Appunti di Sistemi Operativi

Tommaso Macchioni

2019

Premessa

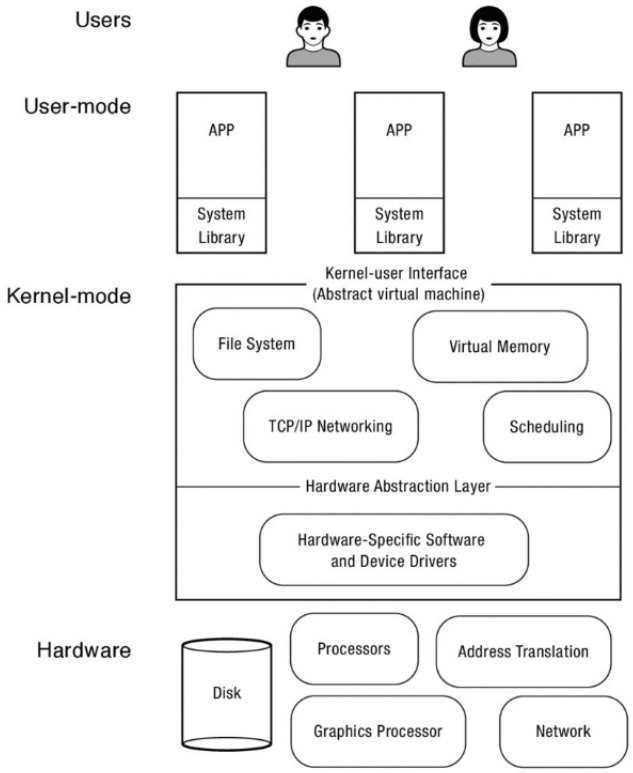
Questi appunti sono stati realizzati integrando le nozioni apprese durante il corso di "*Sistemi Operativi e Laboratorio*" tenuto dal prof. Maurizio Angelo Bonuccelli (*Universita di Pisa*, 2018/19) con i contenuti presenti in "*Operating Systems: Principles and Practice*" di Thomas Anderson e Michael Dahlin.

Indice

1 Introduzione

1.1 Cos'è un Sistema Operativo

Un sistema operativo è un software per gestire le risorse di un computer per i propri utenti e le proprie applicazioni.



Se il SO si attaccasse direttamente all'hardware e dovessi cambiare parti dell'hardware, dovrei cambiare il SO. Quindi nel mezzo ai due viene messo uno strato software che si chiama *Hardware Abstraction Layer*.

Quest'ultimo è un software specifico che viene venduto con l'hardware che rende uniforme tutti gli hardware diversi dal punto di vista del SO. Il SO lo chiameremo anche *Kernel*. Da un punto di vista *linguistico* il SO è effettivamente un software ad alto livello mentre dal punto di vista *funzionale* viene invece differenziato dalle altre applicazioni ad alto livello. Perché il SO è fatto da gestione della CPU, TCP/IP Networking ecc.

Il SO inoltre ci mette a disposizione delle *librerie di sistema*(e.g. fonts dei caratteri e le icone) ma che non fanno parte del SO vero e proprio.

Il SO svolge vari ruoli:

- **Arbitro:** perché gestisce le risorse condivise tra diverse applicazioni attive nella stessa macchina fisica. Ad esempio, un SO può fermare un programma e farne partire un altro, decide a quale applicazione dare una certa risorsa, isola ogni applicazione in modo da evitare che un bug interferisca con le altre e protegge dai virus.
- **Illusionista:** perché da l'illusione ad ogni processo (e quindi ad ogni utente che scrive un programma) di avere a disposizione tutto il computer (memoria e processore) anche se in realtà non ce l'ha.
- **Colla:** perché fa da interfaccia comune a tutte le applicazioni che utilizziamo e da degli strumenti in modo da fare le applicazione più facilmente.

1.2 Quali caratteristiche deve avere

Un SO deve essere:

- **Affidabile e disponibile:** è la caratteristica più importante. Un sistema deve fare ciò per cui è stato progettato e deve essere preciso nella risposta: o mi rispondi correttamente oppure mi restituisci errore e non una via di mezzo (questo perché inizialmente i computer venivano utilizzati per operazioni militari). Disponibile perché tendenzialmente deve funzionare sempre.
- **Sicuro:** perché non deve essere compromesso da attacchi maliziosi e la *privacy* è un aspetto della sicurezza: i dati memorizzati nel computer devono essere accessibili solo da utenti autorizzati.
- **Portatile:** in modo tale che possano essere scritte le applicazioni indipendentemente dall'hardware e dal SO che sto utilizzando. Lo stesso SO devo poterlo spostare un hardware all'altro. Deve essere progettato per supportare applicazioni che non sono ancora state scritte e avviarle in hardware che non sono ancora stati sviluppati.
- **Performante:** deve rispondere il più veloce possibile, eseguire quanti più processi in un'unità di tempo. La percentuale di tempo in cui il processore esegue le operazione del SO (*overhead*) deve essere più piccola possibile: fare poco e farlo bene.

1.3 Passato, Presente e Futuro

1.3.1 I primi Sistemi Operativi

I primi computer erano spesso grandi quanto un'intera stanza, costavano milioni di dollari e venivano usato da una sola persona alla volta. Veniva elaborato l'unico programma che veniva immesso (scritto in *assembler*). Il SO era praticamente inesistente, era sostanzialmente un insieme di librerie: dei programmi che ne agevolavano l'uso.

1.3.2 Sistemi Operativi Multi-Utenti

L'utente passava gran parte del tempo ad immettere il programma e i dati e il computer spendeva un mucchio di tempo a stampare i risultati. Ci siamo accorti che potevamo migliorare la situazione facendo lavorare il computer su una coda di tasks: la stampante stampava i dati del programma che era eseguito precedentemente ed il lettore di schede leggeva i programmi e i dati che venivano eseguito successivamente (*Batch Operating System).* Nei sistemi Batch l'input del programma e dei dati veniva messo sul disco, passando per la CPU, dopodiché quando era il momento di eseguirlo veniva richiamato dal disco e messo in esecuzione e nel frattempo legge

altri dati. I risultati venivano scritti sul disco ed infine il disco li mandava alla stampante. I SO Batch sono stati poi estesi per eseguire multiple applicazioni alla volta: *Multitasking*. L'idea questa volta è di non avere un unico programma attivo ma molti e quando uno di loro attende una risposta dall'esterno, viene messo in pausa e viene ripreso un programma interrotto precedentemente e così via. Il problema del Multitasking è che è poco equo: se un programma ha bisogno di fare poco input/output rischia di monopolizzare il computer.

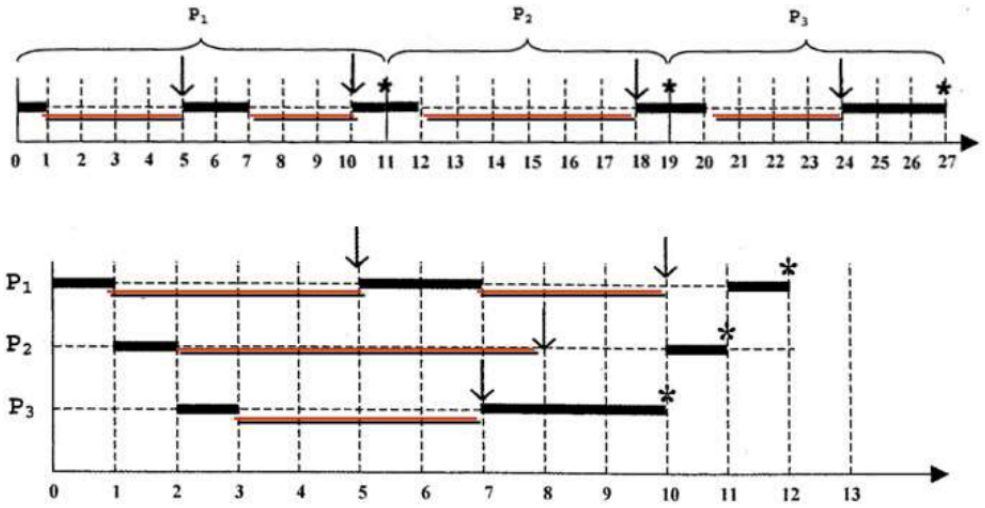


Figura 1: Singletasking vs Multitasking

1.3.3 Sistemi Operativi Time-Sharing

Lo sono Windows, MacOS e Linux. Sono progettati per supportare uso interattivo del computer. Un programma è in esecuzione al massimo per una certa quantità di tempo detta *quanto di tempo*. Se eventualmente viene digitato qualcosa da tastiera o avviene qualsiasi altra interruzione, viene messo in fondo ad una lista di programmi pronti a ripartire e il SO gestisce l'interruzione.

1.3.4 Sistemi Operativi Moderni e Futuri

Oggigiorno, abbiamo una vasta diversità di computer con differenti SO. Il futuro invece sta andando verso un'informatica divisa in tre parti: *tiny OS*, processori minuscoli con processore da 8bit e poca memoria; *SO "normali"*, quelli usati oggigiorno come desktop, laptop ecc.; *Large-Scale Data Centers*, che sono i centri di elaborazioni usati ad esempio per il cloud.

1.4 Un richiamo sull'architettura utilizzata

Un'architettura tipica avrà:

- Una o più CPU:
  - Registri generali,
  - Registri di stato e di controllo: *Program Counter*(PC), *Stack Pointer*(SP), *Program Status Register*(PS).
- Una memoria principale (RAM)
- Un insieme di periferiche: mouse, disco, tastiera ecc.
- Interruzioni e gestione di queste
- Direct Memory Access (alcune periferiche possono scrivere direttamente in memoria)

Il *Program Status Register* contiene varie informazioni: bit di abilitazione/disabilitazione delle interruzioni, 1 bit che dice chi è in esecuzione tra utente e kernel, codici condizione che servono per stabilire che cosa è successo nell'ultima istruzione (divisione per zero, overflow, riporto ecc.) ecc.

Inoltre considereremo il *ciclo fetch-execute*:

- se arriva un'interruzione e queste sono abilitate: prima si termina l'istruzione assembler corrente e dopo si gestisce l'interruzione.
- altrimenti:
  - si carica l'istruzione di indirizzo PC,
  - si esegue l'istruzione,
  - $PC = PC + 4$ (perché si assume che le istruzioni occupino 4 bytes (32bit)).

2 Astrazione del Kernel

Una delle sfide che il nucleo deve riuscire a vincere riguarda la *protezione*: deve isolare malfunzionamenti (bug o malware) da parte di programmi o utenti in modo da non corrompere altri programmi o il SO stesso. Ad esempio i programmi degli utenti devono essere eseguiti con privilegi limitati.

2.1 Astrazione dei processi

Un *processo* è un'astrazione usata dal SO e rappresenta l'attività di esecuzione di un programma e un programma può avere più processi (e.g. ogni scheda di un browser è un processo differente). Anche il SO è un programma che genera più processi.

Abbiamo due tipi di processi:

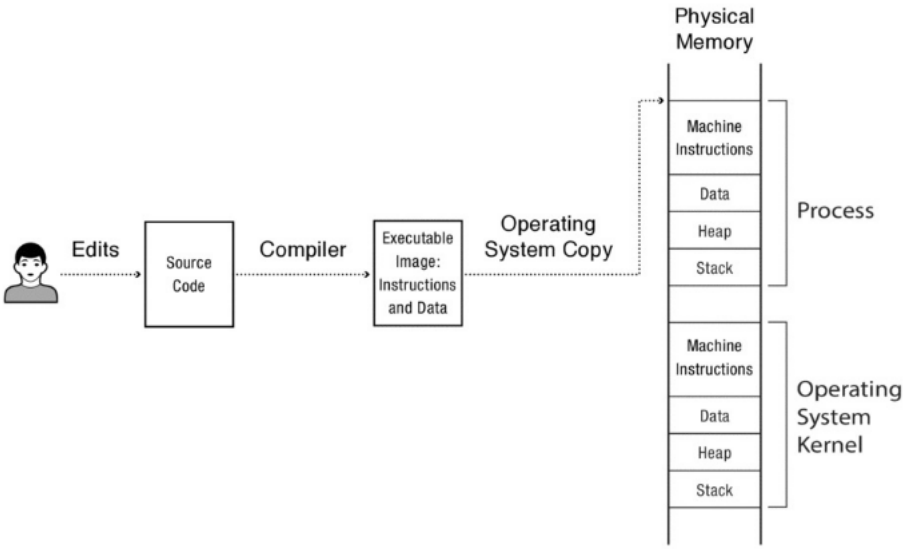
- processi utente,
- processi del kernel

Un programmatore scrive un programma in un linguaggio di alto livello. Un compilatore converte il codice del programma in una sequenza di istruzioni macchina e la memorizza in un file chiamata l'*immagine eseguibile*. Il compilatore inoltre definisce ogni dato statico di cui il programma ha bisogno, insieme ai valori iniziare, e l'include nell'immagine eseguibile.

Per eseguire il programma, il SO copia le istruzioni e i dati dall'immagine eseguibile nella memoria fisica (RAM) e fa spazio in memoria per:

- lo *Stack*: per mantenere lo stato delle variabili locali durante le chiamate di procedure,
- lo *Heap*: per ogni struttura allocata dinamicamente che il programma potrebbe aver bisogno.

Ovviamente il SO deve già stato caricato in memoria col proprio Stack e Heap.



Un processo è un'*istanza* di un programma con diritti limitati, come un oggetto è un'istanza di una classe nel paradigma object-oriented. Ogni programma può avere zero, uno o più processi che lo eseguono. Per ogni istanza di un programma c'è un processo con la propria copia del programma in memoria.

Il SO tiene traccia dei vari processi nel computer tramite l'utilizzo di una struttura dati chiamata *Process Control Block* (PCB). Il PCB memorizza tutte le informazioni di cui il SO ha bisogno per un particolare processo: dov'è memorizzato in memoria, dove risiede la sua immagine eseguibile nel disco, quale utente ha chiesto di eseguirlo, quali privilegi ha il processo ed altro. Ad ogni processo è assegnato un numero che identifica proprio l'indice nella tabella che contiene tutti i PCB.

2.2 Dual-Mode Operation

Per garantire che un processo non danneggi altri processi o il SO stesso, è stato implementato un metodo, chiamato *Dual-Mode operation*, rappresentato da 1 bit nel *processor status register* che identifica la modalità corrente del processore:

- In **User-mode** il processore controlla ogni istruzione prima di eseguirla in modo da verificare che è una operazione permessa al processo,
- In **Kernel-mode** il SO può eseguire qualsiasi operazione in qualsiasi posto (memoria, disco, periferiche ecc.).

Affinché il SO possa proteggere le applicazioni e gli utenti l'uno con l'altro l'hardware deve supportare tre cose:

- **Istruzioni privilegiate.** Tutte le potenziali istruzioni non sicure sono proibite quando è in esecuzione la modalità utente.
- **Protezione della memoria.** Ogni accesso alle zone di memoria fuori da quella riservata al processo deve essere proibita.
- **Interruzioni da timer.** Indipendentemente da ciò che fa un processo, il kernel deve essere in grado periodicamente di riottenere il controllo.

2.2.1 Istruzioni privilegiate

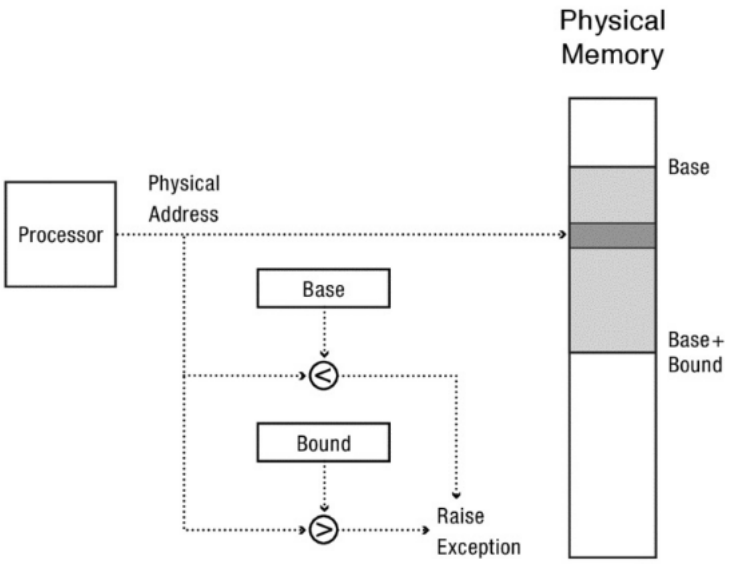
L'isolazione di un processo è possibile sole se c'è un modo per limitare che i programmi in esecuzione in modalità utente possano cambiare il proprio livello di privilegio. Vedremo che un processo può indirettamente cambiare il proprio livello di privilegio eseguendo un'istruzione speciale, chiamata *system call*, per trasferire il controllo al kernel ad una precisa locazione definita dal SO. Alcune istruzioni inoltre, possono essere utilizzate solo dal kernel. Queste istruzioni sono chiamate *istruzioni privilegiate*. In sostanza un programma può eseguire solo un sottoinsieme di tutte le istruzioni possibili mentre il SO, eseguito in kernel-mode, deve avere pieno possesso dell'hardware. Se un processo prova ad eseguire istruzioni privilegiate solitamente viene killato dal kernel e si avvisa l'utente.

2.2.2 Protezione della memoria

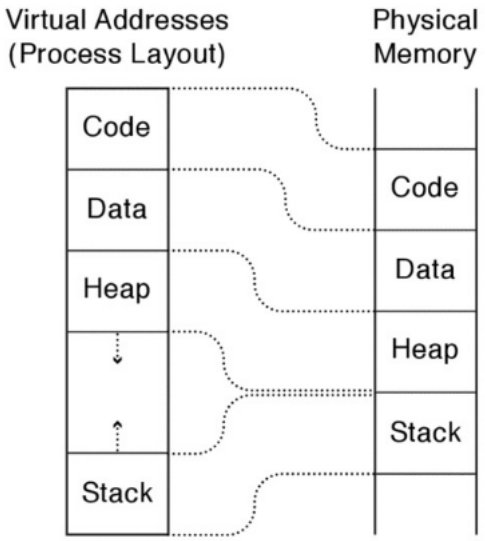
Per prevenire l'accesso, da parte di un programma di un utente, a porzioni di memoria non concesse ci sono vari approcci:

- **Base and bound.** La base specifica l'inizio della regione di memoria del processo mentre il limite (bound) identifica la fine. I registri che memorizzano questi dati possono essere modificati solo da istruzioni privilegiate, ovvero dal SO eseguito in kernel-mode. Ogni volta che il processore fetcha un'istruzione, controlla che il PC sia tra questi due valori. Se ci rientra viene prelevata altrimenti viene trasferito il controllo al kernel. Ovviamente il kernel è eseguito senza un registro di base e di limite. Quando un programma viene eseguito, il kernel cerca un blocco contiguo di memoria abbastanza grande da memorizzare l'intero programma: i suoi dati, lo heap lo stack.

  L'utilizzo di registri fisici per la base e il limite pone alcuni problemi: si pone un limite sull'ammontare di memoria allocata per un programma ma i programmi hanno porzioni di memoria espandibili: lo Stack e lo Heap, che crescono in zone opposte della regione di memoria allocata. oppure non permette di condividere memoria tra processi diversi ecc.



- **Virtual adresses.** La memoria di ogni processo parte da zero. Ogni processo crede di avere l'intera macchina. L'hardware traduce questi indirizzi virtuali in indirizzi fisici di memoria.



2.2.3 Interruzioni da timer

Se ad esempio un programma entra in un loop infinito, il SO deve essere in grado di riottenere il controllo. Però il SO ha bisogno del processore per eseguire le istruzioni per fermarlo e quindi tutti i computer includono un device chiamato *Hardware Timer* che è in grado di interrompere il processore dopo un *quanto di tempo* (o un certo numero di istruzioni). In un multiprocessore avremo un timer per ogni CPU. Il SO setta e resetta il quanto di tempo per ogni processo. Quando occorre un'interruzione da timer, l'hardware trasferisce il controllo al kernel. Un'interruzione può occorrere non solo perché è scaduto il quanto di tempo ma anche se un dispositivo I/O richiede attenzione o ha finito di lavorare.

2.3 Mode-switch

Vediamo adesso come si può switchare in sicurezza da processi utenti a processi del kernel e vice versa.

2.3.1 Da User a Kernel Mode

Ci sono essenzialmente tre ragioni per il kernel di prendere il controllo da un processo utente: interruzioni, eccezioni del processore e system calls. Le interruzioni sono eventi asincroni e sono causati da eventi esterni invece le eccezioni del processore e le system calls sono eventi sincroni e sono causati dall'esecuzione del processo. Indichiamo col termine *trap* qualsiasi trasferimento sincrono che switcha il controllo dall'utente al kernel.

- **Interruzioni.** Un'*interruzione* è un segnale *asincrono* che segnala al processore che si è presentato qualche evento esterno che ha bisogno di attenzione. Ogni volta che il processore esegue un'istruzione, controlla se è arrivata un'interruzione. Se lo è, completa o ferma qualsiasi istruzione in corso ed invece di fetchare l'istruzione successiva viene salvato lo stato corrente e comincia a eseguire un *handler* speciale nel kernel. Ogni interruzione ha il proprio handler e ci sono molteplici tipi di interruzione: interruzione da timer, interruzione I/O (e.g. ogni volta che si sposta il mouse), interruzione da un altro processore (in caso di multiprocessori).

- **Eccezioni del processore.** Un'*eccezione del processore* è un evento hardware causato da un programma utente e che causa un trasferimento di controllo al kernel. Come per le interruzioni, il processore completa o termina le istruzioni correnti, salva lo stato in esecuzione ed esegue un handler del SO. Ad esempio, un'eccezione del processore può avvenire ogni volta che un processo tenta di eseguire un'istruzione privilegiata o di accedere fuori dalla propria regione di memoria, quando un processo divide un intero per zero ecc. In questi casi il SO semplicemente ferma il processo e ritorna un codice di errore all'utente. Da notare che un'eccezione del processore avviene anche quando si inserisce un breakpoint in un programma perché il kernel rimpiazza quell'istruzione macchina che è memorizzata in memoria con un'istruzione speciale che invoca l'eccezione. Quando poi il programma va in esecuzione e raggiunge quest'istruzione, l'hardware switcha nella modalità kernel e il SO ripristina la vecchia istruzione e la passa al debugger al quale trasferisce il controllo.

- **System calls.** I processi utenti possono trasferire volontariamente il controllo al SO per richiedere che questo esegua delle operazioni al posto loro. Una *system call* è una procedura fornita dal kernel e che può essere chiamata a livello utente. Come nei casi precedenti, l'istruzione trap switcha il processore da User-mode a Kernel-mode e comincia ad eseguire uno handler predefinito. Per proteggere il kernel è essenziale che l'hardware trasferisca il controllo ad un indirizzo predefinito perché i processi utenti non possono saltare in posti arbitrari del kernel.

2.3.2 Da Kernel a User Mode

- **Nuovo processo.** Per iniziare un nuovo processo, il kernel copia il programma nella memoria, imposta il PC (Program Counter) alla prima istruzione del processo, imposta la base dello SP (Stack Pointer) e switcha nella modalità utente.

- **Ritorno dopo un'interruzione, eccezione del processore o system call.** Quando il kernel ha finito di gestire la richiesta, ritorna ad eseguire il processo interrotto reimpostando il PC, i suoi registri e ritornando in modalità utente.

- **Switch tra processi differenti.** In alcuni casi, come per l'interruzione da timer, il kernel switcha tra processi. Siccome il processo deve riprendere l'esecuzione da dove era

rimasto, il kernel, tutte le volte salva lo stato del processo nel suo PCB prima di passare al processo successivo.

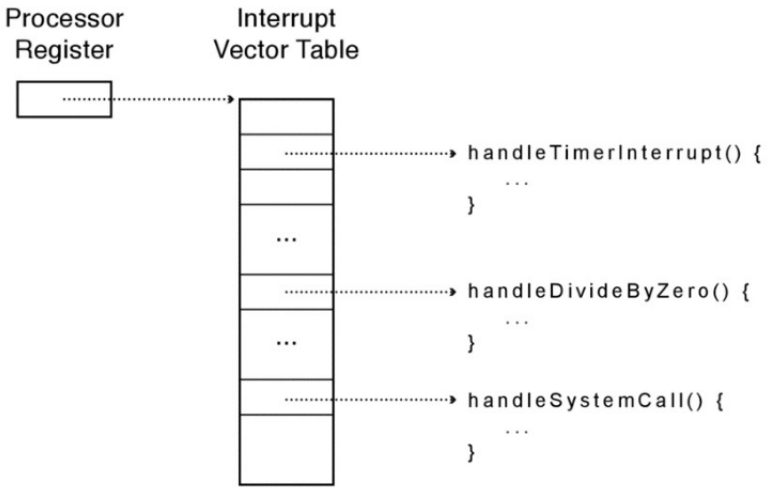
- **Upcall a livello utente.** Molti SO forniscono ai programmi utenti l'abilità di ricevere, asincronicamente, notifiche di eventi.

2.4 Come gestire le interruzioni in modo sicuro

Sia se transitiamo da User-mode a Kernel-mode che vice versa, bisogna assicurarsi che programmi buggati o maliziosi possano corrompere il kernel. Vediamo ora l'hardware e il software necessario per gestire un'interruzione, un'eccezione del processore o una system call, ovvero da utente a kernel.

2.4.1 Interrupt Vector Table

Quando un'interruzione, un'eccezione del processore o una system call occorrono, il SO deve fare azioni differenti a seconda di che evento si tratta. Ad esempio, divide-by-zero exception, una system call per una lettura di un file o un'interruzione da timer. E il salvataggio deve essere fatto in modo atomico: in un solo ciclo di clock. Il processore, per sapere quale codice eseguire, ha un registro speciale che punta ad un'area di memoria del kernel chiamata *Interrupt Vector Table*.



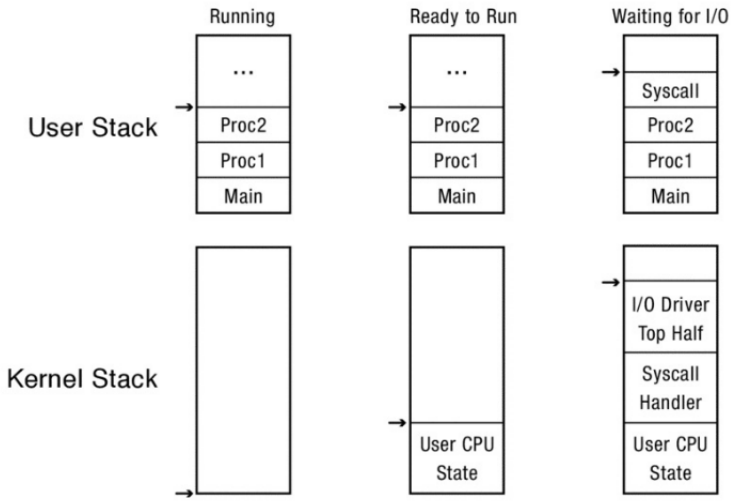
La tabella è in realtà un array di puntatori ed ognuno punta a differenti handler nel kernel. Un *Interrupt Handler* è il termine usato per identificare la procedura chiamata dal kernel a causa di un'interruzione.

Il formato dell'interrupt vector table è specifico per ogni processore. In un x86, ad esempio, le entries 0 - 31 sono per diversi tipi di eccezione del processore (come divide-by-zero); le entries 32-255 sono per diversi tipi di interruzioni (timer, keyboard ecc.) e per convenzione l'entry 64 punta all'handler per le system call.

2.4.2 Interrupt Stack

In molti processori, c'è un registro hardware speciale che punta ad una regione di memoria del kernel chiamato *interrupt stack*. Quando un'interruzione, un'eccezione del processore o una system call causano un cambio di contesto, l'hardware cambia lo Stack Pointer facendolo puntare alla base dell'interrupt stack del kernel. L'hardware automaticamente salva alcuni dei registri del processo interrotto nell'interrupt stack e poi chiama l'handler del kernel. La prima cosa che fa l'handler è quello di salvare i rimanenti registri nello stack e poi comincia a lavorare. Quando poi si ritorna al processo che era stato interrotto, l'handler poppa i registri salvati precedentemente e l'hardware ripristina i registri che aveva salvato ritornando a lavorare al punto in cui era rimasto.

Per memorizzare lo stato del processo non si utilizza lo stack del processo utente ma questo stack speciale perchè lo Stack Pointer del processo utente potrebbe non essere un indirizzo di memoria valido ma l'handler deve continuare a lavorare correttamente e perché in un multiprocessore, gli altri threads che lavorano al solito processo possono modificare la memoria riservata all'utente durante ad esempio una system call e se l'handler del kernel dovesse memorizzare le proprie variabili locali nello stack dell'utente, il programma dell'utente potrebbe modificare l'indirizzo di ritorno del kernel e potenzialmente far eseguire al kernel un codice arbitrario. Molti kernel di SO allocano un interrupt stack per ogni processo utente per una maggior gestione di interruzioni annidate.



2.4.3 Interrupt Masking

Se il processore sta eseguendo ad esempio il salvataggio dei registri prima di eseguire un handler e nel frattempo arrivasse un'interruzione asincrona e la gestissimo, lasceremo il SO in uno stato non valido. Per questo l'hardware prevede un'istruzione privilegiata (altrimenti potrebbe essere disabilitata da malintenzionati) che *disabilità le interruzioni*. Quando vengono riabilitate, ogni interruzione che si è messa in coda durante il mascheramento, viene spedita al processore. Siccome l'hardware ha un buffer limitato, solitamente avremo un buffer per ogni tipo di interruzione.

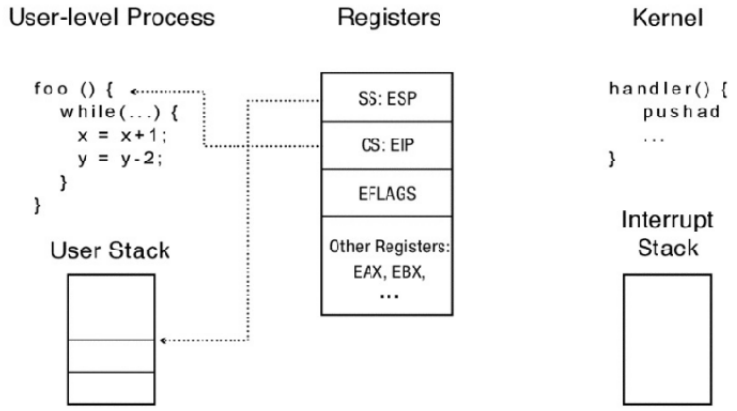
2.5 Esempio architettura x86

Vediamo adesso un esempio di come vengono gestite le interruzioni.

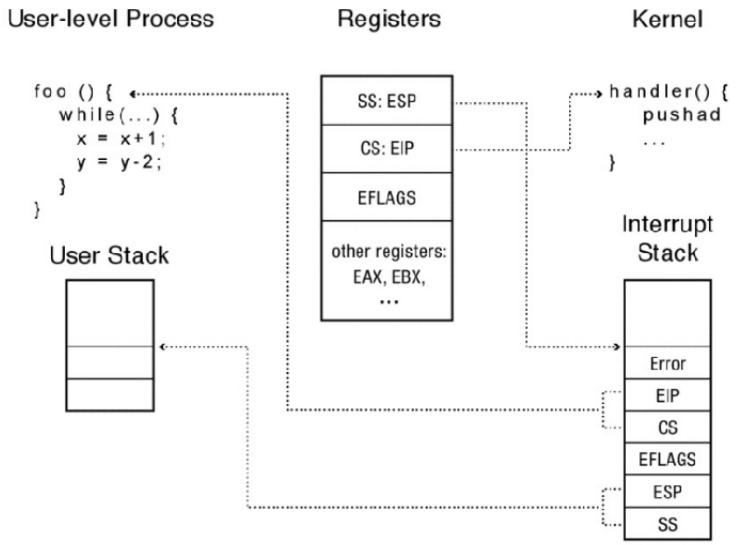
L'architettura x86 è segmentata, quindi il registro di un puntatore contiene due parti: (i) un segmento, una regione di memoria come il codice, i dati o lo stack e (ii) un offset del segmento.

- **SS: ESP.** *SS* indica la base dello Stack, *ESP* indica l'offset.
- **CS: EIP.** *CS* indica la base del segmento del codice, *EIP* indica l'offset. Il livello di privilegio corrente è memorizzato nei bit più bassi del registro CS.
- **Other Registers.** Sono i registri generali.

Quando un processo utente è in esecuzione, lo stato è il seguente:



Quando occorre un'eccezione del processore o una system call, l'hardware salva una piccola quantità dello stato del thread nel kernel interrupt stack e modifica i registri:



1. **Mask interrupts.** L'hardware inizia mascherando ogni interruzione mentre il processore switcha tra User-mode a Kernel-mode.
2. **Save three key values.** L'hardware salva il valore dello Stack Pointer (i registri ESP e SS), i flag dell'esecuzione (il registro EFLAGS), e il Program Counter (i registri EIP e CS) all'interno di registri hardware temporanei.

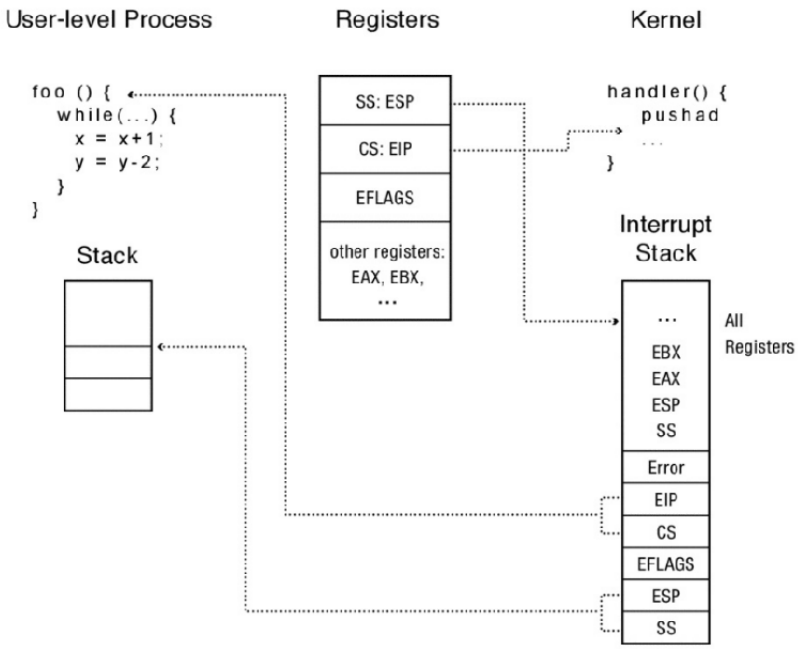
3. **Switch onto the kernel interrupt stack.** L'hardware poi modifica i puntatori SS/ESP alla base del kernel interrupt stack come specifiato in un registro hardware speciale.

4. **Push the three key values onto the new stack.** Poi, l'hardware memorizza i valori salvati internamente, nello stack.

5. **Optionally save an error code.** Alcuni tipi di eccezioni, come un page fault, generano un codice di errore al fine di dare più informazione circa l'evento; per queste eccezioni, l'hardware pusha questo codice in testa allo stack. Per altri tipi di eventi, l'handler dell'interruzione pusha un valore fittizio nello stack così che il formato dello stack sia identico in entrambi i casi.

6. **Invoke the interrupt handler.** In fine, l'hardware modifica il puntatore CS/EIP all'indirizzo della procedura dell'handler. Un registro speciale nel processore contiene la locazione dell'interrupt vector table nella memoria del kernel. Questo registro può essere modificato solo dal kernel. Il tipo dell'interruzione è mappato ad un indice in questo array e il CS/EIP è settato col valore di questo indice.

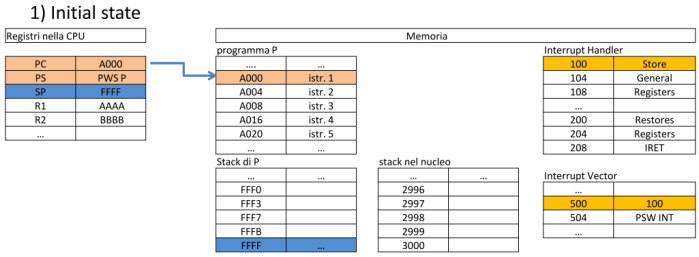
Ora parte il software handler. L'handler deve prima salvare il resto dello stato del processo interrotto prima che li sovrascrivi (i Registri Generali). L'handler pusha il resto dei registri, incluso lo Stack Pointer corrente (quello del kernel), nello stack (`pushad`).



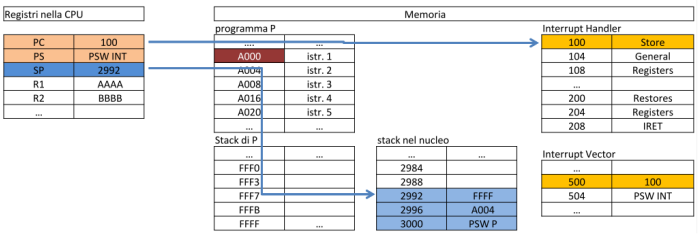
Quando l'handler ha terminato, può ripristinare il processo interrotto. Per far questo, l'handler poppa i registri salvati nello stack. Questo rimemorizza tutti i registri eccetto i flag di esecuzione, il Program Counter e lo Stack Pointer (`popad`. In fine l'handler rimemorizza i puntatori del segmento di codice, il program counter, i flag di esecuzione, il segmento dello stack e lo stack pointer dal kernel interrupt stack. Questo fa si che venga ripristinato lo stato del processo interrotto e così può continuare.

2.5.1 Altro esempio

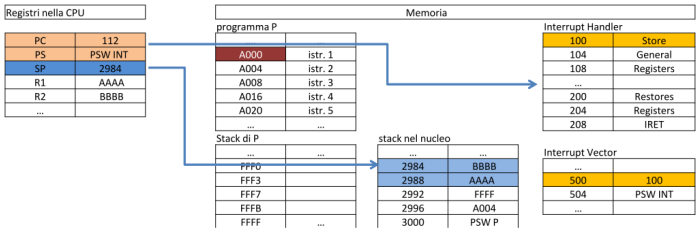
1) Initial state



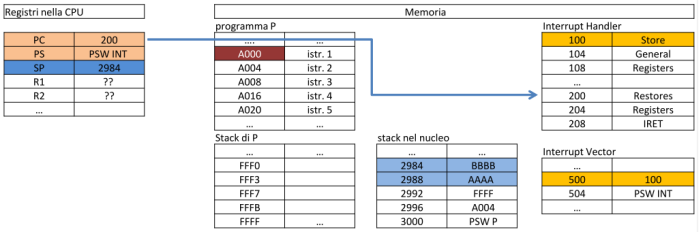
2) Interrupt recognized after instruction A000 (PC incremented to A004)



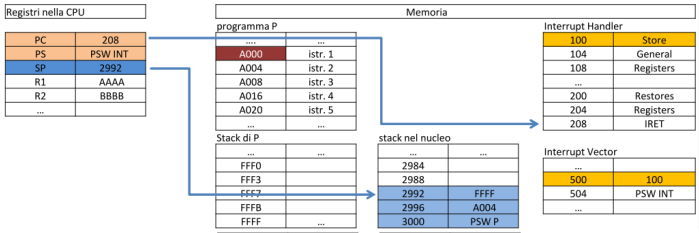
3) Stores general registers



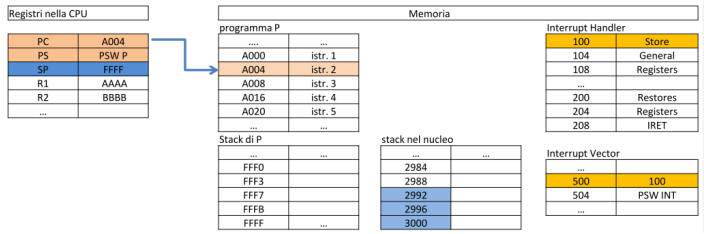
4) Executes interrupt handler



5) Restores general registers



6) Executes IRET (Interrupt Return)

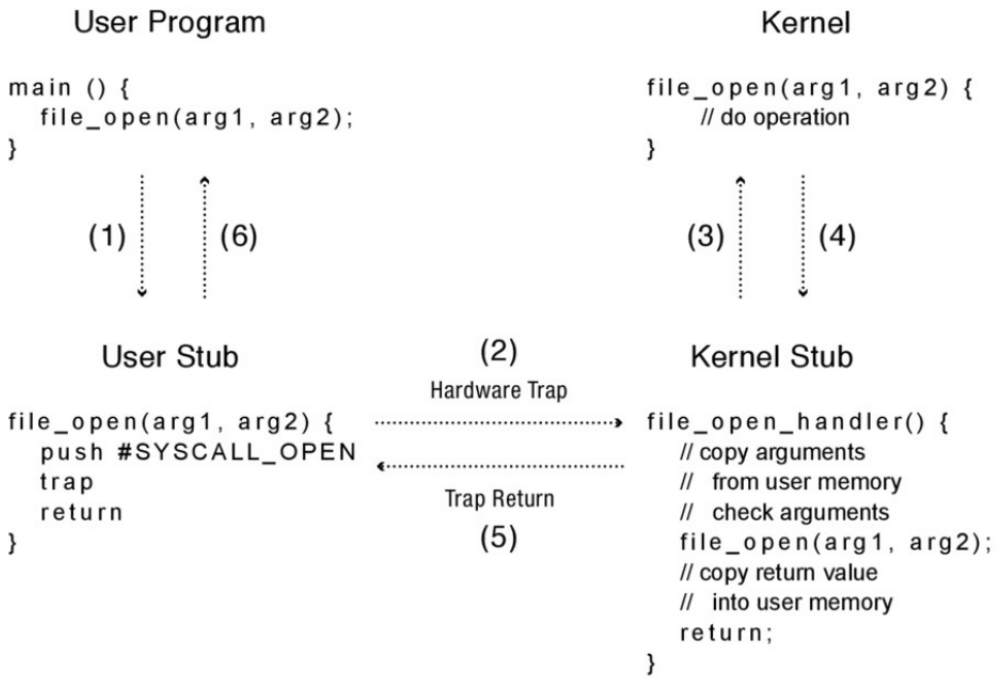


2.6 System Calls

Il SO riserva un ambiente ristretto per i processi in esecuzione per limitare l'impatto di programmi errati o maliziosi. Ogni volta che un processo ha bisogno di eseguire un'operazione fuori dal proprio dominio – per creare un nuovo processo, leggere da tastiera o scrivere nel disco – deve chiedere al SO di eseguire per lui questa operazione attraverso una *system call*.

Vengono trattate esattamente come delle interruzioni. Le system calls danno l'illusione che il SO sia semplicemente un insieme di librerie di routine utilizzabili dai programmi utenti. Infatti, ad un programma utente, il kernel fornisce un insieme di procedure di chiamate di sistema, le quali hanno i propri argomenti e valori di ritorno e che possono essere chiamate come semplici procedure.

Ci sono due procedure che mediano tra l'ambiente del programma utente e il kernel.



1. Il programma utente chiama un *user stub*.
2. L'user stub prende il posto del codice della system call ed esegue un'istruzione trap.
3. L'hardware trasferisce il controllo al kernel e viene eseguito l'handler specifico della system call. L'handler agisce come una stub del kernel, controlla (se sono maliziosi o meno) e copia gli argomenti e chiama l'implementazione kernel della system call.
4. Dopo che la system call ha terminato, si ritorna all'handler.
5. L'handler ritorna al livello utente alla prossima istruzione dell'user stub (con una IRET).
6. La stub ritorna al chiamante.

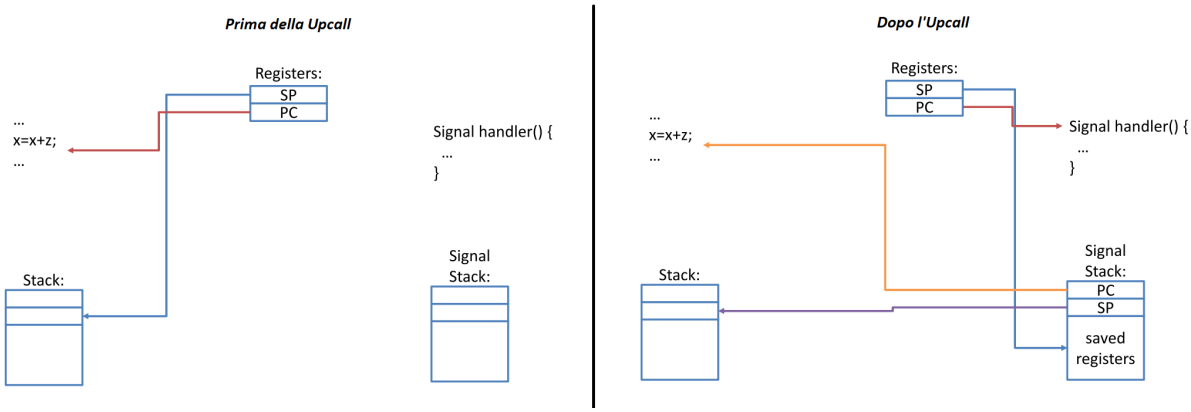
Vediamo in particolare quali sono le azioni di una stub del kernel (l'handler):

- **Localizzare gli argomenti della system call.** Al contrario delle normal procedure kernel, gli argomenti di una system call sono memorizzati nello stack dell'utente. L'indirizzo dello stack è un indirizzo logico e non fisico e quindi potrebbe essere corrotto. La stub deve quindi verificare che siano indirizzi legali (rientrano nel dominio dell'utente). Se è così, la stub lo converte in indirizzo fisico così che il kernel possa utilizzarlo con sicurezza.
- **Convalidare i parametri.** Il kernel deve anche proteggersi da eventuali contenuti maliziosi o errati degli argomenti. Se viene individuato un errore, il kernel lo restituisce al programma utente altrimenti, il kernel esegue l'operazione al posto dell'applicazione.
- **Copia prima controllo.** In molti casi, il kernel copia i parametri della system call nella memoria del kernel prima di controllarli. Si fa questo per prevenire la modifica, da parte dell'applicazione, dei parametri dopo che lo stub ha controllato i valori ma prima che i parametri siano usati nell'implementazione della routine. Viene chiamato attacco *time of check vs. time of use* (TOCTOU). Ad esempio, un'applicazione potrebbe chiamare la open di un file valido ma subito dopo modificarne il nome in modo da accedere a file privati di un utente.
- **Copia a ritroso dei risultati.** Il programma utente per accedere ai risultati della system call, la stub deve copiarli dalla memoria kernel a quella dell'utente.

2.7 Upcalls

Come per il kernel, anche le applicazioni potrebbero avere la necessità di essere interrotte se si presenta un evento asincrono. C'è bisogno di *virtualizzare* alcune parti del kernel in modo che le applicazioni si comportino come dei sistemi operativi. Le interruzioni e eccezioni virtualizzate le chiameremo *upcalss.*. In UNIX sono chiamate *signals*; in Windows, *asynchronous events*.

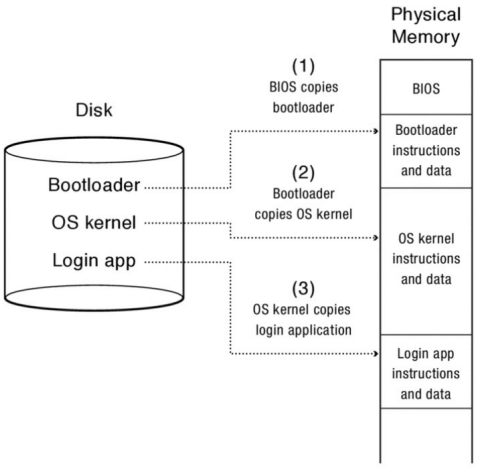
Alcuni esempi: il processo di un programma utente potrebbe avere molti threads e potrebbe settare un timer upcall periodico per andare da un thread ad un altro; un programma di gestione delle e-mails potrebbe aspettare l'arrivo di una una applicazione potrebbe utilizzare una propria gestione delle eccezioni e per questo il SO potrebbe informarla quando le riceve dal processore e di fargliele gestire invece di gestirle lui stesso ecc.



2.8 Booting

Quando un computer viene avviato, setta il Program Counter della macchina in modo da partire ad eseguire un posizione di memoria predefinita. Poiché il computer non sta ancora girando, le istruzioni macchina iniziali devono essere fetchate ed eseguite immediatamente dopo che venga acceso (il computer) e prima che il sistema abbia la possibilità di inizializzare la sua DRAM. Tipicamente i sistemi utilizzano una memoria hardware di sola lettura (*Boot ROM*) per memorizzare le proprie istruzioni di avvio. In molti computer x86, il programma di avvio è chiamato *BIOS* (Basic Input/Output System).

Si potrebbe pensare di memorizzare l'intero SO nella ROM ma al di là del fatto che le istruzioni all'interno vengono fissare quando il computer è fabbricato, il problema principale è che il SO ha bisogno di essere aggiornato e la ROM è relativamente lente e costosa. Questo induce a dover mettere solo una piccola parte del codice nel BIOS.



Il BIOS legge un blocco di bytes fissato da una posizione fissata nel disco, chiamato *bootloader*, nella memoria. Dopodiché salta alla prima istruzione nel blocco. Nelle macchine più recenti, il BIOS controlla anche che il bootloader non sia corrotto, per far questo il bootloader viene memorizzato con una *segnatura cifrata* e il BIOS controlla che sia valida.

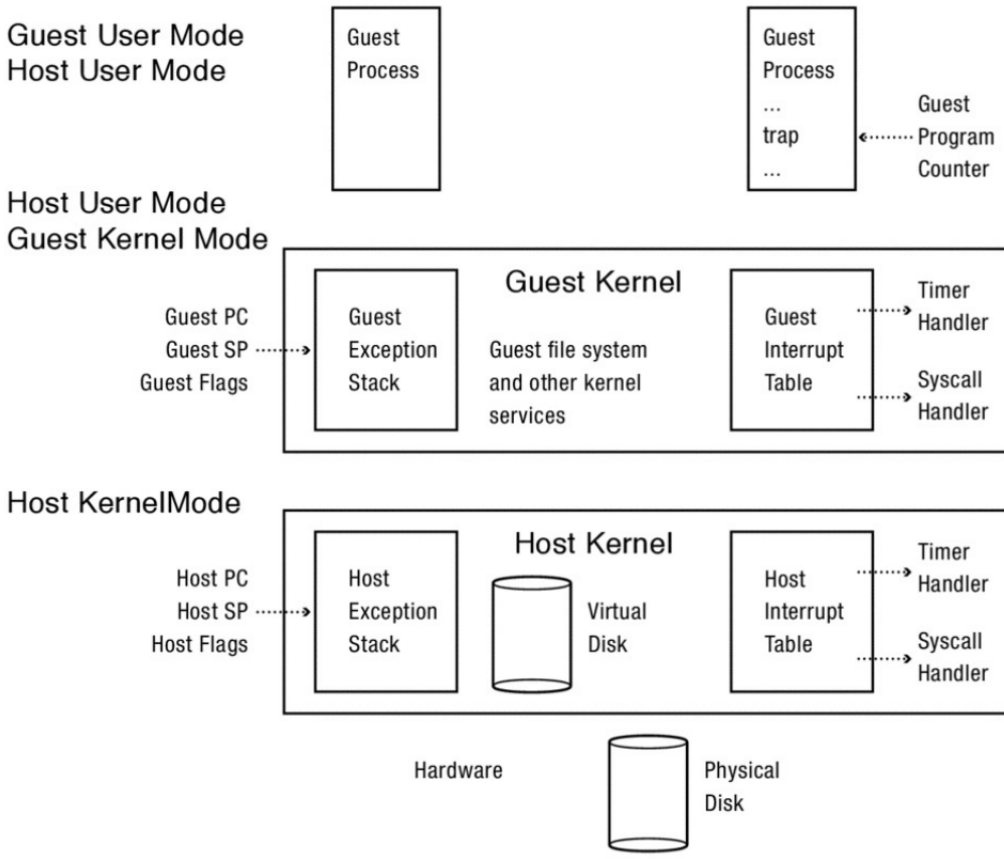
Il bootloader a questo punto carica il kernel nella memoria e ci salta. Anche qui, il bootloader può controllare che la segnatura del SO verifichi che non sia stata corrotta. L'immagine eseguibile del kernel è solitamente memorizzata nel file system. Quindi, il BIOS deve leggere un blocco di memoria dal disco per trovare il bootloader e quest'ultimo ha bisogno di sapere come leggere dal file system per trovare e leggere l'immagine del SO.

Quando il kernel si avvia, può inizializzare le proprie strutture dati, incluso il far puntare l'interrupt vector ai vari handler (per le interruzioni, eccezioni del processore e system call). Il kernel poi comincia il primo processo, tipicamente la pagina di login. Per far avviare questo processo, il SO legge il codice del programma di login dal disco e salta alla sua prima istruzione.

2.9 Macchine Virtuali

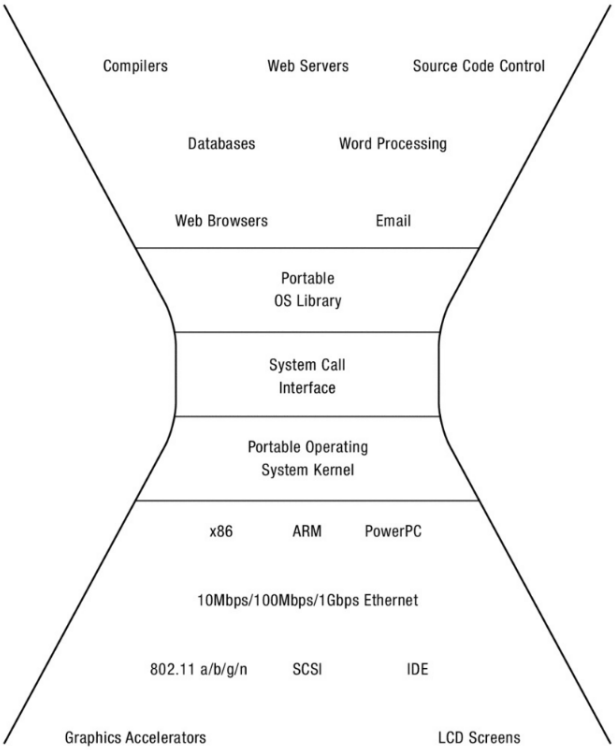
Una volta c'erano due tipi di SO: *UNIX* (Linux, Windows, MacOS) e a *macchine virtuali*. Il SO che fornisce lo strato di macchina virtuale è chiamato *host* mentre quello che gira nella macchina virtuale è chiamato *guest*. Il sistema operativo host dà l'illusione al kernel del guest di girare su un hardware reale. Le operazioni normali vengono eseguite dal processo utente nel SO guest, le system call vengono passate al guest kernel mentre tutte le altre (tipo le interruzioni da disco) vengono gesiste dal SO host che le esegue sull'hardware e alla fine le passa a quello guest.

Affinché funzioni tutto correttamente il SO host deve virtualizzare l'hardware. Se ad esempio il guest kernel deve lavorare sul disco chiederà all'host kernel di lavorare ma questo lavorerà sul disco virtuale e non su quello fisico.



Le macchine virtuali servono principalmente per avere una doppia protezione: i controlli di liceità vengono già fatti dal guest kernel, se sono lecite le passa all'host altrimenti le ferma.

3 Strutture dei Sistemi Operativi

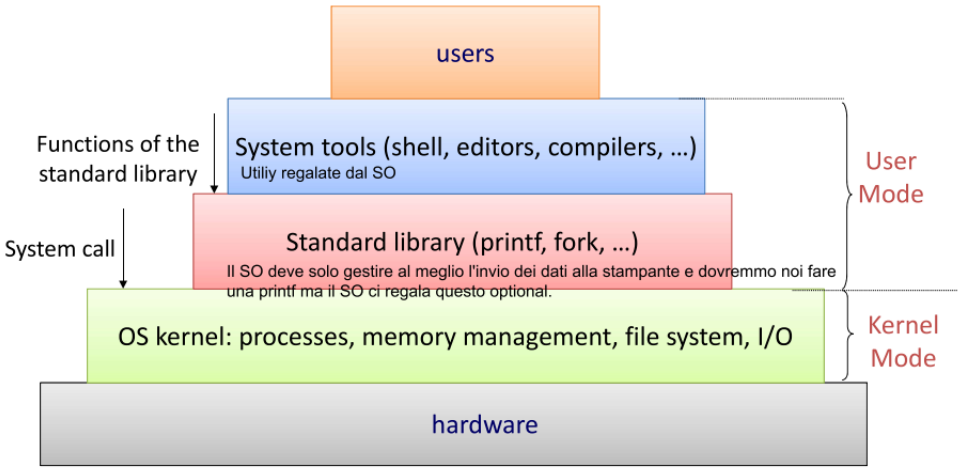


Lo strato identificato con *Portable Operating System Kernel* non è altro che il kernel e serve per uniformare l'hardware rispetto al SO.

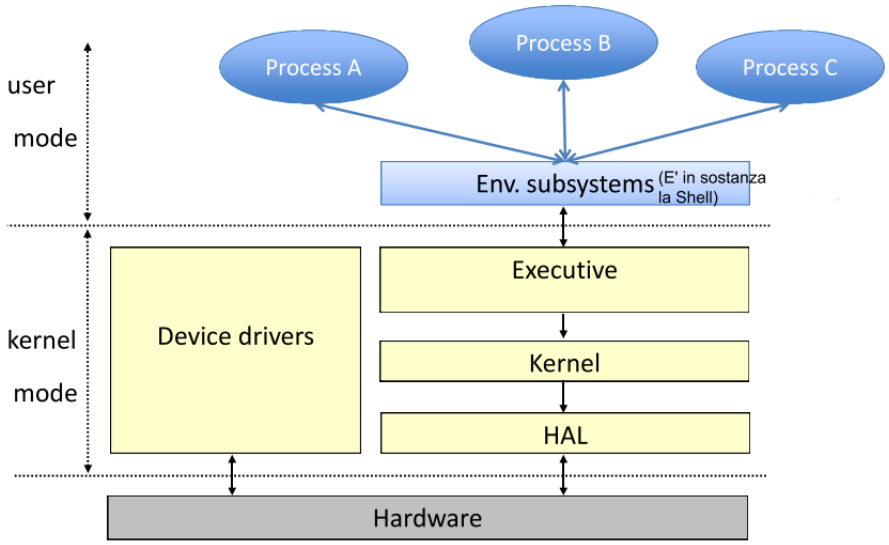
La *System Call Interface* è un'interfaccia che permette agli utenti di interagire direttamente del sistema operativo (non fa parte del kernel).

Ed infine ci sono le librerie, *Portable OS Library*, che vengono eseguite in spazi utente, "regalate" all'utente insieme al SO (e.g. le librerie dei fonts) ma che non sono il SO stesso.

Unix



Windows



I principali punti di interesse saranno:

- **Creazione e gestione dei processi.** Vedremo quale sono le system call per gestire i processi (`fork, exit` e `wait` per la creazione).
- **Input/output.** Vedremo quale sono le system call:`open, read, write, close`.
- **Comunicazione tra processi.** Vedremo quale sono le system call: `pipe, dup, select, connect`.

3.1 Creazione e gestione dei processi – Windows

In Windows, c'è una routine chiamata `CreateProcess`:

```
boolean CreateProcess(char *prog, char *args);
```

Chiameremo il processo creatore il *padre* e il processo creato, il *figlio*.

Quali sono i passi che deve fare? Il kernel ha bisogno di:

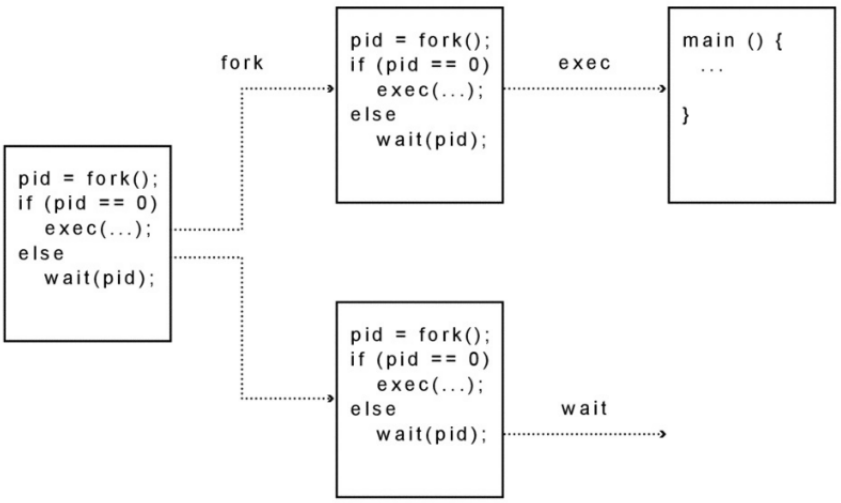
- Cerare ed inizializzare il Process Control Block (PCB) nel kernel.
- Creare ed inizializzare un nuovo spazio di indirizzamento.
- Caricare il programma `prog` nello spazio di indirizzamento.
- Copiare gli argomenti `args` nella memoria nello spazio di indirizzamento.
- Preparare l'hardware affinché possa essere messo in esecuzione.
- Informare lo schedulatore che il nuovo processo è pronto a partire.

In realtà la creazione dei processi è più compilata:

```
if (!CreateProcess(NULL,   // No module name (use command line)
      argv[1],             // Command line
      NULL,                // Process handle not inheritable
      NULL,                // Thread handle not inheritable
      FALSE,               // Sert handle inheritable to FALSE
      0,                   // No creation flags
      NULL,                // Use parent's environment block
      NULL,                // Use parent's starting directory
      &si,                 // Pointer to STARTUPINFO structure
      &pi )                // Pointer to PROCESS_INFORMATION structure
)
```

3.2 Creazione e gestione dei processi – UNIX

UNIX ha un approccio differente. UNIX divide `CreateProcess` in due steps, chiamati `fork` e `exec`. La `fork` clona il processo padre e al processo figlio creato gli viene scelto un nuovo PCB. Siccome il processo figlio ha il solito codice del padre, per eseguire processi nuovi il figlio invoca la `exec`. La `exec` cambia il processo in esecuzione. Se vogliamo che il processo padre aspetti che finisca il processo figlio, il padre invoca una `wait`.

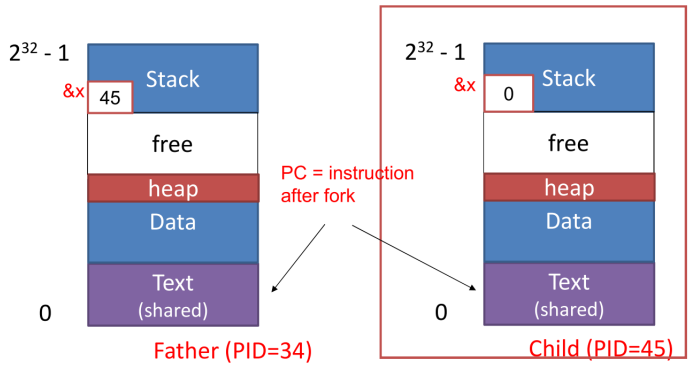


3.2.1 UNIX fork

I passi per implementare la `fork` nel kernel sono i seguenti:

- Creare e inizializzare il Process Control Block (PCB) nel kernel.
- Creare un nuovo spazio di indirizzamento.
- Inizializzare lo spazio di indirizzamento con una copia dell'intero contenuto dello spazio di indirizzamento del padre (memoria logica).
- Ereditare il contesto di esecuzione del padre (il PCB del padre viene copiato del PCB del figlio).
- Informare lo schedulatore che un nuovo processo è pronto per partire.

Un aspetto strano della `fork` è che la system call ritorna due valore: uno al padre e uno al figlio. Al padre, UNIX ritorna l'ID del processo figlio (PID, ovvero l'indice del PCB del figlio nel vettore dei PCB); al figlio ritorna 0 se la `fork` ha avuto successo; un numero negativo altrimenti. Questo per distinguere qual è il clone e quale l'originale.



Da notare che a seconda della nostra architettura (e.g. multicore) oppure se avviene un'interruzione dopo che ho chiamato la `fork` (e.g. timer interrupt) possiamo avere effetti differenti:

```
int child_pid = fork();

if (child_pid == 0) {   // I'm the child process.
    printf("I am process #%d\n", getpid());
    return 0;
} else {                // I'm the parent process.
    printf("I am the parent of process #%d\n", child_pid);
    return 0;
}

Possible output:
    I am the parent of process 495
    I am process 495

Another less likely but still possible output:
    I am process 456
    I am the parent of process 456
```

3.2.2 UNIX exec

La `exec` completa il passo necessario a far partire un nuovo programma. Il processo figlio solitamente chiama la `exec` una volta che è ritornato dalla `fork` e configura l'ambiente di esecuzione per un nuovo processo. La `exec` fa i seguenti passi:

- Carica il programma `prog` nello spazio di indirizzamento corrente.
- Copia gli argomenti `args` nella memoria dello spazio di indirizzamento.
- Inizializza il contesto hardware per partire l'esecuzione di "start".

La `exec` non crea un nuovo processo ma sostituisce i segmenti codice e dati del processo correntemente in esecuzione nello stato di utente con quelli di un altro programma contenuto in un file eseguibile specificato. Il processo che ha chiamato l'`exec` mantiene il solito PID, il solito PCB (strutture del processo e dell'utente), vengono resettati i segnali pendenti, mantenuto lo stesso stack del kernel e mantiene le stesse risorse assegnate al padre (e.g. files). La `exec` restituisce un codice di errore se non ha successo, niente se ha successo.

3.2.3 UNIX wait

La system call `wait` mette in pausa il processo padre finché il figlio non ha terminato, crashato o finito. Siccome il padre potrebbe aver generato molti processi figli, `wait` è parametrizzato col PID del figlio:

```
int wait(int *status);
```

Il valore restituito è lo stato del PID o un codice di errore.

3.2.4 UNIX exit

Un processo può terminare se durante la sua esecuzione ha generato un'eccezione (divide-by-zero, accesso fuori dal proprio spazio di memoria ecc.) oppure se ha invocato la system call `exit`:

```
void exit(int status);
```

Il processo terminato con una `exit` restituisce un *exit value* al padre. Questi valori vengono ricevuti dal padre che magari era in stato di wait, si libera quindi dallo stato di wait e viene messo nella coda dei *pronti*.

Da notare che il padre potrebbe non aver fatto ancora la `wait` perché stava svolgendo altre istruzioni e quindi il processo che aveva invocato la `exit` passa nello stato di *zombie* perché è "morto" ma non può terminare finché non sono stati restituiti i valori al padre. Oppure il padre potrebbe NON aver fatto la `wait` ed aver già terminato e quindi il processo figlio si dice che diventa *orfano* e viene adottato dal primo processo attivo a ritroso (c'è n'è sempre uno vivo: il processo *init*, generato appena si accende il computer).

3.3 Implementazione di una Shell

La Shell è in sintesi un'ambiente di programmazione ed è l'interfaccia tra l'utente e il SO in cui possiamo far girare i nostri programmi. Viene eseguita a livello utente e permette di chiamare system calls. Un utente deve poter, al minimo, scrivere delle system call, ovvero dei *comandi* e generare nuovi processi, ovvero fare la *fork*.

3.4 Input/Output

I computer hanno una vasta gamma di dispositivi di input e output: tastiera, mouse, disco, porte USB, Ethernet, WiFi, display, hardware timer, microfono, camera, accelerometro e GPS. Per gestire questa diversità i primi sistemi specializzavano l'API per ogni dispositivo, personalizzandola per le specifiche caratteristiche di ognuno ma ogni volta che veniva inventato un nuovo dispositivo hardware, l'interfaccia della system call doveva essere aggiornata per gestirlo.

Una delle prime innovazioni di UNIX è stato proprio quello di regolarizzare tutti i dispositivi di input e output dietro una singola interfaccia. Infatti, UNIX usa la stessa interfaccia per leggere e scrivere files e per la comunicazione tra processi. Le idee di base dell'interfaccia I/O di UNIX sono le seguenti:

- **Uniformity.** Tutti i dispositivi I/O, le operazioni per i file, e la comunicazione tra processi utilizzano lo stesso insieme di system calls: *open, close, read* e *write*
- **Open before use.** Prima che un applicazione svolga operazioni di I/O, deve prima invocare la `open` sul dispositivo, il file o il canale di comunicazione. Questo fornisce al SO la possibilità di controllare i permessi di accesso. Alcuni dispositivi, come la stampante, permettono un accesso alla volta – la `open` può ritornare errore se è già in uso.

- **Byte-oriented.** Tutti i dispositivi, anche quelli che trasferiscono blocchi di dati prefissati, sono acceduti con array di byte. Ovvero UNIX vede la memoria come un vettore di byte e non di parole da 32/64 bits. Similmente, l'accesso ai file e ai canali di comunicazione è in termini di bytes, anche se memorizziamo strutture dati in file e inviamo strutte dati tra canali.

- **Kernel-buffered reads.** Lo stream dei dati, dalla rete o dalla tastiera, è memorizzato in un kernel buffer ed è poi ritornato all'applicazione che l'ha richiesto. Questo permette alla system call `read` di esser uguale per ogni device.

- **Kernel-buffered writes.** Allo stesso modo, i dati in uscita sono memorizzato in un kernel buffer per trasmetterli quando il dispositivo è disponibile. Normalmente, la system call `write` copia i dati nel kernel buffer e ritorna immediatamente. Questo disgiunge l'applicazione dal dispositivo, permettendo ad ogni device di andare alla propria velocità. Se l'applicazione genera dati più velocemente di quanto possa ricevere il dispositivo, la `write` si ferma nel kernel finché non c'è abbastanza spazio per memorizzare il nuovo dato nel buffer.

- **Explicit close.** Quando un'applicazione ha finito di lavorare con un dispositivo o un file, può chiamare la `close`. Questo segnala al SO che può decrementare il contatore di riferimenti sul dispositivo e cestina qualsiasi struttura dati del kernel inutilizzata.

4 Concorrenza e Threads

Si utilizza la parola *concorrenza* per riferirsi ad attività multiple che possono avvenire nello stesso tempo. Gestire correttamente la concorrenza è una sfida chiave per gli sviluppatori di sistemi operativi. Le astrazioni sono utilizzate per esprimere e gestire la concorrenza dai sistemi operativi odierni, servono per ridurre la complessità e per migliorare sia l'affidabilità che le prestazioni del sistema.

Dal punto di vista del programmatore, l'idea di base è quella di scrivere un programma concorrente come un insieme di esecuzioni sequenziali (e.g. procedure, funzioni ecc.), chiamati *threads*, che può mettere in esecuzione, indipendentemente l'uno da l'altro. I threads interagiscono e condividono risultati in modo preciso.

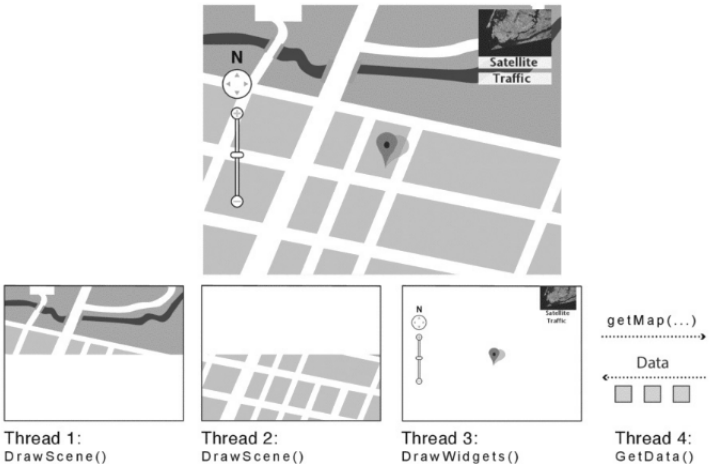
Ogni thread si comporta come se avesse il processore tutto per sé. Le astrazioni sui thread permettono ai programmatori di creare quanti threads necessita senza preoccuparsi del numero esatto di processori fisici di cui il computer dispone o di cosa stia facendo ad ogni istante un processore. Ovviamente i threads sono una astrazione: l'hardware ha un numero limitato di processori (nel peggiore dei casi solo uno).

Il lavoro del SO è di fornire l'illusione di avere infiniti processori virtuali anche se l'hardware è molto limitato. Per fare ciò sospende e riattiva, in maniera trasparente, i singoli threads, così che ad ogni istante solo un sottoinsieme di threads è attivo.

In un programma possiamo rappresentare ogni task indipendente con un thread. Ogni thread fornisce l'astrazione di una esecuzione sequenziale simile ai tradizionali modelli di programmazione. Infatti, possiamo pensare ad un programma tradizionale come un *single-thread* con una sola sequenza logica di passi in cui ogni istruzione segue quella precedente.

Invece in un programma *multi-thread* può avere più thread eseguiti contemporaneamente in cui ogni singolo thread segue una propria sequenza di passi.

Ad esempio, un programmatore potrebbe scrivere del codice in cui ad ogni thread assegna il compito di disegnare una porzione dello schermo per visualizzare una porzione di una mappa.



4.1 Threads vs. Processi

Un *processo* è l'esecuzione di un programma con diritti limitati mentre un *thread* è una sequenza indipendente di istruzioni eseguiti all'interno di un programma. Vediamo come i sistemi operativi combinano le due cose:

- **Un thread per processo.** Un'applicazione single-thread ha una sequenza di istruzioni, eseguita dall'inizio alla fine. Il kernel esegue queste istruzioni in user mode per restringere l'accesso ad operazioni privilegiate o alla memoria del sistema. Il processo esegue system call per chiedere al kernel di eseguirle al posto suo.

- **Molti thread per processo.** In alternativa, un programma potrebbe essere diviso in molti threads concorrenti, ognuno dei quali è eseguito con i soliti limitati diritti del processo. Ad ogni istante, un sottoinsieme dei thread del processo possono essere in esecuzione mentre i restati sono sospesi. Ogni thread che è eseguito in un processo può invocare system call (quindi essere sospeso fino al ritorno della chiamata) mentre gli altri thread possono continuare a lavorare. Similmente, quando un processore ricevere un interruzione di I/O, ferma uno dei thread attivi ed esegue l'handler; quando l'handler ha terminato, il kernel può riattivare il thread.

- **Molti processi single-thread.** Venti anni fa, molti sistemi operativi supportavano multipli processi ma aventi un solo thread per processo. Per il kernel però, ogni processo sembra un thread.

- **Molti kernel thread.** In questo caso, ogni *kernel thread* è eseguito con privilegi del kernel: può eseguire istruzioni privilegiate, accedere alla memoria del sistema ed eseguire comandi direttamente ai dispositivi I/O.

Per via dell'utilità dei thread, quasi tutti i sistemi operativi odierni supportano sia la possibilità di avere molti thread per processo sia molti kernel thread.

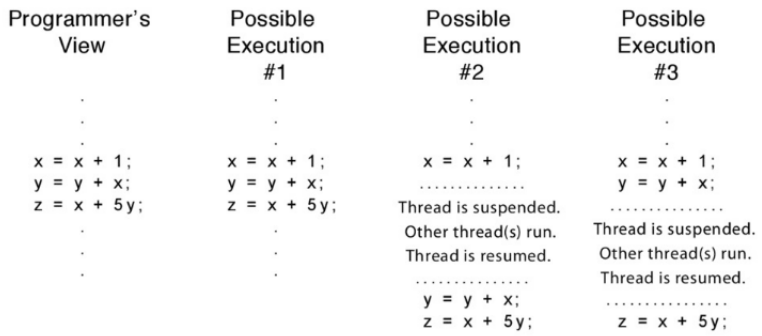
4.2 Thread Abstraction

Un *thread* è *una singola sequenza di esecuzione* che rappresenta *un task schedulabile indipendente*:

- **Una singola sequenza di esecuzione.** Ogni thread esegue una sequenza di istruzioni – assegnamenti, loop, procedure ecc. – proprio come nei modelli tradizionali di programmi sequenziali.
- **Un task schedulabile idipendente.** Il SO può eseguire, sospendere o riattivare un thread quando vuole.

Per mappare un insieme di threads su un insieme fissato di processori il SO include un *thread scheduler* che può alternare l'esecuzione di quelli attivi con quelli in stato di *ready* e che quindi non sono ancora in esecuzione. L'astrazione fa apparire ogni thread come se fosse un singolo stream di esecuzione; questo significa che il programmatore può tener di conto della sequenza delle istruzioni all'interno di un thread e non se e quando questa sequenza potrebbe essere temporaneamente sospesa per far eseguire un altro thread.

Ogni thread dunque, è eseguito su un processore virtuale dedicato con velocità impredicibile e variabile. Dal punto di vista del codice del thread, ogni istruzione sembra eseguita una dopo l'altra. Nonostante ciò, lo schedulatore potrebbe sospendere un thread tra un'istruzione e la successiva e riattivarlo più tardi. E' come se il thread fosse eseguito in un processore che a volte è molto lento.



Da notare che un kernel interrupt handler non è un thread perché anche se è sempre una singola sequenza di istruzioni, viene eseguita dall'inizio alla fine, non schedulabile ed è attivato da un evento hardware invece che da una decisione del thread scheduler. Inoltre, una volta che l'handler è partito, non può essere interrotto (se non da un'interruzione prioritaria).

4.3 Per-Thread State e Thread Control Block (TCP)

Il SO ha bisogno di una struttura dati per rappresentare lo stato di un thread. Questa struttura dati è chiamata *Thread Control Block (TCB)*. Il SO, per ogni thread, crea un TCB.

Il TCB ha due tipi di informazioni per-thread:

1. Lo stato della computazione che deve essere eseguita dal thread.
2. Metadati del thread che il SO utilizza per gestirlo.

4.3.1 Stato della computazione

Per creare thread multipli e per iniziare e fermare ogni thread, il SO deve allocare spazio nel TCB per lo stato corrente di ogni computazione del thread: un puntatore allo stack del thread per memorizzare le informazioni dei frame (variabili locali, parametri, indirizzo di ritorno ecc.) e una copia dei propri registri del processore (registri generali, PC e SP) così da poter essere rieseguito nella stato in cui era stato sospeso.

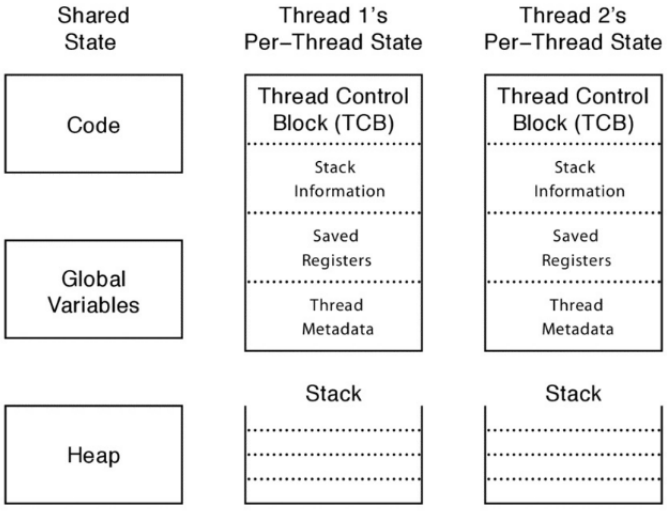
4.3.2 Metadati

I metadati includono il *thread ID*, la priorità, lo stato (e.g. se è in attesa di un evento o è pronto per essere messo nel processore).

4.4 Shared State

Al contrario dello stato per-thread che è allocato per ogni thread, alcuni stati sono condivisi tra i thread che stanno girando nel solito processo o all'interno del kernel. In particolare, il codice del programma (condiviso da tutti i thread in un processo, sebbene ogni thread può eseguire parti diverse del codice), le variabili globali allocate staticamente e le variabili dello heap allocate dinamicamente.

Nonostante ci sia un importante divisione logica tra lo stato per-thread e lo shared state, il SO solitamente non rinforza questa divisione. Per evitare comportamenti inaspettati è perciò importante, quando si scrivono programmi multi-thread, sapere quali variabili sono designate per essere condivise tra thread (variabili globali, oggetti nello heap) e quali per essere private (variabili local/automatic).

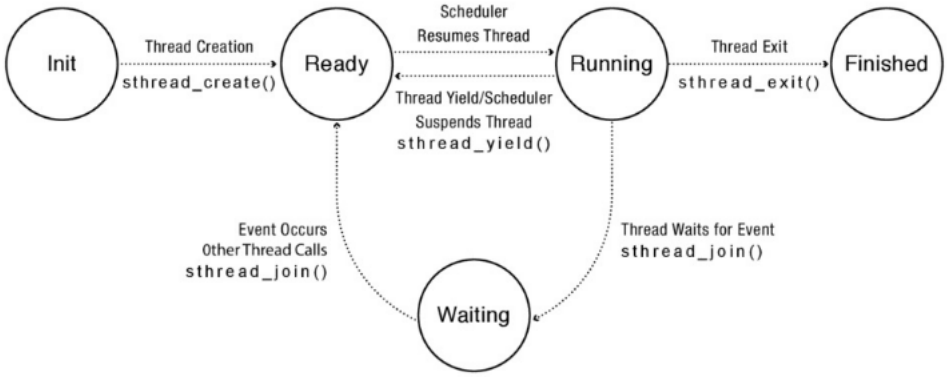


4.4.1 Dov'è il mio TCB?

Si potrebbe pensare che, per un thread, trovare il proprio TCB sia cosa facile: basta memorizzare un puntatore al TCB in una variabile globale. Il problema è che ogni thread gira nel solito processo, quindi utilizzano tutti il solito codice. Ogni thread vedrebbe nel medesimo posto il solito TCB.

Mentre in un uniprocessore questa strategia può funzionare, in un multiprocessore no. In alcuni sistemi la thread library può mantenere un array globale di puntatori dove l'indice identifica il processore ed il contenuto dell'i-esima posizione è proprio il TCB del thread attivo nell'i-esimo processore. In questo modo un thread può trovare il proprio TCB guardando l'id del proprio processore.

4.5 Thread Life Cycle



State of Thread	Location of Thread Control Block (TCB)	Location of Registers
INIT	Being Created	TCB
READY	Ready List	TCB
RUNNING	Running List	Processor
WAITING	Synchronization Variable's Waiting List	TCB
FINISHED	Finished List then Deleted	TCB or Deleted

4.6 Thread Operations

Una semplice API, basata sullo standard POSIX "pthread", è la seguente:

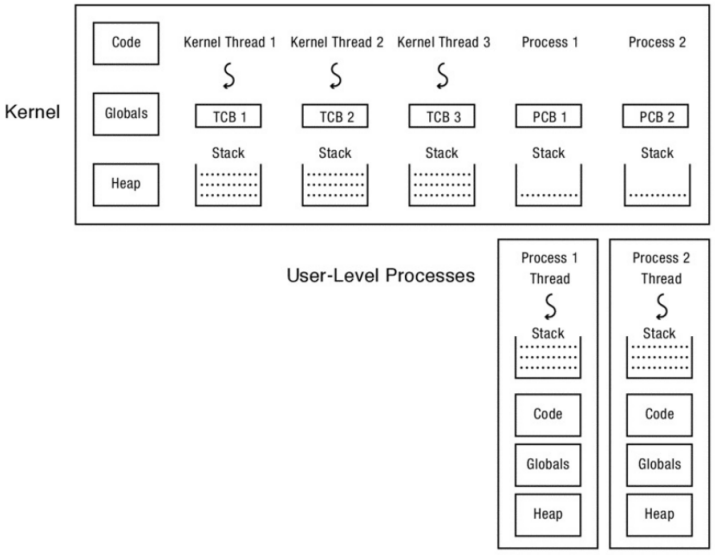
`void thread_create(thread, func, arg)`	Crea un nuovo thread e memorizza le informazioni in `thread`. In concomitanza, `thread` esegue la funzione `func` con gli argomenti `arg`.
`void thread_yield()`	Il thread che la chiama rilascia volontariamente il processore. Lo schedulatore può riattivarlo quando vuole.
`int thread_join(thread)`	Attende che finisca `thread` e poi riceve il valore passato dalla `thread_exit`. La `thread_join` può essere chiamata solo una volta per ogni thread.
`void thread_exit(ret)`	Termina il thread corrente. Memorizza il valore `ret` nella struttura dati di questo thread.

Il concetto è simile alle astrazioni UNIX per i processi: `thread_create` è analoga alla `fork` e `exec`, mentre la `thread_join` è analoga alla `wait`.

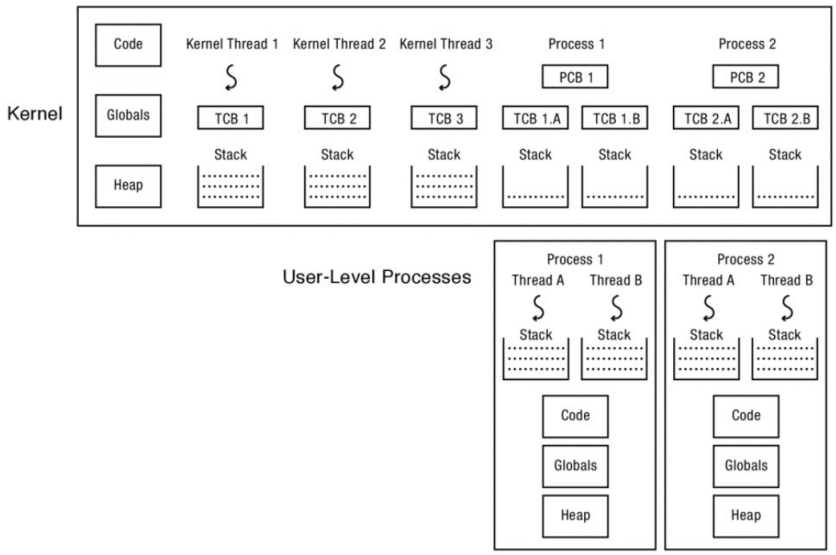
4.7 Implementazione dei Kernel Threads

La specifica dell'implementazione varia a seconda del contesto:

- **Kernel threads.** Il caso più semplice è implementare i threads all'interno del SO, e che condividono uno o più processori. Un thread del kernel esegue codice del kernel e modifica le sue strutture dati. Quasi tutti i SO odierni supportano i kernel thread.
- **Kernel threads e processi single-threaded.** Un SO con kernel threads potrebbe esseguire alcuni processi utente single-threaded. Questo processi possono invocare system call che girano contemporaneamente coi kernel thread all'interno del kernel.



- **Processi multi-threaded che usano i kernel threads.** Molti sistemi operativi forniscono un insieme di routine di libreria e system call per dare la possibilità alle applicazione di usare più thread all'interno di un singolo processo user-level. Questi thread eseguono codice utente e hanno accesso alle strutture dati a livello utente. Inoltre, chiamano system call nel kernel. Per questo, hanno bisogno di un kernel interrupt stack proprio come un normale processo single-threaded.



- **Thread user-level.** Per evitare di dover chiamare una system call per ogni operazione dei thread, alcuni sistemi supportano un modello in cui le operazione dei thread a livello utente – `create, yield, join, exit` – sono implementate completamente in una libreria a livello utente, senza dover invocare il kernel.

4.7.1 Creazione di un Thread

Lo scopo di `thread_create` è quello di eseguire una chiamata ad un procedura asincrona, la `func`, con gli argomenti `args`. Quando il thread sarà in esecuzione, sarà eseguita `func(arg)`.

Ci sono tre step per creare un thread:

1. **Allocare lo stato per-thread.** Il primo step nella costruzione di un thread è quello di allocare spazio per lo stato per-thread: il TCB e lo stack. Come abbiamo già detto, il TCB è una struttura dati che il sistema dei thread utilizza per gestirlo.

2. **Inizializzare lo stato per-thread.** Per inizializzare il TCB, i nuovi registri del thread vengono riempiti con ciò che è necessario non appena il thread va in stato di *RUNNING*. Quando il thread viene assegnato ad un processore, vogliamo che parta l'esecuzione di `func(arg)`. In realtà, viene avviata una `stub` che poi eseguirà la `func`.

   Abbiamo bisogno di questo passaggio in modo da poter assicurare l'esecuzione della `thread_exit` perché `func` potrebbe non chiamarla. Nello pseudo-codice che segue, sono stati pushati `func` e `arg` nello stack in modo tale che `stub` li trovi e ne abbia accesso come una normale procedura.

3. **Inserire il TCB nella lista dei ready.** L'ultimo step è quello di settare lo stato del thread in *READY* ed inserire il nuovo TCB nella lista dei ready, abilitando il thread ad essere schedulato.

Questo potrebbe essere lo pseudo-codice per la creazione di un thread:

```
// func is a pointer to a procedure the thread will run.
// arg is the argument to be passed to that procedure.
void
thread_create(thread_t *thread, void (*func)(int), int arg) {
    // Allocate TCB and stack
    TCB *tcb = new TCB();

    thread->tcb = tcb;
    tcb->stack_size = INITIAL_STACK_SIZE;
    tcb->stack = new Stack(INITIAL_STACK_SIZE);

    // Initialize registers so that when thread is resumed, it will start runnin
    // stub.  The stack starts at the top of the allocated region and grows down
    tcb->sp = tcb->stack + INITIAL_STACK_SIZE;
    tcb->pc = stub;

    // Create a stack frame by pushing stub's arguments and start address
    // onto the stack: func, arg
    *(tcb->sp) = arg;
    tcb->sp--;
    *(tcb->sp) = func;
    tcb->sp--;

    // Create another stack frame so that thread_switch works correctly.
    // This routine is explained later in the chapter.
    thread_dummySwitchFrame(tcb);

    tcb->state = READY;
    readyList.add(tcb);    // Put tcb on ready list
}

void
stub(void (*func)(int), int arg) {
    (*func)(arg);           // Execute the function func()
    thread_exit(0);         // If func() does not call exit,  call it here.
}
```

4.7.2 Eliminazione di un Thread

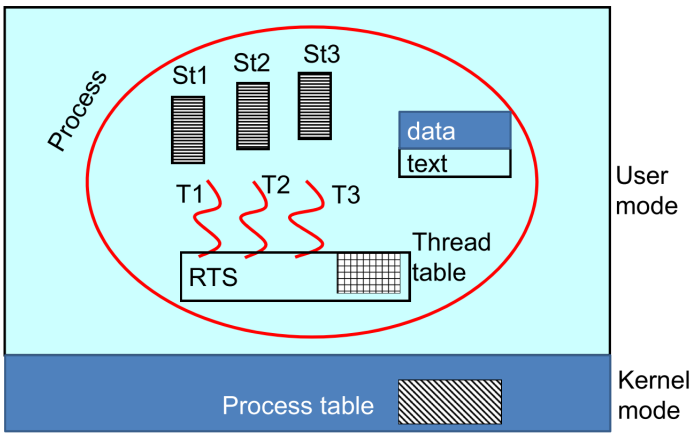
Quando un thread chiama la `thread_exit`, seguono due step per eliminarlo:

- Rimuovere il thread nella lista dei ready in modo da non poter essere più eseguito.
- Liberare lo stato per-thread allocato per il thread.

Nonostante sembri semplice, c'è un'importante sottigliezza: se un thread si rimuove dalla lista dei ready e libera il proprio stato per-thread, potrebbe interrompersi (e.g. se si rimuove nella ready list e prima che finisca, avviene un'interruzione e deve essere sospeso, non c'è modo di riesumarlo). Per ovviare a questo problema, un thread non elimina mai il proprio stato ma un altro lo fa per lui. In uscita, il thread transita nello stato *FINISHED*, sposta il proprio TCB dalla lista dei ready in una lista di thread finiti. Si passa al thread successivo nella lista dei ready e una volta che il thread finito non sta più girando, è sicuro, per un altro thread, liberare il suo stato.

4.8 User-level threads

I threads sono gestiti da una libreria a livello utente e non sono implementati a livello kernel. La libreria fornisce il supporto per la creazione, lo scheduling e la gestione dei thread senza alcun supporto da parte del kernel. All'interno del processo ci sarà una tabella dei propri thread e se uno di questi chiede, ad esempio, un input/output di dati che richiede l'intervento del kernel, questo non lo riconosce come thread singolo fatto di una componente di una collettività di vari thread ma lo riconosce come un unico processo e blocca tutti i thread di quel processo.



PRO

- Creazione, terminazione e context switch molto efficiente:
  - non c'è bisogno di invocare system call ma basta invocare delle chiamate alla libreria (non c'è bisogno di fare il cambio stato utente-stato supervisore),
  - in caso di context switch lo spazio di indirizzamento rimane lo stesso.
- Possono essere implementati in qualsiasi SO che non supporta il multithreading (e.g. prime versioni di UNIX).

CONTRO

- Una system call bloccante blocca tutti thread del processo. Perché il processore vede solo i processi,
- Non ha i vantaggi dell'architettura multiprocessore.

4.9 Kernel-level threads

Sono i thread implementati nel kernel. Avremo una tabella dei thread al suo interno. La creazione, la terminazione e il context switch sono attivate da system call. Diversi thread di uno stesso processo possono girare in parallelo in differenti processori.

Inoltre lo scheduling è implementato dal SO e se un thread si blocca, non si bloccano gli altri thread del medesimo processo.

4.10 Thread Context Switch

Un *thread context switch* sospende l'esecuzione di un thread corrente e riesuma l'esecuzione di un altro thread. Quando si commuta di contesto si salva i registri del thread corrente nel suo TCB e nello stack, poi si memorizza i registri del processore col contenuto del TCB e lo stack del nuovo thread.

Una commutazione di contesto può avvenire sia volontariamente tramite una chiamata del thread che involontariamente a causa di un'interruzione o un'eccezione del processore.

- **Volontariamente.** Tramite la `thread_yield`, `thread_join` o la `thread_exit` un thread può volontariamente avviare una commutazione di contesto col prossimo thread nella lista dei ready.

- **Involontariamente.** Una *interruzione* o un'*eccezione del processore* potrebbero invocare un interrupt handler. Viene quindi salvato lo stato del thread che è in esecuzione ed poi eseguito il codice dell'handler. L'handler può decidere che un'altro thread venga messo in esecuzione. Alternativamente, se il thread che era in esecuzione deve continuare a girare, l'handler rimemorizza il suo stato nel processore e riparte la sua esecuzione.

Siccome vogliamo evitare che ci sia un'interruzione involontaria durante una volontaria, quando avviene lo scambio dei due thread, dobbiamo posticipare le interruzioni fino a che non è stato completato lo scambio.

4.10.1 Context Switch Volontaria - Kernel Thread

Un thread chiama la `thread_yield` per lasciare volontariamente il processore ad un altro thread. Si salvano i registri nel suo TCB, si passa al nuovo thread ed infine si prende lo stato dal TCB di questo nuovo thread e si memorizza nel processore.

Lo pseudo codice per la `thread_yield` potrebbe essere il segue:

```
// We enter as oldThread, but we return as newThread.
// Returns with newThread's registers and stack.
void thread_switch(oldThreadTCB, newThreadTCB) {
    pushad;                 // Push general register values onto the old stack.
    oldThreadTCB->sp = %esp; // Save the old thread's stack pointer.
    %esp = newThreadTCB->sp; // Switch to the new stack.
    popad;            // Pop register values from the new stack.
    return;
}

void thread_yield() {
    TCB *chosenTCB, *finishedTCB;

    // Prevent an interrupt from stopping us in the middle of a switch.
    disableInterrupts();

    // Choose another TCB from the ready list.
    chosenTCB = readyList.getNextThread();
    if (chosenTCB == NULL) {
        // Nothing else to run, so go back to running the original thread.
    } else {
        // Move running thread onto the ready list.
        runningThread->state = ready;
        readyList.add(runningThread);
        thread_switch(runningThread, chosenTCB); // Switch to the new thread.
        runningThread->state = running;
    }

    // Delete any threads on the finished list.
    while ((finishedTCB = finishedList->getNextThread()) != NULL) {
        delete finishedTCB->stack;
        delete finishedTCB;
    }
    enableInterrupts();
}
```

4.10.2 Context Switch Involontaria - Kernel Thread

Il meccanismo è molto simile a quello visto per i processi.

1. **Salvataggio dello stato.** Si salva i registri del thread che sta girando così che l'handler possa essere eseguito senza disordinare il thread interrotto.

   L'hardware salva una porzione di stato quando occorre un'interruzione o un'eccezione, e il software salva il resto quando l'handler viene eseguito.

2. **Handler del kernel.** Viene eseguito il codice dell'handler per gestire l'interruzione o l'eccezione. Siccome siamo già in kernel mode, in questo step, non abbiamo bisogno di cambiare da modalità utente a modalità kernel. Inoltre non abbiamo bisogno di cambiare lo Stack Pointer per puntare alla base dell'interrupt stack perché possiamo semplicemente pushare lo stato salvato o le variabili dell'handler nello stack corrente, partendo dallo Stack Pointer corrente.

3. **Ripristino dello stato.** Si ripristina i registri del prossimo thread nella lista dei thread così da poterlo riprendere da dove era stato interrotto.

4.11 Thread switch - Overhead

Quando facciamo un cambio di contesto, il tempo in cui il processore impiega a fare operazione del SO lo indichiamo come un *overhead*. Da un punto di vista dell'utente infatti, è tempo sprecato. Questo è dovuto a vari fattori:

- Salvataggio e ripristino dei registri (quindi dello stato)
- Gestione della coda dei TCB pronti, in attesa ecc.
- Subito dopo un cambio di contesto avrò tanti *cache fault*, cambiare la tabella presente nella MMU ecc.

L'ultimo punto sarebbe molto ridotto se i thread fossero implementati a livello utente perché per il SO non esisterebbero i thread ma vedrebbe solo i processi.

4.12 Esempi vari

Gli esempi che seguono sono uguali identici a quelli visti per i processi. Al momento non c'è alcuna differenza tra thread e processo.

4.12.1 Esempio 1

Consideriamo un processore con registri speciali Stack Pointer (PC), Program Status Word (PS) e l'user-level Stack Pointer (SP), e due registri generali R1 ed R2. L'interrupt vector è nella memoria e il SO utilizza un singolo kernel stack (condiviso da tutti i thread).

Quando si riceve un'interruzione avvengono i seguenti passi:

- Sets kernel mode;
- Disable interrupts;
- Saves PC & PS & SP on the kernel stack
- Loads the new PC & PS from the interrupt vector
  - Consequently jumps to the interrupt handler in the kernel

Hardware

The IRET instruction:

- Enableinterrupts;
- Sets user mode;
- Restores PC, PS & SP from the kernel stack; (consequently jumps back to the address at which the RUNNING thread ad been interrupted in the past)

The interrupt handler:

- First saves the general registers on the kernel stack
- at the end restores the general registers from the kernel stack and executes IRET

Software

4.12.2 Esempio 2

Un thread T1 invoca una system call. Alla fine della system call rimane in stato di *RUNNING*.

1. Situazione iniziale (USER MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

base kernel SP	0FFF

2. Dopo l'interruzione (KERNEL MODE): si salvano i registri PC, PS e SP di T1 nel kernel stack e contemporaneamente si modificano questi registri rispettivamente con la prima istruzione dell'handler (presa dall'interrupt vector), la PS e col nuovo valore dello Stack Pointer (ora punta alla prima posizione ibera del kernel stack).

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	
0FFB	
0FFA	

registers	
PC	5000
PS	AA45
SP	0FFC
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

base kernel SP	0FFF

3. Prima dell'IRET (KERNEL MODE): l'handler come prima istruzione aveva salvato i registri generali R1 e R2, relativi a T1, nel kernel stack e il PC dell'handler sarà ad una certa posizione.

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	4500
0FFB	CD31
0FFA	

registers	
PC	5000+?
PS	AA45
SP	0FFA
R1	??
R2	??

4. Durante l'esecuzione dell'IRET all'indirizzo 5100 (KERNEL MODE): l'handler rimemorizza in R1 ed R2 i valori dei registri di T1 prendendoli dal kernel stack, lo Stack Pointer è aggiornato.

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	
0FFB	
0FFA	

registers	
PC	5100
PS	AA45
SP	0FFC
R1	4500
R2	CD31

5. Alla fine dell'IRET (USER MODE): vengono prelevati i valori di PC, PS e SP di T1 dal kernel stack e vengono reinseriti nei registri del processoro. Il thread continua la sua esecuzione.

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

4.12.3 Esempio 3

Come prima ma adesso la system call blocca T1 e viene messo T2 in stato di *RUNNING*.

1) Initial situation during the execution of SVC instruction (USER MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

base kernel SP	0FFF

2) After interrupt (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	
0FFB	
0FFA	

registers	
PC	5000
PS	AA45
SP	0FFC
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

base kernel SP	0FFF

3) After temporary storage of registers (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
0FFE	16F2
0FFD	2880
0FFC	4500
0FFB	CD31
0FFA	

registers	
PC	5000+?
PS	AA45
SP	0FFA
R1	??
R2	??

4) After storage of registers of T1 and restore of registers of T2 (KERNEL MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	A12C
0FFE	16F2
0FFD	A275
0FFC	25CC
0FFB	F012
0FFA	

registers	
PC	5000+?
PS	AA45
SP	0FFA
R1	??
R2	??

5) During extraction of IRET at address 5100 (KERNEL MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	A12C
0FFE	16F2
0FFD	A275
0FFC	
0FFB	
0FFA	

registers	
PC	5000+?
PS	AA45
SP	0FFC
R1	25CC
R2	F012

6) at the end of IRET (USER MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

registers	
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

4.13 Riassunto

Il programma principale è formato da tante procedure piccole allora il concetto di processo che le riprende tutte non rispecchia più questo modo di programmare. E' stato dunque pensato di considerare queste "procedurine" come dei sottoprocessi che sono stati chiamai *thread*. I primi SO consideravano solo i processi e quindi i thread dovevano essere gestiti a livello utente. Il linguaggio Java è stato il primo ad implementarli.

Java vedeva l'esecuzioni delle applicazioni come l'esecuzione di tante procedure ma il SO non li vedeva quindi quando facevo girare un programma Java, Java me lo implementava in thread ma doveva gestirli lui. Per il SO era un unico programma e quindi un unico processo. Il thread a livello utente, come un processo, ha bisogno di utilizzare e perdere l'uso del processore, di invocare system call ecc. Ogni thread avrà bisogno di un TCB ma siccome il SO non riconosce i thread, la TCB deve essere realizzata da parte del supporto di programmazione utilizzato (e.g. all'interno del proprio processo) mentre il kernel avrà solo la tabella dei PCB. Il mio ambiente di programmazione (Java) dovrà allocare, per ogni thread, un TCB, lo Stack, lo *stub* per collegarlo al processo principale ecc. Anche la gestione dello scheduling deve essere gestita dal sistema di programmazione (Java) con l'utilizzo di costrutti quali, ad esempio, la`thread_yield` per rilasciare il processore. Questi costrutti (sostanzialmente delle librerie) me li fornisce il linguaggio di programmazione perché altrimenti l'utente, se volesse realizzare lui i thread, se li dovrebbe scrivere. Se un thread richiede un I/O di dati che richiede l'intervento del kernel, questo non lo riconosce come un thread singolo ma come un unico processo e quindi blocca l'intero processo ed inoltre se il mio sistema è multiprocessore non posso eseguire più thread del solito processo.

I thread di un processo collaborano tra di loro per risolvere un unico problema e si devono scambiare informazioni. Ogni thread dovrà avere una propria memoria e quindi i TCB avranno le solite informazioni del PCB di un processo single-threaded. Lo Heap, i dati (variabili globali) e il codice di un processo saranno comuni a tutti i thread di quel processo. I vantaggi di avere i thread a livello utente è che non è necessario salvare lo stato di un thread a firmware e software quando avviene un cambio di contesto e non c'è bisogno di invocare system call, quindi di cambiare da user-mode a kernel-mode. Tutto questo perché avviene tutto a livello software interno e non c'è bisogno di scomodare il kernel. Inoltre i programmi implementati con user-kernel sono portatili perché possono girare su ogni SO che abbia o meno l'implementazione del thread a livello kernel. Gli svantaggi sono dovuti al fatto che se il SO non supporta i thread, se un thread si blocca, il kernel blocca tutto il processo quindi un maggior rallentamento.

Successivamente le aziende hanno deciso di investire soldi per implementare i thread anche a livello kernel. Quindi, adesso, non solo il SO supporta i thread a livello utente ma loro stessi sono realizzati a thread. La tabella dei TCB sarà all'interno del SO. La creazione, il cambio di contesto, il rilascio del processore ecc. vengono fatte con delle system call. In questo caso ovviamente se si blocca un thread di un processo, non si blocca il processo stesso. Avrò all'interno del kernel, la tabella del PCB i quali ognuno a sua volta avrà un puntatore alla tabella dei TCB dei propri thread.

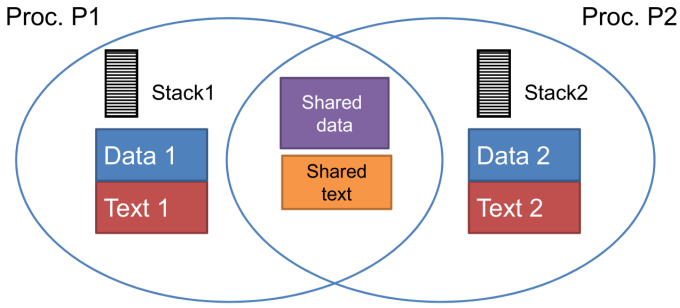
5 Accesso sincronizzato e oggetti condivisi

Vediamo ora i meccanismo che il SO mette a disposizione per coordinare i thread in modo tale che collaborino tra loro. Le procedure ed i programmi devono concorrere nell'acquisizione di risorse e cooperare tra di loro. Ci sono due modelli, in generale, per coordinare l'uso di risorse da parte di più thread: *ambiente globale* e *ambiente locale*.

5.1 Cooperation models

5.1.1 Ambiente globale

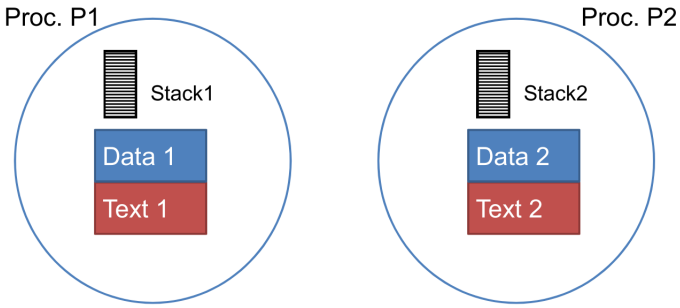
Nell'ambiente globale ho una sola copia dei dati condivisi allocati un blocco di memoria condivisa da più thread che cooperano tra loro.



Nel caso dei thread: ognuno ha il proprio stato *per-thread* (e.g. stack e registri del thread) ed entrambi condividono lo *shared state* (e.g. variabii condivise nello heap). I thread cooperanti leggono e scrivono nello *shared state*.

5.1.2 Ambiente locale

Non esiste il concetto di memoria condivisa: i thread si chiedono a vicenda i dati, passandosi una copia. Devo dunque stabilire un canale di comunicazione dove i thread si scambiano informazioni.



5.2 Challenges

5.2.1 Race conditions (Corsa critica)

L'esecuzione di un programma multi-threaded dipende dall'interleaving di accesso alla memoria condivisa tra i differenti thread. In effetti è come se si thread facessero una corsa ed il risultato dell'esecuzione del programma dipende da chi la vince.

Esempi:

Thread A	**Thread B**
`x = 1;`	`x = 2;`

Il risultato può essere `x = 1` oppure `x = 2`, dipende da quale thread vince o perde la corsa.

Thread A	**Thread B**
`x = y + 1;`	`y = y * 2;`

Supponendo `y = 12`, il risultato può essere `x = 13` o `x = 25`.

Thread A	**Thread B**
`x = x + 1;`	`x = x + 2;`

Il risultato è ovviamente `x = 3` ma purtroppo nemmeno in questo caso è così scontato perché in molti processori queste semplici istruzioni producono alcune istruzioni assembler e a seconda del loro ordine, il risultato cambia:

One Interleaving		**Another Interleaving**		**Yet Another Interleaving**	
load r1, x		load r1, x		load r1, x	
add r2, r1, 1			load r1, x		load r1, x
store x, r2		add r2, r1, 1		add r2, r1, 1	
	load r1, x		add r2, r1, 2		add r2, r1, 2
	add r2, r1, 2	store x, r2			store x, r2
	store x, r2		store x, r2	store x, r2	
final: x == 3		**final: x == 2**		**final: x == 1**	

5.2.2 Too Much Milk

Nonostante uno, in principio, potrebbe ragionare attentamente sui tutti i possibili casi di interleaving, nella pratica è difficile e porta ad errori.

Vediamo ora un problema che ci mostra un esempio di accesso interleaved ad uno stato condiviso: *due coinquilini condividono un frigorifero ed entrambi fanno in modo che in esso non manchi mai il latte. E' possibile che accada quanto segue*

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Assumiamo che le istruzioni siano esattamente eseguite in ordine di come sono scritte: né il compilatore né l'architettura le riordinano. E si cerca a tutti i costi di mantenere le seguenti proprietà:

- **Safety:** Una sola persona compra il latte.
- **Liveness:** Se c'è bisogno di comprare il latte, qualcuno lo compra.

Soluzione 1. L'idea di base è che un coinquilino lasci una nota sul frigo prima di andare in negozio.

```
if (milk==0) {        // if no milk
    if (note==0) {    // if no note
        note = 1;     // leave note
        milk++;       // buy milk
        note = 0;     // remove note
    }
}
```

Si possono presentare vari problemi, ad esempio

```
// Thread A
if (milk==0) {

    if (note==0) {
        note = 1;
        milk++;
        note = 0;
    }
}
```

```
// Thread B

if (milk==0) {
    if (note==0) {
        note = 1;
        milk++;
        note = 0;
    }
}
```

```
Oh no!
```

Soluzione 2. Nella soluzione 1, il coinquilino controlla la nota prima di settarla. Questo apre la possibilità che un coinquilino abbia già deciso di comprare il latte prima di notificare l'altro della decisione. Se usiamo due variabili per le note, un coinquilino può creare una nota prima di controllare l'altra nota, il latte e prendere la decisione di comprarlo.

Path A

```
noteA = 1;          // leave note
if (noteB==0) {     // if no note
    if (milk==0) { // if no milk
        milk++;     // buy milk
    }
}
noteA = 0;          // remove note
```

Path B

```
noteB = 1;          // leave note
if (noteA==0) {     // if no note
    if (milk==0) { // if no milk
        milk++;     // buy milk
    }               //
}                   //
noteB = 0;          // remove note
```

Il problema è che è possibile che il primo thread perda l'uso del processore dopo aver settato la nota e il secondo subentra e lascia la nota. A questo punto entrambi i thread non compreranno mai il latte.

Soluzione 3. In questa soluzione ci assicuriamo che almeno un thread determini sia se l'altro thread ha comprato il latte o meno, prima di decidere se comprare comprarlo lui o meno.

Path A

```
noteA = 1;          // leave note A
while (noteB==1) { // wait for no note
    ;               // spin
}
if (milk==0) {      // if no milk M
    milk++;         // buy milk
}
noteA = 0;          // remove note A
```

Path B

```
noteB = 1;          // leave note B
if (noteA==0) {     // if no note A
    if (milk==0) {  // if no milk
        milk++;     // buy milk
    }               //
}                   //
noteB = 0;          // remove note B
```

5.3 Locks

Le risorse comuni tipiche dei processi/thread sono a livello sotfware: variabili globali, statiche e heap del processo. Abbiamo visto che le risorse comuni devono essere accedute da un'entità alla volta ed il meccanismo per risolvere i problemi che ne derivano sono le *lock*(serrature/lucchetti).

Una *lock* è una variabile di sincronizzazione che fornisce la *muta esclusione* – quando un thread tiene una lock, nessun altro thread può tenerla. Un programma associa ogni serratura con un sottoinsieme dello shared state e richiede che un thread abbia la serratura per eccedere a quello stato. Un solo thread alla volta può accedere a quello shared state.

5.3.1 Locks: API e proprietà

Una lock permette la mutua esclusione fornendo due metodi: `lock.acquire()` e `lock.release()`. Questi metodi sono definiti come segue:

- Una lock può trovarsi in due stati: *BUSY* oppure *FREE*.
- Una lock è inizialmente nello stato di *FREE*.
- `lock.acquire()` attende finché la lock non è *FREE* e poi automaticamente setta la lock in *BUSY*.
- `lock.release()` setta la lock in *FREE*. Se ci sono operazioni di `acquire` in sospeso, questo cambio di stato, permette ad una di loro di procedere.

Nell'esempio Too Much Milk:

```
lock.acquire();
if (milk == 0) {      // if no milk
    milk++;           // buy milk
}
lock.release();
```

Proprietà formali. Una lock può essere definita più precisamente come segue. Un thread *tiene una lock* se è ritornata da una `acquire` più che da una `release`.

Una lock deve assicurare le seguenti tre proprietà:

1. **Mutua esclusione.** Una lock può essere tenuta da al più un thread.
2. **Progress.** Se nessun thread tiene una lock e ogni thread cerca di acquisirla, allora almeno un thread deve acquisirla.
3. **Attesa limitata.** Se un thread T cerca di acquisire una lock, allora il numero di volte che altri thread riescono ad acquisirla deve essere limitato prima che lo faccia T.

Definiamo come *sezione critica* una sequenza di codice che solo un thread per volta può eseguire. Gli oggetti condivisi sono allocati come tutti gli altri oggetti. Possono essere allocati dinamicamente nello heap (`malloc` o `new`) oppure staticamente nella memoria globale dichiarando variabili statiche nel programma.

Diversi threads devono avere la possibilità di accedere ad oggetti condivisi. Se gli oggetti condivisi sono variabili globali allora il codice di un thread per riferirsi ad uno di essi può utilizzare il nome globale dell'oggetto stesso; il compilatore calcola l'indirizzo corrispondente. Se gli oggetti condivisi sono stati allocati dinamicamente allora il thread deve utilizzare un puntatore per riferirsi ad uno di essi. In linea generale è meglio definire oggetti condivisi nello heap e non nello stack per motivi di sicurezza.

Se ho più variabili/oggetti condivise/condivisi se utilizzo un'unica lock per tutte, se un thread acquisisce la serratura per lavorare su una di esse, gli altri thread non hanno la possibilità di lavorare sulle altre. Per ovviare a questo è possibile assegnare ad ogni variabile condivisa una specifica serratura e quando il thread vuole lavorare su una di esse può chiamare ad esempio una `lock.acquire(nome_lock_variabile)`. Ovviamente dobbiamo stare molto attenti per evitare che ci siano delle situazioni di stallo.

Esempio di uso della lock:

```
tryget() {
  item = NULL;
  lock.acquire();
  if (nelem>0) {
      item = buf[front];
      front = (front++)%size;
      nelem – –;
  }
  lock.release();
  return item;
}
```

```
tryput(item) {
  lock.acquire();
  if (nelem < size) {
      buf[last] = item;
      last = (last ++)%size;
      nelem ++;
  }
  lock.release();
}
```

Initially: nelem = front = last = 0; lock = FREE;
size is buffer capacity

5.4 Variabili Condizione

Le *variabili condizione* forniscono, ad un thread, un modo per aspettare che un altro thread faccia delle operazioni. Ad esempio un thread per la simulazione del tempo metereologico potrebbe aspettare che un altro thread calcoli le temperature di ogni regione. In tutti i casi, vogliamo che un thread aspetti alcune azioni che modificano lo stato del sistema affinché possa fare ulteriori progressi.

Una *variabile condizione* è un oggetto di sincronizzazione che permette ad un thread di attendere efficacemente un cambiamento dello shared state che è protetto da una lock. Una variabile condizione ha tre metodi:

- **CV::wait(Lock *lock).** Questa chiamata *rilascia la lock* atomicamente e *sospende l'esecuzione del thread che la chiama*, piazzandolo in una lista di attesa ("condition varable's waiting list "). Più tardi, quando il thread chiamante è riabilitato, *riacquisisce la lock* prima di ritornare dalla chiamata della `wait`.

- **CV::signal().** Questa chiamata rimuove un thread dalla lista di attesa (noi vediamo la versione di Per Brinch Hansen in cui ne rimuove uno a caso) e lo marca come idoneo a poter essere eseguito (i.e. aggiunge il thread alla lista dei pronti). Se non ci sono thread nella lista di attesa, `signal` non ha effetto.

- **CV::broadcast().** Questa chiamate rimuove tutti i thread dalla lista di attesa e li marca come idonei a poter essere eseguiti. Se non ci sono thread nella lista di attesa, `broadcast` non ha effetto.

Tutti e tre i metodi devono essere chiamati solo quando un thread tiene la lock. Un thread che sta aspettando un certo cambiamento deve ispezionare lo stato della variabile condivisa interessata in un loop. Così facendo, se non è stata aggiornata chiama sempre una `wait` in modo da poter rilasciare la lock e far lavorare qualche altro thread se invece è stata aggiornata esce dal loop e fa le operazioni che deve fare. Similmente per la `signal`, l'unica ragione per un thread per usare una `signal` o una `broadcast` è quando ha cambiato un certo oggetto condiviso e questo potrebbe interessare a qualche thread in stato di attesa.

Le variabili condizioni sono state progettate attentamente per lavorare in squadra con le lock e lo stato condiviso.

- Una variabile condizione è *memoryless*.

  La variabile condizione, di per sé, non ha uno stato interno a parte che una coda di thread in attesa. Le variabili condizioni non hanno bisogno di un proprio stato perché sono sempre usate all'interno di oggetti condivisi che hanno un proprio stato.

  Se non ci sono thread nella lista di attesa, una `signal` o una `broadcast` non hanno effetto.

- `CV::wait` *rilascia atomicamente* la lock.

  Un thread chiama sempre una `wait` quando è in possesso di una lock. L'atomicità assicura che non ci siano separazioni tra il controllare lo stato di un oggetto condiviso, decidere di aspettare, aggiungere il thread in attesa nella coda e rilasciare la lock.

  Se un thread rilascia la lock prima di chiamare la `wait`, potrebbe mancare una signal o una broadcast e aspettare per sempre. Ad esempio consideriamo il caso in cui un thread T_1 controlli lo stato di un oggetto e decida di aspettare, rilasci la lock anticipatamente

senza mettersi nella lista dei thread in attesa. In quel preciso momento, T_2 anticipa T_1. T_2 acquisisce la lock, cambia lo stato dell'oggetto che T_1 aspettava e chiama la `signal`, ma la waiting list è vuota e quindi la `signal` non ha effetto. In fine T_1 viene eseguito di nuovo e si inserisce nella waiting list e sospende quindi la sua esecuzione. In questo caso T_1 sarà in stato di wait molto probabilmente per sempre.

Una volta che la `wait` rilascia la lock, qualsiasi thread potrebbe girare prima che la `wait` riacquisisca la lock dopo una `signal`. Nel frattempo, lo stato delle variabili potrebbe cambiare. Il codice quindi, non deve assumere che se una cosa era vera prima della `wait` allora rimarrà vera quando si ritorna dalla `wait`. L'unica assunzione che si dovrebbe fare dal ritorno della `wait` è che il thread possiederà la lock e che le normali invarianti che aveva all'inizio della sezione critica sono vere.

- Quando un thread in stato di attesa viene riabilitato tramite una `signal` o una `broadcast`, *potrebbe non essere eseguito immediatamente.*

  Quando un thread che sta aspettando viene riabilitato, viene spostato nella coda dei pronti dello scheduler con nessuna priorità speciale, e lo schedulatore potrebbe farlo rieseguire in qualsiasi momento.

I punti sopracitati implicano una cosa molto importante per i programmatori: *la* `wait` *deve essere chiamata all'interno di un loop.*

Siccome la `wait` rilascia la lock, e siccome non ci sono garanzie sulla atomicità tra la `signal` o la `broadcast` e il ritorno di chiamata della `wait`, non ci sono garanzie che sia mantenuto il medesimo stato di ciò che viene controllato. Perciò, un thread che è stato di attesa deve sempre attendere in un loop, ricontrollando lo stato finché non è vero il predicato desiderato. Ovvero:

```
...
while (predicateOnStateVariables(...))  {
   wait(&lock);
}
...
```

e non:

```
...
if (predicateOnStateVariables(...))  {
   wait(&lock);
}
...
```

5.4.1 Esempio chiarificatore

Esempio per chiarire perché non bastano le lock ma servono anche le variabili condizione.

Immaginiamo di avere una serie di thread consumatori che prendono i lavori da fare da una coda e li eseguono, ed un solo thread produttore che mette i lavori in coda. Siccome la coda è condivisa abbiamo bisogno di una lock. Ma se ci fosse solo questa per applicare la mutua esclusione tra i thread, ogni volta che viene eseguito un thread lavoratore, l'acquisisce e se non ci sono lavori da fare la rilascia e poi prova a riacquisirla e così via. Questo porta ad un utilizzo inutile della CPU. Più la CPU viene utilizzata dai thread consumatori e meno la può utilizzare il thread produttore. Inoltre, c'è una grande contesa per la lock. Quindi la cosa migliore da fare è di far aspettare un thread consumatore una certa variabile condizione e risvegliarsi soltanto quando è vera (ovvero che c'è del lavoro da fare).

5.4.2 Semantica Mesa vs. Semantica Hoare

La semantica delle variabili condizione vista fino ad ora è da attribuirsi a Per Brinch Hansen e che è stata poi utilizzata in "Mesa", un primo linguaggio di programmazione utilizzato dallo Xerox PARC, da qui il nome *semantica di Mesa*.

Esiste anche un'altra versione proposta da Tony Hoare, chiamata appunto *semantica di Hoare*. In questa versione, quando un thread chiama la `signal`, l'esecuzione di quel thread viene sospesa e la lock passa in mano ad uno dei thread in stato di attesa e che viene immediatamente riesumato (quindi eseguito). Dopo, quando il thread riesumato rilascia la lock, la lock ritorna in mano al thread che aveva chiamato la `signal` e la sua esecuzione continua.

5.5 Implementazione della sincronizzazione

Sia le lock che le variabili condizione hanno un proprio stato. Per le lock lo stato è rappresentato da *BUSY* o *FREE* e una coda di zero o più thread che aspettano che diventi *FREE*. Per le variabili condizione lo stato è rappresentato dalla coda dei thread che aspettano di essere segnalati. In entrambi i casi, lo scopo è di modificare atomicamente queste strutture dati. Inoltre le lock sono esse stesse delle risorse condivise, per assurdo, per proteggere delle risorse condivise abbiamo creato un meccanismo esso stesso condiviso. Analogamente le variabili condizione e i semafori (vedi sezione successiva). Quindi per garantire l'atomicità delle operazioni si utilizzano due primitive hardware:

- **Disabilitazione delle interruzioni.** In un uniprocessore possiamo rendere una sequenza di istruzioni atomica disabilitando le interruzioni.
- **Istruzioni atomiche di read-modify-write.** In un multiprocessore, disabilitare le interruzioni non è abbastanza. L'architettura fornisce delle istruzioni speciali per leggere ed aggiornare atomicamente una parola di memoria.

5.5.1 Implementazione Lock su Uniprocessori

In un uniprocessore, ogni sequenza di istruzioni di un thread appare atomica ad un altro thread se non avviene un cambio di contesto nel mezzo della sequenza. Quindi, in un uniprocessore, un thread può effettuare una sequenza di istruzioni atomicamente disabilitando le interruzioni (e trattenersi da effettuare chiamate a funzioni di libreria che potrebbero effettuare un cambio di contesto) durante la sequenza. Questo suggerisce un banale approccio per implementare le lock in un uniprocessore:

```
Lock::acquire() { disableInterrupts(); }

Lock::release() { enableInterrupts(); }
```

Questo approccio può funzionare nel kernel quando tutto il codice è presumibilmente scritto con attenzione e siamo certi che rilasci la lock in fretta ma non possiamo fidarci del codice di un utente e potrebbe disabilitare le interruzioni per molto tempo e monopolizzarlo.

Una soluzione più efficace deriva dall'osservazione che se una lock è *BUSY*, è inutile far girare il thread finché la lock è *FREE*. Invece dovremmo fare un cambio di contesto al prossimo thread pronto.

```
class Lock {
  private:
    int value = FREE;
    Queue waiting;
  public:
    void acquire();
    void release();
}

Lock::acquire() {
    TCB *chosenTCB;

    disableInterrupts();
    if (value == BUSY) {
        waiting.add(runningThread);
        runningThread->state = WAITING;
        chosenTCB = readyList.remove();
        thread_switch(runningThread,
                      chosenTCB);
        runningThread->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}
```

```
Lock::release() {
// next thread to hold lock
    TCB *next;

    disableInterrupts();
    if (waiting.notEmpty()) {
    // move one TCB from waiting
    // to ready
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
```

Se la lock è *BUSY* quando un thread tenta di acquisirla, il thread sposta il proprio TCB nella lista dei *sospesi per la serratura*. Il threah poi si sospende e si scambia col prossimo thread della lista dei pronti. Il thread rimane in sospensione finché un altro thread non chiamerà una `release`. Se un thread è in attesa per la lock, una chiamata alla `release` non setta il valore della lock in *FREE*. Invece, lo lascia *BUSY*.

Da notare che un thread può essere sospeso per due motivi: o perché attende che la lock sia *FREE* (questo avviene nella `release`) e quindi va nella lista dei sospesi per la serratura oppure perché sta aspettando che sia vero un certo predicato su una variabile condivisa (vedi sopra con la `wait`).

5.5.2 Implementazione Lock su Multiprocessori

Nei multiprocessori le interruzioni vengono gestite da un solo processore e quindi la disabilitazione delle interruzioni vale solo uno e gli altri vanno avanti. Per realizzare il solito meccanismo di

accesso in mutua esclusione delle variabili condivise, lo faccio fare dall'arbitraggio della memoria. Questo perché il valore della lock starà in memoria e se due processori cercano di arrivarci contemporaneamente, sarà l'arbitraggio della memoria che permetterà ad uno solo di leggere e scrivere e gli altri aspettano. Quello che ha la possibilità di leggere e scrivere deve farlo in un colpo solo dunque ci devono essere nei multiprocessori delle operazioni *read-modify-write*.

Queste istruzioni in un'unica istruzione macchina, quindi in maniera atomica, devono leggere un valore dalla memoria, ci lavora, lo modifica e poi ci riscrive sopra. Così facendo quando l'arbitraggio della memoria dà il via libera all'altro processore il valore è già stato modificato.

Il processore che non è riuscito ad avere accesso alla locazione di memoria rimane in *attesa attiva*: continua ad accedere a quella cella di memoria in loop. Si chiamano *spinlock*.

```
class SpinLock {
  private:
    int value = 0; // 0 = FREE; 1 = BUSY

  public:
    void acquire() {
        while (test_and_set(&value)) // while BUSY
            ; // spin
    }

    void release() {
        value = 0;
        memory_barrier();
    }
}
```

Un'implementazione a livello basso potrebbe essere la seguente:

```
// &spinLockValue is a memory cell containing a binary value: Free (0) or BUSY (1)

// TSL R, &spinLockValue :
    writes the content of &spinLockValue in R and writes BUSY (1) in &LockValue

SpinlockAcquire&Lockvalue) {
     Loop:          TSL R, & spinLockValue
                    CMP R, BUSY
                    JEQ Loop:
                    RET  // at this point &lockValue == BUSY!!!!
}

SpinlockRelease() {
    MOV #FREE, &spinLockValue     // this unlocks a thread in the loop, if any
}
```

E l'implementazione della lock potrebbe essere la seguente:

```
LockAcquire(){
   spinLock.Acquire();
   if (value == BUSY){
     waiting.add(current TCB);
     sched.suspend(&spinLock); *
   } else {
      value = BUSY;
      spinLock.Release();
}
* scheduler: marks thread as
    waiting; release spinlock;
    schedules next thread;
```

```
LockRelease() {
   spinLock.Acquire();
   if (!waiting.Empty()){
     thread = waiting.Remove();
     sched.makeReady (thread,
    &spinLock); *
   } else {
     value = FREE;
   }
   spinLock.Release();
}
* scheduler: marks thread as
  ready, put it in the ready list.
```

L'unica differenza da quella negli uniprocessori è che invece di abilitare e disabilitare le interruzioni, effettua la `spinLock.acquire()` e la `spinLock.release()`.

5.6 Semafori

E' una soluzione ideata da *Dijkstra* per unire le lock e le variabili condizione. E' un meccanismo un po' rigido perché metto insieme due cose ed impedisco all'utente di vederli come oggetti separati ma glieli vendo tutti insieme impacchettati (il valore della lock: BUSY o FREE; e la coda di quelli che aspettano). Il vantaggio è che se devo permettere a più entità di accedere ad una determinata risorsa, con la variabile condizione ci devo lavorare mentre i semafori me lo implementano loro. Un *semaforo* è una coppia: *valore* (lock: BUSY o FREE) e *coda* (variabile condizione), su cui posso eseguire due operazioni: la *P* e la *V*. I semafori sono definiti come segue:

- Un semaforo ha un valore NON negativo.
- Quando viene creato un semaforo, il suo valore può essere inizializzato a qualsiasi valore non negativo.
- Semaphore::P() attente finché il valore non è positivo. Poi, decrementa di 1 automaticamente il valore e ritorna.
- Semaphore::V() aumenta automaticamente di 1 il valore. Se ci sono thread che stanno attendendo in P, ne viene abilitato uno, che chiamerà la P, decrementerà il valore e ritornerà.
- Non sono permesse altre operazioni sui semafori; in particolare, nessun thread può leggere direttamente il valore corrente del semaforo.

Il valore del semaforo corrisponde effettivamente a quante entità possono utilizzare la risorsa ovvero entrare nella sezione critica. E' chiaro che se viene chiamata la V, non ha senso aumentare il valore di 1 e decrementarlo subito dopo quindi conviene: quando si effettua la V si va a vedere se c'è qualcuno in coda, se c'è lo risveglio senza toccare il valore altrimenti lo incremento di 1.

P e V sono delle system call e quindi realizzate dal kernel e sono operazioni atomiche: all'inizio della P e la V disabilito le interruzioni e alla fine riabilitarle. Tra la scelta di utilizzare lock+var.cond. e i semafori generalmente si utilizzano quest'ultimi ma sono del tutto equivalenti e possono essere implementati l'uno con l'altro.

Mentre le variabili condizione non hanno politica di risveglio (uno a caso), i semafori gestiscono la coda dei TCB in attesa con una politica FIFO.

5.6.1 P e V Implementazione multiprocessore

```
P(sem){
  spinLock.Acquire();
  disableInterrupts ();
  if (sem.value == 0){
    waiting.add(current TCB);
    suspend(&spinLock); *
} else {
     sem.value --;
     spinLock.Release();
     enableInterrupts ();
}
* suspends, invokes scheduler,
    context switch & enable
    interrupts
```

```
V(sem) {
  spinLock.Acquire();
  disableInterrupts ();
  if (!waiting.Empty()){
    thread = waiting.Remove();
    readyList.Append(thread);
  } else {
    sem.value ++;
  }
  spinLock.Release();
  enableInterrupts ();
}
```

5.6.2 Esempio

Semaphore Bounded Buffer

```
get() {                          put(item) {
  empty.P();                       full.P();
  mutex.P();                       mutex.P();
  item = buf[front]                buf[last] = item;
  front= (front+1) % size;         last = (last +1) % size;
  mutex.V();                       mutex.V();
  full.V();                        empty.V();
  return item;                   }
}
Initially: front = last = 0; size is buffer capacity
empty/full are semaphores (initialized to 0 and size)
Mutex is a semaphore initialized to 1
```

Il semaforo `empty` mi dice se c'è qualcosa nel buffer e realizza la variabile condizione. Mentre il semaforo `mutex` impedisce ad altri di "entrare dentro" e realizza la lock. Il semaforo `full` viene invece utilizzato da coloro che vogliono scrivere nel buffer e non supereranno la P se il buffer è pieno.

Il valore iniziale di `empty` deve essere 0 perché se si dovesse partire dalla `get` si andrebbe a leggere in un buffer in cui non c'è niente. Analogamente `full` dovrà essere inizializzato alla dimensione del buffer.

5.6.3 Implementare le variabili condizione con i semafori

TAKE 1

```
wait(lock) {
  lock.release();
  sem.P();
  lock.acquire();
}
signal() {
  sem.V();
}
```

TAKE 2

```
wait(lock) {
  lock.release();
  sem.P();
  lock.acquire();
}
signal() {
  if semaphore is not empty
     sem.V();
}
```

TAKE 3

```
wait(lock) {
  sem = new Semaphore;
  queue.Append(sem);   // queue of waiting threads
  lock.release();
  sem.P();
  lock.acquire();
}
signal() {
  if !queue.Empty() {
     sem = queue.Remove();
     sem.V();          // wake up waiter
  }
}
```

6 Multi-Object Synchronization

6.1 Risorse

Una *risorsa* è un qualunque oggetto (tipicamente dei dati tipo le lock, CPU, la memoria, stampante ecc.) necessario ai thread per poter lavorare. Le risorse possono essere:

- **Prerilasciabili.** Se la funzione di gestione può sottrarre questa risorsa ad un'entità prima che questo l'abbia effettivamente rilasciata. Esempio: processore, blocchi o partizioni di memoria.
- **Non prerilasciabili.** Se la funzione di gestione non può sottrarla al processo al quale è assegnata. Esempio: stampanti.

Ci sono due tipi di problemi associati all'utilizzo di più risorse:

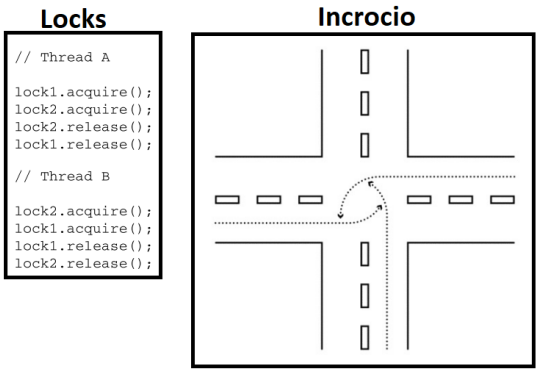
- **Starvation.** Significa *morte per fare* e avviene quando uno o più thread non riescono ad andare avanti perché la risorsa di cui hanno bisogno non riescono a prenderla mai. Ce l'ha sempre qualcun altro.
- **Deadlock.** Avviene quando ogni thread dell'insieme è in attesa che si liberi una risorsa che solo un altro thread dell'insieme può dargli.

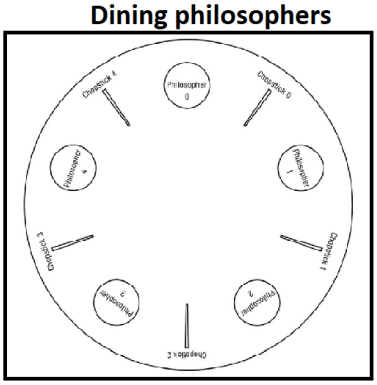
Tutte le risorse devono essere *bloccanti*, ovvero che un thread non può prendere una risorsa che non è disponibile. Se un thread ha una risorsa e ha bisogno di un'altra, la prima non la cede per poter prendere l'altra: finché ce l'ha è sua a meno che non la rilasci esplicitamente.

6.2 Deadlock

Come detto precedentemente una *deadlock* è un attesa ciclica tra un insieme di thread, dove ogni thread attende, in un ciclo, che l'altro faccia qualche operazione.

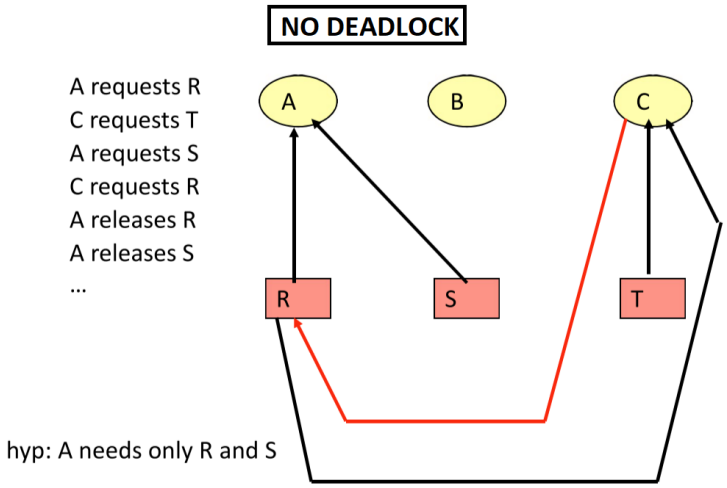
Esempi:

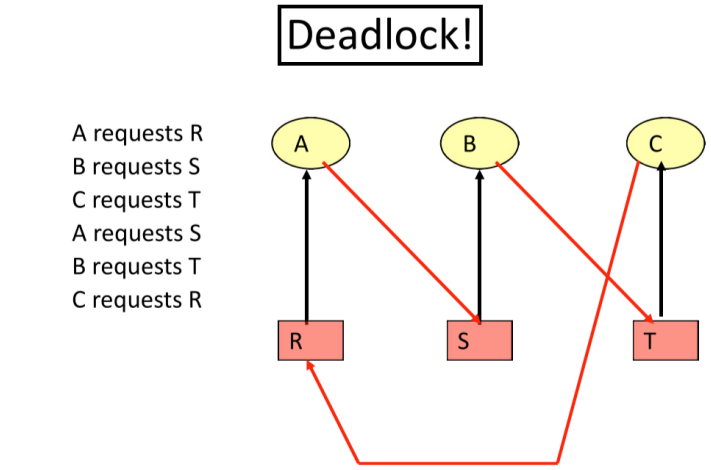




Ci sono quattro condizioni necessarie affinché si verifichi una deadlock:

1. **Risorse limitate.** Ci sono un numero finito di thread che possono simultaneamente utilizzare una risorsa.
2. **No prerilascio.** Una volta che un thread acquisisce una risorsa, non è possibile revocargliela fino a che il thread stesso non la rilascia (nell'esempio dell'incrocio è come se nessun macchina facesse retromarcia).
3. **Wait while holding.** Un thread tiene una risorsa mentre ne chiede un'altra.
4. **Attesa circolare.** C'è un insieme di thread in attesa in cui ogni thread attende una risorsa tenuta da un altro.





Per gestire uno stallo possiamo:

- Ignorare il problema, fingendo che non esista.
- Identificarlo e risolverlo.
- Adottare una prevenzione statica: che non si presenti mai.
- Adottare una prevenzione dinamica (Algoritmo del Banchiere).

6.2.1 Identificarlo e risolverlo

Si costruisce il grafo come negli esempi precedenti, si trovano i cicli e si aggiustano:

- **Rimuovendo un arco oppure un nodo (uccidendo un thread).** Il thread viene rimesso in esecuzione da capo e quindi tutte le risorse che aveva vengono rilasciate (è come se fosse un rollback iniziale).
- **Rollback**. Si inserisce nei programmi dei checkpoint, in cui tutto lo stato dei processi (memoria, dispositivi e risorse comprese) viene salvato su un file. Uno o più processi coinvolti si riportano ad uno dei checkpoint salvati, con conseguente rilascio delle risorse.

6.2.2 Prevenzione statica

Per farlo si cerca di eliminare una delle quattro condizioni per la verifica del deadlock:

- **Lock ordering (evitare attese circolari).** Si chiedono sempre con lo stesso ordine: se ad esempio ho `lock1`, `lock2`, `lock3`, se ho bisogno sia della 1 che della 3 chiedo prima la 1 e poi la 3.
- **No "wait while holding".** Si progetta un sistema affinché un thread rilasci le proprie risorse se è in stato di attesa.
- **Richiedere tutte le risorse.** Ogni thread sa in anticipo le risorse di cui avrà bisogno quindi fa richiesta per tutte e finché non ce l'ha tutte non va avanti. Se c'è qualche risorsa non accessibile *"wait without holding"*.
- **Numero illimitato di risorse (Spooling).** SI virtualizza la risorsa. Esempio: a regola per mandare una seconda stampa dovremmo attendere che la stampante abbia finito ma in realtà facciamo l'ordine che verrà messo in una coda e gestito da un processo che gestisce la stampante.

6.2.3 Prevenzione dinamica - Algoritmo del Banchiere

Le ipotesi che facciamo è che abbiamo più risorse di vario tipo. Identifichiamo con:

- **Molteplicità** M : numero di copie disponibile per ogni risorsa.
- **Disponibilità** D : numero di risorse che sono correntemente disponibili.

E possiamo avere due tipi di richieste:

- **Richiesta singola.** Ovvero ogni thread chiede una sola copia di un tipo di risorsa alla volta.
- **Richiesta multipla.** Un processo può chiedere k copie di uno stesso tipo di risorsa in un colpo solo:
  - Se $k \leq D$ allora gli assegniamo le risorse.
  - Se $k > D$ allora non gliele assegniamo ed il thread attende.

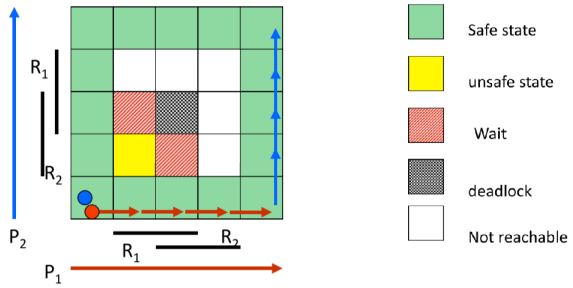
L'**algoritmo del banchiere** è stato idealizzato da *Dijkstra* nel 1965. Mettendosi nei panni di un banchiere (che identifica il gestore delle risorse), se dessi soldi a chiunque me li chiedesse potrei ritrovarmi senza soldi e non poterli dare più a nessuno, l'algoritmo del banchiere dice sostanzialmente di dare i soldi ad uno solo, non accettare nessun'altra richiesta ed aspettare che me li restituisca. A quel punto posso accettarne un'altra. Affinché funzioni il tutto, ogni thread deve dichiarare inizialmente di quante copie necessita durante tutta la sua vita (se non lo sa allora non si può applicare l'algoritmo). Andiamo a vedere se, dando una certa risorsa ad un thread, ho un modo per fallire ovvero per non andare in deadlock.

Il sistema può essere in diversi stati. Uno *stato* è identificato da quali risorse sono assegnate a certi thread. Lo stato può essere in:

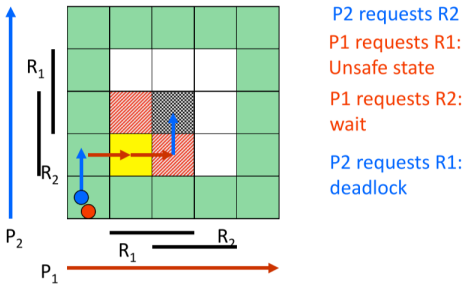
- **Stato sicuro.** Se nello stato corrente ho almeno un modo per evitare il deadlock facendo terminare tutti i thread.
- **Stato insicuro.** Se c'è almeno una sequenza di richieste future che potrebbero portare ad una deadlock indipendentemente dal quale sia l'ordine.
- **Stato di deadlock.** Se il sistema è in deadlock.

Esempi:

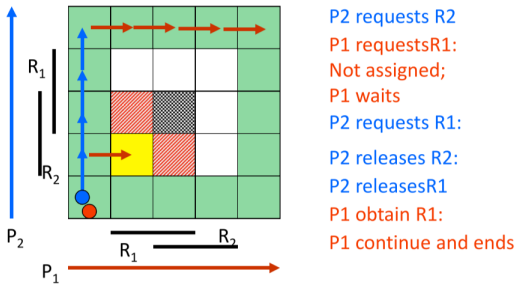
1) System that evolves in safe states



2) A system in an unsafe state can reach a deadlock



3) The banker does not accept requests that lead to unsafe states



In sintesi:

- Si parte da uno stato sicuro ed ogni thread presenta, per ogni tipo di risorsa, di quante copie avrà bisogno.
- Ad una richiesta di un thread A il banchiere controlla se l'eventuale assegnazione delle risorse richieste porta sempre ad uno stato sicuro, simulando quanto segue:
  1. Blocca tutti i thread tranne il richiedente e soddisfa le sue richieste (quando finisce si riavrà le risorse che teneva).
  2. Sceglie un altro thread (magari quello che richiede meno risorse) e fa la solita cosa di prima. Per ogni thread.
  3. Se con l'ultimo raggiunge uno stato sicuro allora si può concedere le risorse richieste dal thread A.

Esempio 1:

Un sistema con processi P1, P2, P3, P4 e risorse dei tipi R1, R2, R3, R4, rispettivamente di molteplicità [4, 5, 5, 5], i processi dichiarano inizialmente le seguenti esigenze:

ESIGENZA INIZIALE				
	R1	R2	R3	R4
P1	2	3	1	1
P2	2	1	1	2
P3	0	1	0	2
P4	0	2	5	2

Stato (sicuro) raggiunto dal sistema al tempo *t:*

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	1	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

	R1	R2	R3	R4
P1	0	2	0	0
P2	0	1	0	0
P3	0	0	0	2
P4	0	0	3	0

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
0	1	1	0

Verifica dello stato sicuro:
1) l'esigenza di P2 può essere soddisfatta e P2 può terminare

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	2	1	1
P2	-	-	-	-
P3	0	1	0	0
P4	0	2	2	2

	R1	R2	R3	R4
P1	0	2	0	0
P2	-	-	-	-
P3	0	0	0	2
P4	0	0	3	0

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
2	1	2	2

Verifica dello stato sicuro:
1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
2) l'esigenza di P3 può essere soddisfatta e P3 può terminare

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	1	1	1
P2	-	-	-	-
P3	-	-	-	-
P4	0	2	2	2

	R1	R2	R3	R4
P1	0	2	0	0
P2	-	-	-	-
P3	-	-	-	-
P4	0	0	3	0

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
2	2	2	2

Verifica dello stato sicuro:

1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
2) l'esigenza di P3 può essere soddisfatta e P3 può terminare
3) l'esigenza di P1 può essere soddisfatta e P1 può terminare

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	0	2	2	2

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	0	0	3	0

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
4	3	3	3

Verifica dello stato sicuro:

1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
2) l'esigenza di P3 può essere soddisfatta e P3 può terminare
3) l'esigenza di P1 può essere soddisfatta e P1 può terminare
4) l'esigenza di P4 può essere soddisfatta e P4 può terminare

STATO SICURO !

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	-	-	-	-

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	-	-	-	-

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
4	5	5	5

7 Scheduling

Per *scheduling* si intende "l'allocazione di risorse nel tempo". In informatica si intende "l'allocazione del processore nel tempo". Ovviamente i thread non utilizzano solo il processore ma anche la memoria, il disco ecc. ma già considerando solo il processore molti problemi sono NP completi.

7.1 Definizioni

- **Task/Job.** Una richiesta utente. Quando parliamo di scheduling utilizzeremo il termine *task* anziché thread o processo. Questo perché essi stessi possono essere composti da più task utente.
- **Throughtput.** Quanti task possono essere completati per unità di tempo.
- **Latenza/Tempo di risposta.** Quanto tempo impiega un task per terminare.
- **Predicibilità.** Quanto la performance è consistente nel tempo.
- **Equità.** Uguaglianza nel numero e nella tempistica delle risorse date ad ogni task.
- **Starvation.** (Fame) L'assenza di progressi per un task dovuta al dare sempre le risorse ad un task con priorità maggiore.
- **Workload.** Insieme di task che devono essere eseguiti.
- **Preemptive scheduler.** (Scheduler con pre-rilascio) Un processo viene temporaneamente interrotto al fine di permettere l'esecuzione di un altro processo.
- **Work-conserving.** Se una risorsa è libera e alcuni la vogliono allora si da ad almeno uno di loro. (A seconda di che risorse si utilizzano e che problema di scheduling abbiamo, potrebbe essere più conveniente non assegnare la risorsa per un po' di tempo anche se qualcuno la sta richiedendo).

Noi considereremo solo lo scheduler con pre-rilascio, work-conserving e solo in architettura uniprocessore.

7.2 First-In-First-Out (FIFO)

Non è preemptive. Forse la politica di scheduling più semplice possibile è la *FIFO*: ogni task viene fatto nell'ordine in cui arrivano. Quando cominciamo a lavorare su di un task, non ci fermiamo finché non è completato o non rinuncia al processore.

Vantaggi:

- Overhead minimizzato: prendere dalla testa della lista e metterne un altro in coda.
- Fairness: ogni task aspetta il proprio turno.

Svantaggi:

- Se un task richiede molto più tempo di altri, questi devono aspettare che finisca: la latenza media si prolunga di molto.

7.3 Shortest Job First (SJF)

Non è preemptive. Quando il processore si libera viene scelto il task col minor numero di tempo per essere completato. In generale non sappiamo quanto impiega un task per essere eseguito quindi questa politica è solo sperimentale.

Vantaggi:

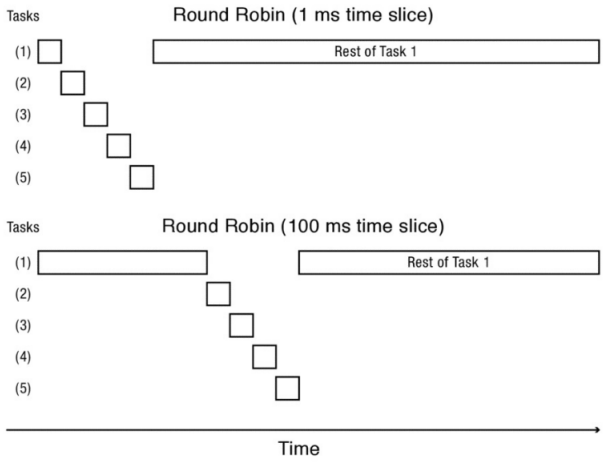
- Minimizza la latenza media.

Svantaggi:

- Rischio di starvation (No fairness): se un certo numero di task corti arrivano, quelli più lunghi potrebbero non essere mai completati.

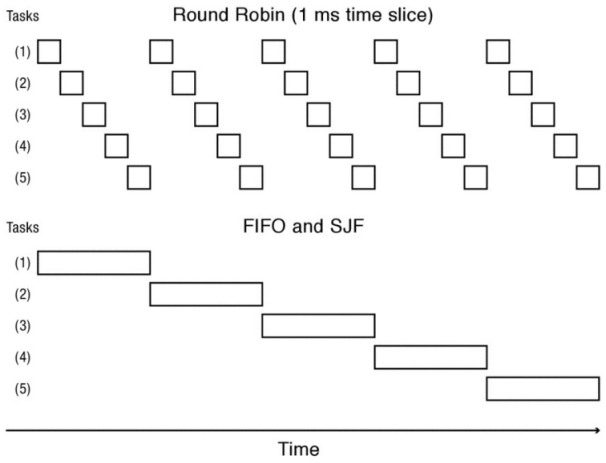
7.4 Round Robin

Nella politica *round robin* i task fanno a turno nel prendere il processore per una quantità limitata di tempo a meno che non si sospenda prima o termini prima. Lo scheduler assegna il processore al primo task della lista dei ready e setta il *quanto di tempo*: scaduto il quanto si solleva un interruzione da timer. Dunque una volta scaduto, se il task non è stato completato, quest'ultimo viene pre-rilasciato e il processore viene dato al prossimo task nella ready list. Il task pre-rilasciato viene messo nella coda della lista.

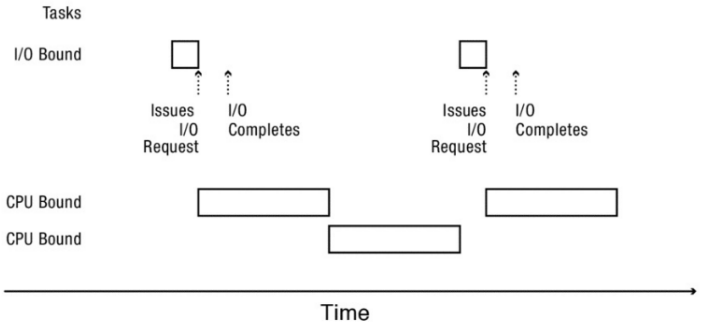


Se prendo il quanto di tempo infinito (o almeno quanto il task più lungo) finisco per essere FIFO, se lo prendo troppo corto il processore spenderebbe tutto il tempo a fare cambi di contesto (overhead).

Un caso in cui la Round Robin si "comporta male" avviene se si presentano, ad esempio, 5 task lunghi 5ms ognuno. Infatti in questo caso la politica FIFO sarebbe ideale:



Un'altra situazione in cui la politica Round Robin non è delle migliori è quando si alternano task **I/O-bound**(si bloccano spesso per input-output ma richiedono poco utilizzo del processore) e task **CPU-bound**(pochissime volte input-output ma poi utilizzano molto il processore). Infatti anche se si task I/O-bound finirebbero presto, ogni volta che si bloccano per operazioni di input-output vengono messi in fondo alla lista dei ready e i task CPU-bound occupano il processore:



Vantaggi:

- No starvation.
- Fairness.

Svantaggi:

- Quelli visti precedentemente.

7.5 Max-Min Fairness

L'idea è quella di *massimizzare la minima allocazione data ad un task.* In altri termini significa: schedulare iterativamente il task più corto poi dividere il tempo rimanente utilizzando la stessa tecnica *Max-Min.*

Esempio:

Tasks	**Richiesta CPU**
Task1	20%
Task2	25%
Task3	40%
Task4	50%

1. $\frac{100\%}{4} = 25\%$ da dare ad ogni task.
2. Rimane 5% dal *Task1* e si divide tra gli altri $\frac{5\%}{3} = 1.66..\%$:

   Task1 20%

   Task2 26.66..%

   Task3 26.66..%

   Task4 26.66..%

3. Rimane 1.66..% dal *Task2* e si divide tra i restanti $\frac{1.66..\%}{2} = 0.84\%$. Dunque il risultato finale sarà:

   Task1 20%

   Task2 25%

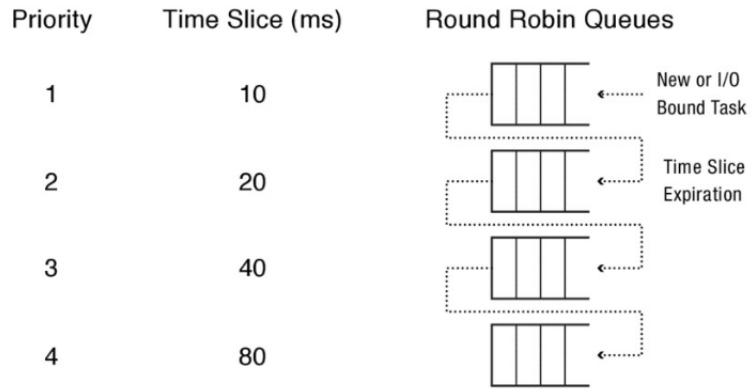
   Task3 27.5%

   Task4 27.5%

7.6 Multi-Level Feedback Queue (MFQ)

E' l'algoritmo di scheduling utilizzato oggigiorno da quasi tutti i sistemi operativi (Linux, MacOS, Windows). Non è ottimo ma è un buon compromesso tra tutte le metriche richieste dallo scheduler. *MFQ* è un'estensione del *Round Robin.* Invece di avere una singola coda, MFQ ha molte code Round Robin, ognuna con un differente livello di priorità e quanto di tempo. I task con alta priorità (numero più basso) pre-rilasciano quelli con priorità più bassa (numero più alto), mentre i task di medesima priorità sono schedulati Round Robin. Inoltre, i task di maggior priorità hanno un quanto di tempo più corto di quelli con priorità più bassa.

I task sono spostati in modo da favorire quelli più corti da quelli più lunghi. Un nuovo task entra nella coda di priorità più alta. Ogni volta che un task termina il proprio quanto di tempo, scende la gerarchia; ogni volta che un task rilascia volontariamente il processore perché sta aspettando un'operazione di I/O, rimane nel solito livello (o promosso).



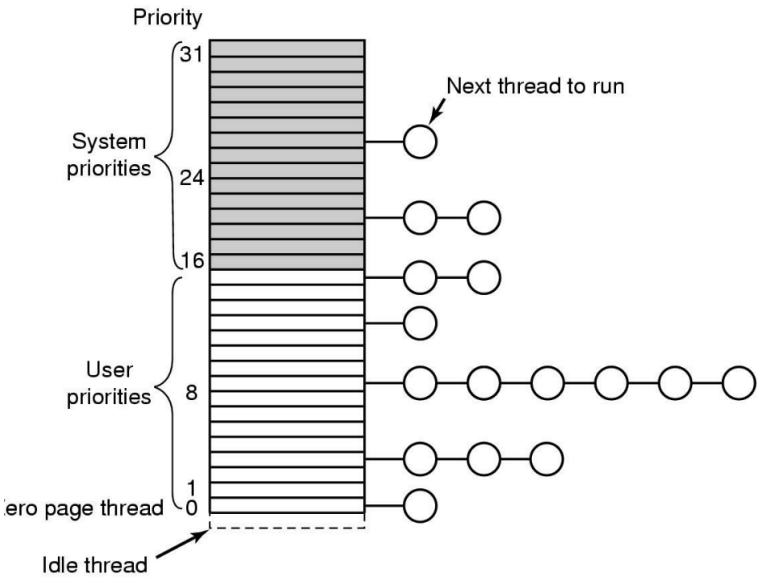
Un nuovo task CPU-bound inizialmente avrà priorità maggiore (perché è nuovo) ma terminerà presto il suo quanto di tempo e scenderà di livello e poi a quello dopo. Invece un task I/O-bound che necessità di poche operazioni sarà quasi sempre schedulato in fretta tenendo il disco occupato. I task CPU-bound hanno un lungo quanto di tempo per minimizzare l'overhead per il cambio di contesto.

L'algoritmo descritto fino ad ora non è sempre ottimo per quanto riguarda starvation e fairness: se ci sono troppi task I/O-bound (quindi rimangono sempre nel solito livello), quelli CPU-bound rischiano di non ottenere mai il processore. Per evitare ciò, MFQ monitora ogni processo affinché riceva la "giusta parte" di risorse. Ad ogni livello, Linux mantiene due code – i task che hanno già raggiunto la loro "giusta parte" sono schedulati solo se ogni altro task del medesimo livello hanno ricevuto la propria "giusta parte".

Periodicamente, qualsiasi task che riceve meno della propria "giusta parte" verrà promosso e qualsiasi task che ne riceve di più verrà degradato.

7.6.1 Esempio: Windows scheduler

Windows ha 32 code: maggiore è il numero, maggiore è la priorità. Le prime 16 riservate al SO, le altre all'utente.



L'*Idle thread* viene messo in esecuzione quando le code sono vuote (non c'è nessun task da mettere in esecuzione). Ha priorità -1 ed è semplicemente un ciclo di `nop`.

Un nuovo thread utente nasce con priorità 8. La priorità diminuisce ogni qualvolta utilizza l'intero quanto di tempo. La priorità aumenta se:

- viene riattivato dopo un'operazione di I/O:
  - +1 se era un'operazione su disco,
  - +6 sulla linea seriale,
  - +8 sulla tastiera o su audio card
  - ...
- viene riattivato dopo aver atteso su una *mutex/semaforo*:
  - +1 se in background,
  - +2 se in foreground.
- non viene messo in esecuzione per una certa quantità di tempo: la priorità diventa 15 per due interi quanti di tempo.

Quando una finestra viene messa in foreground, il quanto di tempo aumenta.

7.7 Sommario

FIFO è semplice e con poco overhead ma può avere una latenza scadente se i thread hanno tempi di esecuzione diversi.

Se dobbiamo schedulare solo il processore, lo SJF è ottimo in termini di latenza media mentre va abbastanza male in termini di varianza.

In generale, Round Robin va abbastanza bene perché tende ad approssimare la SJF se i thread hanno tempi di esecuzione abbastanza diversi tra loro. Se i tempi sono simili allora la Round Robin è scadente da un punto di vista di ritardo (latenza). Va male con thread misti: CPU-bound e I/O-bound. Però Round Robin e Max-min fairness evitano la starvation.

Quello utilizzato oggigiorno è il MFQ perché va abbastanza bene.

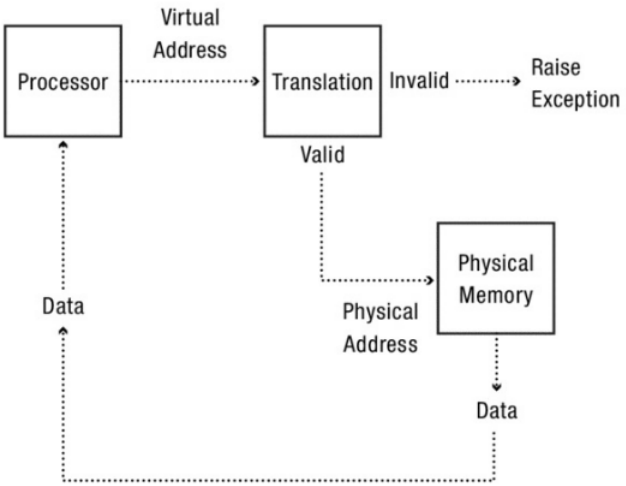
8 Address Translation

8.1 Address Translation Concept

La traduzione degli indirizzi sembrerebbe una semplice funzione. Il traduttore prende ogni riferimento a istruzioni o dati generati da un processo, controlla se l'indirizzo è legale e lo converte in un indirizzo fisico che può essere usato per fetchare o memorizzare istruzioni o dati. La traduzione è solitamente implementata ad hardware e il kernel configura l'hardware per compiere i propri obiettivi.

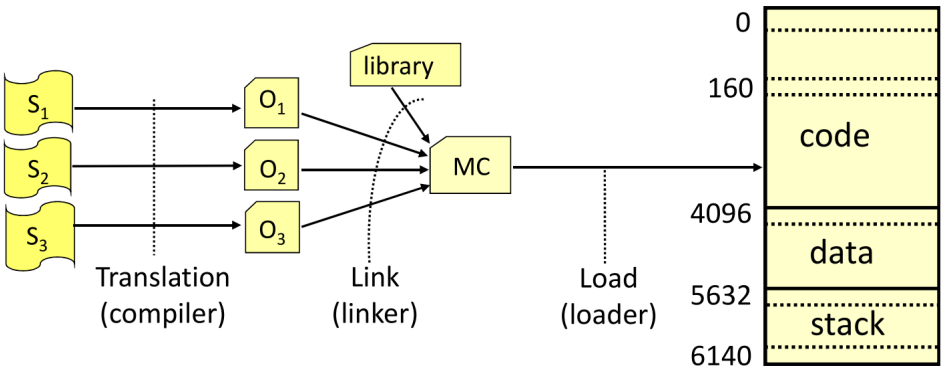
Qualsiasi sia la sua implementazione, ciò che vogliamo raggiungere è:

- **Protezione della memoria.** Vogliamo poter limitare l'accesso di una regione di memoria di un determinato processo.
- **Condivisione della memoria.** Vogliamo che più processi possano condividere determinate regioni di memoria.
- **Collocazione flessibile della memoria.** Vogliamo che il SO sia flessibile nel posizionare un processo ovunque nella memoria fisica.
- **Indirizzi sparsi.** Molti programmi hanno molte regioni di memoria dinamiche che possono cambiare di dimensione nel corso della loro esecuzione: lo heap, lo stack ecc. Gli indirizzi fisici in cui piazzo il processo non necessariamente sono contigue.
- **Efficiente ricerca a runtime.** La traduzione di un indirizzo avviene ogni volta che fetchamo un'istruzione e carichiamo o memorizziamo dati. Sarebbe poco pratico se la ricerca di uno di essi prendesse più tempo dell'esecuzione stessa di un'istruzione.
- **Compact translation tables.** Vogliamo inoltre che l'overhead per la traduzione sia minimo.
- **Portabilità.** Architetture hardware differenti implementano diversamente la traduzione. Un SO deve essere hardware-indipendente.



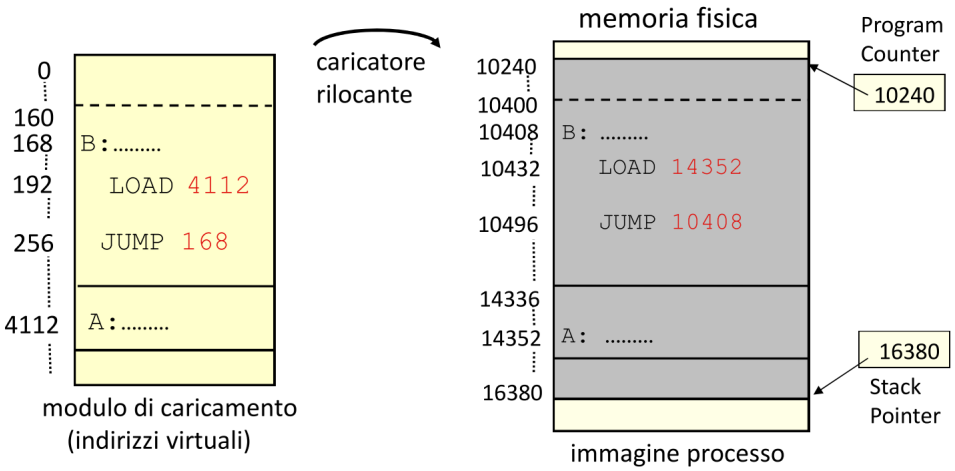
Un processo vede la propria memoria e utilizza i propri indirizzi. Li chiameremo *indirizzi virtuali*, perché non corrispondono necessariamente a quelli utilizzati effettivamente. Al contrario, per il sistema ci sono solo *indirizzi fisici* – vere locazioni di memoria. Il meccanismo di traduzione converte gli indirizzi virtuali in quelli fisici.

I sorgenti vengono compilati in moduli oggetto e poi, tramite il *linker*, vengono concatenati in un *modulo caricabile*(l'eseguibile) insieme alle librerie di sistema. L'immagine dell'eseguibile partirà sempre dall'indirizzo 0 (indirizzo virtuale). Il *loader* avrà il compito di caricarlo in memoria.



8.1.1 Rilocazione statica

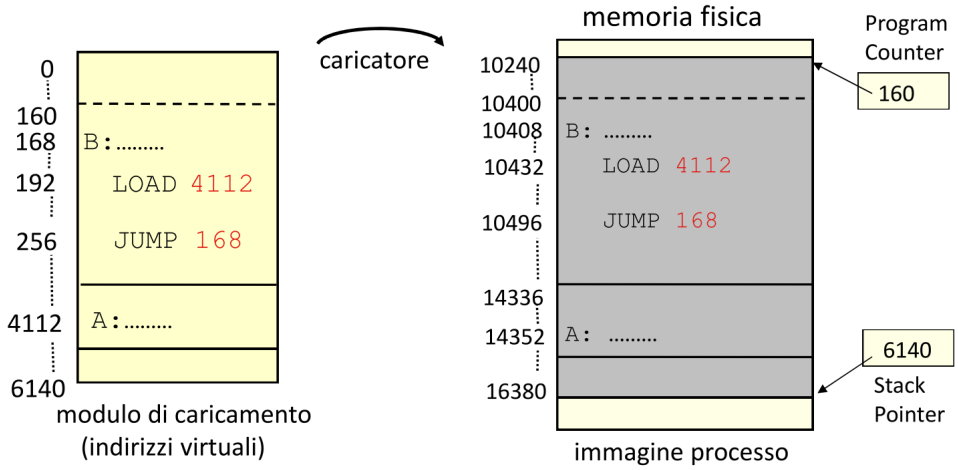
Alcuni sistemi operativi necessitano del *caricatore rilocante* che aggiusta gli indirizzi dell'eseguibile prima che venga eseguito. I SO che lo necessitano sono quelli nei quali un programma non è sempre caricato nella stessa locazione di memoria e i puntatori sono indirizzi assoluti invece che relativi (offset) alla base.



Solitamente non si utilizza perché ogni qualvolta si carica il programma dobbiamo convertire tutti i riferimenti ad indirizzi virtuali in indirizzi fisici.

8.1.2 Rilocazione dinamica

Solitamente si utilizza questa. Gli indirizzi virtuali non vengono tradotti in fisici (al contrario di quella statica). La traduzione avviene durante l'esecuzione e ogni qualvolta ci servono vengono tradotti. Abbiamo bisogno dunque, di un meccanismo hardware che mi fa la traduzione.



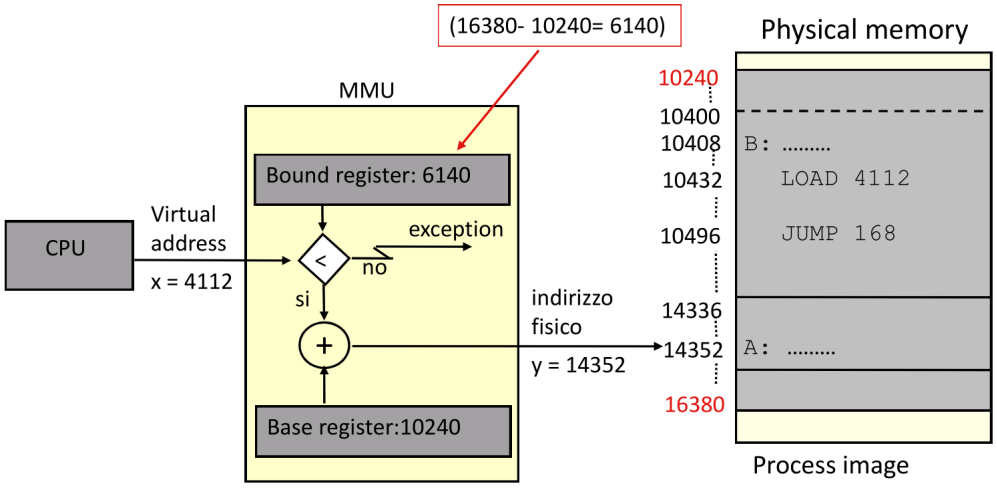
Il processore lavora con indirizzi virtuali, il meccanismo hardware controlla se è un indirizzo legale ed eventualmente lo traduce in indirizzo fisico. Dopodiché si va in memoria a prelevare il dato e lo si passa al processore (fetch). Se non è legale, generalmente, si uccide il processo.

8.2 Virtual Base and Bound

Continuando a vedere ogni processo come una cosa monolitica: ogni volta che si carica in memoria viene posizionato in locazioni differenti ma tutto il processo risiede in parole continue. La rilocazione dinamica vista precedentemente la implementiamo salvandoci l'indirizzo di partenza del processo(*Base*) in un registro e si somma a tutti gli indirizzi virtuali del processo.

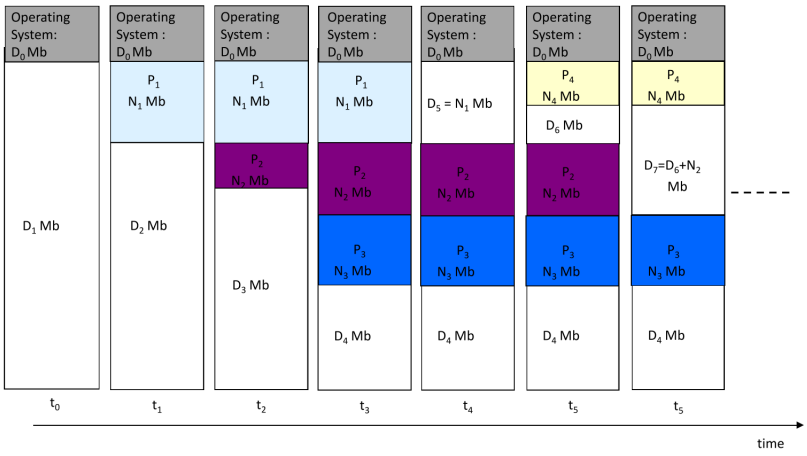
Per verificare se un processo sta accedendo alla propria regione di memoria, ci salviamo la dimensione(*Bound*) del processo e se un indirizzo virtuale è minore di questo numero si solleva un'eccezione.

La traduzione degli indirizzi viene svolta dalla *MMU* nella quale risiedono i due registri *Base* e *Bound* Entrambi i registri risiedono nel TCB o PCB.



8.2.1 Partizionamento e frammentazione

Si parla di *partizioni variabili* perché la memoria è partizionata in zone, occupate dai processi, di dimensione diversa.



Il problema sul dove piazzare un nuovo processo si chiama *Problema del Bin Packing* ed è NP-completo. Alcuni algoritmi possibili:

- **First-fit.** Tra tutte le partizioni libere, si piazza nella prima che è grande abbastanza per contenere il processo.
- **Best-fit.** Tra tutte le partizioni libere, si piazza nella più piccola che può contenere il processo.

Questo tipo di tecnica conduce ad avere due tipi di problemi:

- **Frammentazione interna.** Se vengono utilizzate partizioni di dimensione fissa, ogni volta che alloco un processo, avrò una parte di memoria inutilizzata o non necessaria.
- **Frammentazione esterna.** Si presenta quando i frammenti di memoria tra le partizioni sono troppo piccoli per contenere un nuovo processo.

Questo problema si presenta soprattutto sul disco fisso e spesso viene *deframmentato*: si spostano tutti i file in modo che siano contigui.

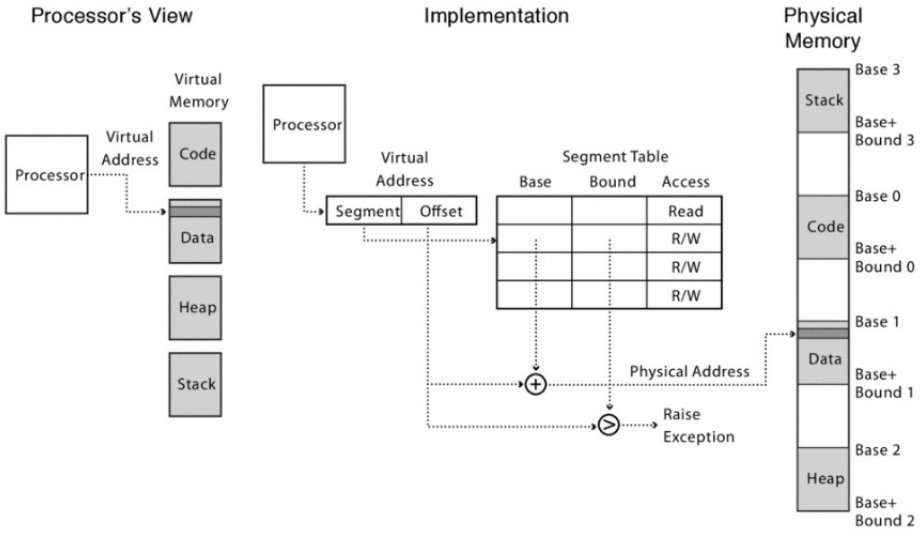
Sommario *Virtual Base and Bound*:

- **Pro**
  - molto veloce,
  - molto semplice,
  - per spostare un processo basta cambiare valore ai registri: *Base* e *Bound*.
- **Contro**
  - non ci sono controlli all'interno del processo: un processo potrebbe riscrivere il proprio codice,
  - non si possono condividere codice/dati con altri processi,
  - lo stack e lo heap non possono aumentare di dimensione.

8.3 Segmentation

Lo svantaggio del *Base and Bound* è che vediamo il processo come un'entità monolitica quando in realtà è composto da oggetti di natura diversa: il codice, i dati, lo stack e lo heap.

Con la segmentazione, invece di avere una sola coppia di registri (base e bound) per processo, abbiamo un array di coppie di registri, chiamato *Segment Table*. Ogni entry dell'array controlla un segmento dello spazio virtuale. La memoria fisica di ogni segmento è memorizzata in maniera contigua ma segmenti diversi possono risedere in locazioni differenti.



I bit più significativi dell'indirizzo virtuale indicizzano l'array, il resto (offset) viene aggiunto alla base, formando così l'indirizzo fisico, e viene controllato che non superi il limite (bound). In più, il SO può assegnare permessi differenti a differenti segmenti (e.g. solo lettura per il codice e lettura-scrittura per i dati).

Se un programma tenta di leggere o scrivere in porzioni che non gli appartengono viene sollevata un'eccezione. In UNIX si chiama *segmentation fault.*

Oltre che semplice da implementare e da gestire, la memoria segmentata è sia potente che ampiamente usata (con alcune migliorie descritte successivamente).

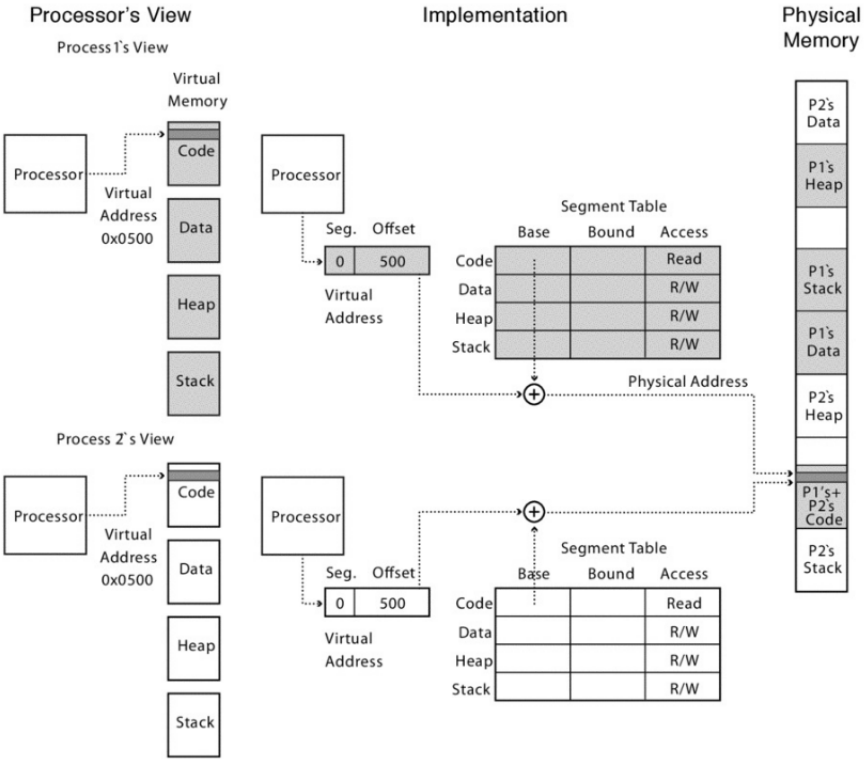
La stessa tabella sarà memorizzata in memoria ed avrò due registri base e bound. Con i segmenti, il SO permette la condivisione di porzioni di memoria tra processi diversi ma di tenere altre regioni protette. Ad esempio, due processi potrebbero condividere il segmento di codice settando un'entry nella *Segment Table* che punta alla solita regione di memoria fisica (ovvero che punta ai medesimi base e bound).

Quando abbiamo descritto la chiamata di sistema UNIX `fork`, la quale crea un nuovo processo facendo una copia del processo padre; il processo padre e quello figlio sono identici a parte per il valore di ritorno della fork. Il figlio può poi impostare i propri I/O ed eventualmente usare la system call UNIX `exec` per eseguire un nuovo programma.

In realtà, quando forkiamo un processo, possiamo semplicemente fare una copia della Segment Table. Non c'è bisogno di copiare ogni porzione di memoria. Ovviamente vogliamo che il figlio sia una copia del padre e non che punti alla solita regione di memoria del padre: se il figlio volesse cambiare i dati, dovrebbe cambiare solo i propri e non anche quelli del padre.

Possiamo far funzionare il tutto in maniera efficiente utilizzando la *copy-and-write*. Durante la `fork`, tutti i segmenti condivisi tra padre e figlio sono marcati "read-only" in entrambe le Segment Table. Non appena uno dei due modifica un segmento, viene sollevata un'eccezione e viene effettuata un'intera copia di quel segmento sul momento. Di solito, il processo figlio

modifica solo il proprio stack prima di chiamare la `exec`, perciò, ci sarà solo bisogno di fare una copia fisica solo dello stack.



Quando un SO riutilizza spazio in memoria che era stato utilizzato precedentemente, deve azzerare il suo contenuto. Altrimenti dati privati di un'applicazione potrebbero essere letti da un altra. Ovviamente vogliamo pagare l'overhead per l'azzeramento della porzione di memoria solo se la utilizzeremo.

Questo è un particolare problema dello stack e heap quando sono allocati dinamicamente: non sappiamo di quanta memoria necessita un programma che parte. Lo heap potrebbe essere grande qualche kilobytes o gigabytes.

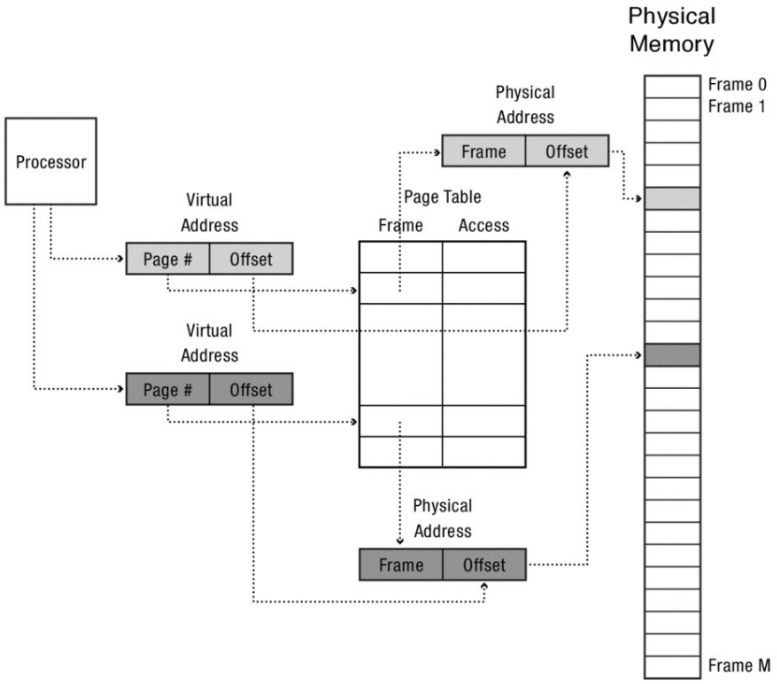
Per affrontare questo problema il SO può utilizzare la tecnica *zero-on-reference*. Con la zero-on-reference, il SO alloca una regione di memoria per lo heap ma azzera solo i primi kilobytes. Setta il registro di bound in modo da limitare il programma alla sola parte azzerata e se questo volesse espandere il proprio heap, si solleva un'eccezione e il kernel azzera altra memoria addizionale.

Sommario *Segmentation:*

- **Pro**
  - condivisione di segmenti tra processi,
  - protezione dei segmenti,
  - allocazione dinamica dello stack e heap,
  - si può utilizzare la copy-on-write.
- **Contro**
  - gestione complessa della memoria: devo eseguire first-fit o best-fit per ogni segmento,
  - ho sempre frammentazione esterna.

8.4 Paged Memory

Un'alternativa alla memoria segmentata è quella *paginata*. Con la paginazione, la memoria è divisa in porzioni di dimensione fissata chiamate *page frames* (pagine). La traduzione degli indirizzi è simile a quella della segmentazione. Invece di una segment table le quali entries contengono i puntatori a segmenti di dimensione variabile, c'è una *Page Table* per ogni processo le quali entries contengono i puntatori alle pagine. Siccome le pagine sono di dimensione fissa e multipli di due, le entries della tabella devono fornire i bit più significativi dell'indirizzo della pagina. Non c'è più bisogno di un bound per l'offset. Ogni processo ha la propria page table che viene memorizzata in memoria (nel TCP o PCB) e a livello hardware ci sarà un registro che punta all'inzio della sua page table e uno per la lunghezza della tabella.



Quindi, mentre un programma vede la propria memoria linearmente, in realtà è interamente spezzettata nella memoria fisica. Il processore eseguirà un'istruzione dopo l'altra usando indirizzi virtuali perché questi sono sempre lineari ma in memoria fisica, due istruzioni contigue potrebbero risiedere in due pagine differenti.

La paginazione affronta la principale limitazione della segmentazione: l'allocazione di spazio libero è semplice. Il SO può rappresentare la memoria fisica come un bit map, dove ogni bit rappresenta una pagina fisica che è libera oppure occupata.

Anche la condivisione di memoria tra processi è conveniente: per ogni processo che condivide una pagina bisogna impostare l'entry della page table in modo che punti alla solita pagina fisica (questo per ogni pagina condivisa). Dal momento che vogliamo sapere quando rilasciare la memoria di un processo finito, la memoria condivisa richiede di avere qualche meccanismo per sapere se le pagine condivise sono ancora utilizzate. La struttura per gestire questo meccanismo si chiama *core map*. La core map è una struttura dati usata per tenere traccia dello stato di una pagina fisica, come quale processo la riferisce.

Molte delle ottimizzazioni discusse con la segmentazione possono esser fatte anche con la paginazione. Per la copy-on-write, dobbiamo copiare le entries della page table e settarli in read-only;

quando viene effettuata una store su una di esse possiamo fare una copia reale della pagina prima di ricominciare il processo. Anche per la zero-on-reference possiamo settare la entry della page table in cima allo stack in modo che sia invalido e che causi una trap nel kernel. Così possiamo espandere lo stack di quanto vogliamo.

Le page tables ci forniscono altre funzionalità. Ad esempio, possiamo far partire un programma prima che tutto il suo codice e i suoi dati siano caricati in memoria. Inizialmente, il SO marca tutte le entries della page table, di un nuovo processo, come read-only (per le pagine del codice) o read-write (per quelle dei dati). Una volta che le prime pagine sono in memoria il SO può iniziare ad eseguire il programma in user-mode mentre il kernel continua a trasferire il resto del codice in background. Se poi il programma salta ad un'istruzione che non è ancora stata caricata, l'hardware solleverà un'eccezione e il kernel può fermare il programma fino a che la pagina non è disponibile. Inoltre, il compilatore può riconoscere quale sono le pagine ideali da far caricare in memoria inizialmente.

Uno svantaggio della paginazione riguarda la gestione dello spazio di indirizzamento virtuale. I compilatori tipicamente si aspettano di avere lo stack di esecuzione contiguo (in indirizzi virtuali) e di una dimensione arbitraria; ogni nuova chiamata di procedura assume che ci sia memoria disponibile nello stack. In un processo single-threaded potremmo posizionare lo stack e lo heap, in memoria virtuale, l'uno sopra l'altro e che crescano l'uno vero l'altro. Il problema si presenta in un processo con più thread in cui abbiamo bisogno che ogni thread abbia il proprio stack con la possibilità di crescere.

La grandezza della page table è proporzionale alla dimensione dello spazio di indirizzamento e non alla memoria fisica. Più lo spazio virtuale è spoglio, più c'è overhead per la page table. La maggior parte delle entries saranno invalide perché rappresenterebbero parti della memoria virtuale che non sono in uso, ma la memoria fisica è sempre necessaria per tutte quelle entries.

Potremmo ridurre lo spazio della page table scegliendo pagine più grandi. Ma quanto grandi dovrebbero essere? Una pagina troppo grande è uno spreco di spazio se un processo non utilizza tutta la memoria all'interno della pagina (frammentazione interna). Questo significa che o le pagine sono troppo grandi (page table ridotta) o la page table è grande (pagine piccole) quindi si cerca di trovare un compromesso.

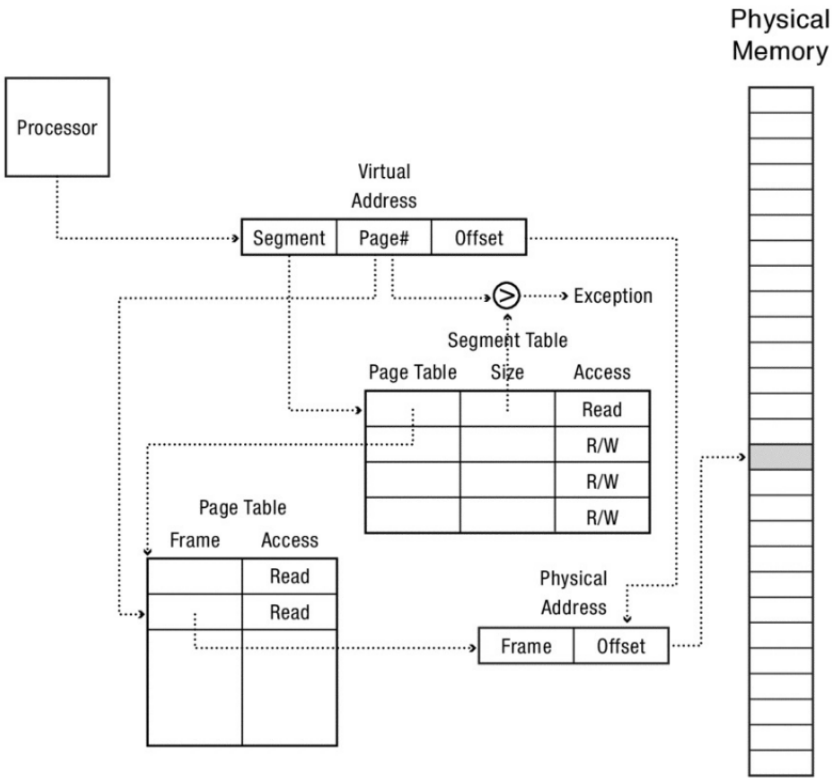
8.5 Multi-Level Translation

Molti SO anziché utilizzare un array (tabella) utilizzano un albero per la traduzione degli indirizzi. Ogni sistema ha le proprie caratteristiche ma alcune sono comuni a tutti. Permettono protezione, condivisione e flessibilità della memoria ed efficiente allocazione e ricerca di indirizzi sparsi.

8.5.1 Paged Segmentation

Partiamo con un sistema con un albero a due livelli. Con la *segmentazione paginata*, la memoria è segmentata, ma ogni entries della segment table, invece di puntare direttamente ad una regione contigua di memoria fisica, punta ad una page table nella quale saranno presenti i puntatori alle pagine del segmento corrispondente. Il "bound" (size) della segment table rappresenta la grandezza della page table, ovvero la lunghezza del segmento in pagine. Siccome la paginazione è utilizzata al livello più basso, la lunghezza di tutti i segmenti è un multiplo della dimensione di una pagina.

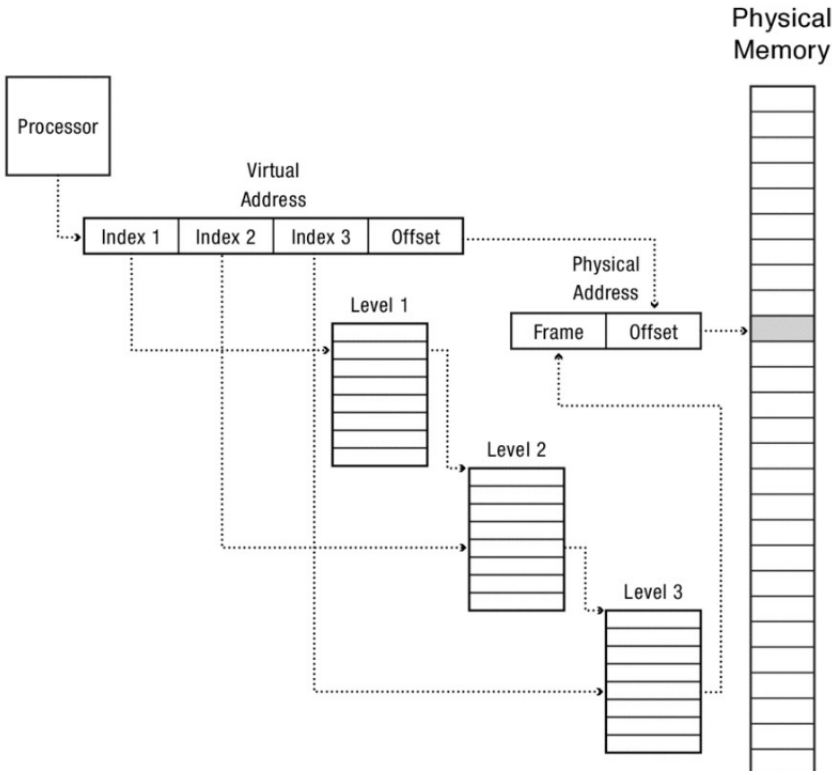
Un indirizzo virtuale ha tre componenti: il numero del segmento, il numero della pagina virtuale interna al segmento e un offset all'interno della pagina. Il numero del segmento indicizza la segment table che contiene la page table del segmento cercato. Il numero della pagina indicizza la page table che contiene l'indirizzo fisico della pagina ricercata. L'indirizzo fisico finale è composto dall'indirizzo iniziale della pagina fisica concatenato con l'offset. Il SO può restringere l'accesso sia ad interi segmenti che a singole pagine.



Nonostante le segment tables sono solitamente memorizzate in registri speciali ad hardware, le page tables per ogni segmento sono più grandi ed è dunque preferibile memorizzarle in memoria fisica.

8.5.2 Multi-Level Paging

Un'altra implementazione quasi equivalente è quella di utilizzare più livelli di page tables. La page table iniziale contiene le entries che puntano ad una seconda page table le cui entries puntano a un'altra page table e così via. In molti SO che utilizzano più livelli di page table, ogni livello è grande abbastanza da stare in una pagina fisica. Solo quella iniziale deve essere riempita; nelle altre sono allocate solo quelle porzioni dello spazio di indirizzamento virtuale che sono in uso. I permessi di accesso e la condivisione possono essere specificati ad ogni livello.



I campi "index" dell'indirizzo virtuale rappresentano l'offset da concatenare con l'indirizzo iniziale del livello. In un registro hardware avrò l'indirizzo del *livello 1* che concatenato col valore `index 1` mi restituisce l'indirizzo del *livello 2* che concatenato col valore `index 2` mi restituisce l'indirizzo del *livello 3* che concatenato col valore `index 3` mi restituisce l'indirizzo della pagina fisica. Infine, concateno quest'ultimo col campo `offset` per trovare l'indirizzo fisico finale.

8.5.3 Multi-Level Paged Segmentation

Possiamo combinare questi due approcci segmentando la memoria ed ogni segmento è gestito da una page table multi-level. Questo è l'approccio utilizzato dall'architettura x86, sia per i modelli 32-bit che 64-bit.

Prima descriviamo quelli 32-bit. La terminologia dell'x86 è leggermente differente da quella già descritta. La x86, per ogni processo, ha una *Global Descriptor Table* (GDT) che è la nostra segment table. La GDT è memorizzata in memoria; ogni entry (descriptor) punta alla (multi-level) page table di quel segmento insieme alla lunghezza del segmento ed i suoi permessi d'accesso. Per iniziare un processo, il SO crea una GDT ed inizializza un registro, il *Global Descriptor Table Register* (GDTR), che contiene l'indirizzo e la lunghezza della GDT.

Storicamente, nella x86 il processore utilizza due registri per specificare il numero del segmento (l'indice nella GDT) e l'indirizzo virtuale. Per i 32-bit l'indirizzo virtuale all'interno di una entry della GDT è composto per i primi 10 bit dall'indice della page table di primo livello, i successivi 10 per quella di secondo livello e gli ultimi 12 rappresentano l'offset all'interno della pagina. Ogni elemento di una page table (livello) occupa 4Bytes e le pagine sono grandi 4KB, così che la page table di Livello 1 e quelle di Livello 2 risiedono tutte in una singola pagina fisica. Nel modello 64-bit sono presenti 3 livelli per segmento.

Sommario *Multi-Level Translation:*

- **Pro**
  - possiamo allocare solo le page table che ci servono,
  - allocazione della memoria semplice,
  - condivisione a livello di segmento o pagina.
- **Contro**
  - la traduzione da virtuale a fisico non è immediata: visita di un albero. (vedremo che questo limite si supera con l'utilizzo della *cache*)

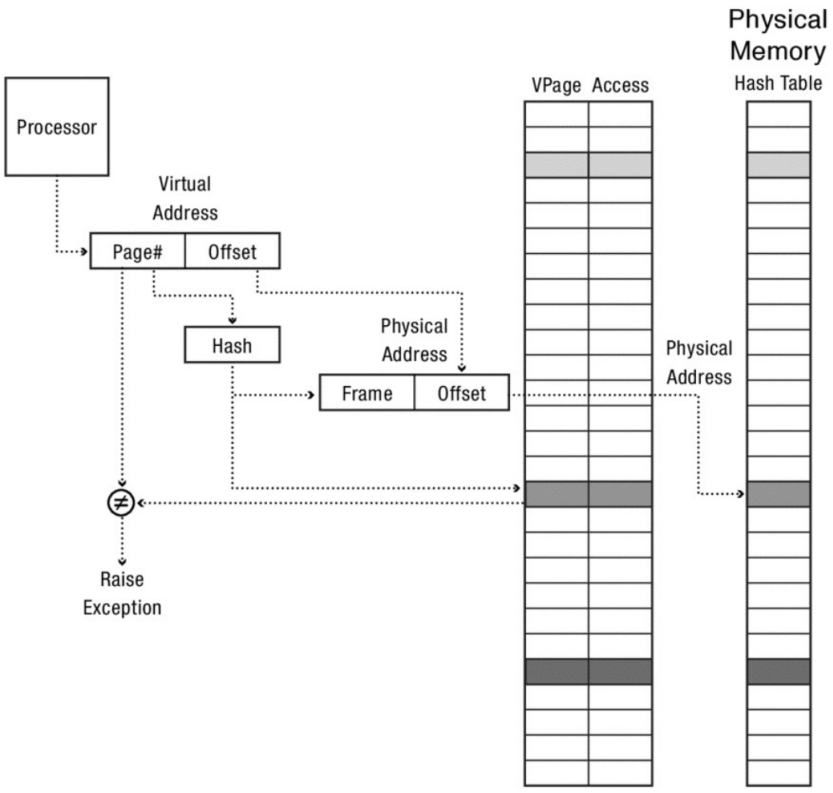
8.6 Portabilità

I SO per poter essere largamente utilizzati devono girare su architetture differenti. Essi hanno dunque, delle proprie strutture dati per la gestione della memoria che rispecchia, ma non sono identiche, quelle hardware. Consiste di tre parti:

- **Lista di oggetti di memoria (segmenti)**. Gli oggetti di memoria sono segmenti logici. Sia se l'hardware sottostante è segmentato o meno, il kernel memory manager ha bisogno di tenere traccia di quali regioni di memoria rappresentano i dati sottostanti, come il codice del programma, codice di libreria, dati condivisi tra processi, regione di copy-on-write, o memory-mapped file. Ad esempio, quando parte un processo, il kernel potrebbe controllare la lista degli oggetti per controllare che il codice sia già in memoria; similmente, quando un processo apre una libreria, può controllare se è già stata linkata da un altro processo.
- **Traduzione da virtuale a fisico.**
- **Traduzione da fisico a virtuale.**

Però non tutti i SO utilizzano questa struttura dati (e.g. Apple OS X, IBM Power PC): invece di utilizzare un albero, utilizzano una tabella hash.

Per motivi storici, la tabelle hash per la traduzione di indirizzi è chiamata *Inverted Page Table*. Con la Inverted Page Table, il numero di pagina virtuale viene "hashata" in una tabella di dimensione proporzionale al numero di pagine fisiche. Ogni entry nella hash table contiene la seguente tupla:

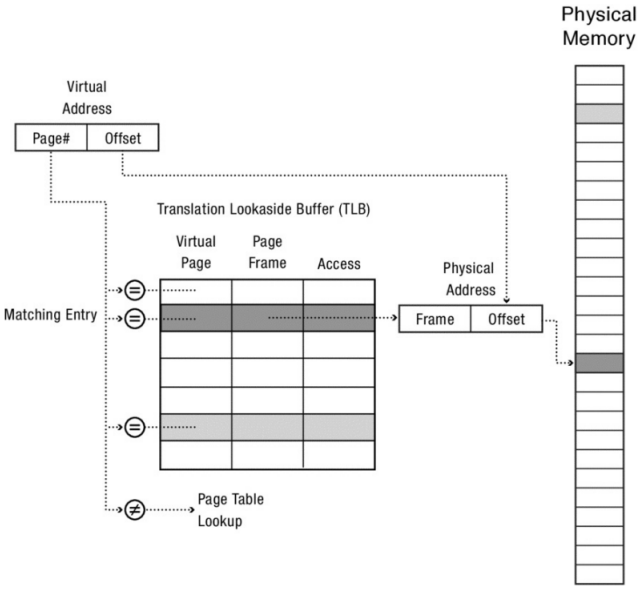


Con l'utilizzo di una funzione di hashing ovviamente si incorre nel problema della collisione e quindi andrebbero apportate delle opportune modifiche affinché queste non avvengono. Abbiamo però dei vantaggi dal punto di vista di velocità se riesco ad implementarla efficacemente.

8.7 Translation Lookaside Buffers

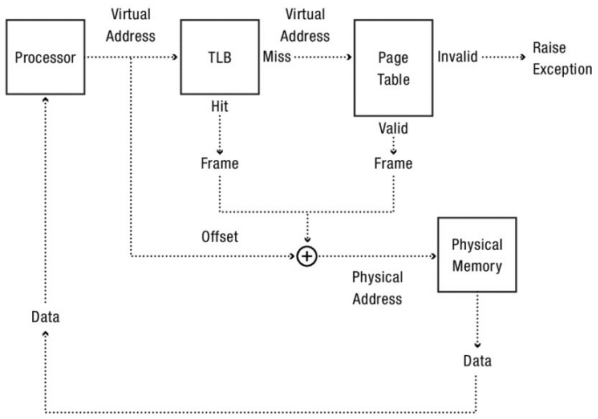
Ritornando sull'utilizzo degli alberi, possiamo migliorare ulteriormente le performance. Dopo tutto, il processore normalmente esegue le istruzioni in sequenza. L'hardware prima traduce il program counter per un'istruzione e poi percorre la multi-level translation table per cercare la cella di memoria fisica dove risiede quell'istruzione. Quando viene incrementato il PC, il processore percorre di nuovo tutti i livelli per cercare quella successiva che sarà quasi sicuramente nella stessa pagina fisica (*principio di località*). Il processore sta dunque facendo il solito lavoro per ottenere il solito risultato. Per questo si introduce un *Translation Lookaside Buffer (TLB).*

Un TLB è un piccola tabella hardware che contiene il risultato delle recenti traduzioni di indirizzi. Ogni entry del TLB mappa una pagina virtuale in una fisica e si riesce a controllare simultaneamente tutte le entry.



Se c'è una corrispondenza, il processore utilizza quell'entry per formare l'indirizzo fisico. Questo si chiama un *TLB hit.* Se ho un TLB hit, l'hardware deve ancora controllare i permessi.

Una *TLB miss* occorre se nessuna delle entry della TLB matcha. In questo caso, l'hardware scorre tutti i livelli come al solito. Quando poi la traduzione è stata completata, la pagina fisica viene utilizzata per formare l'indirizzo fisico e si aggiunge la traduzione nel TLB, rimpiazzando eventualmente la entry che non viene utilizzata da più tempo.



Il costo per una traduzione:

```
Cost of TLB lookup + Prob(TLB miss) * cost of page table lookup
```

8.7.1 Software-loaded TLB

Potremmo allora pensare di utilizzare solo il TLB anziché la multi-level page table: se abbiamo un TLB hit bene, altrimenti se abbiamo una TLB miss si fa una trap al kernel che cerca l'indirizzo e carica la traduzione nel TLB.

Questo approccio semplifica drammaticamente la progettazione del SO perché non avrebbe più bisogno di tenere due insiemi di page table, una dell'hardware e una per sé stesso. Se avviene una TLB miss, il SO consulterebbe la propria struttura dati per determinare quali dati devono essere caricati nel TLB.

Nonostante sia conveniente per il SO, un *Software-loaded TLB* è piuttosto lento ed costa più fare una trap al kernel che fare la traduzione via hardware.

Come vedremo, il contenuto delle entries della page table può essere memorizzato in una on-chip hardware cache.

8.8 Superpagine

Un modo per migliorare il rate di TLB hit è quello di utilizzare le *superpagine.* Un superpagina è un insieme di pagine fisiche contigue che mappano una regione contigua di memoria virtuale, in cui le pagine sono allineate in modo da avere i bit più significativi, dell'indirizzo, uguali.

Le superpagine complicano l'allocazione della memoria per un SO perché richiede a quest'ultimo di allocare pezzi di memoria di dimensione diversa. Il vantaggio però è quello di avere un numero minore di entries del TLB. Ogni entry nel TLB ha un flag per identificare se si riferisce ad una pagina o una superpagina.

Un esempio di utilizzo delle superpagine riguarda il Video Frame Buffer. Quando lo schermo viene ridisegnato, il processore tocca ogni pixel. Se ogni entry del TLB dovesse mappare pagine da 4KB anche con un TLB on-chip di 256 entries si avrebbe solo 1MB del frame buffer. Questo indurrebbe al TLB di dover cercare le nuove pagine nella page table. Un caso ben più grave si presenta quando si disegna solo una linea verticale dello schermo. Il frame buffer rappresenta una matrice salvata riga per riga, perciò ogni linea orizzontale di pixel è in una pagina differente. Dunque per ogni singolo pixel di una linea verticale dovremmo caricare una nuova entry del TLB! Con le superpagine, l'intero frame buffer può essere mappato con una singola entry.

8.9 TLB Consistency

Ci sono tre problemi da considerare:

- **Process context switch.** Cosa accade quando avviene un cambio di contesto? L'indirizzi virtuali del nuovo processo non sono più validi per il nuovo processo. Altrimenti, il nuovo processo potrebbe leggere le strutture dati di quello vecchio oppure causare un crash o scovare informazioni sensibili. Durante un cambio di contesto, dobbiamo cambiare il registro hardware relativo alla page table per puntare a quella del nuovo processo. Inoltre, il TLB contiene le copie delle traduzioni e dei permessi delle pagine del vecchio processo. Un approccio potrebbe essere quello di scartare il contenuto del TLB (*flush*) ad ogni context switch. Siccome non è una scelta performante, molti processori utilizzano un *tagged TLB.*

  Con un tagged TLB, il SO memorizza l'ID del processo corrente in un registro hardware ad ogni cambio di contesto. Quando si fa una ricerca si ignorano le entires con ID diversi.

- **Modifica pagine.** Cosa succede quando il SO modifica i diritti di accesso di una pagina? La modifica va apportata sia nella page table che nel TLB. Se dovessi modificarla solo in una delle due rischio di avere un'inconsistenza. Un'alternativa potrebbe essere quella di modificare solo la entry nel TLB, segnandomi che è stata modificata, e quando la devo buttar fuori per far spazio ad altro ricopio la modifica anche nella page table.

  Cosa succede se il SO sposta le pagine di un processo? O si indica che quell'elemento contiene informazioni vecchie oppure ci scriviamo direttamente quelle nuove.

- **TLB shootdown.** Su un multiprocessore ogni processore ha una copia del TLB, quindi quando si modifica un'entry della page table, questa deve essere scartata su tutti i TLB. Tipicamente, solo il processore corrente può invalidare il proprio TLB, dunque rimuovere l'entry per tutti i processori richiede che il SO interrompa ogni processore per richiedergli di rimuovere l'entry nel loro TLB. Questa operazione si chiama *TLB shootdown*

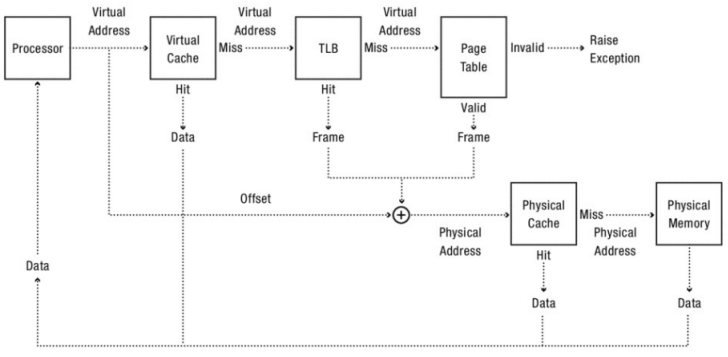
8.10 Virtually Addressed Caches

Un altro step per migliorare la traduzione degli indirizzi è quello di includere una *virtually address caches* ovvero una cache nella quale risiedono copie di memoria fisica, indicizzate da indirizzi virtuali. Quando c'è un match, il processore può utilizzare immediatamente i dati senza dover aspettare che vengano prelevati dalla memoria principale (ovvero ricercandoli nella TLB o nella page table). Spesso, come il TLB, la virtually addressed cache viene divisa in due: una metà per le istruzioni e l'altra per i dati.

8.11 Physically Addressed Caches

Molte architetture includono anche una *physically addressed cache* che è consultata come una cache di secondo livello dopo quella virtuale (quella di prima) e il TLB ma prima della memoria principale.

Una volta che abbiamo l'indirizzo fisico, lo cerchiamo in questa cache e se è presente possiamo restituire il dato al processore senza dover accedere alla memoria principale che è più lenta.



9 Caching and Virtual Memory

La cache è una copia di un dato che può essere acceduta più velocemente che dell'originale. Abbiamo già visto alcuni esempi di cache:

- **TLB.** I processori moderni utilizzano un translation lookaside buffer per mantenere i risultati recenti delle traduzioni di indirizzi di una multi-level page table.
- **Virtually address caches.** Molti dei processori moderni includono una virtually addressd cahce vicina al processore. Ogni entry della cache memorizza il valore associato ad un indirizzo virtuale, dando la possibilità di prelevare più velocemente quel dato.
- **Physically addressed caches.** Altri includono inoltre, una cache di secondo (o terzo livello) prima della memoria principale. Ogni entry memorizza un dato indicizzato con un indirizzo fisico.

In altre parole, le cache sono le fondamenta per rendere i computer più veloci. Ciononostante, hanno anche alcuni svantaggi.

Tutte le cache cercano di affrontare le seguenti sfide:

- **Individuare la copia cache.** Siccome le cache sono progettate per aumentare le prestazioni, una domanda chiave è come possiamo sapere velocemente se nella cache c'è il dato cercato oppure no.
- **Politica di rimpiazzo.** Molte cache hanno limiti fisici riguardo a quanti elementi possono memorizzare; quando un nuovo dato arriva in cache, il sistema deve decidere quali dati tenere in cache e quali rimpiazzare.
- **Coerenza.** Come vediamo quando una copia di un dato in cache è obsoleta? In realtà non vedremo questa parte ma ci focalizzeremo sui primi due punti.

9.1 Cache Concept

Iniziamo definendo alcuni termini. Quando abbiamo bisogno di leggere un valore di una certa locazione di memoria, consultiamo prima la cache, e questa, o ci risponde col valore (se la cache lo conosce) oppure invia la richiesta al livello successo nella gerarchia. Se la cache ha il valore si dice *cache hit*. Se la cache non lo ha si dice *cache miss*.

Affinché una cache sia utile deve avere due proprietà. Prima, il costo per recuperare i dati dalla cache deve essere inferiore che fetchare i dati dalla memoria. In altre parole, il costo di un cache hit deve essere minore di un cache miss, altrimenti converrebbe non usarla.

Seconda, la probabilità di un cache hit deve essere abbastanza alto altrimenti non varrebbe la pena utilizzarla. Una fonte di predicibilità è la *località temporale*: i programmi tendono a riferire alle stesse istruzioni o dati ai quali hanno acceduto recentemente.

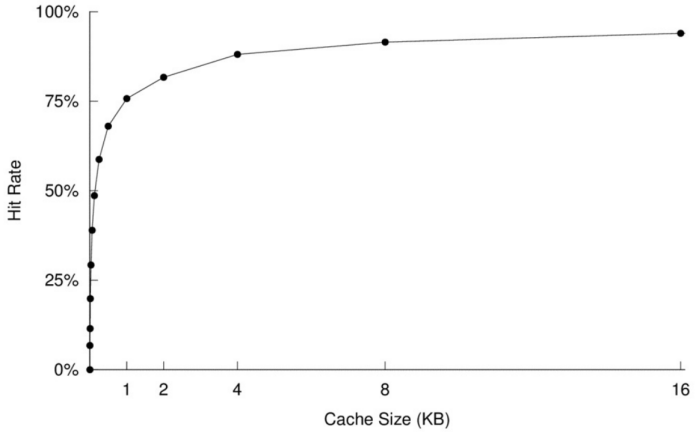
Un altra fonte di predicibilità è la *località spaziale*: i programmi tendono a riferire i dati vicini ai dati utilizzati di recente. Infatti, le cache sono spesso progettate affinché carichino blocchi di dati in un istante, invece che una singola locazione.

Un altro modo per sfruttare la località spaziale è quello di fare il *prefetch* dei dati nella cache prima che il processore li richiedi.

9.2 Working Set Model

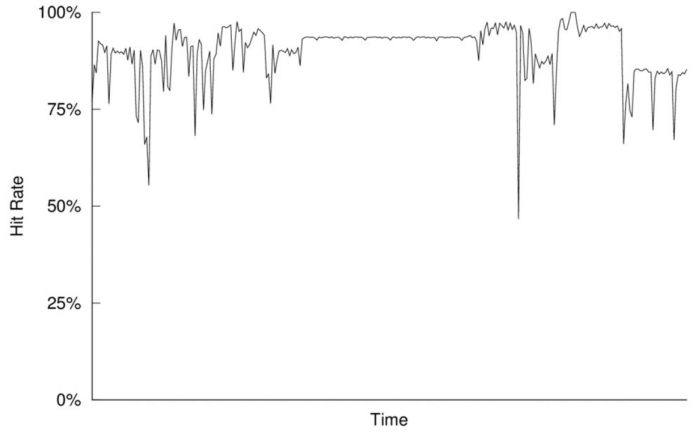
Per quanto riguarda un programma, una cache sufficientemente grande avrà un alto rate di cache hit. Se la cache potrebbe contenere tutti i dati e le istruzioni di un programma, avremmo zero cache miss una volta che è stato tutto caricato. Se invece fosse estremamente piccola avremmo un elevato numero di cache miss (dovremmo rimpiazzare continuamente locazioni).

Il *Working Set* definisce l'ammontare di memoria necessaria ad un programma in un dato momento. In altre parole è l'insieme delle pagine che un processo accederà nel futuro e che dovrebbero essere in memoria affinché si possa avere il massimo dei progressi, in termini di esecuzione, di quel processo. Tanto più il working set può stare nella cache, più cache hit avremo e la performance dell'applicazione sarà buona.



Un concetto vicino a quello di working set è quello del *thrashing*. Il fenomeno del thrasing avviene quando la cache è troppo piccola per mantenere il working set di un programma e conseguentemente il numero di cache miss è maggiore dei cache hit. Ad ogni cache miss, dobbiamo "sfrattare" un blocco per far spazio ad uno nuovo. Ciononostante, il nuovo blocco potrebbe venir nuovamente rimpiazzato prima che venga riutilizzato.

Programmi diversi, e utenti diversi, avranno working set di dimensione diversa. Anche all'interno dello stesso programma, differenti fasi di esecuzione potrebbero avere working set di dimensione diversa. Il risultato di questi cambi di fase comporta che la cache avrà un rate di cache miss "esplosivo": periodi di pochi cache miss succeduti da periodi di molti cache miss. Anche i cambi di contesto fra processi causano cache miss "esplosive" non appena la cache scarta il working set del vecchio processo con quello del nuovo processo.



9.3 Politiche di rimpiazzo

Se la pagina ricercata non è in memoria e abbiamo un cache miss, quale blocco rimpiazziamo? La politica di rimpiazzo può avere un notevole impatto per quanto concerne il cache hit rate.

Come con lo scheduling del processore, ci sono numerose opzioni per la politica di rimpiazzo. Non c'è un'unica risposta corretta! Molte politiche di rimpiazzo sono ottime per un determinato workload e pessime per altri, in termini di cache hit rate.

9.4 Random

Una pratica politica di rimpiazzo è quella di scegliere di rimpiazzare un blocco a caso. Soprattutto per una cache hardware di primo livello, il sistema potrebbe non avere tempo per prendere una decisione più complessa, e il costo di una decisione sbagliata potrebbe essere irrisorio se l'oggetto è nel livello cache successivo.

Lo svantaggio di questa politica è anche la sua stessa forza. Qualsiasi sia il pattern di accesso, la politica random non sarà mai pessima. Non prenderà mai la peggior decisione possibile, o perlomeno, non in media. Ciononostante, è impredicibile e potrebbe ostacolare un'applicazione che era stato progettato per gestire attentamente il proprio uso in diversi livelli di cache.

9.5 First-In-First-Out (FIFO)

Un'altra alternativa quella di rimpiazzare il blocco cache o la pagina che è in memoria da più tempo, ovvero, FIFO. Sfortunatamente, FIFO può essere la peggior politica di rimpiazzo per la maggior parte dei workload. Ad esempio, un programma che cicla ripetutamente su un array in memoria il quale è troppo largo per stare interamente nella cache. Su una scansione ripetuta, FIFO rimpiazzerà sempre il blocco o la pagina che servirà subito dopo.

Ref.	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

9.6 MIN

Se la FIFO può essere pessima per alcuni workload: quale politica di rimpiazzo è ottima per minimizzare i cache miss? La politica ottima, chiamata *MIN*, è quella di rimpiazzare quel blocco che verrà utilizzato più lontano nel tempo.

Come con la Shortest Job First, MIN richiedere la conoscenza del futuro, e quindi non possiamo implementarla direttamente. Invece, possiamo utilizzarla come obiettivo: trovare quei meccanismi che sono efficaci nel predire quale blocco sarà utilizzato nel futuro prossimo, così da tenerlo nella cache.

Se riuscissimo a predire il futuro, potremmo fare anche meglio della MIN con la tecnica del prefetching in modo da far arrivare i blocchi "giusto in tempo", proprio quando ne abbiamo bisogno. Nel caso ottimo, ridurrebbe il numero di cache miss a zero.

9.7 Least Recently Used (LRU)

Un modo per predire il future è guardare il passato. Se i programmi esibiscono località temporale, le locazioni che riferiranno nel futuro saranno, molto probabilmente, le solite alle quali hanno riferito recentemente.

Una politica di rimpiazzo che cattura questi effetti è quella di rimpiazzare i blocchi che non vengono usati la più tempo, o gli ultimi usati recentemente (LRU). Via software, la LRU, è facile da implementare: ad ogni cache hit, si sposta il blocco in testa alla lista e ad ogni cache miss si rimpiazza quello in coda. Via hardware, mantenere una lista di blocchi di cache è troppo complesso da implementare; perciò abbiamo bisogno di approssimare la LRU (vedi dopo).

In alcuni casi, la LRU può essere ottima:

LRU

Ref.	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+				+			+	
2		B			+								+		
3				C					E			+			
4						D		+		+					C

FIFO

Ref.	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+		E						
2		B			+						A		+		
3				C								+	B		
4						D		+		+					C

MIN

Ref.	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+				+			+	
2		B			+								+		C
3				C					E			+			
4						D		+		+					

In questo caso, LRU si comporta come la MIN, ma non è sempre così. Infatti, la LRU può talvolta essere la peggior politica. Questo avviene ogni volta che il blocco usato meno di recente è quello che verrà riferito dopo. Ad esempio quando un programma fa ripetute scansioni della memoria:

LRU

Ref.	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

MIN

Ref.	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A					+					+			+	
2		B					+					+	C		
3			C					+	D					+	
4				D	E				+						+

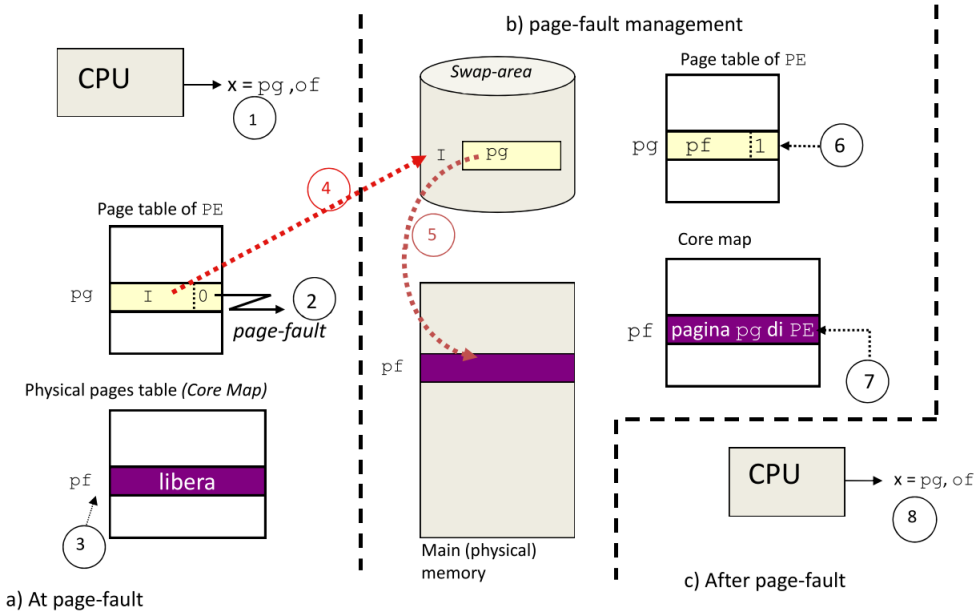
9.8 Case Study: On demand paging

Concentrandosi sul caricamento di pagine dal disco alla memoria principale, la strategia migliore è quella di caricare una pagina quando ne ho bisogno: *on demand paging*. La page table, tra le altre cose, contiene il campo che identifica l'indirizzo fisico della pagina in memoria principale, se è presente, o sul disco, altrimenti. Per sapere se è presente o meno c'è un bit chiamato *bit di presenza*. Infine contiene due bit per i diritti (Read e/o Write), e due bit utilizzati dall'algoritmo di sostituzione: *modified bit* (chiamato anche *dirty bit*) e *used bit*.

Numero pagina fisica se P=1; Indirizzo di disco pagina se P=0	R	W	U	M	P

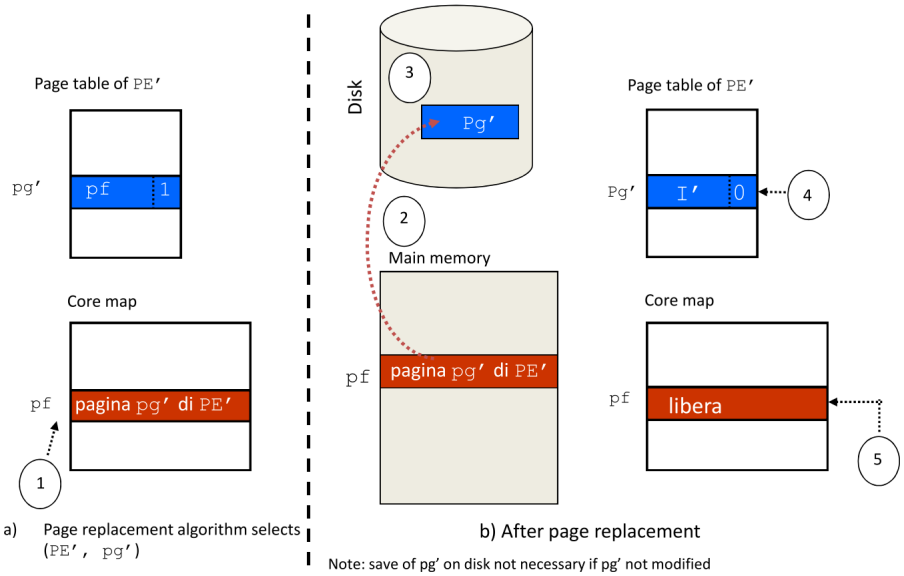
Dunque, se P=0 comporta un *page fault* e devo andare a recuperare le informazioni dal disco. Questi sono i passi che accadono:

1. Il processore produce un indirizzo virtuale ed abbiamo un TLB miss,
2. Si cerca nella page table e andiamo alla riga indicizzata dall'indirizzo virtuale. Il bit di presenza è 0: page fault,
3. Si cerca una pagina fisica libera in memoria principale o se ne rimpiazza una se è piena,
4. Trap al kernel e si converte l'indirizzo della page table in un indirizzo di file + offset e si localizza la pagina nel disco,
5. Si carica la pagina dal disco nella pagina fisica trovata precedentemente,
6. Si aggiorna la page table: si aggiunge il numero della nuova pagina fisica e P = 1,
7. Si aggiorna la Core map indicando che la pagina fisica è occupata e anche da quale pagina,
8. A questo punto si riesegue il processo, la CPU richiede di nuovo, TLB miss, la pagina ora si trova e si può eseguire.



Se invece devo rimpiazzare una pagina di memoria principale e ricaricarla sul disco:

- Selezionare la vecchia pagina da rimpiazzare,
- Trovare tutte le entries della page table che puntano alla vecchia pagina,
- Settare ogni entry della page table come invalida,
- Rimuovere ogni entries del TLB,
- Scrivere le modifiche nella pagine del disco, se necessario (se è stata modificata)



Come si fa a sapere se una pagina è stata modificata? Ad hardware, ogni volta che viene fatta una store, si modifica il flag modified = 1 sia nella page table che il TLB del processo. Nonostante sia veloce, è troppo complesso. Un'altra alternativa è farlo via software dove ogni volta che viene effettuata una `store` si chiama il SO il quale andrà a modifica il flag. Anche questa soluzione è poco efficiente in termini di velocità.

Dunque, alcuni SO utilizzano la seguente strategia: una pagina non modificata viene settata con permessi di accesso read-only, anche se il programma la può scriverci. Il programma viene poi eseguito normalmente e appena avviene un'istruzione di `store` sulla pagina, l'hardware solleva un'eccezione di memoria. Il SO può ora prendere nota che la pagina è stata "sporcata" e aggiornare l'accesso da read-only a read-write e far ripartire il processo. Tutto questo è analogo per l'used bit.

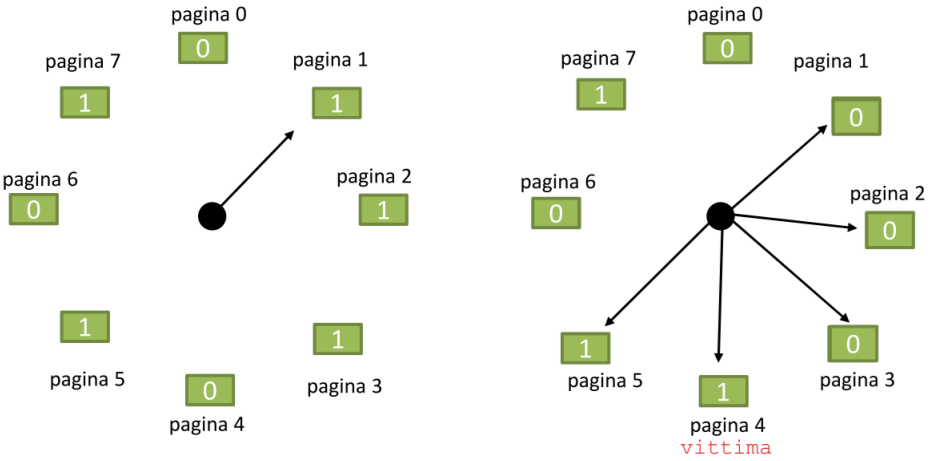
9.9 LRU approssimato - Second chanche

Come già detto, implementare la LRU ad hardware è costoso quindi siamo costretti ad approssimare la politica. L'hardware mantiene una minima quantità di informazioni relative agli accessi di ogni pagina così da poter far lavorare il SO e approssimare la LRU.

Molte architetture utilizzano un *use bit* in ogni entry della page table. Inizialmente sono tutti a zero, quando poi riferisco ad una pagina, metto ad 1 il suo use bit (quando viene portata nel TLB).

Il SO può utilizzare l'use bit in vari modi ma l'utilizzo più comune è quello dell'*algoritmo dell'orologio*. La politica di rimpiazzo si chiama *Second chance*.

La lancetta scorre finché non trova una pagina col bit ad 0, a questo punto si rimpiazza quella pagina e si avanza la lancetta alla posizione successiva.



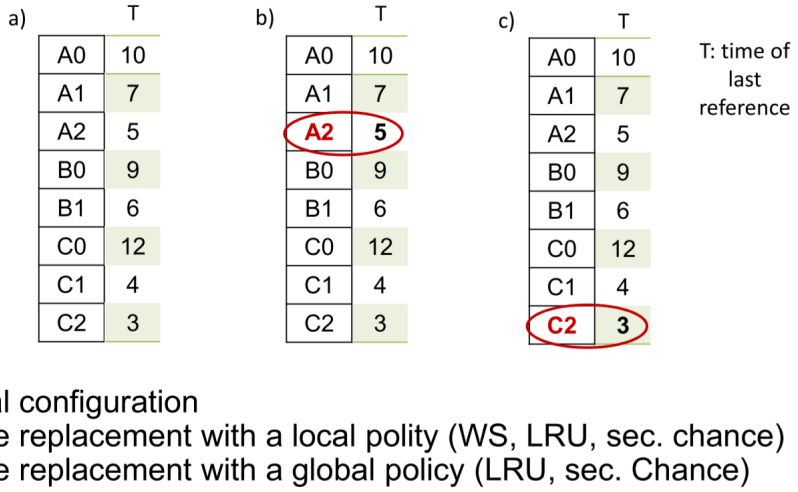
Questa politica può essere generalizzata all'n-esima chanche.

9.10 Rimpiazzo locale e globale

Le politiche di rimpiazzo di dividono in due categorie:

- **Algoritmi globali.** Le pagine selezionate per il rimpiazzo sono scelte tra tutte le pagine della memoria principale a prescindere dal chi appartiene la pagina. In questo caso, se utilizzo n-th chance, la "distanza passata" è quella del clock della macchina (tempo assoluto). I processi che avanzano lentamente hanno un valore di utilizzo delle loro pagine più vecchio di altri e quindi c'è il rischio di trashing.
- **Algoritmi locali.** Le pagine selezionate per il rimpiazzo sono scelte tra tutte le pagine del processo che ha provocato il cache miss. Tende ad essere più equa che utilizzano algoritmi globali. La "distanza passata" si basa su un proprio tempo (orologio) (tempo relativo).

Esempio:



a) Initial configuration
b) Page replacement with a local polity (WS, LRU, sec. chance)
c) Page replacement with a global policy (LRU, sec. Chance)

9.11 Working Set algorithm

Come abbiamo già detto, supponendo che il futuro sia speculare al passato, il *working set* è l'insieme delle pagine che sono state riferite negli ultimi k accessi alla memoria. Questo però è troppo costoso da implementare perché per ogni load e store dobbiamo sospendere il processo e aggiornare le informazioni sui riferimenti. Un altro modo è quello di definire il working set come l'insieme delle pagine che sono state riferite in un certo periodo di tempo P (implementabile con l'utilizzo dell'use bit).

Utilizzando il principio del working set, ad ogni processo gli si riserva, in memoria, un insieme di pagine dove verrà caricato il proprio WS. Quando non abbiamo posto per una nuova pagina, ne rimpiazziamo una di quel processo. Questo significa che la politica di rimpiazzo basata sul WS è locale.

In realtà è difficile conoscere il WS di un processo quindi, in realtà, ci basiamo su un altro concetto, quello dell'*insieme residente*. Ovvero quelle pagine, di quel processo, che sono attualmente in memoria e in generale è diverso dal WS. Si cerca di avere un resident set uguale al working set (quest'ultimo si riferisce più al futuro).

Ogni pagina ha un *Bit R* ("referred" o "use" bit) che indica se la pagina è stata riferita nell'ultimo periodo e un valore che indica l'ultimo riferimento in termini di tempo. Ogni processo ha un proprio *tempo virtuale* ed ad ogni periodo o page fault:

Page table

Time of last reference	Bit R	...
2084	1	
2003	0	
1980	1	
1213	0	
2014	1	
2020	1	
1604	0	

Current virtual time: 2204

For each page: {
if (R==1)
time of last reference = current virtual time; R=0

else if (R==0) && (age>P)
removes the page
}

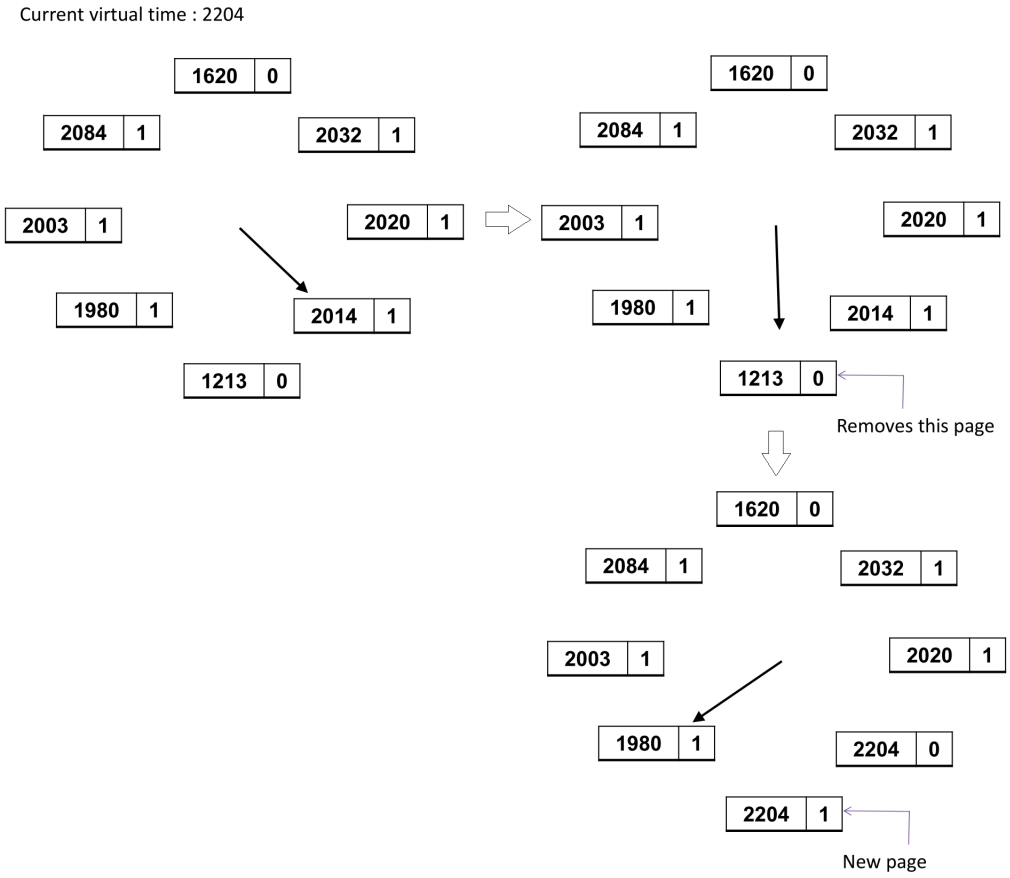
Se ho un page fault devo rimpiazzarne una

if (age<=P for each page)
removes the page with smaller time of last reference

Age: current virtual time – time of last reference

Questo algoritmo è piuttosto pesante in termini di elaborazione perché richiede, tutte le volte che ho un page fault devo scandire tutta la page table. Questo algoritmo è una specializzazione dell'LRU: divido le pagine in quelle riferite e non riferite, quelle non riferite recentemente le divido in quelle vecchie e quelle nuove e se ne ho una molto vecchia la scarico altrimenti applico l'LRU tra quelle non riferite e vecchie.

Possiamo dunque approssimarlo utilizzando l'algoritmo dell'orologio (second chance) e prende il nome di *Working Set Clock*:



Solitamente si lascia un pool di pagine libere in modo da velocizzare la gestione dei page fault.

On demand paging. Inizialmente quando un processo viene creato (o un thread), non è caricato niente in memoria. Mano mano che il processo chiede informazioni, si creano dei page fault e questi permettono di caricare le informazioni in memoria e si va avanti. Inizialmente dunque, ci sono parecchi page fault.

Prepaging. Quando un processo viene caricato, vengono già caricate le pagine in memoria (tramite DMA) parallelamente alla sua esecuzione e in background continuo a caricarne altre. Non è facile implementarlo perché è difficile sapere quali sono le pagine che mi serviranno all'inizio.

Gli algoritmi di allocazione delle pagine, per sapere quali pagine allocare/mantenere in memoria (Resident set), sono dinamici e non statici. Ovvero si basano sul Page Fault Frequency (PFF). E' un parametro dinamico, calcolato durante l'esecuzione di un processo dal SO. Quando la frequenza dei page fault reali è maggiore di quella "naturale" significa che il resident set è stato mal stimato e me ne serve uno più grande.

Se si presenta il fenomeno del thrashing, una soluzione è quella di ridurre il numero di processi in memoria principale in modo da ridurre la competizione tra processi. Per fare ciò, si utilizza la tecnica dello *swaps out*. Ovvero si sposta dalla memoria al disco senza ucciderlo (viene soltanto bloccato). In futuro, quando le condizioni lo permettono, verranno rimessi dentro in memoria principale (*swaps in*). Ovviamente quando avviene lo swaps out, vengono caricate sul disco, solo quelle pagine che sono state modificate perché le altre ci sono già.

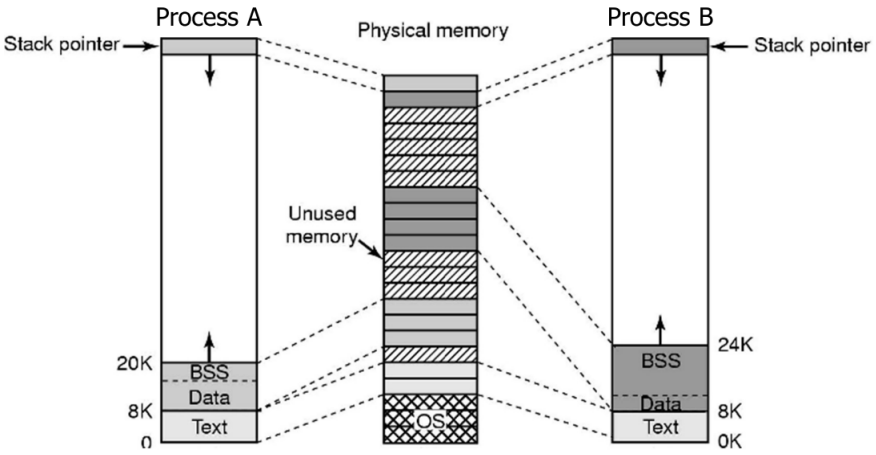
9.12 Case Study: UNIX

Lo sviluppo di UNIX è nato nei Laboratori Bell negli anni sessanta. Negli anni si sono susseguite varie versioni fino ad arrivare alle versioni considerate standard, chiamate BDS (Berkeley Software Distribution). Dalla BSD v.3 viene utilizzata la segmentazione paginata e la memoria virtuale basata sulla paginazione on-demand.

Per realizzare la paginazione on-demand si utilizza una **Core map**: una struttura del kernel che contiene l'allocazione delle pagine fisiche e dice se è libera oppure no. Se non è libera, indica qual è il suo processo.

L'algoritmo di rimpiazzo è il second-chance.

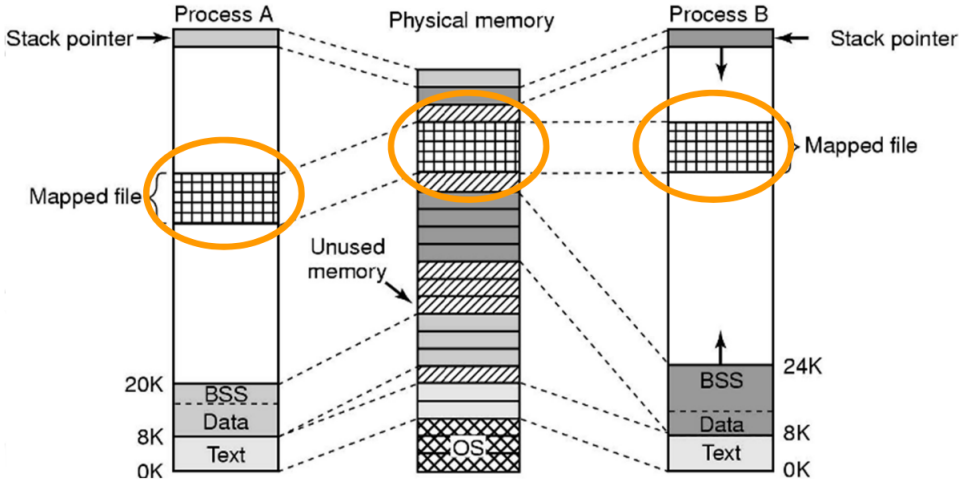
La memoria è organizzata come segue:



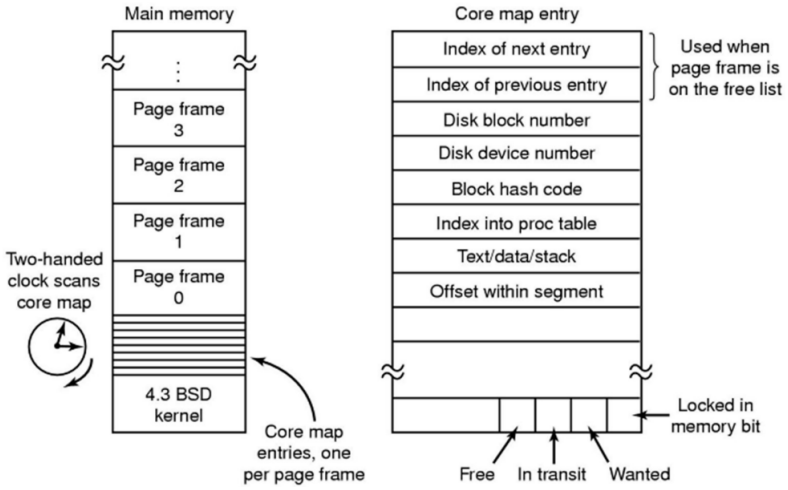
9.12.1 Memory-mapped file

L'idea è quella di caricare in memoria porzioni di file in modo da utilizzare operazioni di load/-store anziché di read/write. Il vantaggio di ciò è il non dover distinguere se un file è su disco o in memoria, lo tratto come se fosse in memoria. Altro vantaggio è quello di avere la Zero-copy I/O: mano mano che uso il file, il SO lo carica in memoria e grazie al DMA riesco a caricare il resto mentre ci lavoro.

La mappa è condivisa tra processi:



9.12.2 Paginazione



9.12.3 Rimpiazzo pagine (BSD)

L'algoritmo di rimpiazzo è il second-chanche (globale) oppure altre varianti. Unix mantiene sempre una riserva di pagine libere e l'algoritmo di sostituzione viene eseguito dal pager *Page Daemon* che utilizza tre parametri:

- *minfree*: numero minimo di pagine libere necessario per evitare swapping dei processi,
- *desfree*: numero desiderato di pagine libere,
- *lotsfree*: numero minimo di pagine libere per evitare sostituzione di pagine.

$$lotsfree > desfree > minfree$$

Uno sketch dell'algoritmo è il seguente:

- ***if*** (*#freeblocks* ≥ *lotsfree*) return //*no operation required*
- ***if*** (*minfree* ≤ *#freeblocks* < *lotsfree*) ***or***
  (*#freeblocks* < *minfree* ***and***
  Average[*#freeblocks,* Δ*t*] ≥ *desfree*))
  replage pages until *#freeblocks* = *lotsfree* + *k* (with *k*>0)
- ***if*** ((*#freeblocks* < *minfree* ***and***
  Average[*#freeblocks,* Δ*t*] < *desfree*))
  swapout processes

C'è una relazione con la teoria del working set:

- *If #freeblocks* $<$ *minfree* significa che ho avuto un gran numero di page fault dall'ultima esecuzione del page daemon e quindi esisteranno dei processi col resident set minore del working set che causano thrashing.
- *If Average[#freeblocks,* Δt*]* $\geq$ *desfree* significa che è un problema temporaneo che si sistemerà da solo. Se invece è $<$ significa che dura da molto tempo e quindi non basta scaricare delle pagine e caricarne altre ma bisogna effettuare lo Swap out.

Il page daemon seleziona le vittime in base a vari criteri: priorità, quantità memoria richiesta, da quanto tempo non viene messo in esecuzione ecc. Lo swap out continua finché
#freeblocks ≥ lotsfree + k (with k>0)

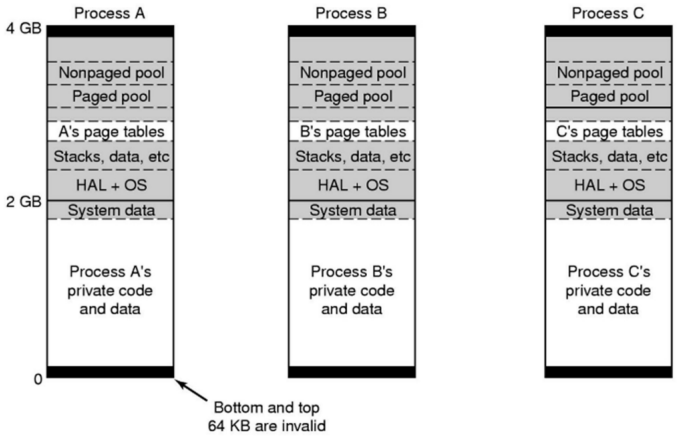
Lo swap in avviene quando il numero di pagine libere è grande abbastanza. Il page daemon seleziona una o più processi in base: al tempo che è stato swappato fuori, quantità memoria richiesta ecc.

Viene effettuato lo swap in di uno o più processi finché è garantito che
#freeblocks ≥ lotsfree + k (with k>0)

9.13 Case Study: Windows (32 bit)

La dimensione della memoria virtuale è di 4Gbyte (indirizzo virtuale di 32 bits). La memoria virtuale è paginata con paginazione a domanda con pagine di dimensione fissa(le dimensioni della pagina dipendono dalla particolare macchina fisica). Lo spazio virtuale di ogni processo è divisa in due sottospazi di 2Gbyte ciascuno:

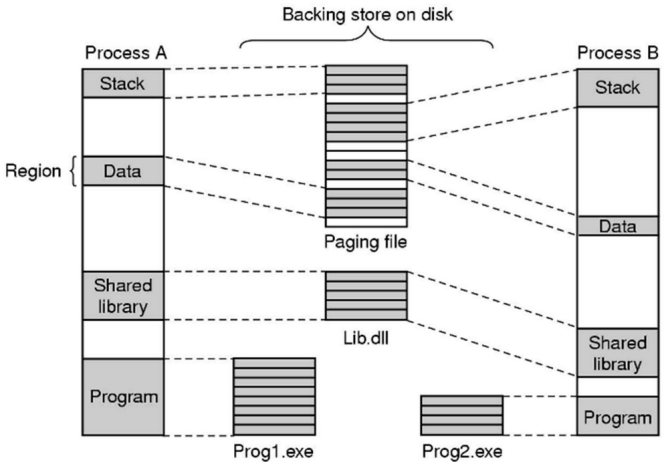
- il sottospazio virtuale inferiore è privato
- quello superiore è condiviso tra tutti i processi e mappa il sistema operativo.



9.13.1 Memoria Virtuale

Lo spazio virtuale è unico ed è suddiviso in regioni (simili ai segmenti di Unix). Ogni pagina logica può essere:

- *free*: se non è assegnata a nessuna regione e quindi un accesso ad una pagina free determina un page fault non gestibile (processo killato).
- *reserved*: è una pagina non ancora in uso ma che è stata riservata per espandere una regione e non viene mappata nella tabella delle pagine (e.g. riservata per l'espansione dello stack).
- *committed*: se appartiene a una regione già mappata nella tabella delle pagine. Quindi un accesso ad una pagina committed non presente in memoria risulta in un page fault, che determina il caricamento della pagina solo se questa non si trova in una lista di pagine eliminata dal working set.



9.13.2 Gestione dei page fault

Windows adotta l'algoritmo del wroking set (locale) ed working set è sinonimo di insieme di resident set. Le pagine residenti di ogni processo stanno in un range i cui valori di *max* e *min* sono inizializzati di default ma possono cambiare durante la vita del processo.

Quando avviene il page fault di un processo, la pagina richiesta viene sempre caricata in un blocco libero di memoria. Quindi il suo insieme residente aumenta di uno, se è maggiore del massimo, il *Working Set Manager* (l'equivalente del Page Daemon di Unix) scarica alcune pagine.

Anche Windows mantiene un certo numero di pagine libere per velocizzare il caricamento in memoria delle pagine. Questo viene assicurato da due processi speciali:

- *Balance Set Manager*: quando sono poche le pagine libere, ne scarica alcune in base al Wroking Set Manager.
- *Working Set Manager*: implementa la politica di rimpiazzo delle pagine.

9.13.3 Working Set Manager

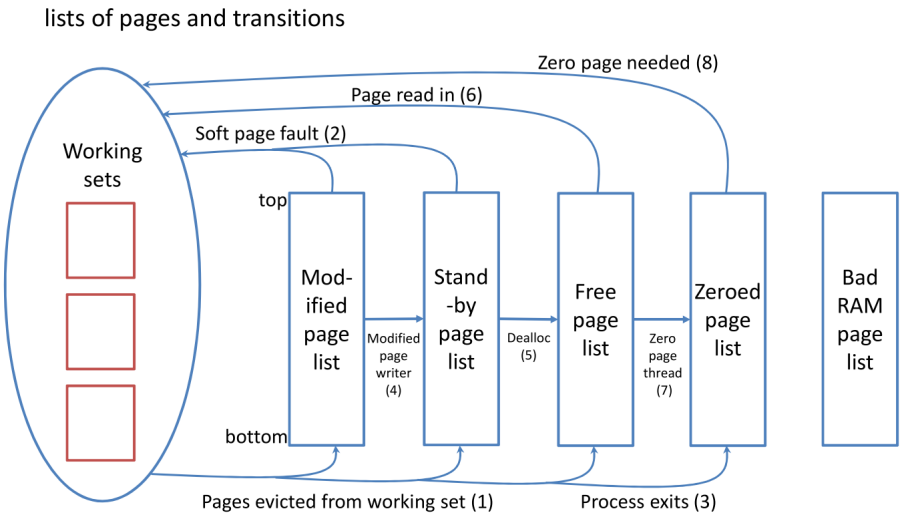
Per ogni processo col numero di resident set $>$ *min*:

- Per ogni pagina p col *reference bit* $R = 1$: `reset R and count(p)`
- Per ogni pagina p con $R = 0$: `increase count(p)`
- Seleziona $x - max$ pagine in ordine decrescente di *count(p)*

count(p) è un'approssimazione del tempo dell'ultimo riferimento della pagina p.

Se il numero di blocchi liberi rimane lo stesso basso, nonostante questa tecnica, allora vengono rimosse le pagine di altri processi con $x > min$ (diventa quindi un algoritmo globale).

9.13.4 Gestione delle pagine



Le pagine da scaricare, se sono state modificate vengono messe in una lista di pagine modificate (*Modified page list* altrimenti vanno in un'altra lista (*Stand-by page list*. Così se ho di nuovo bisogno di una pagina che è in stand-by non importa prenderla dal disco ma direttamente dalla lista.

10 File Systems: Introduction and Overview

10.1 The File System Abstraction

I computer devono essere in grado di memorizzare i dati in modo affidabile. Infatti, per funzionare del tutto, un computer, deve riuscire a memorizzare i programmi da eseguire e lo stesso sistema operativo.

Oggigiorno, chiunque utilizzi un computer, ha una certa familiarità col l'astrazione ad alto livello del file system. Il file system fornisce un modo, agli utenti, di organizzare i propri dati e di memorizzarli per un lungo periodo di tempo. Più precisamente, un *file system* è un'astrazione del sistema operativo che fornisce dati persistenti. Un *dato persistente* è memorizzato fintato che non viene eliminato esplicitamente, anche se il computer crasha o si spegne. I dati, a cui possiamo dare un nome, possono essere acceduti da un identificatore leggibile da noi umani che il file system associa al file.

Ci sono due cosa fondamentali che fanno parte all'astrazione del file system: i *files*, i quali definiscono un insieme di dati, e le *directories*, le quali definiscono nomi per file.

File. Un file è una collezione nota (nel senso ha un nome) di dati. I files forniscono un più alto livello di astrazione rispetto al sottostante dispositivo di storage: permettono con un nome di riferirsi ad una certa quantità di dati. E' più conveniente riferirsi ai dati con un nome che col numero di blocco di disco.

Le informazioni di un file hanno due parti: i metadati e i dati. I *metadati* di un file sono le informazioni che lo riguardano. Ad esempio, la dimensione, l'ultima modifica, il proprietario ecc.

I *dati* di un file possono essere qualsiasi informazione che viene messa da un utente o una applicazione. Dal punto di vista del file system, i dati di un file è solo un array di bytes. Le applicazioni possono usare questi bytes per memorizzare qualsiasi informazione che vogliono in qualsiasi formato scelgono. Alcuni dati hanno una semplice struttura. Ad esempio, un file testo ASCII contiene una sequenza di bytes che vengono interpretati come lettere nell'alfabeto. Al contrario, alcune strutture possono essere complesse. Ad esempio, un file .doc può contenere testo, informazioni di formattazione, oggetti ed immagini oppure un file ELF (Executable and Linkable File), che può contenere oggetti compilati e codice eseguibile ecc.

Directory. Le directories forniscono un nome per i files. In particolare, una *file directory* è una lista di nomi leggibili per gli umani e una struttura che mappa ogni nome ad un file o directory sottostante.

La stringa che identifica un file o una directory (e.g. /home/tom/Boia/deh.txt) viene chiamata *path*. Il simbolo / separata i componenti del path, e ogni componente rappresenta una entry nella directory.

Se si pensa ad una directory come un albero, allora la radice dell'albero è una directory chiamata, *root directory*. Path come /bin/ls che cominciano con / , si chiamano *path assoluti*. Sono interpretati come relativi alla root directory.

Path come TortaDiCeci/Non/Cecina che non iniziano con / , si chiamano *path relativi* e sono interpretati, dal SO, relativi alla *current working directory* del processo.

Il collegamento tra un nome è il file sottostante si chiama *hard link*. Se un SO permette multipli hard links per il solito file, allora la gerarchia della directory potrebbe non essere più un albero.

Volume. Un volume è una collezione di risorse memorizzate fisicamente che formano un dispositivo di storage logico.

Un volume è un'astrazione che corrisponde ad un disco logico. Nel caso più semplice, un volume corrisponde all'intero disco fisico. Alternativamente, un disco fisico può essere partizionato e memorizzare più volumi oppure alcuni dischi fisici possono essere combinati in modo che un singolo volume li abbracci tutti.

10.2 Access to files

Le operazioni per l'accesso ai file sono: *read* e *write*. Per eseguire queste operazioni su di un file è necessario reperire prima alcune sue informazioni: l'indirizzo nel disco, diritti di accesso ecc. Per questo, il file va prima aperto con l'operazione di *open*. Quando poi queste informazioni non mi servono più, chiameremo la *close*. In modo da deallocare le informazioni dalla memoria.

I metodi di acesso sono due: *sequenziale* e *diretto*. Sequenziale significa che per accedere ad un byte, devo aver dovuto letto i bytes precedenti. Diretti invece quando vado direttamente al byte. Questi due metodi sono indipendenti sia dal dispositivo fisico che dal file system.

10.2.1 Sequental access

Il file è una sequenza di records logici. Per accedere ad ogni record R_i, è necessario accedere prima a tutti i precedenti $(i-1)$ records:

Questo tipo di accesso di utilizza, implicitamente, quando scriviamo un testo.

Le operazioni utilizzate sono:

- *readnext:* leggere il prossimo record logico,
- *writenext:* scrive il prossimo record logico.

10.2.2 Direct access

Il file è un insieme $\{R_1, R_2, ..., R_N\}$ di record logici e l'utente può accedere direttamente ad ogni record logico.

Le operazioni utilizzate sono:

- *read(f, i, &V):* legge il record logico i del file f; i dati letti sono memorizzati nel buffer V,
- *write(f, i, &V):* scrive il contenuto del buffer V nel record logico i del file f.

10.3 UNIX File System API

Creazione ed eliminazione files

Creazione di un file	*create(pathName)*
Linking e unlinking	*link(existingName, newName)* *unlink(pathName)*
Creazione ed eliminazione directory	*mkdir(pathName)* *rmdir(pathName)*

Open e close

Preparazione accesso di un file	*fileDescriptor open(pathName)*
Rilascio delle risorse associate al file	*close(fileDescriptor)*

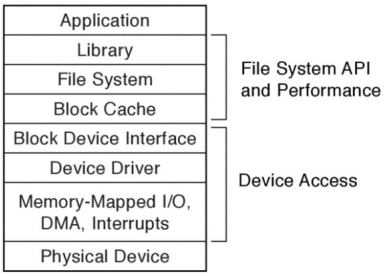
Accesso file

Lettura	*read(fileDescriptor, buf, len)*
Scrittura	*write(fileDescriptor, len, buf)*
Cambio posizione nel file	*seek(fileDescriptor, offset)*
Trasferisce tutte le modifiche del file nel disco	*fsync(fileDescriptor)*

10.4 Software Layers

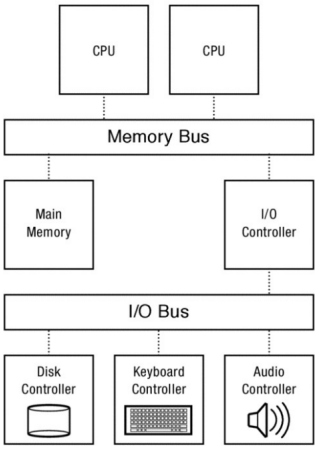
Il SO implementa l'astrazione del file system attraverso una serie di "strati" software. Questi strati hanno due compiti:

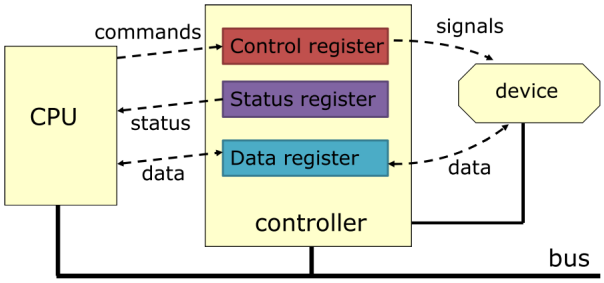
- **API e performance.** Il più alto livello dello stack software – librerie user-level, file systems kernel-level, ed il blocco cache del kernel – fornisce un API conveniente per accedere a file noti e lavorare per minimizzare l'accesso di storage via caching, scrittura bufferizzata e prefetching.
- **Accesso ai dispositivi.** Il livello più basso del software stack fornisce al SO i metodi per accedere ad un largo range di dispositivi I/O. I driver dei dispositivi nascondono i dettagli dello specifico hardware I/O fornendo codice hardware specifico per ogni dispositivo, e piazzando il codice dietro una semplice e più generale interfaccia che il resto del SO può utilizzare come una block device interface. I drivers sono eseguiti come normali codici a livello kernel, e che utilizzano la memoria ed il processore, ma devono interagire con i dispositivi I/O. La memoria ed il processore comunicano con i propri dispositivi I/O utilizzando Memory-Mapped I/O, DMA e Interruzioni.



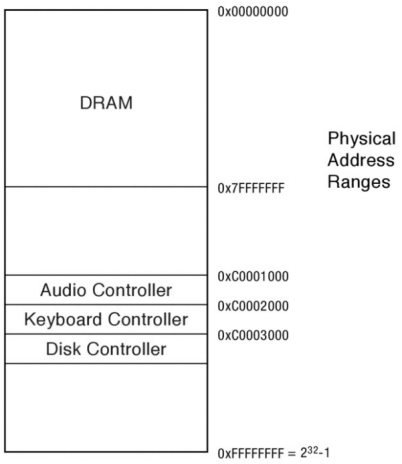
10.4.1 Accesso ai dispositivi

- **Memory-mapped I/O.** I dispositivi I/O sono solitamente connessi ad un bus I/O che è connesso il bus di memoria del sistema. Ogni dispositivo I/O ha un controller con un certo numero di registri che possono essere scritti e letti per trasmettere comandi e dati da e verso il dispositivo. Ad esempio, un semplice keyboard controller potrebbe avere un registro che può essere letto per capire quale tasto è stato premuto più recentemente e un altro registro che può essere scritto per accendere o spegnere la luce del caps-lock.





Per permettere ai registri di controllo I/O di essere letti e scritti, i sistemi implementano memory-mapped I/O. Memory-mapped I/O mappa ogni registro di controllo del dispositivo in un range di indirizzi fisici nel bus di memoria. Le letture e le scritture da parte della CPU a questi indirizzi fisici non vanno in memoria principale. Invece, vanno nei registri dei controller dei dispositivi I/O. In questo modo, i driver della tastiera possono capire il valore dell'ultimo tasto premuto leggendo dall'indirizzo fisico, diciamo, 0xC00002000.



- **DMA.** Molti dispositivi I/O, soprattutto quelli di memorizzazione, trasferiscono i dati in massa. Ad esempio, i SO non leggono una parola o due dal disco, tipicamente fanno trasferimenti di almeno qualche kilobytes alla volta. Invece che richiedere alla CPU di leggere o scrivere ogni parola, i dispositivi I/O posso accedere direttamente alla memoria. Quando si usa l'accesso diretto in memoria (DMA), i dispositivi I/O copiano un blocco di dati nella propria memoria interna e nella memoria principale del sistema.

  Per settare un trasferimento DMA, un semplice SO potrebbe usare memory-mapped I/O per fornire un indirizzo fisico, trasferire la lunghezza, e il codice dell'operazione al dispositivo. Poi, il dispositivo copia i dati nel o dall'indirizzo senza richiedere coinvolgimenti ulteriori del processore.
  Dopo aver settato un trasferimento DMA, il SO non deve utilizzare quell'indirizzo fisico per altri scopi finché il trasferimento DMA non è stato completato. Il SO, dunque, si appunta le pagine target in memoria così da non poter essere riusate.

- **Interrupts.** Il SO ha bisogno di sapere quando i dispositivi I/O hanno completato una richiesta o quando arriva un nuovo input esterno. Un'opzione è quella del *polling*, utilizzare ripetutamente memory-mapped I/O per leggere un registro di stato sul dispositivo. Siccome i dispositivi I/O sono spesso molto più lenti della CPU e siccome l'input ricevuto da un dispositivo può arrivare in tempi irregolari, è solitamente meglio, per i dispositivi, utilizzare un'interruzione per notificare importanti eventi al SO.

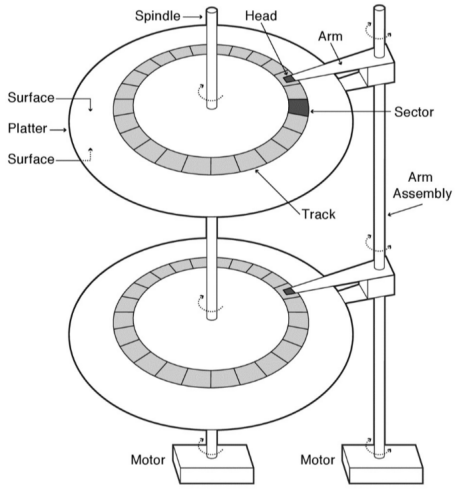
11 Storage devices

Nonostante oggigiorno ci siano dispositivi di storage con grande capacità e basso costo, hanno però una performance drasticamente peggiore della memoria volatile DRAM.

Discuteremo dei *dischi magnetici* e delle *memorie flash*. Entrambi sono largamente usati. I dischi magnetici vengono utilizzati principalmente nei server, workstation e desktop. Le memorie flash invece negli smartphone, tablet e laptop.

11.1 Dischi magnetici

Il disco magnetico è una tecnologia di storage non volatile. I driver del disco memorizzano i dati su di un film metallico molto fine sopra un disco di alluminio, vetro o ceramica che ruota rapidamente.



Solitamente è formato da due dischi (platter), uno sopra l'altro. Ogni disco ha due *superfici* (surface) (o *facce*) e le informazioni vengono memorizzate nelle *tracce* (track) (o *cilindro*), dove a sua volta sono suddivise in *settori* (sector). Per leggere e scrivere sul disco si utilizza un braccio (arm) con una testina (head). Essendo un apparecchio elettromeccanico, la parte meccanica apporta una certa riduzione di velocità.

Le *facce* sono in ordine crescente partendo dall'alto (faccia 0 e 1 sul platter 1 e faccia 2 e 3 sul platter 2 ecc.), le tracce sono contante in ordine crescente partendo dall'esterno.

Tutte le *tracce* di un certo numero si chiamano cilindro di quel numero (e.g. l'insieme costituito da traccia 0 faccia 0 e traccia 0 faccia 1 e traccia 0 faccia 2 ecc. si chiama cilindro 0). Le tracce sono grandi circa 1 micron e tra l'una e l'altra c'è una regione di spazio inutilizzato per evitare che la testina ne legga più di una.

Nei *settori* vengono memorizzate le informazioni. Per arrivare ad un'informazione ho bisogno di tre coordinate: #cilindro, #faccia e #settore. Un settore contiene 256/512 bytes di dati ma la lettura e la scrittura avviene per *blocchi* di settori grandi 2/4/8 KBytes in modo da avere un blocco di disco mappato in una pagina di memoria principale.

A livello di sistema operativo, il disco viene indirizzato in modo virtuale e quindi visto come un vettore di byte consecutivi. In quale è diviso prima in settori poi in facce ed infine in cilindri/tracce. Dunque dato il settore numero b, e una tripla $< c, f, s >$:

$$b = c(\#faces * \#sectors) + f(\#sectors) + s$$

dove *#faces* e *#sectors* sono rispettivamente, il numero di facce totali e il numero di settori di una traccia per faccia.

Conseguenze:

$$c = b\, div\, (\#faces * \#sectors)$$
$$f = (b\, mod\, (\#faces * \#sectors))\, div\, \#sectors$$
$$s = (b\, mod\, (\#faces * \#sectors))\, mod\, \#sectors$$

11.1.1 Performance

Il tempo che impiega il disco per leggere, dipende da tre fattori:

- *seek time* il tempo per muovere la testina sulla traccia giusta (1-20ms),
- *rotation time* il tempo per ruotare sul settore giusto (4-15ms),
- *transfer time* il tempo per il trasferimento.

I dati vengono trasferita prima dall'apparecchiatura al controllore e dopo dal buffer del controllore in memoria.

Dunque:

$$\text{Disk Latency} = \text{seek time} + \text{rotation time} + \text{transfer time}$$

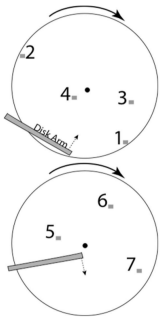
11.1.2 Disk Scheduling

Come mi conviene schedulare le richieste sul disco?

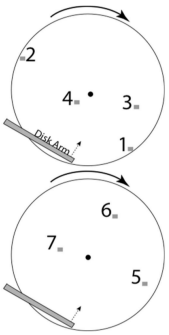
FIFO Le richieste al disco si soddisfano nell'ordine in cui sono arrivate. Arrivando richieste casuali, lo svantaggio è di fare avanti e indietro con la testina. Ovvero molta latenza.

Shortest seek time first Le richieste vengono ordinate in modo tale da minimizzare il seek time. Meglio della FIFO ma se la maggior parte delle richieste sono molto vicine, le richieste in posizioni lontane non vengono mai soddisfatte.

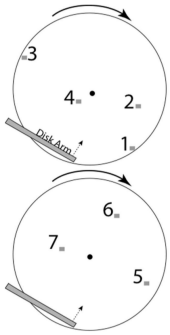
SCAN Si muove la testina in una direzione, finché tutte le richieste non sono soddisfatte e poi si inverte la direzione.



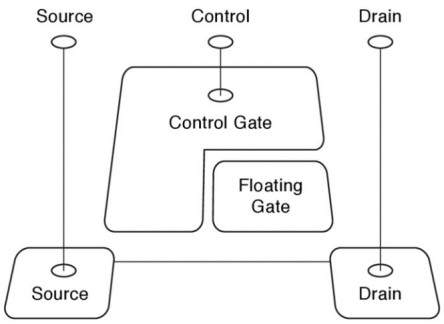
CSCAN Come la SCAN ma poi si riparte dala richiesta più lontana.



R-CSCAN Come la CSCAN ma si tiene anche conto di dove si trova il settore



11.2 Flash Memory (SSD)



Le scritture non possono avvenire sopra a qualcosa che era già stato scritto. Le celle di memoria devono essere prima "pulite/imbiancate". Per effettuare prima le scritture le memorie flash lasciano un certo numero di celle contigue che sono già state imbiancate in modo da poterci scrivere subito. Questi blocchi imbiancanti tipicamente, vanno da 128 a 512 KB. I blocchi fisici sono grossi come le pagine di memoria principale in modo da avere una corrispondenza uno a uno (2/4 KB).

Per evitare il costo di pulizia per ogni scrittura, le pagine vengono pulite in anticipo. In questo modo ho sempre pagine pulite per effettuare una scrittura. Questo significa che la scrittura non è arbitraria ma è possibile effettuarla solo in pagine precedentemente imbiancate. Ci sarà una lista di quelle pagine che devono essere pulite e questa operazione si svolgerà in background in modo da non disturbare la CPU.

In realtà avrò tre liste: una per le pagine occupate di ogni file, una per quelle pulite e una per quelle che devono essere pulite. Se si vuole riscrivere un blocco, si scrive in una pagina pulita e si mette quella vecchia nella lista di quei blocchi che devono essere pulite.

Una pagina fisica in memoria flash può essere scritta solo un numero limitato di volte dopodiché si danneggia.

11.2.1 Flash Translation Layer

Per sapere dove sono le pagine di un determinato file, viene utilizzato il *Flash Translation Layer*. Ovvero il livello di traduzione degli indirizzi per le memorie flash fatto a firmware. Mappa le pagine logiche in quelle fisiche. Quindi il SO indirizza le pagine in memoria flash con indirizzi logici, la traduzione la effettua il flash translation layer.

Il flash translation layer dovrà avere le liste descritte precedentemente e anche una lista delle pagine che sono danneggiate.

Il comando del file system per interagire con le memorie flash si chiama `TRIM` e non `read` e `write` come sul disco magnetico.

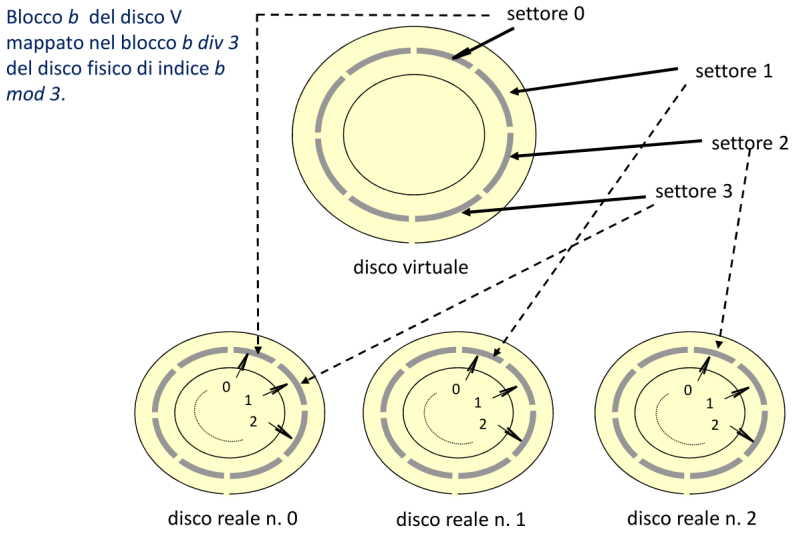
11.3 RAID

Se ho la necessità di gestire un sistema di calcolo abbastanza grande, non basta l'utilizzo di un solo disco. L'organizzazione utilizzata oggigiorno si chiama *RAID (Redundant Arrays of Independent Disks)*. Il RAID è un sistema che "spalma" i dati in maniera ridondante in dischi multipli in modo da tollerare malfunzionamenti di singoli dischi. Il SO vede i vari dischi come un unico disco virtuale grande quanto la somma delle singole dimensione dei dischi.

Si sfrutta il parallelismo per ottenere un accesso più veloce: i blocchi consecutivi di uno stesso file sono distribuiti sui dischi dell'array in modo da permettere operazioni contemporanee.

Si sfrutta la ridondanza per accrescere l'affidabilità: la ridondanza permette di correggere gli errori di certe classi.

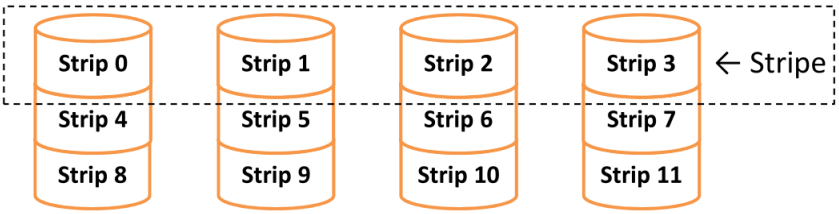
Per la scrittura/lettura simultanea, la memoria sarà interallacciata:



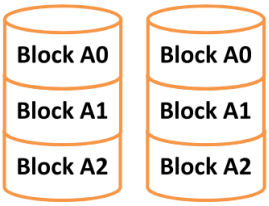
11.3.1 Livelli RAID

Definiamo lo *striping* come il metodo che il RAID usa per incrementare le performance dell'insieme di hard disk e consiste in pratica nel dividere i file in segmenti (strip). Uno *strip* è un insieme di diversi blocchi sequenziali di un disco. L'insieme degli strip di un file vengono dunque divisi tra i vari dischi. Si chiama *stripe* invece, un insieme di strip ed il suo lo strip di parità.

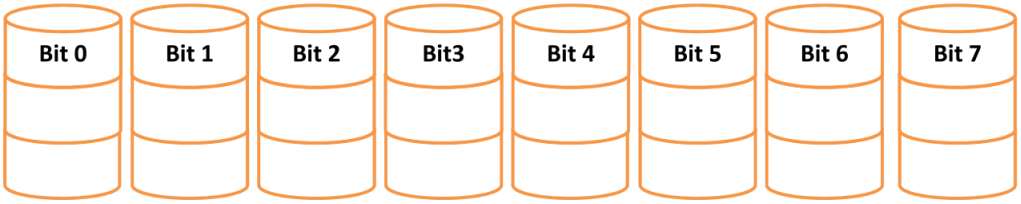
Livello 0 *Dischi asincroni, nessuna ridondanza.* I dati di un file sono divisi tra i vari dischi e così la lettura e la scrittura avviene molto più velocemente. La capacità totale è la somma delle capacità dei dischi. Basta che si rompa un disco per perdere i dati.



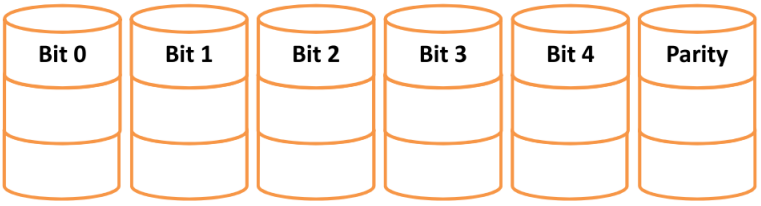
Livello 1 *Dischi asincroni, disco con copie ridondanti (mirror).* Le scritture su un disco comportano le stesse scritture sull'altro. E' una vera e propria copia di backup. Se uno si rompe c'è sempre l'altro. Se il controller lo permette, si può aumentare le prestazioni di lettura leggendo un po' da un disco e un po' da un altro.



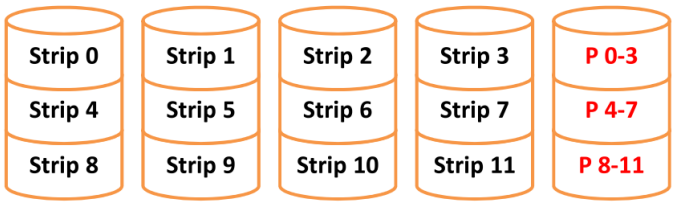
Livello 2 *Dischi sincroni, i dischi ridondanti contengono codici per la correzione degli errori.* I dati sono divisi al livello di bit (invece che di blocco) (stripe bit-level) e usa un codice di Hamming per la correzione d'errore che permette di correggere errori su singoli bit. Questi dischi sono sincronizzati dal controllore, in modo tale che la testina di ciascun disco sia nella stessa posizione in ogni disco. Lettura o scrittura contemporanea di tutti i bit della strip.



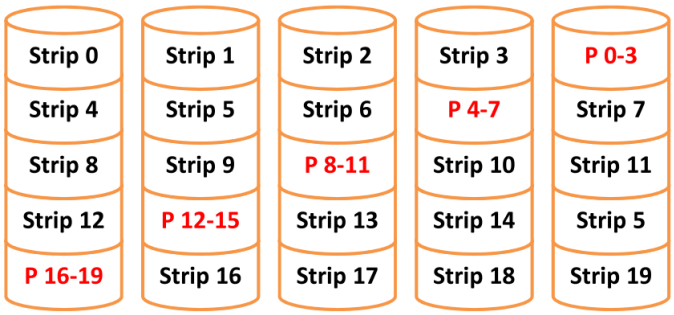
Livello 3 *Dischi sincroni, un solo disco ridondante.* Si usa una divisione al livello di bit o byte (stripe bit o byte level), con un disco dedicato alla parità.



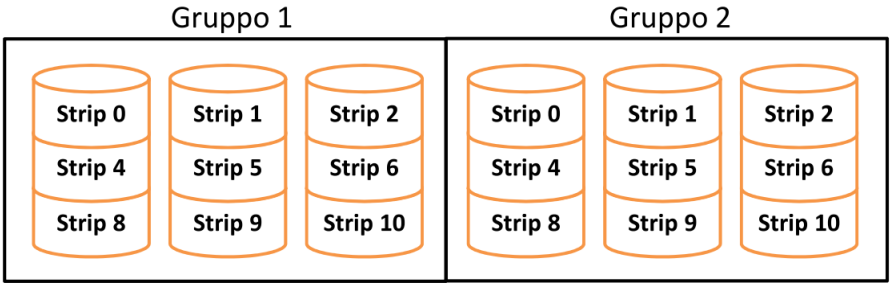
Livello 4 *Dischi asincroni, un solo disco ridondante.* Simile al RAID 3. Si usa una divisione al livello di byte (stripe byte-level) con un disco dedicato alla parità. Se si guasta il disco di parità non possiamo più correggere gli errori.



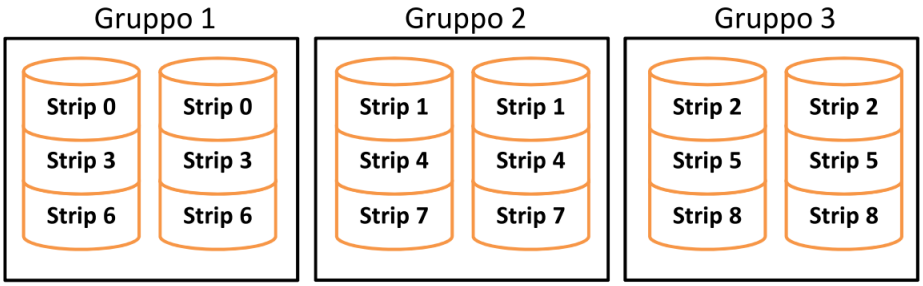
Livello 5 *Dischi asincroni, parità distribuita tra tutti i dischi.* Come il RAID 4 ma la parità viene suddivisa tra i dischi.



Livello 1+0 *Dischi asincroni, mirror di stripes.* I dischi sono divisi in gruppi e all'interno di ogni gruppo utilizza RAID 0. Un gruppo è il mirror dell'altro.



Livello 0+1 *Dischi asincroni, stripe di mirror.* I dischi sono divisi in gruppi, ogni gruppo utilizza RAID 1 e i vari gruppi sono interallacciati come nel RAID 0.

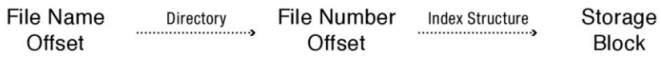


12 File Systems

Il *File System* è la parte del SO che gestisce le informazioni, quindi il recupero (scrittura e lettura), delle informazioni che sono memorizzate sul disco. L'utente interagisce con i propri dati come se questi fossero memorizzati in una memoria virtuale. Il SO maschera anche il memory-mapped file. Noi accediamo tramite indirizzi virtuali e la traduzione in indirizzo fisico (sul disco) la svolge il file system. Sarà il SO (file system) a sapere se l'informazione sta già in memoria principale o sul disco. Tutto questo vale se ho un disco magnetico perché con una memoria flash, lo stesso SO vede una memoria virtuale e la stessa memoria flash, via firmware, farà la traduzione.

La situazione è analoga a quella che riguarda i processi, la memoria virtuale dei file è contigua ma fisicamente sono pezzi di dati sparsi. Ci sono però due differenze sostanziali: di dimensioni e dell'apparecchiatura fisica utilizzata. Di dimensioni perché si parla di dati di dimensione molto più grande e dell'apparecchiatura perché per svolgere le operazioni il più velocemente possibile, devo tenere conto delle caratteristiche fisiche del disco. Come già visto, anche qui ho il problema della frammentazione: conviene avere blocchi piccoli o grandi? Si fa una via di mezzo.

Dunque come abbiamo già detto, i file systems devono mappare i nomi dei file e l'offset in blocchi fisici in modo da avere un accesso efficiente. Nonostante ci siano diversi file systems, la maggior parte è basata su quattro idee chiave: directories, index structures, free space maps e locality heuristics.



Directories and index structures. Il file system mappa i nomi dei file e l'offset in uno specifico blocco di disco, in due step:

Primo, utilizza le directories per mappare i nomi dei file in linguaggio umano in numeri. Le directories sono spesso semplici file speciali che contengono la lista della mappatura "file name -¿ file number".

Secondo, una volta che il nome del file è stato tradotto in un numero, il file system utilizza una *index structure* per localizzare i blocchi del file. Spesso la index structure è un albero.

Free space maps. I file system implementano le *free space maps* per tracciare quali blocchi di disco sono liberi e quali sono in uso ogni volta che un file cresce o si modifica. Come minimo una free space map deve far in modo che il file system possa trovare un blocco libero quando un file ha bisogno di crescere, ma siccome la località spaziale è importante, i moderni file system implementano le free space maps in modo che il file system trovi un blocco libero vicino alla locazione cercata. Ad esempio, molti file system implementano le free space maps con bitmaps.

Locality heuristics. Le directory e le index strucutres permettono ai file system di localizzare i file data e i metadati desiderati, a prescindere da dove sono fisicamente memorizzati, e le free space maps permettono di localizzare lo spazio libero vicino. Questi meccanismi permettono ai file system di avere varie politiche per decidere dove un dato blocco di un dato file deve essere memorizzato.

Questi sono gli attuali file system utilizzati nei moderni SO:

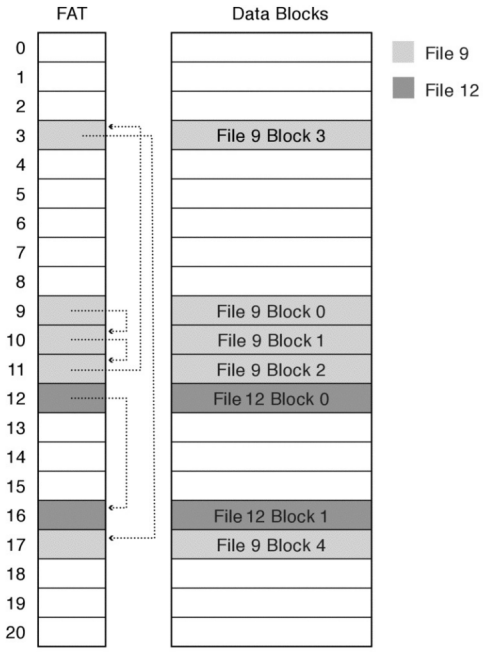
	FAT	FFS	NTFS
Index structure	Linked list	Tree (fixed, asym.)	Tree (dynamic)
granularity	block	block	extent
free space allocation	FAT array	Bitmap (fixed location)	Bitmap (file)
Locality	defragmentation	Block groups + reserve space	Extents Best fit defrag

- **FAT.** Il file system *Microsoft File Allocation Table (FAT)* nasce negli anni '70. La FAT utilizza una index structure estremamente semplice: una *linked list*. La FAT oggigiorno è ancora largamente usata in dispositivi come pennine USB e camere digitali.
- **FFS.** L'*Unix Fast File System (FFS)* è stato rilasciato a metà degli anni '80 e conserva molte delle strutture dati nel file system originale Unix di Ritchie e Thompson. L'FFS utilizza un *tree-based multi-level index* come index structure per migliorare l'efficienza degli accessi random ed utilizza una collezione di locality heuristics per ottenere una buona località spaziale. Oggigiorno i popolari file system ext2 e ext3 sono basati sull'FFS.
- **NTFS.** Il *Microsoft New Technology File System (NTFS)* è stato introdotto nei primi anni '90 come alternativa al FAT. Come per l'FFS, l'NTFS utilizza una index structure tree-based, ma l'albero è più flessibile di quello del'FFS. In più, l'NTFS ha ottimizzato anche la propria index structure. Oggigiorno l'NTFS rimane il principale file system dei SO Microsoft.

12.1 Microsoft File Allocation Table (FAT)

La più recente versione di FAT è la FAT-32 che supporta volumi con massimo 2^{28} blocchi e file fino a $2^{32} - 1$ bytes.

Index structures. La FAT è così nominata per la sua *tabella di allocazione di file*, un array di 32-bit entries in un area riservata del volume. Ogni file nel sistema corrisponde ad una linked list di FAT entries, dove ogni FAT entry contiene un puntatore alla prossima FAT entry del file od un valore speciale (EOF). La FAT ha una sola entry per ogni blocco del volume, e i blocchi del file sono i blocchi che corrispondono alle FAT entries del file: se la FAT entry i è la j-esima FAT entry di un file, allora il blocco di memoria i è il j-esimo blocco dati del file.



Dunque, dato il numero del file possiamo trovare la prima FAT entry e ed il blocco del file, e data la prima FAT entry possiamo trovare il resto delle entry di quel file ed i relativi blocchi.

Free space tracking. Se un blocco è libero, allora contiene 0. In questo modo, il file system può trovare un blocco libero scansionando la FAT. Ci sarà una lista concatenata per le entry libere.

Pro:

- E' facile trovare blocchi liberi
- E' facile fare un append ad un file
- E' facile eliminare un file

Contro:

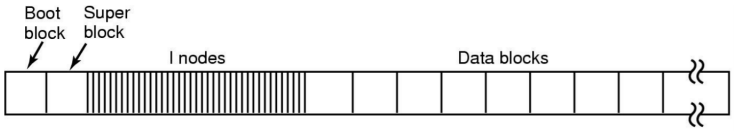
- Dimensione della FAT (per accedere alle informazioni la FAT la carico in memoria principale)
- Limitazione dimensione file (in una FAT32 posso utilizzare 32bit per rappresentare la dimensione di un file, quindi la massima dimensione è $2^{32} - 1$Byte)

- Metadati limitati e no protezione: non ho informazioni sul proprietario del file quindi tutti possono leggere tutti i file.
- Frammentazione (a lungo andare i blocchi liberi cominciano ad essere sparpagliati)

Se ho più file di quanto la FAT può indirizzare, non li vedrei perché non non posso fare la traduzione da indirizzo virtuale a fisico. Quindi il numero massimo di blocchi è dato dalla dimensione del file system. Se ho L bit, posso indirizzare 2^L blocchi. Se ogni blocco del disco è di B Byte, in totale il file system può rappresentare $B * 2^L$ Bytes. Se ogni elemento della FAT è grande N Bytes, la FAT è grande $N * 2^L$ Bytes.

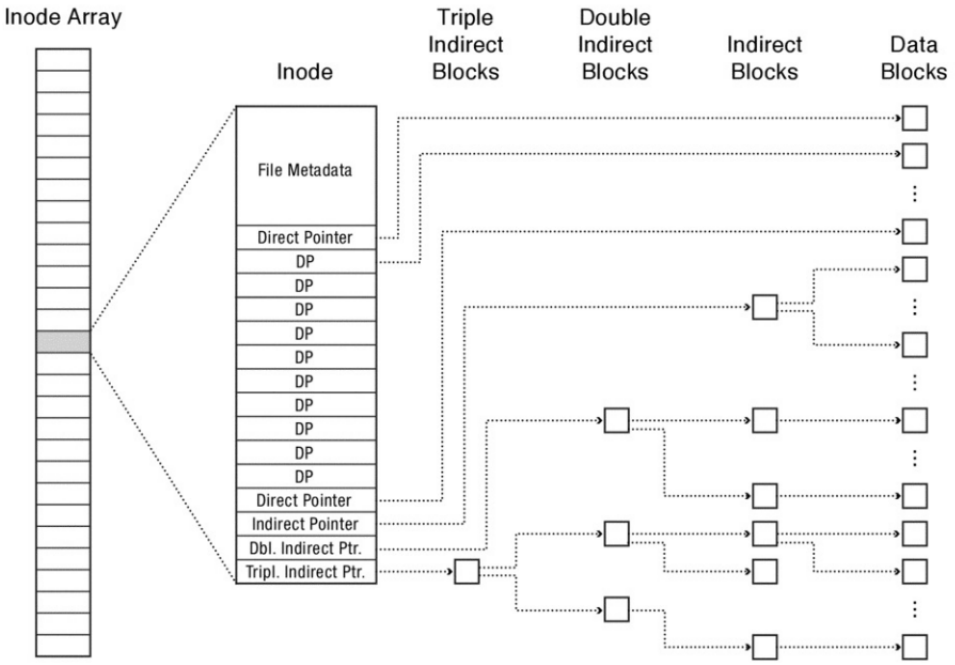
12.2 Berkeley Fast File System (FFS)

L'Unix Fast File System (FFS) illustra importanti idee sia per indicizzazione dei blocchi dei file (così possono essere localizzati velocemente) sia per posizionare dati sul disco per avere una buona località.



In particolare, l'index structure dell'FFS è chiamata *multi-level index*. E' una struttura ad albero che permette all'FFS di localizzare qualsiasi blocco di un file ed è efficiente sia per grandi che per piccoli file.

Index structures. Per mantenere traccia dei blocchi dati che appartengono ad ogni file, l'FFS utilizza un albero fisso e asimmetrico chiamato *multi-level index*.



Ogni file è un albero con foglie che rappresentano blocchi di dati di dimensione fissa (e.g. 4KB). Ogni albero di un file è radicato ad un *inode* che contiene i metadati del file (e.g. il proprietario, permessi di accesso, data di creazione, ultima modifica ecc.)

L'inode di un file (root) contiene inoltre un array di puntatori per localizzare i blocchi di dati del file (le foglie). Alcuni di questi puntatori, puntano direttamente ai blocchi dati altre invece puntano a nodi interni all'albero. Tipicamente, un nodo contiene 15 puntatori. I primi 12 sono *puntatori diretti* e puntano ai primi 12 blocchi dati del file.

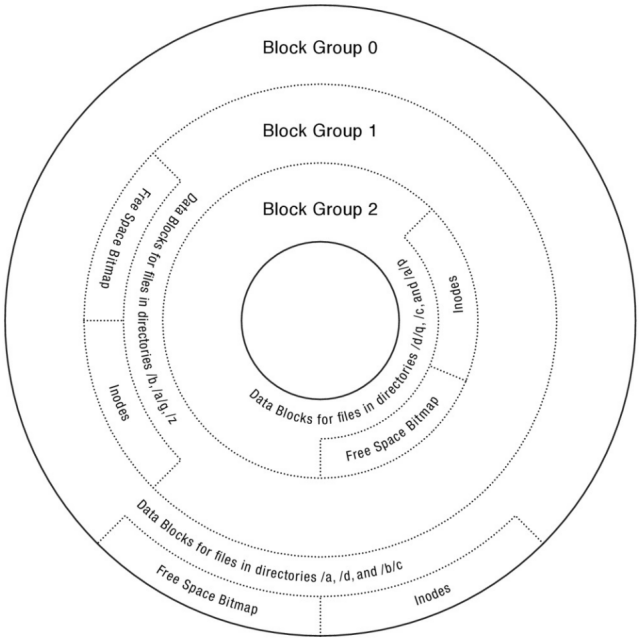
Il 13esimo puntatore è un *puntatore indiretto*, il quale punta ad un nodo interno all'albero, chiamato *blocco indiretto*; un blocco indiretto è un blocco di disco ma che contiene un array di puntatori diretti. Con blocchi da 4KB e puntatori ai blocchi da 4Bytes, un blocco indiretto può contenere al massimo 1024 puntatori diretti, che permettono di indirizzi un file grande 4MB.

Il 14esimo puntatore è un *puntatore doppiamente indiretto*, il quale punta ad un nodo interno all'albero chiamato *blocco doppiamente indiretto*; un blocco doppiamente indiretto è un array di puntatori indiretti, ogni dei quali punta ad un blocco indiretto.

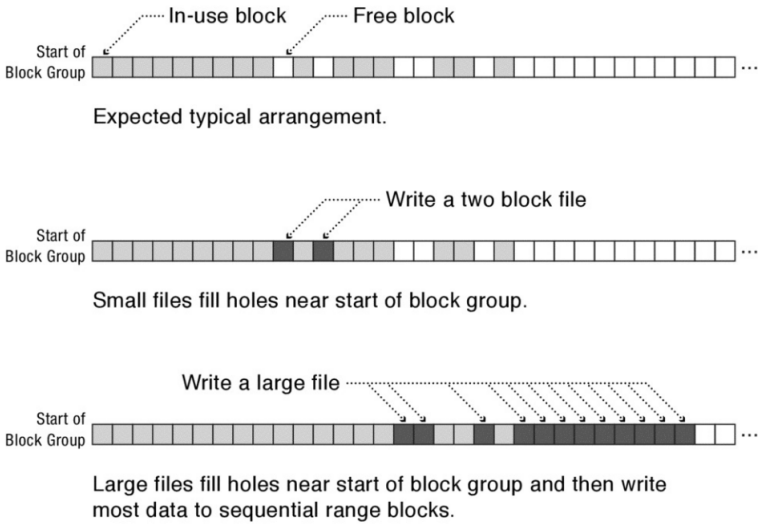
Infine, il 15esimo puntatore è un *puntatore triplamente indiretto*, il quale punta ad un nodo interno all'albero chiamato *blocco triplamente indiretto*; contiene un array di puntatori doppiamente indiretti.

Tutti gli inodes del file system si trovano in un *array di inode* che è memorizzato in una locazione fissa nel disco. Il numero di un file, chiamato *inumber* nell'FFS, è l'indice nell'array di inode: per aprire un file (e.g. Area51.txt), dobbiamo vedere all'interno della sua directory per trovare il suo numero(e.g. 666) e poi dobbiamo vedere l'entry appropriata nell'array di inode (e.g. entry 424242) per trovare i suoi metadati.

L'FFS divide il disco in *block group*: i file della stessa directory risiedono nel solito gruppo e le sottodirectories risiedono in gruppi differenti. Ogni gruppo ha una propria *Free Space Bitmap* per localizzare i blocchi liberi e un proprio array di inode.



I blocchi liberi di allocano la politica *First Fit*: i blocchi da caricare vengono posizionati nei primi blocchi liberi che trovo nel block group.



Pro:

- Traduzione indirizzi e località efficiente sia per file piccoli che grandi
- Non occupa molta memoria

Contro:

- Se il file è piccolo ho uno spreco (e.g. un file di 1 Byte richiede tutto un inode)
- Per evitare la frammentazione (dovuta alla first fit), bisogna mantenere il 10-20% di spazio libero
- Decodifica inefficiente quando il file è contiguo

12.3 Microsoft New Technology File System (NTFS)

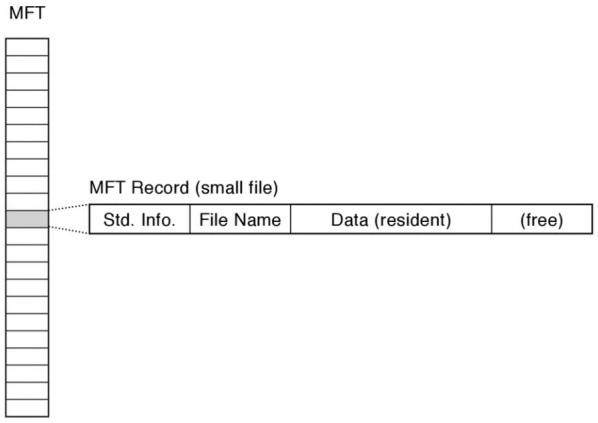
Rilasciato nel 1993, l'NTFS ha perfezionato la FAT con molte funzionalità incluso una nuova index structures per migliorare le performances.

Index structures. Mentre l'FFS traccia i blocchi di file con un albero fissato, l'NTFS e molti file system recenti (ext4, btrfs ecc), tracciano le *estensioni* con un alberi flessibili. Invece di tracciare i singoli blocchi, l'NTFS traccia le *estensioni*. Una estensione è una regione di dimensione variabile di un file che è memorizzata in una regione contigua sul disco.

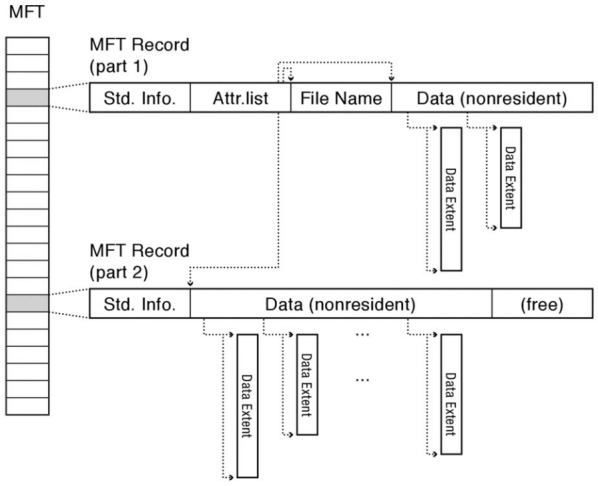
Le radici degli alberi sono memorizzate nella *Master File Table (MFT)* che è simile all'array di inode dell'FFS. Questa tabella è un array di record grandi 1KB, ognuno dei quali memorizza delle informazioni.

Ci sono delle informazioni standard, ovvero metadati, poi il nome del file e se il file è molto piccolo ci salvo direttamente nel record i suoi dati. Se non avessi spazio ci metto la base e il limite (base and bound) per ogni estensione (ovvero le regioni di disco dove risiedono le varie parti del file).

Se il file è piccolo:



Se il file è medio:



In generale, se non mi basta si crea un albero simile ai puntatori indiretti dell'FFS e si va in profondità.