]!=0 && (ticket[k]<ticket[j] || (ticket[k]==ticket[j] && k<j))) no op; 06 } 07 *sezione critica* 08 ticket[k]=0;

Esaminiamo con calma il codice.
Le prime tre righe fungono sostanzialmente da "prendi ticket". Dobbiamo immaginare una struttura simile a quella del supermercato, in cui ogni processo può ottenere un biglietto numerato (contenuto nell'array di interi **ticket**). Dalle righe 00, 01 e 02 possiamo però capire che (in caso di interrupt) potrebbero tranquillamente capitare ticket di uguale numero a due o più processi. La variabile **max_ticket**, anch'essa globale, contiene il numero massimo del ticket attualmente in uso [non ci occupiamo della sua gestione].
L'array di boolean **choosing** ci indica chi sta prelevando un ticket.
Sappiamo come standard che ticket[i]=0 significa non avere interesse nell'eseguire la propria sezione critica.

Il for alla riga 03 non fa altro che scorrere tutti gli altri j processi (meno il corrente ovviamente). Questo è uno pseudo codice!
Il while alla riga 04 ci dice che se il processo j in questione sta scegliendo un ticket, attendiamo la fine della scelta (si ricordi che il controllo è effettuato su tutti i j processi).
Il while alla riga 05 ha una espressione booleana piuttosto complessa. La prima parte indica se il processo j ha interesse nell'eseguire la sua sezione critica (è != da 0). `ticket[j]<ticket[i]` significa invece "se il ticket di i è più grande del del ticket di j". L'ultima parte invece si riferisce all'uguaglianza tra i ticket (come detto prima) e quindi viene verificata la precedenza come ordinamento dei processi (j<i). Se tutto questo è vero, chiaramente bisogna far aspettare il processo. Altrimenti può eseguire la sua sezione critica, e quindi impostare il suo ticket a 0 (cioè non ha più interesse).

Questo sistema, seppur perfettamente funzionante, è piuttosto complesso e ed è praticamente inutilizzato.

Soluzioni hardware

Abbiamo sostanzialmente due soluzioni hardware possibili. L'idea principale è quella di disabilitare gli interrupt in una porzione di codice, in modo da avere una atomicità vera e propria. Questa implementazione però vuole anche dire cercare di ridurre all'osso il codice "atomizzato", poiché disabilitare gli interrupt non è una azione molto "bella". Vediamo le due tecniche nel dettaglio, ricordando sempre che i due codici sono indivisibili ed eseguiti durante la disabilitazione degli interrupt.

Test and Set

Abbiamo una variabile globale condivisa boolean **lock** (lucchetto). Il significato di quella variabile è, in caso di true, l'assegnamento di una data risorsa ad un dato processo. Vediamo il codice.

Test and Set

```
testAndSet(boolean *var){
  boolean valore=*var;
  *var = true;
  return valore;
}
```

Questo semplice codice ci permette di returnare il valore attuale di una variabile booleana, modificandone però sempre il contenuto in "true".

Il processo riporterà quindi questo codice:

P1

```
while(testAndSet(&lock))
no op;

*sezione critica*

lock=false;
```

In sostanza viene eseguita la funzione testAndSet, con il valore di lock. Se qualcuno sta già utilizzando lock, verrà ritornato il valore "true" (quindi il ciclo proseguirà). Nel caso in cui però lock=false, la funzione returna false (quindi il while viene saltato) ma la variabile viene settata a true in modo da impedire a tutti gli altri di utilizzare la risorsa. Questo metodo è ottimale, ma purtroppo lascia aperta la possibilità di starvation.

Swap

Utilizziamo questa volta due variabile, **lock** e **key.** Attenzione però! La prima variabile è sempre condivisa, mentre invece key è propria di ogni processo. Vediamo subito i due codici.

Swap

```
testAndSet(boolean *a, boolean *b){
  boolean temp = *a;
  *a=*b;
  *b = temp;
}
```

Questo codice non fa null'altro che scambiare il contenuto di due variabili, a e b. Vediamo come utilizzarlo all'interno del processo.

P1

```
key=true;
while(key) swap(&lock, &key)

*sezione critica*

lock=false;
```

Un processo imposta la sua chiave a true, cioè desidera eseguire la sua sezione critica. A questo punto, fintanto che la sua key è a true, effettua uno scambio tra lock e key. Nel momento in cui lock sarà falsa, verrà scambiata in uno dei cicli del while con la key, quindi la condizione del while sarà falsa, e quindi potremo eseguire la sezione critica. Lo scambio avrà però portato il valore di key (true) dentro a lock, quindi tutti gli altri processi continueranno a scambiare la loro key con lock, cioè true con true, rimanendo nel ciclo.
Nel momento in cui il processo ha finito, setta la lock a false, dando l'occasione a qualche altro processo di scambiarla con la key, e quindi di inibire il ciclo while. E così via.

I semafori

Tutte queste soluzioni possono funzionare, ma noi utilizzeremo una tecnica sviluppata da Dijkstra, chiamata a **semafori**. Un semaforo non è null'altro che una variabile **s**, intera. I semafori sono gestiti dal **sistema operativo**. Dato che vi sono diverse implementazioni, vediamo un caso generale e poi i singoli casi di utilizzo.

Abbiamo una variabile intera s compresa tra 0 e 1 (quindi o zero o uno) e due funzioni, **P(s)** (detto anche down o wait), e **V(s)** (detto anche up o signal). Vediamo i codici delle due funzioni.

P(s)

```
P(s){
  while(s==0) no op;
  s--;

}
```

V(s)

```
V(s){
  s++;
}
```

Questi due codici (semplici in maniera quasi disarmante!) vengono utilizzati nei processi in questo modo:

P1

```
P(s);

*sezione
critica*

V(s);
```

In sostanza, lo stato "0" è uno stato di fermo: il processo non può eseguire, mentre lo stato "1" è il via. Ad esempio, partendo da s=1, il primo processo lo porterà a 0. Fintanto che quel processo non avrà eseguito la sua V(s) nessun altro processo con il codice P(s) non potrà eseguire.

Abbiamo però ancora il problema dell'attesa attiva! Al posto del no op, viene quindi implementata una soluzione alternativa che mette delle system call nel codice, che spostano l'intero processo allo stato di waiting o di ready (in questo modo la CPU si libera e permette ad altri processi di eseguire). Il codice modificato è:

P(s)

```
P(s){
     while(s==0)
block();
  s--;
```

V(s)

```
V(s){
  s++;
  wake-up();
}
```

Non resta che vedere le diverse tipologie di semafori.

I semafori "mutex"

Questi semafori regolano una mutua esclusione, e cioè implementano la condizione "o esegue uno, o l'altro". In questa versione abbiamo una s che può scendere sotto a zero, ed il codice è il seguente.

P(s)

```
P(s){
  s--;
  while(s<0) block();

}
```

V(s)

```
V(s){
  s++;
  if(s<0) wake-up();
}
```

Come è abbastanza intuibile, il valore di s ora ha anche un senso semantico!

Infatti, |s| indica la quantità dei processi in coda.

I semafori "contatori"

Supponiamo ora di avere quattro differenti risorse, e di volerle gestire assegnandole a diversi processi. S può ora assumere valori > 1. Semplicemente, P(s) dovrà ora decrementare il valore di s (che sarà inizializzato al numero di risorse disponibili) mentre V(s) dovrà incrementarlo (per segnalare che la risorsa è nuovamente libera). La condizione s=0 sarà invece quella di attesa. Il codice è quindi identico a quello "base":

P(s)	V(s)

`P(s){`
`  while(s==0) block();`
`  s--;`
`}`
	`V(s){`
	`  s++;`
	`}`

Classici problemi di sincronizzazione

Ci sono dei sistemi classici per quanto concerne la sincronizzazione della memoria condivisa. Questi sono:
1) Produttori e consumatori
2) Lettori e scrittori
3) Cinque filosofi
4) Problema del barbiere

Vediamoli.

Produttori e consumatori

Abbiamo un buffer, dove i produttori devono depositare il loro prodotto, e dove i consumatori devono prelevare i prodotti già presenti nel buffer. Ovviamente le due azioni devono essere atomiche. Si usano pertanto 3 semafori: 1 semaforo mutex, 1 semaforo contatore **libere** ed un altro semaforo contatore **occupate**. Vediamo i due codici.

Produttore	Consumatore

`forever {`
`  *codice di produzione elemento*`
`  P(libere);`
`  P(mutex);`
`  *inserimento dato nel buffer*`
`  V(mutex);`
`  V(occupate);`
`}`
	`forever {`
	`  P(occupate);`
	`  P(mutex);`
	`  *prelevamento dato dal buffer*`
	`  V(mutex);`
	`  V(libere);`
	`}`

I codici sono molto comprensibili.
Il produttore prima produce l'elemento, quindi attende che ci siano spazi liberi, quindi attende di poterlo inserire (magari un consumatore o un altro produttore stavano scrivendo), inserisce il dato, libera la posizione d'utilizzo del buffer, incrementa la quantità di posizioni occupate.
Il consumatore prima verifica che ci sia almeno una posizione occupata, quindi vede se può prelevare, preleva, chiude la sessione di prelevamento, quindi incrementa le posizioni libere.

Lettori e scrittori

Il sistema è piuttosto simile a quello dei produttori e dei consumatori. In questo caso però vogliamo una mutua esclusione tra scrittori e lettori, ma vogliamo anche permettere a più lettori di "leggere insieme", poiché non andando a modificare i dati, non rischiano di "rovinare nulla". Abbiamo quindi un semaforo mutex **scrittura** ed un int **num_lettori** oltre ad un ulteriore mutex.
Vediamo i codici:

Scrittore

```
forever {
  P(scrittura)
  *scrittura dell'elemento*
  V(scrittura)
}
```

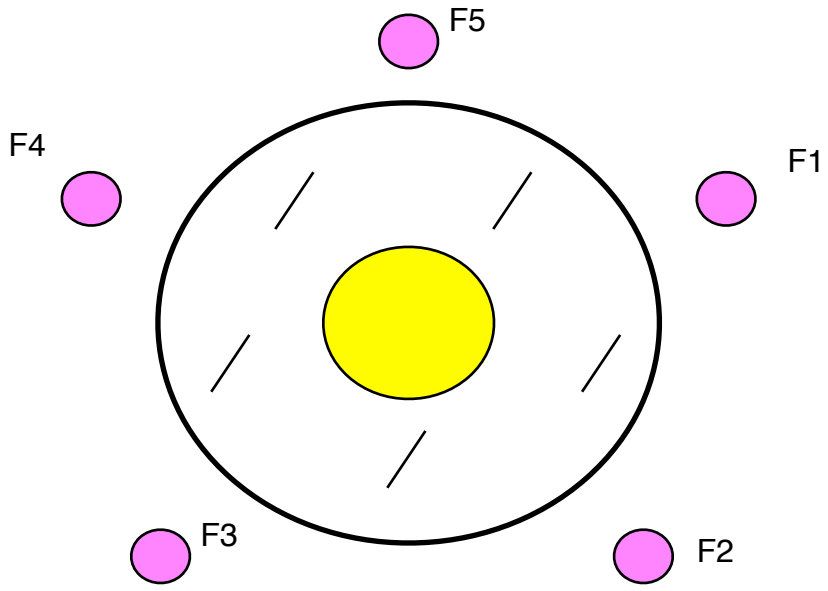
Lettore

```
forever {
  P(mutex);
  num_lettori++;
  if(num_lettori==1)
    P(scrittura)
  V(mutex)
  *lettura dell'elemento*
  P(mutex);
  num_lettori--;
  if(num_lettori==0)
    V(scrittura);
  V(mutex);
}
```

La difficoltà chiaramente sta tutta nella parte del lettore. I lettore prima di tutto si aggiunge alla lista di coloro che vogliono leggere, quindi verifica se è l'unico. Se lo è, chiede di poter usufruire della risorsa,in ogni caso attiva il mutex, legge e lo riabilita. Quindi si toglie dai lettori, verifica se è l'ultimo, se lo è ripassa la palla agli scrittori, altrimenti disabilita il mutex e conclude così il suo ciclo.

Il problema dei cinque filosofi

Passiamo ora ad una metafora, per comprendere il problema.



Abbiamo cinque filosofi messi ordinatamente intorno ad un tavoli (Fi). Tra ognuno di loro è posizionata una bacchetta, ed in centro al tavolo si trova un prelibato piatto di riso cinese.
Ogni filosofo esegue un codice le genere:

```
loop {
      *pensa*
      *mangia*
}
```

Come è risaputo, per poter mangiare il riso è necessario possedere due bacchette, pertanto, un filosofo, per poter mangiare, necessita di due delle bacchette che sono sul tavolo.
Questa situazione è decisamente sfavorevole, è infatti assai probabile trovarsi in situazioni di:
- **deadlock** (tutti i filosofi hanno in mano una bacchetta sola, e nessuno vuole cederla, pertanto rimangono bloccati così)
- Uno dei filosofi non riesce mai a mangiare (starvation)
- Ogni filosofo prende una bacchetta, poi si rendono conto della situazione di stallo, tutti posano la bacchetta, tutti riprendono la stessa, e così via (**livelock**, cioè un deadlock ma con attività)

Implementazioni risolutive
1) Dato un codice generale, come questo:

```
loop {
      *pensa*
      P(bacchetta[i]);
      P(bacchetta[(i+1)%5]); //prende la bacchetta alla destra
      *mangia*
      V(bacchetta[i]);
      V(bacchetta[(i+1)%5]);
}
```

Ci rendiamo conto che se tutti eseguissero la P(bacchetta[i]) in maniera consequenziale, nessuno potrebbe ottenere la seconda. Viene perciò introdotto un sistema secondo il quale un filosofo, non va sulla bacchetta destra, bensì sulla sinistra. In questo modo avremo una contesa fra due filosofi (che si risolverà in maniera casuale), ma intanto uno degli altri potrà mangiare, liberare la risorsa, e permettere quindi a tutti gli altri di cibarsi.
Questo sistema non impedisce però lo starvation! Vediamone un'altra.
2) Bacchette pulite, bacchette sporche
Ogni bacchetta può essere pulita o sporca. Quando F vuole mangiare, invia un messaggio ai suoi vicini per per ottenere la bacchetta. Il filosofo che riceve la richiesta, se ha la bacchetta sporca (cioè ha già mangiato, cioè ha già usufruito della risorsa per un certo tempo), sposta il suo stato a pulita e la passa al filosofo richiedente, mentre invece se la bacchetta è pulita rimane al filosofo che la sta usando.
Ogni volta che un filosofo mangia, chiaramente, sporca la forchetta.

Il problema del barbiere
Abbiamo un barbiere che ama servire i suoi clienti, ma se non ve ne sono, dorme. Il barbiere ha una sala d'attesa. Quando un cliente arriva dal barbiere controlla la sala d'attesa: se è tutta piena il cliente se ne va, se ci sono ancora posti il cliente si siede, se è vuota il cliente va a svegliare il barbiere, il quale si prepara e poi serve il cliente.
Tutto ciò è gestito tramite tre semafori.
* Clienti = 0;
* Barbiere = 0;
* Variabile personeInAttesa=0;
* mutex per le persone in attesa = 1;

Il codici sono i seguenti

Cliente

```
forever {
 P(mutex);
 if(in_attesa < N){
   in_attesa++;
   V(mutex);
   V(clienti); // messo in coda
   P(barbiere);
   *operazioni*
  }
  else
   V(mutex);
}
```

Barbiere

```
forever {
  while(1){
    P(clienti);
    P(mutex);
    in_attesa--;
    V(mutex);
    V(barbiere);
    *operazioni*
  }
}
```

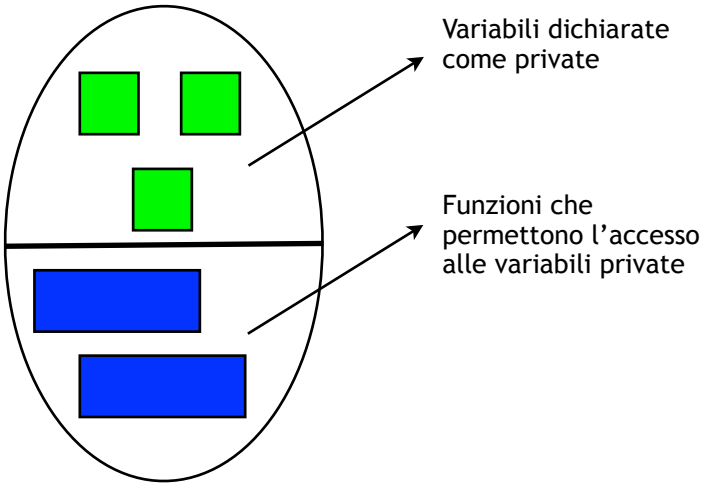
Il cliente in sostanza richiede di divenire una persona in attesa. Se è possibile aggiungerne, allora viene fatto +1, il cliente viene messo in coda, richiede il barbiere, compie le sue azioni, quindi se ne va.

Il barbiere fa un ciclo continuo (non si ferma mai, se non interrotto da qualche P), attendendo la presenza di un cliente, quindi chiedendo l'utilizzo della variabile in_attesa (per eliminare un cliente in attesa), in seguito restituendo la possibilità di utilizzare la stessa variabile, e infine servendo il cliente (`V(barbiere)` sblocca il `P(barbiere)` a cui è fermo il processo cliente).

Il monitor

Il **monitor** è un costrutto implementato in certi linguaggi di programmazione che permette la gestione in mutua esclusione di alcune variabili. Questo sistema permette, dato un accesso casuale di più processi alla risorsa, un accesso ordinato (e in mutua esclusione, come già detto).

Un monitor è costituito generalmente in questa maniera:



Il tipo di queste variabili è "**condition**", le quali ci permettono una migliore sincronizzazione tra i processi.

Una generica variabile condition viene dichiarata normalmente come:
`condition x;`
e possiede due funzioni per essere gestita
`wait(x);`
che sospende il processo che la esegue, e
`signal(x);`
che causa il risveglio di un processo sospeso, in caso lo sia, altrimenti è totalmente ininfluente.

Vediamo una applicazione del monitor, premettendo però che dovremo apportare delle modifiche successive affinché funzioni. Abbiamo una dichiarazione iniziale, seguita dalla dichiarazione di due funzioni.

P1

```
boolean inUso=false;
condition disponibile;

void prendiRisorsa(){
  inUso=false;
  if(inUso) wait(disponibile);
  inUso=true;
}

void rilasciaRisorsa(){
  inUso=false;
  signal(disponibile);
}
```

Chiaramente le due funzioni prendiRisorsa() e rilasciaRisorsa() devono essere eseguite in maniera atomica.

Durante l'esecuzione della funzione `signal(disponibile)` viene risvegliato un processo dormiente, quindi ci si pone la domanda: se la riga `signal(disponibile)` la funzione proseguisse, sia la funzione risvegliata che quella "risvegliatrice" vorrebbero poter continuare la loro esecuzione.
Ecco quindi un'altra competizione. Vi sono due politiche differenti, dai nomi piuttosto palesi.
1) Segnala e attendi --> continua il processo svegliato
2) Segnala e prosegui --> continua il processo svegliante.

Applicazione: i cinque filosofi con utilizzo del monitor
Abbiamo la definizione di diversi metodi:

Inizializzazione comune ai filosofi

```
monitor5Fil{
  enum {pensa, affamato, mangia} stato[5];
  condition aspetta[5];
}
```

Funzione prende

```
prende(int i){
  stato[i]=affamato;
  verifica(i);
  if(stato[i]!=mangia){
    wait(aspetta[i]);
  }
}
```

Funziona posa

```
posa(int i){
  stato[i]=pensa;
  verifica((i+1)%5);
  verifica((i+4)%5);
}
```

Funzione verifica

```
verifica(int i){
  if((stato[(i+a)%5]!=mangia) &&
(stato[(i+a)%5]!=mangia) &&
(stato[i]==affamato)){
    stato[i]=mangia;
    signal(aspetta[i]);
  }
}
```

Il funzionamento è abbastanza chiaro già dal codice.

Riassunto da: Enrico Mensa

Sistemi Operativi
Il deadlock

Capitolo 4

Indice degli argomenti

Il deadlock

Ne abbiamo già parlato profusamente, il deadlock è una situazione di stallo che si presenta nella condivisione di risorse (e non solo).

Classi ed istanze di risorse
Dicesi classe di risorsa l'insieme di risorse di uno stesso tipo. La singola risorsa contenuta in una classe di risorse è detta istanza di risorsa.

Condizioni necessarie per il deadlock (*servono tutte e quattro*)
1) Mutua esclusione (caratteristico della risorsa)
2) Possesso e attesa (facoltà di appropriarsi di una risorsa e chiederne un'altra)
3) Non pre-emptive (nessuno può togliere la risorsa ad un processo)
4) Attesa circolare (P_1 aspetta P_2 che aspetta P_1 ecc...)

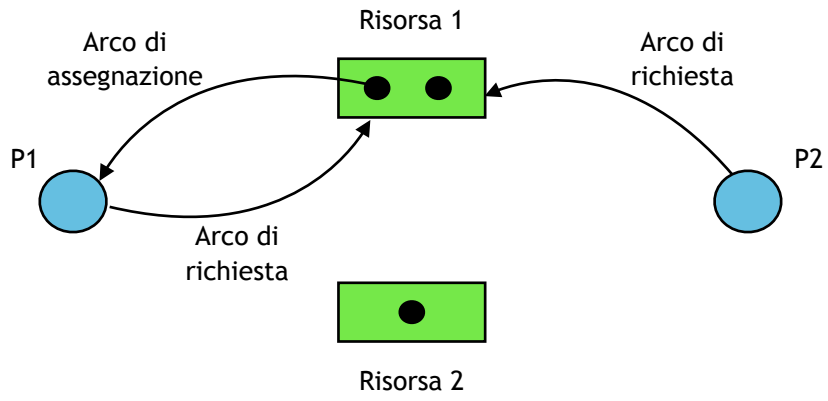
Grafo di allocazione
Per comprendere al meglio le situazioni di deadlock si utilizza un metodo standard per disegnare il susseguirsi dei processi: il grafo di allocazione.

G = <V, E> dove V sono i vertici, ed E sono gli archi.

Vertici
I vertici sono rappresentati da rettangoli in caso di classi di risorse (con un quantitativo di puntini all'interno equivalente al numero delle sue istanze. I processi, invece, sono rappresentati da pallini.

Archi
Gli archi possono essere archi di richiesta (freccia dal processo alla risorsa) ed archi di assegnazione (freccia dall'istanza al processo).

Esempio



Un esempio: lo spooler di stampa
Un esempio di deadlock un minimo articolato è lo spooler di stampa. Si ha un processo che deve inserire in un buffer dei dati relativi ad un file, i quali devono essere interpretati e gestiti dallo spooler il quale comunica con la periferica in questione, per esempio, la stampante. Chiaramente la stampante è più lenta del processo, quindi si avrà un accumulo dei dati. Nessun problema se il buffer è sufficientemente grande per contenere tutti i dati in arrivo, ma se il buffer si dovesse riempire, nessun processo potrebbe partire in stampa, quindi il buffer non si libererebbe mai, bloccando l'intero sistema.

Una soluzione può essere la prelazione, oppure lo streaming, tramite il quale mentre il processo scrive i dati lo spooler inizia a prelevarli e poi li elabora tutti insieme: un sistema che permette di liberare il buffer evitando l'intasamento.

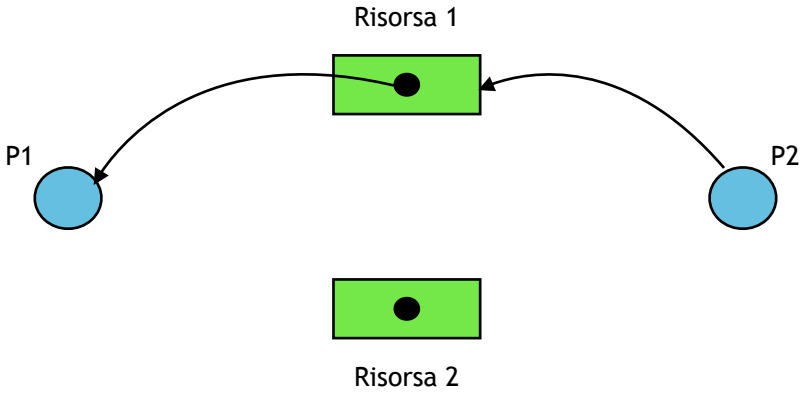
Risoluzioni generali per il problema del deadlock

Date le quattro condizioni necessarie affinché il deadlock si verifichi, è intuitivo pensare che basti bloccarne una per impedire al deadlock di presentarsi.
È su questa base che Havender fonda le sue tre teorie risolutive (è impossibile togliere la condizione della mutua esclusione, in quanto propria della risorsa).

Possesso e attesa
I processi richiedono tutte le risorse di cui avranno bisogno appena inizia l'esecuzione. Se tali risorse sono tutte libere, allora il processo può iniziare, altrimenti se anche solo una delle risorse di cui necessita non dovesse essere presente, il processo rilascia tutte le sue risorse e attende di averle tutte. Questo metodo, seppur funzionante, genera starvation (oltre a rallentare l'esecuzione).

Preemptive
Diamo la possibilità alle risorse di essere prese dai processi che le stanno utilizzando. Quando un processo richiede una risorsa che è al momento utilizzata da altri, rilascia tutte le sue risorse. Questo metodo, seppur funzionante, fa perdere tutto il lavoro compiuto dal processo prima della sua richiesta "critica".

Ordinamento delle classi di risorse
Diamo un ordinamento preciso alle classi di risorse, ad esempio $R_1 < R_2 < R_3$
Una volta stabilito un ordine, obbligo i processi a richiedere le risorse in quel dato ordine.



Se P_1 volesse R_1, e P_2 volesse R_2, si potrebbe creare un deadlock. Ma invece, obbligando P_2 ad attendere per R_1, la situazione si risolve.

L'importanza di rilevare il deadlock

Questi metodi funzionano, ma hanno molti punti deboli e rallentamenti. Si è così pensato di "intercettare" i deadlock, ed in caso risolverli. Nascono così i termini di **safe state** e **unsafe state**.
Il sistema operativo, ancora una volta, dovrà intervenire per "monitorare" ciò che stanno facendo i processi, i quali dal canto loro dovranno fornire più informazioni sulla loro esecuzione: precisamente, uno stato delle risorse (cioè quante risorse stanno usando e quante ne vogliono ancora usare). Un safe state è

una situazione che è in grado di mutare (proseguire) in una sequenza sicura, mentre un unsafe state garantisce una sicura evoluzione in un deadlock.
Capiamo meglio questi termini!

Una **sequenza sicura** è un susseguirsi di assegnamenti di risorse che porta l'esecuzione da un safe state all'altro, e che garantisce a tutti i processi di terminare senza deadlock.

Esempio
Abbiamo questa distribuzione delle risorse, in un certo attimo. Il sistema operativo può compiere molte scelte!

	Allocate	Richieste
P1	3	2
P2	4	1
P3	1	4
libere	2	

	Allocate	Richieste
P1	5	0
P2	4	1
P3	1	4
libere	0	

	Allocate	Richieste
P1	-	-
P2	4	1
P3	1	4
libere	5	

	Allocate	Richieste
P1	-	-
P2	5	0
P3	1	4
libere	4	

	Allocate	Richieste
P1	-	-
P2	-	-
P3	1	4
libere	5	

eccetera..

Attenzione però! Questa scelta avrebbe creato un deadlock subito!

	Allocate	Richieste
P1	3	2
P2	4	1
P3	1	4
libere	2	

	Allocate	Richieste
P1	3	2
P2	4	1
P3	4	1
libere	0	

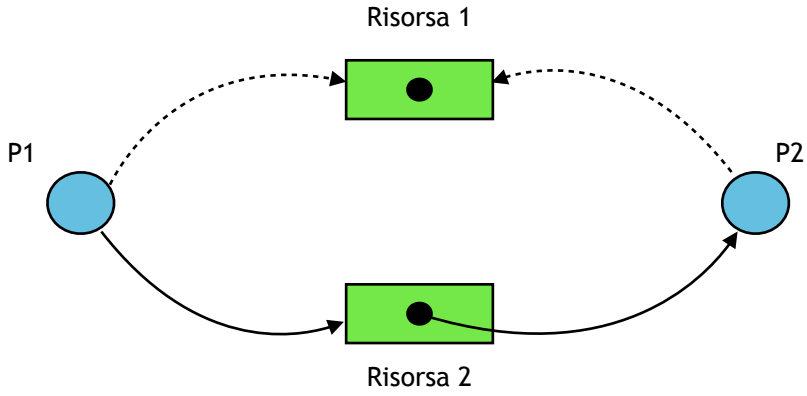
Secondo quale metodo il sistema operativo sceglie quale strada fare?
Semplicemente si ricerca il possibile ordinamento dei processi tale che le richieste di P_i possano essere soddisfatte utilizzando le risorse libere più le risorse usate dai processi tali che $j < i$ (i precedenti).

Come è possibile rappresentare tutto questo?

Metodi di prevenzione del deadlock (deadlock avoidance)

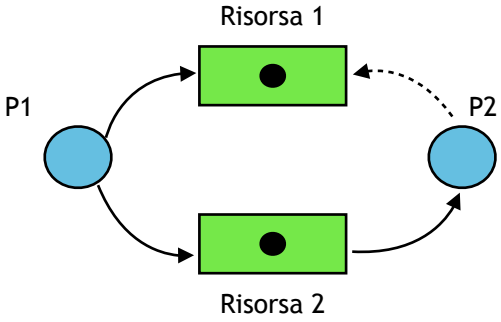
Vi sono due metodi utilizzati per prevenire i deadlock (prima che accadano! Sono previsioni!)
1) Grafi di allocazione con archi di reclamo (claim edge)
2) Algoritmo del banchiere

Grafi di allocazione con archi di reclamo
Abbiamo già visto cosa sono i grafi allocazione. Aggiungiamo ora un arco, detto di reclamo, indicato con la linea tratteggiata. Il suo significato è che, durante l'esecuzione, prima o poi, il processo necessiterà della risorsa puntata dalla freccia.

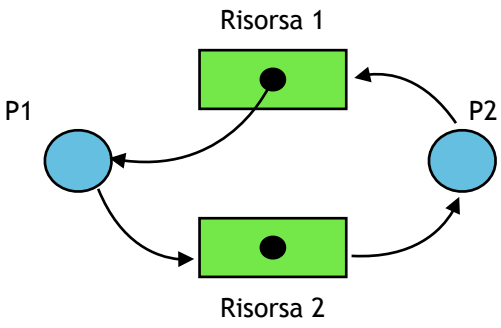


Uno stato safe.

Supponiamo che ora P_1 effettui la sua richiesta, saremmo in questa situazione:



Ora, se P_1 dovesse ottenere la risorsa e P_2 effettuasse la sua richiesta, avremmo una situazione di deadlock!



Pertanto verrà data la priorità a P_2 evitando così il deadlock.

Algoritmo del banchiere
L'algoritmo del banchiere consta a sua volta di due sotto-algoritmi. Li esamineremo a parte.

A) Algoritmo che controlla se un certo stato è safe (proiezione nel futuro)

Utilizzeremo le seguenti strutture dati:
* disponibile[M] con M=classi di risorse da gestire. disponibile[i] indica quante istanze sono libere nella classe di risorsa iesima.
* assegnate[N][M] con N=numero di processi ed M come sopra. assegnate[i][k] indica quante risorse della classe k sono assegnate al processo iesimo.
* necessarie[N][M] con N e M come sopra. necessarie[i][k] indica di quante risorse di classe k il processo iesimo avrà bisogno nella sua esecuzione (dichiarazione all'inizio!)

Convenzione! *dire che x > y significa che per ogni i, a[i]>y[i]*

1) Siano lavoro e fine due array di lunghezza rispettivamente M ed N
2) lavoro = disponibili
3) fine[i]=false per ogni i compresa tra [0..N]
4) Cerca un i tra [0..N] tale che fine[i]=false && necessarie[i] <= lavoro //*cerca un processo iesimo le cui richieste siano soddisfacibli, cioè che non chieda più risorse di quelle disponibili (in lavoro)*
5) Se i è stato trovato,
    5.1) lavoro = lavoro + assegnate[i] //*restituisce le risorse che aveva*
    5.2) fine[i] = true
    5.3) goto(4)

Se alla fine dell'algoritmo il punto 4 non trova più nessuna i, ogni processo è stato soddisfatto quindi lo stato è sicuro.

Esempio (con una sola classe di risorsa)

	Ass.	Ric.
P1	5	5
P2	2	2
P3	2	7
disp	3	

Abbiamo pertanto:

disponibili = 3
necessarie = {5, 2, 7}
assegnate = {5, 2, 2}
lavoro = 3
fine = {F, F, F}

Eseguiamo l'algoritmo passo a passo!
4) i = 1, 5<=3 --> NO!
4) i = 2, 2<=3 --> SI!
5) lavoro = lavoro + assegnate[2] = 3 + 2 (ora lavoro vale 5) ; falso = {F, T, F}
4) i = 1, 5<=5 --> SI!
5) lavoro = lavoro + assegnate[1] = 5 + 2 (ora lavoro vale 7); falso = {T, T, F}
4) i = 3, 5<=7 --> SI!
5) lavoro = lavoro + assegnate[3] = 7 + 2 (ora lavoro vale 9); falso = {T, T, T}

Era uno stato sicuro. *Sequenza sicura:* ***P2, P1, P3***.

B) Algoritmo che decide come assegnare le risorse (basandosi sull'algoritmo A)

Utilizzeremo le seguenti strutture dati:
* disponibile[M] con M=classi di risorse da gestire. disponibile[i] indica quante istanze sono libere nella classe di risorsa iesima.
* assegnate[N][M] con N=numero di processi ed M come sopra. assegnate[i][k] indica quante risorse della classe k sono assegnate al processo iesimo.
* richieste[N][M] con N e M come sopra. richieste[i][k] indica di quante risorse di classe k il processo iesimo avrà bisogno immediatamente (non dichiarate dall'inizio!)
* necessarie[N][M] con N e M come sopra. necessarie[i][k] indica di quante risorse di classe k il processo iesimo avrà bisogno nella sua esecuzione (dichiarazione all'inizio!)

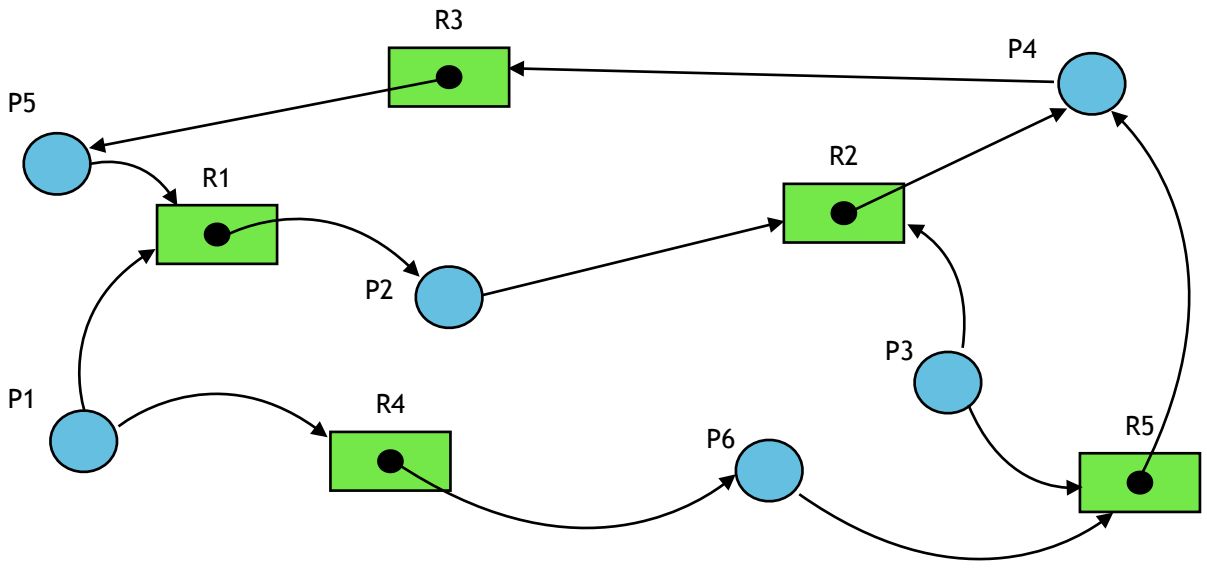
Convenzione! *dire che x > y significa che per ogni i, a[i]>y[i]*

1) Controlliamo la coerenza: se richieste > necessarie[i] ERRORE! Pi sospeso. (ha richiesto più risorse di quanto avesse dichiarato all'inizio)
2) Se richieste > disponibili, Pi sta in attesa
3) Altrimenti, simula l'esecuzione della richiesta
    3.1) disponibili[i] = disponibili[i] - richieste[i]
    3.2) assegnate[i] = assegnate[i] + richieste[i]
    3.3) necessarie[i] = necessarie[i] - richieste[i]
4) Verifica se lo stato raggiunto è sicuro (tramite l'algoritmo A!)
    4.1) Se è sicuro, esegui l'assegnazione
    4.2) Se non è sicuro :
        - ripristina lo stato precedente
        - sospendi Pi

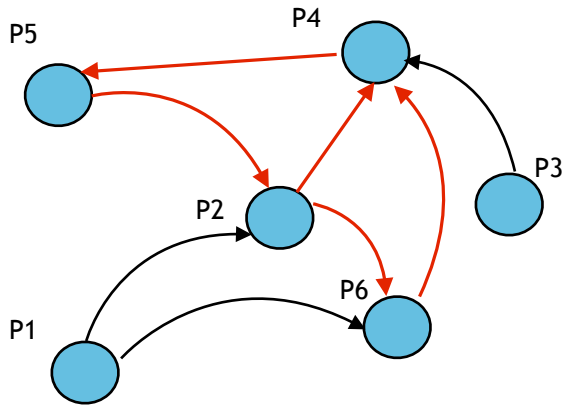
Metodi di rilevamento del deadlock (deadlock detection)

Abbiamo anche due metodi differenti per rilevare un deadlock (una volta che è accaduto!)
1) Grafi di allocazioni delle risorse (una classe di risorse)
2) Algoritmo per più classi di risorse

Grafi di allocazione delle risorse
Prendiamo un grafo qualsiasi come il seguente



Evidentemente è piuttosto complesso capire la situazione. Si adotta pertanto una semplificazione, cioè gli archi ora avranno il significato di "Pi aspetta una certa risorsa che però ha Pj" (con la freccia da Pi a Pj).
Il grafo di prima risulta ora:



È ora molto più evidente il deadlock!

Algoritmo per più classi di risorse
Chiaramente il metodo precedente non può funzionare con più istanze, sarebbe troppo complesso vedere "a occhio" il deadlock.
L'algoritmo è molto simile a quello del banchiere, ma ora si sta parlando di intervenire a deadlock avvenuto. È un controllo a posteriori!

Utilizzeremo le seguenti strutture dati:
* disponibile[M] con M=classi di risorse da gestire. disponibile[i] indica quante istanze sono libere nella classe di risorsa iesima.
* assegnate[N][M] con N=numero di processi ed M come sopra. assegnate[i][k] indica quante risorse della classe k sono assegnate al processo iesimo.
* richieste[N][M] con N e M come sopra. richieste[i][k] indica di quante risorse di classe k il processo iesimo avrà bisogno immediatamente (non dichiarate dall'inizio!)

Convenzione! *dire che x > y significa che per ogni i, a[i]>y[i]*

1) lavoro = disponibili
2) Per ogni i
    if(assegnate[i] = {0,0,0...}) //non ha risorse assegnate
        fine[i] = vero; *//se non ha risorse assegnate, non può dare problemi*
    else
        fine[i]=falso; *//potrebbe darne*
3) while (esiste un i tale che fine[i]=falso && richieste[i]<=lavoro) //finché c'è un processo candidabile
    3.1) lavoro = lavoro + assegnate[i] //quelle che rilascia
    3.2) fine[i] = vero
4) se tutte le richieste sono state varate, allora fine[i]=vero per ogni i, altrimenti abbiamo un deadlock.

Dato l'elevato costo computazionale, quando conviene effettuare questi controlli?
* Quando un processo fa una richiesta e ottiene come risposta "occupato"
* Quando l'uso della CPU diminuisce ma i processi rimangono in ugual numero

I provvedimenti possono essere: killare il processo, usare prelazione sulle risorse.

Riassunto da: Enrico Mensa

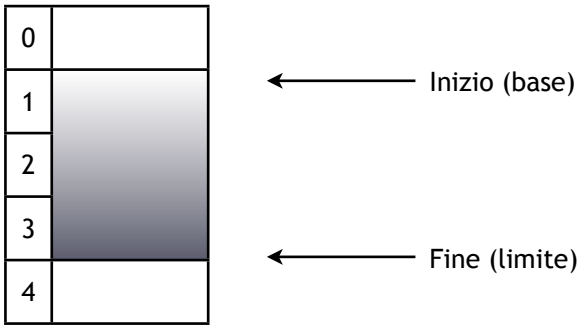
Sistemi Operativi
La memorizzazione dei processi

Capitolo 5

Indice degli argomenti

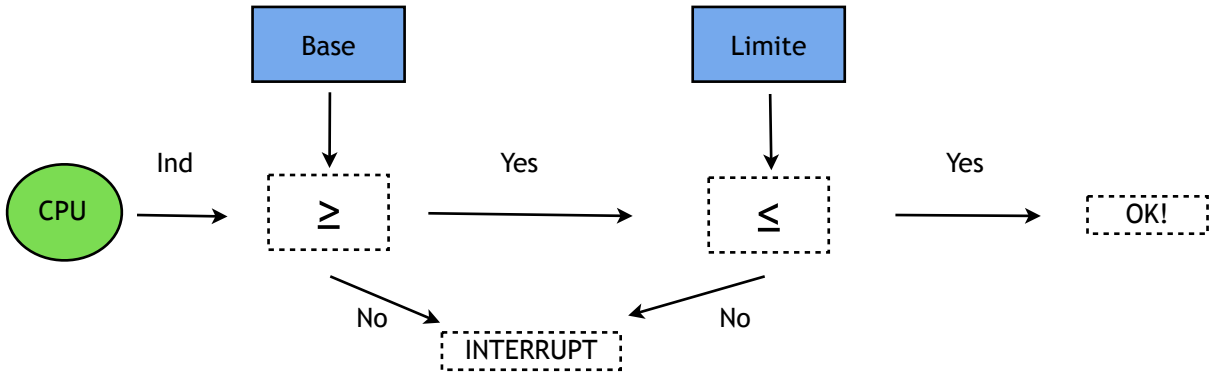
Dove vengono memorizzati i processi?

Come sappiamo, i processi sono compositi di due porzioni: indirizzi e dati. Tali sezioni sono entrambe salvate nella memoria RAM. Come è intuibile, vi sono diverse maniere per poter immagazzinare i processi in memoria. Le prime sono hardware, le seconde software (le vedremo meglio in seguito).
Sostanzialmente l'obbiettivo primario è quello di verificare se un certo indirizzo appartenga o meno ad una certa regione di memoria.

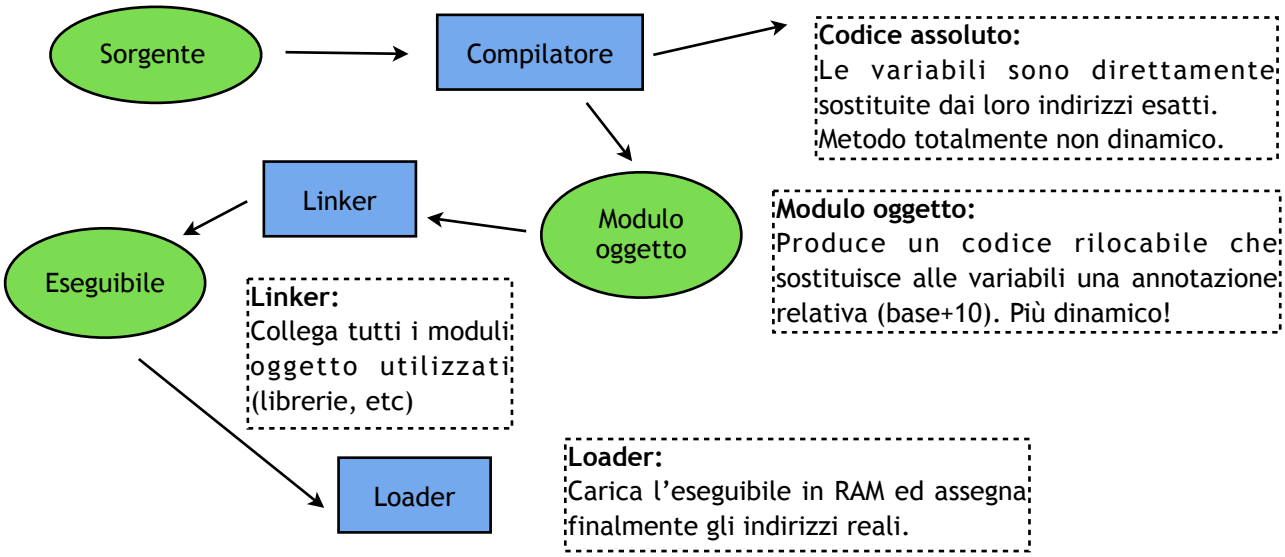


Le regioni di memoria sono facilmente definibili tramite due registri che contengono due indirizzi (che delimitano la regione), base e limite.

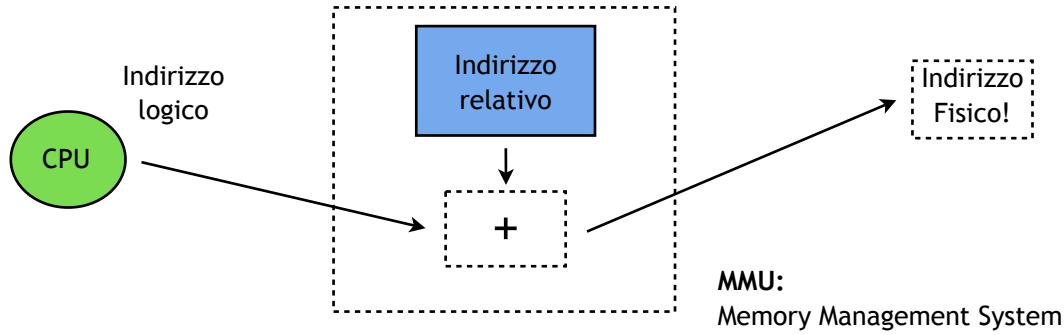
Tramite un semplicissimo algoritmo è possibile capire se un certo indirizzo faccia parte o meno di una certa regione. La CPU richiede un indirizzo dove andare a scrivere...



Il processo di compilazione
Per poter comprendere al meglio come vengono immagazzinati i processi in memoria, è necessario capire anche come vengono trattati i file e quindi come vengono salvati. La compilazione si compone delle seguenti fasi:



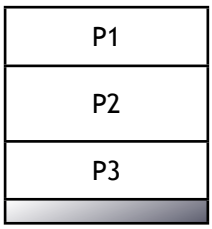
Questo sistema detto con **binding statico** (infatti il collegamento effettivo viene fatto solamente nella fase di creazione dell'eseguibile), seppur ottimo, ha un problema... se prendiamo ad esempio il fenomeno dello swapping vedremo che i vari processi, scambiandosi, dovranno avere SEMPRE le stesse dimensioni per poter effettuare lo swap (gli indirizzi sono infatti fissi).
Si utilizza pertanto un altro metodo, detto **binding dinamico** che sostanzialmente sposta l'assegnazione degli indirizzi nella esecuzione stessa del processo. La CPU richiederà ora un **indirizzo logico**, non più un indirizzo fisico. La traduzione tra i due avverrà in questa maniera:



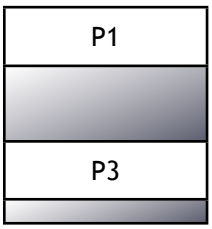
Prendiamo ad esempio un editor di testo. Nella fase di linking, verranno collegati ad esso tutte le varie librerie necessarie del caso. Il tutto verrà poi messo in RAM.
Apriamo ora una nuova pagina del suddetto editor: andare a caricare nuovamente le librerie sarebbe quantomeno ridicolo!
Bisogna quindi effettuare degli opportuni controlli. Questa implementazione prende il nome di **linking e loading dinamici**. Come è possibile crearla?
Semplicemente nell'eseguibile vi sarà il cosiddetto stub, che ricerca in RAM una data istruzione necessaria: se è presente non fa nulla, se invece non c'è il codice dello stub si auto-sostituisce con quello della funzione desiderata.

Modulo di allocazione contigua

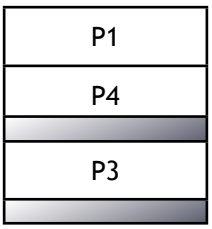
Uno dei metodi per immagazzinare i processi è il modulo di allocazione contigua.
Ogniqualvolta un processo deve essere messo in RAM, gli viene assegnata una area di memoria contigua.



Ad un certo punto, il processo P2 potrebbe essere disallocato, lasciando dello spazio libero. Ora i processi non sono più contigui. Questo stato viene detto frammentazione esterna.



Ora P4 vorrebbe essere allocato, il quale occupa occupa poco meno della memoria di P3



È evidente che lo spazio lasciato sotto P4 sia inutilizzabile per metterci altri processi, e al contempo sia inutile per qualsiasi scopo. Questo fenomeno viene detto frammentazione interna. Ora, supponendo di avere un P5 che starebbe tra lo spazio avanzato di P4 e quello sotto P3, non possiamo comunque eseguirlo poiché lo spazio non è contiguo.
Se lo spazio avanzato dalla frammentazione è così piccolo che certamente un processo non potrà starvi, quella porzione viene assegnata al processo che si impossessa alla sua adiacente.

Politiche di allocazione per la modulazione contigua
Esistono vari metodi per risolvere (trattare, in realtà) la questione.

1) Best-fit (gli spazi avanzati sono salvati in una lista, essa viene scorsa e viene scelto lo spazio con dimensione più vicina a quella di cui il processo ha bisogno)
2) First - fit (viene preso il primo elemento della lista sufficientemente grande)
3) Worst - fit (viene preso il primo elemento della lista - che è ordinata dal più grande al più piccolo spazio -)

Dal punto di vista computazionale, i primi due metodi sono molto simili, mentre l'ultimo è facilmente implementabile anche se palesemente più inefficiente.

La traduzione tra indirizzi logici e fisici viene effettuata tramite l'ausilio dell'**MMU** (come visto sopra).

Il codice transiente
Sotto tutta l'area dedicata al sistema operativo, c'è una zona detta "codice transiente", che potrebbe essere allocata dal SO in caso di necessità. In questa implementazione, l'area deve essere necessariamente contigua.

La paginazione

0	P1
1	P2
2	P2
3	P1
4	P3

Costituzione della RAM

0	Pagina1
1	Pagina2
..	..

Costituzione di un processo

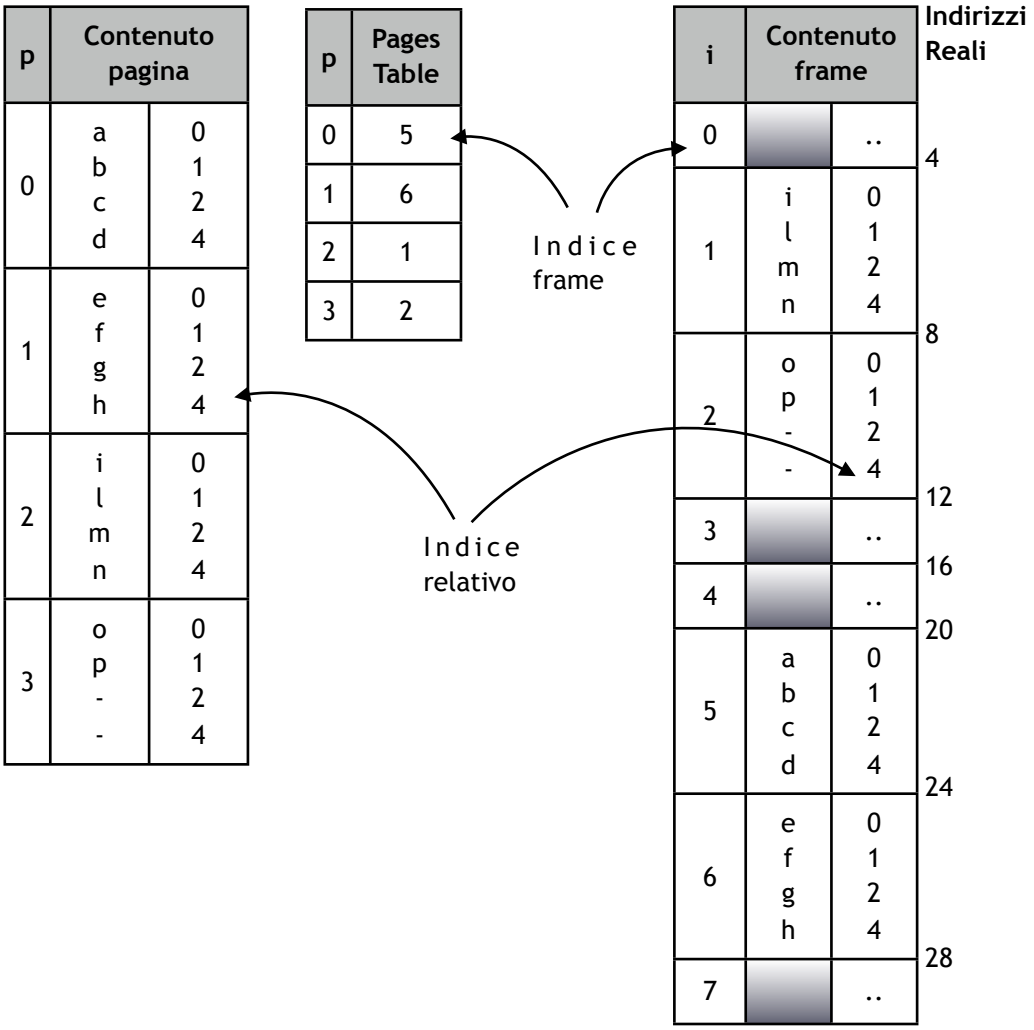
Un altro metodo per immagazzinare i processi in RAM, è la paginazione. Dividiamo la RAM in sezioni di uguali dimensioni dette **frames**. Dividiamo quindi i nostri processi in parti, dette **pagine**. La dimensione di un frame corrisponde esattamente a quella di una pagina. La dimensione di una pagina, che, per qualche motivo, potrebbe essere spostata in memoria secondaria, prende il nome di **blocco**.

Chiaramente questo metodo risolve il problema della frammentazione! L'implementazione è più complessa poiché i processi sono "dispersi" nella RAM, ma viene offerta molto più dinamicità: inoltre lo spreco è di circa mezza pagina per ogni processo.

Gli indirizzi logici sono contigui, mentre quelli fisici no. Come può il sistema tradurre una richiesta (dato un indirizzo logico) della CPU in un indirizzo fisico?
Dobbiamo innanzitutto introdurre la nozione di **Tabella delle pagine** (da ora PT), che istituisce una precisa correlazione tra ogni pagina e l'indice del frame corrispondente in memoria.
La CPU comunicherà una coppia logica <p,o> dove p è il numero di pagina e dove o è l'offset della variabile desiderata all'interno della pagina. Tramite p, controllando nella TB, otterremo l'indice del frame che contiene la pagina cui siamo interessati. Tramite l'intuibile calcolo BaseFrame = f*dimframe (dove dimFrame è la dimensione di un singolo frame) troveremo l'indirizzo di base del frame, dal quale, sommando o, otterremo l'indirizzo della variabile desiderata.
La questione potrebbe sembrare piuttosto complessa, ma è in realtà relativamente banale. Si sta cercando di creare una correlazione sempre vera tra pagine e RAM.

Esempio di paginazione

Facciamo un esempio. Abbiamo un processo P_x che ha la seguente suddivisione in pagine, la seguente tabella delle pagine. La situazione della RAM è riportata di fianco. La dimensione di un frame è di 4.



Diciamo di voler ottenere la variabile "l".

La CPU invierà la coppia <2,1> (pagina 2, l è in offset 1 all'interno della pagina). Quindi tramite la PT, si ottiene BaseFrame=4*1 = 4 (dimensioneFrame * indiceFrame). Infatti l si trova nella pagina che inizia dall'indirizzo di RAM 4. Ora, facendo 4+1 (la o), otteniamo 5, che è l'indirizzo fisico della l.

Il codice transiente

In questa implementazione il codice transiente non è affatto un problema, semplicemente anch'esso sarà assegnato ad un frame libero e verrà trattato alla stessa maniera.

La tabella delle pagine (esclusiva per ogni processo, chiaramente), è però inizializzata con delle dimensioni "superiori", per permettere al processo di, eventualmente, ampliarsi nel tempo. Pertanto avremo alcune correlazioni pagina-frame valide, e altre no (semplicemente è un'area riservata a quello scopo, ma non ancora attiva). Si utilizzerà quindi un bit di controllo per capire quali sono "attive" e quali no. Tale bit viene chiamato **bit di validità**.

La condivisione delle pagine
Questa implementazione si presta molto bene alla condivisione delle pagine. Cioè, potremmo avere processi che eseguono lo stesso codice (quindi hanno pagine corrispondenti), come ad esempio i figli dopo una fork. In tal caso, utilizzare diversi frame per tenere copie dello stesso codice è assolutamente inaccettabile. Si effettuano quindi dei controlli prima di allocare la pagina nel frame.
Possiamo avere due tipi di condivisione: **lettura** (l'esempio appena effettuato), oppure **lettura/scrittura** (memoria condivisa).

La rilocabilità
Anche qui la paginazione soddisfa pienamente le nostre aspettative! In caso di swapping, sarà sufficiente aggiornare gli indici dei frames nella PT.

Lo spazio?
È bene sapere che il sistema di paginazione viene progettato lasciando grandissimi spazi "di manovra" per le evoluzioni future. Ad esempio, in un sistema con parole a 32 bit, potremmo avere pagine con indirizzi a 32 bit (quindi 2^{32} pagine memorizzabili). Supponendo che ogni pagina sia grande 4Kb, possiamo subito calcolare che abbiamo la possibilità di accedere a **64TB** di memoria RAM! Una quantità quantomeno interessante!

La scelta della dimensione delle pagine
Qual'è la dimensione perfetta per una pagina? Abbiamo pro e contro.
Una pagina piccola implica una minima quantità di frammentazione interna (pag/2), ma un alto numero di computazioni per spostare le pagine. Viceversa per le pagine grandi.

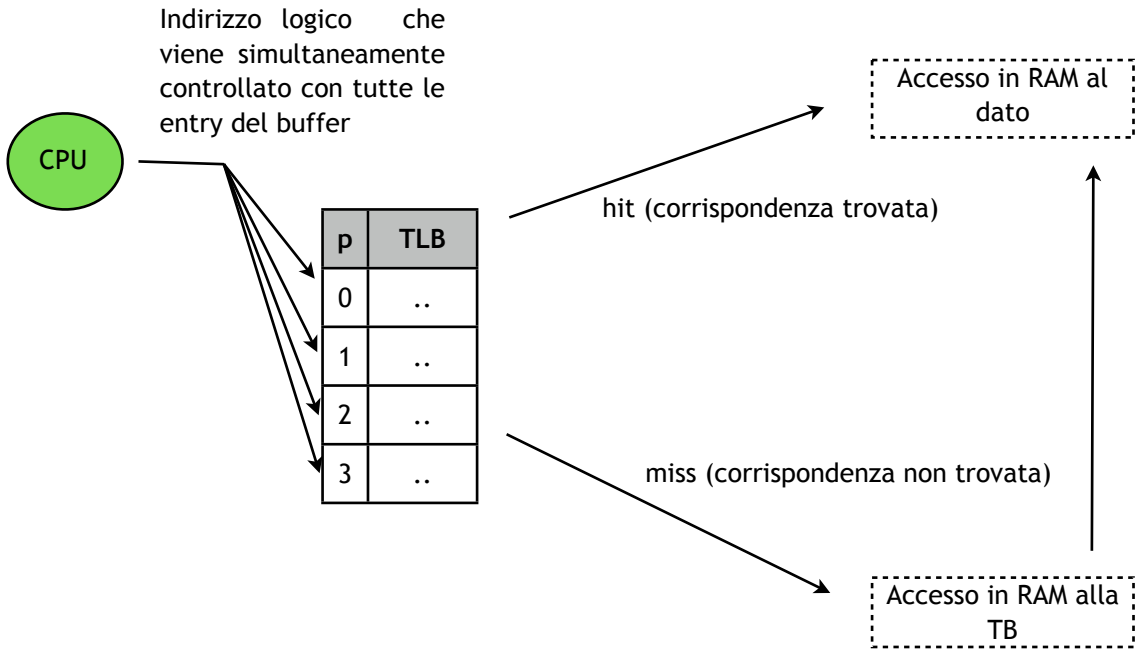
Come vengono allocate le pagine nei frame?
Proviamo a rappresentare con uno pseudo-codice ciò che deve eseguire il sistema operativo per allocare una pagina in un frame.

```
if(#frameLiberi >= numPagP){
  perogni pagina di P
    - crea una entry nella page table
    - entry = frame libero
    - aggiorna page table
}
```

Dove si trova la PT?
La Process Table si trova, chiaramente nel PCB. Ma, effettivamente, dove si trova? Può essere messa sostanzialmente in due posti diversi.
A) Registri appositi della CPU (supporto dell'hardware) - il dispatcher si curerà di sostituire anche questo durante il context switch. La soluzione è molto costosa, conviene solo quando ci sono meno di 256 pagine da salvare nella tabella (poche)
B) La PT viene sistemata nella RAM in maniera contigua. Questo significherebbe però effettuare ben due accessi in RAM ogniqualvolta si voglia raggiungere una variabile... viene introdotta quindi una memoria cache (molto veloce), che contiene una TB "modificata", che prende ora il nome di **Translation Look Aside Buffer - TLB**.

Implementando la soluzione b si ha sostanzialmente un buffer delle pagine, che vengono accumulate fino a che c'è posto (vedremo dopo cosa capita in caso di buffer pieno).

La richiesta della CPU verrà quindi così trattata:



È chiaro: nel caso di miss vengono effettuati due accessi in ram, nel caso di hit invece uno soltanto. È altrettanto palese che la decisione di quali pagine debbano rimanere dentro alla TLB sia discriminante per l'efficenza di questa soluzione.

Proviamo a fare un semplice esempio.
T. Accesso alla RAM = 100nsec
T. Accesso alla TLB = 20nsec
T.M.A. (dipende dalla percentuale hit/miss) = prob_hit*(TLB + T.A. RAM) + prob_miss(T.A. RAM + T.A. RAM)
Diciamo di avere una hit ratio = 80%
Si avrà TMA = 0.8 * (120) + 0.2 * (200) = 136nsec. *Ottimo!*

Dato un miss, si inserirà la entry del TB appena usata nel TLB. Nel caso di TLB pieno, verrà eliminata la entry più vecchia a favore della nuova (detta **Last Recently Used**).

Ulteriori problematiche del TBL
Come detto sopra, il TBL è un buffer condiviso, quindi potremmo trovare anche altre pagine "avanzate" dai processi precedenti. Si userà pertanto una colonna in più chiamata Address Space Identifier (**ASID**) che identifica "quali pagine sono di chi".
Inoltre, nello swapping, rischiamo di "riprendere" vecchie pagine che intanto sono state modificate... insomma, vi sono svariate problematiche subdole.

Il problema della dimensione delle pagine
Un'altro importante aspetto da considerarsi è la dimensione delle pagine, che può essere molto grande (quindi di difficile gestione). Per questo problema, vi sono diverse soluzioni. Vediamole.

1) *Paginazione della tabella delle pagine* - Dividere in pagine la tabella delle pagine stessa. Seppur possa sembrare una idea strampalata, sarebbe sufficiente introdurre una **Tabella delle pagine esterna** in grado di memorizzare le informazioni relative alle pagine che costituiscono la tabella delle pagine.

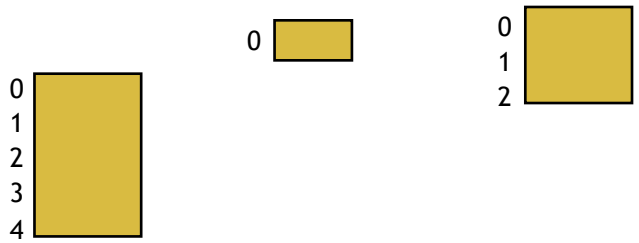
A questo punto la CPU non invierà soltanto una coppia <p,o> bensì una tripletta <p_1, p_2, o> dove p_1 rappresenta la pagina della tabella esterna delle pagine, p_2 indica la pagina nella tabella delle pagine, infine o indica il solito offset relativo alla pagina/frame.

2) *Hash Table* - la CPU invia una coppia <p,o>. La p viene fatta passare attraverso una function(p) detta funzione di hash che restituisce un indice di una tabella, detta **hash table**. All'interno della tabella, non c'è un indice di un frame bensì una lista di coppie <p,f>. Viene cercata la coincidenza tra la p richiesta dalla CPU e la p contenuta nella lista di coppie che è stata selezionata grazie alla funzione di hash. La ricerca è sequenziale (non molto performante).

3) *Tabella delle pagine invertite* - La RAM possiede, di per se, una tabella che indica quali frame sono occupati e quali non lo sono. Si è quindi pensato di arricchire questa tabella, collegando ad ogni indice di frame una coppia <pid, p> che indica quale processo ha quale pagina assegnata nel determinato #frame. La CPU questa volta invierà una tripletta <pid, p, o>... la ricerca nella tabella delle pagine invertite sarà, appunto, inversa, cioè cercheremo il <pid, p> nella parte destra e otterremo l'indice #frame come risultato. Avremo quindi la solita coppia <#frame, o>.

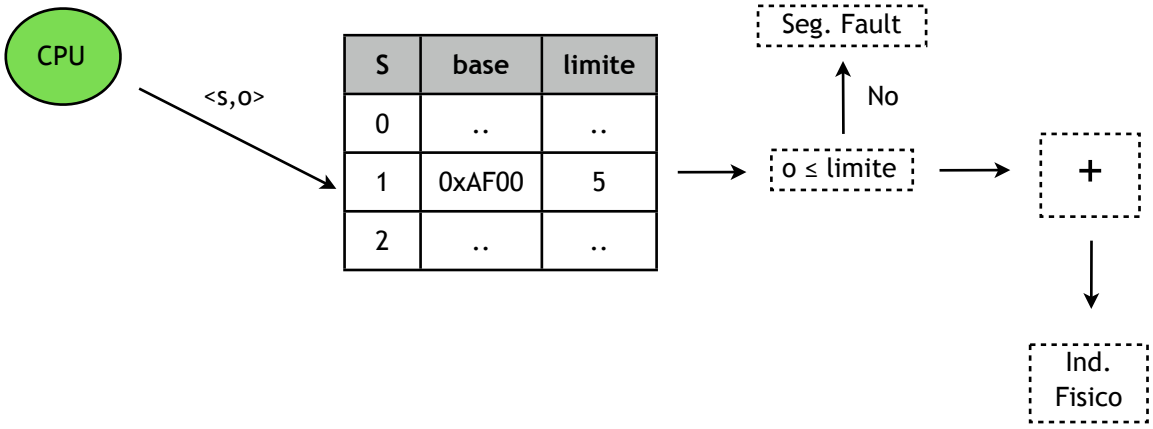
La segmentazione

Questa tecnica in sostanza fonde parte della filosofia della paginazione con il sistema ad allocazione contigua. I segmenti sono aree di memoria della RAM divise per contenuto, di dimensione differente l'una dall'altra, quindi con una quantità diversa di indirizzi relativi (dipendenti appunto dalla lunghezza del segmento. Ad esempio potremo avere:



Il compilatore dovrà, a priori, assegnare delle zone differenti di memoria per:
1) Codice
2) Variabili Globali
3) Stack
4) Heap
5) Librerie standard

La traduzione indirizzo logico -> indirizzo fisico avverrà in questo modo:
La CPU invia una coppia <s,o>, dove s definisce un indice nella **tabella dei segmenti**. Ad s corrisponderà una coppia <base,limite> dove base è l'indirizzo della prima cella di quel segmento, e limite è il numero di celle che occupa.

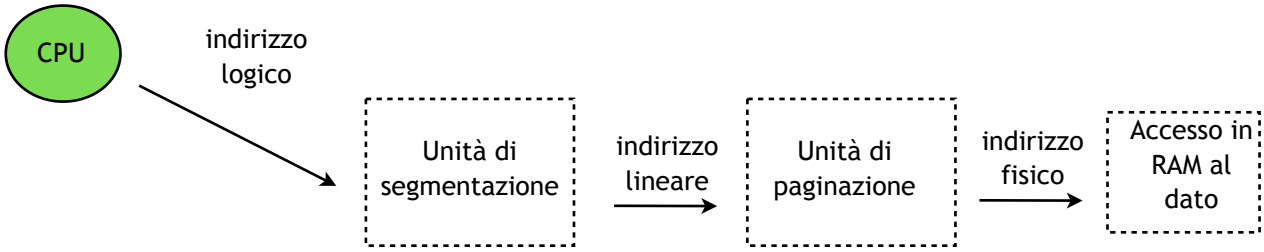


Raffronto tra paginazione e segmentazione

Funzionalità	Paginazione	Segmentazione
Spazio degli indirizzi lineare	Uno	Molti
Procedure dati possono essere gestite separatamente	No	Sì
La condivisione delle procedure è facilitata?	No	Sì

La segmentazione paginata
Prenderemo per esempio un processore Intel Pentium.

La richiesta da parte della CPU di un indirizzo fisico seguirà questo modello:



L'implementazione di questo sistema sicuramente semplifica la modifica dei segmenti, che sono ora null'altro che pagine. L'indirizzo logico sarà costituito da una tripletta <s, g/l, o> dove s è la corrispondenza da cercare nella tabella dei segmenti (ci darà quindi la base), g/l è il tipo di variabile (globale o locale) - rappresenta il limite, mentre l'offset è sempre il solito.
Una volta ottenuti base, limite ed offset, essi vengono codificati nel cosiddetto indirizzo lineare, una serie di bit divisa in tre parti che verranno interpretate dall'unità di paginazione come <p1, p2, o> (in questo caso si avrà quindi una paginazione esterna (p1) che porta alla PT.

Riassunto da: Enrico Mensa

Sistemi Operativi
La memoria virtuale

Capitolo 6

Indice degli argomenti

La memoria virtuale

Sarebbe comodo per noi lasciare che i processi credano di avere tutto lo spazio che vogliono, e poi gestire tramite il sistema operativo questa illusione. Questa cosa esiste, e si chiama memoria virtuale. I processi hanno alcune pagine in memoria secondaria, altre in RAM. Questo sistema, chiaramente, porta con se diverse nuove parti da implementare. Vediamole.

Il caricamento delle pagine

Parliamo innanzi tutto di come capire il funzionamento (ed il costo) dell'operazione di caricamento di una pagina. Si ricorda che una pagina invalida è una pagina che non è caricata in RAM.

Una pagina deve essere caricata se si verifica una o più di queste situazioni:
1) L'istruzione attuale del PC si trova in una pagina invalida
2) L'istruzione attuale richiede qualche operatore che si trova in una pagina invalida
3) L'istruzione attuale vuole depositare il suo risultato in una pagina invalida

In questi casi avviene il cosiddetto **page fault**.
In caso di page fault, interviene il **pager** il quale gestisce l'evento in questa maniera:
* Individua un frame libero (*si veda dopo l'algoritmo*)
* Richiede una copiatura da memoria secondaria (desidera il blocco corrispondente alla pagina)
* Aggiorna la tabella delle pagine
* Riavvia l'istruzione interrotta

I tre casi differenti in cui il page fault può accadere infieriscono più o meno pesantemente sull'esecuzione. Conoscendo il ciclo fetch->decode->execute, se il pagefault avviene poiché si verifica il primo evento, allora semplicemente verrà fatta la fetch. Se invece accade il secondo, dovremo rifare da capo la fetch e la decode (infatti la pagina era già stata messa in RAM quindi fetchata e decodificata). Caso ancora peggiore è il terzo, dove la pagina era anche stata eseguita (dobbiamo rifare tutto da capo).

È oramai chiaro che il page fault è un problema grave che porta via molto tempo. Viene così introdotto un nuovo metodo, il **paging on demand**.

Paging on demand

I processi alla loro partenza non caricano nessuna pagina. Poi mano a mano caricano le pagine che servono loro, e così i page fault all'inizio molto presenti diminuiscono drasticamente.
Per capire la cosa, necessitiamo di calcolare il *tempo medio di accesso effettivo alla pagina* che ha formula Tmae = p*TGPF + (1-p) * tma, dove TGPF sta per "tempo gestione page fault", e p è la probabilità che un page fault accada.
Orientativamente un page fault viene gestito in millisecondi, mentre un accesso alla ram (tma) è dell'ordine di nanosecondi. È chiaro che l'unica maniera di ottimizzare il sistema sia quella di diminuire p in maniera drastica.

Vediamo i vari passi con cui viene trattato un page fault (questa volta in maniera più dettagliata). Verrà effettuata una swap con la memoria secondaria per ottenere la pagina desiderata (il problema di quale venga scelta per essere tolta verrà esaminato successivamente):
***** Interrupt (page fault)
***** Salvataggio dello stato della PCB e dei registri della CPU
***** Controllo della correttezza del riferimento

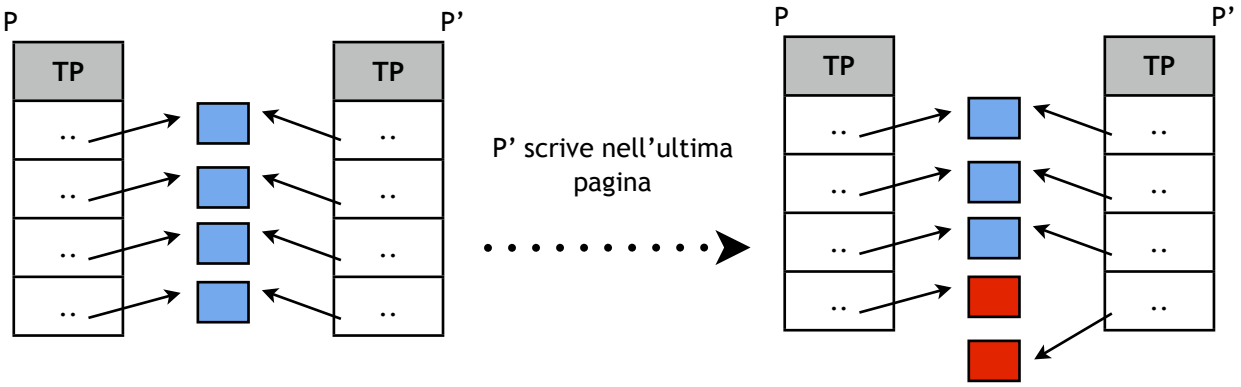
***** Lettura e copiatura delle pagine tra memoria secondaria e RAM.
***** Context switch (tanto stiamo aspettando la memoria secondaria...)
***** Interrupt (la nuova pagina è pronta!)
***** Lo stato del processo che necessitava della pagina viene messo a ready
***** La tabella delle pagine viene aggiornata

L'area di swap
Nella memoria secondaria viene definita un'area di swap. Quest'area conterrà le pagine inutilizzate (inattive). Tale area può essere organizzata sottoforma di file (cresce/decresce a piacere, ma rallenta l'accesso poiché deve passare attraverso la struttura del filesystem) piuttosto che come partizione del disco (più veloce come accesso, ma di dimensione fissa).

La scelta di chi escludere
Come si è detto, per effettuare una swap è necessario (ovviamente) decidere quale pagina deve lasciare la RAM. La scelta è importantissima! Uno dei sistemi è l'utilizzo di un **dirty bit**, cioè un bit che ci dice se la pagina ha avuto una scrittura oppure no. Il criterio di scelta terrà in considerazione il dirty bit: con una pagina "sporca", cioè scritta, dovremmo prima riscrivere la pagina in memoria secondaria e quindi caricare la nuova sopra la sporca... invece con una pagina "pulita", possiamo tranquillamente sovrascrivere (tanto abbiamo ancora la copia in memoria secondaria intatta).

Copiatura su scrittura
Nell'istante in cui viene fatta una fork, si potrebbe pensare che le pagine del processo padre vengano copiate, generandone altre due (di proprietà del processo figlio).
Questo non è quel che accade.

In realtà P e P' tengono in condivisione le loro pagine fino a che una delle due non ne modifica il contenuto. A quel punto, la pagina viene sdoppiata, generando una situazione nella quale alcune pagine sono condivise mentre altre sono proprie del padre/figlio.



Queste situazioni si verificano sia nel caso di scrittura sulla pagina, sia nella esecuzione di execl (che danno a P un nuovo stack ed un nuovo heap (che saranno contenuti in nuove pagine, quindi).

Implementazione della paginazione on demand

La paginazione on demand richiede sostanzialmente due algoritmi:
1 - Algoritmo di allocazione dei frames
2 - Algoritmo di sostituzione delle pagine (quale pagina deve lasciare la RAM per fare spazio alla nuova?)

Algoritmi di sostituzione delle pagine

Vedremo:
1) FIFO
2) Ottimale
3) LRU (least recently used)
4) Varianti di LRU (second chance e bit supplementare di riferimento)
5) Algoritmi basati sul conteggio (last frequently used, most frequently used)

FIFO (First In-First Out)
L'algoritmo FIFO (first-in, first-out) è molto semplice. Il processo P ha dei frames (limitati) assegnati, inoltre ogni pagina ha un contatore assegnato (detto time marker). La scelta della pagina da killare dipenderà ovviamente da questo marker: chi possiede il marker più vecchio verrà sostituito.

Esempio
Come convenzione (usata in questi riassunti) ogni richiesta di pagina verrà codificata con una lettera, e nella tabella delle richieste verrà indicato con "X" il page fault, mentre "-" ne indica l'assenza.
La tabella dei stato della RAM avrà in alto il codice della richiesta che è stata eseguita in quell'istante. La riga PF ci dice se c'è stato (X) oppure no (-) un page fault per adempiere a quella richiesta.
Non tutti i passaggi sono esplicitati (tipicamente i primi - la RAM inizia vuota - no). La tabella di stato della RAM indica inoltre l'indice del frame ed il marker temporale.

FIFO

Cod. Richiesta	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*
Pag. Richiesta	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**4**
PF	X	X	X	X	-	X	X	X

Stato iniziale

i	Pag	Mark
0	-	-
1	-	-
2	-	-

→

Richiesta A/B/C

i	Pag	Mark
0	7	T0
1	0	T1
2	1	T2

→

Richiesta D

i	Pag	Mark
0	2	T3
1	0	T1
2	1	T2

→

Richiesta E

i	Pag	Mark
0	2	T3
1	0	T1
2	1	T2

Richiesta F

i	Pag	Mark
0	2	T3
1	3	T4
2	1	T2

→

Richiesta G

i	Pag	Mark
0	2	T3
1	3	T4
2	0	T5

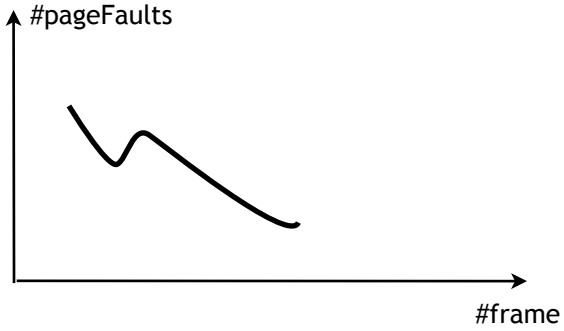
→

Richiesta H

i	Pag	Mark
0	4	T6
1	3	T4
2	0	T5

Chiaramente questo metodo ha dei forti punti deboli: c'è stato page fault nella richiesta G (pagina 0 desiderata), ma avevamo appena tolto la pagina 0 nella richiesta F!

L'algoritmo FIFO porta inoltre una anomalia, detta **anomalia di Belady**. Tracciando un grafico #frame/ #pageFault vedremo che si ha un punto in cui aumentare il numero di frame è inutile. La cosa non è affatto positiva, poiché intuitivamente con più frame dovremmo avere sempre meno page faults.



Algoritmo ottimale

L'algoritmo ideale sarebbe quello che:

***** Minimizza i page fault

***** Non porta l'anomalia di Belady

Purtroppo questa soluzione non è così facilmente implementabile, anzi, sembra essere inarrivabile. Pertanto, tutti gli algoritmi mostrati in seguito usano questo algoritmo ottimale (teoricamente fattibile, praticamente no) come "ideale", e ne sono quindi un'approssimazione.

LRU - (Least Recently Used)

Questo algoritmo è simile al FIFO, però aggiorna i marker stando all'ultimo accesso alla pagina. Rispetto a prima, ogni volta in cui si effettuerà l'accesso ad una pagina (r/w), il suo marker verrà aggiornato all'istante di tempo attuale. Vediamo anche qui un esempio.

Sequenza di richiesta delle pagine: 7,0,1,2,0,3,4,2,3,0,3,2,1

LRU

Cod. Richiesta	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*L*	*M*	*N*	*O*	*P*
Pag. Richiesta	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**
PF	X	X	X	X	-	X	-	X	X	X	X	-	-	X

Stato iniziale

i	Pag	Mark
0	-	-
1	-	-
2	-	-

→

Richiesta A/B/C

i	Pag	Mark
0	7	T0
1	0	T1
2	1	T2

→

Richiesta D

i	Pag	Mark
0	2	T3
1	0	T1
2	1	T2

→

Richiesta E

i	Pag	Mark
0	2	T3
1	0	T4
2	1	T2

Richiesta F

i	Pag	Mark
0	2	T3
1	0	T4
2	3	T5

→

Richiesta G

i	Pag	Mark
0	2	T3
1	0	T6
2	3	T5

→

Richiesta H

i	Pag	Mark
0	4	T7
1	0	T6
2	3	T5

→

Richiesta I

i	Pag	Mark
0	4	T7
1	0	T6
2	2	T8

Richiesta L		
i	Pag	Mark
0	4	T7
1	3	T9
2	2	T8

Richiesta M		
i	Pag	Mark
0	0	T10
1	3	T9
2	2	T8

Richiesta N		
i	Pag	Mark
0	0	T10
1	3	T11
2	2	T8

Richiesta O		
i	Pag	Mark
0	0	T10
1	3	T11
2	2	T12

Richiesta P		
i	Pag	Mark
0	1	T13
1	3	T11
2	2	T12

I page fault sono leggermente diminuiti. Questo algoritmo è un buon punto di partenza (anche se lo scorrimento sequenziale dei marker e l'alto numero di PF non sono affatto buona cosa!), infatti a seconda della sua implementazione possiamo ottenere algoritmi più o meno perfomanti. Per evitare lo scorrimento sequenziale potremmo pensare di ordinare le varie pagine in uno stack, ordinato per uso delle pagine stesse. Effettueremmo una pop dal basso ed una push dall'alto. Vediamo comunque le altre possibili implementazioni.

LRU con bit supplementare di riferimento (Least Recently Used - 1° implementazione)

Una delle possibili implementazioni è quella di introdurre un bit detto bit di riferimento. Quando la pagina viene utilizzata, il suo bit di viene messo a 1. Ovviamente, quando siamo alla ricerca di una pagina da "killare", sceglieremo tra quelle a zero. Il bit di riferimento diventa utile quando lo adoperiamo in concomitanza con i **registri a scorrimento**. Tali registri sono sostanzialmente una cronologia delle variazioni dei bit di riferimento. Ogni volta che una richiesta viene effettuata, nei registri a scorrimento (uno per ogni pagina) viene salvato lo stato attuale del bit di riferimento. Osservando in un dato istante i registri sapremo quando la pagina è stata utilizzata. Quando il registro è pieno, viene semplicemente shiftato il contenuto a destra (perdendo i record più vecchi. Vediamo subito un esempio con un registro a scorrimento a 4 celle.

Nota: n.P è il numero di pagina, BR è il bit di riferimento, RS è il registro di scorrimento a 4 celle.

LRU (Var I)

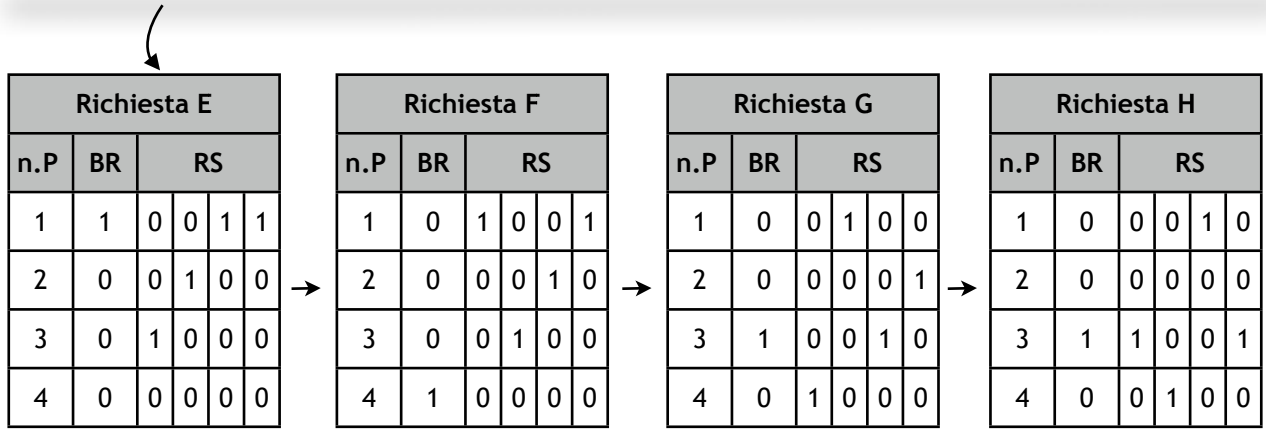
Cod. Richiesta	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*
Pag. Richiesta	**1**	**1**	**2**	**3**	**1**	**4**	**3**	**3**

Richiesta A						
n.P	BR	RS				
1	1	-	-	-	-	
2	0	-	-	-	-	
3	0	-	-	-	-	
4	0	-	-	-	-	

Richiesta B						
n.P	BR	RS				
1	1	-	-	-	1	
2	0	-	-	-	0	
3	0	-	-	-	0	
4	0	-	-	-	0	

Richiesta C						
n.P	BR	RS				
1	0	-	-	1	1	
2	1	-	-	0	0	
3	0	-	-	0	0	
4	0	-		0	0	

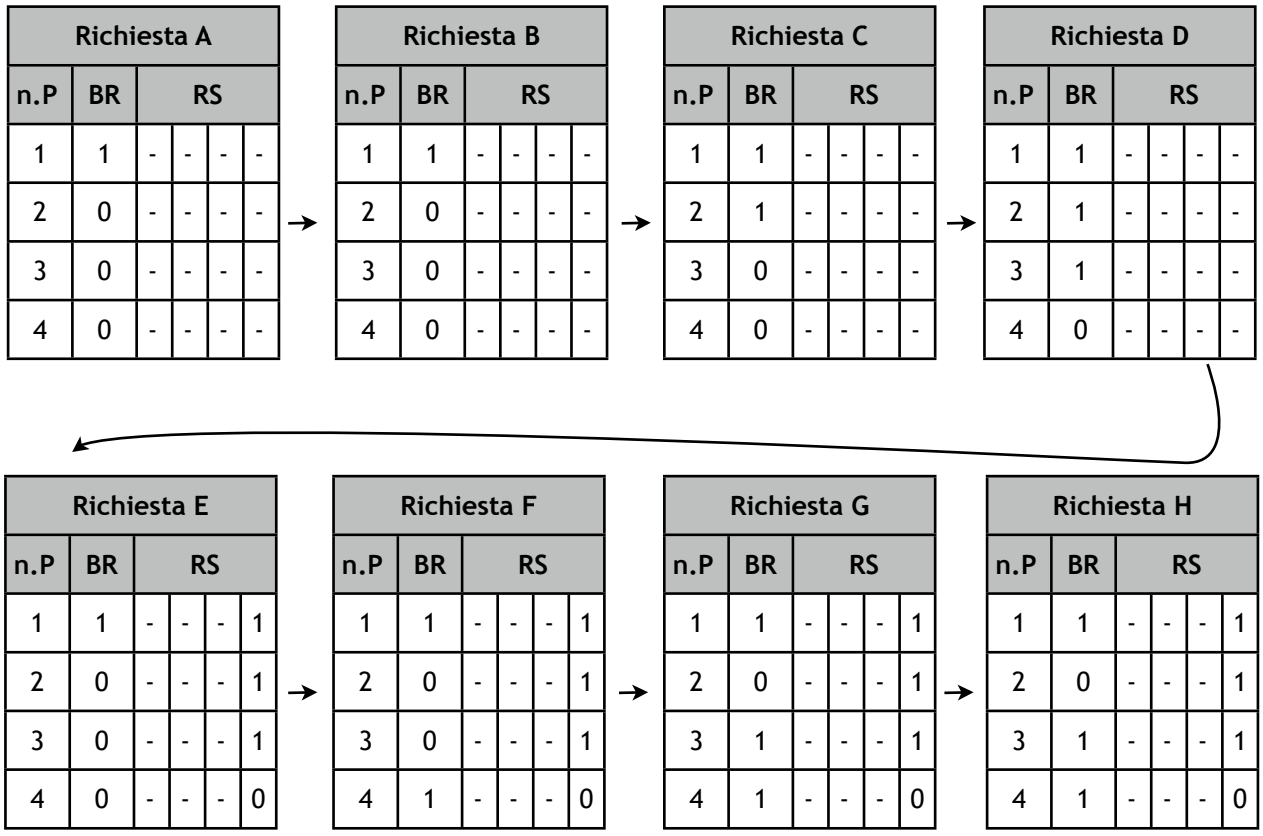
Richiesta D						
n.P	BR	RS				
1	0	-	0	1	1	
2	0	-	1	0	0	
3	1	-	0	0	0	
4	0	-	0	0	0	



Richiesta E							Richiesta F							Richiesta G							Richiesta H						
n.P	BR	RS					n.P	BR	RS					n.P	BR	RS					n.P	BR	RS				
1	1	0	0	1	1		1	0	1	0	0	1		1	0	0	1	0	0		1	0	0	0	1	0	
2	0	0	1	0	0		2	0	0	0	1	0		2	0	0	0	0	1		2	0	0	0	0	0	
3	0	1	0	0	0		3	0	0	1	0	0		3	1	0	0	1	0		3	1	1	0	0	1	
4	0	0	0	0	0		4	1	0	0	0	0		4	0	1	0	0	0		4	0	0	1	0	0	

Supponiamo che avvenga una richiesta della pagina 5, dopo la richiesta H. A questo punto, leggendo i registri, vedremo che la pagina 1 ha un valore di utilizzo pari a due, la pagina due ha un valore pari a 0, la pagina 3 ha invece un valore 10 e infine la pagina 4 ha un valore 4. La pagina che verrà spostata in memoria secondaria sarà quindi la pagina due.

C'è una nota dolente però! Aggiornare ad ogni caricamento di pagina un registro è una operazione che può portare enormi rallentamenti. Si può pertanto "aumentare la grana" della precisione, ad esempio aggiornando i registri ogni quattro richieste invece che ogni singola richiesta. In tal caso avremo questa situazione:



Richiesta A							Richiesta B							Richiesta C							Richiesta D						
n.P	BR	RS					n.P	BR	RS					n.P	BR	RS					n.P	BR	RS				
1	1	-	-	-	-		1	1	-	-	-	-		1	1	-	-	-	-		1	1	-	-	-	-	
2	0	-	-	-	-		2	0	-	-	-	-		2	1	-	-	-	-		2	1	-	-	-	-	
3	0	-	-	-	-		3	0	-	-	-	-		3	0	-	-	-	-		3	1	-	-	-	-	
4	0	-	-	-	-		4	0	-	-	-	-		4	0	-	-	-	-		4	0	-	-	-	-	

Richiesta E							Richiesta F							Richiesta G							Richiesta H						
n.P	BR	RS					n.P	BR	RS					n.P	BR	RS					n.P	BR	RS				
1	1	-	-	-	1		1	1	-	-	-	1		1	1	-	-	-	1		1	1	-	-	-	1	
2	0	-	-	-	1		2	0	-	-	-	1		2	0	-	-	-	1		2	0	-	-	-	1	
3	0	-	-	-	1		3	0	-	-	-	1		3	1	-	-	-	1		3	1	-	-	-	1	
4	0	-	-	-	0		4	1	-	-	-	0		4	1	-	-	-	0		4	1	-	-	-	0	

La prossima richiesta, porterebbe il secondo record nei registri, fornendo così i valori alle pagine (rispettivamente): 3, 1, 3, 3, 2. La pagina da killare sarebbe quindi la seconda. In ogni caso, la precisione è minore ma comunque buona.

LRU di seconda chance (Least Recently Used - 2° implementazione)
Una seconda implementazione del sistema LRU è la seguente: si ha il bit di riferimento ed un puntatore. Data una pagina puntata nell'istante in cui si cerca una vittima, se il suo bit è a zero, essa viene killata ed il puntatore incrementato. Se il bit è invece ad 1, allora il bit viene portato a zero ed il puntatore incrementato (ecco il concetto di seconda chance).
Quando una pagina entra, ha bit di riferimento ad 1 (ovviamente, dato che è appena stata utilizzata. Ricordiamo che il bit di riferimento si modifica quando la pagina viene utilizzata). Quando il puntatore arriva all'ultima pagina, ricomincia da capo. Vediamo un esempio.
Nota: P è la pagina contenuta nel frame, BR è il bit di riferimento, Pu è il puntatore.

LRU (Var II)

Cod. Richiesta	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*L*	*M*	*N*	*O*	*P*
Pag. Richiesta	**1**	**2**	**3**	**4**	**1**	**1**	**2**	**5**	**2**	**1**	**5**	**5**	**6**	**7**
PF	X	X	X	X	-	-	-	X	-	X	-	-	X	X

Stato Iniziale		
BR	Pu	P
0	->	-
0		-
0		-
0		-

→

Richiesta A/B/C/D		
BR	Pu	P
1	->	1
1		2
1		3
1		4

→

Richiesta E/F/G		
BR	Pu	P
1	->	1
1		2
1		3
1		4

→

Richiesta H		
BR	Pu	P
1		5
0	->	2
0		3
0		4

ha fatto tutto il giro, annullando tutti gli uno, quindi ha sostituito la pag 1 con la pag 5 ed ha incrementato il puntatore

Richiesta I		
BR	Pu	P
1		5
1	->	2
0		3
0		4

→

Richiesta L		
BR	Pu	P
1		5
0		2
1		1
0	->	4

→

Richiesta M/N		
BR	Pu	P
1		5
0		2
1		1
0	->	4

→

Richiesta O		
BR	Pu	P
1	->	5
0		2
1		1
1		6

→

Richiesta P		
BR	Pu	P
0		5
1		7
1	->	1
1		6

Una ulteriore variante di questo algoritmo è l'implementazione di coppie di bit, invece di un bit di riferimento soltanto. Abbiamo una coppia <bit di riferimento, bit di modifica> dove bit di modifica vale è se sono state effettuate solo read, 1 se è stata effettuata almeno una write. Questo crea un livello di "priorità" nella scelta.

1) <0,0> - Obbiettivo perfetto
2) <0,1> - Abbiamo solamente scritto, ma è una pagina poco usata. Ricopiamola in memoria secondaria.
3) <1,0> - È una pagina che usiamo spesso.
4) <1,1> - La usiamo spesso e scriviamo anche! Pessimo obbiettivo!

Algoritmi basati sul conteggio
Vi sono due algoritmi diversi:
1) least frequently used
2) most frequently used

La prima tecnica punta, tramite un contatore di utilizzi, a selezionare quali pagine vadano eliminate. Una pagina poco utilizzata, tendenzialmente, verrà utilizzata poco anche nel futuro.
Ma questo genera un problema nelle pagine appena introdotte, che hanno ovviamente un conteggio di utilizzo molto basso.
La seconda tecnica, invece, si pone il problema contrario: una pagina che è stata molto utilizzata fin dall'inizio ma poi non viene più adoperata rischia di tenere un contatore molto alto nonostante questo non rispecchi il reale utilizzo attuale.

Ulteriori migliorie
Vi sono altri sistemi che vengono combinati insieme a quelli visti in precedenza, sempre con l'intento di diminuire i page fault.
Pool of free frames
Per ogni processo vengono associati alcuni frame liberi in sovrappiù.
Nel momento in cui avviene un page fault:
- Viene applicato l'algoritmo per selezionare la vittima
- La vittima non viene cancellata ma viene aggiunta al pool of free frames
- La nuova pagina viene caricata in uno dei frame considerati "liberi".

In sostanza le vittime stanno nel pool of free frames per un po' di tempo dopo il loro "killaggio", in modo che se dovesse essere stata effettuata una scelta erronea, sia comunque possibile reperire la pagina in RAM.

Algoritmi di allocazione dei frames

Vi sono due principali categorie:
1) Algoritmi di allocazione dei frames per i processi utente
2) Algoritmi di allocazione dei frames per processi kernel (del sistema operativo)

Algoritmi di allocazione dei frames per i processi utente

Definiamo come M il numero di processi in coda ready e come N il numero di frame liberi in RAM.
La domanda principale che ci poniamo è: **come distribuisco gli N frame agli M processi?**
Con il *paging on demand* puro, la risposta è: non assegnamo nessun frame a nessun processo (mano a mano il processo richiederà le pagine necessarie).
Una seconda domanda è: **esiste un minimo numero di frame che converrebbe assegnare ad ogni processo?**
E ancora, **conviene assegnare una scala di priorità ai processi?**
Vediamo qualche risposta.

Nota: si definiscono **pagine attive** quelle pagine che sono spesso utilizzate (si trovano nella località attuale del processo). È chiaro che il problema più grave è quando #frame usabili da P < #pagine attive di P, poiché ogni nuova richiesta di pagina ci sarà un page fault.

La distribuzione uniforme (*metodo senza priorità*)
Questo semplicissimo metodo non fa altro che dividere N/M e divide così i frame liberi fra i processi in coda ready. La sua implementazione è estremamente agevole ma chiaramente non è un metodo molto preciso per assegnare frames.

La distribuzione proporzionale (*metodo senza priorità*)
Si cerca di dare al ad ogni processo la quantità di memoria che il processo stesso vorrebbe occupare.
Definiamo come VM^i la quantità di memoria desiderata da un certo processo P^i.
Allora la memoria assegnata sarà = $(VM^i / \Sigma VM^j) * N$, con j che va da 1 ad M (si percorre tutti i processi).

La strategia locale (*metodo con priorità*)
La strategia locale, in caso di page fault, cerca una pagina da killare, la trova e la nuova pagina rimpiazza la vecchia (in sostanza è l'algoritmo di sostituzione delle pagine a gestire la situazione).

La strategia globale (*metodo con priorità*)
In questo caso, quando avviene un page fault il sistema operativo controlla se qualche altro processo (a priorità minore) ha almeno un frame libero assegnato, se sì li "deturpa" al processo a priorità minore assegnandolo invece a quello a priorità maggiore.
Come è intuibile questo effetto iterato molte volte porta ad avere processi di alta priorità con moltissimi frame e processi a priorità minore senza frame (probabilmente in una situazione di #frame < #pagineAttive). Questo comportamento degenere viene detto **trashing**.
A questo punto, la CPU si ritrova a fare sempre meno (i processi sono tutti incespicati fra i page fault) e così crede di poter gestire ancora più processi, e ne carica altri in memoria peggiorando ancora di più la situazione!

Il problema si sposta quindi sul capire quante e quali siano le pagine attive di un certo processo.

Il working set
Il working set è l'insieme delle pagine attive di un certo processo. Il metodo per capire quali esse siano è veramente banale. Viene salvato uno storico delle pagine usate, dopodiché si definisce un δ che ci indica quanti accessi vogliamo "guardare" (partendo dall'ultimo, chiaramente).
Il working set viene definito inserendo quelle pagine che appaiono almeno una volta fra le δ prese come campione.
Esempio
Definiamo δ = 5. Abbiamo uno storico: 1,1,1,2,3,3,1,1,4,4,5,3,4,5
Dunque 4,5,3,4,5 sono le pagine "guardate" tramite il delta, pertanto il nostro W/S = {4,5,3}.

La somma dei working set (D)
Se la somma dei working set di tutti i processi è minore di N allora non è un problema, possiamo tranquillamente assegnare tanti frames quanti richiesti dai processi. Se invece D>N vengono assegnati sei sottoinsiemi dei frames ad ogni processo.

Salvare il working set è piuttosto agevole, possiamo conservarlo nel PCB così quando effettuiamo swap-in/ swap-out abbiamo già anche il vecchio working set e possiamo ripartire assegnando un numero appropriato di frames.

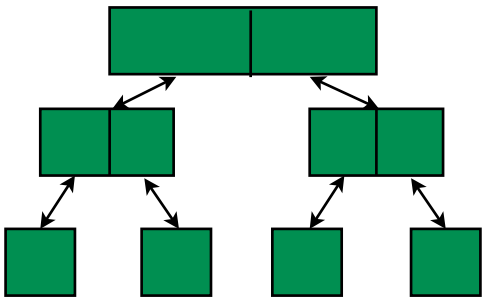
Un altro modo per controllare il trashing è quello di tenere sotto controllo la frequenza dei page faults.

Algoritmi di allocazione dei frames per i processi kernel

L'allocazione di processi kernel deve essere decisamente più accorta rispetto all'allocazione di semplici processi. Questo poiché spessissimo i processi kernel hanno bisogno di salvare variabili temporanee di piccole dimensioni, etc, ed utilizzando gli stessi sistemi sopra esplicati genererebbero una frammentazione interna elevatissima.

Sono pertanto stati elaborati due differenti algoritmi:
1) Il sistema gemellare (buddy)
2) Il sistema a slab (*implementato in Solaris*)

Il sistema gemellare
Supponiamo che ad un certo processo kernel venga riservata una certa sezione della memoria. Ora, il processo deve salvare un dato di 16kb. La memoria assegnata verrà suddivisa di metà in metà (ecco il perché del nome gemelli) fino ad ottenere una partizione sufficientemente grande per contenere il dato ma allo stesso tempo senza sprecare spazio. Avremo quindi la memoria suddivisa in questo modo:



Ed in fondo potremo salvare il nostro dato da 16kb. Intanto gli altri buddy saranno riempiti parzialmente o non, e conterranno altri dati.
Quando due gemelli sono liberi entrambi, vengono fusi nuovamente.
Questa tecnica è stata usata moltissimo, anche se genera frammentazione nel caso in cui dovessimo allocare un dato di 17k per esempio. Dovremmo riservargli ben 32k di memoria.

Il sistema a slab
Per risolvere la frammentazione generata dal sistema precedente, è stato inventata questa tecnica della a slab (lastre, in inglese).
Vengono allocate delle zone di memoria (dette appunto slab) divise secondo il **tipo** delle variabili che conterranno. Ad esempio avremo il tipo delle code di messaggi, etc.

A seconda del tipo del dato che stiamo allocando, quello finirà in una certa slab piuttosto che in un'altra. Nel momento in cui una slab è piena, ne viene allocata un'altra. L'insieme delle slab allocate per un certo tipo, viene detta **cache**.

Il problema della frammentazione è finalmente risolto.

Ultime riflessioni sulla memorizzazione dei processi

Abbiamo spesso appurato che i processi utilizzano file e periferiche di input/output (come stampanti, ad esempio). Come vengono gestite le comunicazioni fra le sezioni di memoria relative a questi elementi ed i processi che le adoperano, dato che i dati di file/stampanti non fanno parte della sezione dati del processo?

Mappatura dei file in RAM
I file vengono mappati in RAM per poter essere utilizzati.
Come sappiamo, la tabella delle pagine dei processi ha sempre qualche entry "inutilizzata". È quindi sufficiente mettere un riferimento dei frame contenenti il file mappato (che verrà caricato piano piano in RAM) nella tabella delle pagine del processo che sta usando il file stesso.

Questa tecnica è ottima anche quando si ha da condividere il file tra più processi, infatti sarà sufficiente mettere i riferimenti anche nella tabella delle pagine dell'altro processo.
Potremo quindi avere P e P' che utilizzano un file, ognuno dei quali ha un riferimento logico per arrivare a quello fisico. I due riferimenti logici sono probabilmente differenti, ma questo non è un problema!

Mappatura dei device di I/O
La comunicazione con i device funziona in modo del tutto simile, ma i frame riferiti conterranno il controller del device. Questo ci fa capire perché talvolta eseguendo un programma C abbiamo le stampe sfalsate: le printf vengono accumulate in qualche frame e solo successivamente inviate al device.

Gli sprechi della memoria condivisa
Dovendo creare una ideale memoria condivisa, avremo da compiere tre allocazioni: un array, un indice di testa ed un indice di coda. Rischiamo di metterli tutti e tre su tre frame differenti generando una frammentazione interna enorme!
Conviene quindi dichiarare una struct e inserirvi i tre dati, quindi allocarla in un colpo solo.

Attenzione al codice!
Il modo con cui scriviamo un codice può influire incredibilmente sulla sua efficienza.
Prendiamo ad esempio un'operazione di inizializzazione di una matrice 128x128 (*Nota: le matrici vengono allocate per righe!)*:

```
for(int i = 0; i<128; i++)
      for(int k = 0; k<128; k++)
            a[i][k]=0;
```

Queste semplici righe di codice ci fanno scorrere la matrice per righe (per ogni riga tutte le colonne).
Supponendo di essere molto sfortunati ed avere ogni riga in una pagina differente, faremo 128 page fault.
Con questo codice, che compie le medesime operazioni:

```
for(int i = 0; i<128; i++)
      for(int k = 0; k<128; k++)
            a[k][i]=0;
```

avremo ben 128^2 page fault! Ogni ciclo del for interno avremo un page fault, quindi 128 per ogni riga (che sono appunto 128).

Conoscere come vengono allocate le nostre strutture dati può essere davvero importante.

Sistemi Operativi
Il file system

Capitolo 7

Indice degli argomenti

Il concetto di file

Cosa è un file?
Un file si può considerare come un insieme di informazioni correlate, identificate da un nome e "spiegate" tramite alcuni metadati (come ad esempio il **tipo**).
Essendo un dato un agglomerato di informazioni, esse possono essere codificate in svariati modi.

Il tipo di un file
A seconda del sistema operativo possiamo capire il tipo di un file in due maniere:
1) estensione: contenuta nella forma *nomeFile.estensione*, l'estensione contraddistingue in maniera univoca il tipo del file "nomeFile".
2) magic number: si tratta di una serie di cifre che identificano il tipo del file e si trovano nel file stesso.

File alfanumerici e binari
I file si dividono in queste due grandi categorie:
1) File alfanumerici (*sorgenti, testo*): sono file leggibili dagli utenti e sono salvati "in righe".
2) File binary (*file oggetto, eseguibili*): sono file non leggibili dagli utenti se non tramite appositi codificatori.

Il descrittore
I file sono descritti dal cosiddetto descrittore, il quale contiene informazioni e metadati ed inoltre un puntatore ai dati del file, cioè ciò che per l'utente è l'unica sostanza del file stesso.

I due livelli di astrazione: logico e fisico

Possiamo introdurre lo studio dei file considerandone gli aspetti logici e quelli fisici.

Livello logico	Livello fisico
- File e directory - Organizzazione delle directory - Operazioni su file e directory - Meccanismi di protezione	- Implementazione delle directory - Rappresentazione dei file - File system virtuale - Allocazione dei file - Ripristino mounting

Il livello logico: Operazioni sui file

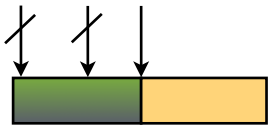
1) Creazione di un file
    1) Trovare lo spazio in memoria
    2) Operazioni sulla struttura dati del file system
2) Lettura/scrittura di un file
    1) Identificare la posizione del file
    2) Identificare la posizione del puntatore all'interno del file
    3) Aggiornare i metadati

3) Riposizionamento di un file
    1) Identificare la posizione del file
    2) Identificare la posizione del puntatore all'interno del file
    3) Spostare il puntatore
    4) Aggiornare i metadati
4) Cancellazione di un file
    1) Identificare la posizione del file
    2) Liberare la memoria occupata dal file
    3) Aggiornare i metadati

Nota:
Per evitare di ripetere le operazioni (1) e (2) durante le letture, sono state creati i metodi come fopen che ritornano un handler, il quale si occupa di aggiornare periodicamente gli indici senza farlo ogni lettura.

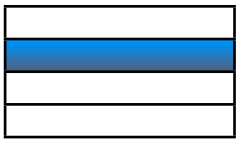
Lettura e scrittura
Vi sono sostanzialmente tre tipi differenti di accesso ai file:

Sequenziale



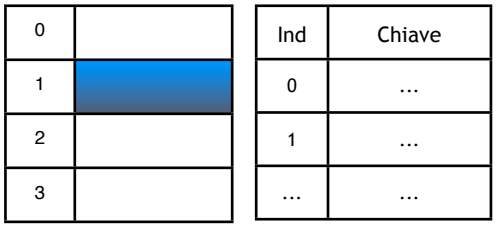
Un puntatore segue sequenzialmente il file, dall'inizio fino alla fine.

Diretto



I dati vengono organizzati in forma di record (tutti delle stesse dimensioni).
È possibile effettuare un accesso diretto (all'elemento j - esimo).

A indice



Tramite l'accesso ad una tabella di chiavi ed indici, data una chiave possiamo arrivare direttamente al dato che desideravamo.

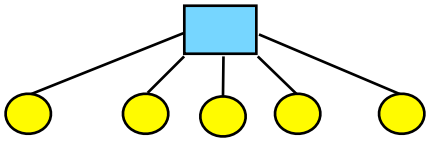
Metodi di controllo
I metodi di controllo sono piuttosto semplici. Abbiamo il **lock**, il quale si suddivide in due tipi:
- lock condiviso (vengono impostate delle restrizioni per alcune operazioni o gruppi)
- lock esclusivo (solo un processo può accedere al file)
L'implementazione può essere poi a **lock consigliato** (viene solo avvertito il processo owner che qualcun altro ha fatto accesso al file) oppure a **lock obbligatorio** (vengono applicate le regole sopra descritte alla lettera).

Un'altra soluzione può essere l'**ACL** (**access control list**) che definisce una lista di "chi può utilizzare il file".

Le directory

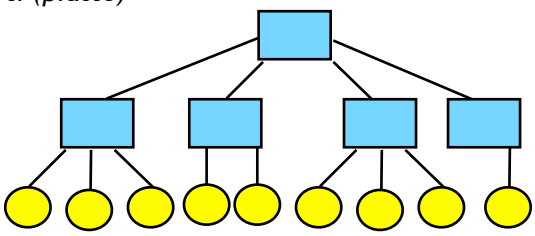
Il concetto di directory ha subito diverse evoluzioni nel tempo. Vediamole.

File system piatto
La prima ideazione del file system era una directory radice con tutti i file dentro:



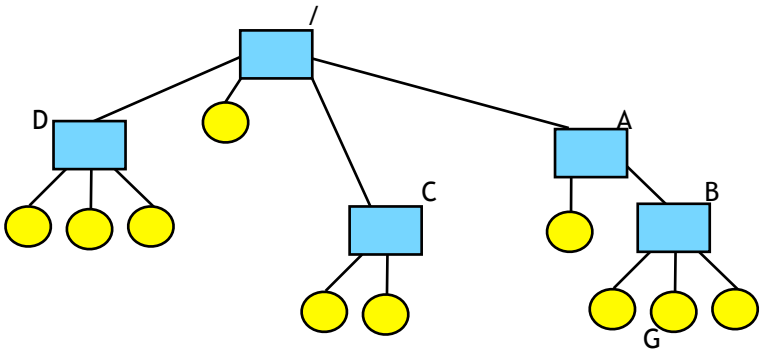
Una volta introdotto il sistema con più utenti, è stato necessario introdurre un sistema leggermente più gerarchico, dove ogni utente ha i suoi file:

File system suddiviso in parti (piatte)

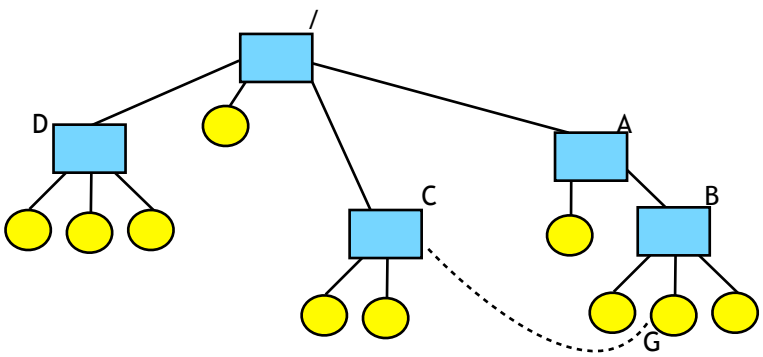


Questo sistema risolve il problema della condivisione fra utenti ma magari gli utenti vorrebbero poter usare stessi nomi per file diversi.

Albero gerarchico di Ritche



Ora per arrivare ad una cartella è sufficiente esplicitare un **percorso**: /A/B/G
La necessità di condividere le cartelle ha poi generato una variante, detta *grafo aciclico*:



Il grafo aciclico introduce il concetto di **link** (collegamento). Ora il file G è accessibile non solo più dal percorso /A/B/G, ma anche da /C/G.

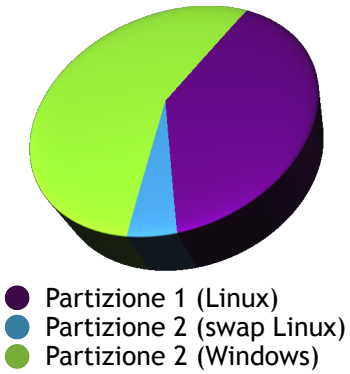
Esiste ancora un'altra possibilità, detta *grafo ciclico*, nel quale è possibile effettuare collegamenti anche all'indietro, e cioè creare un collegamento che da D porta ad /.

I tipi di link
Nella rimozione di un link, vogliamo eliminare anche il file correlato oppure no?
Esistono due soluzioni per tipare i link:
A)
Link simbolico: è un semplice collegamento, la sua eliminazione non modifica il file
Hard link: è un collegamento vero e proprio, la sua eliminazione comporta l'eliminazione del file collegato.
B)
Il sistema operativo conta il numero di collegamenti cui un file è collegato: quando il numero diventa 0, allora il file viene eliminato.

L'operazione di mount

L'operazione di mount consiste nell'agganciare differenti alberi di file system in uno solo, rendendo così trasparente all'utente l'intera l'implementazione. L'utente non vedrà altro che cartelle.
Il mount può accadere in due casi: nel bootstrap oppure durante l'esecuzione. Ma prima di vedere nel dettaglio i due casi, parliamo delle partizioni.

Perché partizionare?
Partizionare il disco significa dividerlo logicamente in più sezioni, sulle quali è possibile agire in maniera distinta ed indipendente.



I motivi per farlo sono diversi. In ogni caso, linux, crea due partizioni del disco di default, una delle quali è l'area di swap mentre l'altra contiene il sistema operativo vero e proprio (contente quindi la home).

Partizionare il disco ci permette di avere contemporaneamente sullo stesso supporto più file system.

Come detto i motivi per cui partizionare sono vari.

1) Il nostro hard disk è troppo grande (diciamo M) rispetto alla dimensione che il nostro file system è in grado di indirizzare (diciamo N). Creiamo così una partizione P1 con dimensioni < N e una P2 < N. Ovviamente la somma delle due partizioni darà M.
2) Vogliamo installare diversi file system per applicazioni differenti (ad esempio un file system di rete, un file system più performante per la nostra applicazione, ecc).
3) Vogliamo dividere il sistema operativo dal resto dei dati, così da poterlo (volendo) cambiare più semplicemente, ed in caso di corruzioni del sistema non perdere i nostri dati.

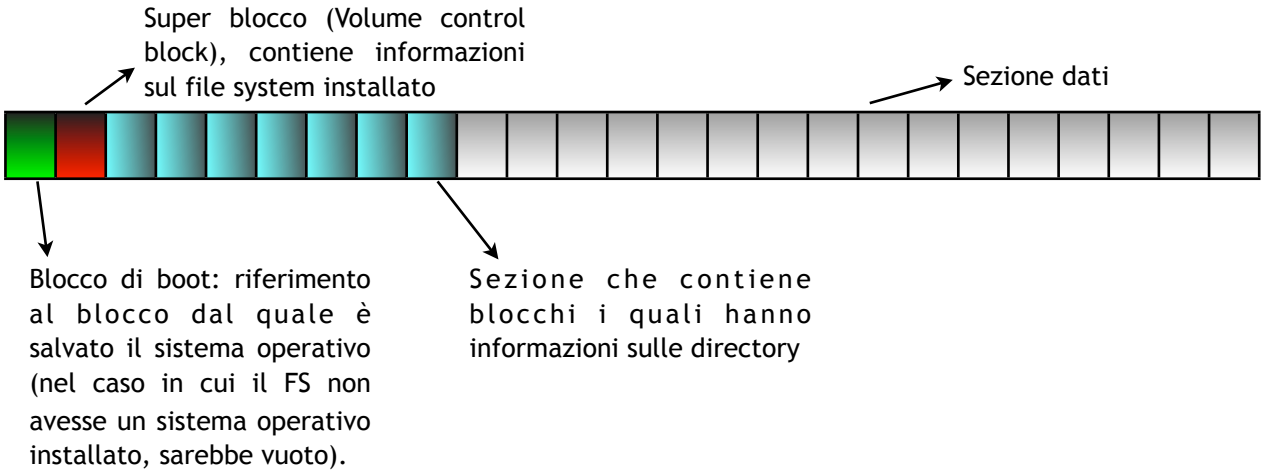
Mount in bootstrap e mount in esecuzione
I due tipi di mounting (collegamento tra gli alberi di diversi file system) sono effettuati semplicemente in momenti diversi. Il primo è (come dice il nome) effettuato nel momento di boostrap e comprende il collegamento tra le partizioni della memoria secondaria. La cartella di riferimento in linux è etc/ftstab.
Il mounting in esecuzione è quello che accade ad esempio all'inserimento di un dispositivo USB. La cartella di riferimento in linux è /media.

Livello fisico

Il ruolo del livello fisico è quello di far sì che la struttura ordinata vista dall'utente sia effettivamente sotto controllo. La memoria secondaria è organizzata in blocchi: quantità di byte fissa e definita. Tutto questo viene effettuato tramite le seguenti astrazioni:

1) *Livello logico:* file e directory sono gestiti a livello di metadati. Abbiamo un **inode** che tiene le informazioni relative al file/directory
2) *Modulo di organizzazione dei file:* vengono tradotti gli indirizzi logici in indirizzi fisici. Vengono anche tenute informazioni riguardanti il numero di blocchi liberi
3) *File system di base*: i comandi che arrivano dai livelli superiori vengono passati al driver (operazioni di lettura e scrittura)
4) *Driver del dispositivo*: le operazioni vengono eseguite (tramite la comunicazione col controller hardware)

A livello molto basso, questo è ciò che abbiamo:



La gestione delle directory
Come si è visto c'è una sezione della memoria dedicata alla gestione delle directory. Come potremmo gestirle? Al loro interno potremmo ad esempio piazzare una serie di coppie <nomeFile, #inode1>, ma la ricerca sequenziale sarebbe poco performante oltre che inappropriata.

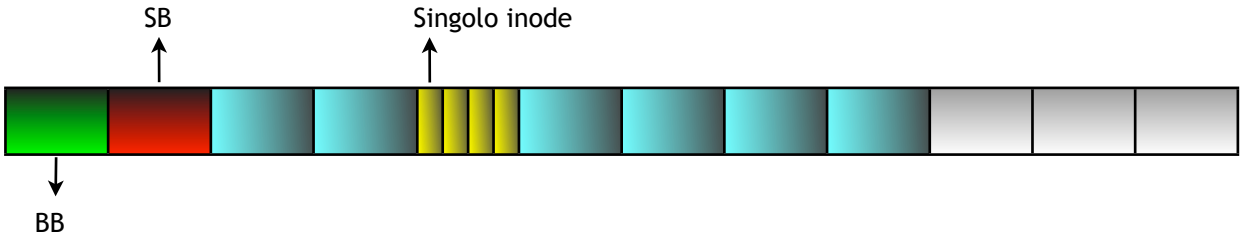
Introduciamo quindi il concetto di inode, per capire meglio la soluzione adottata.

L'inode ha due "parti" :
(1) su disco
(2) in RAM (*in-core inode*)

L'inode su disco (File Control Block)

L'inode su disco mantiene le seguenti informazioni:
✓ Proprietario del file
✓ Tipo del file
✓ Tipo di codifica dei diritti di accesso
✓ Dimensione del file
✓ Numero di link al file
✓ Data/ora di ultima modifica
✓ Posizione dei dati dal file rispetto al suo inode

Inoltre, il programmatore del file system deve prendere alcune decisioni implementative a riguardo dei nomi dei file, come ad esempio qual'è la massima lunghezza del nome di un file oppure quali caratteri sono permessi.



Gli inode sono di dimensione fissa, e sono contenuti nei blocchi sopra indicati in blu. Dato che ogni blocco ha dimensione fissa e anche gli inode hanno questa caratteristica, il numero di inode contenuti in un singolo blocco è costante. Anche il numero di inode totali è fisso, quindi il sistema non sarà in grado di salvare più di un certo quantitativo di file. Nel momento in cui si installa il sistema, la maggior parte degli inode saranno chiaramente inutilizzati.

Alcuni degli inode saranno assegnati a directory.

Dunque, dato un nomeFile possiamo ottenere il suo #inode (un indice assoluto ed univoco).
Dato il suo #inode non resta che tradurre questo numero per ottenere prima il blocco che contiene l'inode stesso, e poi identificare la sua posizione relativa per poi ricollegarsi ai dati dei file.

(1) *Identificazione del blocco che contiene l'inode:*

B = (#inode / #inode_per_blocco) + #offset_dei_blocchi_iniziali

(2) *Individuazione del displacement:*

D = (#inode % #inode_per_blocco) * dimensione_inode

La coppia <B,D> identificherà un singolo inode, grazie al quale possiamo ricavare l'indirizzo dei dati.

L'inode in RAM

L'inode in RAM mantiene le seguenti informazioni:
✓ Alcuni degli inode del disco
✓ Stato (locked, diversità dalla copia in RAM rispetto a quella su disco, è un punto di mount?)
✓ Codice che identifica a quale device l'inode appartenga

Inoltre la RAM serba le seguenti strutture dati globali:
✓ Mount table (tiene traccia dei devices montati)
✓ Tabella file aperti
✓ Tabella delle directory
ed una non globale (di ogni processo)
✓ Tabella dei file utilizzati (ad esempio stdin, stdout, ecc)

Dal nome del file al suo inode

Abbiamo ampliamente parlato di inode, ma non abbiamo specificato come è possibile ottenerne uno dato un nome di un file.
Innanzi tutto è bene chiarire che un nome di un file è in realtà un path, cioè il percorso che c'è dalla radice del file system fino al file stesso. Qualcosa come /cartella1/cartella2/nomeFile.x
Il path è reperibile in due modi, potrebbe derivare dalla root oppure dalla cartella corrente (working directory - wd).
Una volta ottenuto un percorso (sottoforma di stringa) vediamo qual'è l'algoritmo che può interpretare il path in maniera corretta.

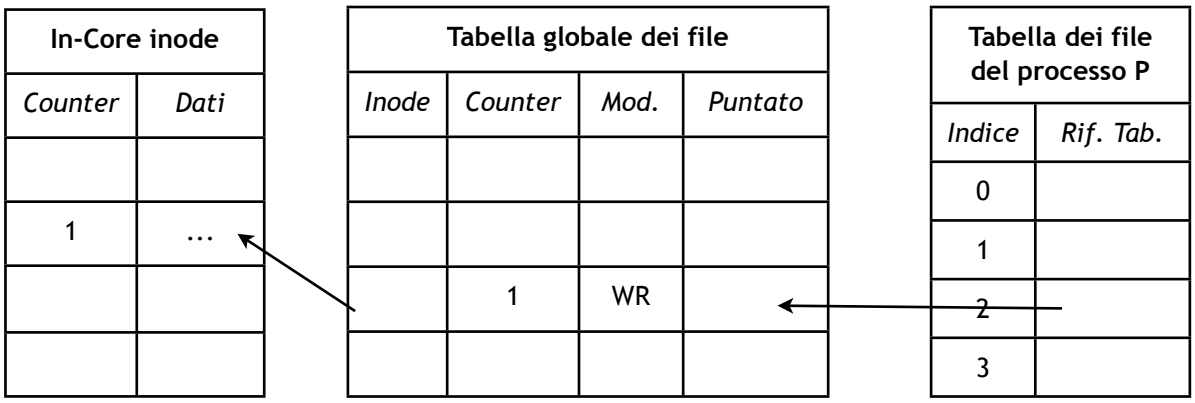
```
if(current = '/') //se il primo carattere è una slash
      current_i = root_inode;
else
      current_i= wd_inode;

repeat
      current = leggi_elem_successivo;
      if(current == NULL) //fine path
            return current_i;
      else if("current è contenuto in current_i"){ //il nome di ciò che ho letto
                                                    //è contenuto effettivamente
                                                    //nell'inode delle cartella
                                                    //precedente
            current_i = inode_identificato
            else
                  return (no_inode); //file non trovato
until (current != NULL)
```

Apertura di un file

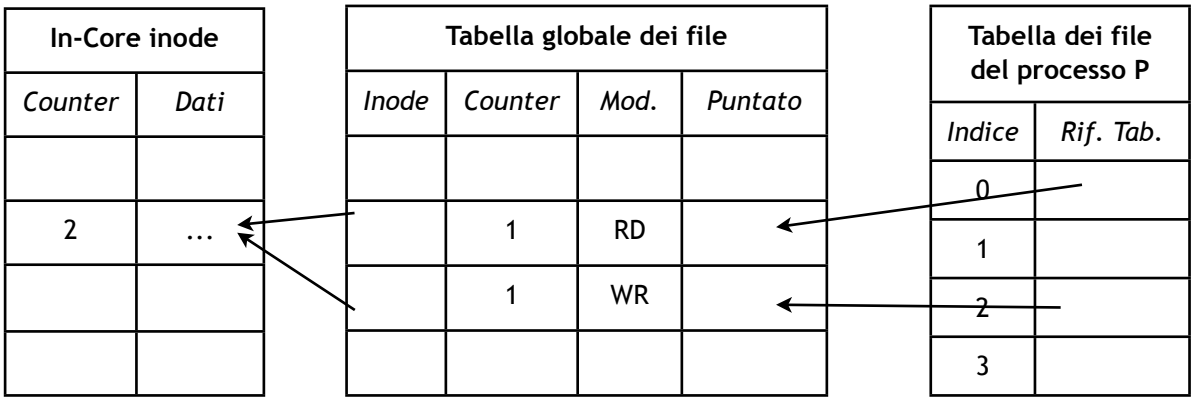
Per aprire un file esiste la system call open("proc.c", O_RDONLY).
Come abbiamo detto in RAM risiedono le seguenti tabelle:
- Tabella degli in-core inode, (dei file attualmente aperti)
- Tabella globale dei file (riferimenti agli in-core inode, puntatore alla posizione cui si è arrivato a scrivere, contatore)
- Per ogni processo esiste una tabella dei file che fa riferimento a quella globale.

Diamo uno sguardo alla RAM e capiamo la situazione.

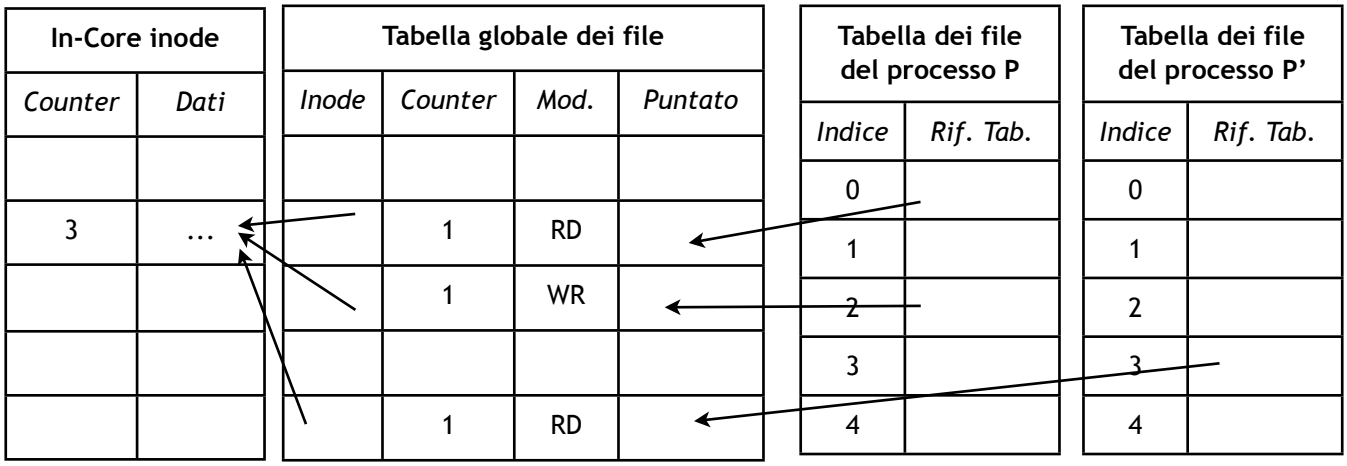
La prima operazione che consideriamo è una open in write da parte del processo P.



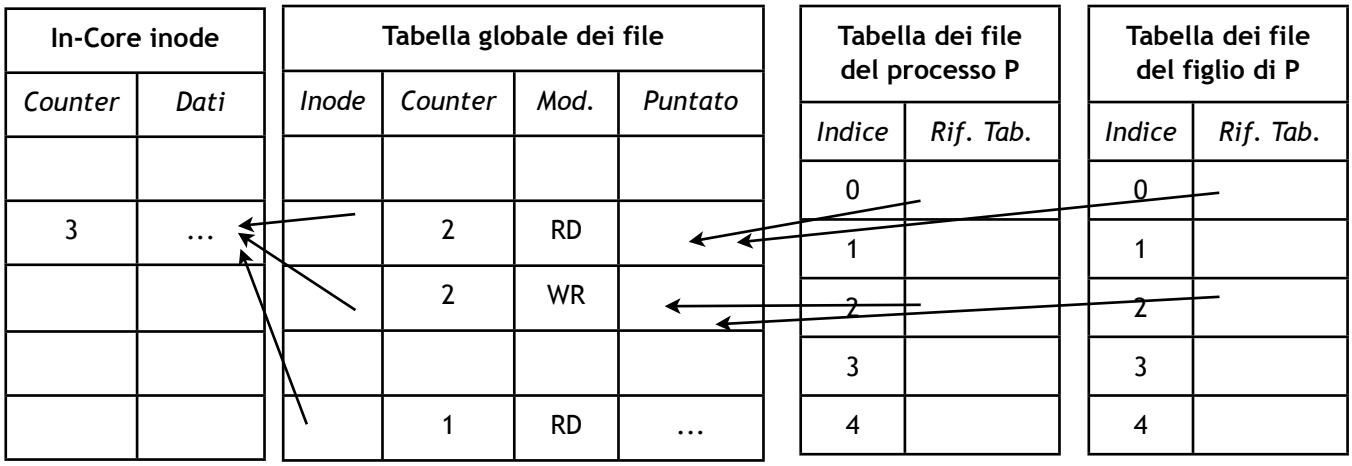
Ora il processo P farà riferimento al record nella tabella globale dei files, la quale fa riferimento all'inode relativo che si trova in ram. La modalità è di lettura.
Ora P vuole accedere al file in scrittura. Questo richiede un nuovo record!



Ora il counter dell'incore node è di due, poiché due file puntano ad esso. Un certo processo P' interagisce, facendo una open in read sul file.



Il counter sulla tabella globale dei file viene modificato in un solo caso: la fork. Supponiamo che il processo P si forki, avremo quindi (la tabella di P' esiste ancora ma non è rappresentata):



Chiusura di un file

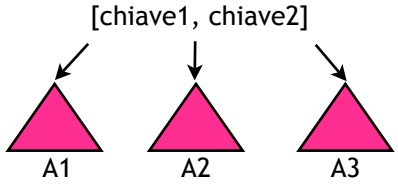
La chiusura di un file è piuttosto semplice grazie ai contatori che abbiamo implementato prima. Se il counter della tabella globale dei file vale N ed il counter dell'incore-node vale M, verrà eseguito qualcosa come:

```
N = N - 1;
if (N == 0) disalloca record_tabella_files_globale
      if(disallocato) {
            M = M - 1;
            if (M == 0) disalloca in-code_node
      }
}
```

L'implementazione delle directory

Come abbiamo visto, le directory potrebbero essere implementati come **listati** di <nomeFile, inode> (ma occorrerebbe una lettura sequenziale che è poco performante).
Un'altra soluzione potrebbe essere una **hash table** o eventualmente un **B-Tree**.
Vedremo l'ultima soluzione.

I B-Tree bilanciati
Se un nodo ha k elementi, allora il nodo deve avere k+1 puntatori ai prossimi.
SIn nodi sono pertanto concepiti in questo modo:



A1 è minore della chiave 1, A2 è compreso tra chiave1 e chiave 2, mentre A3 è maggiore di chiave3.
Le chiavi potrebbero essere, per esempio numeri di blocco. Si definisce **fan-out** la media di chiavi per nodo (cioè quanto l'albero è equilibrato in sostanza). La complessità di spostamento tra nodi è log(n).
Esempio: con una fan-out di 25 e 10.000 elementi totali, un qualsiasi elemento può essere trovato in 4 passi.

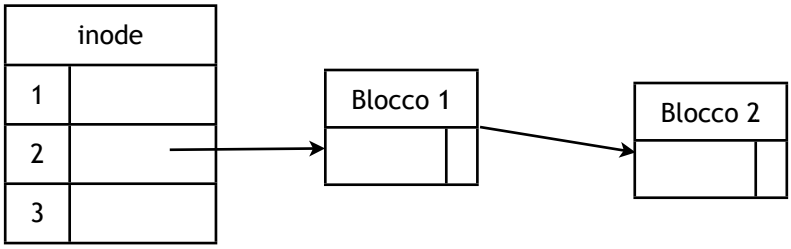
Metodi di allocazione per i file

Come vengono allocati in memoria i file? Abbiamo visto tramite l'inode, ma ci sono diversi modi di gestirlo.
Ricordiamo che i file sono indirizzabili utilizzando tre metodi: sequenziale, diretto ed indicizzato. Data questa considerazione, esamineremo i metodi di allocazione tenendo conto delle necessità dei tre metodi di indirizzamento sopra indicati (accorpando diretto ed indicizzato che hanno le stesse "esigenze").

Allocazione contigua
I blocchi vengono allocati fisicamente in maniera contigua.
L'indirizzamento sequenziale è ovviamente favorito ma anche il diretto/indicizzato è performante (essendo i blocchi vicini, possiamo facilmente raggiungere quelli puntati).
Sorge però un problema quando il file supera la dimensione dei blocchi a lui assegnati. Le soluzioni possibili potrebbero essere:
- Shiftare tutti i blocchi successivi per ottenere lo spazio necessario per immagazzinare le nuovi parti del file
- Spostare i blocchi del file in questione in una zona con più spazio libero.

Entrambe le operazioni genererebbero però frammentazione esterna derivante dalla cancellazione dei blocchi.

Allocazione concatenata
Si potrebbe pensare di dedicare una piccola parte di ogni blocco per un puntatore ad un nuovo blocco. Avremmo una situazione del genere:



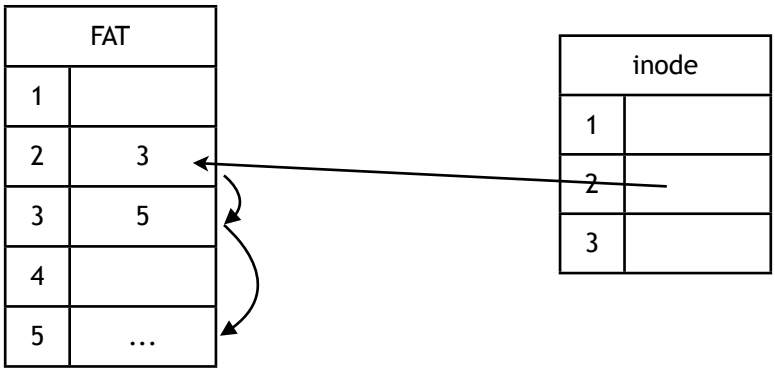
Nel momento il cui il file aumenta le sue dimensioni, è sufficiente allocare un nuovo blocco e collegarlo all'ultimo.
La ricerca sequenziale è funzionale, ma quella ad accesso diretto/indicizzato è invece più difficoltosa: il costo è quello di percorrere una lista che potrebbe avere blocchi posizionati in maniera disparata nell'hard disk (ed il movimento delle testine è costoso in termini di tempo!).
Nel caso in cui uno dei blocchi si corrompa, inoltre, l'intero resto del file è perduto!

FAT (File allocation table)

Viene sostanzialmente inserito il sistema di lista precedente in un array, cioè i puntatori immagazzinati in celle di memoria contigue (una tabella).

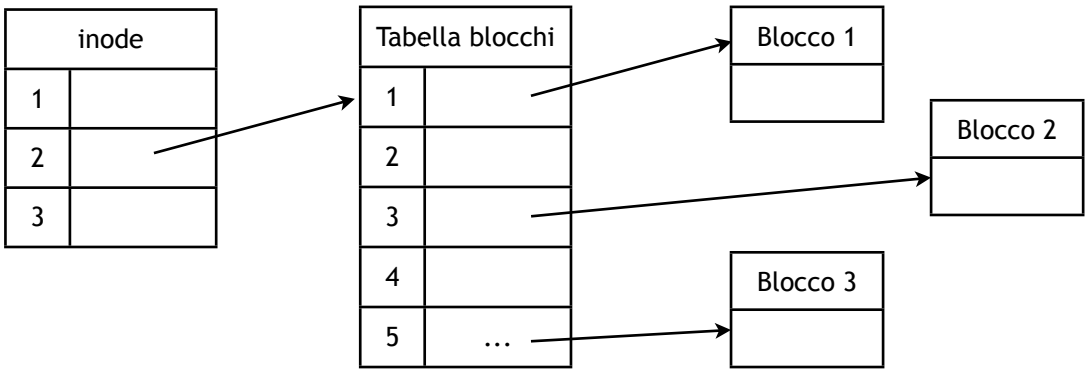
Questa soluzione risolve il problema precedente della distribuzione dei blocchi poiché l'array essendo contiguo è distribuito in pochi blocchi, quindi l'accesso in memoria è veloce.

I file system FAT sono nominati FATX, dove X è il numero di bit riservati per salvare un record nella tabella.



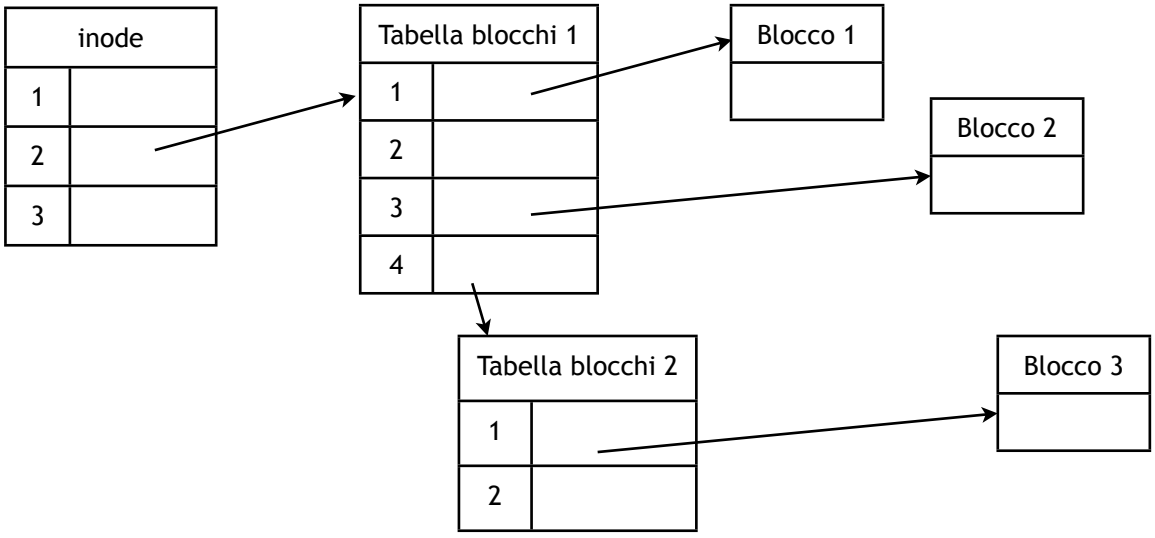
Allocazione a blocco indice

L'inode tiene l'indirizzo di una tabella la quale tiene tutti gli indirizzi dei blocchi in cui c'è una parte del file. Avremo quindi una situazione del genere:

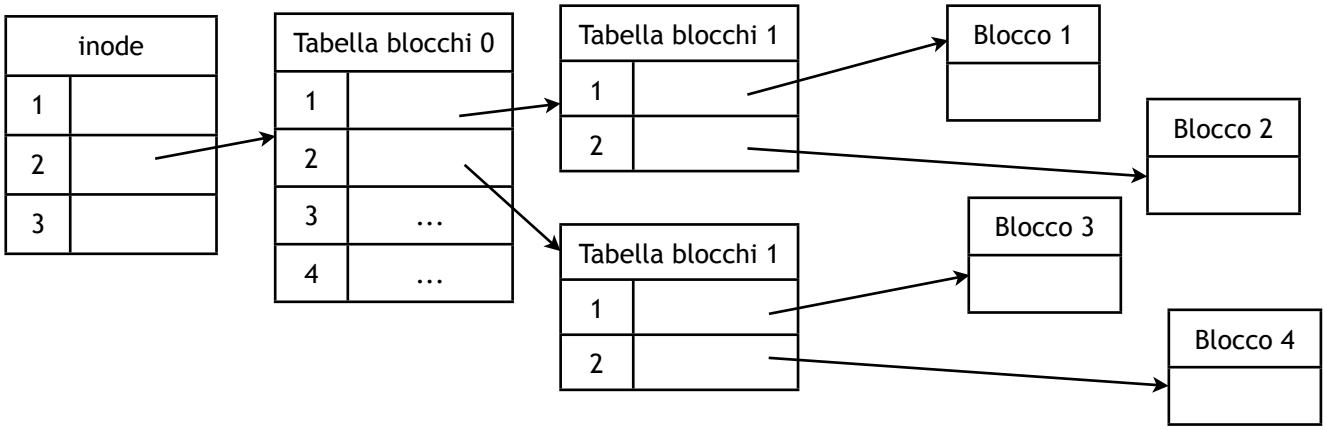


La ricerca sequenziale perde in velocità rispetto ai metodi precedenti, però è favorito l'accesso diretto. Anche qui la massima dimensione del file può essere un problema: le tabelle dei blocchi sono di dimensioni finite.

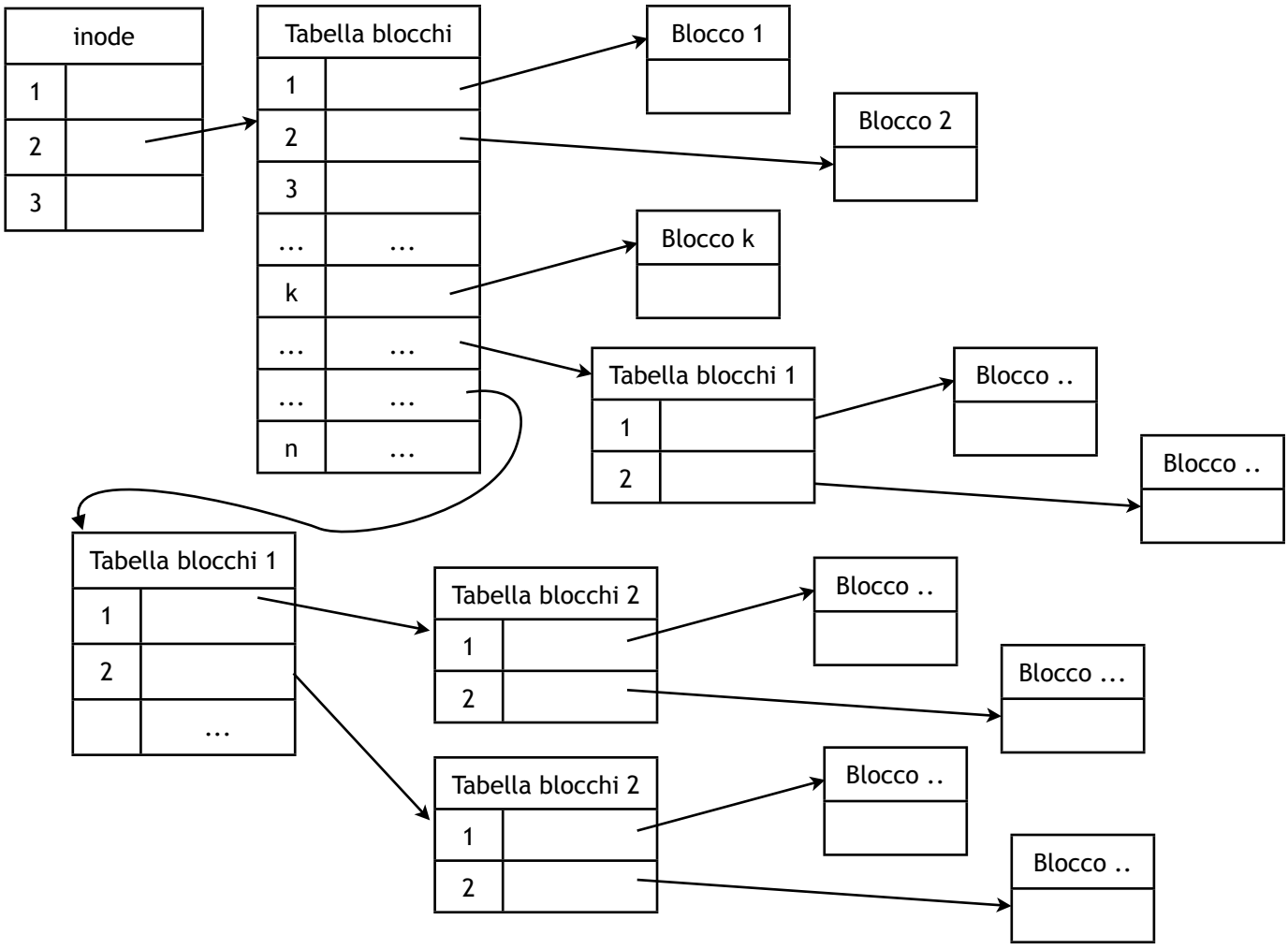
Risolvere questo problema è semplice, potremmo pensare di implementare delle **tabelle a blocco indice concatenate**, ottenendo quindi un effetto del genere:



Questa soluzione ibrida potrebbe sembrare la migliore, ma invece una implementazione differente può essere ancora più performante: **i blocchi indice multilivello**.
All'inode è collegato un blocco indice al quale è collegato un ulteriore blocco indice che è finalmente collegato ai blocchi dati.



Questa implementazione ha però un enorme difetto: l'accesso ad un dato ci costa ben tre accessi in memoria. Si arriva così a parlare della versione che viene usata anche nei sistemi unix, una versione che trae il meglio da ogni metodo.



Questa struttura ha i primi k nodi ad accesso diretto (così finché il file è di dimensione contenuta, l'accesso ad esso diventa molto veloce).
Poi, i successivi n-k posti sono occupati da tabelle indicizzate multilivello, mano a mano con più livelli.

Ad esempio, in unix abbiamo 15/19 blocchi indirizzati direttamente, una tabella multilivello (ad un livello), una tabella multilivello (a due livelli) ed una coda multilivello (a tre livelli).

La tabella iniziale ha una dimensione massima, ma è così grande da non preoccupare l'utente. Con la configurazione precedente (quella di unix), infatti, possiamo indicizzare file grandi fino a 1TB.

La gestione dei blocchi liberi

In qualche modo il supporto deve poter conoscere la quantità di blocchi liberi ancora disponibili. Abbiamo diversi modi (tutti comunque parecchio semplici):

Vettore di bit
Abbiamo un vettore di bit, ognuno dei quali è riservato ad un blocco. Se un blocco è pieno, il suo bit è a zero, se un blocco è vuoto è ad uno.
Avremo quindi qualcosa del genere: 00011101110101010110.
Questa memorizzazione, per quanto rudimentale, ci permette di individuare anche le zone di **extent**, cioè le zone del disco che hanno più blocchi consecutivi liberi [quelle con più uni affiancati] (e quindi dove si tende a memorizzare un nuovo file).

Lista concatenata di blocchi liberi
Si implementa una lista concatenata di blocchi liberi, è semplice prelevarne uno così come inserirlo.

Indice dei blocchi liberi (gerarchico)

Si salva in una tabella qualcosa del genere:

Tabella blocchi liberi	
2	7
5	1
7	2
8	1

Questa rappresentazione ci dice che a partire dal blocco 2 abbiamo 7 blocchi liberi. Questo sempre per cercare di localizzare le extent.

Riassunto da: Enrico Mensa

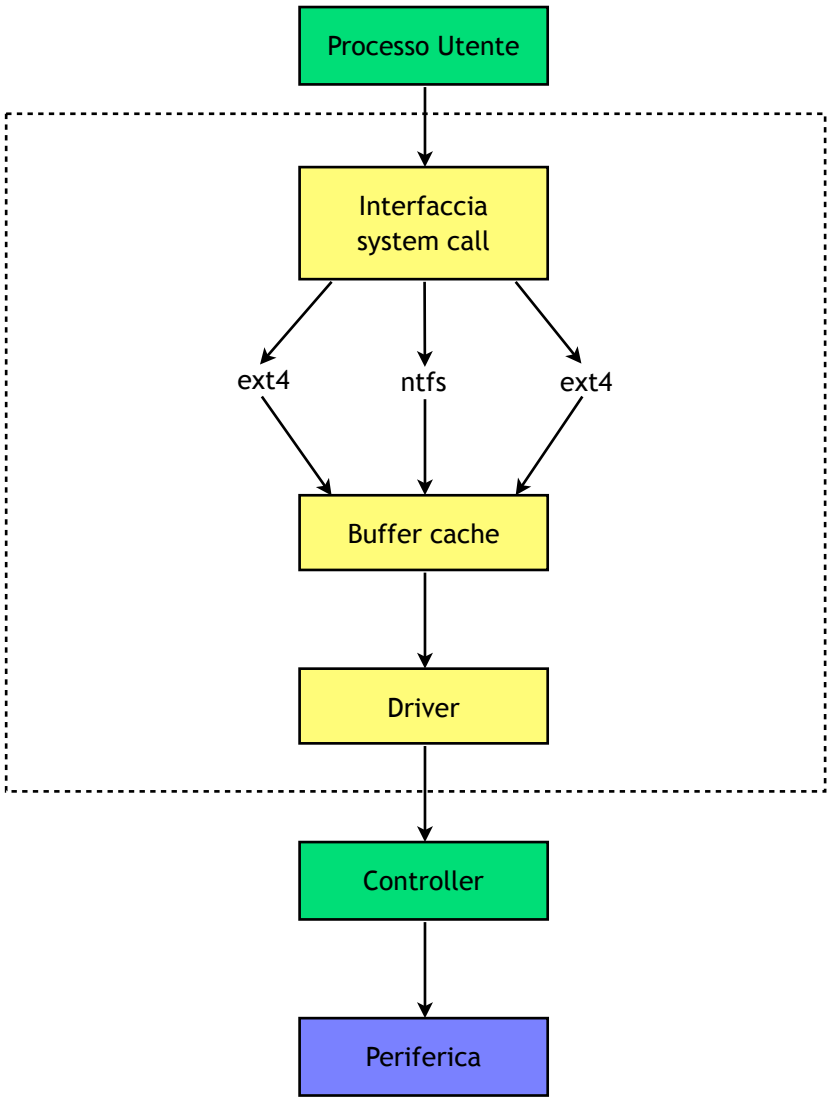
Sistemi Operativi
Il file system virtuale

Capitolo 8

Indice degli argomenti

L'albero del filesystem

Tutto ciò che abbiamo studiato sul file system è implementato in una struttura più ampia e complessa: il file system virtuale. Questa è la sua struttura:



L'interfaccia system call permette a tutti i file system di comunicare con il processo utente e viceversa, in modo che egli non si accorga di nulla. Le operazioni minime fornite sono quelle di **open**, **close**, **read**, **write**.

Ogni file system che voglia essere allacciabile agli altri deve implementare l'interfaccia del VFS (*virtual file system*).

Il buffer cache
Il buffer cache è, come dice il nome stesso, sia un buffer e sia una memoria cache. La prima parte è derivante dal fatto che i processi scrivono in questo buffer il quale conserva i dati prima della scrittura sulla periferica, che è solitamente molto più lenta del sistema. Viene quindi implementato un sistema **produttore/consumatore** dove il produttore è il processo utente e il consumatore è il controller della periferica.

Il frangente cache sta invece nel fatto che quando viene letto un file, viene applicato il principio di località e cioè vengono caricati i dati adiacenti a quello richiesto (poiché si prospetta un utilizzo prossimo).

Attenzione! Se il buffer dovesse essere sovrascritto troppo velocemente, il controller potrebbe non essere in grado di memorizzare subito tutti i dati. Vengono così implementati dei controlli ad hoc.

Il mount/unmount sicuro

Come detto sopra, si rischia di rendere inconsistenti i dati molto spesso.
A questa problematica, il vecchio filesystem ext2 prova ad ovviare in questa maniera: viene assegnata alla memoria una flag. Alle operazioni:

mount --> flag ad unclear (ci *potrebbe* essere qualcosa nel buffer)
unmount ---> flag a clear (il buffer è necessariamente pulito)

Ogni volta che viene effettuato una operazione di mount si controlla che la memoria sia clean. Se non lo è, viene effettuata una operazione detta **fsck**, cioè si cerca di recuperare i dati.
Tale operazione è direttamente proporzionale alla dimensione della memoria!
In grandi database come potrebbero essere quelli di un ospedale, l'intero servizio potrebbe essere shutdownato per intere ore. Si necessita di una soluzione più performante:

Il journaling
Si è così pensato di creare un file di log particolare che tiene in memoria il contenuto del disco. Più specificatamente, abbiamo due tipi di informazioni:

(1) sta per avvenire una write di: DATO
(2) è avvenuta la write di: DATO.

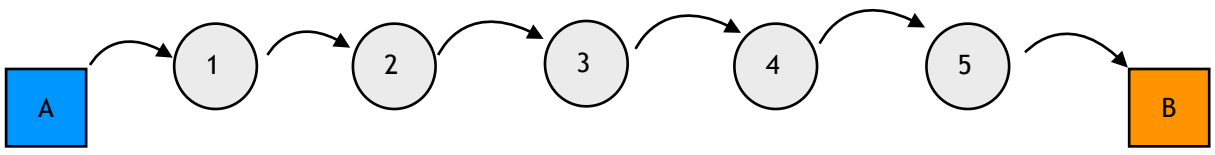
Così, ad ogni avvio, è sufficiente fare un check sul file di log... se è stato interrotto tra la fase uno e la fase due, è sufficiente recuperare DATO dalla riga 1.

Chiaramente il file di log non può avere una lunga storia occuperebbe tantissimo, ma non ci sarebbe in ogni caso utile: dobbiamo solamente serbare gli ultimi avvenimenti.

File system a transazione

Una transazione è una serie di passaggi logici indistinguibili (cioè da soli non avrebbero senso e sarebbero inconsistenti).
Possiamo pensare ad un aggiornamento di qualche sistema operativo o programma.

Nel momento in cui si compie una read/write, nel caso in cui qualcosa vada storto, bisogna essere in grado di riportare le cose allo stato iniziale per evitare di rovinare i dati su cui si stava lavorando.



Abbiamo quindi bisogno di sapere due cose:
La transazione è andata a buon fine? Se sì, si effettua una operazione di **commit** (si avverte che tutto è andato bene).
Se qualcosa è andato storto, si effettua una **rollback** cioè si torna allo stato iniziale (nel nostro caso, A).

Potremmo essere interessati a operazioni di questo genere su diversi hardware:
- volatili (RAM)
- non volatili (EEPROM)
- stabili (Hard Disk)

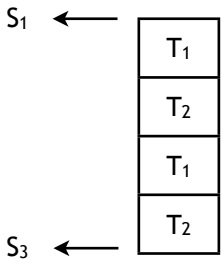
Implementazione della rollback su hardware volatili
Si utilizza una tecnica già vista nel journaling: il log file. Il log file prende la configurazione già vista:

```
<checkpoint>
<Ti, start>
<Ti, ID_VRB, OLD_VAL, NEW_VAL>
<Ti, end>
```

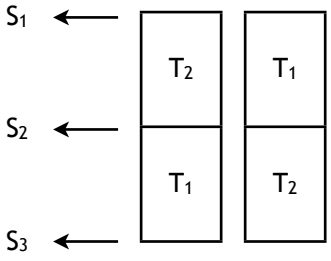
Nel caso in cui il log file non contenga la end della transazione, allora viene semplicemente eseguita la rollback (`ID_VRB = OLD_VAL`).
I checkpoint non sono altro che punti sotto i quali siamo sicuri che la memoria abbia già salvato i dati, e quindi non è necessario leggere sopra di essi.

Transazioni e concorrenze

Abbiamo due variabili, A e B.
Vorremmo poter effettuare dei cambiamenti ad entrambe. Si potrebbe pensare di far utilizzare la risorsa in mutua esclusione, ma per motivi di performance questa scelta non viene effettuata.
Si permette invece una esecuzione concorrente, purché venga rispettato il vincolo che dallo stato S_1 si arrivi allo stato finale (diciamo S_3) senza altri stati. Ovvero, supponendo di



vale la regola che tale esecuzione deve essere come se l'esecuzione fosse una di queste due:



avere questa esecuzione:

Le operazioni conflittuali ci sono quando:
(1) si fa riferimento alle stesse variabili
(2) le scritture sono sequenziali
(3) è coinvolta almeno una operazione di write

Esempio di esecuzione di una transazione

Abbiamo due variabili,

A
3

B
3

e due transazioni

T_1
read(A) A = A+2 write(A) read(B) B = B+3 write(B)

T_2
read(A) A = A+2 write(A) read(B) B = B+3 write(B)

L'interleaving potrebbe portare sia risultati positivi che negativi.
Alla fine vogliamo ottenere A = 7 e B = 11 (cioè compiere le due operazioni in maniera sensata).

Vediamo questa possibile esecuzione:

T_1	T_2
read(A)	-
-	read(A)
write(A)	-
-	write(A)
read(B)	-
write(B)	-
-	read(B)
-	write(B)

I risultati sarebbero per A inconsistenti, per B invece corretti. Abbiamo quindi l'esempio di un interleaving funzionante e di uno errato.

Dobbiamo trovare un modo intelligente per escludere le esecuzioni pericolose e permettere quelle non pericolose.

Primo protocollo per il controllo delle transazioni
Queste gestioni sono basate sul lock.
Ogni transazione ha uno stato, s (shared) oppure x (executed).

L'esecuzione diviene quindi un'alternarsi di due fasi:
(1) fase di crescita - si accumulano i lock.
(2) fase di riduzione - mano a mano vengono eseguite le transizioni ed il numero di lock si riduce.

Attenzione! Rischio deadlock!

Secondo protocollo per il controllo delle transizioni (basato su time_stamp)
Come sappiamo, il time_stamp (d'ora in poi TS()) è un indicatore temporale.

Vogliamo rendere sempre vera questa condizione: $TS(T_1) < TS(T_2)$ in maniera da rispettare la sequenza T_1 e poi T_2.

Dobbiamo inoltre definire R(D) e W(D) [dove D è un dato] che sono rispettivamente il time_stamp di lettura e di scrittura più alti mai ottenuti dalle transizioni che hanno operato su quel dato D.

Non resta ora che definire le due regole di lettura e scrittura per capire di cosa stiamo parlando.

Regola di lettura per il protocollo

(1) Se T desidera leggere D:

(1.a) TS(T) < W(D)

qualcuno ha già scritto su D quindi interrompo T, effettuo un rollback, assegno un nuovo time_stamp a T.

(1.b) TS(T) >= W(D)

si consente la lettura a T

aggiorniamo R(D); R(D) = max(R(D), TS(T));

Regola di scrittura per il protocollo

(2) Se T desidera scrivere D:

(2.a) TS(T) < R(D)

la variabile è già stata letta da qualcuno quindi interrompo T, effettuo un rollback, assegno un nuovo time_stamp a T.

(2.b) TS(T) < W(D)

sovrascriveremmo la scrittura di qualcun altro quindi interrompo T, effettuo un rollback, assegno un nuovo time_stamp a T

(2.c) entrambe le condizioni superiori sono false, quindi possiamo permettere la scrittura ed aggiorno W(D) = TS(T).

Esempio

$TS(T_1)$	$TS(T_2)$
1	2

R(A)	W(A)
0	0
R(B)	**W(B)**
0	0

T_1	T_2
read(A) read(B) write(A) write(B)	write(B) read(A) write(B)

Si utilizzerà un interleaving casuale

Esegue T_1 :

read(A) --> ($TS(T_1)$ < R(A)) --> (1 < 0)? No!

- lettura consentita; R(A) = 1;

read(B) --> ($TS(T_1)$ < R(B)) --> (1 < 0)? No!

- lettura consentita; R(B) = 1;

R(A)
1
R(B)
1

Esegue T_2 :

write(B) --> ($TS(T_2)$ < R(B)) --> (2 < 1)? No!

--> ($TS(T_2)$ < W(B)) --> (2 < 0)? No!

- scrittura consentita; W(B) = 2;

read(A) --> ($TS(T_2)$ < W(A)) --> (2 < 0)? No!

- lettura consentita; R(A) = 2;

W(B)
2
R(A)
2

Esegue T_1 :

write(A) --> ($TS(T_1)$ < R(A)) --> (1 < 2)? **SI!** ---> Rollback! Ora avremo

$TS(T_1)$	$TS(T_2)$
3	2

Riassunto da: Enrico Mensa

Sistemi Operativi
La memoria secondaria

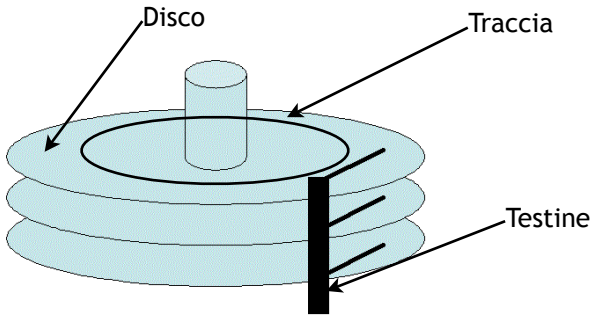
Capitolo 9

Indice degli argomenti

La memoria secondaria

La memoria secondaria è un termine piuttosto generico.
Esistono supporti di vario genere.

Supporti magnetici
Il primo supporto magnetico nasce nel 1888.
Da questa immagine possiamo capire come è composito.



Ogni traccia è divisa in settori. Un insieme di tracce "verticali" è detta cilindro.
I dischi magnetici sono percorsi a velocità angolare costante (i dischi girano a velocità costante).

Nel calcolo del tempo di accesso ad un blocco vanno considerati quindi:
- tempo di seek (identificazione della traccia su cui scrivere/leggere)
- tempo di latenza (movimento del disco per posizionare in maniera corretta la testina)

Il tempo può essere velocizzato tramite lo scheduler del disco (di cui parleremo dopo).

Supporti ottici
Altri supporti sono quelli ottici, che non tratteremo, come i CD per esempio. I dati sono immagazzinati "a spirale", quindi hanno velocità lineare costante. Noi parleremo solamente di dischi magnetici.

La formattazione

Esistono due tipi di formattazione del disco.

La formattazione fisica
Effettuata dalle case produttrici, la formattazione fisica genera i settori. All'inizio di un settore (**in testa**) viene sistemato il suo numero, mentre alla fine (**in coda**) viene messo un codice di correzione (come ad esempio potrebbe essere Hamming.

La formattazione logica
La formattazione logica, fatta dall'utente, è portata dall'installazione del sistema operativo, il quale utilizzerà un dato file system.

Lo scheduling del disco

Le richieste di scrittura e lettura vengono accumulate nella cosiddetta **coda di richieste**. Così come è stato per i processi, anche questo tipo di richieste hanno bisogno di essere in qualche maniera ordinate.
I fattori da tenersi in considerazione sono:
- quantità di read/write
- system call
- indirizzamento della RAM
- indirizzamento sul disco

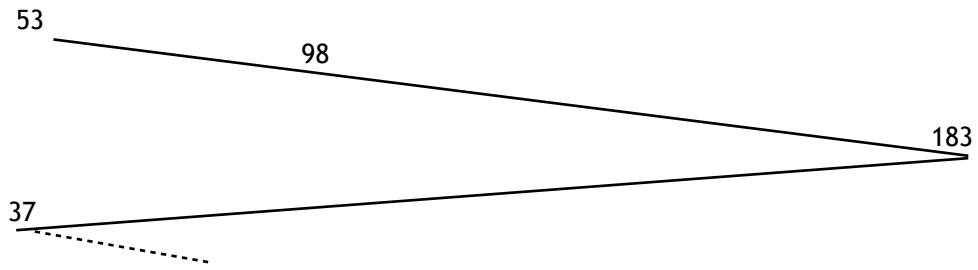
Noi tratteremo quelle politiche che dipendono solo dall'ultima condizione (e che quindi modificano il tempo di seek - ovvero il numero di tracce esaminate).
Nascono così differenti politiche di scheduling.

LCFS (Last Come First Served)
Questa politica non prevede alcun tipo di ordinamento delle richieste, esse vengono evase nell'esatto ordine in cui sono arrivate. L'assenza di un filtraggio rende questa politica particolarmente snella.
Potrebbe sembrare banale, ma è un sistema che funziona per un utilizzo casalingo dove non si hanno richieste incessanti.
Non c'è rischio di starvation.

Esempio
Prendiamo questa serie di richieste, dove ogni numero rappresenta il numero del settore desiderato.

Current	req. 1	req. 2	req. 3	req. 4	req. 5	req. 6	req. 7	req. 8
53	98	183	37	122	14	124	65	67

Rappresentiamo con una linea l'esecuzione:



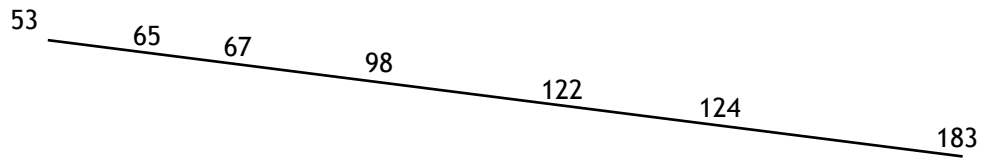
Tempo di seek medio: (98-53) + (183-98) + (183-37) + ... = 640

Il tempo è parecchio alto! Si capisce, da 183 a 37 c'è una lunghissima strada percorsa senza fare nulla!

SSTF (Shortest Seek Time First)
Come facilmente intuibile dal nome, la lista di richieste viene riordinata stando al tempo di seek minore per evadere la prossima richiesta.

Avremo quindi:

Current	req. 1	req. 2	req. 3	req. 4	req. 5	req. 6	req. 7	req. 8
53	65	67	98	122	124	183	37	14



Tempo di seek medio: (65-53) + (67-65) + (98-67) + ... = 236

Abbiamo un tempo decisamente migliore, ma abbiamo oltremodo un rischio: se ricevessimo per un lungo periodo richieste in un certo range di settori, avremmo starvation.

È inoltre da considerarsi il tempo di ordinamento delle richieste: conviene quando si ha un grosso computer con moltissime richieste (magari che richiedono parecchio tempo).

Scan (ascensore) e C-Scan
Cambiando totalmente ragionamento, potremmo pensare di far muovere continuamente la testina dall'esterno all'interno del disco operando le letture e le scritture quando si passa sui settori desiderati. Chiaramente le richieste vengono riordinate in base alla loro posizione rispetto al movimento della testina.
Questo problema risolve la starvation del sistema SSTF, ma presenta altri due problemi:
1) In ogni caso, anche se le richieste sono concentrate al centro, la testina si muove dal primo all'ultimo settore di quella porzione di disco (quindi compie movimenti inutili)
2) Rischiamo di incappare nel fenomeno di **accumulo**. La testina, muovendosi (supponiamo) verso destra, lascerà dietro di se tutte richieste evase. Dietro ad essa quindi si accumuleranno molte nuove richieste da evadere, ma la testina invece proseguendo verso destra evaderà quelle dall'altro lato! Si genera quindi un accumulo di richieste che verrà smaltito solo quando la testina tornerà al punto iniziale (a sinistra)

La **C-Scan** risolve l'accumulo: una volta effettuate le prime n richieste, la testina si muove all'indietro molto velocemente (senza fare operazioni) e ricomincia da capo. La C sta infatti per "circular".
Il tempo medio di seek è di 360 tracce.

Look e C-Look
Per risolvere il primo problema, si introduce una nuova funzionalità: la testina si muoverà velocemente fino alla prima richiesta, poi proseguirà lentamente, per poi tornare ad essere veloce quando le richieste in quella zona sono esaurite.
Esiste anche la variante circolare **C-Look** (come sopra).

L'ordinamento delle richieste non è comunque totalmente libero: è necessario tenere conto che certe operazioni sono invertibili, basti pensare al journaling (bisogna necessariamente prima scrivere sul file di log!).w

I difetti fisici del disco

Anche se l'utente non lo sa, all'interno del suo disco una parte dei **blocchi sono inutilizzabili**. Questo accade poiché il procedimento di magnetizzazione del disco è piuttosto delicato e quindi rischia di fallire in rari casi.

Durante la formattazione logica, il sistema operativo può rendersi conto del fatto che i blocchi sono danneggiati e quindi evitarli.

In un file system FAT ed un hard disk con interfaccia **IDE** (per esempio), semplicemente, avremo dei collegamenti "morti" nella tabella.
È comunque possibile che durante la vita del disco, altri blocchi si corrompano.
Nel caso di una operazione di scrittura, noteremo che il blocco non è idoneo e semplicemente verrà cercato un altro blocco. Nel caso di lettura però vorremmo salvare il dato in qualche maniera.

Consideriamo ora una interfaccia **SCASI**: il controller sarà in grado, in caso di blocco corrotto, di effettuare dei tentativi di recupero.
Inoltre, per ogni traccia vengono riservati alcuni blocchi in più che vengono (in caso di necessità) rimappati sui blocchi ritenuti non funzionanti.

Riassunto da: Enrico Mensa

Sistemi Operativi
Bootstrap, RAID ed ultimi argomenti

Capitolo 10

Indice degli argomenti

Il bootstrap

Il bootstrap è quella situazione in cui un processo lancia un processo che lancia altri processi a sua volta.
Il primo bootstrap che viene effettuato dalla macchina è a carico del **BIOS** (basic input output system), il quale lancia init, il processo padre di tutti gli altri.

Il BIOS è un piccolo chip installato sulla ROM, una memoria statica. La CPU sa dove si trova la ROM e riesce così a rintracciare il BIOS.
Il BIOS di occupa di due cose:
(1) Esegue una serie di operazioni (dette POST - *Power On Self Test*) che permettono alla macchina di riconoscere le periferiche base.
(2) Una volta letto il boot loader primario, avvia il boot loader secondario.

Il boot può essere **hard** oppure **soft**. L'hard reboot viene effettuato quando la macchina è stata spenta in maniera inaspettata (o comunque senza effettuare le operazioni di shutdown). Il soft boot invece è effettuato in seguito ad un reboot manuale oppure ad una ibernazione.

La periferica di boot
Al giorno d'oggi è facilmente possibile avviare un sistema operativo da chiavetta USB piuttosto che una installazione da CD. Come è possibile?
All'interno del BIOS è possibile impostare un'ordine di priorità delle periferiche, affinché il boot loader vada a cercare il lancio del sistema operativo "in ordine": in assenza della prima periferica, passa alla seconda e così via.

Il boot loader secondario
Il boot loader secondario ha una posizione fissa, si trova nell'MBR (Master Boot Record). La sua staticità dipende dal fatto che ogni sistema operativo ed ogni macchina devono essere in grado di riconoscerlo.
L'utente ha però la possibilità di modificare il contenuto dell'MBR. Infatti, possiamo installare differenti sistemi operativi sulla stessa macchina utilizzando un boot loader adatto.
Ogni boot loader ha un numero finito di file system riconoscibili, inoltre può fornire un menù di selezione piuttosto che il lancio diretto del sistema operativo.
Tra i più conosciuti boot loader secondari troviamo:
- NTLDR: proprietario Windows, riconosce file system windows e basta.
- LILO: vecchio boot loader usato in ambiente unix, supporta 16 file system differenti
- GRUB: versione rinnovata di LILO.

Vediamo più nel dettaglio i loro passi base:

NTLDR
- Accede al FS
- Verifica se c'è salvata una immagine del sistema (ibernazione), se non la trova, esegue boot.ini

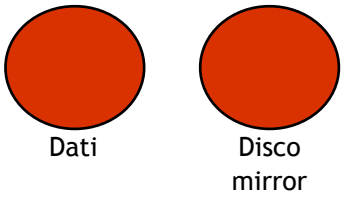
boot.ini è un file di configurazione del tipo:
[boot loader]
    timeout = 30
    default = multi(0).disc(0)...
[...]

GRUB
- Grub stage 1: legge i vari sistemi operativi installati (MBR)
- Grub stage 1.5: i 30k successivi all'MBR danno avvio allo stage 2, che si trova nel disco
- Grub stage 2: menù di selezione del sistema operativo

RAID

Le tecniche RAID (Random array of Indipendent Disk) mirano a salvaguardare il salvataggio dei propri dati. Sono dei sistemi di sicurezza che constano di una batteria di hard disk che "contano" come fossero uno solo.
Tramite il RAID vengono salvate più informazioni oltre ai dati.
I punti forti del RAID sono la **ridondanza** e l'**affidabilità.**
I sistemi RAID vengono spesso utilizzati con NAS (Network Attached Storage).
Vi sono due macro tipologie di RAID:
1) Mirroring
2) Striping

Mirroring
Il mirroring è un sistema che copia "a specchio" su un hard disk parallelo il nostro disco dati. Avremo quindi:

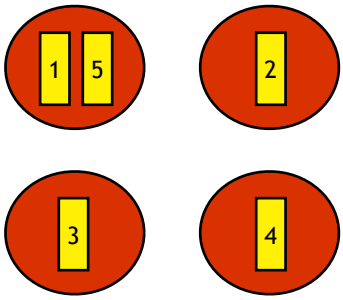


È possibile leggere contemporaneamente dai due dischi, questo raddoppia la velocità di lettura.
La scrittura, però, deve essere effettuata su entrambi i dischi quindi la sua velocità rimane costante.
La memoria utilizzabile è 1/2 rispetto a quella fisica.

Alla rottura di un disco è saggio provvedere alla sostituzione del secondo, poiché i dischi hanno spesso "vita simile".

Striping
Di filosofia totalmente diversa è lo striping: i dati vengono "spalmati" (ovvero divisi) sui diversi dischi. È possibile dividere i dati per **blocchi** oppure per **bytes**.

Divisione in blocchi

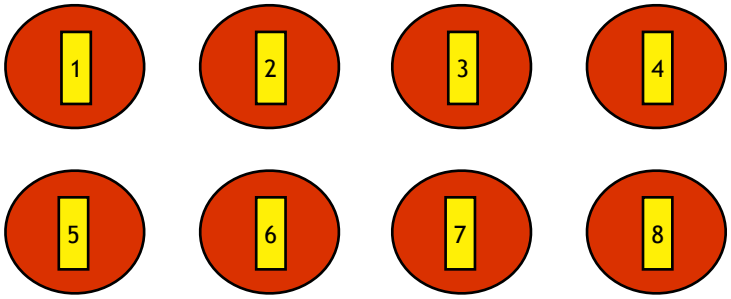


I blocchi del file (da 1 a 5 nell'immagine) vengono divisi sui dischi.

La lettura e anche la scrittura possono avvenire parallelamente (la scrittura per quanto concerne singoli blocchi!).

Non c'è ridondanza, e la rottura di un disco porta alla totale perdita dei dati. La tecnica di striping, infatti, non viene mai usata "a se stante".

Divisione in bytes



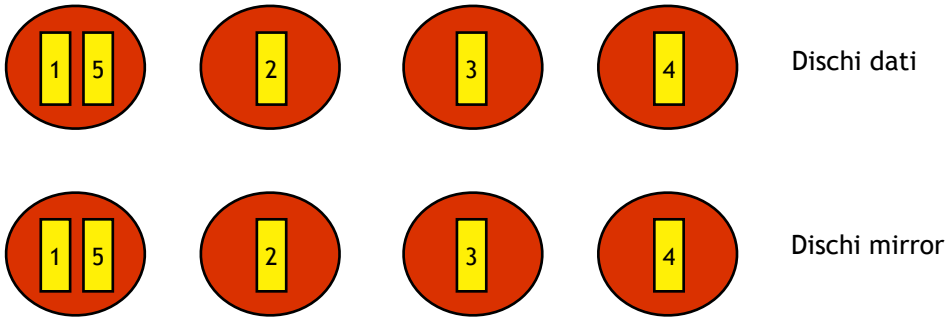
Tecnica del tutto simile alla divisione in blocchi, ma questa volta ogni singolo byte viene "spalmato" sugli otto dischi minimi necessari, bit per bit.
Ovvero gli otto quadrati dell'immagine sono un byte solo suddiviso in bit.

Entrambi i sistemi non portano alla salvezza dei dati, però. Il sistema di striping viene quindi correlato di codici/tecniche diverse per poter recuperare i dati nel caso in cui un disco si rompa. A seconda dell'accostamento del codice di errore, abbiamo RAID di "livello" differente. Vediamoli.

I livelli dei RAID
- Livello 0: Striping a livello di blocchi senza ridondanza
- Livello 1: Mirroring
- Livello 2: ECC (*error correcting codes*)
- Livello 3: Striping con bit di parità intercalati
- Livello 4: Striping con blocchi di parità intercalati
- Livello 5: Striping con blocchi intercalati di parità distribuiti
- Livello 6: Schemi di ridondanza P + Q
- Livello 0+1: combinazione del livello 0 con il livello 1.

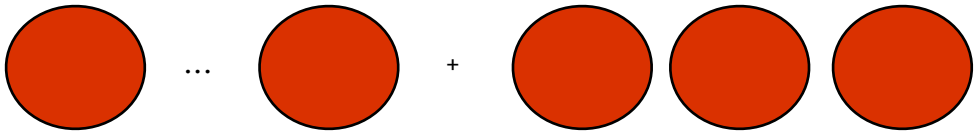
Vediamone alcuni nel dettaglio

Livello 0 +1



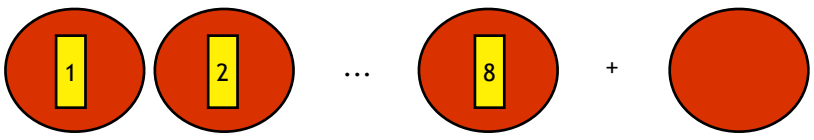
Le prestazioni in lettura/scrittura sono migliorate per via della moltitudine di dischi. Piuttosto dispendioso come sistema.

Livello 2: ECC



Abbiamo tre dischi che hanno un codice di correzione in grado di risolvere eventuali problemi dei dischi dati.

Livello 3: Striping con bit di parità



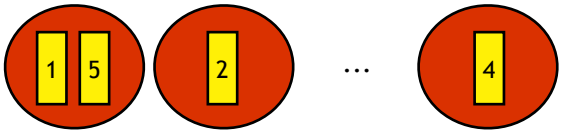
Questo metodo è del tutto simile al livello 2 ma l'ultimo disco ha un bit di priorità che controlla un byte nei dischi precedenti (ovvero ogni byte ha un bit di parità riservato nell'ultimo disco).

Livello 4: Striping con blocchi di parità intercalati

Dischi dati divisi con striping a blocchi Un disco di correzione con blocco di parità

Questo sistema riserva ai blocco dell'ultimo disco la funzione d blocco di parità. Per esempio, il primo blocco di parità controllerà la parità di tutti i primi blocchi dei dischi dati.

Livello 5: Striping con blocchi intercalati di parità distribuiti



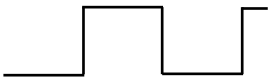
Dischi dati divisi con striping a blocchi, con blocchi di parità insieme a quelli dati.

Questa tecnica è identica alla precedente ma l'ultimo disco è a sua volta distribuito all'interno all'interno dei dischi dati.

La memoria terziaria

La memoria terziaria è sostanzialmente divisibile in due categorie:
1) Memorie di tipo ottico
2) Memorie di tipo flash

Le memorie di tipo ottico
I supporti ottici nascono nel 1992, creati dalla Phillips.
I supporti sono alti 12 cm, e sono costituiti da uno strato di plastica, uno di lamina e uno di smalto.
I dati sui supporti ottici vengono salvati in maniera ellittica, divisi in tracce, e scritti tramite un laser (maggiore è la frequenza del laser, maggiori sono i dati scrivibili sul supporto).
La codifica, più precisamente, è questa:



Ogni linea verticale è un uno,
ogni linea orizzontale è una serie di zero.

Ve ne sono di tre generazioni.

1° generazione:
Compact Disc (CD), scritti con laser a frequenza infrarossa

2° generazione:
DVD, DviX, scritti con laser a frequenza rossa (più alta di prima)

1° generazione:
Blue Ray, scritti con laser a frequenza blu (ancora più alta)

Le memorie di tipo flash
Le memorie di tipo flash sono di principalmente di due tipi. Abbiamo memorie **NOR** (accesso diretto - parecchio costose, usate per la costruzione del BIOS) e memorie **NAND** (accesso sequenziale, piuttosto comune).

Le memorie flash hanno le seguenti peculiarità:
1 - Non vi sono parti meccaniche (è possibile muovere il dispositivo durante l'utilizzo)
2 - È possibile sovrascrivere i dati un numero di volte che si aggira intorno ai 10^3 - 10^6

Le memorie flash sono sostanzialmente costituite di due parti: **floating gate** e **control gate**. La distinzione fra "zero" e "uno" viene fatto calcolando la quantità di elettroni contenuti in una singola cella. Se quel numero supera un certo N, allora il bit vale 1, altrimenti vale 0.

Le memorie flash sono facilmente leggibili, anche se non si può dire la stessa cosa per l'operazione di scrittura. Tali memorie, infatti, hanno difficoltà nel portare una singola cella da 0 a 1. Una memoria flash vuota ha tutti i bit a 1 (esempio: 1111).
La scrittura è possibile solo se il blocco (serie di bit) di output avrà una quantità di zeri maggiore o uguale a quella del blocco prima dell'operazione di scrittura. Portare un bit da 0 a 1 significa cancellarlo, e questa operazione è possibile solamente su "grandi" quantità di bit tutti in una volta, e non singolarmente. Pertanto, 1111 --> 1101 è una operazione legale, ma 0101 --> 0100 no! Per ovviare a questo problema, nel momento della scrittura di cerca un blocco costituito di soli uni, e viene rimappato sul blocco che si voleva utilizzare. Il vecchio blocco verrà poi "riciclato" prima o poi, tramite una cancellazione "a blocchi".
Esempi di file system per le memorie flash sono **JFF2** (journaling flash filesystem) e **YAFFS** (yet another flash filesystem).

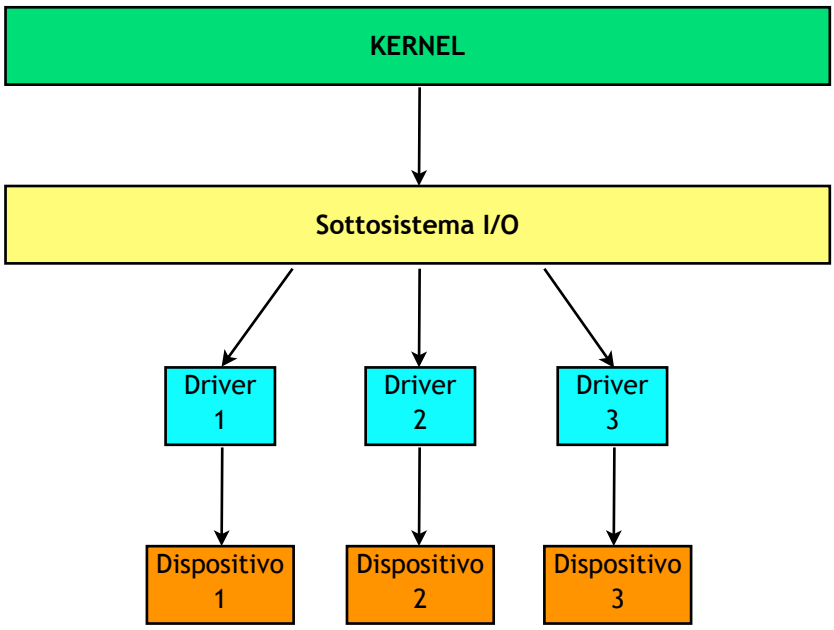
Sottosistema di I/O

Esiste un ulteriore livello che si frappone tra i driver ed il kernel. Il sottosistema di I/O si occupa della gestione dei vari devices (che sono molti!) connessi al computer. I devices di distinguono per:

- **Tipologia di accesso**
  - sequenziale (*esempio: modem*)
  - diretto (*esempio: hard disk*)
- **Tipologie di trasferimento**
  - a caratteri (stream di byte) (*esempio: tastiera*)
  - a blocchi (*esempio: hard disk)*
- **Sincroni/Asincroni** (rispetto alla CPU)
- **Condivisibili/Utilizzabili in mutua esclusione**

I driver sono molteplici per lo stesso device poiché non esiste una convenzione con cui i sistemi operativi possano comunicare con un driver. Questo significa che ogni sistema operativo deve avere i suoi driver per utilizzare un certo dispositivo.

Architettura generale



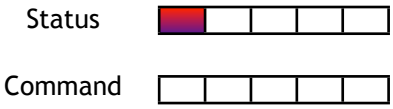
Tutto il funzionamento di questo sistema si basa sul concetto di interrupt, senza il quale i devices non potrebbero "comunicare" con la CPU.
Ma gli interrupt non sono l'unico sistema per implementare questa cosa.

L'alternativa agli interrupt - i protocolli handshaking
Stando a quanto sappiamo, quando un processo chiede una lettura/scrittura, viene messo in waiting affinché la CPU possa eseguire qualcos'altro nel tempo di latenza dell'operazione. Al termine dell'operazione, il device lancia un interrupt alla CPU per risvegliare il processo.

Supponiamo invece che ora la CPU stia in attesa della lettura/scrittura.

Abbiamo in questo caso un sistema **handshaking**. Vediamone uno in particolare, il **PIO** (*Program I/O*).

Questo sistema viene gestito tramite due registri,



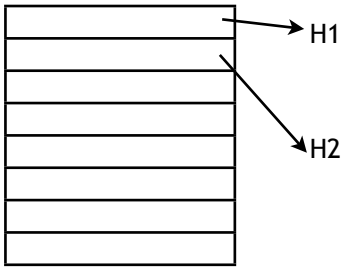
In status abbiamo il **bit busy** che ci dice se il controller è busy oppure no (ovvero se una operazione è già in corso). Il registro command avrà bit dedicati per ogni comando.
L'algoritmo eseguito è il seguente:

Sistema operativo: "Leggi il bit busy finché non è libero (vale 0)"
Sistema operativo: "Setta a 1 il bit *write* di command"
Sistema operativo: "Informo il controller che l'operazione è correttamente specificata; setta a 1 il bit *command_ready* del registro command"
Controller: "Setto busy a 1"
Controller: /*esegue operazione*/
Controller: "Riporto i bit *busy* e *command_ready* a 0"

L'attesa degli altri processi viene gestita in due maniere differenti:
- **Attesa attiva**: si esegue la prima istruzione (while).
- **Polling**: si effettua un controllo sul bit busy ogni quanto di tempo, per evitare un'attesa attiva costante.

Questo sistema non viene comunque utilizzato poiché inefficiente rispetto agli interrupt.

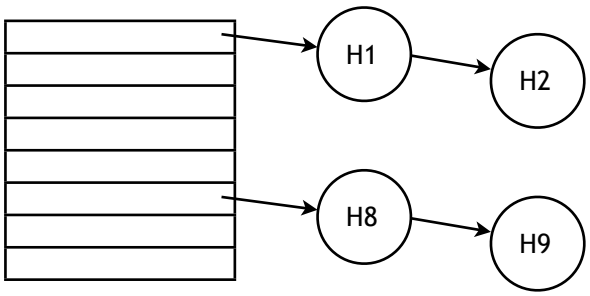
Il vettore degli interrupt



Come sappiamo il vettore degli interrupt è un array contenente l'indirizzo delle prime istruzioni di ogni handler (in grado di gestire un dato interrupt).

Abbiamo sempre visto questo array come statico, ma in realtà esso viene generato all'avvio nella fase di bootstrap (quando vengono esaminati i devices).

Talvolta capita che la quantità di handler sia troppo vasta per poter essere tenuta in un array solo. Viene creata quindi una sorta di hash table degli interrupt:



Vi sono pertanto delle liste di handler che sono consultabili in caso di bisogno.

Agli interrupt vengono inoltre assegnate, talvolta, delle priorità. Un interrupt di priorità A non potrà essere interrotto da uno di priorità B se B < A. Due priorità quasi sempre implementate sono gli interrupt **mascherabili** e **non mascherabili**.