

# Lezione 1

---

## Introduzione e definizione di sistema operativo

Una prima definizione di **sistema operativo** che si diede, negli anni 60, fu la seguente: il software che controlla l'hardware, in particolar modo che ne gestisce le risorse.

In realtà ora il discorso non è più così semplice per via della continua evoluzione che, sia software, che hardware, hanno subito col passare degli anni. In particolare, l'hardware di oggi necessita di una grandissima varietà di applicazioni software (o **processi**), ma affinché questi ultimi funzionino efficacemente devono essere eseguiti in maniera **concorrente**: i processi devono accedere alle risorse del processore uno alla volta, seguendo uno specifico ordine, ed è compito del sistema operativo garantire che questo tipo di lavoro avvenga in totale sicurezza ed efficacia. È bene tenere a mente infatti che solo un processo per volta può essere in esecuzione sul processore, quindi lo **switch tra i processi** è di fondamentale importanza, questo ci fa capire che i sistemi operativi hanno anche il fondamentale compito di essere **gestori di risorse**.

Di conseguenza, possiamo definire il **sistema operativo**, come uno strato software che separa hardware e software, permettendo l'interazione tra le due componenti. Il cuore del sistema operativo è rappresentato dal **kernel**, ovvero il software che contiene i principali componenti del sistema operativo.

Una caratteristica altrettanto importante è la gestione di risorse condivise, ogni processo avrà bisogno delle proprie risorse, e capita di tanto in tanto, che più processi necessitino delle stesse risorse contemporaneamente. È quindi necessario implementare un meccanismo di **mutua esclusione** che garantisce la condivisione sicura di una certa risorsa garantendo al contempo che quella risorsa sia in futuro disponibile per altri processi così da evitare **deadlock** o **posticipazioni infinite**.

## Cenni storici

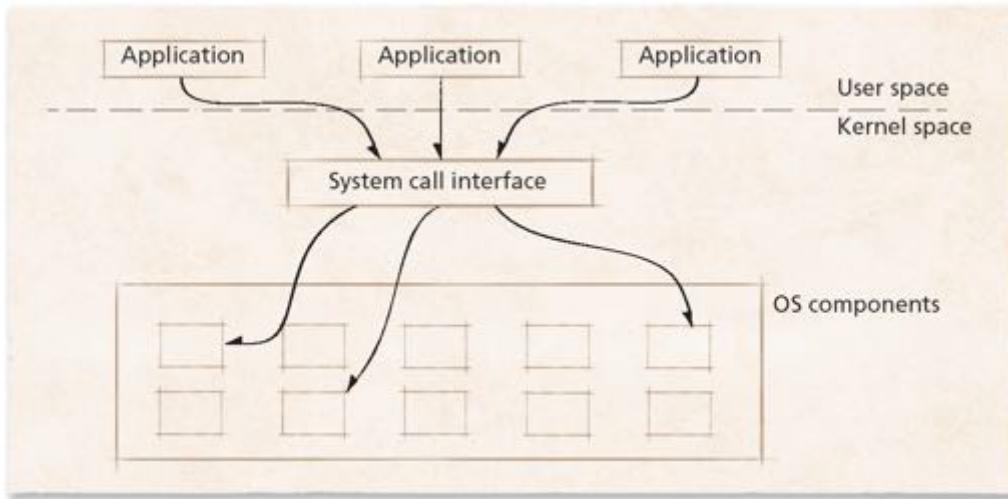
Nei primi anni 50 nasce il primo sistema operativo totalmente sviluppato da IBM per le sue macchine. Questo primo sistema operativo poteva compiere un solo **job** alla volta: un **job** costituiva l'insieme delle istruzioni di programma corrispondenti a un particolare compito computazionale, chiamato **task**. I job, all'epoca, rimanevano in esecuzione per giorni.

Agli albori dell'informatica i sistemi operativi nacquero con lo scopo di adattarsi ad uno specifico hardware, ma negli anni 60 IBM creò un sistema chiamato System360 che riuscì a mettere insieme diversi OS in grado di gestire diversi sistemi hardware: si poteva scrivere codice leggibile ed eseguibile su macchine diverse, oltre che la retrocompatibilità. Sempre in questo periodo nacquero i **sistemi multiprogrammati**, ovvero quelli in grado di gestire più job allo stesso tempo. Con questi sistemi nasce l'esigenza di condividere le risorse tra più processi.

Una successiva novità furono i **sistemi timesharing** che diminuirono i **tempi di turnaround**, ovvero il tempo trascorso tra l'invio di un job e il suo completamento, introducendo così gli **utenti interattivi**, l'utente poteva rimanere in attesa della esecuzione del job interagendo con esso.

In seguito, con l'avvento dei microchip i calcolatori si sono ridotti molto in dimensioni. Nascono così i primi PC, con le prime interfacce grafiche così da essere fruibili anche da non esperti del settore. Negli anni successivi si affaccia sul mercato windows che prende il suo nome dalla possibilità di gestire i processi sotto forma di finestra interattiva.

Vediamo ora un'utile grafico per comprendere come avvengono le interazioni tra applicazioni e sistema operativo:



Il sistema operativo fornisce una serie di API, utilizzabili dagli utenti, che hanno lo scopo di mettere a disposizione le cosiddette chiamate di sistema (**System call**) per mezzo delle quali un programma chiede al sistema operativo un certo lavoro.

Nella figura in alto vediamo due zone separate da una linea tratteggiata: lo **user space** è quel livello dove si interfaccia l'utente tramite le applicazioni e che contiene una serie di componenti software che non sono parte del sistema operativo e dunque non ne hanno accesso diretto, se non tramite una **System call interface**, come accennato prima. Al di sotto della linea tratteggiata troviamo invece il **kernel space** che contiene invece componenti software che sono parte del sistema operativo e ne hanno diretto accesso.

Ogni sistema ha le sue peculiarità, e per questo motivo ognuno di essi è preferibile ad un altro in base al task che si vuole eseguire. Possiamo trovare sistemi operativi, con funzionalità e gestione delle risorse differenti, in molti ambienti:

- **Embedded systems**

- - Caratterizzati da un set ristretto di risorse specializzate.
  - Girano su dispositivi quali smartphone e tablet.
  - La loro efficienza dipende dalla quantità di codice usata, che deve essere minima.

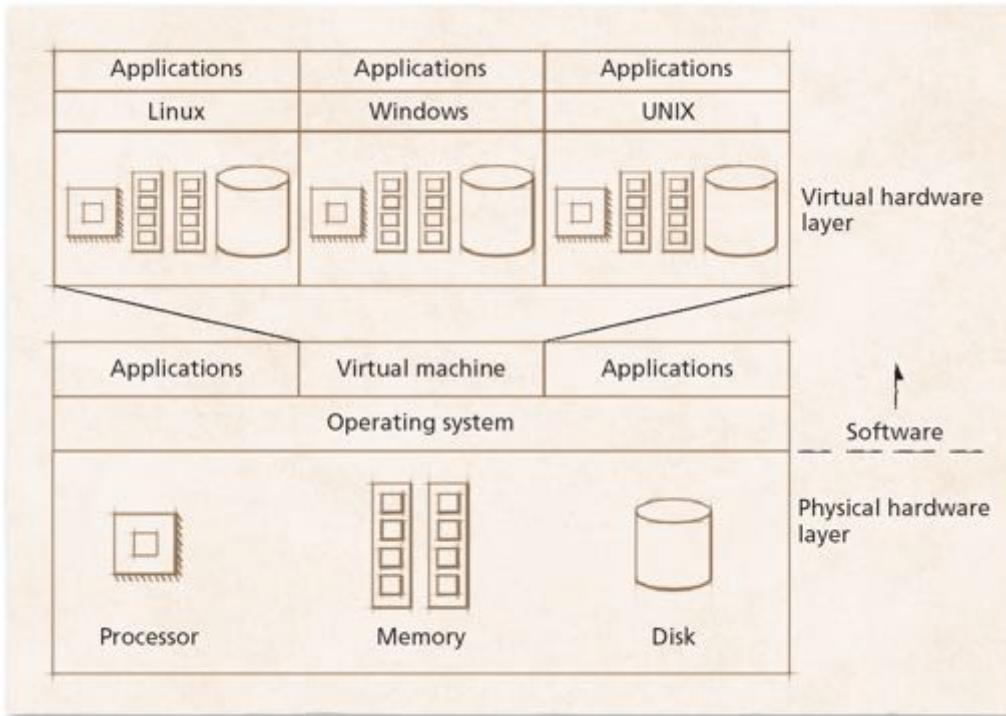
- **Sistemi real time**

- - Richiedono che i compiti siano svolti entro un preciso lasso di tempo, spesso anche molto breve.
  - Consentono ai processi di rispondere immediatamente agli eventi critici.

- **Virtual machine**

- - Sono astrazioni software di un computer che spesso esegue un'applicazione appoggiandosi al sistema operativo della macchina.
  - Un OS a macchina virtuale gestisce le risorse fornite esclusivamente dalla macchina virtuale.
  - Danno la possibilità di eseguire più istanze di sistemi operativi diversi contemporaneamente
  - Lo scopo principale è la simulazione: si utilizzano hardware e software della macchina nativa per emulare hardware o software di macchine non presenti nel sistema. Una VM può infatti creare componenti software che rappresentano i contenuti di sistemi fisici quali memoria, hard disk ecc..
  - Sono meno efficienti delle macchine reali poiché l'accesso indiretto o simulato all'hardware incrementa il numero di operazioni software per la risoluzione di un'operazione hardware.

Concludiamo vedendo una schematizzazione delle virtual machine:



### Struttura del kernel

L'insieme di componenti software che compongono il nucleo dell'OS si trovano nel **kernel**: troviamo qui lo **schedulatore di processi**, il **memory manager**, un **gestore dei dispositivi di I/O**, un **gestore di comunicazione tra processi (IPC)**, un **gestore di file system**. Vediamoli nel dettaglio singolarmente:

- **Schedulatore dei processi**: determina quando e per quanto un processo sarà in esecuzione sul processore.
- **Memory manager**: il gestore della memoria determina quando e quanta memoria allocare per ogni singolo processo e cosa fare nella eventualità in cui la memoria finisce.
- **Gestore di dispositivi I/O**: soddisfa richieste I/O da e verso i dispositivi hardware.
- **Gestione della comunicazione tra processi (InterProcess Communication)**: permette la comunicazione tra i processi.
- **Gestore del file system**: organizza raccolte di dati su dispositivi di memorizzazione e fornisce una interfaccia per accedervi.

Compito del kernel è quello di gestire la esecuzione concorrente dei processi, in particolare, le singole componenti di un programma che vengono eseguite indipendentemente per svolgere compiti diversi usando però uno stesso spazio di memoria per condividere risorse tra di loro, vengono chiamati **thread**. I **thread** sono indipendenti l'uno dall'altro, ma appartengono allo stesso programma e di conseguenza condividono anche lo stesso spazio in memoria.

Vediamo altre caratteristiche che devono avere i sistemi operativi:

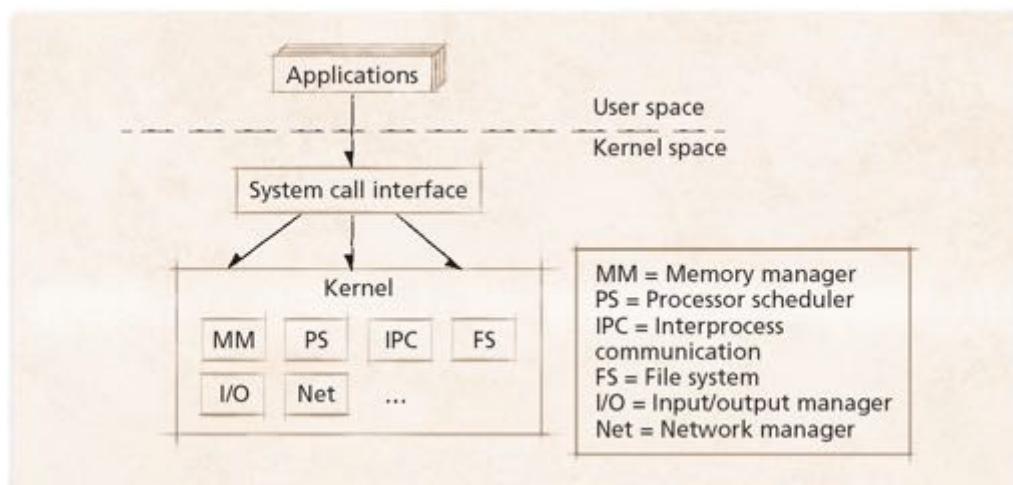
- **Efficienza**: devono avere un alto throughput e un basso tempo di turnaround, ovvero il tempo di completamento di un job. Per throughput intendiamo la quantità di lavoro che un processore può eseguire in un certo intervallo di tempo.
- **Robustezza**: devono essere tolleranti ai guasti e affidabili. Il sistema non deve bloccarsi tutto nel caso in cui ci siano errori nelle applicazioni.
- **Scalabilità**: devono essere in grado di supportare e utilizzare le risorse che vengono aggiunte man mano al sistema.
- **Portabilità**: devono essere progettati per funzionare su un gran numero di configurazioni hardware.

- **Espandibilità:** devono essere in grado di adattarsi bene alle nuove tecnologie così da svolgere compiti che in fase di progettazione non erano stati previsti.
- **Sicurezza:** devono impedire ad utenti e software vari di accedere a risorse e servizi senza opportuna autorizzazione.
- **Interattività:** devono permettere all'utente di rispondere velocemente alle azioni degli utenti.
- **Usabilità:** devono essere semplici ed intuitivi nell'utilizzo, anche grazie al supporto di opportune GUI (Graphic User Interface).

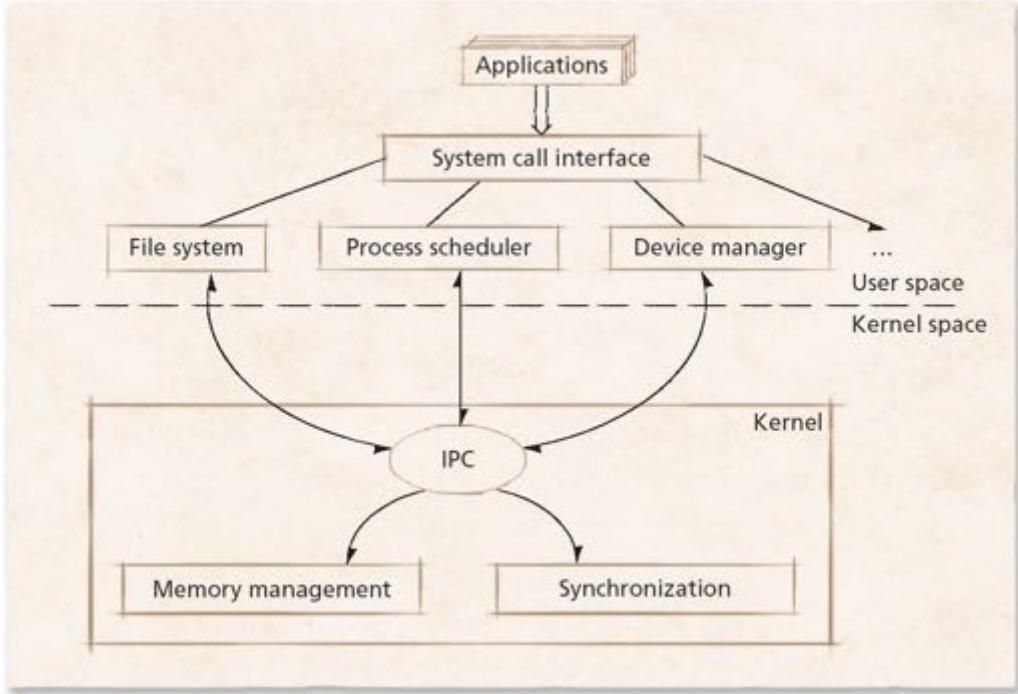
## Architetture dei sistemi operativi

Esistono varie architetture in base alle quali varia l'organizzazione delle strutture del sistema operativo. Vediamo ora le 3 principali architetture; gli attuali sistemi operativi utilizzano un mix di queste 3 architetture:

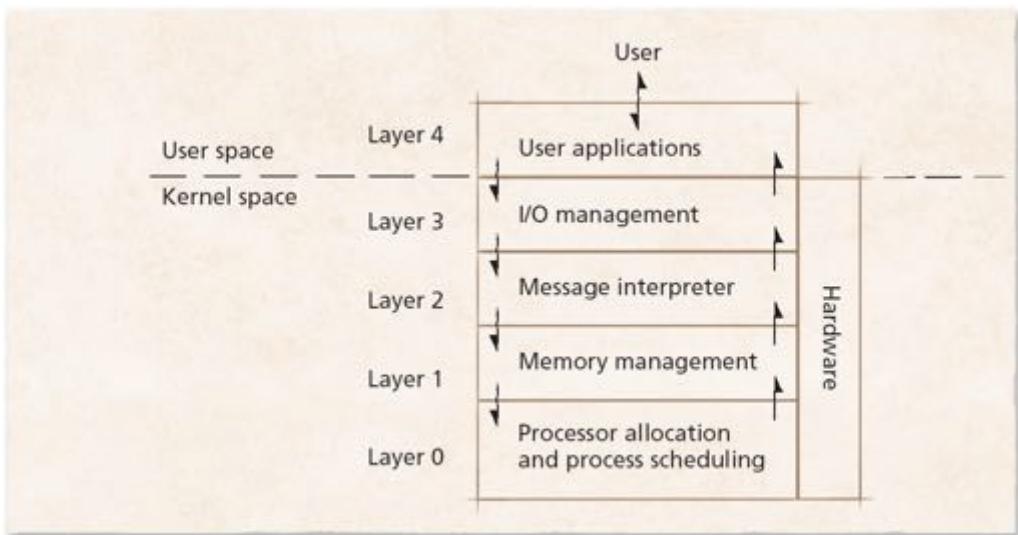
- **Architetture monolitiche:** tutte le componenti sono all'interno del kernel e possono comunicare tra di loro direttamente con semplici chiamate di sistema. La comunicazione diretta tra i componenti interni al kernel rende questo tipo di architettura particolarmente efficiente, può però risultare difficile isolare fonti di errori o malfunzionamenti in quanto tutte le componenti sono interne al kernel. Inoltre, dato che l'intero codice viene eseguito con piene possibilità di accesso a risorse del sistema, i kernel di questo tipo di architettura risultano esposti a danni dovuti a codice accidentalmente o intenzionalmente errato.



- **Architettura a Microkernel:** l'architettura a microkernel fornisce pochi servizi essenziali di base nel tentativo di mantenere il kernel piccolo e scalabile. Questi servizi di base solitamente riguardano la gestione della memoria a basso livello, la comunicazione tra processi (IPC) e i meccanismi di base per la sincronizzazione. In queste architetture la maggior parte dei componenti del sistema operativo vengono eseguiti al di fuori del kernel con bassi privilegi di accesso. Questo porta a grande modularità, portabilità e scalabilità, senza dimenticare che il microkernel per funzionare non ha bisogno di tutti i componenti, dunque questa architettura risponde molto bene anche ai guasti. Il prezzo da pagare è un gran numero di comunicazioni tra le componenti che va inevitabilmente ad incidere negativamente sulle prestazioni.



- **Architetture a strati:** le architetture a strati dividono in strati le componenti del sistema operativo che hanno funzionalità simili. Ogni strato comunica con quello immediatamente inferiore o superiore, e gli strati di livello inferiore forniscono servizi a quelli di livello superiore mediante un'interfaccia che nasconde le operazioni svolte. Questo tipo di architettura, in quanto modulare, consente la modifica di ogni singolo strato indipendentemente dagli altri, oltre ad un più facile isolamento di errori. Questo rende il sistema molto robusto, tuttavia una semplice richiesta di un processo deve attraversare molti livelli e questo provoca una efficienza nelle prestazioni minore rispetto alla soluzione monolitica. Inoltre anche in questa architettura gli strati hanno libero accesso alle risorse di sistema e al kernel favorendo dunque la propagazione di codice accidentalmente o intenzionalmente errato.



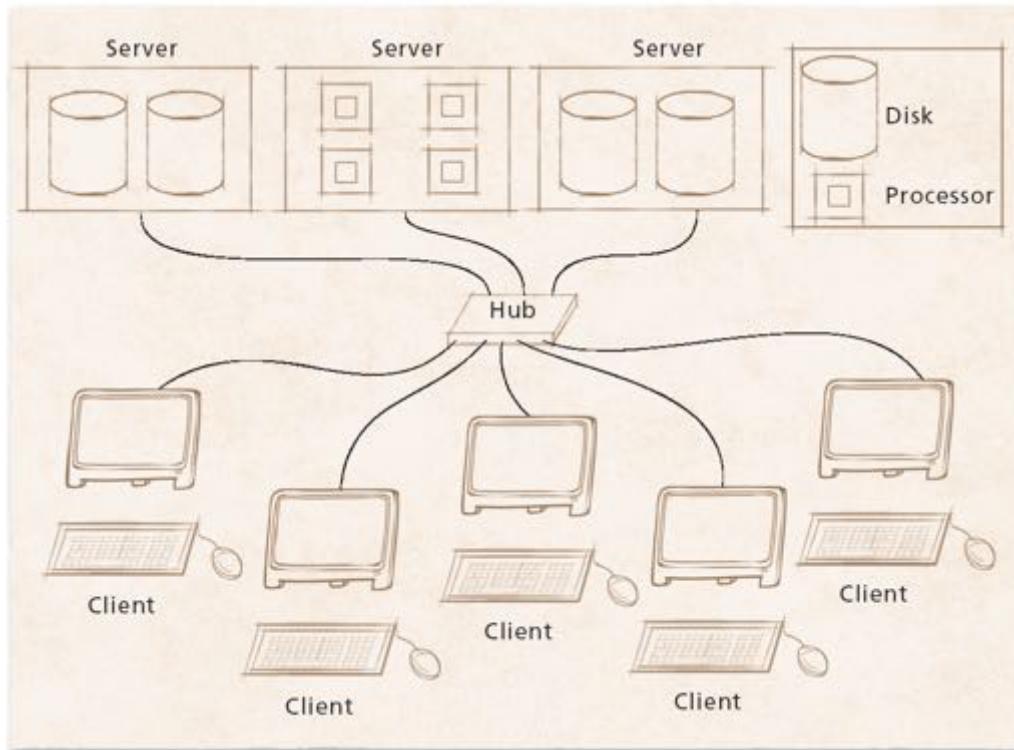
Nei sistemi operativi odierni le 3 architetture vengono fuse in un approccio ibrido che limita i difetti dei 3 approcci e ne risalta i pregi.

### Sistemi distribuiti ( o sistemi di rete)

L'ultima tipologia di sistemi operativi che vedremo sono i **sistemi distribuiti**, in cui il sistema operativo vero e proprio in esecuzione su una macchina può accedere in maniera trasparente a risorse presenti in altre macchine. La struttura di questa architettura si basa sul modello client-server: immaginiamo i client come dei semplici terminali che accedono a unità di elaborazione di

remoto (server) che mettono a disposizione una serie di processi e memorie condivise.

Un sistema operativo distribuito, quindi, è un singolo sistema operativo in grado di gestire risorse distribuite su più macchine.



## Lezione 2

Abbiamo visto come un **sistema operativo** sia un gestore di risorse, un layer di software che si interpone in maniera trasparente tra il livello software e l'hardware per gestire in maniera affidabile, sicura e concorrente le applicazioni. L'OS utilizza tutti i componenti di una macchina: il processore, la memoria, i dispositivi di storage, i dispositivi di IO, per quanto concerne il livello hardware; mentre a livello software utilizza e gestisce i processi, i thread e i file.

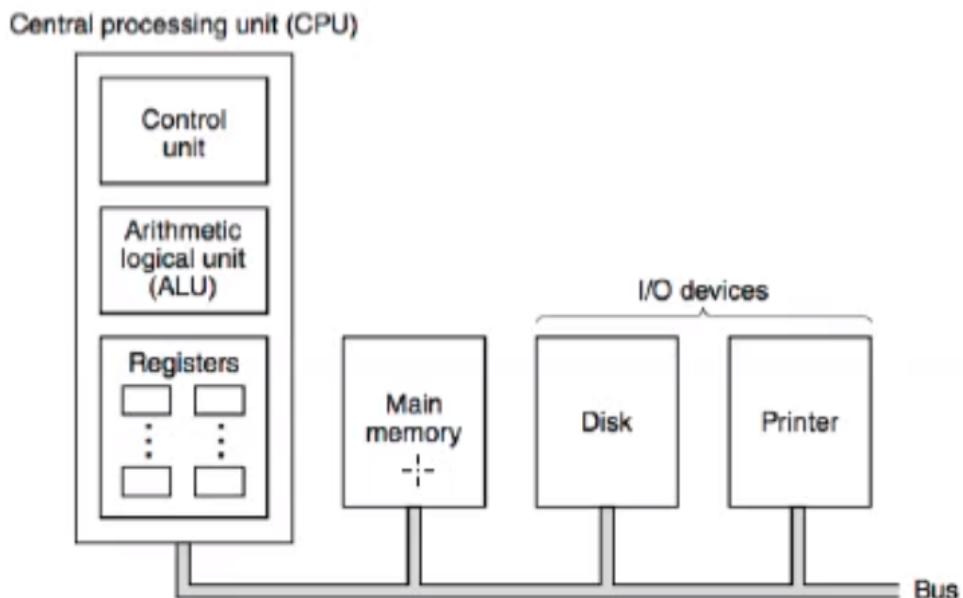
Nell'arco degli anni l'evoluzione dell'OS è andata di pari passo con l'evoluzione dell'hardware, quindi c'è stata una sorta di collegamento tra hardware e OS. Ora gli OS sono però indipendenti dall'hardware su cui girano, questo grazie a dei **device driver** che permettono di utilizzare tutti i dispositivi di I/O indipendentemente dal sistema operativo della macchina.

Il tipo di programmazione che abbiamo fatto finora è quella di applicazioni, ovvero quella che sfrutta i compilatori per leggere linguaggi di alto livello, ed eseguire, dopo l'operazione di compilazione, delle operazioni di basso livello. Esistono però altri tipi di programmazione, tra cui quella di sistema, cioè quella programmazione di basso livello che si va ad implementare direttamente sull'hardware. Questo tipo di programmazione è legata allo sviluppo del sistema operativo. Più in generale tutte le operazioni di gestione del sistema sono demandate alla programmazione di sistema.

### Architettura dei calcolatori

Vediamo quali sono le componenti principali in un calcolatore, collegate tramite bus, e si possono sintetizzare in:

- **Scheda madre**
- **Processore**
- **Memoria principale**
- **Dispositivi di I/O**



Vediamoli nel dettaglio singolarmente.

- **Scheda madre:** nella scheda madre montiamo la maggior parte dei componenti hardware del PC, che sono collegati tra di loro con l'unione di tante "tracce" elettriche, chiamati **bus**. Nella scheda madre abbiamo CPU e RAM che comunicano tramite bus, più è veloce il bus, maggiore sarà la trasmissione di dati da un componente all'altra.

Uno dei componenti presenti nella MOBO è il **BIOS** (Basic IO System): contiene delle funzionalità di base per la gestione di dispositivi IO. È un chip presente a bordo della scheda madre che viene utilizzato per l'inizializzazione e caricamento dell'OS. Questo avviene tramite l'operazione di **bootstrapping**, ovvero di caricamento dell'OS dalla memoria secondaria a quella primaria.

Negli anni il BIOS si è evoluto in un sistema un po' più innovativo dal punto di vista grafico, chiamato UEFI.

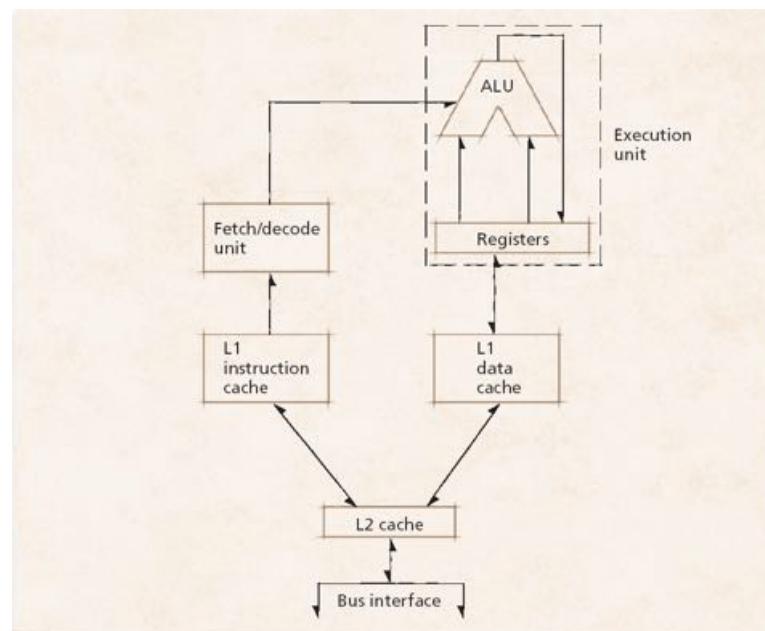
Infine nella MOBO troviamo un chipset che contiene tutti i **controllori (controller)** dei vari bus della CPU e i vari coprocessori.

- **Processore:** è un componente hardware che esegue un flusso di istruzioni in linguaggio macchina. È caratterizzato da una **ALU (Arithmetic Logic Unit)** che esegue le operazioni aritmetico-logiche, una **CU (Control Unit)**, **coprocessori** che eseguono specifiche istruzioni e compiti (tipo la grafica con la GPU), **registri** ad alta velocità che servono a velocizzare le operazioni di lettura e scrittura al processore. Troviamo poi le memorie **cache** e l'unità di **Fetch/Decode** il cui scopo è quello di eseguire il ciclo di **Fetch-Decode-Execute**, che serve nell'esecuzione delle istruzioni tramite alcuni passaggi:

1. Carica la prossima istruzione nella memoria principale.
2. Aggiorna il Program Counter, che viene incrementato di uno di volta in volta e dice al processore quale è la prossima istruzione da eseguire.
3. Determina il tipo di istruzione caricata nel registro.
4. Se l'istruzione usa una parola in memoria, ne trova la locazione (tramite MAR).
5. Prende la parola e la carica nei registri di memoria (MBR e IR).
6. Esegue l'istruzione.
7. Ritorna allo step 1.

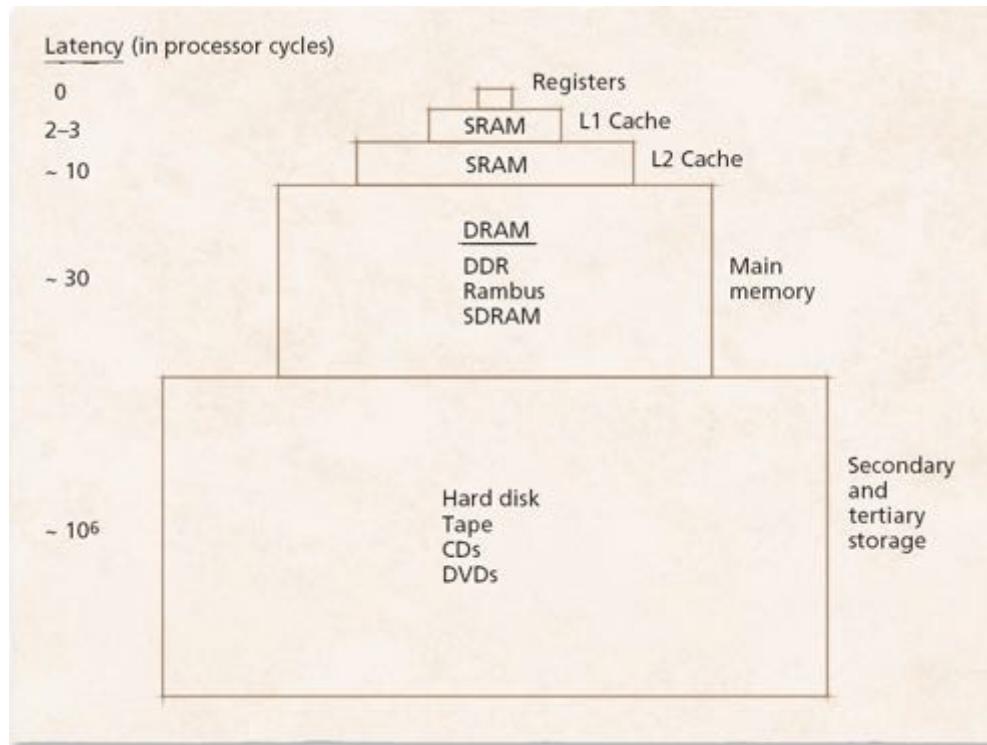
Ogni architettura ha il suo set di operazioni chiamato **instruction set**, che si distingue grazie alle dimensioni delle istruzioni stesse, che determinano anche il tipo di architettura.

Vediamo nello specifico l'architettura di un processore:



Tutte le operazioni svolte dalla CPU sono vincolate dal **clock** che altro non è che un segnale elettrico generato da un generatore di clock. Il clock misura la frequenza con quale lavora il processore, ovvero quante operazioni fa il processore in un ciclo completo di clock.

- **Memorie:** Affinché le prestazioni offerte dalla CPU siano garantite è necessario l'utilizzo di **registri** del processore che sono memorie che come velocità si attestano, bene o male, alla frequenza del clock e si trovano all'interno della CPU stessa. In particolar modo la gerarchia delle memorie, in ordine decrescente di velocità, è la seguente: **Registri; Cache L1; Cache L2; Memoria principale (RAM); Storage (HDD o SSD).**



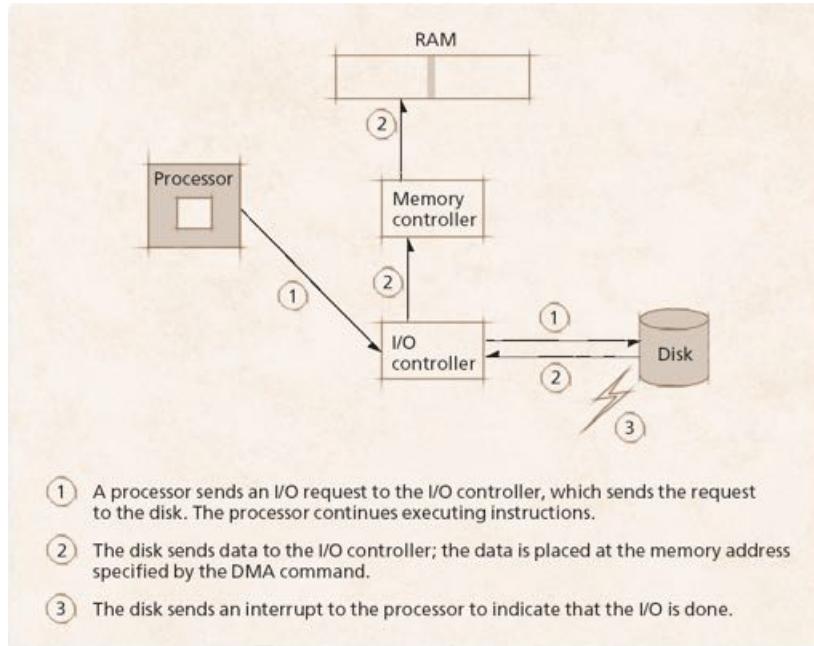
Ognuna di queste memoria ha uno scopo diverso, in base all'utilizzo varia la frequenza e la sua capacità. Tutto il problema di fondo della gestione dell'OS è che i dati devono essere sempre caricati nella memoria principale prima di accedere al processore, dato che la memoria principale ha prestazioni decisamente minori della CPU si utilizzano i registri e le cache per fare da tampone ed evitare colli di bottiglia.

- **Bus:** sono collezioni di tracce elettriche (connessioni elettriche) alla cui estremità si trovano delle porte collegate alle periferiche di I/O. Si dividono in 3 tipologie principali: **control**, **address** e **data**: la differenza sta in ciò che trasportano, i primi istruzioni di controllo, i secondi gli indirizzi destinatario e mittente (in memoria ovviamente) dei dati e i terzi che trasferiscono dati.

Uno dei bus che abbiamo visto, nonché più utilizzato, è quello SATA, evoluzione dell'ATA.

- **DMA (Direct Memory Access):** tipicamente i dati vanno trasferiti dalla memoria al processore, con tempistiche non sempre adatte alla frequenza del processore: il processore eseguirà poi le istruzioni che ha nei registri ed eventualmente invierà dati dalla memoria centrale alle periferiche di I/O, e viceversa. Questo avviene tramite **interrupt**: quando un'operazione di I/O termina, la riuscita viene segnalata mediante un interrupt al processore.

Il **DMA** permette di fare avvenire direttamente il passaggio di dati da memoria a i dispositivi di I/O senza passare dal processore così da migliorare notevolmente le prestazioni.



Utilizzando il DMA, quindi, il processore passa il controllo diretto dei dati ad un controller esterno chiamato **controller I/O**, ciò significa che quando faccio richiesta di un dato questo dato passa dal disco alla memoria principale senza passare dal processore. Questo può causare alcuni problemi di sicurezza, infatti con i recenti bus si possono collegare al calcolatore delle periferiche con unità di elaborazione autonome. Ciò che può avvenire è che questa unità di elaborazione, tramite controller I/O, scambi dati col sistema, senza però darne notifica al processore, quindi qualsiasi informazione nel disco può essere trasferita in RAM senza che sia verificato dall'OS prima. Un'utente malevolo potrebbe trasferire contenuti nella RAM senza alcun controllo quindi, l'OS non se ne accorgerebbe, così come non se ne accorgerebbe neanche il processore. Per questo motivo sono state progettate unità di gestione della memoria più complesse che effettuano verifica dei dati in maniera da creare uno strato intermedio tra il dispositivo esterno e il controller di I/O e verificare i dati. Per fare questo si utilizzano un set di indirizzi virtuali. Se non utilizzassimo il DMA, lo schema di sopra sarebbe materialmente senza un controller I/O.

- **Periferiche:** sono quell'insieme di dispositivi collegati al calcolatore che di per sé non sono necessari per eseguire operazioni. Infatti possiamo definirli come dispositivi utili ma non necessari.

Alcune periferiche sono collegate alla macchina tramite USB (Universal Serial Bus), che adesso si trova alla versione 3.0, oppure tramite porta Thunderbolt.

# Lezione 3

Riprendiamo il discorso di ieri sulle funzionalità del calcolatore.

## User mode e Kernel mode

L'organizzazione del calcolatore prevede che sia il processore a svolgere le istruzioni, e tra le varie cose che deve fare c'è anche la gestione dei processi.

Non abbiamo ancora dato una definizione formale di processi, ma immaginiamoli almeno per il momento, come delle istanze in esecuzione di un certo programma. Ciò che deve fare il processore non è soltanto l'esecuzione del processo ma implementare anche alcuni meccanismi di protezione, necessari se si pensa al fatto che i processi non devono svolgere operazioni che vadano ad intaccare il sistema in generale. Per implementare questa protezione, i sistemi vengono eseguiti secondo due diverse modalità.

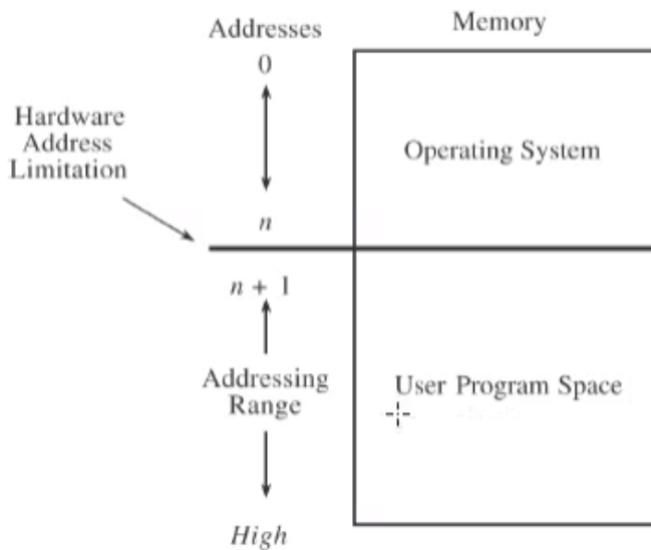
Una delle due modalità è la **User Mode**, che fa in modo che tutti i processi a livello utente possano usare solo una piccola parte delle istruzioni sicure e accedere solo ad alcune zone di memoria.

L'altra modalità è la **Kernel Mode**, in questa modalità il processore non ha restrizioni ed esegue istruzioni privilegiate e accede ad aree di memoria riservate.

In generale il principio che si utilizza nella gestione dei processi è quello dei **minori privilegi possibili**, cioè garantiamo all'utente l'accesso solo a quelle risorse indispensabili per svolgere il suo task.

Inoltre i processi non possono accedere a zone di memoria che non gli competono, ma solo a quelle che gli sono state assegnate. Questa operazione si fa utilizzando degli ulteriori registri chiamati **bound register** che delimitano la porzione di memoria a cui un processo può accedere e, quando un processore deve allocare un processo in memoria, a verificare le zone di memoria non ancora utilizzate. Le politiche di allocazione dei processi non sono banali come si può pensare.

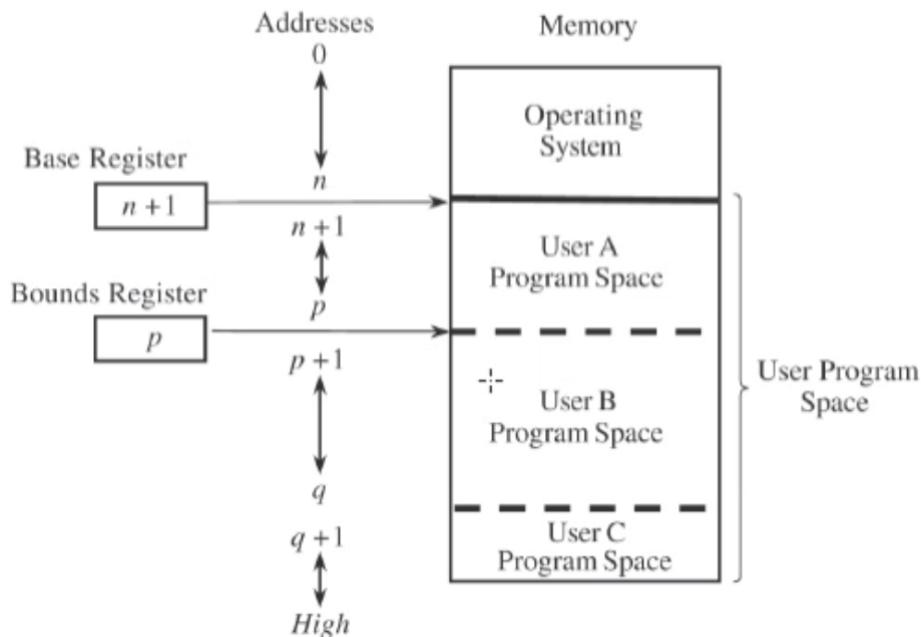
Il bound register, in particolare, specifica l'indirizzo di inizio e di fine di una sezione di memoria dedicata ad un processo come in figura:



Si delimita tramite un indirizzo di confine (**Hardware Address Limitation** nella figura di sopra) l'insieme di indirizzi referenziati dai processi di sistema e l'insieme di indirizzi referenziati dai processi di applicazione (utente). Se un processo utente tenta di accedere ad un indirizzo tra 0 ed  $n$  il processore si accorgerebbe dell'errore e ne vieterebbe l'accesso.

Anche in questi registri, come avviene nel resto delle memorie, si definisce una gerarchia, in particolare si utilizza un registro a parte che contiene l'indirizzo di confine tra lo spazio per i processi di sistema e quello per i processi di applicazione.

Il limite di questo approccio è che questo registro delimita solo due entità (spazio per l'OS e spazio utente), ma in realtà i sistemi sono condivisi tra più utenti, quindi abbiamo la necessità di più registri per delimitare lo spazio tra i vari utenti. Si usano quindi delle coppie di registri per delimitare gli spazi tra i vari utenti come in figura:



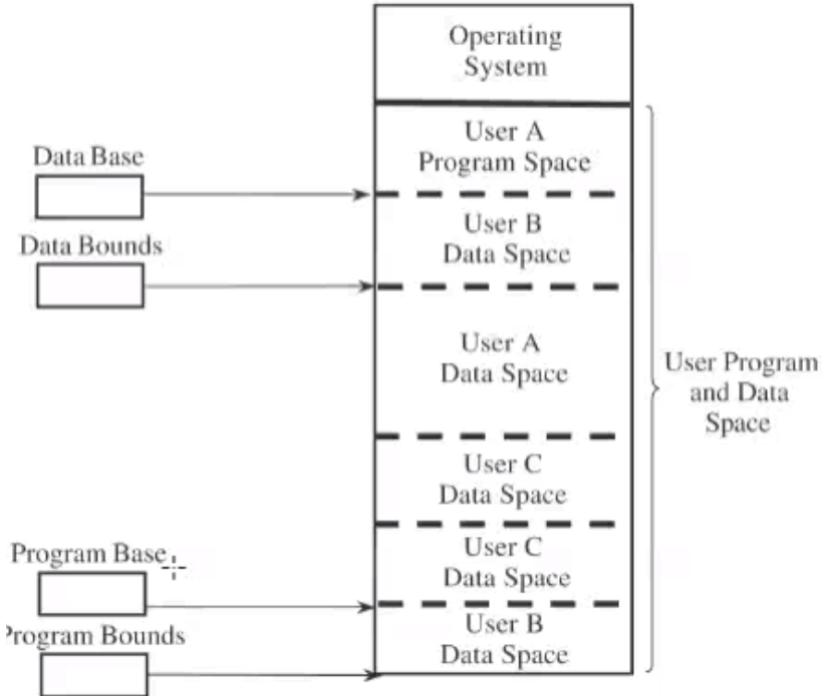
Queste operazioni di divisione della memoria sono gestite in maniera trasparente, solo l'OS sa quale spazio è dedicato ad un utente.

### Context Switch

Immaginiamo di essere in un ufficio postale e di essere in fila per ottenere un servizio. Quando arriva il nostro turno veniamo serviti fino a quando l'operatore non completa tutte le operazioni che abbiamo richiesto, solo allora usciremo dalle poste. Questo è, più o meno, lo stesso concetto dell'esecuzione dei processi nell'OS.

L'operatore rappresenta il processore, e noi che richiediamo un servizio siamo i processi. A differenza di come avviene alle poste però, affinché il sistema sia efficiente non si può eseguire un processo solo dopo aver aspettato il completamento di quello prima, questo perché si creerebbero dei tempi di attesa molto lunghi. Per ovviare a ciò un processo accede alle risorse del processore quindi solo per una piccola quantità di tempo. Il processore deve quindi gestire il processo corrente fino a quando può, scaduto questo arco di tempo il processore può servire un altro processo, senza però perdere le informazioni del processo precedente che è ora in attesa. Questo passaggio di gestione di processi da parte del processore si chiama **context switch**.

Nell'ottica della protezione di memoria, il processore deve anche memorizzare i registri bound relativi a ogni processo che sta servendo e in generale deve occuparsi di memorizzare gli stati dei processi per non perdere i progressi. Un ulteriore livello di separazione è dato dalla delimitazione dello spazio riservato ai dati con lo spazio destinato alle istruzioni, vediamolo in figura:



### Interruzioni ed eccezioni

Abbiamo parlato nella precedente lezione di interrupt al processore facendo riferimento alle operazioni di I/O e al DMA. Gli **interrupt** e le **eccezioni** sono degli utili strumenti per evitare richieste periodiche, e troppo frequenti, da parte del processore sullo stato dei dispositivi del calcolatore, che prendono il nome di **pooling**. Infatti, controllare frequentemente lo stato dei dispositivi può essere svantaggioso perché terrebbe il processore impegnato in operazioni e richieste non necessarie. Con l'interrupt invece si fa in modo che il processore chieda lo stato del dispositivo e nel mentre possa lavorare ad altro, successivamente arriverà una risposta da parte del dispositivo sul suo stato. Questo meccanismo è possibile mediante all'utilizzo di opportuni **controller di I/O**.

È possibile prevedere gli interrupt mediante **timer**, per esempio, come avviene nelle tecniche di scheduling (tecniche per la gestione dei processi) che fanno uso di timer che per scandire una periodicità nella esecuzione dei processi. Ciò è importante per evitare il monopolio del processore da parte di un singolo processore.

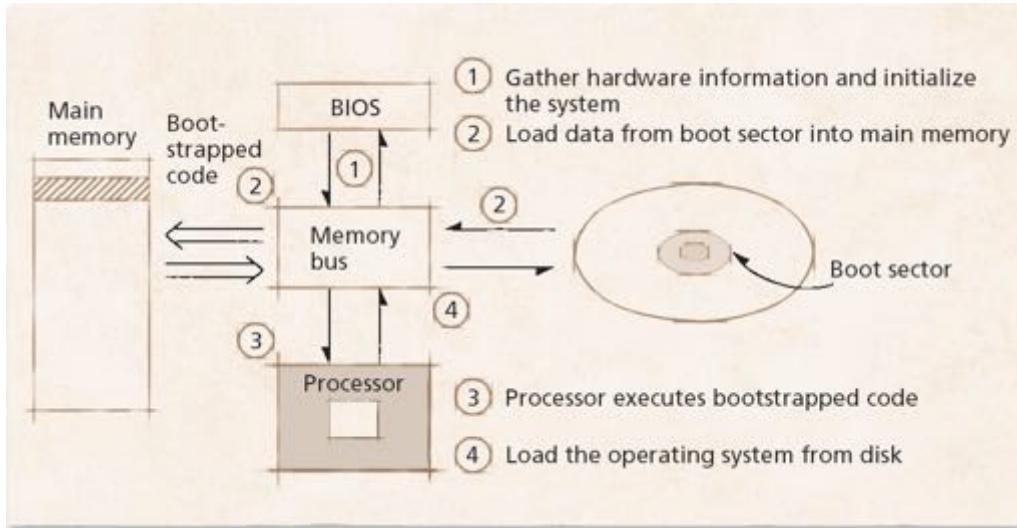
Abbiamo poi i **clock**, che si riferiscono all'ora attuale, e si utilizzano per tenere conto di ora e data attuale nelle operazioni svolte.

### Bootstrapping

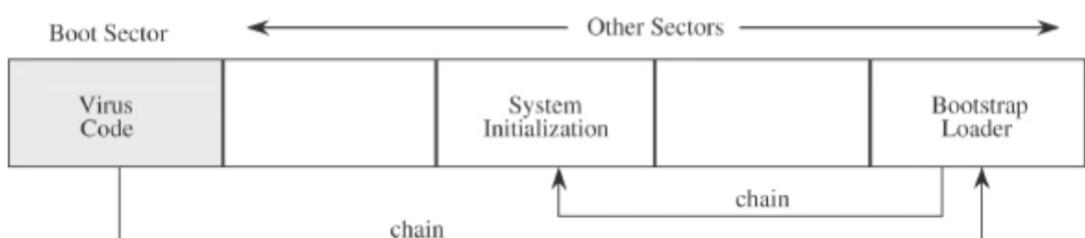
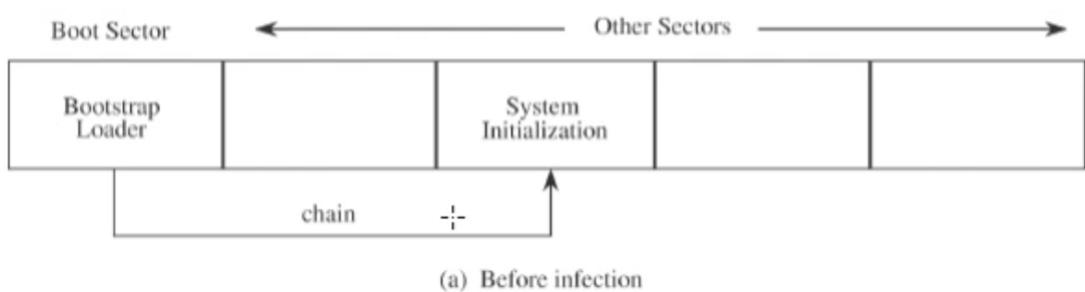
Il procedimento di **bootstrapping**, o **boot**, è l'operazione che consente di caricare alcune componenti di base dell'OS in memoria. Tutto ciò di cui parliamo in questo corso, infatti, presuppone che ci sia un sistema operativo, che essendo un software, va caricato in memoria.

Nel momento in cui accendiamo il PC infatti la prima operazione svolta dal BIOS è l'inizializzazione dell'hardware del sistema e il caricamento in memoria principale di alcune informazioni essenziali (500 byte circa), queste informazioni vengono prese da una regione del disco rigido chiamato **boot sector**. Se il BIOS non trova il boot sector, si può modificare l'ordine di boot, ovvero specificare al BIOS in quale ordine analizzare le periferiche per trovare questa regione di memoria dove si trovano le informazioni necessarie al bootstrapping.

Riassumendo possiamo dire che il BIOS ha quindi lo scopo principale di copiare il contenuto del boot sector in memoria principale, durante una fase che prende il nome di bootstrapping. Dopo averlo fatto si esegue un jump di memoria, cioè termina il bootstrapping e si "salta alle" (*leggasi: si eseguono*) istruzioni che consentono di caricare le componenti rimanenti di sistema. Questa operazione prende il nome di **bootstrap loader** o **booting**. Vediamo quanto descritto in figura:



Un problema della operazione di bootstrapping è legata ai virus. In particolare, ai tempi in cui venivano utilizzati i floppy disk, alcuni floppy potevano essere portatori di un virus che aveva effetto sul boot sector. Nel momento in cui effettuiamo l'operazione di booting non abbiamo ancora caricato completamente l'OS, ma come abbiamo visto prima le operazioni di controllo e protezione della memoria sono svolte proprio dall'OS, quindi tutta l'operazione di copia dal boot sector alla memoria principale avviene senza controllo da parte del sistema. Completato il caricamento del boot sector il processore carica in memoria tutto il resto del sistema mediante un'operazione che prende il nome di **chaining**: si caricano blocchi di memoria in successione, collegati tramite link. Inserendo un virus in questa operazione di chaining, tra i blocchi di sistema da caricare, il codice del virus verrà caricato a sua volta assieme al sistema in memoria principale. Graficamente:



## Plug and Play

Le tecnologie **plug and play** consentono di caricare all'interno del sistema, in modo automatico e senza bisogno dell'utente, una serie di driver che consentono l'utilizzo e l'accesso a periferiche.

Per realizzare il plug and play, si utilizza un meccanismo di identificazione del dispositivo mediante un ID univoco che viene comunicato all'OS. In base a questo ID il sistema riconosce il device connesso ed esegue l'installazione dei driver.

La periferica deve anche specificare quali operazioni supporta e a che servizi deve accedere. Di contro, per collegare dispositivi plug and play, vengono svolti anche altri task quali la previsione di una creazione di un canale DMA utile alla gestione del dispositivo stesso che comportano un utilizzo di risorse e del processore aggiuntivo.

### Caching, Buffering e Spooling

Abbiamo visto ieri come sia necessario gestire la differenza di velocità tra processore e memoria principale. A tal fine possono essere utili altre memorie: le **cache**. Le cache sono di due tipi: **L1** e **L2**, che si dividono in base a velocità e capienza.

Il vantaggio nell'utilizzo della cache sta nel sapere prevedere cosa bisogna salvare nella cache stessa; conviene salvare nella cache dati ricorrenti utilizzati dal processore. I due elementi per valutare l'efficienza della cache sono la **cache hit** e la **cache miss** che rappresentano rispettivamente dei valori euristici che ci indicano quale è il rapporto tra il successo e il fallimento (ovvero: trovare o non trovare un dato specifico nella cache).

Un concetto simile alla cache è quello dei **buffer**, infatti, visto che c'è differenza di velocità tra i differenti dispositivi del calcolatore è necessario prevedere delle zone di memoria dove salvare le informazioni nell'attesa che il dispositivo di lettura/scrittura sia pronto a leggere questi dati. Man mano che i dati vengono letti dal buffer questo buffer viene svuotato. Il processore utilizza il buffer anche per scrivere dati così da liberare spazio nei registri.

Infine troviamo lo **spooling**: una tecnica che fa uso di un dispositivo intermedio che permette di fare comunicare processi e dispositivi di I/O che hanno una bassa velocità (ad esempio le stampanti). Questo procedimento permette di inviare in modo asincrono dati alla periferiche che li leggerà e li processerà con le sue tempistiche. Questo consente al processore di demandare il compito al dispositivo e nel mentre svolgere altri task.

# Lezione 4

## Linguaggi di programmazione

Oggi andremo avanti vedendo un paio di esempi di **linguaggi di programmazione**, tra cui **assembly**.

Cominciamo dicendo che i linguaggi si dividono in **linguaggi di basso livello** e di **alto livello**.

Quelli di alto livello sono comprensibili da chi li legge, mentre per essere compresi dal processore hanno bisogno di essere tradotti in linguaggio macchina da altri programmi chiamati compilatori/interpreti. Quelli di basso livello sono comunque comprensibili dall'uomo, ma hanno una sintassi decisamente più striminzita e si trovano dal punto di vista logico più vicini alla macchina che all'uomo, assembly rientra tra questi linguaggi.

Esiste poi il **linguaggio macchina**, ovvero quello che il processore legge e comprende, che serve a dare direttive elementari (operazioni elementari) a quest'ultimo. Noi non sappiamo leggere il linguaggio macchina ma può farlo solo il processore, o meglio l'architettura, per cui è pensato, infatti ad ogni architettura è associato un particolare linguaggio macchina. Le istruzioni in linguaggio macchina sono formate da stringhe di numeri (0 o 1 principalmente) e per questo motivo negli anni si sono sviluppati i linguaggi di alto livello.

### Assembly

Nasce così assembly, un linguaggio di basso livello che permette di tradurre istruzioni di linguaggio macchina in delle istruzioni English-like comprensibili dall'uomo. L'operazione di traduzione viene effettuata da un **programma assemblatore**. Vediamo un eSEMPIO di come si possa tradurre linguaggio macchina in assembly:

1300042774 1400593419 1200274027		LOAD      BASEPAY ADD        OVERPAY STORE     GROSSPAY
--	---	---

Assembly permette di rendere la programmazione più veloce per l'uomo per via della sua comprensione più immediata, per lo stesso motivo ha consentito di ridurre i bug in generale nei programmi.

In Assembly abbiamo linee di codice che sono costituite da 4 campi:

1. **Etichetta:** stringa che serve a definire il nome simbolico di un certo indirizzo.
2. **OpCode:** codice mnemonico che ci permette di svolgere un'operazione come se fosse un operatore. Detto in altri termini l'opcode determina l'esecuzione di una specifica istruzione (LOAD, ADD, STORE ecc.).
3. **Operandi:** oggetti dell'azione specificata dall'OpCode che variano a seconda dell'istruzione e alla modalità di indirizzamento.
4. **Commenti:** testo aggiunto a piacimento del programmatore che ha la possibilità di aggiungere commenti come in tutti gli altri linguaggi.

In Assembly esistono, come accennato prima, varie modalità di indirizzamento, la più semplice è l'**indirizzamento immediato**: in questo tipo di indirizzamento specifichiamo l'OpCode seguito da un operando proprio (cioè un valore costante), se il valore rappresentato dall'operando è una costante si parla allora di indirizzamento immediato. Vediamo un esempio di indirizzamento immediato:

Questo indirizzamento non richiede alcun puntatore alla memoria ma di contro limita un po' il numero di operazioni che possiamo fare limitandoci all'utilizzo delle costanti.

Per fare operazioni fare più complesse utilizziamo l'**indirizzamento diretto**. Nell'indirizzamento diretto per caricare un operando in memoria specifichiamo il suo indirizzo in memoria. Vediamo un esempio:

LDA R1, 67

//R1=M[67]

In questo caso il processore assegna al registro R1 il valore che è salvato nella posizione 67 della memoria, indipendentemente dal suo contenuto, che può essere eventualmente nullo.

Questo tipo di indirizzamento specifica un indirizzo costante ciò significa, in riferimento all'esempio di prima, che caricherò sempre ciò che si trova in memoria all'indirizzo 67, non sappiamo però se il contenuto rimarrà costante. Il valore della cella 67 potrà dunque cambiare, ma la posizione a cui punta R1 no. Se voglio sfruttare questo indirizzamento devo di volta in volta cambiare i contenuti degli indirizzi. Inoltre, questo metodo, viene utilizzato per memorizzare e fare acceso a variabili globali (gestite a tempo di compilazione).

Esiste un terzo tipo di indirizzamento detto **indirizzamento indiretto** che utilizza due registri superando i limiti che dicevamo pocanzi sull'indirizzamento diretto. Vediamo un esempio:

LDA R1, R2

//R1=M[R2]

In questo caso utilizziamo due registri, in particolare si scrive in R1 ciò che viene puntato dal registro R2, ma quest'ultimo parametro lo possiamo variare, cosa che nell'accesso diretto non si poteva fare e che risulta essere una comodità. Un indirizzo di memoria utilizzato in questa maniera viene chiamato **puntatore**.

Vediamo ora una serie di istruzioni permesse in assembly; notiamo come in base al tipo di OpCode cambino le operazioni fatte sui registri:

ADD R1, R2, R3	//R1 <- R2 + R3
ADDI R1, R2, addr	//R1 <- R2 + addr (I:immediate)
AND R1, R1, R2	//R1 <- R1 AND R2 (bitwise)
LOAD R1, addr	//R1 <- RAM[addr]
STORE R1, addr	//RAM[addr] <- R1

Un'altra operazione estremamente rilevante in assembly è il **jump (JMP)**, chiamato **salto** in italiano, che ci permette di implementare operazioni cicliche di alto livello quali for e while. Abbiamo due tipi di jump: **incondizionati** e **con condizione**. Spieghiamoli partendo da un frammento di codice di esempio:

```
mov eax, 1
inc_again:
    inc eax
    jmp inc_again
    mov ebx, eax
```

L'istruzione **JMP** precedente esegue un salto incondizionato a ciò che c'è all'interno dell'operando, ovvero ad *inc\_again*, cioè un indirizzo. In questo specifico esempio, questo indirizzo a cui ci riferiamo è l'indirizzo della label contrassegnata da *inc\_again*. Dal punto di vista logico stiamo effettuando un loop infinito visto che la terza operazione del ciclo non sarà mai raggiunta dato che non vi è alcuna condizione per uscire dall'loop. Quindi il jump incondizionato è

un jump che viene eseguito sempre, a prescindere dalla condizione. In questo riconosciamo che il jump è incondizionato perché non vi è alcuna condizione affinché esso non avvenga.

In altri termini possiamo dire che effettuare un jump corrisponde a sostituire nel PC l'indirizzo dell'operando affiancato al comando di jump.

Vediamo ora come si può effettuare un **jump con condizione** partendo dal seguente pezzo di codifica di esempio:

```
while(x <= 10){  
    x++;  
}
```

In questo esempio abbiamo un semplice while che continua a verificarsi fintantoché  $x$  è minore o uguale a 10: ad ogni iterazione andiamo ad aumentare il valore di  $x$ . Come possiamo tradurlo assembly? Al momento non possiamo, dobbiamo infatti riscriverlo scomponendolo in operazioni ancora più elementari come ad esempio l'*if* e il *do while*. Un modo diverso per scrivere il *while* di sopra è il seguente:

```
if(x <= 10){  
    do  
    {  
        x++;  
    }while(x <= 10)  
}
```

La prima condizione (*if*) di fatto implica un confronto, se questo confronto dovesse dare esito positivo effettuo l'incremento, dopodiché effettuo nuovamente il confronto e così via. Questa codifica è più complicata ma ci aiuta a trovare la logica per tradurlo in assembly. Se  $x$  è minore o uguale a 10 entriamo nell'*if* ed effettuiamo un ciclo finché  $x$  è minore o uguale a 10. Siamo pronti ad effettuare il passaggio a codice assembly:

```

mov eax, $x /*assegna al primo operando (registro del processore eax) un valore
costante x, che presupponiamo essere già stato inizializzato in precedenza.*/
cmp eax, 0x0A /*confronta il registro eax con il valore esadecimale 0A ovvero
10 */
jg end /*viene fatto jump se il valore di eax è maggiore di 10 (jg sta per
jump greater). Queste 2 ultime operazioni lavorano assieme. Prima faccio il
confronto di eax con 10, che ci può dire se i due valori sono uguali, oppure che
uno è minore o maggiore dell'altro. Scrivendo il jump dopo sto implicitamente
dando una condizione al jump: se il confronto è vero (eax>10) ritorna
all'indirizzo indicato dal jump, ovvero alla etichetta end. Con questo jump
abbiamo implementato l'if del codice di prima.*/
beginning: /*etichetta utilizzata per indicare blocco di codice e in
particolare per dare un riferimento al secondo jump. Con questa etichetta
associata al jump abbiamo implementato il do while del codice in c di sopra.
Questa etichetta beginning indica infatti l'inizio del blocco do while*/
inc eax /*incremento x di 1 */
cmp eax, 0x0A /*confronto nuovamente eax con 10 espresso in esadecimale*/
jle beginning /*questa operazione dipende dall'esito del confronto precedente
e in particolare l'esito sarà positivo se il valore di eax sarà minore (less L)
o uguale (equal E) a 10. Se è vera la condizione effettuo nuovamente un jump a
beginning e procedo così via fino a quando il confronto di sopra non sarà falso
e quindi si passerà alla prossima istruzione*/
end: /*etichetta che rappresenta la fine del blocco if. Se si arriva
qua il valore di eax sarà necessariamente maggiore di 10.*/

```

Vediamo ora il **set di istruzioni IJVM** (tipo di assembly):

Hex	Mnemonic	Meaning
0x10	BIPUSH byte	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO offset	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ offset	Pop word from stack and branch if it is zero
0x9B	IFLT offset	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ offset	Pop two words from stack; branch if equal
0x84	IINC varnum const	Add a constant to a local variable
0x15	ILOAD varnum	Push local variable onto stack
0xB6	INVOKEVIRTUAL disp	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE varnum	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W index	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Questo tipo di operazioni prevedono che la memoria sia organizzata come uno **stack**, ovvero che l'ultimo elemento scritto in memoria sarà il primo ad essere letto e rimosso. Il funzionamento dello stack si implementa tramite le due operazioni chiamate **PUSH** e **POP**, rispettivamente inserimento e rimozione.

Proviamo ora a tradurre il seguente esempio di codice in assembly IJVM:

```

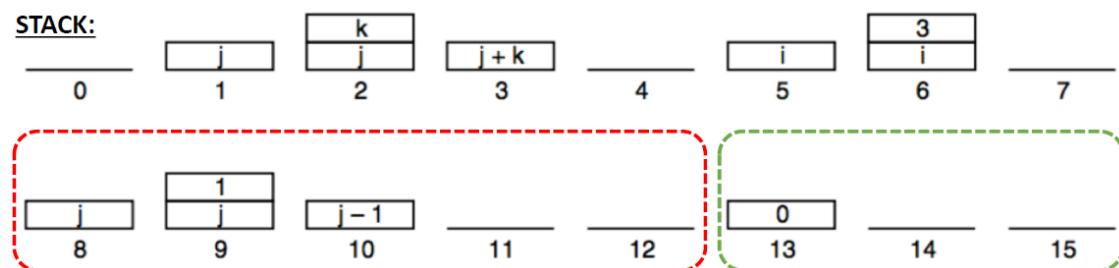
i = j + k;
if(i == 3)
    k = 0;
else
    j = j - 1;

```

Di seguito troviamo come si sviluppa la codifica in IJVM e l'andamento dello stack del pezzo di codice appena visto. Dopo l'immagine daremo una breve descrizione di quello che succede passo per passo:

i = j + k;	1	ILOAD j	// i = j + k
if (i == 3)	2	ILOAD k	
k = 0;	3	IADD	
else	4	ISTORE i	
j = j - 1;	5	ILOAD i	// if (i == 3)
	6	BIPUSH 3	
	7	IF_ICMPEQ L1	
	8	ILOAD j	// j = j - 1
	9	BIPUSH 1	
	10	ISUB	
	11	ISTORE j	
	12	GOTO L2	
	13 L1:	BIPUSH 0	// k = 0
	14	ISTORE k	
	15 L2:		

JUMP



Lo stack, all'inizio del codice è vuoto. Per assegnare i valori di *j* e *k* dobbiamo caricarli nello stack con le operazioni *ILOAD i* e *ILOAD k*. Per sommarli effettuiamo una operazione di *IADD* che somma automaticamente gli ultimi due valori presenti nello stack lasciando nello stack un unico valore che è la somma delle due variabili. Dobbiamo adesso assegnare ad *i* il risultato della somma con l'istruzione *ISTORE i*. Lo stack è tornato ora vuoto a causa dell'operazione POP effettuato da *ISTORE*. Adesso abbiamo il confronto (*i==3*), per farlo dobbiamo caricare *i* nello stack col codice *ILOAD i*. Per fare il confronto tra *i* e 3 dobbiamo caricare anche 3 nello stack tramite il comando *BIPUSH 3* che inserisce nello stack il valore costante 3. Adesso nello stack abbiamo 3 e subito sotto *i*. Facciamo il confronto con l'istruzione *IF\_ICMPEQ L1* che fa il pop dei due valori da confrontare e in base all'esito del confronto fa un jump alla fine del blocco di istruzioni dell'*if* oppure entra dentro il blocco di istruzioni subito dopo. L'operando dell'istruzione è *L1*, che è un etichetta che serve a identificare un blocco di codice, in particolare se il confronto risulta vero con l'*if* si effettua il jump ad *L1*. Ad *L1* abbiamo un *BIPUSH 0* necessario per potere inizializzare *k* a 0 e tramite *ISTORE k* assegniamo *k* a 0 (*k=0*) svuotando lo stack. Ma se il confronto dell'*if* non è vero dobbiamo eseguire l'*else*, cioè caricare nello stack il valore di *j* con *ILOAD j*, caricare 1 nello stack con *BIPUSH 1*, effettuare la sottrazione con *ISUB* e salvare il risultato in *j* con *ISTORE j*. A questo punto effettuiamo un *GOTO* alla etichetta *L2*.

L'etichetta *L2* si trova dopo avere eseguito l'assegnazione di *k*.

## Compilatori e interpreti

Dopo aver visto qualche esempio di codice assembly parliamo ora di **interpreti** e **compilatori**. Come dicevamo prima la programmazione assembly è sicuramente più veloce e intuitiva (per l'uomo) di quella fatta in linguaggio macchina, ma non è comunque esente da problemi, infatti abbiamo visto nei due precedenti esempi che dei semplici cicli o confronti risultano lunghi e ostici da scrivere.

Per sopperire a questo problema si sono sviluppati nei decenni i linguaggi di alto livello, molto simili come sintassi a linguaggi parlati come l'inglese. Lo "svantaggio", se così vogliamo definirlo, di questi linguaggi è che in ogni caso il calcolatore dovrà tradurre il codice di alto livello in assembly e poi ancora in linguaggio macchina rallentando i tempi di esecuzione rispetto ai programmi nativi in linguaggio macchina. Tutto ciò in favore di una leggibilità e comprensibilità sintattica decisamente maggiore.

Gli elementi che fanno queste operazioni di traduzione di linguaggi di alto livello sono gli **interpreti** e i **compilatori**:

- **Compilatori:** si occupano di tradurre tutto il programma, nel suo insieme, in un unico codice oggetto e poi eseguirlo in blocco.
- **Interprete:** non fa una traduzione in blocco ma va traducendo le singole istruzioni via via che va avanti. Nel caso di alcuni linguaggi interpretati, il codice di alto livello si traduce in **BYTECODE** che rappresenta il codice macchina, non del nostro calcolatore, bensì della virtual machine sul quale gira il linguaggio di programmazione interpretato (ad esempio java con la JVM).

Ovviamente si preferisce un linguaggio compilato nel caso in cui si voglia una elevata velocità di esecuzione. I linguaggi compilati non hanno infatti bisogno di strumenti intermedi per la traduzione come ad esempio avviene per i linguaggi interpretati con le virtual machine. Nei linguaggi interpretati, oltre alla virtual machine ci devono essere in esecuzione anche tutta una serie di programmi a supporto della vm, ciò rallenta i tempi di esecuzione. Il vantaggio di questi linguaggi sta però nella loro portabilità dato che sono indipendenti dal sistema su cui girano, e dispongono inoltre di una migliore gestione degli errori dato che il linguaggio viene tradotto riga per riga e non in blocco a differenza di quelli compilati.

# Lezione 5

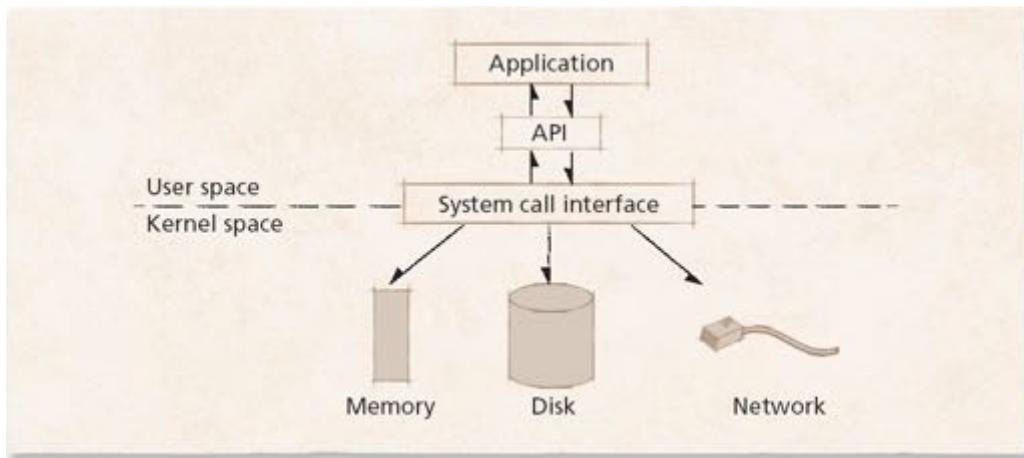
## API

Il livello utente (o applicativo) è separato dal livello del sistema operativo per evitare che gli utenti possano accedere a zone di memoria che non gli competono. Per questo motivo si utilizzano delle **API (Application Programming Interface)** che forniscono agli sviluppatori un insieme di chiamate (funzioni) di alto livello il cui scopo è gestire funzionalità di basso livello, normalmente gestite dall'OS. Le API, ci danno quindi l'opportunità di sfruttare determinate caratteristiche dell'hardware utilizzando semplici funzioni di alto livello.

Ciò che avviene in particolare è che i processi eseguono chiamate di funzione definite dalle API, le quali possono a loro volta invocare le **chiamate di sistema (system call)** per richiedere servizi al sistema operativo.

L'obiettivo è quello di fare richieste che poi si traducono in chiamate di sistema, eseguite dall'OS, per gestire memoria e hardware in generale.

Vediamo meglio cosa accade in figura:



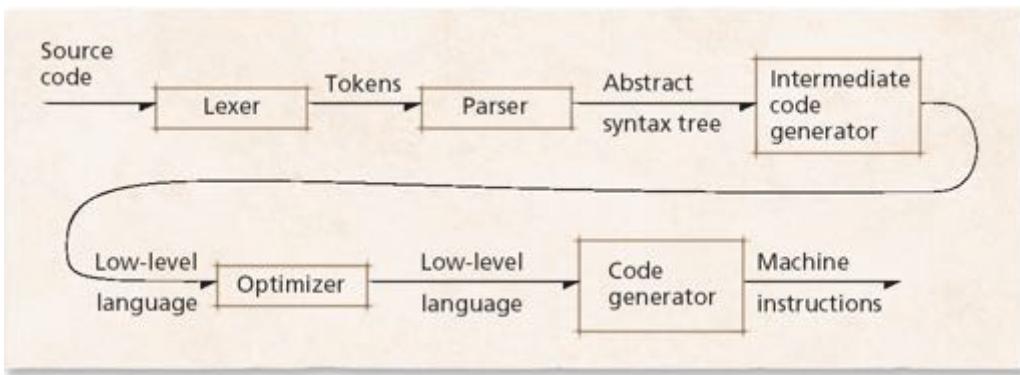
## Compilazione (compiling), collegamento (linking) e caricamento (loading)

Affinché un programma scritto in linguaggio di alto livello sia eseguibile bisogna che quest'ultimo venga tradotto in linguaggio macchina, successivamente che venga collegato ad altri programmi da cui dipende per funzionare ed infine va caricato in memoria. Ognuna di queste fasi va studiata nel dettaglio.

### Compilazione (compiling)

Affinché un certo programma scritto in un linguaggio di alto livello possa essere eseguito dal calcolatore è necessario tradurlo nelle corrispondenti istruzioni in linguaggio macchina, questa operazione di traduzione prende il nome di **compilazione**. Tipicamente la traduzione non riguarda un solo file sorgente, ma il collegamento di più file ognuno dei quali serve a qualcosa.

L'input della fase di compilazione è il **codice sorgente** in linguaggio di alto livello, l'output è il **codice oggetto** scritto in linguaggio macchina pronto da essere eseguito. In particolare uno schema che permette di riassumere la compilazione è il seguente:



Notiamo che la compilazione si divide in più fasi, ognuna associata ad un componente dell'OS diverso:

- **Lexer:** analizza il codice sorgente e separa i vari caratteri in token, ovvero dei pezzi di codice particolarmente significativi. Al lexer arriva in input il codice sorgente, l'output saranno i **token** ottenuti dal sorgente (esempi di token sono: *int*, *i*, *=*, *if*, *for* ecc).
- **Parser:** fa l'analisi sintattica dei token. I token prodotti infatti devono essere analizzati per capire se sono sintatticamente corretti (cioè se formano degli **statement**).
- **Intermediate code generator:** il passo successivo è la traduzione di questo flusso di token (statement) nelle corrispondenti sequenze di istruzioni di basso livello (in assembly solitamente).
- **Optimizer:** questo insieme di istruzioni passa all'optimizer che ottimizza la sequenza di istruzioni dato che l'operazione di traduzione fatta dal **ICG** non è perfetta. L'optimizer cerca di migliorare l'efficienza del codice e di ridurre lo spazio che quest'ultimo occupa in memoria.
- **Code generator:** l'output dell'optimizer viene passato al code generator che effettua la traduzione finale in linguaggio macchina producendo un oggetto eseguibile.

### Collegamento (linking)

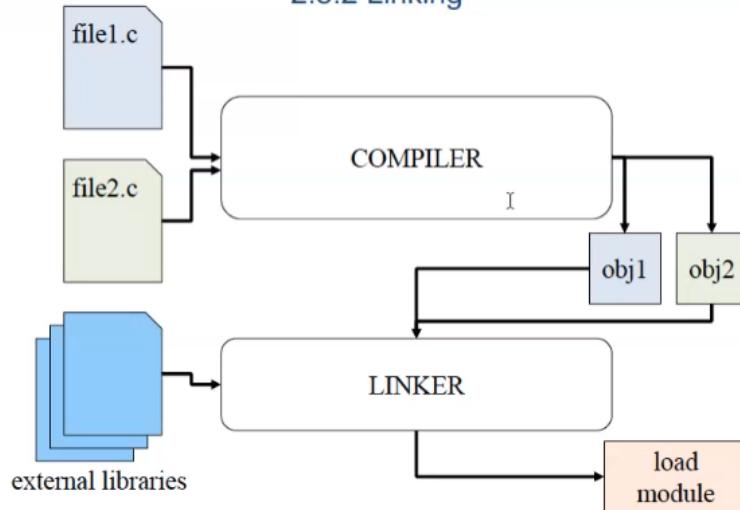
Abbiamo accennato prima come in alcuni casi sia necessario collegare tra loro più programmi affinché si possa eseguire un codice. Infatti capita spesso che i programmi siano formati da sottoprogrammi indipendenti, che prendono il nome di **moduli**. Un esempio di moduli che tutti noi conosciamo sono le librerie di sistema.

Questi moduli vanno quindi collegati tra di loro per ottenere un eseguibile funzionante.

Definiamo quindi collegamento l'operazione per la quale possiamo dividere i programmi in più moduli singolarmente eseguibili collegabili tra loro. Questo tipo di operazione è utile soprattutto per fare analisi di parte di codice. Tipicamente se il programma non fa riferimento ad altri moduli il linking non avviene.

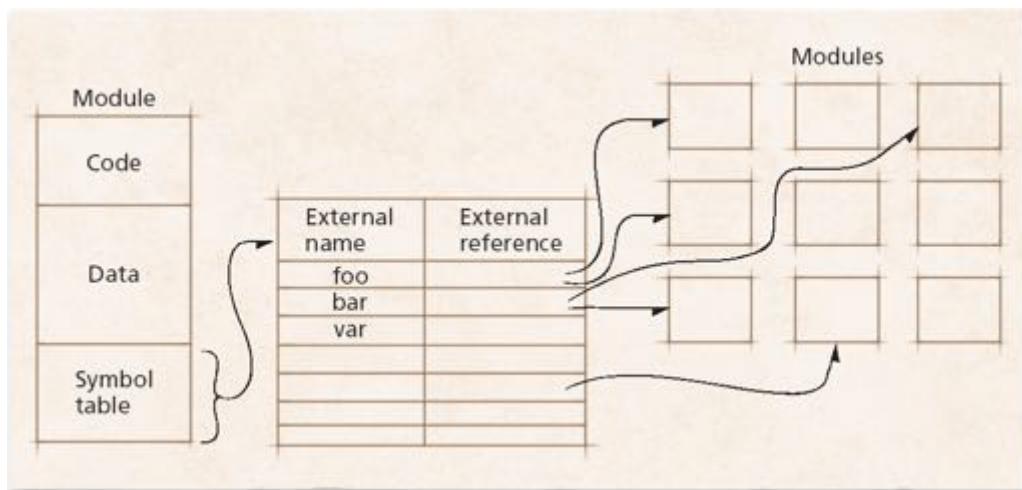
Vediamolo graficamente:

## 2.8.2 Linking

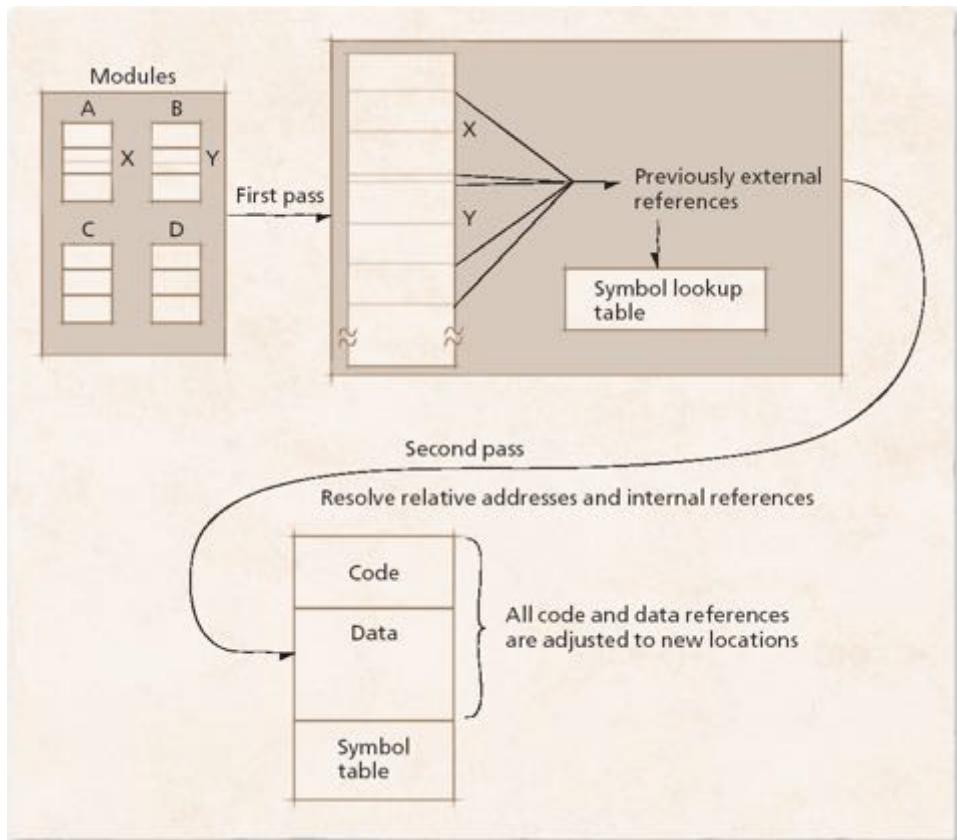


Se nel file1.c si fa riferimento a qualcosa presente nel file2.c allora i due codici oggetto vanno linkati. L'output della fase di linking dà come risultato un **modulo di caricamento** che mette insieme i codici oggetto dei vari programmi pronto ad essere caricato in memoria.

Se il programma fa riferimento a funzioni o dati di un altro modulo, allora il compilatore li etichetterà in **external reference (riferimenti esterni)**, viceversa se il programma rende disponibili le proprie funzioni o i propri dati essi verranno etichettati come **external name (nomi esterni)**. La corrispondenza tra **external reference** ed **external name** viene salvato in una tabella chiamata **symbol table (tabella dei simboli)**. In figura:



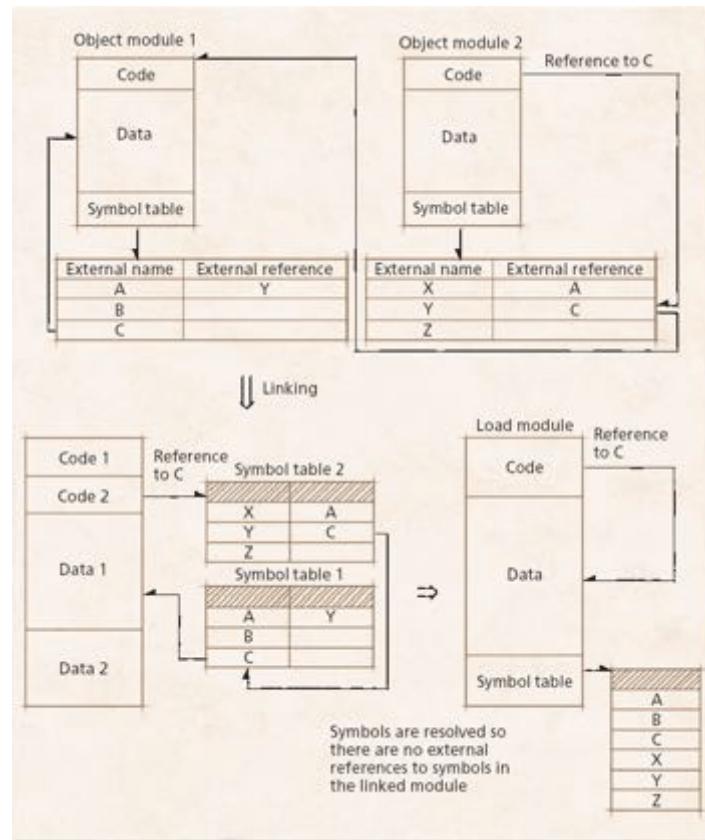
Un'altra operazione che fa il linker è quella di ricollocare **indirizzi relativi** ai codici oggetto dei singoli file. Vediamolo meglio con un [esempio](#):



Immaginiamo di avere 4 moduli (A B C D), come in figura, ognuno dei quali è riferito al codice oggetto di specifici programmi che però poi vengono unificati in un unico modulo di caricamento. Nel codice oggetto di A si fa riferimento ad una variabile x, mentre in B si fa riferimento alla variabile y, se consideriamo i moduli indipendenti allora la variabile x di A sarà associata ad un certo indirizzo 1, ad esempio. Nel modulo B la variabile y si troverà nello stesso indirizzo 1. Cioè, nell'ambito dei moduli separati le variabili si troveranno allo stesso indirizzo. Il linker in questa fase ha lo scopo di riposizionare x e y in posizioni univoche all'interno del nuovo modulo di caricamento. Materialmente posiziona le istruzioni di A in una parte del modulo di loading e le istruzioni di B in un'altra parte del modulo di loading.

Come si mantiene questa associazione? Come detto prima si fa mediante una tabella dei simboli dove memorizziamo le corrispondenze riportandole anche nel modulo unificato.

Infine vediamo un'altra delle operazioni svolte dal linker: la **risoluzione dei simboli**, cioè l'operazione di risoluzione degli external reference rispetto agli external name. In pratica in questa operazione gli external reference all'interno di un modulo sono convertiti nel corrispondente nome esterno in un altro modulo. Vediamolo meglio con un [esempio](#):



Nell'esempio sopra vediamo che il codice oggetto del modulo 2 fa riferimento ad una variabile c. Per capire cosa sia c cerchiamo nella symbol table cosa sia, e ci accorgiamo che è un external reference, quindi devo andare a cercare in un altro modulo un riferimento ad una variabile esterna, nello specifico cerco c nel modulo 1 dove c stessa compare come external name. In modo simile vediamo che nel modulo 1 viene utilizzato viene fatto riferimento ad un external reference y presente nel modulo 2 e così via.

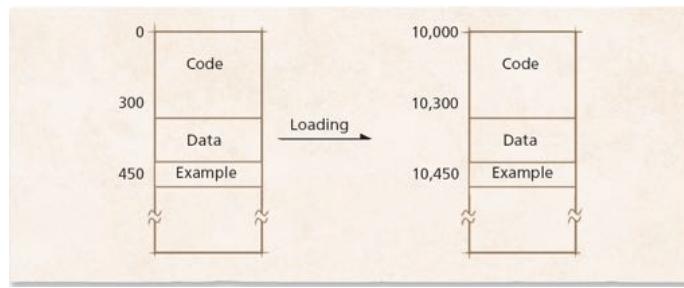
Dopo aver risolto i riferimenti esterni eseguo le operazioni di linking mettendo assieme i codici e i dati dei due moduli e risolvendo le due symbol table. La symbol table finale corrispondente è quella che vediamo in basso a destra della figura, che ovviamente non avrà riferimenti esterni perché saranno stati risolti.

Concludiamo dicendo che il linking può essere eseguito in diverse fasi:

- **Tempo di compilazione:** è possibile eseguire il linking in fase di compilazione solo se non ci sono riferimenti esterni. Ma questo non accade praticamente mai per via dell'utilizzo delle librerie condivise.
- **Prima del loading:** questo linking è possibile per tutti quei programmi differenti che però fanno riferimento a stesse funzioni di libreria senza includerle nel loro codice oggetto.
- **A tempo di caricamento**
- **Runtime:** è comodo quando è necessario linking di file di grande dimensione così da risolvere i riferimenti via via che si presentano e non occupando così spazio inutile in memoria.

## Loading

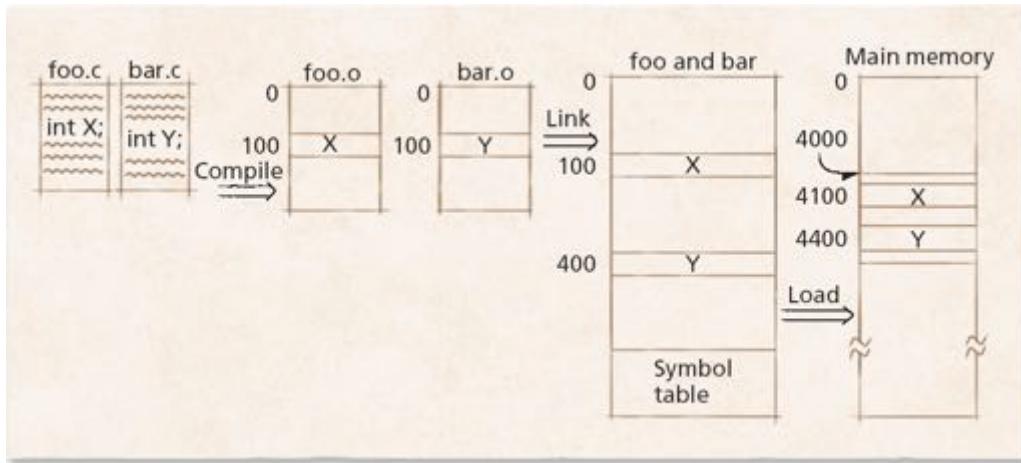
Dopo che il linker ha creato il modulo di loading, quest'ultimo passa ad un programma chiamato **loader** che carica il modulo in memoria. Il loader è responsabile di caricare istruzioni e dati in una particolare zona di memoria. In particolare il loader traduce gli indirizzi relativi nel nuovo **indirizzo assoluto** a partire dalla posizione definita per l'allocazione del modulo.



Esistono 3 modalità di caricamento:

- **Relocatable loading:** questo tipo di caricamento viene utilizzato quando il modulo caricabile contiene indirizzi relativi che devono essere convertiti in reali indirizzi di memoria. Il loader deve quindi richiedere un blocco di memoria in cui posizionare il programma e trasferire gli indirizzi del programma per farli corrispondere alle reali posizioni di memoria.
- **Dynamic loading:** nel caricamento dinamico i moduli vengono caricati solo al momento dell'utilizzo degli stessi.
- **Absolute loading:** tipo di caricamento utile per sistemi molto semplici tipo i sistemi embedded in cui il load module specifica già gli indirizzi fisici. Abbiamo questo caricamento quando il modulo di caricamento stesso specifica gli indirizzi di memoria in cui essere caricato, in quel caso, se è possibile, il loader caricherà direttamente in quegli spazi di memoria.

Riepiloghiamo ora tutto il processo di compilazione, collegamento e caricamento dei programmi con questo esempio:



Si parte dai due file sorgente che vengono compilati. Dopo si creano i file oggetto dove le variabili x e y vengono assegnate a due indirizzi di memoria relativi con valore 100, ognuno specifico di un modulo, dopodiché col linking viene creato un singolo modulo unificato dove vengono ridefinite le posizioni delle variabili e la symbol table e infine nella fase di loading si vanno a caricare i dati in memoria traducendo gli indirizzi relativi (da 0 a 400) in assoluti (da 4000 a 4400) rispettando l'ordine di caricamento originale del programma in memoria.

## Firmware

Oltre all'hardware e al software molti calcolatori contengono il cosiddetto **firmware**, che consiste di istruzioni eseguibili conservate su un supporto di memorizzazione non volatile collegato a un dispositivo. Il firmware è programmato con la **microprogrammazione**, che è uno strato di programmazione al di sotto del linguaggio macchina. Le istruzioni del **micropogramma** prendono il nome di **microcodice**, che include semplici ma fondamentali istruzioni necessarie per realizzare tutte le operazioni in linguaggio macchina.

Al giorno d'oggi, molti dispositivi quale dischi rigidi e altre periferiche, contengono processori in miniatura le cui istruzioni sono spesso implementate in microcodice.

## Middleware

I calcolatori che compongono un **sistema distribuito** sono spesso eterogenei, nel senso che hanno hardware, OS e architetture diverse. Per permettere la comunicazione tra calcolatori così diversi si fa uso del **middleware**, il cui scopo è appunto permettere le interazioni tra diversi processi in esecuzione su uno o più computer collegati in rete, questo, a patto che tutti questi calcolatori abbiano installato il medesimo middleware.

Inoltre, il middleware, semplifica notevolmente lo sviluppo di applicazioni in quanto permette agli sviluppatori di non conoscere i dettagli di come il middleware svolge i suoi compiti, lasciandoli così liberi di concentrarsi sullo sviluppo.

## Processi

Da ora in poi cominceremo a parlare di **processi** e **thread** in sistemi unix, nello specifico Linux.

Cominciamo questo nuovo capitolo tornando un attimo ai concetti di **operazioni concorrenti** e al concetto di **parallelismo**. Le **operazioni concorrenti** sono le operazioni che vengono gestite dall'OS parallelamente. Nel caso degli OS il **parallelismo** fa riferimento alla capacità di potere gestire più operazioni contemporaneamente, anche se in un certo istante il processore sarà in grado di eseguire un solo processo. Per implementare il parallelismo si deve switchare tra più processi tramite il **context switching**.

Obiettivo fondamentale dell'esecuzione concorrente è che i processi non si ostacolino l'uno con l'altro.

Vediamo ora la definizione di **processo**: definiamo processo un qualsiasi programma in esecuzione. Ogni **processo** ha il suo **spazio di indirizzamento** caratterizzato da:

- **Text region**: la zona dello spazio di indirizzamento dove vengono memorizzati il codice e le istruzioni che il processore deve eseguire per quel processo.
- **Data region**: la zona dello spazio di indirizzamento dove vengono memorizzate le variabili e i dati allocati dinamicamente durante l'esecuzione del processo.
- **Stack region**: zona che rappresenta lo stato attuale delle variabili e delle funzioni attive nel momento di esecuzione del processo.

Il compito dell'OS nella gestione dei processi è quello di garantire che ogni processo possa accedere al processore per un tempo sufficiente da essere eseguito e completato. Questo si scontra col fatto che il processore esegue un solo processo alla volta, ecco perché la necessità della esecuzione concorrente e del parallelismo.

Possiamo considerare semplicemente sistemi con un singolo processore, nel caso di sistemi multicore il parallelismo sta nella contesa dei processori disponibili, ma dal punto di vista logico nulla cambia.

## Stati dei processi

Il processo non ha un singolo stato, ma il suo ciclo di vita varia secondo un'insieme di stati ben definiti che sono i seguenti:

- **Running**: stato nel quale il processo è eseguito dal processore in quel momento. L'associazione tra processo e processore la fa un modulo chiamato **dispatcher** con l'operazione di dispatching.
- **Ready**: stato precedente a quello di running, un processo è in questo stato quando è pronto ad essere eseguito dal processore, qualora sia disponibile. Si entra in questo stato quando cariciamo il processo in memoria (materialmente, quando clicchiamo sulla icona di un programma) e se ne esce quando viene assegnato al processore entrando così in fase di

running.

I processi in fase di ready vengono inseriti in una lista chiamata **ready list** che servirà a gestire il parallelismo e a fare scorrere i processi in attesa man mano che il processore completa altri processi.

- **Blocked:** quando il processo in stato di running viene sospeso a causa di un certo evento, tipo una operazione di IO, allora entra in stato di blocked. Nel momento in cui l'operazione di I/O, o più in generale l'evento che lo ha fatto entrare in blocco, termina il processo è libero di uscire dallo stato di blocked. Come avviene con i processi ready, anche quelli blocked vengono posti in una lista apposta chiamata **blocked list**, la quale contiene tutti quei processi che erano in esecuzione e che hanno rinunciato alla esecuzione per completare una operazione (solitamente di I/O). Completata quella operazione il processo bloccato passa nella lista di processi in ready.

Precisiamo che, dato che il processore può eseguire un solo processo alla volta, allora in un sistema (singolo processore) ci sarà uno ed un solo processo in stato di running mentre ce ne potranno essere più in stato ready e blocked. Tra questi ultimi due tipi di processi si devono stabilire delle priorità per la loro gestione, il caso meno performante prevede che la lista venga gestita e mantenuta secondo un metodo di accesso FIFO (first in, first out). In generale, vedremo che, la gestione di queste liste di ready viene fatta dall'OS secondo opportune tecniche di scheduling.

### Gestione dei processi e transizioni di stato

Oltre alla gestione attiva dei processi tramite il meccanismo di esecuzione concorrente l'OS deve garantire anche altri servizi che elenchiamo qui di seguito:

- Creazione dei processi.
- Rimozione dei processi.
- Sospensione dei processi.
- Ripristino dei processi.
- Cambiamento della priorità dei processi.
- Decisione dei processi da mandare in esecuzione.
- Fare comunicare i processi (IPC).

Occupiamoci ora delle transizioni di stato.

Quando mandiamo in esecuzione un programma viene creato il processo associato e viene inserito in ready list. Tutte le operazioni avvengono come evoluzione da uno stato ad un altro, e via via che i processi vengono eseguiti, vengono anche rimossi dalla lista ready così da permettere agli altri processi di avanzare. Quando un processo riesce ad arrivare alla testa della lista si prepara alla transizione di stato da quello ready a quello running. Questa operazione, come detto nel paragrafo precedente, prende il nome di **dispatching** e viene eseguita da un componente chiamato **dispatcher**.

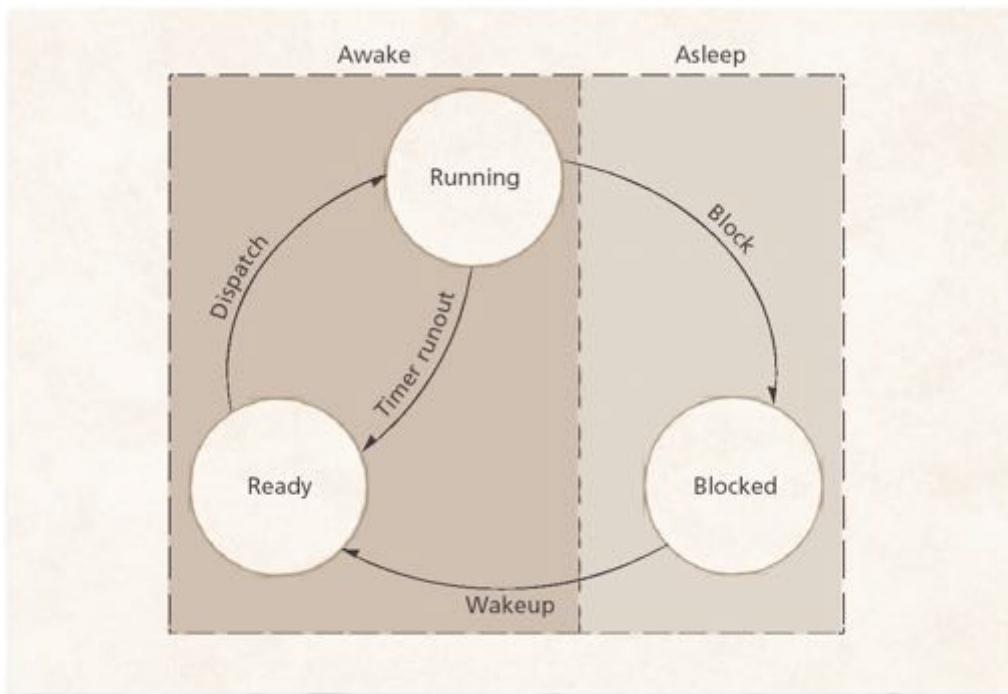
Sia i processi in ready che quelli in running vengono chiamati **processi attivi**, nello stato blocked vengono detti invece **processi dormienti**. Per gestire le transizioni di stato l'OS fa uso di interrupt che abbiamo già visto quando parlavamo della schema madre. Gli interrupt sono dei segnali che l'OS utilizza per controllare il processore, ad esempio, se assegno un processore ad un processo e si verificano delle situazioni critiche in cui il processo va a monopolizzare il processore allora è previsto che l'OS possa interrompere l'esecuzione del processo. A tale scopo, assieme agli interrupt, si definiscono dei **quantum di tempo**, al completamento dei quali, l'OS invia dei segnali al processore per stoppare l'esecuzione del processo attualmente in running. In questo caso l'interrupt consente di favorire il passaggio di stato dallo stato running a ready. Il quantum viene determinato da algoritmi di scheduling appropriati.

Mentre la transizione a running a ready avviene per opera dell'OS, la transizione da running da blocked può avvenire anche per segnalazione stessa del processo. Ad esempio quando un processo fa richiesta di un'operazione di I/O può autosospendersi evitando la monopolizzazione del processore.

Infine, i processi che vanno in stato di blocco non possono essere eseguiti direttamente ma prima dovranno transitare nuovamente nello stato di ready. Ricapitoliamo quindi le transizioni:

- Nella fase di **dispatching** (assegnazione del processore al processo) si passa dallo stato ready a quello running.
- Esaurito un quantum (**time runout**) si passa da running a ready per evitare il monopolio del processore.
- Quando il processo si **blocca** per effettuare una operazione di IO si passa da running a blocked.
- Finita l'operazione che ha bloccato il processo si **sveglia** e passa da blocked a ready.

Vediamo le transizioni in questo diagramma di stato:



### PCB (Process Control Block)

Per gestire l'esecuzione concorrente l'OS deve identificare i processi e memorizzare le informazioni utili a gestirli. L'idea di base è di associare ad un processo una struttura dati contenente tutte le informazioni utili del processo; questa struttura prende il nome di **PCB (Process Control Block)**.

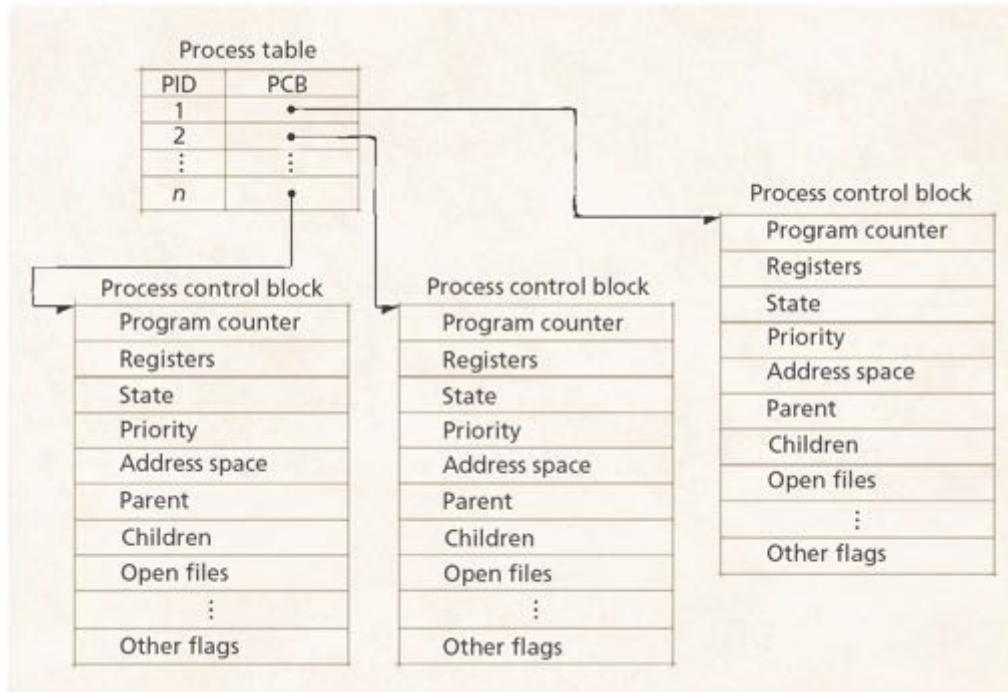
Ma a cosa serve, nella pratica, questo PCB? Supponiamo di avere un processo in esecuzione, e che esso venga fermato per un qualsiasi motivo, sarà allora necessario salvare lo stato e le informazioni del processo per evitare che si perdano i progressi fatti mentre era in esecuzione. Per fare ciò semplicemente carichiamo i dati nel PCB, che viene creato alla creazione del processo.

Vediamo cosa troviamo all'interno del PCB:

- **PID (Processor ID)** cioè un ID numerico univoco associato al processo.
- **Stato del processo**.
- **Program counter** per fare capire al processore quale istruzione deve eseguire del processo.
- La **priorità assegnata al processo** nell'algoritmo di scheduling.
- Le **credenziali del processo**, cioè l'insieme di risorse accessibili al singolo processo.

- **Puntatore al processo padre.** I processi del sistema infatti vengono gestiti con una gerarchia ad albero dove ogni processo si divide in **processi padre e/o processi figlio**. Ogni processo avrà sicuramente un processo padre di riferimento, ma al contrario non è detto che tutti i processi abbiano processi figli.
- **Puntatori ai processi figli** eventuali.
- **Puntatore per dati e istruzioni** del processo allocate in memoria.
- **Puntatori per risorse** del processo, ad esempio i file a cui fa riferimento.
- **Contesto di esecuzione**, ovvero il contenuto dei registri prima che il processo cambiasse stato di running.

Ad ogni processo quindi si associa un PCB identificato dal PID del processo. Il sistema operativo memorizza in una tabella chiamata **process table** un puntatore per ogni PCB e il relativo PID così da poterli trovare in maniera veloce. Vediamo meglio quanto detto in figura:



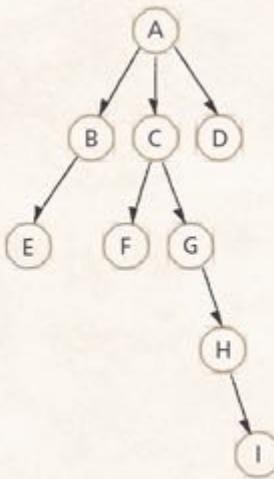
Nel momento in cui un processo viene terminato ed esce dai 3 stati visti precedentemente i dati ad esso associati non sono più necessari e quindi possiamo liberare tutte le risorse e lo spazio in memoria utilizzato per il processo, conseguentemente elimineremo anche il PCB e la riga che lo identifica nella process table.

### Gerarchia dei processi

La creazione di nuovi processi è legata alla **gerarchia dei processi**, infatti la creazione di un processo figlio passa sempre dal processo padre ad esso associato. L'albero dei processi prevede che ciascun figlio abbia solo un processo padre.

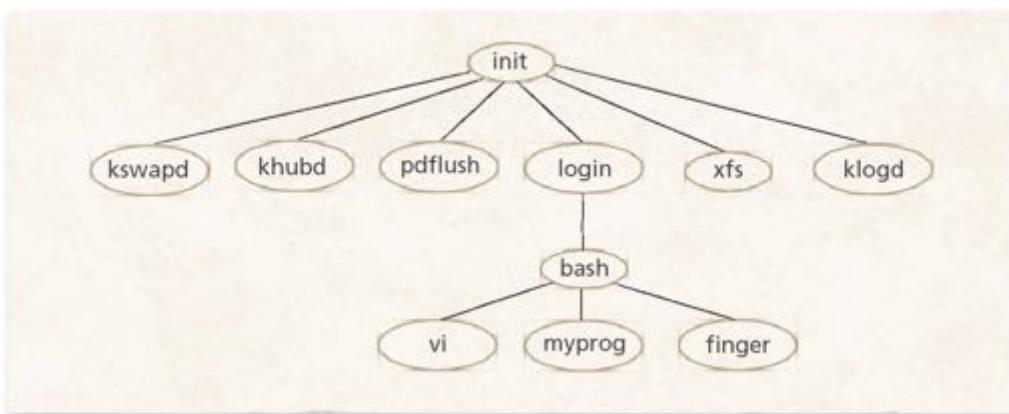
Mentre, la rimozione dei processi richiede la cancellazione di tutti i dati relativi al processo e, da sistema a sistema cambia il modo in cui l'OS gestisce la cancellazione dei processi. In particolare, spesso e volentieri, distruggere un processo padre significa terminare anche i relativi processi figli, ma ci possono essere casi in cui i padri consentono ai figli di continuare la loro esecuzione nonostante la terminazione del padre, questo comporta l'eredità del processo figlio che viene agganciato ad un nuovo processo padre.

Vediamo la **gerarchia ad albero** dei processi:



Se ad esempio eliminassi il processo G e avessi ancora attivi i due figli H e I l'OS avrebbe due opzioni: o terminare entrambi i processi figli oppure fare ereditare H e/o I ad un altro processo. Il processo che solitamente eredita i processi orfani è il **processo radice**, in questo specifico caso A.

Nel caso particolare dei processi linux:

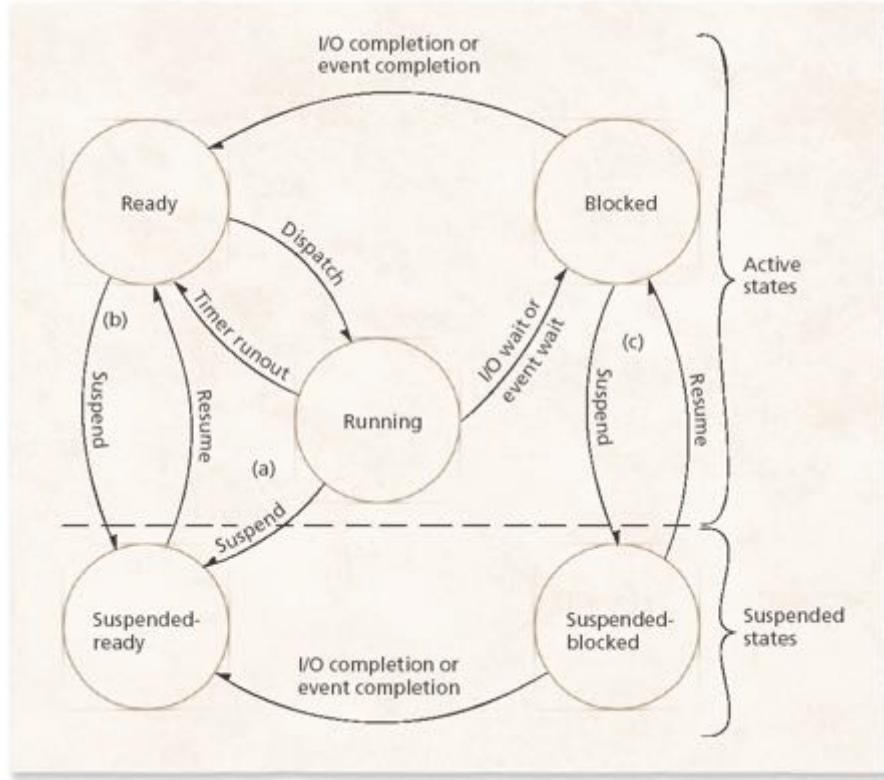


Il padre di tutti i processi in Linux è **init**. Uno dei figli è il processo **login** il cui figlio è **bash** (**terminale**) con cui posso eseguire altri programmi. Se termino **bash** i 3 programmi figli potrebbero essere terminati con lui o eventualmente essere ereditati da **init**.

### Sospensione dei processi

Il diagramma degli stati del processo che abbiamo visto prima è corretto, ma incompleto. Per completarlo dobbiamo infatti considerare altri due stati intermedi che prendono il nome di **sospensione blocked** e **sospensione ready**. Entrambi questi stati discendono dal concetto di **sospensione**.

In particolare, un processo si dice essere in **sospensione** quando non compete più per l'esecuzione del processore ma in ogni caso va in uno stato di permanenza nel sistema. Questo stato si dirama in altri due stati che come detto prima prendono il nome di: **suspended-ready** e **suspended-blocked**. Facendo il **ripristino** di processi suspended-blocked questi ultimi andranno in **block list**, viceversa in **ready list**, questo dipende da quale era lo stato precedente allo stato di sospensione. Graficamente:

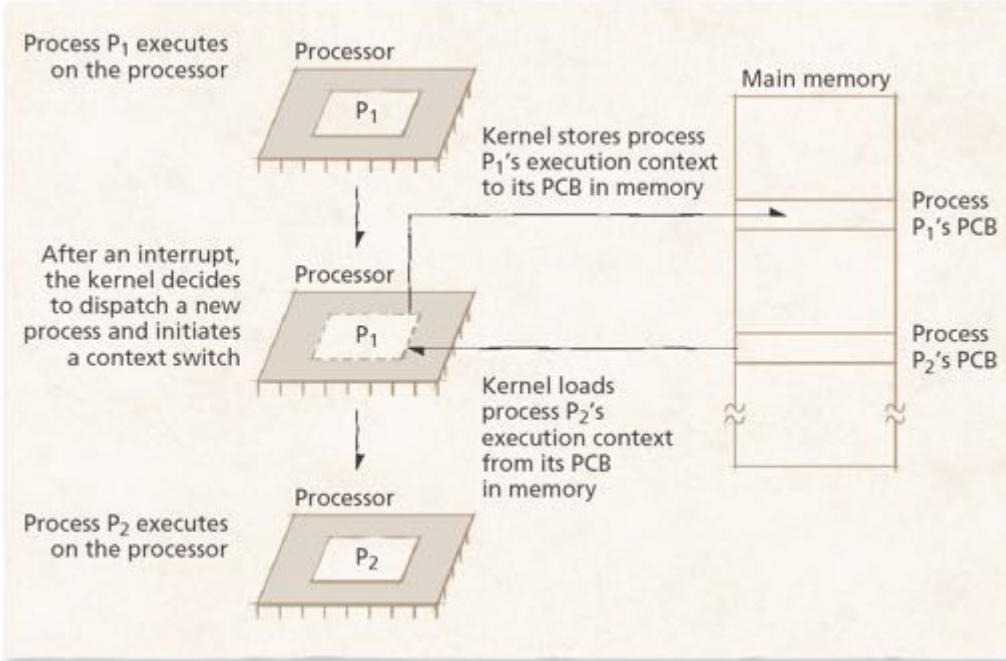


### Cambio di contesto (Context switching)

Il **cambio di contesto** è un'operazione che consente di terminare la esecuzione di un processo in running rimuovendolo e sostituendolo con uno in lista ready. Per eseguire un cambio di contesto è necessario prima memorizzare il PCB del processo running, poi caricare il PCB del processo ready da eseguire.

Il context switching deve essere totalmente trasparente ai processi nel senso che i processi non devono essere consapevoli di essere rimossi dal processore. Allo stesso tempo il processore non deve mai interrompere le operazioni che sta facendo per eseguire un context switching, non a caso, il cambio di contesto è una operazione di **overhead**, ovvero che di per se non serve ad eseguire i calcoli per cui quel sistema è stato pensato, perciò per evitare sovraccarichi del processore va realizzato da hardware che ha come unico scopo quello di effettuare il context switching.

Si può schematizzare in questa maniera:



Dopo un interrupt il kernel dell'OS decide di mandare in esecuzione un nuovo processo col context switching. Viene memorizzato il PCB del processo 1 in memoria e viene caricato in memoria il processo 2 assieme al suo PCB, che il processore potrà eseguire avendo però prima caricato nei suoi registri il PCB del processo relativo, ammesso esistesse già prima, in caso contrario lo creerebbe.

# Lezione 6

---

## Interrupt ed eccezioni

È possibile inviare dei segnali attraverso i dispositivi hardware e, nello specifico, questi segnali saranno diretti verso i processi. Questi segnali prendono il nome di **interrupt** e servono a gestire il verificarsi di alcune condizioni specifiche. Gli interrupt vengono gestiti dal processore e ne esistono di due diverse tipologie:

- **Sincroni:** si hanno in risposta diretta e sequenziale ad un certo evento. In generale, un fenomeno sincrono avviene immediatamente dopo l'evento che lo ha scaturito. Gli interrupt sincroni sono riferiti ai processi in running.
- **Asincrono:** si hanno in un momento diverso da quello che ha scatenato l'evento. Dipendono da fenomeni esterni rispetto all'evento che stiamo osservando (ovvero dal processo in esecuzione). Questo tipo di interrupt avviene quando si verificano degli eventi di tipo hardware esterni al processo in esecuzione ma che devono essere comunque comunicati al processore e dunque ciò verrà fatto in modo asincrono.

Queste operazioni rappresentano un metodo efficace di notifica di eventi e hanno un basso overhead, ovvero non inficiano più di tanto nel carico di lavoro del processore.

Un'alternativa agli interrupt è l'operazione di **pooling**: piuttosto che aspettare dei segnali che arrivano dall'esterno, il processore va a chiedere spesso lo stato dei device. Questo aumenta ovviamente l'overhead ed è quindi una operazione sconsigliata.

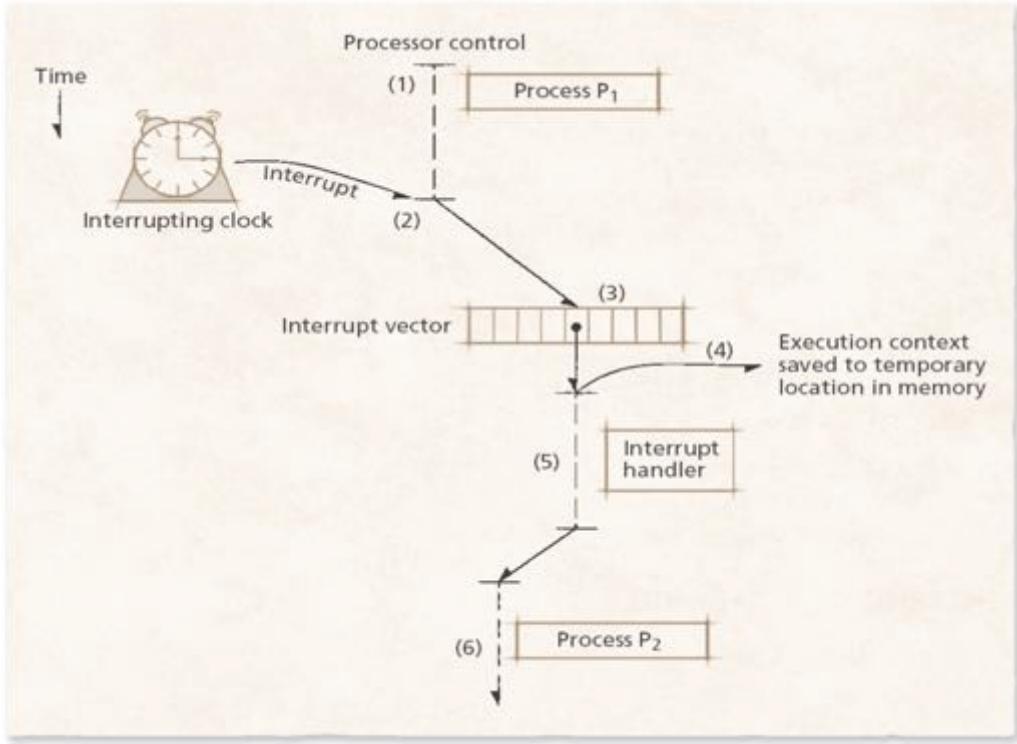
Per **overhead** intendiamo una serie di task che servono a supportare una certa operazione eseguita dal processore. La gestione degli interrupt va fatta parallelamente agli altri task del sistema e dunque va fatta efficacemente per evitare di sovraccaricare il processore.

Per gestire gli interrupt a livello fisico si utilizzano le linee di interrupt, ovvero linee elettriche stampate nella scheda madre, gestite da un controller specifico del processore che ha il solo obiettivo di gestire gli interrupt. Di per se l'interrupt richiede di sospendere ciò che si sta facendo per svolgere un altro compito. Ma non tutti gli interrupt hanno la stessa priorità, ognuno ha associata una priorità diversa. Nel momento in cui in partono degli interrupt viene infatti creata una coda di interrupt che verranno gestiti solo dopo che il processore avrà completato l'operazione corrente, questo perché come sappiamo il processore non può interrompere bruscamente una istruzione per gestire l'interrupt, bensì dovrà completarla, mettere in pausa il processo e poi svolgere l'interrupt.

La gestione degli interrupt dipende dalle politiche del sistema, ad ogni tipo di interrupt corrisponde una azione in risposta data dal sistema. Successivamente, dopo aver gestito l'interrupt si può riprendere l'esecuzione del primo processo in ready list o, in alternativa, in alcuni sistemi invece si riprende il processo che era stato messo in pausa per gestire l'interrupt, ma questo dipende dalle politiche di scheduling. Di base però il nuovo processo ad andare in running sarà il primo in ready list.

Quindi dopo un interrupt un nuovo processo può essere mandato in running, quale sarà questo processo lo deciderà lo scheduler.

Vediamo ora di riassumere in figura quanto accade:



Concettualmente interrupt ed **eccezioni** sono la stessa cosa, entrambi notificano determinati eventi, ma in alcune architetture si fa una differenza nel senso che, per interrupt si intendono generici eventi di cui si vuole dare notifica, mentre **l'eccezione** è tipicamente riferita ad errori di sistema. Questa differenza è una specifica di alcune architetture particolari ma, di per se il segnale è lo stesso, cambia solo l'etichetta associata al segnale.

Le eccezioni si differenziano in eccezioni che possono essere gestite ed eccezioni che non possono essere gestite.

Vediamo ora quali possono essere le cause scatenanti di interrupt ed eccezioni:

- **Interruzioni**

- **I/O**: inizializzati da device di I/O.
- **Timer**: quando è passata una certa quantità di tempo.
- **Processore**: fatti dal processore per notificare dispositivi del sistema.

- **Eccezioni**

- **Aborto**: l'esecuzione del processo non può continuare e il processo va terminato.  
Questa eccezione quindi non può essere gestita.
- **Fault e Trap**: eccezioni che possono essere gestite e i processi possono essere recuperati.

## Comunicazione tra processi

Un ulteriore meccanismo che forniscono gli OS è quello di **comunicazione tra processi (IPC)**. Questo è necessario perché per alcune operazioni può essere utile, se non addirittura necessario, che i task svolti da più processi siano connessi tra di loro così da coordinarsi per il raggiungimento di un obiettivo comune. Quest'ultimo concetto si definisce **sincronizzazione** e permette a più processi di essere sincronizzati. I processi sincronizzati potranno comunicare tra di loro in modo tale da eseguire un task senza che ci siano errori.

## Segnali (signal)

Gli interrupt di tipo software prendono il nome di **segnali (signal)** e vengono mandati da un processo all'altro per comunicare eventi. I signal vengono gestiti dal sistema operativo. I signal possono essere gestiti, si possono ignorare e si possono anche bloccare, in particolare nel momento in cui li blocchiamo è come se un processo dicesse all'OS che non vuole ricevere signal di quel tipo, questa operazione viene detta di **masking** e serve ad evitare sovraccarico del processore.

Il meccanismo di **send-receive** prevede che vadano specificati processo mittente e destinatario nei signal. In particolare i processi possono scambiarsi signal che prendono il nome di **messaggi**. Distinguiamo due politiche nell'invio di messaggi:

- **Blocking-send**: comunicazione di tipo sincrona. Quando un processo invia un messaggio blocking-send, prima di andare alla istruzione successiva devo aspettare che il destinatario lo abbiamo ricevuto e ce lo notifichi.
- **Non blocking-send**: consente al mittente di inviare un messaggio e immediatamente continuare con la istruzione successiva senza aspettare conferma di ricezione del messaggio.

Tale situazione prevede un meccanismo di memorizzazione dei messaggi temporanea in attesa che quest'ultimo venga letto dal destinatario.

Vedremo, a tal proposito, uno specifico tipo implementazione di scambio di messaggi chiamato **pipe**. La **pipe** è un parte di memoria che svolge il ruolo di buffer, ovvero che consente a un processo di scrivere su di essa un messaggio e ad un altro processo di leggere man mano cosa c'è scritto. La caratteristica della pipe è che l'accesso è sincronizzato ed è gestito dall'OS. Via via che i dati vengono letti dalla pipe, vengono anche automaticamente rimossi per fare spazio ad altri dati. Quando il **processo reader** completa la lettura di tutti i dati nel buffer il OS termina l'esecuzione e l'OS dà quindi la possibilità al **processo writer** di scrivere.

## Processi UNIX

I processi in UNIX interagiscono con il sistema operativo tramite le chiamate di sistema. In UNIX i processi sono caratterizzati da priorità che hanno valori compresi tra -20 e 19. Più basso è il valore maggiore è la priorità del processo. I processi che effettuano operazione di manutenzione periodicamente vengono chiamati processi demoni e hanno tipicamente una bassa priorità.

Vediamo alcune chiamate di sistema in UNIX:

<i>System Call</i>	<i>Description</i>
fork	Spawns a child process and allocates to that process a copy of its parent's resources.
exec	Loads a process's instructions and data into its address space from a file.
wait	Causes the calling process to block until its child process has terminated.
signal	Allows a process to specify a signal handler for a particular signal type.
exit	Terminates the calling process.
nice	Modifies a process's scheduling priority.

Come sappiamo esistono delle chiamate di sistema che svolgono task di basso livello e sono applicate dalle applicazioni sviluppate dagli utenti. Una di queste chiamate di sistema è **fork()**, che serve per clonare i processi. Un processo facendo una chiama a fork crea una copia esatta di se stesso continuando il flusso di esecuzione dal punto in cui fork() è stata terminata in avanti. Il processo clone non farà le stesse cose del processo padre, i due processi infatti diventano totalmente indipendenti. L'unica cosa che dobbiamo tenere a mente è che con l'utilizzo di alcuni istruzioni è possibile fare in modo che il processo clonato per terminare aspetti la terminazione

del processo clone **Exec()** consente al processo di eseguire all'interno del proprio spazio un altro processo. Solitamente va utilizzato assieme a fork() subito dopo la creazione del processo figlio. **Wait()** consente al processo padre di aspettare la terminazione dei processi figli prima di terminare. **Signal()** permette di inviare segnali da un processo all'altro e di gestire i segnali. **Exit()** serve a distruggere il processo che l'ha invocata liberando la memoria da qualsiasi dato e istruzione di quest'ultimo. **Nice()** modifica la priorità di scheduling dei processi.

Vedremo in maniera specifica ognuna di queste system call nelle lezioni successive.

# Lezione 7

## Shell Linux

La **shell** è un interprete di comandi ed è il terminale che ci consente di interagire con l'OS. Quella più comune si chiama **BASH (Bourne again shell)** ed è una evoluzione della shell standard di linux, che si trova dentro /bin.

La shell si presenta solitamente con una riga del genere:

```
myusername@mymachinename:~$
```

La prima parte indica lo user che sta usando la shell e successivamente la macchina che sta usano. A seguire troviamo la tilde che indica che siamo nella home e il simbolo di dollaro che indica che siamo un utente normale (non admin, se fossimo admin vedremmo il simbolo #).

## Comandi di base

- **ls**: mostra contenuto cartella. Può essere usato con una serie di parametri:
  - **-a**: visualizza anche gli elementi di quella directory i cui nomi cominciano col punto.
  - **-l**: sta per long. Serve a visualizzare in maniera estesa i file contenuti in una directory. Nello specifico è possibile fare il reindirizzamento dei comandi.
  - **-h**: può essere usata con -l e consente di visualizzare informazioni relative alla lunghezza dei file in Byte, Kb, MB ecc..
- **cd**: cambiare la directory.
- **cp**: copia file o directory.
- **mv**: sposta o rinomina un file o una directory.
- **rm**: rimuove un file o una directory.
- **mkdir**: crea una directory.
- **pwd**: stampa la directory corrente.

Vediamo ora un esempio di uso del comando ls:

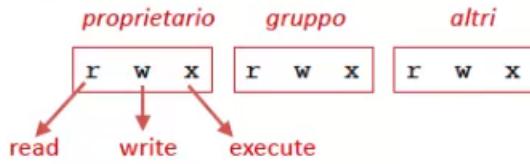
```
marco@mubuntu:~/Scrivania/demo$ ls -l
totale 12
drwxr-xr-x  2 marco marco  4096  mar 11 14:50  test
-rw-r--r--  1 marco marco    12  mar 11 14:49  test1.txt
-rw-r--r--  1 marco marco     6  mar 11 14:50  test2.txt
```

La prima linea visualizza il numero di blocchi usati dai file all'interno della directory. Riconosciamo questo valore perché etichettato da "Totale". La dimensione dei blocchi di default è 512 byte.

Notiamo che ogni file è diviso in 8 colonne che rappresentano ognuno una informazione diversa. In particolare cominciamo analizzando la prima colonna che notiamo essere formata da una serie di caratteri il cui significato è legato ai permessi che ha quel file (o directory), nello specifico vediamo cosa significano queste lettere nella tabella qui sotto:

-	file regolare
d	directory
b	dispositivo a blocchi
c	dispositivo a caratteri
l	collegamento simbolico
p	named pipe
s	socket in dominio Unix

## PERMESSI



Abbiamo 3 tipi di permessi, che sono **r** per la lettura, **w** per la scrittura e **x** per l'esecuzione. In ordine da sinistra verso destra i caratteri rappresentano i permessi per il proprietario, quelli per il gruppo e quelli per gli altri utenti.

La seconda colonna indica un valore numerico conosciuto come link (cioè il numero di collegamenti fisici al file o alla directory).

Nelle due colonne successive abbiamo rispettivamente il proprietario e il gruppo del file/directory.

La quinta colonna rappresenta la dimensione dell'elemento in byte.

Poi abbiamo la data di ultima modifica e infine il nome del file.

Vediamo ora qualche esempio di permessi:

<b>drwxr-xr-x</b>	directory leggibile, scrivibile e attraversabile dal proprietario, leggibile e attraversabile per il gruppo e per gli altri
<b>drwx-----</b>	directory leggibile, scrivibile e attraversabile dal proprietario, ma inaccessibile per tutti gli altri
<b>drwxrwxrwx</b>	directory leggibile, scrivibile e attraversabile da tutti
<b>-rw-----</b>	file leggibile e scrivibile solo dal proprietario
<b>-rw-r--r--</b>	file leggibile da tutti, ma scrivibile solo dal proprietario
<b>-----r--</b>	file leggibile da tutti eccetto il proprietario e gli utenti appartenenti al gruppo di utenti assegnato al file.

I permessi li possiamo assegnare e combinare come vogliamo, non abbiamo limitazioni in tal senso, purché rispettino la sintassi vista.

I permessi possiamo rappresentarli, volendo, anche in notazione ottale come si vede nella seguente tabella:

Cifra	simboli	Permessi
0	---	nessun permesso
1	--x	esecuzione
2	-w-	scrittura
3	-wx	scrittura ed esecuzione
4	r--	lettura
5	r-x	lettura ed esecuzione
6	rw-	lettura e scrittura
7	rwx	lettura, scrittura, esecuzione

Uno dei comandi usati per cambiare i permessi è **chmod** che ci permette di usare una serie di opzioni. In particolare la struttura è la seguente:

```
chmod [opzioni] [--] modalità file1 [file2...]
```

Utilizziamo le seguenti lettere per selezionare di che classi stiamo parlando:

- **u**: seleziona la classe relativa al proprietario
- **g**: seleziona la classe relativa al gruppo
- **o**: selezione relativa agli altri utenti
- **a**: seleziona tutte le classi

Inoltre vengono utilizzati dei simboli per specificare come stiamo variando i permessi, se stiamo aggiungendo, rimuovendo o impostando permessi per le classi selezionate:

- Simbolo **+** : aggiunge permessi specificati alle classi selezionate; non rimuove permessi già concessi ma non specificati.
- Simbolo **=** : imposta i permessi specificati nelle classi selezionate; rimuove eventuali permessi già concessi ma non specificati.
- Simbolo **-** : rimuove i permessi specificati dalle classi selezionate.

Vediamo qualche esempio:

```
chmod "u=rwx", "g=r*", "o=x" nomeFile  
chmod "o=+rx" nomefile  
chmod "u=+rwx", "go=-" nomefile
```

In alternativa si può scrivere con notazione ottale nella seguente maniera:

```
chmod 734 nomefile  
chmod -R 777 nomedirectory
```

Dove ogni numero rappresenta un permesso per ogni classe (utente, gruppo, others in ordine). Il **-R** del secondo esempio sta per "ricorsiva", ovvero permette di applicare i permessi a tutti gli elementi della directory.

Il comando **chown** modifica i permessi relativi al proprietario di un file. Possiamo anche cambiare il proprietario di un file mediante il comando:

```
chown nuovoProprietario nomeFile
```

Se voglio modificare il gruppo eseguo:

```
chown nuovoProprietario nuovoGruppo nomeFile
```

Se voglio modificare solo il gruppo e non il proprietario farò:

```
chown :nuovoGruppo nomeFile
```

Ciascuno dei gruppi avrà politiche di accesso ai file proprie.

### Caratteri Jolly

La shell di linux è anche un interprete. È in grado di espandere alcuni caratteri per tradurli in stringhe dal significato più complesso. Ad esempio:

- \*: qualsiasi carattere scritto 0 o più volte.
- ?: singolo carattere.
- []: singoli caratteri o insieme di caratteri se separati da un trattino.

Vediamo come possono essere utili questi caratteri:

```
user@ubuntuPC:~/example$ ls
libby1.txt libby1.jpg libby2.jpg libby3.jpg libby4.jpg
libby5.jpg libby6.jpg libby7.jpg libby8.jpg libby9.jpg
libby10.jpg libby11.jpg libby12.jpg
```

Comando	Cosa individua
<code>rm libby1*.*</code>	libby10.jpg fino a libby12.jpg, libby1.txt e libby1.jpg
<code>rm libby*.jpg</code>	libby1.jpg fino a libby12.jpg, ma non libby1.txt
<code>rm *txt</code>	libby1.txt ma non libby1.jpg fino a libby12.jpg
<code>rm libby*</code>	libby1.jpg fino a libby12.jpg, ed anche libby1.txt
<code>rm *</code>	Tutti i file nella directory.

```
user@ubuntuPC:~/example$ ls
libby1.txt libby1.jpg libby2.jpg libby3.jpg libby4.jpg
libby5.jpg libby6.jpg libby7.jpg libby8.jpg libby9.jpg
libby10.jpg libby11.jpg libby12.jpg
```

Comando	Cosa individua
<code>rm libby1[12].jpg</code>	libby11.jpg e libby12.jpg, ma non libby10.jpg
<code>rm libby1[0].jpg</code>	libby10.jpg, ma non libby1.jpg
<code>rm libby[6-8].jpg</code>	libby6.jpg fino a libby8.jpg, e niente altro.

# Lezione 8

## Programmazione di processi

Un **processo** è un'istanza di un programma in esecuzione. Esempi di processi possono essere due finestre dello stesso terminale, che saranno due istanze distinte e separate. Il processo deve essere identificato dentro linux in maniera univoca mediante un ID numerico a 16 bit chiamato **Process ID (PID)**.

Ricordiamo che i processi in linux sono organizzati mediante una gerarchia, quindi ogni processo sarà caratterizzato da un padre ed eventuali figli. Essendo la gerarchia ad albero, avremo un processo radice, che in linux è il processo init.

### Funzioni getpid() e getppid()

Per riconoscere i processi utilizziamo due chiamate di sistema incluse nella **libreria <unistd.h>**. Le prime due funzioni che vediamo sono la funzione **getpid()** e **getppid()** che non accettano argomenti. Queste due funzioni restituiscono l'id del processo corrente e l'id del processo padre del processo chiamante. Vediamo un [esempio](#) di programma in c che utilizza queste due funzioni:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(){
    printf("The process ID is %d\n", (int) getpid());
    printf("The parent process ID is %d\n", (int) getppid());
    return 0;
}
```

Il valore di ritorno di queste due funzioni è di tipo **pid\_t** (che è una sorta di signed int) ed è una struttura usata in unix per contenere il PID che è un intero (ecco perché eseguiamo il cast ad int). Per potere leggere e interpretare questa struttura correttamente dobbiamo importare la libreria **<sys/types.h>**. Eseguendo questo codice più volte otteniamo processi di volta in volta diversi, per ogni esecuzione di questo programma il PID cambierà. Se compiliamo questo programma da terminale allora il processo genitore sarà la finestra di terminale da cui abbiamo invocato il processo.

### Comando ps

Esiste un comando di sistema chiamato **ps** che consente di visualizzare lo stato di tutti i processi in esecuzione sul sistema. Nello specifico, tornando all'esempio di prima, questo comando può essere utile per verificare la correttezza o meno dei dati ottenuti dal programma compilato precedentemente.

Questo comando ha diverse opzioni, ad esempio, **-e** che specifica di visualizzare tutti i processi in esecuzione in qualsiasi finestra (non solo quella corrente), mentre con **-o** specifichiamo quali elementi vogliamo visualizzare per ogni processo (ad esempio: ps -o pid, ppid, command).

### Funzione system()

Esistono delle funzioni di sistema che ci consentono di creare processi con determinate caratteristiche. Nello specifico se voglio creare un processo che usa un comando di sistema, si può utilizzare la funzione **system()** che crea un processo che esegue il comando di sistema che noi passiamo come parametro. Vediamolo meglio nell'[esempio](#) di seguito:

```
#include <stdlib.h>

int main(){
    int return_value;
    return_value = system('ls -l /');
    return return_value;
}
```

L'output del programma sarà l'output del comando di sistema ls -l eseguito dal terminale dal quale stiamo compilando.

### Funzione fork()

Vediamo ora di riprendere la funzione **fork()** e della funzione **exec()**. **fork()** crea un processo figlio copia esatta del processo padre. **exec()** consente di incorporare all'interno di un processo il codice di un altro processo. Capiamo meglio la loro utilità con qualche esempio.

Per utilizzare fork() devo importare la libreria **unistd.h** tramite il comando **#include <unistd.h>**. La chiamata alla funzione fork() non avrà parametri e ritornerà una struttura di tipo pid\_t (cioè: **pid\_t fork(void)**).

Immaginiamo di avere un processo e nel codice abbiamo una chiamata a fork(). Con la chiamata a fork() il processo viene duplicato, da quel momento in poi sia processo padre sia processo figlio continuano la loro esecuzione dal punto successivo al quale è stata richiamata fork. I due processi padre e figlio si distinguono mediante il loro PID.

Subito dopo aver fatto fork, la funzione ritorna una struttura pid\_t con dei valori di PID. Nello specifico il processo padre esegue fork e il risultato della funzione fork è il PID del processo figlio che ha appena creato mediante duplicazione. Il processo figlio continua la sua esecuzione dal punto successivo alla chiamata fork, cioè non esegue la fork. Il PID che leggerà il processo figlio è 0, quindi quello che ritorna la funzione fork al figlio è 0.

Dal momento che fork duplica i processi per clonazione significa che padre e figlio verranno eseguiti in spazi di memoria separati, anche se uguali l'uno all'altro (almeno prima di continuare la loro esecuzione). Questo significa che dalla istruzione successiva a fork in poi le operazioni che avverranno su un processo non avranno effetto sull'altro processo, e viceversa. Tutto l'insieme di indirizzi a cui fa riferimento il padre sono replicati nel figlio. In particolare se il padre apre dei **file descriptor** (sono degli indici utilizzati in unix che servono per avere accesso a stream di file e di I/O, in particolare sono come dei puntatori a questi stream), allora essi vengono duplicati nel figlio. Alcune informazioni non vengono duplicate con fork, ad esempio i memory lock e anche statistiche relative alle risorse di sistema utilizzate da un processo.

Vediamo un primo [esempio](#) di fork:

```

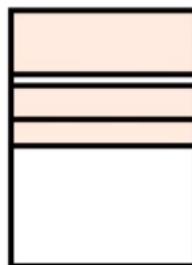
00: #include <unistd.h>
01:
02: int main() {
03:
04:     pid_t pid;
05:
06:     pid = fork();           ←
07:
08:     if (pid == 0) {
09:         // code of the child process
10:     }
11:
12:     else {
13:         // code of the parent process
14:     }
15:
16:     return 0;
17: }

```

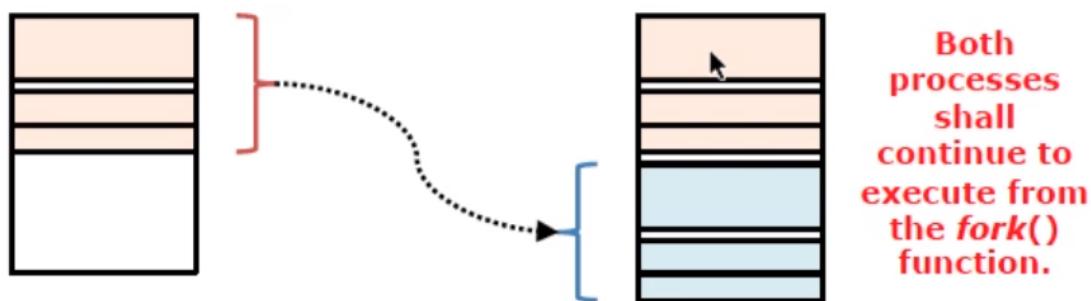
Vediamo le righe una alla volta:

0. Include della libreria UnixStandard.h
1. Abbiamo il main.
2. Creiamo una variabile di tipo pid\_t di nome pid.
3. Faccio la chiamata a fork, facendo la chiamata supponendo che la fork venga effettuata correttamente allora il valore che assegno a pid sarà 0 per il processo figlio e il PID del figlio per il processo padre. Quindi da ora in poi PID sarà differente tra figlio e padre. Teniamo a mente che la variabile pid in realtà non è il PID del processo, ogni figlio ha il suo PID. È solo la variabile che prende il valore 0 alla creazione della fork.
4. Per distinguere i comportamenti di padre e figlio faccio un if per separare il codice che viene eseguito dal figlio ( $pid == 0$ ), da quello che viene eseguito dal padre (else).
5. Eventualmente se la fork dà errore il valore di ritorno in PID sarà -1.

Fino a prima della chiamata fork non esiste alcun processo padre né figlio, ma esisterà un unico processo allocato in un'unica zona di memoria:



Il processo che esegue la chiamata fork diventa processo padre:



Tutti e due i processi continueranno la loro esecuzione dalla fork in avanti (dalla riga 8). Per il figlio avremo:

### child

```
00: #include <unistd.h>
01:         }
02: int main() {
03:
04:     pid_t pid;
05:
06:     pid = fork();
07:
08:     if (pid == 0) {
09:         // code of the child process
10:     }
11:
12:     else {
13:         // code of the parent process
14:     }
15:
16:     return 0;
17: }
```

Entrambi faranno il controllo per vedere se il loro pid è uguale a 0 o meno. Il padre passerà direttamente alla riga 13 ovviamente:

### parent

```
00: #include <unistd.h>
01:
02: int main() {
03:
04:     pid_t pid;
05:
06:     pid = fork();
07:
08:     if (pid == 0) {
09:         // code of the child process
10:     }
11:
12:     else {
13:         // code of the parent process
14:     }
15:
16:     return 0;
17: }
```

Mentre il figlio eseguirà i comandi dentro l'if:

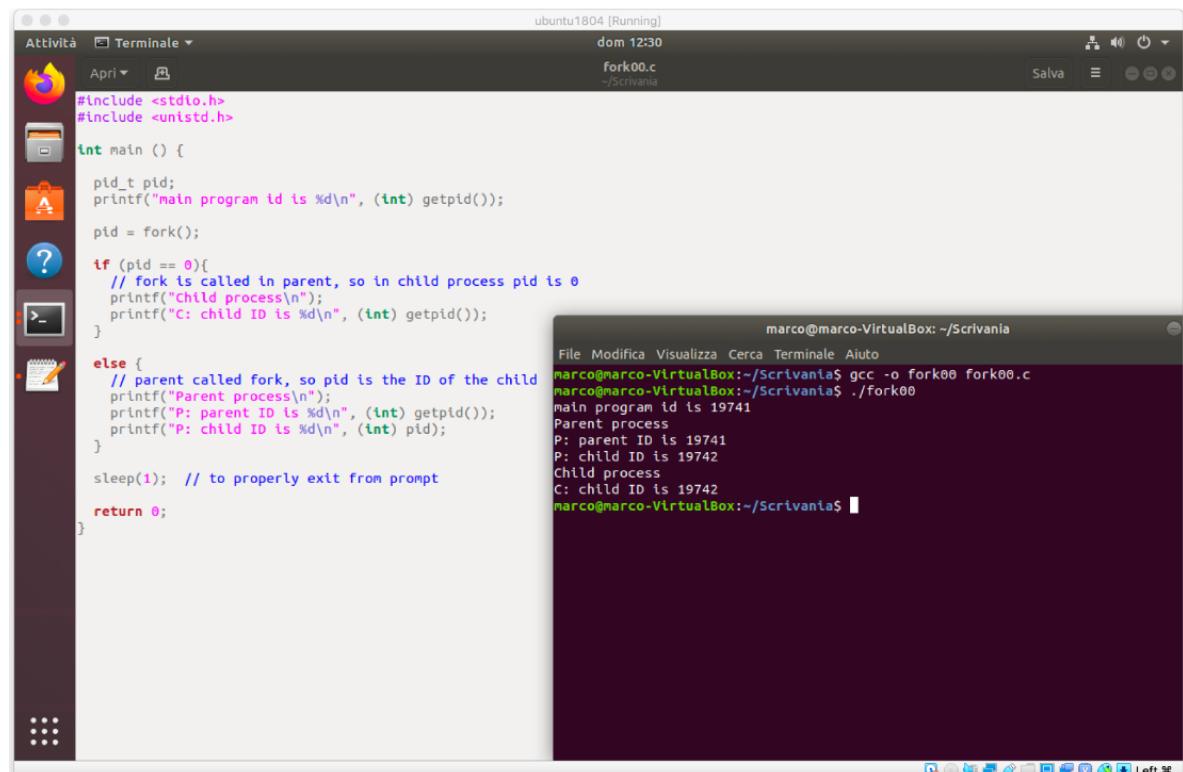
## child

```
00: #include <unistd.h>
01:
02: int main() {
03:
04:     pid_t pid;
05:
06:     pid = fork();
07:
08:     if (pid == 0) {
09:         // code of the child process
10:     }
11:
12:     else {
13:         // code of the parent process
14:     }
15:
16:     return 0;
17: }
```

Alla fine entrambi faranno il return 0;

Vediamo un esempio completo con l'utilizzo di fork:

La funzione fork restituisce quindi al processo padre il PID del processo figlio, mentre restituisce al processo figlio il valore 0. Per ottenere quindi il PID reale del figlio, dentro l'if dovremo mettere la printf di getpid (printf("%d", getpid())), e non di pid, che sennò ci restituirebbe 0.



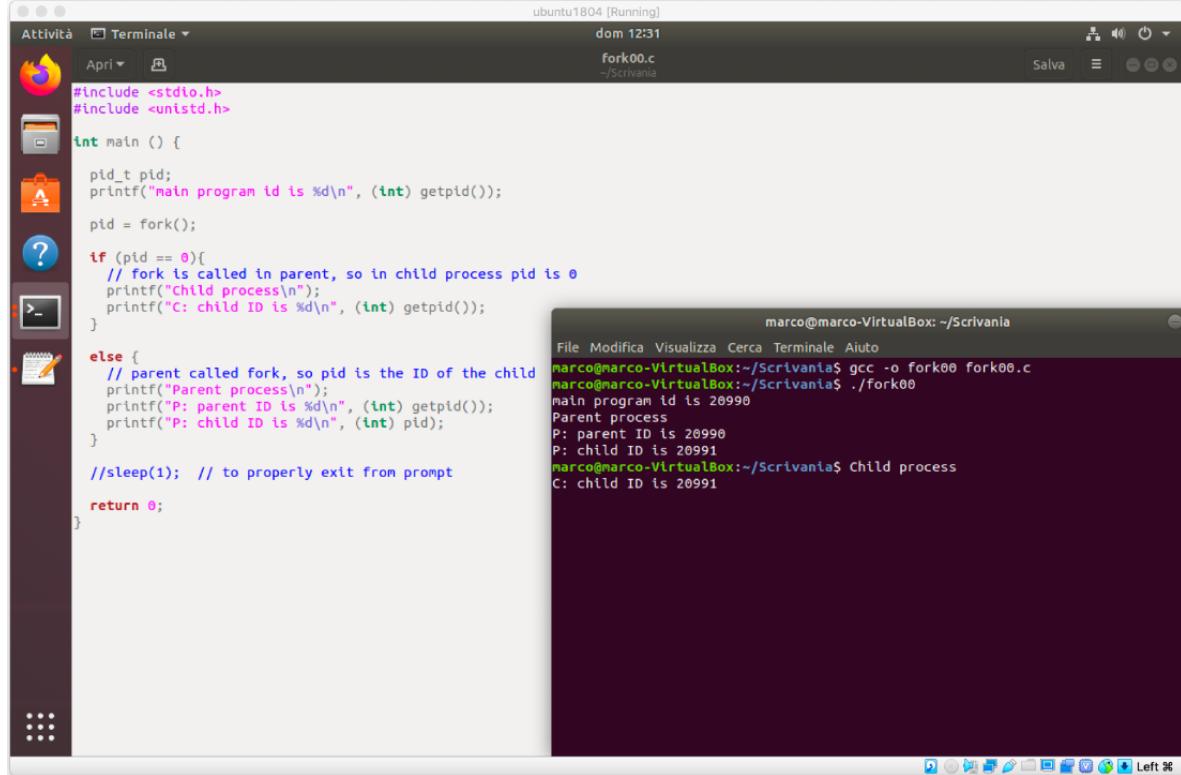
```
#include <stdio.h>
#include <unistd.h>

int main () {
    pid_t pid;
    printf("main program id is %d\n", (int) getpid());
    pid = fork();
    if (pid == 0){
        // fork is called in parent, so in child process pid is 0
        printf("Child process\n");
        printf("C: child ID is %d\n", (int) getpid());
    }
    else {
        // parent called fork, so pid is the ID of the child
        printf("Parent process\n");
        printf("P: parent ID is %d\n", (int) getpid());
        printf("P: child ID is %d\n", (int) pid);
    }
    sleep(1); // to properly exit from prompt
    return 0;
}
```

```
marco@marco-VirtualBox:~/Scrivania$ gcc -o fork00 fork00.c
marco@marco-VirtualBox:~/Scrivania$ ./fork00
main program id is 19741
Parent process
P: parent ID is 19741
P: child ID is 19742
Child process
C: child ID is 19742
marco@marco-VirtualBox:~/Scrivania$
```

Notiamo che quando si crea un nuovo processo, quest'ultimo va nello stato di ready, e in questo caso specifico il processo padre sarà in stato di running e il figlio sarà in ready. Come verranno eseguiti ed assegnati i processi al processore dipende come al solito dalle politiche di scheduling. Comunque non c'è una regola che dice che prima viene eseguito il padre e poi il figlio, o viceversa, l'ordine in cui si alternano i processi dipende solamente dallo scheduler.

La funzione sleep(1) nel codice serve a forzare il processo ad andare in sospensione per una quantità di tempo determinata, in questo caso 1 secondo. Ciò significa che nel momento in cui il padre è andato in sleep, contemporaneamente è andato in running il figlio ed entrambi vengono terminati ed effettuato il return. Se non mettessimo lo sleep(1) avremmo il padre che termina prima del figlio e quindi nel terminale vedremo due esecuzioni distinte di processi e non solo una come nel caso in cui mettiamo sleep(1) in cui nella medesima istanza vengono eseguiti i due processi differenti. Per ottenere una corretta esecuzione dobbiamo inserire la funzione sleep, dunque. Notiamo cosa accade commentando lo sleep(1):



```
#include <stdio.h>
#include <unistd.h>

int main () {
    pid_t pid;
    printf("main program id is %d\n", (int) getpid());
    pid = fork();
    if (pid == 0){
        // fork is called in parent, so in child process pid is 0
        printf("child process\n");
        printf("C: child ID is %d\n", (int) getpid());
    }
    else {
        // parent called fork, so pid is the ID of the child
        printf("Parent process\n");
        printf("P: parent ID is %d\n", (int) getpid());
        printf("P: child ID is %d\n", (int) pid);
    }
    //sleep(1); // to properly exit from prompt
    return 0;
}
```

The terminal window shows the following output:

```
marco@marco-VirtualBox:~/Scrivania$ gcc -o fork00 fork00.c
marco@marco-VirtualBox:~/Scrivania$ ./fork00
main program id is 20990
Parent process
P: parent ID is 20990
P: child ID is 20991
marco@marco-VirtualBox:~/Scrivania$ Child process
C: chlld ID is 20991
```

Vedremo meglio dopo che sleep non va utilizzato perché interferisce con le politiche di scheduling, ma esiste un altro modo corretto per far sì che il processo padre termini aspettando la terminazione del figlio prima. Questo metodo è implementato mediante la funzione **wait()** che vedremo dopo.

Vediamo un altro [esempio](#) di esecuzione concorrente di processi padre e figlio mediante la chiamata fork:

The screenshot shows a terminal window titled "ubuntu1804 [Running]" with the date "dom 12:36". The window contains a text editor with the file "counters.c" open. The code is as follows:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h> // for random

int main () {
    printf("-beginning of program\n");

    int counter = 0;
    int N = 20;

    pid_t pid = fork();

    if(pid==0) {
        // child
        for(int i = 0; i < N; ++i) {
            printf("C process: counter=%d\t(%d)\n",++counter,(int) getpid());
            fflush(stdout); // clean the buffer (needed to write to file in the right order)
            sleep((double)rand()/(double)RAND_MAX); // random in [0,1]
        }
    } else if (pid > 0) {
        // parent
        for(int j = 0; j < N; ++j) {
            printf("P process: counter=%d\t(%d)\n",++counter,(int) getpid());
            fflush(stdout); // clean the buffer (needed to write to file in the right order)
            sleep((double)rand()/(double)RAND_MAX); // random in [0,1]
        }
    } else {
        // fork error
        printf("fork error\n");
        return(1);
    }

    printf("-end of program\n");
    sleep(1);

    return 0;
}
```

The terminal window has a toolbar with various icons for file operations like Open, Save, and Print.

Analizziamo riga per riga:

0. Abbiamo l'inizializzazione di un contatore a 0 e N=20.
1. Eseguiamo la fork creando il processo figlio.
2. Il processo child e quello parent eseguono lo stesso codice e inizieranno a incrementare di volta in volta il valore di i fino a 20. Subito dopo la print c'è la funzione fflush() utile a pulire il buffer.
3. Per distinguere l'esecuzione dei due processi specifico un valore di sleep casuale che varierà tra 0 e 1. Questi due cicli andranno ripetuti per 20 volte. La funzione dello sleep è semplicemente quella di rallentare l'esecuzione.

Vediamo cosa otteniamo in output:

```

marco@marco-VirtualBox:~/Scrivania$ ./counters
--beginning of program
P process: counter =1      (6850)
C process: counter =1      (6851)
P process: counter =2      (6850)
C process: counter =2      (6851)
P process: counter =3      (6850)
C process: counter =3      (6851)
P process: counter =4      (6850)
C process: counter =4      (6851)
P process: counter =5      (6850)
C process: counter =5      (6851)
P process: counter =6      (6850)
C process: counter =6      (6851)
P process: counter =7      (6850)
C process: counter =7      (6851)
P process: counter =8      (6850)
C process: counter =8      (6851)
P process: counter =9      (6850)
C process: counter =9      (6851)
P process: counter =10     (6850)
C process: counter =10     (6851)
P process: counter =11     (6850)
C process: counter =11     (6851)
P process: counter =12     (6850)
C process: counter =12     (6851)
P process: counter =13     (6850)
C process: counter =13     (6851) □

```



```

□ P process: counter =14      (6850)
C process: counter =14      (6851)
P process: counter =15      (6850)
C process: counter =15      (6851)
P process: counter =16      (6850)
C process: counter =16      (6851)
P process: counter =17      (6850)
C process: counter =17      (6851)
P process: counter =18      (6850)
C process: counter =18      (6851)
P process: counter =19      (6850)
C process: counter =19      (6851)
P process: counter =20      (6850)
C process: counter =20      (6851)
--end of program
--end of program
marco@marco-VirtualBox:~/Scrivania$

```

In questa prima esecuzione otteniamo una alternanza perfetta di processo padre e processo figlio nell'uso del processore. Alla fine dell'esecuzione otterremo la stringa "end of program" sia per padre sia per figlio. È chiaro che questo output è dovuto ad una coincidenza fortuita dato che i processi padre e figlio si alternano perfettamente. Rieseguendo il programma infatti otterremo un output diverso:

```

marco@marco-VirtualBox:~/Scrivania$ ./counters
--beginning of program
P process: counter =1      (4438)
P process: counter =2      (4438)
C process: counter =1      (4439)
C process: counter =2      (4439)
P process: counter =3      (4438)
P process: counter =4      (4438)
P process: counter =5      (4438)
P process: counter =6      (4438)
C process: counter =3      (4439)
C process: counter =4      (4439)
P process: counter =7      (4438)
P process: counter =8      (4438)
C process: counter =5      (4439)
C process: counter =6      (4439)
P process: counter =9      (4438)
C process: counter =7      (4439)
P process: counter =10     (4438)
P process: counter =11     (4438)
P process: counter =12     (4438)
P process: counter =13     (4438)
P process: counter =14     (4438)
C process: counter =8      (4439)
C process: counter =9      (4439)
P process: counter =15     (4438)
C process: counter =10     (4439)
P process: counter =16     (4438) □

```



```

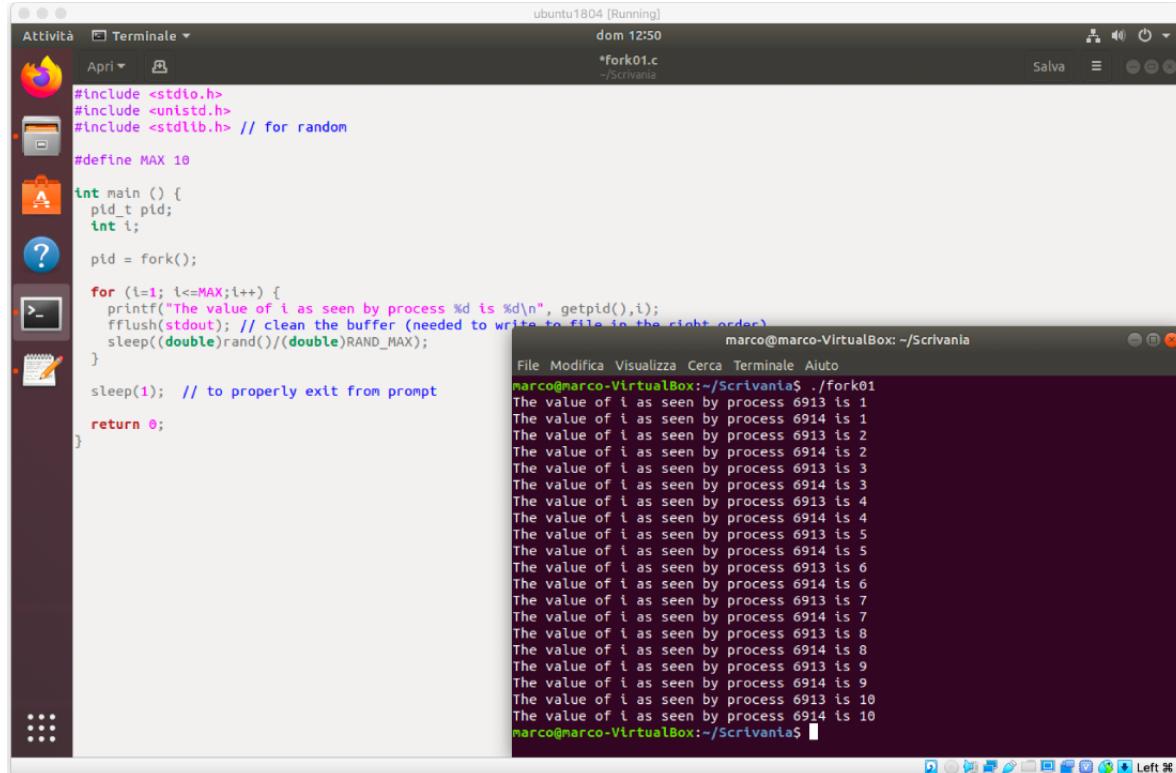
□ C process: counter =11      (4439)
P process: counter =17      (4438)
C process: counter =12      (4439)
P process: counter =18      (4438)
C process: counter =13      (4439)
P process: counter =19      (4438)
C process: counter =14      (4439)
P process: counter =20      (4438)
C process: counter =15      (4439)
--end of program
C process: counter =16      (4439)
C process: counter =17      (4439)
C process: counter =18      (4439)
C process: counter =19      (4439)
C process: counter =20      (4439)
--end of program
marco@marco-VirtualBox:~/Scrivania$

```

Questa esecuzione è invece totalmente randomica come ci aspettavamo inizialmente. Vediamo che i due processi terminano in momenti diversi, ce ne accorgiamo dalla locazione delle due "end of program" che sono distanziate a differenza di prima. Ma a cosa è dovuto questa alternanza? Sia allo sleep randomico sia alle politiche di scheduling che saranno cambiate rispetto a prima.

Per forzare un processo ad andare in esecuzione prima dell'altro dovremmo utilizzare tecniche diverse dallo sleep che interferisce col normale funzionamento dell'OS.

Vediamo un ulteriore esempio di fork:



```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h> // for random

#define MAX 10

int main () {
    pid_t pid;
    int i;

    pid = fork();

    for (i=1; i<=MAX;i++) {
        printf("The value of i as seen by process %d is %d\n", getpid(),i);
        fflush(stdout); // clean the buffer (needed to write to file in the right order)
        sleep((double)rand()/(double)RAND_MAX);
    }

    sleep(1); // to properly exit from prompt

    return 0;
}
```

The value of i as seen by process 6913 is 1  
The value of i as seen by process 6914 is 1  
The value of i as seen by process 6913 is 2  
The value of i as seen by process 6914 is 2  
The value of i as seen by process 6913 is 3  
The value of i as seen by process 6914 is 3  
The value of i as seen by process 6913 is 4  
The value of i as seen by process 6914 is 4  
The value of i as seen by process 6913 is 5  
The value of i as seen by process 6914 is 5  
The value of i as seen by process 6913 is 6  
The value of i as seen by process 6914 is 6  
The value of i as seen by process 6913 is 7  
The value of i as seen by process 6914 is 7  
The value of i as seen by process 6913 is 8  
The value of i as seen by process 6914 is 8  
The value of i as seen by process 6913 is 9  
The value of i as seen by process 6914 is 9  
The value of i as seen by process 6913 is 10  
The value of i as seen by process 6914 is 10

In questo caso eseguo fork ma non faccio più distinzione tra padre e figlio con l'if e l'else, per cui all'interno di questo for vado a stampare una stringa che stampa il valore della variabile i (distinta tra processi padre e figlio) e il valore del PID legato al processo eseguito in quel momento. Anche in questo esempio è una totale casualità che i due processi si alternino in questa maniera così precisa. Se rieseguiamo il programma altre volte noteremo output differenti e il flusso di esecuzione sarà diverso. Anche qui abbiamo usato sleep(1) per alternare i processi.

Notiamo che se non facessimo così il processo figlio diverrebbe un processo zombie. Un **processo zombie** (*process defunto*) è un processo che, nonostante abbia terminato la propria esecuzione, possiede ancora un PID ed un Process control block, necessario per permettere al proprio processo padre di leggerne il valore di uscita. Per identificare un processo zombie possiamo usare questo codice di esempio:

```

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid;
    pid = fork();

    if(pid > 0) {
        // parent
        printf("parent (%d) sleep for a while\n", getpid());
        sleep(10);
    } else {
        //child
        printf("child (%d) exit immediately\n", getpid());
        exit(0);
    }
}

return 0;

```

Se mando in esecuzione questo programma verrà eseguito il processo padre che dopo avere eseguito le sue istruzioni andrà in sleep. Nel frattempo il processo figlio termina ma lo stato che visualizziamo nel terminale è il puntatore lampeggiante che indica che il processo zombie è ancora in esecuzione. Questo processo terminerà dopo 10 secondi come specificato nel codice dal comando sleep(10).

Se nell'arco di questi 10 secondi usiamo il comando ps -e -o pid,ppid,stat,cmd su un'altra finestra possiamo vedere i processi in esecuzione:

```

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid;
    pid = fork();

    if(pid > 0) {
        // parent
        printf("parent (%d) sleep for a while\n", getpid());
        sleep(10);
    } else {
        //child
        printf("child (%d) exit immediately\n", getpid());
        exit(0);
    }
}

return 0;

```

Process ID	Status	Command
3002	Z	[xfsalloc]
3003	Z	[xfs_mru_cache]
3016	S	[jfsIO]
3019	S	[jfsCommit]
3020	S	[jfsSync]
3466	S	/usr/lib/gvfs/gvfsd-network --spawner :1.13 /org/gtk/gvfs/exec_
3488	S	/usr/lib/gvfs/gvfsd-smb-browse --spawner :1.13 /org/gtk/gvfs/ex
3501	S	/usr/lib/gvfs/gvfsd-dnssd --spawner :1.13 /org/gtk/gvfs/exec_sp
3909	I	[kworker/0:2]
6416	S	/usr/sbin/cupsd -l
6417	S	/usr/sbin/cups-browsed
6925	I	[kworker/u2:1]
6927	S	gedit /home/marco/Scrivania/zombie.c
6965	S	bash
7018	Z	./zombie
7019	Z	[zombie] <defunct>
7020	S	ps -e -o pid,ppid,stat,cmd
11100	S	/lib/systemd/systemd-udevd
16519	S	/usr/bin/whoopsie -f
21564	S	/lib/systemd/systemd-resolved
21567	S	/lib/systemd/systemd-journald
22779	I	[kworker/u2:0]
24014	S	/usr/bin/nautlius --gapplication-service

Quello che otteniamo in output è una lista di processi, alcuni di questi saranno etichettati con la seguente etichetta: [zombie] e la lettera Z, che indica un processo zombie ormai non più in esecuzione. I processi zombie verranno ereditati dal processo radice init. Allora sarà proprio init a gestire la terminazione dei processi zombie utilizzando la funzione wait().

## Funzione wait() e waitpid()

Come funziona **wait()**? Se utilizzata senza una fork prima non serve a nulla se non alla creazione di processi zombie, se invece la utilizziamo con fork e, un figlio termina mentre il suo padre ha fatto una chiamata wait, allora alla terminazione del figlio, l'informazione della terminazione sarà inviata al padre mediante la funzione wait. In particolare **wait** consente al processo padre di monitorare lo stato del processo figlio.

La funzione wait mette il padre in ascolto di notifiche da parte del processo figlio. Wait ha come input un puntatore allo stato del processo figlio e come output restituirà una variabile pid\_t, quindi la sintassi sarà la seguente:

```
pid_t wait(int *status)
```

Un cambiamento di stato può essere dato da alcuni eventi:

- La terminazione di un processo.
- Lo stop di un processo.
- Un resume di un processo.

È importante usare la funzione wait() per evitare che alla terminazione dei processi figli la memoria utilizzata da questi ultimi non venga tenuta occupata ma venga invece liberata. Alla terminazione del figlio, il padre libererà le risorse usate dal figlio. Se questa operazione non viene effettuata dal padre il figlio entra in stato di zombie.

I processi zombie se non vengono cancellati dal padre tengono occupate risorse del sistema, è dunque necessario rimuovere sempre i processi zombie. E questo viene fatto da init che eredita tutti i processi zombie e si occupa di rimuoverli, nel caso in cui il padre non sia più presente per farlo.

Come gestiamo la funzione wait? Lo vediamo con un esempio:

```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main () {
    int child_status;

    pid_t pid;
    printf("main program id is %d\n", (int) getpid());

    pid = fork();

    if (pid == 0){
        // fork is called in parent, so in child process pid is 0
        printf("Child process\n");
        printf("C: child ID is %d\n", (int) getpid());
    }
    else {
        // parent called fork, so pid is the ID of the child
        printf("Parent process\n");
        printf("P: parent ID is %d\n", (int) getpid());
        printf("P: child ID is %d\n", (int) pid);
    }

    wait(&child_status); wait() store status information in the int to which it points

    return 0; child_status is inspected with some system macros
}
```

## wait()

Dato che wait è una chiamata di sistema non sappiamo come venga implementata questa funzione, quello che sappiamo certamente è che la funzione si occupa di memorizzare informazione relative allo stato del processo puntato dal puntatore (parametro della funzione). In particolare nella penultima riga abbiamo usato wait come funzione bloccante, ovvero in maniera tale da costringere il padre ad aspettare la terminazione del figlio prima di terminare a sua volta.

Sapendo ciò possiamo modificare uno degli esempi precedenti nella seguente maniera:

The screenshot shows a desktop environment with a terminal window and a code editor window. The terminal window is titled 'ubuntu1804 [Running]' and shows the command 'fork\_wait.c' being run. The output is as follows:

```
File Modifica Visualizza Cerca Terminale Aiuto
Marco@marco-VirtualBox:~/Scrivania$ gcc -o fork_wait fork_wait.c
Marco@marco-VirtualBox:~/Scrivania$ ./fork_wait
main program id is 7316
Parent process
P: parent ID is 7316
P: child ID is 7317
Child process
C: child ID is 7317
Marco@marco-VirtualBox:~/Scrivania$
```

The code editor window shows the C code for the 'fork\_wait' program, which includes the `#include <sys/types.h>` header instead of `<unistd.h>`.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h> // <== WAIT ==>

int main () {
    printf("main program id is %d\n", (int) getpid());
    int child_status; // <== WAIT ==>
    pid_t pid;
    pid = fork();

    if (pid == 0){
        printf("Child process\n");
        printf("C: child ID is %d\n", (int) getpid());
    }
    else {
        printf("Parent process\n");
        printf("P: parent ID is %d\n", (int) getpid());
        printf("P: child ID is %d\n", (int) pid);
    }

    wait(&child_status); // <== WAIT ==>

    return 0;
}
```

Stavolta l'output è corretto (ordinato) per via della funzione wait, e non di quella sleep come avveniva negli esempi precedenti.

Esiste una variante di wait, ovvero **waitpid()**. **waitpid()** fa la sospensione del processo che l'ha chiamata finché il processo con PID pari a quello passato alla funzione non cambia il suo stato. Si definisce così:

```
pid_t waitpid(pid_t, int *status, int options)
```

Il valore di pid può essere:

- **minore di -1**: stiamo aspettando la terminazione di uno dei figli del processo padre, facenti parte di un gruppo di processi i quali PID sono uguali al valore assoluto di pid\_t (parametro).
- **-1**: aspettiamo la terminazione di un qualsiasi figlio.
- **0**: stiamo aspettando la terminazione di uno qualsiasi dei figli facenti parte del gruppo del processo padre.
- **maggiore di 0**: aspetta la terminazione del processo con PID=pid\_t (parametro).

In altri esempi faremo delle fork multiple con relativa gestione della terminazione dei figli utilizzando wait di vario genere.

# Lezione 9

## Funzione exec()

La funzione exec() permette di sostituire il contenuto di un processo con quello di un altro. Exec è quindi una chiamata di sistema che fa riferimento ad un altro processo, nel momento in cui invoco exec, il contenuto del processo chiamante viene sostituito dalle istruzioni del processo chiamato.

Esistono vari modi per invocare la funzione exec, specifichiamo infatti che exec è una famiglia di funzioni, vediamole:

- **execvp, execpl**: accettano in input dei processi specificandone il nome e cercando nel path corrente il processo con quel nome. Accettano anche un path assoluto relativo ad uno specifico processo.
- **execv, execvp, execve**: accettano in input una lista di puntatori a stringhe, che si conclude con un elemento NULL.
- **execl, execlp, execle**: accettano come input una lista di variabili varargs definibili nel linguaggio C.
- **execve, execle**: accetta come input aggiuntivo delle liste di variabili di sistema.

Noi nello specifico utilizzeremo le execl(), ovvero quelle funzioni exec che accettano in input argomenti quali le varargs del linguaggio C.

Vediamo come usare assieme fork ed exec. Sappiamo che fork permette la clonazione di processi. Ovviamente sappiamo che le istruzioni di processo padre e figlio clonati sono uguali, ma a volte può essere utile che il figlio esegua del codice già esistente (non definito dal padre). Questa operazione si fa con una chiamata ad exec dentro il processo figlio.

Quindi grazie ad exec il processo padre può continuare la sua esecuzione mentre quello figlio eseguirà altro codice di un altro processo.

Vediamo un esempio:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main () {
    int child_status;
    pid_t pid;
    pid = fork();

    if (pid == 0){
        printf("Child process: ls -l\n");
        //execl(const char *path, const char *arg0, ...);
        execl("/bin/ls", "ls", "-l", (char *)0); // the li
    }
    else {
        // parent called fork, so pid is the ID of the child
        printf("Parent process: do nothing\n");
    }

    wait(&child_status);

    return 0;
}
```

File Modifica Visualizza Cerca Terminale Aiuto  
marco@marco-VirtualBox:~/Scrivania/script1\$ ./fork\_exec  
Parent process: do nothing  
Child process: ls -l  
totale 28  
-rwxr-xr-x 1 marco marco 8480 mar 22 15:42 fork\_exec  
-rw-r--r-- 1 Marco Marco 22 mag 5 2019 t1.doc  
-rw-r--r-- 1 marco marco 22 mag 5 2019 t1.txt  
-rw-r--r-- 1 marco marco 11 mag 5 2019 t2.txt  
-rw-r--r-- 1 marco marco 10 mag 5 2019 t3.txt  
marco@marco-VirtualBox:~/Scrivania/script1\$

Il blocco di istruzioni per il processo figlio vede l'utilizzo della funzione execl() che riceve come parametro: una stringa che rappresenta il path relativo al processo che vogliamo invocare, in questo specifico caso è il path di ls, successivamente troviamo altre stringhe che determinano come avviene l'invocazione del programma e una serie di parametri aggiuntivi che sono le opzioni. In definitiva otterremo l'invocazione del processo ls -l. Per indicare che i parametri sono finiti si mette il terminatore del comando.

Eseguendo questo programma otterremo come risultato quello che vediamo nel terminale, cioè l'output del comando ls -l.

Ritornando a questo esempio fatto nella precedente lezione, cerchiamo di capire quale è la differenza tra funzione system ed exec, visto che a prima vista potrebbero sembrare uguali:

```
#include <stdlib.h>

int main ()
{
    int return_value;
    return_value = system ("ls -l /");
    return return_value;
}
```

System viene eseguito e ritorna un valore, cioè fa una chiamata di un processo che effettua un determinato comando di sistema che viene ritornato come valore intero. Ma per fare questo, la funzione system crea un processo con la funzione fork() che a sua volta eseguirà il comando nell'argomento di system con exec(). Quindi system() sottintende a tutti gli effetti l'utilizzo della funzione fork.

Viceversa, quando facciamo la chiamata ad exec stiamo sostituendo il codice del processo figlio e non abbiamo valore di ritorno ma semplicemente viene stampato su terminale il risultato di exec.

### Funzione pipe()

Vediamo ora come i processi possono scambiarsi messaggi tramite l'IPC. Ricordiamo che i processi si scambiano messaggi per potersi sincronizzare o per effettuare delle operazioni congiunte per raggiungere un obiettivo comune. Il primo modo in cui possiamo fare comunicare processi è utilizzando la funzione **pipe()**.

La **pipe** è un device seriale (FIFO) dove i dati sono letti nello stesso ordine in cui sono stati scritti. Per essere utilizzata si fa uso dei **file descriptor** visti nella lezione precedente. Tutti i processi di unix hanno 3 file descriptor di standard:

- **FD0** standard input.
- **FD1** standard output.
- **FD2** standard error.

La pipe consente al processo padre di comunicare al processo figlio. Per fare questo la pipe utilizza due file descriptor, che servono a definire l'**estremità di lettura della pipe** (FD0) e l'**estremità di scrittura della pipe** (FD1). Vediamo un [esempio](#):

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
    int fd[2];
    char buffer[5];

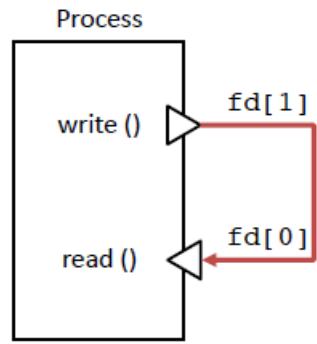
    if(pipe(pipefds) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    char *msg = "ciao\0";
    //int write( int handle, void *buffer, int nbytes );
    write(fd[1], msg, 5);

    //int read( int handle, void *buffer, int nbytes );
    read(fd[0], buffer, 5);

    return EXIT_SUCCESS;
}

```



La pipe consente di fare una comunicazione dalla estremità di scrittura verso l'estremità di scrittura indicati mediante i FD.

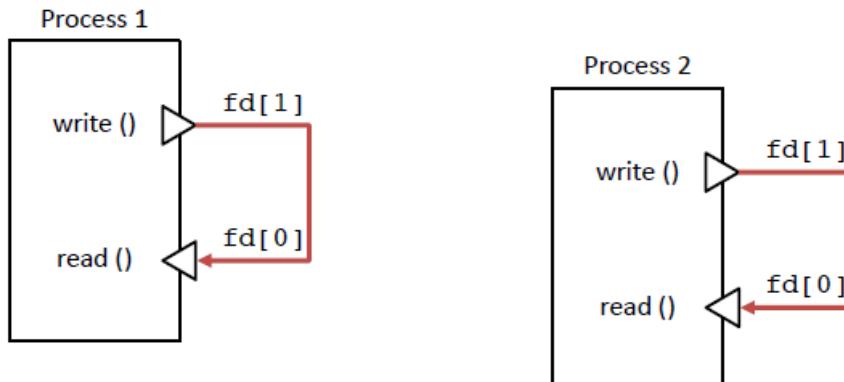
NB: nell'esempio all'interno dell'if c'è la funzione pipe(pipefds), è sbagliato. In realtà va corretto con pipe(fd).

Vediamo come accade ciò analizzando il codice:

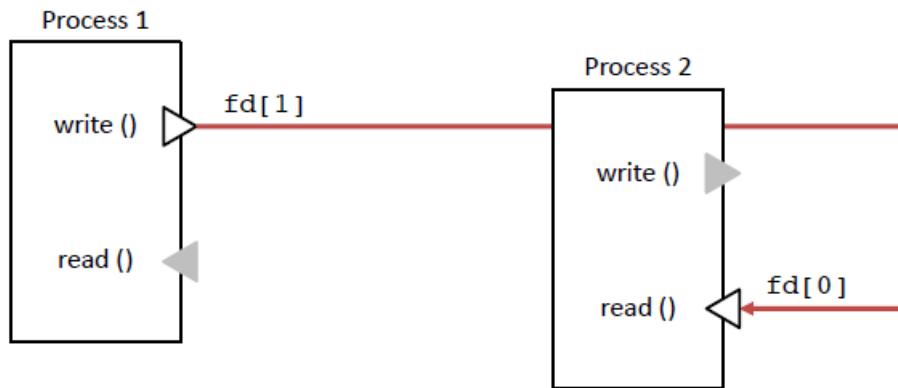
0. Creiamo due variabili fd e buffer rispettivamente per i file descriptor e per il buffer che utilizzeremo per leggere il messaggio trasmesso.
1. Creo una pipe mediante la funzione pipe(fd), se dovesse ritornare -1 c'è stato errore.
2. Utilizzo la funzione write che ha come parametri un FD per l'output (fd[1]), un messaggio da trasmettere, e la dimensione in byte del messaggio. In questo specifico esempio andiamo a scrivere "ciao".
3. In modo simile avrò la funzione read che avrà come parametri il FD relativo alla lettura (fd[0]), il buffer dove salvare messaggi e la dimensione del messaggio.

In questo esempio appena visto l'utilizzo della pipe è limitato ad un solo processo, quando noi in realtà vogliamo fare comunicare processi differenti.

Per fare comunicare due processi utilizziamo pipe assieme al comando fork, creiamo una pipe così che quest'ultima sia condivisa sia da padre che da figlio. Per fare ciò basta invocare la funzione pipe prima della invocazione della funzione fork. Così la pipe sarà in comune tra i due processi che comunicheranno tramite essa:



Come sappiamo infatti la funzione fork permette al processo figlio di ereditare i file descriptor del padre, ecco quindi la situazione in cui ci troveremo dopo aver utilizzato pipe e solo dopo fork:



Per far sì la pipe funzioni correttamente, come nella figura sopra, si deve chiudere l'estremità di lettura della pipe del primo processo (quello che vuole trasmettere un messaggio, quindi scrivere) e allo stesso tempo chiudere l'estremità di scrittura della pipe del secondo processo (quello che deve ricevere il messaggio, quindi leggere). Questo nello specifico provvede alla comunicazione dal processo 1 verso il processo 2, se volessimo invece fare scrivere il processo 2 e far leggere il processo 1 dovremmo chiudere le estremità delle pipe al contrario di come vediamo nel disegno, ovvero: Process2 write() aperto e Process1 read() aperto.

Ci rendiamo conto che la pipe non è bidirezionale.

%ls | less è un comando che implementa la pipe a livello della shell. In particolare il carattere | indica la pipe nel terminale. ls serve a visualizzare il contenuto di una cartella mentre less a visualizzare il contenuto di un file. Quello che fa quindi questo comando è di eseguire ls su una cartella, tramite la pipe trasferiamo l'output di ls a less causando così la lettura di tutti i file di una cartella. La comunicazione viene gestita quindi a livello di sistema con il carattere | che permette la comunicazione tra ls e less, nello specifico less legge dalla pipe l'output scritto precedentemente da ls nella pipe stessa.

Per capire meglio vediamo un esempio di codice:

```

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>

int main() {
    int fd[2];
    pid_t pid;

    char string[] = "Hello\n";
    char readBuffer[80];

    pipe(fd);
    pid = fork();

    if (pid == 0) {
        //-- child
        //close pipe input side
        printf("C: closing pipe input...\n");
        close(fd[0]);
        //send string through the output side
        printf("C: sending string %s",string);
        //int write( int handle, void *buffer, int nbytes );
        write(fd[1],string,(strlen(string)+1));
        exit(0);
    }
    else {
        //-- parent
        //close pipe output side
        printf("P: closing pipe output...\n");
        close(fd[1]);
        //read string from the pipe input
        printf("P: receiving...\n");
        //int read( int handle, void *buffer, int nbytes );
        read(fd[0],readBuffer,sizeof(readBuffer));
        printf("P: received string %s",readBuffer);
    }
    return 0;
}

```

0. Creo l'array di interi per i file descriptor FD.
1. Creo la variabile pid di tipo pid\_t per l'output di fork.

2. Creo un array di caratteri per il messaggio "Hello" che vogliamo trasmettere.
3. Creo un buffer per la lettura.
4. Creo la pipe passando i file descriptor (array fd) da utilizzare come parametro.
5. Effettuo la fork. Da questo momento in poi quello che era stato fatto prima nel processo padre viene duplicato per il processo figlio. Viene quindi duplicata anche la pipe con relativi file descriptor.
6. In questo esempio specifico il figlio scrive, di conseguenza chiude l'estremità di lettura dato che appunto deve solo scrivere. Per effettuare la chiusura utilizza il comando close(fd[0]). Successivamente effettua la scrittura mediante il comando write() al quale passiamo come parametri il FD per la scrittura (fd[1]), il messaggio da scrivere e la lunghezza in byte della scrittura.
7. Nel processo padre dobbiamo implementare la lettura quindi chiudiamo l'estremità di scrittura della pipe (fd[1]). Facciamo poi una chiamata per leggere il messaggio con read() e salviamo il messaggio che leggiamo nel buffer di lettura.
8. **NOTA IMPORTANTE**: negli esempi visti nella precedente lezione abbiamo visto come processo padre e processo figlio vengano eseguiti in ordine casuale, quindi possiamo chiederci in questo caso, se prima che il figlio scriva nella pipe, il padre abbia già letto (in quel caso leggerebbe un buffer vuoto). Ciò non è un problema, infatti la pipe garantisce automaticamente che il primo processo ad essere eseguito sia quello che scrive, e solo dopo verrà eseguito quello che legge.

L'output del codice precedente sarà quindi il seguente:

```
marco@marco-VirtualBox:~/Scrivania$ ./pipe00
P: closing pipe output...
P: receiving...
C: closing pipe input...
C: sending string Hello
P: received string Hello
```

### Signals e funzione signal()

L'altra modalità di comunicazione tra i processi è quella che prevede l'utilizzo dei **signals**. Ricordiamo che i signals sono interrupt di tipo software scambiati tra processi. Esiste un comando di shell chiamato **kill** che implementa i signals. Tra le opzioni previste per il comando c'è il tipo di segnale che si vuole inviare seguito dal PID al quale lo si vuole inviare. La sintassi è la seguente:

```
kill [ -signal | -s signal ] pid ...
kill -l [ signal ]
```

Ovviamente si potrà implementare in linguaggio C mediante l'uso della **libreria signal.h** e con la funzione **kill()**. Nello specifico implemteremo il signal nella seguente maniera:

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

Quello che vedremo nello specifico è la gestione dei signal, in quanto non tutti i signal possono essersi gestiti. Vediamo quindi la lista di signal che esistono tramite l'utilizzo nella shell del comando **kill -l**:

1) SIGHUP	2) <b>SIGINT</b>	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGEMT	8) SIGFPE
9) <b>SIGKILL</b>	10) SIGBUS	11) SIGSEGV	12) SIGSYS
13) SIGPIPE	14) SIGALRM	15) <b>SIGTERM</b>	16) SIGURG
17) <b>SIGSTOP</b>	18) SIGTSTP	19) <b>SIGCONT</b>	20) SIGCHLD
21) SIGTTIN	22) SIGTTOU	23) SIGIO	24) SIGXCPU
25) SIGXFSZ	26) SIGVTALRM	27) SIGPROF	28) SIGWINCH
29) SIGINFO	30) SIGUSR1	31) SIGUSR2	

Vediamone alcuni in maniera più specifica:

- **SIGINT**: interrompe qualsiasi input da tastiera (*leggasi: ctrl+c*).
- **SIGSTOP**: signal che mette in pausa un processo, che eventualmente può essere ripreso. È un signal che non si può ignorare.
- **SIGCONT**: serve a riprendere l'esecuzione di un processo che è stato messo in pausa con SIGSTOP.
- **SIGTERM**: signal che serve a chiedere la terminazione di un processo. Il seguente signal può essere gestito o ignorato.
- **SIGKILL**: ha lo stesso scopo del SIGTERM ma non può essere ignorato né gestito. Il processo che riceve questo signal viene terminato all'istante senza la possibilità di eseguire qualsiasi operazioni per tutelare la sua esecuzione.

Quello che vediamo ora è un esempio di gestione dei segnali:

```

#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>

void sigint();
void sigterm();
void sigcont();

int main() {
    printf("-- test is running\n");
    while(1) {
        // signal has two params:
        // - sig to be handled
        // - handler function
        signal(SIGINT,sigint);
        signal(SIGTERM,sigterm);
        signal(SIGCONT,sigcont);
    }
    return 0;
}

- Open the shell
- Run this code, with and without exit(0)
- While running, open a different shell
• Find the [pid] using: ps -e -o pid,cmd
• Send the signals
    o kill -SIGINT [pid]
    o kill -SIGTERM [pid]
    o kill -SIGSTOP [pid]
    o kill -SIGCONT [pid]
    o kill -SIGKILL [pid]

- REPEAT WITH DIFFERENT SIGNALS

void sigint() {
    printf("--- received SIGINT\n");
    exit(0); // provare con e senza
}

void sigterm() {
    printf("--- received SIGTERM\n");
    exit(0); // provare con e senza
}

void sigcont(){
    printf("--- received SIGCONT\n");
}

```

All'interno del while abbiamo la chiamata della funzione signal, che come parametro riceve il tipo di signal che si vuole gestire seguito dalla funzione che si occuperà della gestione. Quindi:

```
signal(SIG_DA_GESTIRE, funzionePerGestioneDelSig)
```

# Lezione 10

---

## Thread

Via via che gli OS si sono evoluti è nata la necessità di potere eseguire più programmi in maniera concorrente. Con la nascita dei linguaggi di alto livello nacque anche la possibilità per i programmatori di implementare sottotask da eseguire in maniera concorrente.

Si iniziò a parlare di multithreading nel momento in cui per noi è stato possibile scrivere applicazioni sfruttando diversi flussi (thread) di esecuzione all'interno di uno stesso programma, ogni thread rappresenta un pezzo di codice che si può eseguire in maniera indipendente dagli altri thread. In particolare il threading ci permette di implementare alcune funzionalità concorrenti da eseguire in parallelo con altri thread.

Non tutti i linguaggi implementano il multithreading nativamente come accade col Java.

I **Thread** possono essere intesi in modo simili ai processi, in quanto ne condividono molte caratteristiche, sono però decisamente più leggeri nel loro complesso e per questo vengono definiti dei **Lightweight Process**, ovvero dei **processi leggeri**. I thread non esistono però come entità indipendenti, ma sono parti interne al processo, delle vere e proprie porzioni di codice, e nello specifico vengono schedulati singolarmente su un processore. Il loro obiettivo è quello di eseguire set di istruzioni in maniera indipendente dagli altri processi e thread così da implementare il parallelismo, per questo motivo un thread può essere visto come una sottoparte di un processo.

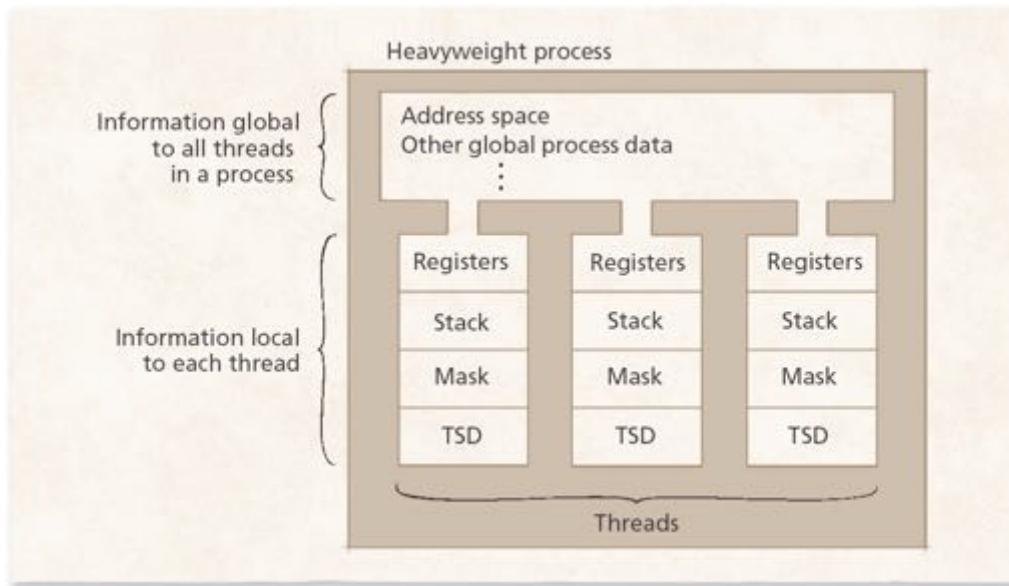
I processi che abbiamo visto fino ad adesso sono visti dal sistema come se fossero un singolo ed unico thread (**processi single-thread**), questo perché il sistema operativo autonomamente non sa analizzare un programma per capire se è possibile eseguire più parti del programma stesso parallelamente. Questo lavoro deve essere fatto esplicitamente dal programmatore che dichiara quali parti del codice possono essere gestite parallelamente. Creare un thread inoltre non richiede che l'OS debba inizializzare risorse per quel thread, mentre la creazione di un altro processo (per esempio con fork), prevede l'allocazione di una zona di memoria nuova per il processo appena creato. Questo perché il thread eredita la memoria del processo da cui è stato creato, ciò comporta un minore sforzo, quindi un minore overhead per la creazione e terminazione di thread, rispetto al corrispettivo per i processi. Anche per questo motivo si dice che i thread sono dei processi leggeri.

Ad esempio lo spazio di indirizzamento relativo al processo viene ereditato dai thread di quel processo.

Dato che il thread deve avere accesso al processore e deve essere schedulato il contenuto dei registri, lo stack e tutte le informazioni necessarie per l'esecuzione devono essere memorizzate anche per i thread, così come avviene per i processi, che nel momento in cui saranno in esecuzione avranno dunque un loro blocco di informazioni da memorizzare e portarsi dietro.

In un sistema con più processori vediamo il massimo vantaggio dei thread dato proprio dal parallelismo, infatti i thread vengono eseguiti ognuno su un processore diverso e vengono gestiti in parallelo.

Graficamente possiamo immaginare quanto detto tramite questa figura:



Vediamo come il processo venga diviso in più thread ognuno con le sue variabili e il suo stack.

I thread sono nati cronologicamente dopo i processi per cui si è mantenuta una distinzione tra thread nativi dell'OS e thread a livello utente. Per esempio in windows abbiamo thread gestiti nativamente dall'OS, mentre in linux non abbiamo una implementazione nativa dei thread ma necessitiamo dell'utilizzo di librerie che vengono gestite chiaramente a livello utente. Il vantaggio di quest'ultima implementazione è dato dalla portabilità dovuta alle librerie stesse, ma questo ha anche dei svantaggi rispetto alla soluzione nativa che è più performante.

### A cosa ci servono quindi i thread?

I processi sono stati tipicamente immaginati per potere eseguire un singolo flusso di istruzioni su un processore, tuttavia negli anni i processi si sono evoluti assieme ai sistemi operativi e nello specifico è nata la necessità da parte delle applicazioni di potere gestire parti di codice diverse in maniera indipendente l'una dall'altra. Questo ha un impatto notevole sul sistema perché se ci sono parti di codice indipendenti, con i thread posso svolgerli in parallelo, migliorando così le performance del sistema.

Ma al di là della parallelizzazione del programma, ci basti pensare che ci sono alcune parti del programma che si occupano delle operazioni di IO, mentre altre che fanno altre operazioni. Mediante l'uso dei thread possiamo fare in modo che i thread in stato di ready vengano eseguiti mentre i thread relativi alle operazioni di IO permangono in stato di blocked senza bloccare però necessariamente l'esecuzione di tutto il processo come avviene con i processi single-thread. Un altro vantaggio dato dai thread è dato dalla cooperazione che è possibile implementare mediante lo spazio di memoria condiviso (e non più tramite le pipe come avevamo visto) dove vengono salvate variabili condivise leggibili da più thread appartenenti allo stesso processo. Questo ha molteplici vantaggi ma comporta dei problemi che possono nascere dall'accesso simultaneo di più thread alla variabile condivisa.

Un ultimo vantaggio dato dai thread è legato ai web server, che come sappiamo ricevono molte richieste di visualizzazione da più client, ognuna delle quali deve essere servita. Comunemente i web server utilizzano i thread per gestire separatamente le richieste fatte dai client. Ogni thread si occupa di gestire una richiesta di un singolo client, e mentre i vari thread rimangono al lavoro per soddisfare le richieste dei client, il processo principale rimane in ascolto di altre richieste da mettere in coda.

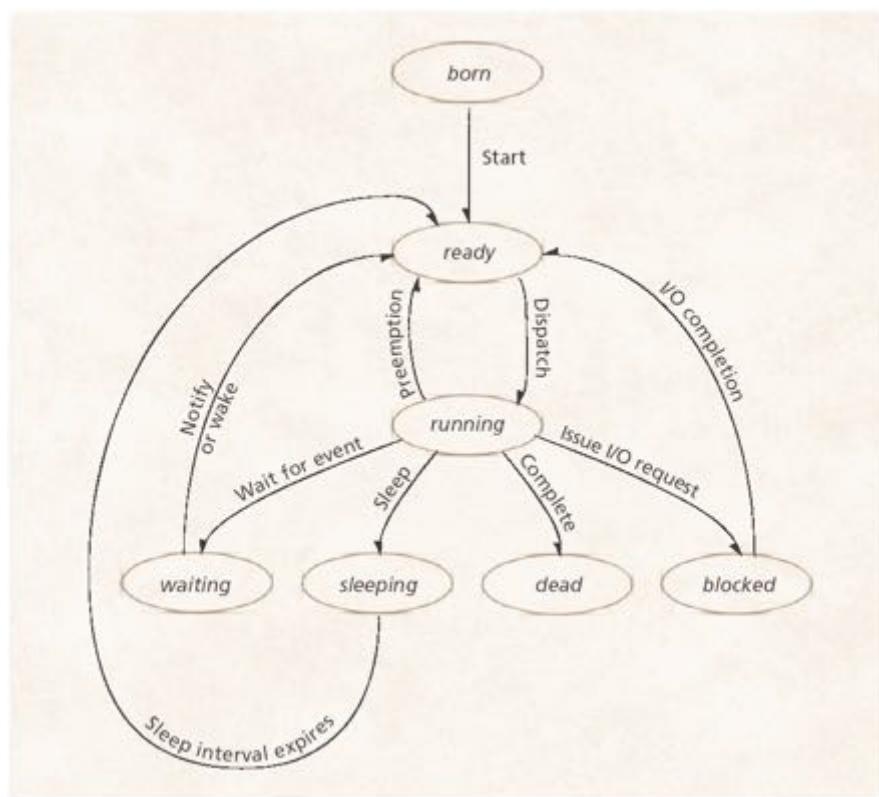
Vediamo ora un esempio pratico che ci fa capire l'utilità dei thread: prendiamo in esame un word processor. Nel momento in cui andiamo a scrivere, ogni tasto cliccato nella tastiera corrisponde ad un interrupt (signal) mandato al processore, in particolare quando si riceve questo interrupt il sistema va a memorizzare il carattere corrispondente in un buffer, a svolgere questa operazione è un particolare thread. Contemporaneamente però potrebbe succedere che scrivendo una parola male, quest'ultima venga evidenziata, questo accade perché parallelamente c'è un altro thread che legge i caratteri che ho scritto nel buffer ed evidenzia le parole scritte grammaticalmente male. allo stesso modo funzionano i salvataggi automatici gestiti tramite un ulteriore thread a cui è associato un timer, che una volta scaduto, si occupa di salvare le informazioni da noi scritte. Ogni thread si occupa quindi di un compito specifico. Ovviamente nella realtà ci sono molti più thread di quelli descritti, ma questo esempio è semplicemente per avere un'idea della potenza di questo strumento.

### Stati dei thread

Proprio perché i thread non sono tipicamente nativi dei sistemi operativi, il primo esempio di diagramma di stato che vediamo relativo agli stati dei thread è ispirato alla gestione **multithreading** in java.

Nel momento in cui viene creato, il thread va in stato di **born**, successivamente in stato di **ready** (o runnable) che è lo stato precedente a quello di **running** (esecuzione), dopo si entra nello stato di completamento (o **dead**). Troviamo poi lo stato di **blocked** (analogo a quello dei processi), di **waiting** (per l'attesa di una notifica tramite signal) e infine lo stato di **sleeping** che fa riferimento ad una transizione per un certo lasso di tempo in uno stato di inattività.

Vediamo il diagramma di stato associato:



**Wait** e **notify** sono le due operazioni che ci permettono di mettere in pausa le operazioni di un thread e successivamente riprendere la sua esecuzione. Gli eventi sono molto simili a quelli già visti per i processi.

I thread svolgono quindi una serie di operazioni:

- **Create**: utilizziamo una libreria dei thread per inizializzare le strutture specifiche di un thread e per la sua creazione, ricordando sempre che lo spazio di indirizzamento del thread

è già dato dalla memoria del processo. Questa creazione è più veloce della creazione di un processo, dato che le uniche informazioni che associamo ad un thread sono il contenuto dei registri, un ID del thread e il program counter.

- **Exit:** la terminazione elimina tutte le informazioni gestite e create nella nascita del thread.
- **Suspend.**
- **Resume.**
- **Sleep.**
- **Wake.**

Proprio per il fatto che i thread sono interni al processo alcune operazioni sono gestite diversamente rispetto a come sono svolte a livello del processo. Nel caso della terminazione, per il processo eliminiamo tutto il PCB e il processo stesso, mentre nel caso del thread andiamo semplicemente ad eliminare un thread e le poche informazioni che salva in memoria senza toccare però il processo generale o gli altri thread.

Un'altra operazione che viene implementata mediante i thread è il **join**, che serve per consentire ad un thread di aspettare la terminazione di un altro thread prima di terminare, questo serve a supportare la cooperazione di più thread e ad evitare che dei processi terminino prima della terminazione dei loro stessi thread. Qualora ciò non dovesse accadere i thread diventerebbero zombie.

### Implementazione dei thread

I diversi sistemi operativi trattano i thread in modo diverso e tipicamente abbiamo 3 modelli di implementazione:

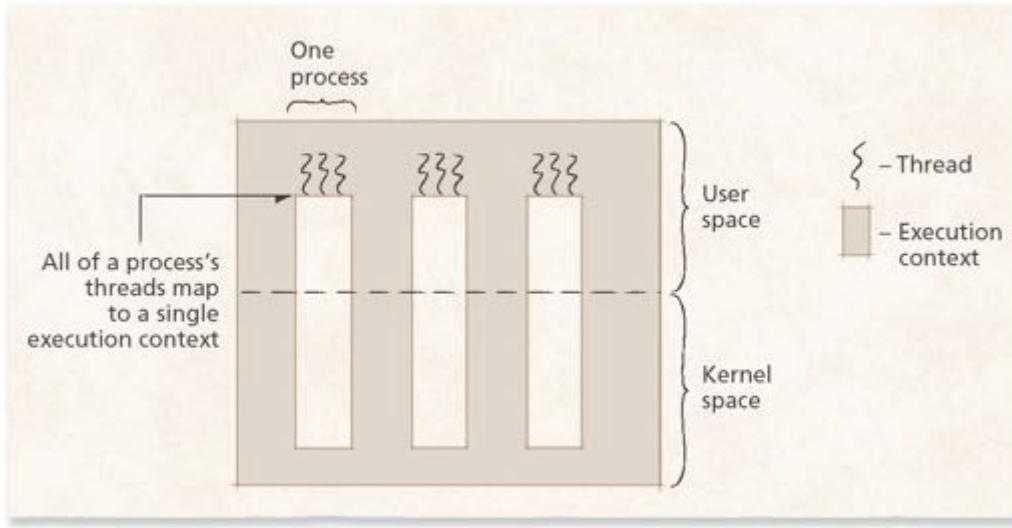
- **Implementazione a livello utente (User-Level)**
- **Implementazione a livello kernel (Kernel-Level)**
- **Implementazione ibrida**

Vediamo tutte queste implementazioni nel dettaglio

#### Implementazione User-Level

È la più semplice e antica implementazione dei thread. In questo caso la gestione dei thread viene demandata ai processi che utilizzavano i thread al loro interno e non al sistema operativo. A livello di sistema operativo si vede quindi il singolo processo. Questo è dovuto al fatto che quando è nata questa implementazione la maggior parte dei processi era single-thread. Questi thread prendono il nome di **user-level thread** perché rimangono invisibili all'OS, l'OS non sa dell'esistenza dei thread, e qualora esistano verrebbero gestiti dai processi stessi. Di contro questa implementazione implica la creazione di thread tramite librerie di alto livello che non permettono ai thread di effettuare operazioni di basso livello, inoltre il dispatch è gestito su una singola unità (processo) piuttosto che su più unità.

Graficamente:



Ogni colonna rappresenta un processo, ciascuno gestisce un insieme di thread ma ciò che vedo a livello di kernel è semplicemente il contesto di esecuzione. La presenza dei thread si nota solo a livello di utente. Per questo motivo l'implementazione a livello utente di questi thread prende il nome di **many-to-one**, poiché i thread vengono visti come una singola unità.

Il vantaggio di questa implementazione è che posso andare a gestire thread di questo tipo anche se l'OS non li supporta nativamente, lo svantaggio è che proprio per la non conoscenza dei thread da parte dell'OS, le performance di un sistema multiprocessore diminuiscono notevolmente in quanto questa soluzione non permette il parallelismo dei thread. Altro svantaggio è dato dal fatto che se un thread si blocca, questo comporterebbe il blocco totale del processo in quanto entità singola.

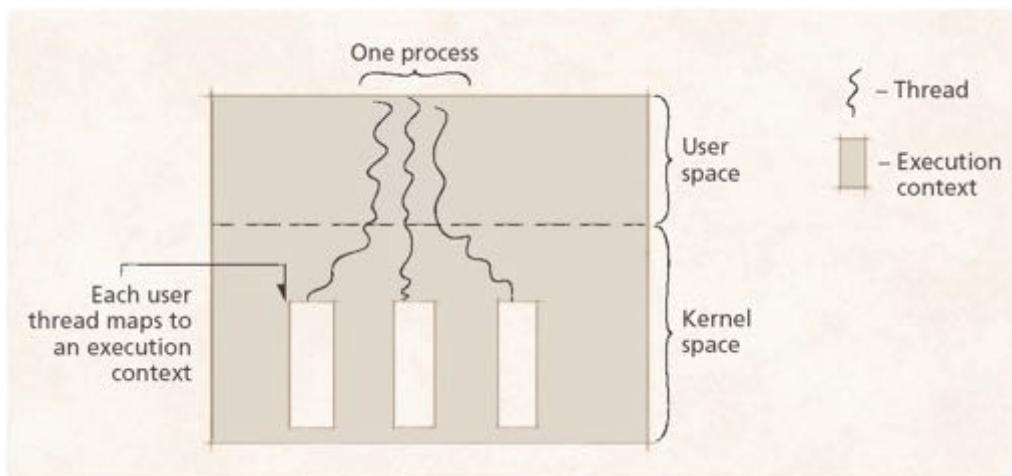
### Implementazione Kernel-Level

I **kernel-level thread** sono gestiti singolarmente, ciascun thread definito a livello utente viene mappato in un thread a livello kernel che l'OS gestisce direttamente come se fosse un processo, ricordando che però rispetto ai processi veri e propri, i thread condividono lo spazio di indirizzamento dei processi.

I thread in questo caso, essendo a livello kernel, possono usare le system call e quindi le chiamate di basso livello, a differenza di come avviene per i thread user-level.

Dato che in questa implementazione ogni thread a livello utente viene mappato in un singolo contesto di esecuzione a livello dell'OS, allora questa implementazione prende il nome di **one-to-one**.

Graficamente:



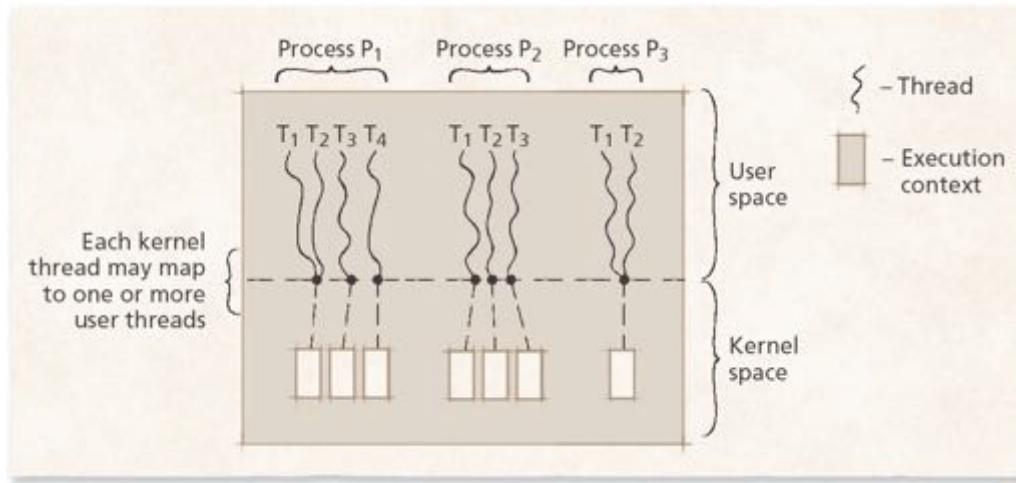
Nello user-space c'è un processo singolo che viene diviso in 3 thread che a livello kernel viene mappato in 3 thread autonomi ognuno col suo contesto di esecuzione.

I vantaggi di questa implementazione è dovuto alla scalabilità elevata, dato che ogni OS può gestire ogni thread ed inoltre il blocco di un thread non blocca l'esecuzione di un intero processo. Lo svantaggio è dato dal fatto che lo scheduling sarà fatto su molte più entità dato che i thread sono in numero molto maggiore rispetto ai processi, implicando una maggiore difficoltà nel gestire liste, priorità e scheduling in generale.

### Approccio ibrido

Si possono combinare i due approcci precedenti andando a fare in modo che più thread di livello utente siano mappati in un insieme di thread a livello kernel. Questo è un approccio è detto **many-to-many**: un processo che ha più thread a livello utente ( $m$ ) mappa questo insieme di thread in un altro insieme di thread a livello kernel ( $n$ ); questi due insiemi hanno un numero diverso di thread però ( $m$  ed  $n$ ), in particolare, l'insieme di thread a livello kernel sarà più piccolo dell'insieme di thread a livello utente relativi allo stesso processo.

Con un'immagine si capisce meglio quanto spiegato:



In questo caso, nel primo processo, mappo 4 thread di livello utente in 3 thread di livello kernel e così via con gli altri processi. Questo consente di gestire diversamente i thread e di ottenere performance migliori perché i thread a livello kernel sono di meno, in numero, di quelli a livello utente e consentono operazioni di scheduling più agevoli.

Questo mapping avviene mediante l'operazione di **thread pooling**, ovvero, una operazione nella quale ogni processo a livello utente indica quanti thread a livello kernel gli servono per essere eseguito. Il thread pooling consente inoltre ai thread kernel di continuare ad esistere anche alla terminazione del thread utente ai quali sono associati. Il thread kernel è quindi riutilizzabile per essere associato ad un nuovo thread utente. Questo particolare tipo di thread kernel permanenti prendono il nome di **worker thread**, perché svolgono diverse funzioni, in base al kernel utente che gli viene assegnato.

Questo tipo di implementazione ha però un problema già visto nella implementazione User-Level: se un solo thread utente finisce in stato blocked, allora l'OS blocca l'intero processo multithread. Un'altra limitazione è data dal fatto che thread di uno stesso processo non si possono eseguire su processori diversi. Le **attivazioni dello scheduler** permettono di superare tale limitazione.

In particolare un'attivazione dello scheduler è un thread kernel in grado di avvisare di certi eventi di una libreria di threading a livello utente.

# Lezione 11

---

## Thread signal

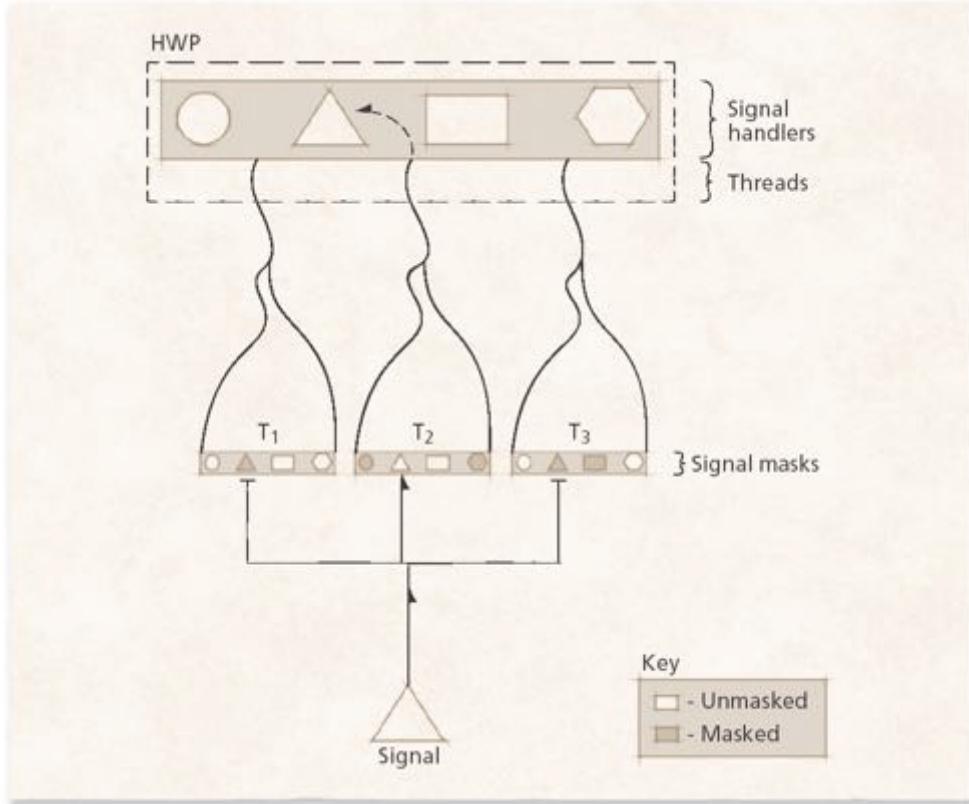
I signal sono presenti anche nella comunicazione tra i thread. Ricevendo un signal il processo ricevente esegue una funzione per gestire il signal, fatto ciò il processo riprende la sua esecuzione, lo stesso avviene per i thread. Abbiamo due tipi di signal:

- **Sincroni:** si verifica come conseguenza diretta di una istruzione che abbiamo eseguito. Esempio di questo tipo di comportamento è quando proviamo a fare una operazione di accesso alla memoria non consentita, ricevendo conseguentemente un signal.
- **Asincroni:** sono indipendenti dal contesto di esecuzione specifico, a prescindere da cosa sta facendo il processo in quel momento può ricevere un signal esterno dovuto al completamento di una operazione di IO o altre operazioni.

Se non implementiamo più thread nel nostro processo (single-thread), il processo ricevente il signal, gestisce il signal come se fosse destinato a se stesso.

Se invece abbiamo processi multithread la gestione dei signal diventa più complessa, infatti in questo caso, l'OS prima di tutto deve capire se il signal è sincrono o meno. Se il signal è **sincrono** l'OS consegna il signal al thread in esecuzione, viceversa, se il signal è **asincrono** il signal viene destinato al processo generico che ha creato il thread ricevente e sempre l'OS dovrà capire a quale thread nello specifico il signal andrà inviato. A ciò si aggiunge un'ulteriore complicazione, infatti dobbiamo prima vedere se i thread che stiamo utilizzando sono user level o kernel level,. Come sappiamo non sempre l'OS riuscirà a vedere i thread e quindi inoltrare un signal diventerà molto più difficile. Se ad esempio stiamo utilizzando una implementazione di tipo **user-level** l'OS non è in grado di stabilire quale è il thread destinatario ma solo il processo a cui appartiene, allora l'OS potrà solo mandare il signal a quel processo, e poi sarà il processo a dover capire a quale thread inoltrarlo.

Per implementare questa operazione si usa il **masking**: una operazione che ci permette di specificare, a livello di singolo processo, per ogni thread, quali sono i signal che sono di interesse che vogliamo ricevere e quali non lo sono. Graficamente possiamo capire meglio in cosa consiste l'operazione:



Ogni thread maschera i signal che non vuole ricevere e non maschera quelli che invece sa gestire e che gli interessa ricevere. Il signal, che è stato inviato dall'OS al processo, sarà "raccolto" solo dai thread abilitati a ricevere quello specifico tipo di thread. Ovviamente la gestione del signal si avrebbe nel momento in cui il processo (o i processi) che lo ha ricevuto ritorna in esecuzione nel processore.

Nel caso del **mapping uno-uno** (quello relativo alla implementazione **kernel-level**), il sistema vede tutti i thread che ci sono quindi lo stesso OS saprà inviare i signal ai thread specifici.

Nel caso del **mapping multi-molti** (implementazione **ibrida**) la gestione dei signal diventa più complicata; il problema in questo caso sta nel fatto che l'OS sa a quale thread inviare il signal, ma quest'ultimo potrebbe non essere in esecuzione in quel momento specifico. Quello che si fa in questo tipo di implementazione è l'utilizzo di un buffer di signal, dove vengono inseriti dei signal in fase di attesa che aspettano di essere inviati allo specifico thread fino a quando quest'ultimo non sarà in esecuzione.

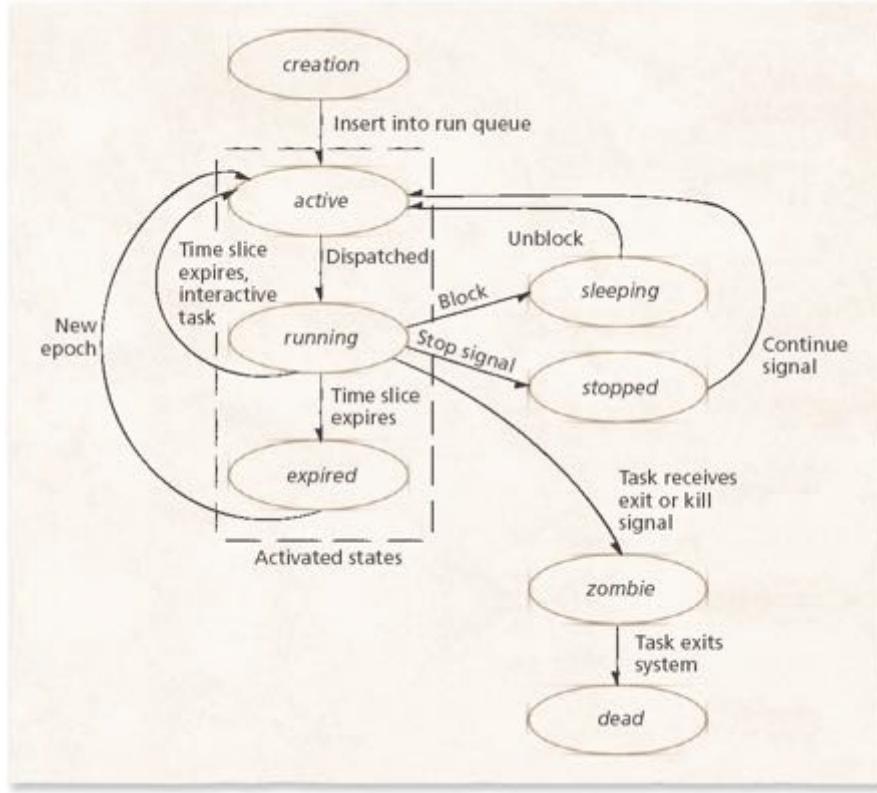
### Terminazione dei thread

Possono esserci dei casi in cui il thread termina prima del previsto, come nel caso di ricezione di un signal di cancellazione che possono arrivare o da un processo o da un altro thread dello stesso processo. Il problema della cancellazione prematura dei thread è che la terminazione improvvisa porta a svariati problemi della gestione condivisa dei dati tra più thread corrompendo spesso il risultato delle istruzioni. Ecco quindi il senso di mascherare signal di cancellazione per thread cooperativi che non devono essere cancellati prematuramente.

### Linux threads

Linux non supporta nativamente i thread, infatti la maggior parte del kernel Linux non fa una distinzione tra processi e thread, questo significa che entrambi saranno trattati e gestiti come task. Dal punto di vista implementativo vedremo che la creazione di thread comporta l'utilizzo di librerie esterne.

Dato che, come dicevamo, in Linux non c'è distinzione tra thread e processi, ciò significa che entrambe queste entità sono gestite in maniera uguale dallo scheduler. Vediamo un grafico che ci mostra il diagramma di stato dei thread Linux:



L'implementazione dei thread che vedremo nel dettaglio è quella che fa uso della libreria **pthreads** dello standard **POSIX**. In questo standard quando un thread genera un signal di tipo sincrono, allora quest'ultimo verrà inviato direttamente al thread in esecuzione. Nel caso invece di signal non specifici allora quest'ultimo verrà inviato all'intero processo dove sarà implementato il masking.

### Windows XP thread

In Windows i thread sono implementati nativamente, ecco perché c'è una distinzione netta tra thread e processi anche nello scheduler. In Windows utilizziamo inoltre delle politiche di **pooling** che permettono la creazione di **worker thread** associati in maniera permanente ad un certo processo, a prescindere da quale uso ne farà quel processo.

### Programmazione di Thread

Ribadiamo che nella fork il processo figlio riceve una copia della memoria del processo padre, tuttavia il figlio può modificare la propria memoria senza influenzare il padre in quanto i due processi sono indipendenti. Quando invece creiamo dei thread, tutti i thread condividono lo stesso spazio di memoria e i file descriptor, ciò significa che la modifica di una variabile fatta da un thread sarà vista da tutti i thread. Lo standard POSIX consente di creare threads tramite la libreria **pthreads**. Per farlo dobbiamo includere la libreria nel nostro codice:

```
#include <pthread.h>
```

E inoltre fondamentale in fase di compilazione aggiungere il comando **-lpthread** tramite il quale si effettua il linking alla libreria esterna.

### Creazione thread e funzione pthread\_create

Ad ogni thread viene associato un **thread ID** specifico. Questo ID viene memorizzato in una struttura dati di tipo **pthread\_t**, similmente a come avviene con i processi.

Inoltre nel momento in cui creiamo un thread, dobbiamo immaginare che quello che farà questo thread sarà specificato nella sua **thread function**: la funzione che contiene il codice che deve eseguire il thread.

Di conseguenza, nel momento in cui la thread function restituisce un valore, il thread avrà terminato la sua esecuzione e potrà essere terminato.

In C abbiamo una specifica funzione per la creazione di thread che fa uso di puntatori. La funzione in questione è **pthread\_create**.

Vediamo come avviene la creazione dei thread:

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine) (void *),
                  void *arg);
```

A cosa servono questi puntatori nella funzione? Quello relativo ai thread argument serve a fornire i dati di input per la thread function, il valore di ritorno è un puntatore all'output che è stato creato, che viene passato al processo che ha creato il thread. Si utilizzano i puntatori perché i thread condividono lo stesso spazio di memoria come sappiamo.

Analizziamo ora i parametri che passiamo alla funzione: il primo parametro è l'**ID univoco del thread** di tipo pthread\_t, il secondo argomento è un **puntatore a dei parametri (dati) di default** che devono essere gestiti dal thread e tipicamente questo valore sarà settato a **NULL** nei nostri esempi, il terzo parametro di input è un **puntatore alla thread function**, infine come ultimo parametro troviamo un **puntatore all'insieme di argomenti di input** che inviamo al thread in fase di creazione, nello specifico questo puntatore specifica su quali dati dovrà lavorare il thread.

Quando chiamiamo la funzione di creazione di thread **pthread\_create** in un programma, quest'ultima verrà eseguita e avverrà la creazione del thread, e da quel momento in poi il flusso di esecuzione del processo continua in maniera sequenziale, con la differenza che ora quel processo avrà un nuovo thread.

Ciò che dobbiamo tenere bene a mente è che tutti i thread nel sistema vengono schedulati in modo asincrono, esattamente come avviene per i processi, quindi non abbiamo alcuna garanzia sull'ordine di esecuzione dei thread e conseguentemente sull'ordine in cui avverranno le operazioni all'interno del processo.

Vediamo un primo eSEMPIO di creazione di thread:

The screenshot shows a desktop environment with a dark theme. A terminal window titled "Scrivania [Running]" is open, showing C code for creating threads. The code includes a function to print characters to stderr and the main program which creates a thread and prints 'o' characters. Below the terminal, a terminal window titled "ubuntu1804 [Running]" shows the command line used to build and run the program.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

/* Prints '-'s to stderr. The parameter is unused. Does not return. */
void* print_X (void* unused) {
    while (1) {
        printf("-\n");
        fflush(stdout); // stdout stream is buffered, so we need
                        // to force it for showing its content

        sleep(1);      // sleep in seconds
    }
    return NULL;
}

/* The main program. */
int main () {
    pthread_t thread_id;

    // thread id - NULL (default attributes) - th func
    pthread_create(&thread_id, NULL, &print_X, NULL);

    /* Print o's continuously */
    while (1) {
        printf("o\n");
        fflush(stdout);
        sleep(1);
    }
    return 0;
}
```

```
marco@marco-VirtualBox:~/Scrivania/thread$ gcc -o thread-create thread-create.c
marco@marco-VirtualBox:~/Scrivania/thread$ ./thread-create
```

Vediamo che abbiamo incluso la libreria per la creazione dei thread. All'interno del main abbiamo la creazione del thread mediante la funzione `pthread_create` a cui è passata come parametro l'ID del thread e la thread function che deve eseguire il thread. Il main va immediatamente avanti con l'istruzione successiva, ovvero il while che stampa un carattere "o" per riga.

Vediamo ora di analizzare la funzione del thread print\_X, nello specifico questa funzione ha in input un puntatore inutilizzato (perché non necessita di dati esterni) e ciò che fa è stampare un trattino (-) per riga. Le istruzioni di print\_X sono quindi analoghe alle istruzioni dentro il while nel main.

Notiamo dall'output che la prima istruzione è quella del while, dopo, per un motivo qualsiasi legato allo scheduler, presumibilmente la funzione sleep(1), viene eseguito il thread. Questo avviene all'infinito. L'output è totalmente casuale ed è un caso che i caratteri si alternino così metodicamente. Se aumentassimo lo sleep della thread function, ad esempio, l'output non seguirebbe più un pattern definito come quello precedente, ma l'alternanza dei caratteri sarebbe casuale, infatti, modificando l'esempio precedente:

The screenshot shows a Linux desktop environment with a terminal window and a code editor.

**Terminal Window:**

```
marco@marco-VirtualBox: ~/Scrivania/thread$ gcc -o thread-create thread-create.c
marco@marco-VirtualBox: ~/Scrivania/thread$ ./thread-create
```

**Code Editor (Scrivania application):**

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

/* Prints '-'s to stderr. The parameter is unused. Does not return. */
void* print_X (void* unused) {
    while (1) {
        printf("-\n");
        fflush(stdout); // stdout stream is buffered, so we need
                        // to force it for showing its content
        sleep(2);       // sleep in seconds
    }
    return NULL;
}

/* The main program. */
int main () {
    pthread_t thread_id;

    // thread_id - NULL (default attributes) - th funct
    pthread_create(&thread_id, NULL, &print_X, NULL);

    /* Print o's continuously */
    while (1) {
        printf("o\n");
        fflush(stdout);
        sleep(1);
    }
    return 0;
}
```

A red circle highlights the `sleep(2);` line in the code editor.

## Monitoraggio dei thread e comandi ps e top

Possiamo monitorare i thread in esecuzione dal terminale mediante comandi di sistema. Utilizzando il **comando ps**, nello specifico:

```
ps -T -p pid
```

posso visualizzare i thread relativi al processo con PID=pid. Ad esempio:

```
ps -T -p 10171 //Visualizza i thread del processo con PID pari a 10171
```

```
marco@marco-VirtualBox:~$ ps -T -p 10171
 PID  SPID TTY      TIME CMD
10171 10171 pts/0    00:00:00 thread-create
10171 10172 pts/0    00:00:00 thread-create
marco@marco-VirtualBox:~$ █
```

Gli ID dei thread vengono etichettati come **SPID** nel terminale, che sta per **Soft Process ID**. Inoltre dobbiamo sapere che la funzione main di un programma genera sempre un thread, quindi nel caso dell'esempio precedente vedremo nel terminale la presenza di due thread, uno è il main, l'altro è il thread vero e proprio.

Possiamo utilizzare anche il comando **top**, secondo la seguente sintassi:

```
top -H -p pid
```

che rispetto al comando con ps è dinamico, mostra cioè in tempo reale la gestione dei thread di uno specifico processo nel sistema operativo. Le informazioni che ci da sono molte di più rispetto al comando analogo effettuato con ps.

```
marco@marco-VirtualBox: ~
File Modifica Visualizza Cerca Terminale Aiuto
top - 11:20:30 up 4:42, 1 user, load average: 0,01, 0,03, 0,02
Threads: 2 total, 0 running, 2 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0,0 us, 0,0 sy, 0,0 ni,100,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
KiB Mem : 1009088 total, 76544 free, 573696 used, 358848 buff/cache
KiB Swap: 483800 total, 28724 free, 455076 used. 213856 avail Mem

 PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
10171 marco    20    0 14888    936    856 S 0,0  0,1   0:00.03 thread-crea+
10172 marco    20    0 14888    936    856 S 0,0  0,1   0:00.01 thread-crea+
```

# Lezione 12

---

## Passaggio di dati ai thread e funzione pthread\_join

Per passare argomenti ai thread dobbiamo attenerci alla sintassi della funzione **pthread\_create()** che prevede il passaggio di parametri solo tramite puntatori come abbiamo già visto nella lezione precedente. In particolare, teniamo a mente che, se dobbiamo passare più dati utilizzeremo comunque un solo puntatore. Per farlo creiamo una **struct** che contiene tutti gli elementi necessari che vogliamo passare come input al thread.

Dal punto di vista della memoria, infatti, una struct, così come un array, è vista dal sistema come una struttura di dati allocati in celle di memoria contigue. Sia gli array che le struct vengono allocate a blocchi, ciò significa che quando andiamo ad allocare le seguenti strutture andiamo a specificarne la dimensione perché il compilatore deve trovare il numero di celle di memoria contigue e allocare la struttura nella sua interezza. Inoltre nel caso di array, oltre che essere contigui, gli elementi sono anche omogenei.

La struct ci permette invece di mettere insieme oggetti che non sono tutti dello stesso tipo.

Quando accediamo a queste strutture tramite puntatori, non facciamo altro che puntare all'indirizzo iniziale di allocazione della struttura dati (array o struct che sia). Capiamo quindi perché nella funzione **pthread\_create()** si fa uso dei puntatori per passare argomenti: tramite un singolo puntatore possiamo passare una intera struttura di dati.

Vediamo un esempio di creazione di thread in maniera da capire meglio come passare dati ai thread:

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>          thread-create2.c

struct print_params {
    char character;
    int times;
};

int main () {
    pthread_t thread1_id;
    pthread_t thread2_id;
    struct print_params thread1_args;
    struct print_params thread2_args;

    thread1_args.character = 'X';
    thread1_args.times = 20;

    thread2_args.character = 'o';
    thread2_args.times = 30;

    pthread_create (&thread1_id, NULL, &print_char, &thread1_args);
    pthread_create (&thread2_id, NULL, &print_char, &thread2_args);

    return 0;
}
```

Anche qui ovviamente abbiamo l'include della libreria per i thread pthread.h. In questo caso, rispetto all'esempio della precedente lezione, passiamo al thread dei dati in input. Lo facciamo definendo una struct di nome print\_params con un char e un intero all'interno.

Dopodiché creiamo il main, e due thread differenti che si occupano di stampare due caratteri diversi un numero di volte differenti. Ora vado a definire due differenti struct, la prima conterrà i parametri del thread1, la seconda del thread2. Specifichiamo poi i caratteri che voglio stampare per ogni thread insieme al numero di volte in cui voglio stampare quel dato carattere.

In particolare il thread1 stamperà 20 volte il carattere "X", mentre il secondo thread stamperà 30 volte il carattere "o".

Per creare il thread utilizzo la stessa sintassi dell'esempio prima, con la differenza che l'ultimo argomento saranno i puntatori ai parametri definiti nella struct puntata da &thread1\_args e &thread2\_args.

La funzione print\_char sarà così definita:

```
void* print_char (void* parameters)
{
    struct print_params* pp = (struct
print_params*) parameters;

    for (int i=0; i<= pp->times; i++) {
        printf("%c", pp->character);
        fflush(stdout);
        sleep(1);
    }
    return NULL;
}
```

La prima cosa da fare è andare a creare una struct di nome pp di tipo print\_params a cui assegnare i parametri ricevuti dalla thread function (castati a struct print\_params, dato che ciò che riceviamo in input è un puntatore). Quello che avviene dopo è un semplice for che va da 0 al numero specificato dalla struttura pp nella variabile *times* e all'interno del ciclo stamperemo il carattere corrispondente al thread. Ogni thread eseguirà questa funzione autonomamente andando a riempire la funzione con i dati definiti in &thread\_args.

Sorge un problema: &thread1\_args e &thread2\_args sono specificati nel main, ma come sappiamo nei thread, a differenza dei processi, si condivide la stessa area di memoria. Alla fine del main eseguiamo le funzione pthread\_create(), ma quest'ultima viene eseguita e ritorna immediatamente e il thread chiamante (cioè il main) continua la sua esecuzione dal punto successivo alla creazione, in questo caso l'altra pthread\_create.

Dopo la creazione del secondo thread il main continua con la successiva istruzione, ovvero return 0. Ma questo comporta la deallocazione delle risorse per quel processo, che a sua volta significa che la memoria allocata per il processo viene deallocata, e con esso anche i thread allocati all'interno della memoria del processo.

Capiamo bene che per questo motivo dobbiamo in qualche modo evitare che il processo main termini prima della terminazione dei suoi thread.

Per farlo utilizziamo il **join**, in particolare la funzione **pthread\_join** che riceve due parametri: prima l'**ID del thread che vogliamo aspettare**, e in secondo un **puntatore a una variabile void che punta al valore di ritorno del thread**. Il join è l'equivalente del wait per i processi.

Il precedente codice cambia quindi nella seguente maniera:

```

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

struct print_params {
    char character;
    int times;
};

int main () {
    pthread_t thread1_id;
    pthread_t thread2_id;

    struct print_params thread1_args;
    struct print_params thread2_args;

    thread1_args.character = 'X';
    thread1_args.times = 20;

    thread2_args.character = 'o';
    thread2_args.times = 30;

    pthread_create (&thread1_id, NULL, &print_char, &thread1_args);
    pthread_create (&thread2_id, NULL, &print_char, &thread2_args);

    pthread_join(thread1_id,NULL); Il secondo argomento è un puntatore al
pthread_join(thread2_id,NULL); valore di ritorno del thread, se non esiste si usa NULL

    return 0;
}

```

```

void* print_char (void* parameters)
{
    struct print_params* pp = (struct
    print_params*) parameters;

    for (int i=0; i<= pp->times; i++) {
        printf("%c",pp->character);
        fflush(stdout);
        sleep(1);
    }
    return NULL;
}

```

Questa volta se andiamo ad eseguire il codice visualizzeremo correttamente la sequenza di caratteri stampata dai due thread. Per vedere se effettivamente abbiamo 3 thread (ricordiamo che il main definisce un thread) utilizziamo il comando top:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
10463	marco	20	0	88620	812	732	S	0,0	0,1	0:00.00	thread-crea+
10464	marco	20	0	88620	812	732	S	0,0	0,1	0:00.00	thread-crea+
10465	marco	20	0	88620	812	732	S	0,0	0,1	0:00.00	thread-crea+

Se vogliamo che il thread restituisca un valore possiamo utilizzare un secondo puntatore, che gestisco nel secondo parametro di pthread\_join.

Vediamo un [esempio](#):

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

struct print_params {
    char character;
    int times;
};

int main () {
    pthread_t thread1_id;
    pthread_t thread2_id;

    int* thread1_returnValue;
    int* thread2_returnValue;

    struct print_params thread1_args;
    struct print_params thread2_args;

    thread1_args.character = 'X';
    thread1_args.times = 20;
    thread2_args.character = 'o';
    thread2_args.times = 30;

    pthread_create (&thread1_id, NULL, &print_char, &thread1_args);
    pthread_create (&thread2_id, NULL, &print_char, &thread2_args);

    pthread_join(thread1_id, (void**) &thread1_returnValue);
    pthread_join(thread2_id, (void**) &thread2_returnValue);

    printf("\n Value returned by Thread 1 is: %d", *thread1_returnValue);
    printf("\n Value returned by Thread 2 is: %d\n", *thread2_returnValue);
    return 0;
}

void* print_char (void* parameters)
{
    struct print_params* pp = (struct print_params*) parameters;

    for (int i=0; i<= pp->times; i++) {
        printf("%c", pp->character);
        fflush(stdout);
        sleep(1);
    }
    return (void*) &(pp->times);
}
```

In questo caso voglio che thread1 e thread2 restituiscano due valori, che nello specifico sono degli interi. Al solito definisco le strutture, i thread e la thread function ecc.. Successivamente faccio il join, per gestire il valore di ritorno, utilizzo il puntatore di tipo intero definito prima, di nome thread1\_returnValue. Per farlo però devo modificare la thread function che restituirà un puntatore void in cui vado a mettere il valore *times* della struct pp.

La `pthread_join` quindi passa al `main` il valore di ritorno, che stamperà questo valore, che altro non è che il numero di volte in cui i thread hanno stampato i caratteri.

Vediamo l'esecuzione:

I thread in java sono creati attraverso la classe **Thread**. Specifichiamo che esistono diversi modi di implementare i thread in java, così come la mutua esclusione, ma noi nel corso delle lezioni vedremo solo i metodi basilari ed essenziali.

Il codice eseguito dal thread (la thread function) è specificato, in un metodo uguale in tutti i thread, di nome **run()**.

Vediamo un eSEMPIO:

```
1 // Fig. 4.9: ThreadTester.java
2 // Multiple threads printing at different intervals.
3
4 public class ThreadTester {
5
6     public static void main( String [] args )
7     {
8         // create and name each thread
9         PrintThread thread1 = new PrintThread( "thread1" );
10        PrintThread thread2 = new PrintThread( "thread2" );
11        PrintThread thread3 = new PrintThread( "thread3" );
12
13        System.err.println( "Starting threads" );
14
15        thread1.start(); // start thread1; place it in ready state
16        thread2.start(); // start thread2; place it in ready state
17        thread3.start(); // start thread3; place it in ready state
18
19        System.err.println( "Threads started, main ends\n" );
20
21    } // end main
22
23 } // end class ThreadTester
24
```

All'interno della classe ThreadTester troviamo un main dove sono dichiarati tre oggetti thread. Creiamo dunque i 3 oggetti Thread1, Thread2 e Thread3. Poi facciamo una print per notificare l'avvio dei thread. Per eseguire i thread utilizziamo il metodo **start()**. Vediamo dunque nello specifico cosa fa la classe PrintThread:

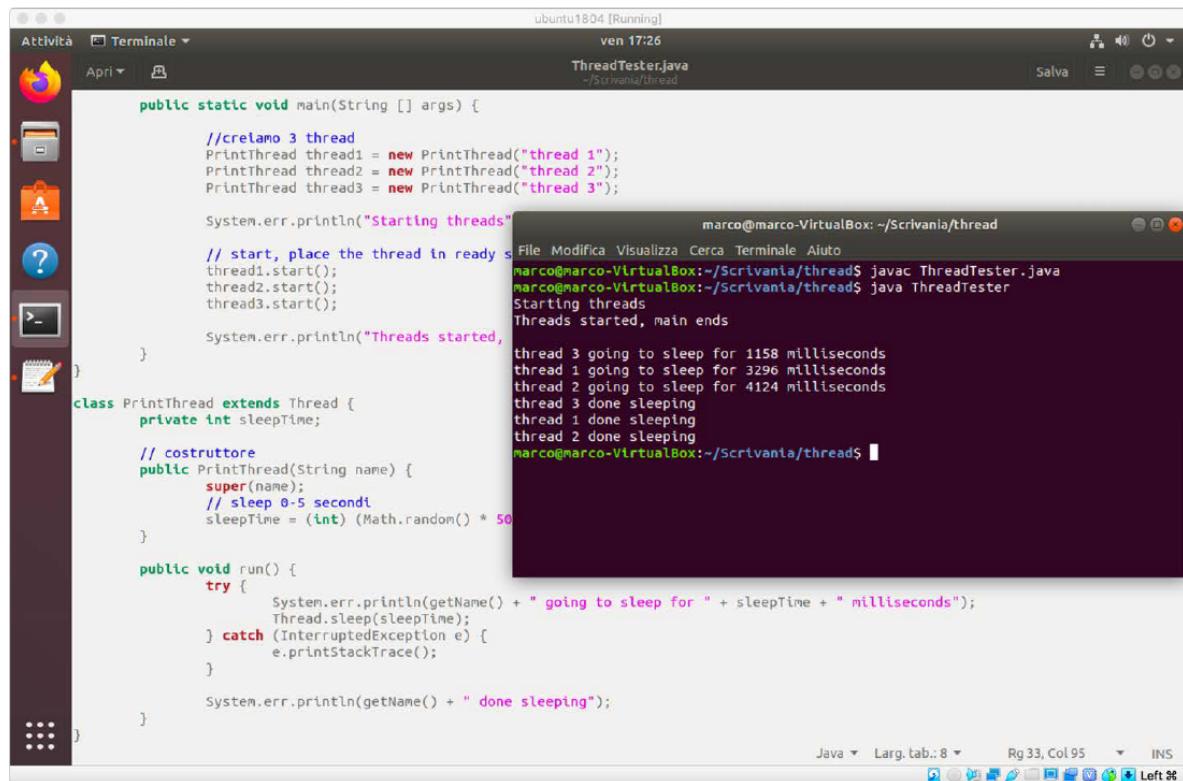
```
25 // class PrintThread controls thread execution
26 class PrintThread extends Thread {
27     private int sleepTime;
28
29     // assign name to thread by calling superclass constructor
30     public PrintThread( String name )
31     {
32         super( name );
33
34         // pick random sleep time between 0 and 5 seconds
35         sleepTime = ( int ) ( Math.random() * 5001 );
36     } // end PrintThread constructor
37
38     // method run is the code to be executed by new thread
39     public void run()
40     {
41         // put thread to sleep for sleepTime amount of time
42         try {
43             System.err.println( getName() + " going to sleep for " +
44             sleepTime + " milliseconds" );
45
```

```

46         Thread.sleep( sleepTime );
47     } // end try
48
49     // if thread interrupted during sleep, print stack trace
50     catch ( InterruptedException exception ) {
51         exception.printStackTrace();
52     } // end catch
53
54     // print thread name
55     System.err.println( getName() + " done sleeping" );
56
57 } // end method run
58
59 } // end class PrintThread

```

Questa classe estende la **classe Thread**. All'interno della classe definisco il costruttore dove all'interno troviamo una variabile per determinare il tempo di sleep (randomico) di quel thread e il nome del thread. Troviamo poi il metodo **run()** dove c'è un blocco try-catch. All'interno del try abbiamo la stampa del nome del thread seguiti da un periodo di sleep variabile per ogni thread. Ecco un possibile risultato che ci aspettiamo in output sulla shell:



The screenshot shows a terminal window titled 'Terminale' on a Linux desktop. The terminal content is as follows:

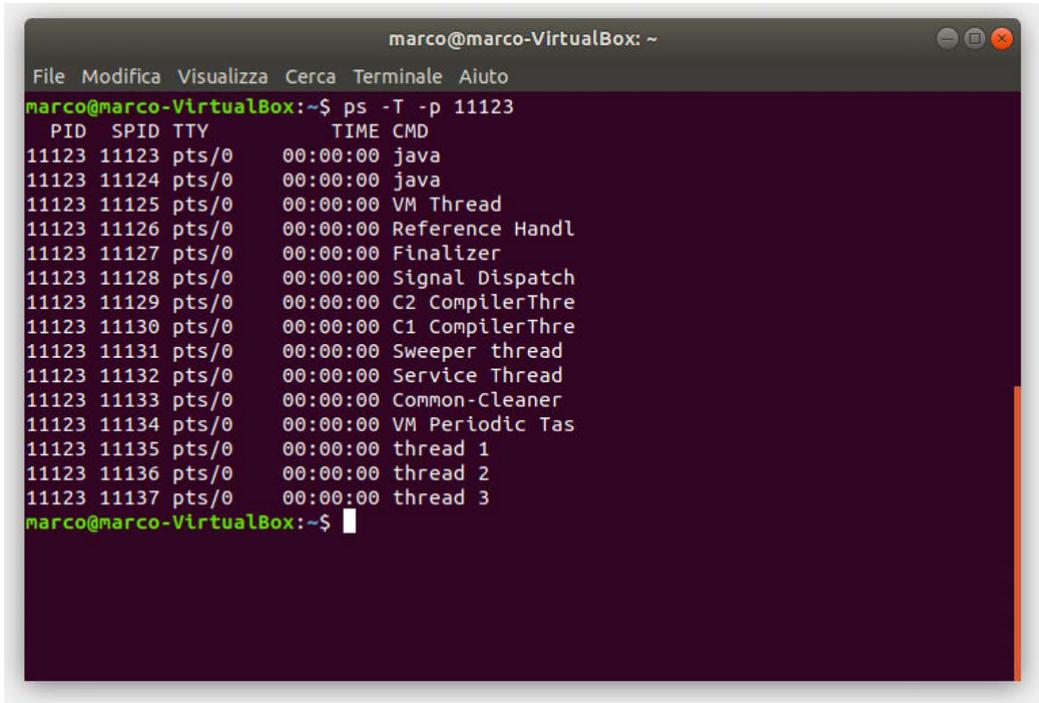
```

ubuntu1804 [Running]
ven 17:26
ThreadTester.java
-/Scrivania/thread
File Modifica Visualizza Cerca Terminale Aiuto
marco@marco-VirtualBox:~/Scrivania/thread$ javac ThreadTester.java
marco@marco-VirtualBox:~/Scrivania/thread$ java ThreadTester
Starting threads
Threads started, main ends

thread 3 going to sleep for 1158 milliseconds
thread 1 going to sleep for 3296 milliseconds
thread 2 going to sleep for 4124 milliseconds
thread 3 done sleeping
thread 1 done sleeping
thread 2 done sleeping
marco@marco-VirtualBox:~/Scrivania/thread$ 

```

Vediamo di capire cosa succede a livello della memoria. Se utilizzo il comando ps per vedere i thread creati in java ottengo un risultato del genere:



```
marco@marco-VirtualBox:~$ ps -T -p 11123
 PID  SPID TTY      TIME CMD
11123 11123 pts/0    00:00:00 java
11123 11124 pts/0    00:00:00 java
11123 11125 pts/0    00:00:00 VM Thread
11123 11126 pts/0    00:00:00 Reference Handl
11123 11127 pts/0    00:00:00 Finalizer
11123 11128 pts/0    00:00:00 Signal Dispatch
11123 11129 pts/0    00:00:00 C2 CompilerThre
11123 11130 pts/0    00:00:00 C1 CompilerThre
11123 11131 pts/0    00:00:00 Sweeper thread
11123 11132 pts/0    00:00:00 Service Thread
11123 11133 pts/0    00:00:00 Common-Cleaner
11123 11134 pts/0    00:00:00 VM Periodic Tas
11123 11135 pts/0    00:00:00 thread 1
11123 11136 pts/0    00:00:00 thread 2
11123 11137 pts/0    00:00:00 thread 3
marco@marco-VirtualBox:~$
```

Notiamo che non ci sono solo i 4 thread che ci aspettavamo, ma molti di più che sono a supporto dei 3 thread principali da noi definiti. Questo a causa della IJVM.

Ovviamente per noi programmatori è molto più semplice creare dei thread così in java, il rovescio della medaglia è che per soli 3 thread, java ne dovrà creare molti di più di supporto che saranno ovviamente schedulati e concorreranno per il processore. In definitiva avremo una implementazione meno efficiente rispetto ai thread definiti in C. |

Utilizzando il comando top otterremmo lo stesso risultato.

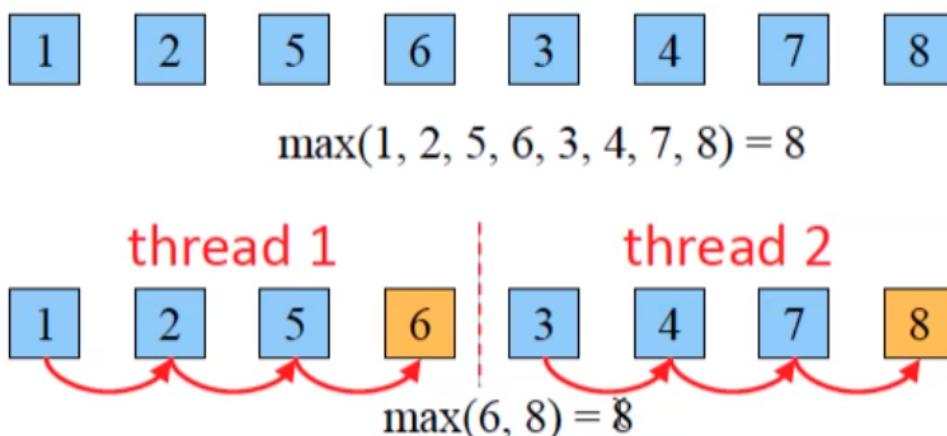
In java possiamo anche utilizzare il comando **jstack** che stampa lo stack relativo ai thread creati all'interno del processo specificato nel valore .

# Lezione 13

Nella lezione precedente abbiamo visto come per l'implementazione dei thread in Java si utilizzi una apposita **classe Thread**. Abbiamo poi visto un primo esempio di creazione di thread in java con relativo metodo **run**. Abbiamo anche visto l'utilizzo del metodo **join** per consentire l'esecuzione dei thread senza che il processo principale termini.

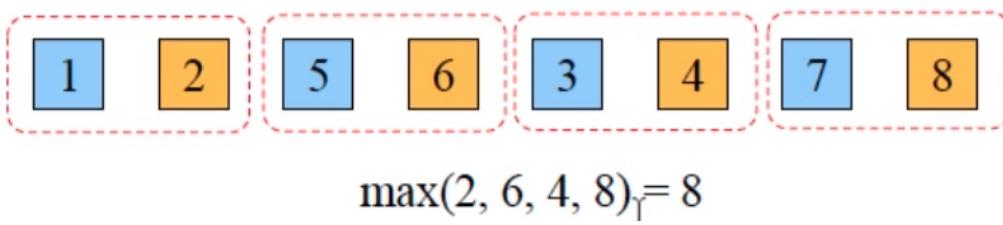
Vediamo ora un esempio ragioniamo su come più thread possano essere utilizzati per calcolare il massimo valore contenuto in un array:

Prendiamo un array e calcoliamone il massimo confrontando a due a due gli elementi e scartando il minore. Questo tipo di algoritmo effettua n confronti. Questo si può implementare con due thread i quali dividono l'array a metà e vanno a trovare il massimo per le due metà. Poi si confronteranno i due massimi locali per trovare quello globale. A farlo potrebbe essere il thread main. Per fare questa operazione però è necessario che il thread main aspetti il completamento delle operazioni degli altri due thread. Graficamente:



Il vantaggio dell'uso dei thread è che ciascuno dei thread può essere implementato su un processore diverso così da avere un vero e proprio parallelismo avendo così effettivamente  $n/2$  confronti. Le operazioni che faccio sul thread di sinistra sono infatti indipendenti da quelle che faccio sull'array di destra.

E se avessi 4 Thread? I ruoli dei thread sarebbero gli stessi:



Il main dovrebbero però fare un confronto tra 4 massimi locali. Notiamo che è utile parallelizzare solo nel momento in cui i confronti si minimizzino.

L'idea quindi è di prendere l'array, dividerlo in due parti da assegnare a due thread i quali calcolano i massimi locali che poi verranno confrontati dal thread main. Vediamo l'implementazione di un programma del genere:

```

public class Thread4Max extends Thread {

    private int[] values;
    private int a,b;
    private int max;

    public static void main(String[] args) throws InterruptedException
    {

        int[] arr = new int[100];

        for (int i=0; i<arr.length; i++) {
            arr[i] = (int) (Math.random() * 1001); //oppure = i;
        }

        int max = getMax(arr);
        System.out.println("Massimo = " + max);
    }
}

```

Definiamo la nuova classe Thread4Max con delle variabili condivise tra i thread quali a, b e max. Dopo troviamo il main dove dichiaro che l'array è fatto da 100 elementi, riempiamo successivamente l'array con numeri casuali. A questo punto il main fa solo una chiamata ad una funzione getMax() che invia in input l'array. Questa funzione restituirà poi il valore massimo che verrà restituito a schermo. Vediamo quindi come è definito il metodo getMax:

```

public static int getMax(int[] values) throws InterruptedException {
    int len = values.length;
    int max;

    // creiamo 2 thread
    Thread4Max thread1 = new Thread4Max(values, 0, len/2);
    System.out.println("Il thread 1 calcola il massimo dei valori in
                       posizione da 1 a " + len/2 );

    Thread4Max thread2 = new Thread4Max(values, len/2, len);
    System.out.println("Il thread 2 calcola il massimo dei valori in
                       posizione da " + len/2 + " a " + len );

    thread1.start();
    thread2.start();

    // aspettiamo il completamento e calcoliamo il max
    thread1.join();
    thread2.join();

    System.out.println("Massimo thread1 = " + thread1.max);
    System.out.println("Massimo thread2 = " + thread2.max);

    if(thread1.max > thread2.max) {
        max = thread1.max;
    } else {
        max = thread2.max;
    }
    return max;
}

```

Questo stesso metodo crea i thread. Nel primo thread definisco un costruttore che lavora sulla prima metà degli elementi dell'array. Il thread2 invece lavora sulla seconda metà dell'array. Dopodiché effettuo lo start dei thread.

Vediamo ora la classe Thread4Max:

```

public Thread4Max(int[] values, int a, int b ) {
    this.a = a;
    this.b = b;
    this.values = values;
}

public void run() {
    max = values[a];

    for(int i=a+1; i<b; i++) {
        if(values[i]>max) {
            max = values[i];
        }
    }
}

```

Quando in getMax() faccio lo start del thread viene eseguito il run definito nella classe thread4Max. Il metodo run() calcola il massimo locale in una metà dell'array. Questo avviene tramite un for che scorre gli elementi passati dai thread e li confronta singolarmente per trovarne il maggiore. Al termine del run() ciascun thread conterrà nella variabile max il massimo locale per ogni metà dell'array.

Torniamo ora al metodo getMax(). Dopo l'esecuzione dei thread effettuiamo il join dei thread. A questo punto posso stampare i due valori massimi calcolati dai due thread ed effettuare infine un confronto per vedere quale dei due valori è il maggiore. Infine effettuiamo il return del valore massimo.

Importante: il codice è particolarmente efficiente su sistemi con due processori dato che i due thread verrebbero eseguiti parallelamente diminuendo così il tempo di esecuzione. Se lavorassimo su sistemi con 4 processori, ad esempio, sarebbe ancora più efficiente dividere il lavoro tra 4 thread che lavorano parallelamente.

Vediamo ora l'output del programma:

```

marco@marco-VirtualBox:~/Scrivania/thread
File Modifica Visualizza Cerca Terminale Aiuto
marco@marco-VirtualBox:~/Scrivania/thread$ java Thread4Max
Il thread 1 calcola il massimo dei valori in posizione da 1 a 50
Il thread 2 calcola il massimo dei valori in posizione da 50 a 100
Massimo thread1 = 987
Massimo thread2 = 977
Massimo = 987
marco@marco-VirtualBox:~/Scrivania/thread$ █

```

# Lezione 14

## Scripting Linux

Il comando **man** ci permette di avere informazione sui vari comandi di sistema e non solo, come se fosse un manuale del sistema. La sintassi del comando è la seguente:

```
$ man [1-9] nomecomando
```

Il seguente codice riceve in input il nome di un programma, una utility o di una funzione. In particolar modo, ogni comando appartiene ad una **sezione** specifica:

1. Programmi eseguibili e comandi della shell.
2. Chiamate al sistema (fornite dal kernel). La funzione fork(), ad esempio, rientra tra queste.
3. Chiamate alle librerie (funzioni all'interno delle librerie di sistema).
4. Comandi relativi a file speciali (solitamente si trovano in /dev).
5. Formati dei file e convenzioni.
6. Giochi.
7. Macro e convenzioni varie.
8. Comandi per l'amministrazione del sistema.
9. *Eventuale*: comandi di root.

Vediamo alcuni esempi:

```
$ man man #visualizziamo il manuale del manuale
```

```
$ man sleep #Non specificando nulla in input ad eccezione del comando sleep, la ricerca viene effettuata su tutte le sezioni e restituisce la pagina della prima pagina trovata
```

```
$ man 1 sleep #Specifico che voglio effettuare la ricerca nella sezione 1, cioè nella sezione dei programmi eseguibili e i comandi della shell
```

```
$ man 3 sleep #Sto andando a cercare il comando sleep nella sezione 3 (ovvero nella sezione delle chiamate alle librerie)
```

Se provo a fare una ricerca su una sezione in cui non è definito il comando cercato, ad esempio, supponiamo di fare:

```
$ man 2 sleep #Cerco nella sezione delle chiamate di sistema
```

Ma la funzione sleep non è una chiamata di sistema, di conseguenza il manuale non trova nulla riguardo il comando sleep nella sezione 2.

Una ulteriore possibilità è quella di invocare il comando man col **parametro -a**, cioè:

```
$ man -a sleep
```

Questo parametro ci permette di cercare in sequenza in tutte le sezioni e visualizzare il manuale del comando per ogni sezione.

Altro esempio:

```
$ man 2 chmod
```

chmod è una chiamata di basso livello fornita dal kernel, quindi in questo caso la ricerca nel manuale produrrà un risultato utile.

```
$ man 5 passwd
```

Questo comando sta facendo riferimento al file delle password che si trova in /etc. Di conseguenza il manuale ci mostrerà la struttura usata all'interno delle password e come può essere amministrato e gestito.

```
$ man 7 shutdown
```

La sezione 8 fa riferimento a routine del sistema. In questo caso cerchiamo informazioni su shutdown. La ricerca produrrà una descrizione del comando e di tutte le opzioni con cui si può richiamare.

### Metacaratteri della shell

La shell è anche un interprete per cui ciò che scrivo all'interno della shell viene interpretato. Per farlo è necessario che ciò che scriviamo rispetti una determinata sintassi ovviamente. Vediamo quindi un po' di sintassi della shell Linux.

Per inserire un **commento** si utilizza il carattere **#**. La shell interpreterà tutto quello che è scritto dopo come un commento. Vediamo un esempio:

```
$ #prova di commento
```

Posso anche gestire all'interno della shell delle **variabili**. Per definire una variabile scrivo il nome della variabile e lo uguagliamo al valore che vogliamo assegnargli. Per esempio:

```
$ var=5
```

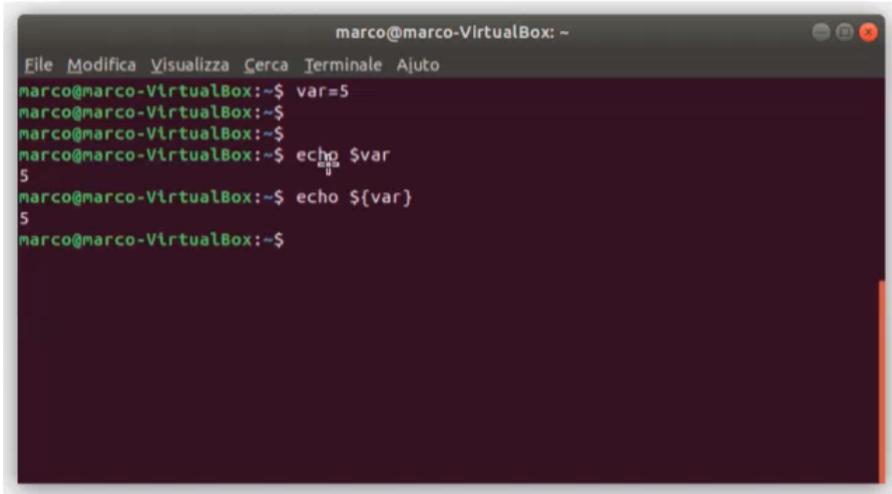
Potrei volere **visualizzare il valore delle variabili** di sistema della shell. Per farlo usiamo **\$** seguito dal nome della variabile che vogliamo visualizzare. È necessario inserire il comando **echo** prima della variabile. Per esempio:

```
$ echo $var
```

La stessa possibilità si ha utilizzando le parentesi graffe nella seguente maniera:

```
${var}
```

Se eseguiSSIMO i precedenti comandi otterremo da terminale il seguente output:



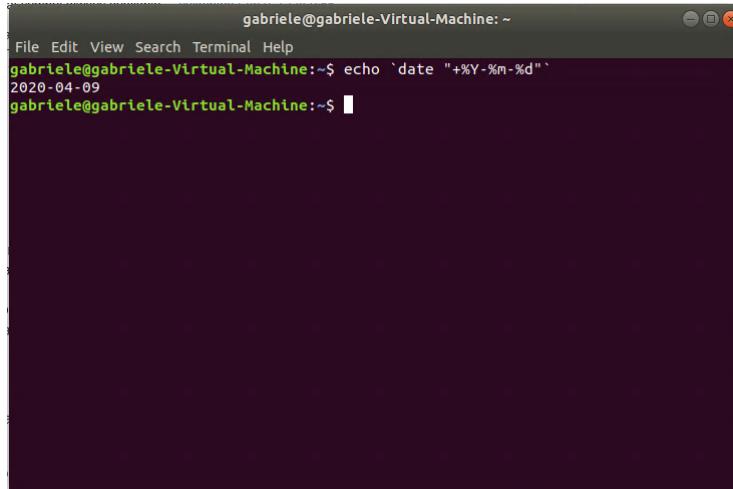
```
marco@marco-VirtualBox:~$ var=5
marco@marco-VirtualBox:~$ echo $var
5
marco@marco-VirtualBox:~$ echo ${var}
5
marco@marco-VirtualBox:~$
```

Se provo ad invocare echo su una variabile che non esiste ottengo output nullo.

Esistono anche dei caratteri speciali che venendo interpretati cambiano il significato del testo che vi è racchiuso all'interno. Vediamo quindi un paio di **metacaratteri**.

- **Backquote:** ` ... ` (sono **apici inversi**) esegue comandi ottenuti tra gli apici inversi e li sostituisce con i relativi output. Vediamo un esempio di utilizzo:

```
$ echo `date "+%Y-%m-%d"`
```



```
gabriele@gabriele-Virtual-Machine:~$ echo `date "+%Y-%m-%d"`
2020-04-09
gabriele@gabriele-Virtual-Machine:~$
```

- I **singoli apici** servono a non interpretare nulla di ciò che scriviamo all'interno. Ad esempio:

```
$ echo 'date "+%D"'
```

- I **doppi apici** interpretano solo alcuni metacaratteri, in particolare: \$, \ e gli apici inversi. Vediamo anche in questo caso un esempio:

```
$ echo 'date $var' #Non viene interpretato
```

```
$ echo "date $var" #Viene interpretato soltanto il $
```

## Operatori

Esistono poi degli **operatori** che vengono utilizzati per eseguire più comandi in sequenza o altre operazioni simili. Vediamo i principali:

- Utilizziamo l'**operatore** ; per eseguire più comandi in sequenza. È un separatore tra comandi differenti. Vediamo un [esempio](#):

```
$ sleep 4 ; echo "Hello" #Concateno i due comandi da eseguire in ordine.  
Eseguo uno sleep di 4 secondi e dopo eseguo echo di Hello.
```

- Utilizziamo l'**operatore &&** se voglio eseguire i comandi successivi solo se i precedenti hanno avuto successo. Ad [esempio](#):

```
$ mkdir a && cd a #Cambia la directory corrente in a solo se è riuscito a  
creare la directory con il comando mkdir
```

- Utilizziamo l'**operatore ||** se voglio eseguire un comando solo se il comando precedente ha fallito. Ad esempio:

```
$ (mv a.txt b.txt && echo 'fatto') || echo 'file inesistente' #Prova a  
rinominare il file a.txt in b.txt, se ci riesce stampa nel terminale  
'fatto', altrimenti stampa 'file inesistente'
```

## Redirect

Vediamo ora un modo di gestire il **redirect** dei flussi di I/O all'interno del terminale. Di default ciò che noi inseriamo nell'interprete di shell viene inserito mediante tastiera e conseguentemente visualizzato sullo standard output. Abbiamo però la possibilità di effettuare il **redirect dello standard input**, dello **standard output** e dello **standard error**.

### Redirect standard output

L'operatore che utilizziamo per il redirect dello standard output è **>**. A sinistra di **>** ci sta un comando, a destra un filename. Vediamo un [esempio](#):

```
$ echo 'ciao' > a.txt #Invece di visualizzare su shell ciao, lo scrive nel file  
a.txt  
$ cat a.txt #Mostra il contenuto del file a.txt
```

Posso usare **>>** per fare un redirect dell'output specificando però che l'operazione va fatta in modalità append, ovvero aggiungendo l'output alla fine del file. Ad [esempio](#):

```
$ ls -l >> a.txt
```

```
$ cat a.txt
```

Riprendendo un attimo il concetto di **file descriptor**, ci ricordiamo che, nello specifico l'identificativo 0 (**FD0**) rappresenta lo standard input, ovvero la tastiera. Mentre **FD1** rappresenta lo standard output, ovvero il terminale. Infine l'**FD2** rappresenta lo standard error, che di default è il terminale.

È possibile effettuare il redirect anche dei file descriptor. Vediamo come farlo con un codice c di [esempio](#):

```
#include <stdio.h>

int main() {
    printf("send to stdout\n");
    fprintf(stderr,"send to stderr\n");
    return 0;
}
```

Innanzitutto il codice effettua una printf nello standard ouput. Successivamente c'è una fprintf che serve a copiare questa stringa nello standard error. Se compilo ed eseguo, ottengo sul terminale la stampa di queste due stringhe, entrambe sul terminale dato che lo standard error è anch'esso il terminale. Se però volessimo creare un file di log per tenere traccia degli errori, allora potremmo utilizzare l'operatore > per andare a scrivere il risultato dell'errore nel file di log. Vediamolo meglio:

```
$ gcc -o test test.c
$ ./test > out.txt 2> err.txt      (redirect di stdout e stderr su 2 file diversi)
```

In questa maniera andiamo a scrivere le stringhe stampate dalle print sul file out.txt e stampiamo inoltre tutto quello che c'è da stampare sullo standard error sul file err.txt che sarà il nostro file di log. Il risultato nella shell sarà il seguente:

```
$ gcc -o test test.c
$ ./test > out.txt 2> err.txt      (redirect di stdout e stderr su 2 file diversi)
$ cat out.txt
send to stdout
$ cat err.txt
send to stderr
```

Se eseguiamo invece i seguenti comandi:

```
$ ./test &> both.txt                  (redirect di stdout e stderr su uno stesso file)
$ cat both.txt
send to stderr
send to stdout
```

Otterremmo il redirect dello standard ouput e dello standard error nello stesso file.

### Redirect dello standard input

I programmi che scriviamo si aspettano normalmente un input inserito da tastiera. Ciò che possiamo fare è cambiare lo standard input tramite l'operatore <. Vediamo un [esempio](#):

```
command < filename #L'input di command è letto dal file di nome filename
```

Un altro [esempio](#) che ci permette di capire l'utilità del redirect dello standard input è il seguente:

```
$ ls > a.txt
$ wc -l < a.txt
```

Questo ci permetterebbe di stampare il numero di linee che sono presenti nel file a.txt. Questo file contiene l'ouput del comando ls come si vede nella prima riga.

### Pipe

L'operatore che consente di realizzare una pipe all'interno della shell è l'operatore |. Il flusso di esecuzione è da sinistra verso destra. Vediamo un [esempio](#):

```
$ ls -1 | less #Questa riga permette di inviare in input a less l'ouput di ls -1
```

Il che equivale a scrivere:

```
$ ls -l > tmp  
$ less < tmp
```

# Lezione 15

Riprendiamo il discorso dell'ultima volta a partire dalla **Pipe**. Ricordiamo che la pipe si può implementare tramite l'**operatore |** che connette l'output di un programma all'input di un altro.

Vediamo un paio di esempi:

```
$ ls > elenco.txt #Reindirizziamo i nomi dei file nel file elenco.txt
```

```
$ ls >> elenco.txt #Duplico i nomi ottenuti da ls appendendo in coda i nomi
```

```
$ sort elenco.txt #Permette di ordinare i nomi in modo alfabetico
```

```
$ sort elenco.txt | unique #Ordino e rimuovo le ripetizioni nell'elenco con unique
```

Posso fare anche altre tipi di operazioni sfruttando la pipe. Ad esempio:

```
$ cat elenco.txt | head -2 #Visualizziamo mediante cat il file di elenco.txt e mostro solo le prime due righe
```

Possiamo anche usare una doppia pipe, ad esempio:

```
$ cat elenco.txt | head -2 | tail #Fa la stessa cosa di prima ma visualizza solo la seconda, delle due righe
```

Sempre tramite pipe possiamo effettuare operazione più avanzate come la creazione di un file:

```
$ >> f1 #Questo comando dice di aggiungere "" al file f1, ma se il file f1 non esiste lo crea.
```

Dopodiché potremmo volerlo duplicare tramite il comando cp:

```
$ cp f1 f2
```

A questo punto potrei fare un archivio che contiene i due file precedenti, per creare un archivio non compresso uso il **comando tar**:

```
$ tar cf A.tar f1 f2
```

Pere verificare il contenuto dell'archivio uso:

```
$ tar tvf A.tar
```

Infine comprimere l'archivio con:

```
$ gzip A.tar
```

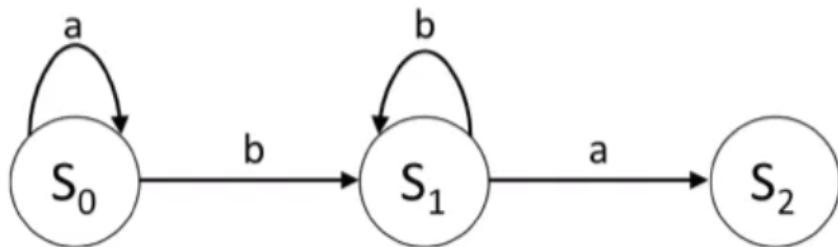
Ma abbiamo la possibilità di creare un archivio complesso con un unico comando così definito:

```
$ >> f1 && cp f1 f2 && tar cvf - f1 f2 | gzip > A.tar.gz
```

Iniziamo ora a vedere gli unici due comandi avanzati (filtr) che vedremo in questo corso.

### Grep (Global Regular Expression Print)

Tipicamente il modo in cui operano i compilatori è quello di identificare pattern sintatticamente corretti. Per identificare queste stringhe definiamo delle regole chiamate **espressioni regolari**, le quali consentono di rappresentare delle stringhe (formate da una serie di caratteri) che possono essere descritte attraverso automi a stati finiti, vediamo un esempio:



Nel diagramma di sopra vediamo 3 stati e tutte le possibili regole (frecce nel diagramma) che permettono di transitare da uno stato all'altro. Da S0 possiamo uscire in due modi, o con la condizione che prevede la lettura di un carattere "a", che ci riporta a S0 stesso, viceversa se trovo un'occorrenza della lettera "b" passerò allo stato S1; in S1 avrò anche qui due frecce, la condizione "b" che riporta a S1 e la condizione "a" che porta allo stato successivo di S2. Stringhe valide per questo esempio potrebbero essere "aba", "aaba", "ba" ecc..

Quello che fa **grep** è usare le espressioni regolari per cercare dei pattern all'interno di ciò che viene specificato in input. Ad esempio potremmo cercare un pattern all'interno di file di testo o di stream tramite l'uso di pipe. Supponiamo di avere il seguente file:

===== heroes.txt ==== Catwoman Batman The Tick Spider Man Black Cat Batgirl Danger Girl Wonder Woman Luke Cage The Punisher Ant Man Dead Girl Aquaman SCUD Spider Woman Blackbolt Martian Manhunter =====	<pre>\$ grep -i man heroes.txt</pre> <p>Cerca riga per riga, occorrenze di <b>man</b> (una <b>m</b>, seguita da una <b>a</b>, seguita da una <b>n</b>) all'interno del file heroes.txt, ignorando (-i) la differenza tra maiuscole/minuscole</p> <p>Restituisce tutte le righe in cui la ricerca ha successo</p>
---	--

Tramite il comando:

```
$ grep -i man heroes.txt
```

Stiamo cercando riga per riga occorrenze di man all'interno del file. Tramite il **parametro -i** stiamo dicendo di ignorare maiuscole e minuscole. Il comando restituirà tutte le righe in cui la ricerca ha avuto successo.

Possiamo utilizzare il **parametro -v** per escludere dal risultato tutte le righe che hanno successo lasciando solo quelle che non matchano con la ricerca.

Per abilitare l'utilizzo di espressioni regolari utiliziamo il **parametro -E**. Vediamo un esempio:

```
$ grep -E '^Bat' heroes.txt
```

In questo esempio l'espressione regolare è data dalla espressione tra virgolette. Il **carattere ^** indica di fare il match sulla stringa 'Bat' solo all'inizio della riga. Usiamo gli apici per isolare la regular expression e non interpretare ciò che è contenuto tra essi.

Rifacendoci all'esempio precedente, in output otterremmo:

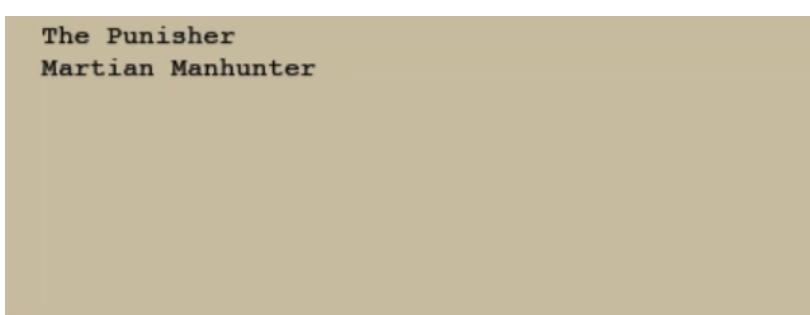


```
Batman
Batgirl
```

Quindi abbiamo implementato una ricerca per posizione dato che cercavamo solo stringhe all'inizio della riga. Si può fare la stessa cosa cercando alla fine della riga utilizzando il **carattere \$**. Vediamo come:

```
$ grep -E 'er$' heroes.txt
```

Il cui risultato sarà:



```
The Punisher
Martian Manhunter
```

Il match per caratteri si può fare utilizzando entrambi i caratteri \$ e ^. Ad esempio eseguendo il comando:

```
$ grep -E '^$' heroes.txt
```

Che cerca una riga vuota nel file heroes.txt. Ovviamente scrivere male l'espressione regolare può comportare il malfunzionamento di grep.

Possiamo usare l'**operatore |** per cercare una istanza oppure un'altra. Ad esempio:

```
$ grep -E -i '^^(bat|cat)' heroes.txt
```

Con questo comando stiamo dicendo di cercare tutte le righe che cominciano o con bat o con cat, ignorando maiuscole e minuscole. Il risultato sarà:

```
Catwoman  
Batman  
Batgirl
```

Con le parentesi [] possiamo raggruppare un insieme di caratteri che devono matchare. Per esempio:

```
$ grep -E '^BbCc' heroes.txt
```

Questo comando sta dicendo al comando di cercare una riga che comincia con B o b, C o c. Si può anche utilizzare il **carattere -** come delimitatore di intervallo:

```
$ grep -E '^A-Z' heroes.txt
```

Di questi operatori ce ne sono diversi, ne vediamo i principali:

- [A-Z] per i caratteri alfabetici maiuscoli.
- [a-z] per i caratteri alfabetici minuscoli.
- [A-Za-z] per i caratteri sia minuscoli che maiuscoli.
- [A-z0-9\_] come sopra più i numeri e l'underscore.
- [A-MXYZ] per tutti i caratteri maiuscoli tra A ed M, e anche X, Y e Z.

L'utilizzo del carattere ^ varia leggermente se lo andiamo a inserire all'interno delle parentesi quadre, capiamolo con un esempio:

```
$ grep -E -i '^b' heroes.txt
```

Questo comando va ad escludere dalla ricerca tutte quelle stringhe che hanno "at" preceduto da "b".

Possiamo utilizzare alcuni operatori per cercare caratteri ripetuti. Supponiamo di avere una lista di username e di volere individuare username al più di 8 caratteri, che iniziano con una lettere e contengono lettere o numeri. Una possibile espressione regolare per soddisfare questa richiesta sarebbe:

```
[a-z][a-z0-9][a-z0-9][a-z0-9][a-z0-9][a-z0-9][a-z0-9][a-z0-9]
```

Ma così andremmo a trovare tutti gli username con 8 caratteri, ma non quelli con meno. Allora possiamo utilizzare altri operatori come l'**operatore {}**, tramite il quale possiamo specificare il numero di volte in cui vogliamo che un carattere possa ripetersi. Quindi una operazione che possiamo scrivere è:

```
^[a-z][a-z0-9]{2,7}$
```

Così stiamo cercando una stringa che cominci con un carattere compreso tra A e z, seguita tra 2 e 7 volte da una lettera o un numero, prima della fine della riga.

In generale, la **stringa X{n, m}** fa il matching di almeno n e non più di m ripetizioni del carattere X. Omettendo m, l'espressione fa il matching di almeno n ripetizioni di X. Ad esempio:

```
^G[o]{2}gle$
```

Ci permette di trovare le stringhe "Google". Se invece scriviamo:

```
^G[o]{2,}gle$
```

Allora è come se stessimo cercando almeno due occorrenze del carattere "o", quindi troveremmo "Google", "Gooole" ecc..

Se invece utilizziamo {0,1} stiamo cercando sia "Gogle" che "Google". Allo stesso possiamo indicare almeno una occorrenza del carattere "o" con {1,}.

Esistono tantissimi operatori utilizzabili per le espressioni regolari, vediamo i principali:

.	<i>Match any single character.</i>
^	<i>Match the empty string that occurs at the beginning of a line or string.</i>
\$	<i>Match the empty string that occurs at the end of a line.</i>
A	<i>Match an uppercase letter A.</i>
a	<i>Match a lowercase a.</i>
\d	<i>Match any single digit.</i>
\D	<i>Match any single non-digit character.</i>
\w	<i>Match any single alphanumeric character; a synonym is [:alnum:].</i>
[A-E]	<i>Match any of uppercase A, B, C, D, or E.</i>
[ ^A-E ]	<i>Match any character except uppercase A, B, C, D, or E.</i>
X?	<i>Match no or one occurrence of the capital letter X.</i>
X*	<i>Match zero or more capital Xs.</i>
X+	<i>Match one or more capital Xs.</i>
X{ n }	<i>Match exactly n capital Xs.</i>
X{ n, m }	<i>Match at least n and no more than m capital Xs. If you omit m, the expression tries to match at least n Xs.</i>
(abc def)+	<i>Match a sequence of at least one abc and def; abc and def would match.</i>

Vediamo un esempio:

```
= demo.txt =
foo.txt
bar.txt
fool.txt
bar1.doc
foobar.txt
foo.doc
bar.doc
dataset.txt
purchase.db
purchase1.db
purchase2.db
purchase3.db
purchase.idx
foo2.txt
=====
```

```
$ grep 'purchase' demo.txt
```

Tutti i file contenuti all'interno di demo.txt il cui nome inizia con purchase

```
purchase.db
purchase1.db
purchase2.db
purchase3.db
purchase.idx
```

Un altro esempio:

```
= demo.txt =
foo.txt
bar.txt
fool.txt
bar1.doc
foobar.txt
foo.doc
bar.doc
dataset.txt
purchase.db
purchase1.db
purchase2.db
purchase3.db
purchase.idx
foo2.txt
=====
```

```
$ grep 'purchase..db' demo.txt
```

Tutti i file contenuti all'interno di demo.txt il cui nome inizia con purchase seguito da due qualsiasi altri caratteri(..) e poi da db

```
purchase1.db
purchase2.db
purchase3.db
```

```
= demo.txt =
foo.txt
bar.txt
fool.txt
bar1.doc
foobar.txt
foo.doc
bar.doc
dataset.txt
purchase.db
purchase1.db
purchase2.db
purchase3.db
purchase.idx
foo2.txt
=====
```

```
$ grep 'purchase.db' demo.txt
```

Tutti i file contenuti all'interno di demo.txt il cui nome inizia con purchase seguito da un **altro** qualsiasi carattere(.) e poi da db

```
purchase.db
```

## Lezione 16

Riprendiamo a parlare di scripting Linux.

Il carattere **backslash** \ funziona come carattere di escape nel grep, ciò significa che ciò che segue il carattere \ non sarà interpretato dal grep. Ad esempio:

```
= demo.txt =
foo.txt
bar.txt
fool.txt
bar1.doc
foobar.txt
foo.doc
bar.doc
dataset.txt
purchase.db
purchase1.db
purchase2.db
purchase3.db
purchase.idx
foo2.txt
=====
```

```
$ grep 'purchase.\.' demo.txt
```

Tutti i file contenuti all'interno di demo.txt il cui nome inizia con purchase seguito da un qualsiasi altro carattere e poi da un punto (il carattere \ funge da escape)

```
purchase1.db
purchase2.db
purchase3.db
```

Vediamo un altro esempio:

```
= demo.txt =
foo.txt
bar.txt
fool.txt
bar1.doc
foobar.txt
foo.doc
bar.doc
dataset.txt
purchase.db
purchase1.db
purchase2.db
purchase3.db
purchase.idx
foo2.txt
=====
```

```
$ grep -i -E '^purchase[0-9]*\.' demo.txt
```

Tutti i file contenuti all'interno di demo.txt il cui nome inizia con purchase seguito da zero o più occorrenze di un numero, e poi da un punto

```
purchase.db
purchase1.db
purchase2.db
purchase3.db
purchase.idx
```

In questo caso stiamo ignorando maiuscole e minuscole, stiamo cercando una qualsiasi espressione regolare (-E) che cominci con purchase seguito da 0-9 (opzionale per via del **carattere ?**) seguito da un punto.

In generale, il comportamento che abbiamo visto fino ad ora è quello di utilizzare grep per eseguire ricerca all'interno di file. Ma ciò si può fare anche con stream mediante l'uso della pipe. Per esempio, colendo usare una pipe, ricordando che la pipe nella shell linux è espressa dall'operatore |, l'esempio precedente diventerebbe:

```
= demo.txt =  
foo.txt  
bar.txt  
fool.txt  
bar1.doc  
foobar.txt  
foo.doc  
bar.doc  
dataset.txt  
purchase.db  
purchase1.db  
purchase2.db  
purchase3.db  
purchase.idx  
foo2.txt  
=====
```



```
$ grep -i -E '^purchase[0-9]?\.+' demo.txt
```

Tutti gli esempi precedenti possono essere riscritti utilizzando la pipe

```
$ cat demo.txt | grep -i -E '^purchase[0-9]?\.+'
```

Vediamo ora un riepilogo delle principali opzioni di grep:

- **-i**: ignore the case of your search term
- **-v**: show lines that *don't* match, instead of those that do
- **-c**: instead of returning matches, return the *number* of matches
- **-x**: return only an exact match
- **-E**: interpret search as an extended regular expression
- **-F**: interpret search as a list of fixed strings, including newlines, dots, etc
- **-f**: get the search patterns from this file
- **-H**: print the filename with each match
- **-m**: stop reading file after *n* number of matches
- **-n**: print the line number of where matches were found
- **-q**: don't output anything, but exit with status 0 if any match is found (check that status with `echo $?`).
- **-A**: print *n* number of lines after the match →
- **-B**: print *n* number of lines before the match
- **-o**: print only the matching part of the line
- **-e**: search literally, and protects patterns starting with a hyphen
- **-w**: find matches surrounded by space
- **--color**: add color to the matched output
- **--help**: get some help
- **-V**: get grep's version

Vediamo qualche altro esempio:

```
Stewart
Christina
- 441
<a href="https://www.google.com">Google</a>
Jill
54r4h
Shazbot123
111
221
Item 1, Item 2, Item 3
TABS    TABS    TABS
23.44.124.67
172.16.23.1
```

```
$ cat file.txt | grep -i jill
```

Cerca Jill all'interno di file.txt ignorando maiuscole/minuscole

Jill

In questo caso facciamo il cat del file.txt e mandare come stream l'output del file a grep che cercherà la stringa "jill" ignorando minuscole e maiuscole.

Possiamo anche cercare tutte le righe che contengono un numero utilizzando la notazione [:digit:] :

```
Stewart
Christina
- 441
<a href="https://www.google.com">Google</a>
Jill
54r4h
Shazbot123
111
221
Item 1, Item 2, Item 3
TABS    TABS    TABS
23.44.124.67
172.16.23.1
```

```
$ cat file.txt | grep -E '[[[:digit:]]]'
```

Restituisce tutte le righe che contengono un numero

```
- 441
54r4h
Shazbot123
221
Item 1, Item 2, Item 3
23.44.124.67
172.16.23.1
```

Se invece volessimo cercare tutto ciò che comincia con una cifra:

```
Stewart
Christina
- 441
<a href="https://www.google.com">Google</a>
Jill
54r4h
Shazbot123
111
221
Item 1, Item 2, Item 3
TABS TABS TABS
23.44.124.67
172.16.23.1
```

```
$ cat file.txt | grep -E '^[:digit:]'
```

Equivale a `cat file.txt | grep -E '^[0-9]'`  
Restituisce tutte le righe che iniziano con un numero:

```
54r4h
221
23.44.124.67
172.16.23.1
```

Immaginiamo ora di volere cercare sempre dal file precedente tutti le righe che rappresentano indirizzi IP:

```
Stewart
Christina
- 441
<a href="https://www.google.com">Google</a>
Jill
54r4h
Shazbot123
111
221
Item 1, Item 2, Item 3
TABS TABS TABS
23.44.124.67
172.16.23.1
```

```
$ cat file.txt | grep -n -E '[0-
9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-
9]{1,3}'
```

Restituisce tutte le righe che contengono un indirizzo IP  
e specifica il numero di riga

```
12:23.44.124.67
13:172.16.23.1
```

## AWK

**Awk** è un linguaggio di scripting che consente di fare le stesse cose che fa grep, ma a differenza di quest'ultimo può anche modificare i contenuti di file e stream, e non solo visualizzarli. La caratteristica di AWK è che oltre al pattern ci da la possibilità di specificare una "action" da eseguire una volta trovato il pattern che si cerca. Di default AWK quando processa una riga va a separare la riga in token individuati dal carattere spazio " ".

Ogni linea di input viene processata come se fosse formata da un insieme di campi separati da spazio, o da un campo separatore FS (field separator) che può essere specificato con il parametro -F (vedi sopra elenco opzioni).

Cominciamo a vedere qualche esempio di AWK:

```
CREDITS,EXPDATE,USER,GROUPS  
99,01 jun 2018,sylvain,team:::admin  
52,01 dec 2018,sonia,team  
52,01 dec 2018,sonia,team  
25,01 jan 2019,sonia,team  
10,01 jan 2019,sylvain,team:::admin  
8,12 jun 2018,öle,team:support
```

stampa tutte le righe

```
$ awk '1 {print}' file2.txt
```

1 = true; l'action print è anche l'action di default  
Il comando equivale quindi a: awk 1 file2.txt

```
CREDITS,EXPDATE,USER,GROUPS  
99,01 jun 2018,sylvain,team:::admin  
52,01 dec 2018,sonia,team  
52,01 dec 2018,sonia,team  
25,01 jan 2019,sonia,team  
10,01 jan 2019,sylvain,team:::admin  
8,12 jun 2018,öle,team:support
```

Notiamo che la prima riga del file contiene i nomi dei campi, mentre dalla seconda riga ci sono i valori che vanno a riempire le varie righe. Possiamo quindi usare AWK per cercare un valore (in questo caso 1, che si traduce nel "true" booleano, ovvero la condizione sempre verificata) e se viene soddisfatto esegue un action compresa tra parentesi graffe (in questo caso la print della riga). L'istruzione si traduce quindi in: per ogni riga verifica il pattern (in questo specifico esempio è sempre verificato) e stampalo. In questo caso il pattern viene eseguito sempre e quindi si ha, semplicemente, la stampa di tutte le righe.

Possiamo, volendo, accedere ad ogni singolo campo con caratteri preceduti dall'**operatore \$**:

```
CREDITS,EXPDATE,USER,GROUPS  
99,01 jun 2018,sylvain,team:::admin  
52,01 dec 2018,sonia,team  
52,01 dec 2018,sonia,team  
25,01 jan 2019,sonia,team  
10,01 jan 2019,sylvain,team:::admin  
8,12 jun 2018,öle,team:support
```

Per ciascuna riga, ciascun campo viene identificato con \$1, \$2, ... \$NF

L'intera riga corrisponde al campo \$0

```
awk '{print $0}' file2.txt
```

Stampa tutte le righe

```
CREDITS,EXPDATE,USER,GROUPS  
99,01 jun 2018,sylvain,team:::admin  
52,01 dec 2018,sonia,team  
52,01 dec 2018,sonia,team  
25,01 jan 2019,sonia,team  
10,01 jan 2019,sylvain,team:::admin  
8,12 jun 2018,öle,team:support
```

In questa maniera stiamo dicendo di stampare tutti i campi (per stampare tutti i campi di una riga basta specificare **\$0**) di tutte le righe del file e stamparli. Se invece eseguo il comando specificando \$1 stamperemo il primo campo (tranne nella prima riga dove non ci sono spazi che fanno da delimitatori). In generale esplicitando \$n staremo dicendo che vogliamo visualizzare l'n-esimo campo. Vediamo ora un eSEMPIO:

\$1

```
CREDITS,EXPDATE,USER,GROUPS
99,01 jun 2018,sylvain,team:::admin
52,01 dec 2018,sonia,team
52,01 dec 2018,sonia,team
25,01 jan 2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12 jun 2018,öle,team:support
```

Per ciascuna riga, ciascun campo viene identificato con \$1, \$2, ... \$NF

L'intera riga corrisponde al campo \$0

```
awk '{print $1}' file2.txt
```

Stampa il primo campo di tutte le righe

```
CREDITS,EXPDATE,USER,GROUPS
99,01
52,01
52,01
25,01
10,01
8,12
```

In questo caso abbiamo utilizzato il delimitatore di default (carattere spazio). Con l'opzione -F, seguita dal separatore, possiamo specificare il separatore che preferiamo. Vediamo un esempio dove il separatore dei campi è la virgola:

\$1

```
CREDITS,EXPDATE,USER,GROUPS
99,01 jun 2018,sylvain,team:::admin
52,01 dec 2018,sonia,team
52,01 dec 2018,sonia,team
25,01 jan 2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12 jun 2018,öle,team:support
```

```
awk -F , '{print $1}' file2.txt
```

Stampa per tutte le righe il primo campo delimitato dal nuovo carattere separatore ,

```
CREDITS
99
52
52
25
10
8
```

Ma negli esempi visti fino ad ora abbiamo sempre stampato un solo campo, o eventualmente tutti non specificando nulla. All'interno di print possiamo però combinare più campi da visualizzare, ad esempio:

**\$1**

**\$3**

```
CREDITS EXPDATE USER, GROUPS
99,01 jun 2018,sylvain,team:::admin
52,01 dec 2018,sonia,team
52,01 dec 2018,sonia,team
25,01 jan 2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12 jun 2018,öle,team:support
```

```
awk -F , '{print $1,$3}' file2.txt
```

Stampa per tutte le righe il primo ed il terzo campo  
delimitato dal carattere separatore ,

```
CREDITS USER
99 sylvain
52 sonia
52 sonia
25 sonia
10 sylvain
8 öle
```

Si può anche utilizzare **\$NF** per stampare l'ultimo campo di ciascuna delle righe:

**\$NF**

```
CREDITS,EXPDATE,USER, GROUPS
99,01 jun 2018,sylvain,team:::admin
52,01 dec 2018,sonia,team
52,01 dec 2018,sonia,team
25,01 jan 2019,sonia,team
10,01 jan 2019,sylvain,team:::admin
8,12 jun 2018,öle,team:support
```

```
awk -F , '{print $NF}' file2.txt
```

Stampa per tutte le righe l'ultimo campo delimitato dal  
carattere separatore ,

```
GROUPS
team:::admin
team
team
team
team:::admin
team:support
```

Un modo in cui possiamo utilizzare AWK è quello di fare l'analisi dell'output di un comando ls.  
Questo perché l'output di ls rispetta la struttura di dati che AWK può filtrare (*informazione spazio  
informazione spazio informazione ecc..*). AWK può essere utilizzato con una pipe per filtrare  
l'output di altri comandi, ad esempio il **comando ls -la**:

```
$ ls -la
total 40
drwxr-xr-x  7 user  staff  224 28 Apr  2019 .
drwxr-xr-x 10 user  staff  320 30 Apr  2019 ..
-rw-r--r--@  1 user  staff   153 26 Apr  2019 demo.txt
-rw-r--r--@  1 user  staff    77  6 Apr  2019 example.txt
-rw-r--r--@  1 user  staff   180 26 Apr  2019 file.txt
-rw-r--r--@  1 user  staff   209 26 Apr  2019 file2.txt
-rw-r--r--@  1 user  staff   173  6 Apr  2019 heroes.txt
```

Su questo risultato posso effettuare dei filtraggi con AWK sfruttando i delimitatori. Vediamo un esempio:

```
$ ls -la | awk '{print $1}'
total
drwxr-xr-x
drwxr-xr-x
-rw-r--r--@
-rw-r--r--@
-rw-r--r--@
-rw-r--r--@
-rw-r--r--@
```

### Espressioni regolari in AWK

Per specificare le espressioni regolari, andiamo a scrivere il comando AWK, con la seguente sintassi:

```
awk '/regex/' filename      #AWK con filtro dato da una espressione regolare
```

In questa maniera potremo trovare tutte le righe, che per esempio, sono relative directory:

```
total 40
drwxr-xr-x  7 user  staff  224 28 Apr  2019 .
drwxr-xr-x 10 user  staff  320 30 Apr  2019 ..
-rw-r--r--@  1 user  staff   153 26 Apr  2019 demo.txt
-rw-r--r--@  1 user  staff    77  6 Apr  2019 example.txt
-rw-r--r--@  1 user  staff   180 26 Apr  2019 file.txt
-rw-r--r--@  1 user  staff   209 26 Apr  2019 file2.txt
-rw-r--r--@  1 user  staff   173  6 Apr  2019 heroes.txt
```

**Se il pattern è individuato da una espressione regolare, si utilizza la sintassi**

**awk '/regex/' filename**

```
$ ls -la | awk '/^d/'
```

**Stampa tutti i campi (action di default) delle righe che iniziano con una d (le directory)**

```
drwxr-xr-x  7 user  staff  224 28 Apr  2019 .
drwxr-xr-x 10 user  staff  320 30 Apr  2019 ..
```

Nella stessa maniera possiamo trovare le righe che rappresentano file o directory:

```
total 40
drwxr-xr-x  7 user  staff  224 28 Apr  2019 .
drwxr-xr-x 10 user  staff  320 30 Apr  2019 ..
-rw-r--r--@  1 user  staff   153 26 Apr  2019 demo.txt
-rw-r--r--@  1 user  staff    77  6 Apr  2019 example.txt
-rw-r--r--@  1 user  staff   180 26 Apr  2019 file.txt
-rw-r--r--@  1 user  staff   209 26 Apr  2019 file2.txt
-rw-r--r--@  1 user  staff   173  6 Apr  2019 heroes.txt
```

```
$ ls -la | awk '/^[-d]/'
```

Stampa tutti i campi (action di default) delle righe che iniziano con – (file) o d (directory)

```
drwxr-xr-x  7 user  staff  224 28 Apr  2019 .
drwxr-xr-x 10 user  staff  320 30 Apr  2019 ..
-rw-r--r--@  1 user  staff   153 26 Apr  2019 demo.txt
-rw-r--r--@  1 user  staff    77  6 Apr  2019 example.txt
-rw-r--r--@  1 user  staff   180 26 Apr  2019 file.txt
-rw-r--r--@  1 user  staff   209 26 Apr  2019 file2.txt
-rw-r--r--@  1 user  staff   173  6 Apr  2019 heroes.txt
```

Se di questi risultati volessimo vedere i nomi dei file o delle directory faremmo:

```
total 40
drwxr-xr-x  7 user  staff  224 28 Apr  2019 .
drwxr-xr-x 10 user  staff  320 30 Apr  2019 ..
-rw-r--r--@  1 user  staff   153 26 Apr  2019 demo.txt
-rw-r--r--@  1 user  staff    77  6 Apr  2019 example.txt
-rw-r--r--@  1 user  staff   180 26 Apr  2019 file.txt
-rw-r--r--@  1 user  staff   209 26 Apr  2019 file2.txt
-rw-r--r--@  1 user  staff   173  6 Apr  2019 heroes.txt
```

```
$ ls -la | awk '/^[-]/ {print $9}'
```

Stampa il campo numero 9 (il filename) delle righe che iniziano con – (file)

```
demo.txt
example.txt
file.txt
file2.txt
heroes.txt
```

Vediamo però che in tutti questi esempi appena visti le espressioni regolari sono definite su tutto il file e non su campi specifici. Ciò però può essere modificato, infatti con AWK possiamo fare le ricerche dei pattern non solo sulla riga intera, ma di fare il match dei pattern solo su determinati campi da noi scelti. Ad esempio, nel 7imo campo del precedente esempio si fa riferimento al mese di modifica del file. Se volessimo cercare i file cercati ad aprile dovremmo fare:

```
$ ls -la | awk '$7 ~ /^A/ {print $9}'
```

Stampa il campo numero 9 (il filename) di tutte le righe  
il cui campo numero 7 (mese di modifica) inizia per A

```
.
```

```
..
```

```
demo.txt
```

```
example.txt
```

```
file.txt
```

```
file2.txt
```

```
heroes.txt
```

(NB: Il - dopo il 7 è in realtà una tilde ~)

Cerchiamo nel campo 7 tutto ciò che inizia con "A". Se lo troviamo stampiamo il nono campo (filename). Chiaramente questa operazione può essere effettuata su qualsiasi campo della riga, non solo sul settimo. Per fare la ricerca specifica quindi basta aggiungere **\$n ~** (con *n* numero del campo su cui voglio eseguire la ricerca).

Questa possibilità può essere allargata a più campi contemporaneamente utilizzando l'**operatore &&** (and) nella seguente maniera:

```
total 40
drwxr-xr-x  7 user  staff  224 28 Apr  2019 .
drwxr-xr-x 10 user  staff  320 30 Apr  2019 ..
-rw-r--r--@  1 user  staff   153 26 Apr  2019 demo.txt
-rw-r--r--@  1 user  staff    77  6 Apr  2019 example.txt
-rw-r--r--@  1 user  staff   180 26 Apr  2019 file.txt
-rw-r--r--@  1 user  staff   209 26 Apr  2019 file2.txt
-rw-r--r--@  1 user  staff   173  6 Apr  2019 heroes.txt
```

Possiamo applicare più  
espressioni regolari su diversi  
campi contemporaneamente

```
$ ls -la | awk '$7 ~ /^A/ && $1 ~ /^-/ {print $NF}'
```

Come l'esempio precedente, ma usiamo **\$NF** invece di  
**\$9** e stampiamo soltanto i file (non . e .. )

```
demo.txt
example.txt
file.txt
file2.txt
heroes.txt
```

(NB: vale lo stesso di sopra per il trattino che segue il 7)

### Creazione di Script

I comandi che abbiamo visto ci permettono di effettuare scripting in linux. In generale, abbiamo la possibilità di creare dei file eseguibili (di nome **script**) all'interno dei quali andiamo a scrivere istruzioni specifiche per eseguire alcune operazioni.

La prima cosa da fare per definire uno script interpretabile tramite bash è di inserire la seguente riga all'interno di ogni file script:

```
#!/bin/sh
```

Questa riga ci serve a specificare quale sarà l'interprete di AWK, in linux sarà il programma sh. La combinazione **#!** si chiama **SHEBANG**, o **HASHBANG**, ed è una sequenza di caratteri collocata sempre all'inizio di uno script, la quale indica al sistema quale interprete utilizzare per eseguire lo script (in questo caso, come detto, sh).

Vediamo un primo esempio di script:

```
#!/bin/sh  
clear  
echo "HELLO WORLD"
```

Questo piccolo script andrà inserito all'interno di un file di testo precedentemente creato. Supponiamo quindi di creare un file di nome *temp.sh* in cui andremo a scrivere il nostro script. Sorge un problema: di default, il file creato non avrà permessi di esecuzione come è possibile vedere grazie al comando ls -l:

```
$ ls -l temp.sh  
-rw-r--r-- 1 user staff 0 8 Apr 16:44 temp.sh
```

Dovremo dunque cambiare i permessi col comando **chmod** in maniera tale da potere dare permessi di esecuzione al file, il quale, altrimenti, non potrebbe essere eseguito:

```
$ chmod +x temp.sh  
$ ls -l temp.sh  
-rwxr-xr-x 1 user staff 0 8 Apr 16:44 temp.sh
```

Infine, per eseguirlo dal bash dobbiamo invocare lo script nella seguente maniera, se ci troviamo già all'interno della directory:

```
./yourscript.sh
```

Se non ci troviamo nella directory dello script, per eseguire lo script, dovremo puntare alla directory nella quale si trova lo script mediante il suo path assoluto:

```
$/path/to/yourscript.sh
```

## Parametri degli script

Possiamo passare degli argomenti di input agli script esattamente come facciamo con i parametri di qualsiasi linguaggio. Quando eseguiamo lo script, oltre al nome dello script, possiamo passare alcuni valori a cui potremo accedere (all'interno dello script) mediante una sintassi ben precisa:

<b>\$#</b>	<b>Numero di argomenti</b>
<b>\$*, \$@</b>	<b>Tutti gli argomenti passati alla shell</b>
<b>\$1, \$2, ...</b>	<b>Argomento di posto 1, 2, ...</b>
<b>\$0</b>	<b>Argomento di posto 0, nome del comando stesso</b>
<b>\$-</b>	<b>Opzioni passate alla shell</b>
<b>\$?</b>	<b>L'ultimo comando eseguito</b>
<b>\$\$</b>	<b>PID della shell</b>

Ad esempio, scrivendo il seguente script ed eseguendolo come vediamo nella 3a riga:

```
#!/bin/bash
echo "num argomenti di input: $#"

$ ./s1.sh a b c d
```

Oterremo in output il numero di parametri passati allo script:

```
gabriele@gabriele-Virtual-Machine:~/Desktop/Operative System$ ./temp a b c d
num argomenti di input: 4
```

Se invece utilizzassimo \$@ otterremmo la lista dei parametri passati:

```
gabriele@gabriele-Virtual-Machine:~/Desktop/Operative System$ ./temp a b c d
num argomenti di input: a b c d
gabriele@gabriele-Virtual-Machine:~/Desktop/Operative System$
```

E così via.

### Ciclo for (scripting)

Vediamo ora come creare un ciclo, utilizzando ciò che abbiamo imparato. Vediamo come scrivere un **ciclo for**:

```
#!/bin/sh
for arg in "$@"
do
    echo "$arg"
done

$./cicloFor.sh a b c d
```

Il corpo del for viene individuato dalle due **parole chiave do e done**. Il risultato sarà:

```
gabriele@gabriele-Virtual-Machine:~/Desktop/Operative System$ ./temp a b c d
a
b
c
d
gabriele@gabriele-Virtual-Machine:~/Desktop/Operative System$
```

Possiamo usare gli argomenti in input per definire filtri parametrici. Prendiamo l'esempio seguente visto prima:

```
$ ls -la | awk '$6 ~ /[^a]/ {print $9}'
```

Che mediante script può essere scritto così:

```
#!/bin/bash
`ls -la | awk -v arg="$1" '$6 ~ "^. "$arg {print $9}'`
```

arg sarà la nostra variabile che indica il primo parametro di input (\$1). Di seguito scriveremo l'istruzione vista prima aggiungendo arg come parametro della espressione regolare. Il carattere ^ (inizio stringa) viene concatenato ad arg utilizzando " ".

### Costrutto If (scripting)

Possiamo implementare anche il **costrutto if**. Nel caso di stringhe abbiamo diversi modi di fare il confronto, questa tabella riassume i modi in cui si può effettuare il **confronto tra stringhe**:

String Comparison	Description
Str1 = Str2	Returns true if the strings are equal
Str1 != Str2	Returns true if the strings are not equal
-n Str1	Returns true if the string is not null
-z Str1	Returns true if the string is null

Ma possiamo effettuare **confronti** anche **tra numeri**:

Numeric Comparison	Description
expr1 -eq expr2	Returns true if the expressions are equal
expr1 -ne expr2	Returns true if the expressions are not equal
expr1 -gt expr2	Returns true if expr1 is greater than expr2
expr1 -ge expr2	Returns true if expr1 is greater than or equal to expr2
expr1 -lt expr2	Returns true if expr1 is less than expr2
expr1 -le expr2	Returns true if expr1 is less than or equal to expr2
! expr1	Negates the result of the expression

Una ulteriore verifica che possiamo fare è sulla tipologia di elementi che stiamo analizzando (**confronto di elementi**):

File Conditionals	Description
-d file	True if the file is a directory
-e file	True if the file exists (note that this is not particularly portable, thus -f is generally used)
-f file	True if the provided string is a file
-g file	True if the group id is set on a file
-r file	True if the file is readable
-s file	True if the file has a non-zero size
-u	True if the user id is set on a file
-w	True if the file is writable
-x	True if the file is an executable

Vediamo ora un esempio di costrutto if che serve a verificare il numero di argomenti di input che riceviamo:

```
#!/bin/bash

if [ $# -eq 0 ]
then
    echo "Specificare un parametro in input"
elif [ $# -gt 1 ]
then
    echo "Troppi parametri di input"
else
    ls -la | awk -v arg="$1" '$6 ~ arg {print $9}'
fi
```

### Costrutto Case (scripting)

È possibile anche implementare il costrutto **switch case** come segue:

## Costrutto CASE

```
case expression in
    pattern1 )
        statements ;;
    pattern2 )
        statements ;;
    ...
esac
```

Un esempio di utilizzo è quello di valutare dei flag che vengono passati allo script e di stampare a schermo una stringa in base a ciò che si è ricevuto in input:

```
#!/bin/bash

case $1 in
    -a | -A | --alpha )
        echo "alpha";;
    -b )
        echo "bravo";;
    -c )
        echo "charlie";;
    * )
        echo "opzione sconosciuta";;
esac
```

## Ciclo While (scripting)

Vediamo quale è la sintassi del **ciclo while**, il quale, come il for, è identificato dalle parole chiave **do** e **done**:

```
while conditionlist do
    lista comandi
done
```

Vediamo un esempio sul calcolo del fattoriale:

```
#!/bin/bash

counter=$1
factorial=1
while [ $counter -gt 0 ] do
    factorial=$(( $factorial * $counter ))
    counter=$(( $counter - 1 ))
done
echo $factorial
```

# Lezione 17

---

## Mutua esclusione ed esecuzione concorrente

Riprendiamo il concetto di **esecuzione concorrente**. Nelle precedenti lezioni avevamo detto che ogni processo è una istanza di un programma in esecuzione, ognuno dei quali ha il proprio contesto di esecuzione. Il problema della **esecuzione concorrente** riguarda il riuscire ad alternare i processi nell'uso del processore. Lo stesso vale per i thread, anche se il concetto di thread abbiamo già visto essere diverso da quello di processo.

In particolare, ricordiamo che quando in un sistema esistono più thread allo stesso tempo allora tali thread si diranno concorrenti. I thread concorrenti possono cooperare tra loro o lavorare in maniera del tutto indipendente.

Dato che i thread permettono la esecuzione concorrente (pur condividendo lo stesso spazio di memoria) è necessario prevedere dei meccanismi che permettono ai thread di cooperare per riuscire a svolgere alcuni task. Uno dei problemi principali legati alla cooperazione tra i thread è che essendoci molte risorse condivise bisogna mantenere lo stato di quella risorsa coerente, ovvero aggiornato e privo di errori.

Il problema che si vuole risolvere è quello della mutua esclusione, infatti soltanto un thread per volta deve potere accedere a quella risorsa. Questo problema si ripresenta nella vita reale, ad esempio, nell'acquisto di un biglietto aereo tramite un programma online, in questo caso la risorsa che vogliamo ottenere è un posto nell'aereo. Il problema sorge quando più persone contemporaneamente si interessano alla stessa risorsa (posto in aereo); se qualcuno di questi utenti decide di fare l'acquisto, dobbiamo garantire che nel momento di questa modifica, nessun'altro utente faccia la stessa operazione. Questo è un esempio concreto di accesso concorrente alle risorse.

Nel calcolatore la risorsa che vogliamo proteggere può essere, ad esempio, una struttura dati, e gli utenti saranno i thread. Dobbiamo dunque prevedere una serie di controlli che impediscano ai thread di modificare contemporaneamente quella risorsa. Questi controlli vanno fatti in maniera efficiente. L'implementazione della mutua esclusione deve essere fatta in maniera trasparente rispetto ai processi e con basso overhead, ed inoltre deve essere garantita in ogni circostanza.

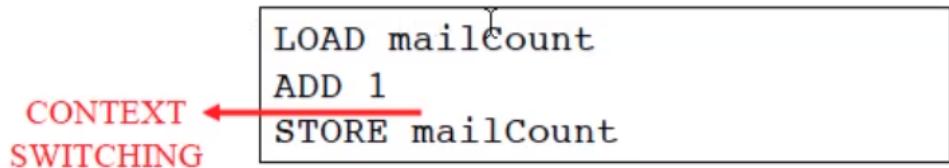
Passando ad un esempio più vicino a noi: immaginiamo un mail server con il processo principale che si occupa di gestire la mail tramite istanze di molti thread concorrenti. Ogni volta che si riceve una mail uno di questi thread deve aumentare una variabile condivisa, se avessimo un singolo processo che gestisce la mail, gestire il contatore sarebbe abbastanza facile. Se questa operazione viene fatta da  $n$  thread diversi dobbiamo fare comunque in modo che ciascun thread lavori sulla stessa variabile contatore condivisa, quindi il totale di mail ricevute è una informazione che il mail server possiede e viene aggiornata in maniera concorrente da più thread.

Il problema sorge quando più thread cercano di aggiornare il contatore contemporaneamente, in questo caso si potrebbero avere degli aggiornamenti sbagliati.

Se volessimo immaginare l'operazione in assembly sarebbe più o meno così:

```
LOAD mailCount  
ADD 1  
STORE mailCount
```

Ipotizziamo una situazione critica come la seguente:



Qui, prima che il thread 1 possa aggiornare il contatore (STORE mailCount), verrà eseguito il context switching, allora il secondo thread verrà caricato e farà la stessa operazione. Il problema è che alla fine di queste operazioni il contatore verrà aumentato solo di 1 invece che di 2, per via del context switching con il secondo thread.

Il problema dell'accesso concorrente alle risorse deriva dal fatto che i dati possono finire in uno stato che non è consistente, ovvero che lo stato non è corretto o aggiornato. Ciò può dipendere dal context switching e per evitare che avvengano problemi simili bisogna implementare l'accesso alle risorse in maniera **mutuamente esclusiva**: significa che un solo thread per volta può accedere ad una determinata risorsa e solo dopo che quest'ultima risorsa sarà sbloccata dal thread stesso potrà essere utilizzata da un altro thread.

Ciò avviene serializzando i thread cercando di mantenere un basso overhead e un basso tempo di attesa, sennò i vantaggi si perderebbero e non avrebbe senso implementarla.

In generale in tutti gli esempi che vedremo di mutua esclusione si andranno a ricondurre ad una relazione nota come la **relazione del Produttore/Consumatore**. Possiamo descrivere questa relazione col seguente scenario:

1. Un thread (o eventualmente più thread) scrive dei dati da immagazzinare in una risorsa condivisa. Questo thread è il **produttore**.
2. Un secondo thread (o eventualmente più thread) legge i dati dalla risorsa condivisa. Questo thread è il **consumatore**.

Con questo tipo di modello è facile ottenere dei dati non corretti, per ottenere un aggiornamento corretto dei valori e una corretta mutua esclusione sarà necessario implementare la **sincronizzazione** tra i thread. Come primo esempio vediamo un problema Java in cui abbiamo un buffer (una singola variabile nel caso nostro) in cui abbiamo un thread produttore che genera numeri e li memorizza nel buffer e un thread consumatore che legge i dati da questo buffer. Questo buffer permette quindi una sorta di comunicazione tra i due thread.

### Esempio del buffer non sincronizzato

Nelle precedenti lezioni abbiamo visto che la pipe implementa di default la sincronizzazione. Scendendo nel dettaglio, quando creiamo la pipe, creiamo anche i due processi, uno per leggere e uno per scrivere. I due processi vengono però schedulati in un ordine che noi non conosciamo, capita infatti che il processo lettore viene eseguito prima dello scrittore, nonostante ciò l'esecuzione è corretta dato che viene gestita in maniera sincronizzata automaticamente dalla pipe. Questo è un tipico esempio di produttore consumatore che riusciamo a gestire con la pipe in maniera sincronizzata. Ma questo non avviene sempre, infatti non sempre utilizzeremo la pipe.

Vediamo un attimo nel dettaglio l'esempio del buffer non sincronizzato per capire come modellare il problema (in particolare dobbiamo capire come riconoscere thread e risorse):

```

// SharedBufferTest creates producer and consumer threads.

public class SharedBufferTest
{
    +
    public static void main( String [] args )
    {
        // create shared object used by threads
        Buffer sharedLocation = new UnSyncBuffer();

        // create producer and consumer objects
        Producer producer = new Producer( sharedLocation );
        Consumer consumer = new Consumer( sharedLocation );

        producer.start(); // start producer thread
        consumer.start(); // start consumer thread
    } // end main

} // end class SharedBufferTest

```

Questa prima classe SharedBufferTest è la classe che contiene il main. Nel main stiamo creando un oggetto sharedLocation di classe Buffer, nello specifico sarà di tipo UnSyncBuffer. Dopo creo un oggetto producer e un consumer, passando ad entrambi la variabile condivisa (sharedLocation). Dopodiché avvio sia producer che consumer. Ricapitolando abbiamo: 1 variabile condivisa (sharedLocation) e 3 thread (consumatore, produttore e main).

Vediamo ora il buffer, che implementiamo come interfaccia:

```

1 // Fig. 5.1: Buffer.java
2 // Buffer interface specifies methods to access buffer data.
3
4 public interface Buffer
5 {
6     public void set( int value ); // place value into Buffer
7     public int get();           // return value from Buffer
8 }

```

La funzione set scrive l'intero all'interno del buffer, mentre get lo legge dal buffer. Utilizziamo l'interfaccia buffer perché quando implemetteremo la sincronizzazione ci servirà fare l'overrun dei metodi get e set. In questo specifico esempio non serve, ma la implementiamo comunque. Vediamo ora la classe UnSyncBuffer:

```

// UnsynchronizedBuffer represents a single shared integer.

public class UnSyncBuffer implements Buffer
{
    private int buffer = -1; // shared by Producer and Consumer

    // place value into buffer
    public void set( int value )
    {
        System.out.println( Thread.currentThread().getName() +
            " writes " + value );
        buffer = value;
    } // end method set

    // return value from buffer
    public int get()
    {
        System.out.println( Thread.currentThread().getName() +
            " reads " + buffer );

        return buffer;
    } // end method get
} // end class UnsynchronizedBuffer

```

All'inizio della classe viene inizializzato il buffer. Dopo definiamo i metodi set e get.

A questo punto devo pensare alla implementazione dei due thread. Vediamo prima il producer:

```

public class Producer extends Thread
{
    private Buffer sharedLocation; // reference to shared object

    // Producer constructor
    public Producer( Buffer shared )
    {
        super( "Producer" ); // create thread named "Producer"
        sharedLocation = shared; // initialize sharedLocation
    } // end Producer constructor

    // Producer method run stores values
    //      in Buffer sharedLocation
    public void run()
    {
        for ( int count = 11; count <= 20; count++ )
        {
            // sleep 0 to 3 seconds, then place value in Buffer
            try
            {
                Thread.sleep( ( int )( Math.random() * 3000 ) );
                sharedLocation.set( count ); // write to the buffer
            } // end try

            // if sleeping thread interrupted, print stack trace
            catch ( InterruptedException exception )
            {
                exception.printStackTrace();
            } // end catch
        } // end for

        System.out.println( getName() + " done producing." +
            "\nTerminating " + getName() + ". " );
    } // end method run
} // end class Producer

```

Questi thread hanno un riferimento alla stessa variabile condivisa. Poi c'è il classico metodo run, nel caso del produttore, ad intervalli di massimo 3 secondi, il thread andrà a scrivere nel buffer il valore count. Inizialmente scriveremo il valore 11, dopo circa 3 secondi scriveremo il valore 12 e così via.. quando esce dal for, il produttore stampa il suo nome e dice di aver completato l'inserimento di valori e può terminare. Notiamo che lo sleep dentro il for serve solo

esclusivamente a non avere una stampa dei valori eccessivamente veloce nel terminale e non ha nulla a che fare con la mutua esclusione né il parallelismo.

Vediamo ora il consumatore:

```
public class Consumer extends Thread
{
    private Buffer sharedLocation; // reference to shared object

    // Consumer constructor
    public Consumer( Buffer shared )
    {
        super( "Consumer" ); // create thread named "Consumer"
        sharedLocation = shared; // initialize sharedLocation
    } // end Consumer constructor

    // read sharedLocation's value four times and sum the values
    public void run()
    {
        int sum = 0;

        // alternate between sleeping and getting Buffer value
        for ( int count = 1; count <= 10; ++count )
        {
            // sleep 0-3 seconds, read Buffer value and add to sum
            try
            {
                Thread.sleep( ( int ) ( Math.random() * 3001 ) );
                sum += sharedLocation.get();
            }

            // if sleeping thread interrupted, print stack trace
            catch ( InterruptedException exception )
            {
                exception.printStackTrace();
            }
        } // end for

        System.err.println( getName() + " read values totaling: "
            + sum + ".\nTerminating " + getName() + ".");
    } // end method run
} // end class Consumer
```

Il consumatore gestisce un contatore che va da 1 a 10 per gestire la lettura, ciò che vorremmo è che tutte le letture corrispondessero a tutte le scritture (quindi senza salti di valori o ripetizioni). Utilizziamo una variabile di nome sum per capire se effettivamente abbiamo letto tutti i valori infatti se tutto va bene dovremo leggere alla fine  $sum = 11 + 12 + \dots + 20$ .

Dopo aver fatto le 10 letture il consumatore termina, a prescindere dalla correttezza delle letture.

Proviamo ad eseguire l'esempio:

```

java SharedBufferTest
Consumer reads -1
Producer writes 11
Consumer reads 11
Producer writes 12
Consumer reads 12
Consumer reads 12
Consumer reads 12
Producer writes 13
Consumer reads 13
Producer writes 14
Consumer reads 14
Producer writes 15
Consumer reads 15
Producer writes 16
Consumer reads 16
Consumer reads 16
Consumer read values totaling: 120.
Terminating Consumer.
Producer writes 17
Producer writes 18
Producer writes 19      +
Producer writes 20
Producer done producing.

```

Per primo viene schedulato il consumer che infatti legge il valore -1 dato che il producer non ha ancora scritto nulla. Dopodiché verrà schedulato il producer ecc. Notiamo che poi avremo la schedulazione del consumer per due esecuzioni di fila producendo delle letture scorrette. Dopo il produttore scrive il valore 13 e ricomincia l'alternanza tra producer e consumer. Dopo un po' il consumer termina perché avrà letto 10 valori (seppur scorretti) ma il producer continuerà a scrivere.

Questo esempio ci fa capire dunque per quale motivo nasce la necessità fondamentale di avere delle operazioni sincronizzate

### Sezioni critiche

In generale, il codice che andiamo a scrivere, è diviso in sezioni logiche. Le parti sensibili del codice in cui si esegue l'accesso a risorse condivise prendono il nome di **critical sections**. Solo questi pezzi di codice hanno la necessità di essere protetti per evitare errori di sincronizzazione.

Soltanto un thread per volta deve infatti potere accedere alla critical section dato che questo comporta la modifica di una risorsa condivisa.

Per gestire l'accesso alle critical sections vengono definite due primitive (che sono anche due metodi di Java):

- **WAIT**: aspetta che un thread sia uscito dalla sua critical section.
- **NOTIFY**: se un thread completa la sua critical section notifica gli altri thread che sta lasciando la critical section.

Queste due primitive devono essere implementate in maniera tale che non si creino dei loop infiniti e che non accada mai che un thread nella critical section vada in wait bloccando completamente l'esecuzione. Inoltre, se un thread in sezione critica termina, il sistema operativo, svolgendo le operazioni di **gestione della terminazione**, deve anche occuparsi di rilasciare la mutua esclusione in modo che gli altri thread possano entrare nelle proprie sezioni critiche.

Nel caso del **produttore**:

- WAIT: il produttore scrive un valore e prima di scrivere il valore successivo fa il wait aspettando la lettura del consumatore.
- NOTIFY: il produttore notifica al consumatore che ha scritto qualcosa nel buffer.

Nel caso del **consumatore**:

- WAIT: il consumatore aspetta che il produttore abbia scritto un valore nel buffer.
- NOTIFY: il consumatore notifica al produttore che ha letto il valore nel buffer.

Riassumendo il meccanismo: inizialmente il produttore sa che deve andare in wait soltanto se il consumatore non ha ancora letto il valore. Il consumatore andrà in wait finché il produttore non avrà scritto un valore.

Quando parte l'esecuzione il produttore scrive nel buffer, va in wait e lo notifica al consumatore. Il consumatore a questo punto si sveglia, legge il valore, va in wait e notifica al produttore (che si sveglierà). E così via..

In java:

- **wait()** è un segnale che sposta un thread dallo stato di running allo stato di waiting.
- **notify()** è un segnale che sposta un thread dallo stato di waiting allo stato di ready e lo rende pronto per una successiva esecuzione.

### Esempio del mail server sincronizzato

A livello logico la critical section si individua da due primitive che ne delimitano inizio e fine, chiamate **primitive di mutua esclusione**:

- **enterMutualExclusion**
- **exitMutualExclusion**

Ciò che si trova tra queste due primitive rappresenta il contenuto della critical section.

Vediamo ora l'eSEMPIO DEL MAIL SERVER (fatto all'inizio della lezione) implementando la sincronizzazione:

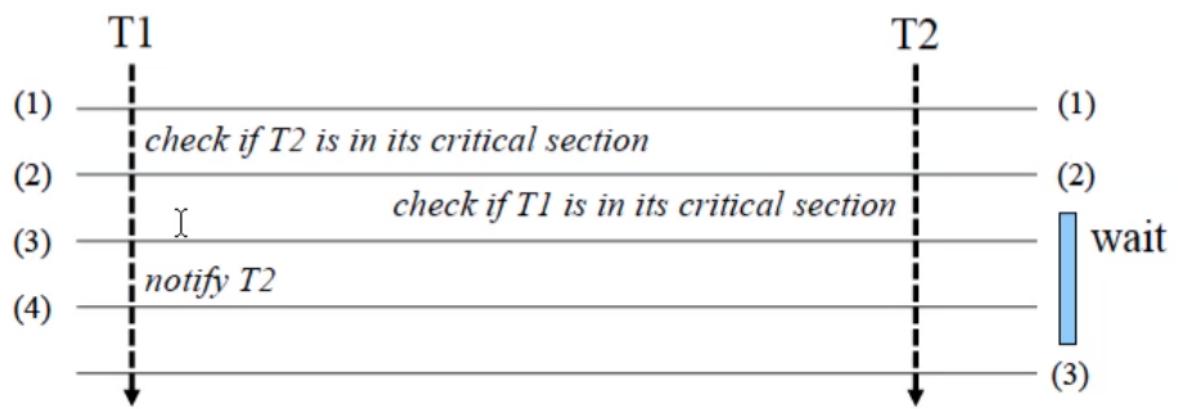
```
while (true) {
    1.   Receive e-mail           // executing outside critical section
    2.   enterMutualExclusion()  // want to enter critical section
    3.   Increment mailcount    // executing inside critical section
    4.   exitMutualExclusion()   // leaving critical section
}
```

Ciò che stiamo proteggendo in questo codice è l'incremento della risorsa condivisa, in questo caso la variabile contatore mailcount. Ricordiamo infatti che ciò che ci interessa proteggere è solo ciò che riguarda le risorse condivise.

Per potere gestire la sincronizzazione della critical section utilizziamo due primitive che hanno lo scopo di comunicare agli altri thread quando un thread sta entrando nella sezione critica e quando sta uscendo.

Ipotizzando che il nostro mail server utilizzi due thread, T1 e T2, e che T1 abbia ricevuto una mail. Allora T1, prima di entrare nella sezione critica per aggiornare la variabile contatore, dovrà verificare che T2 non sia anch'esso nella critical section. Una volta entrato nella critical section, T1 potrà aggiornare il contatore. Supponiamo ora che, proprio mentre T1 è nella sezione critica avvenga il context switching a T2. Ma T2 controllando lo stato di T1 vedrà che T1 è nella sezione critica, di conseguenza andrà nello stato di waiting e il controllo tornerà a T1 che eseguirà l'istruzione 4 uscendo dalla critical section e farà il notify a T2. T2 ricevendo il notify andrà in esecuzione ed entrerà nella sezione critica eseguendo l'istruzione 3 (ovvero l'incremento del mailCount).

Vediamolo graficamente:



# LEZIONE 17 - 06/05/2020

## CAPITOLO 5 - PT2

### IMPLEMENTARE LE PRIMITIVE DELLA MUTUA ESCLUSIONE

Ognuna delle prime soluzioni per la mutua esclusione che discuteremo fornisce una implementazione di `enterMutualExclusion()` e di `exitMutualExclusion()` con le seguenti proprietà:

1. La soluzione è implementata puramente in software su una macchina senza specifiche istruzioni per la mutua esclusione in linguaggio macchina. Ogni istruzione in linguaggio macchina è eseguita **indivisibilmente** – se inizia, viene completata senza interruzioni.
2. Non può essere fatta **nessuna assunzione sulla velocità** relativa dei thread in esecuzione concorrente asincrona. Qualunque soluzione deve tenere conto del fatto che il thread possa essere rimosso dal processo o riprendere l'esecuzione in un qualunque momento e che il tasso di esecuzione di ogni thread potrebbe non essere consistente o predicibile.
3. **Un thread che sta eseguendo istruzioni al di fuori dalla sua sezione critica non può prevenire altri thread** dall'entrare nella loro sezione critica.
4. Un thread non deve essere **indefinitamente posticipato** dall'entrare nella sua sezione critica.

### SOLUZIONI SOFTWARE AL PROBLEMA DELLA MUTUA ESCLUSIONE

Una prima implementazione software elegante della mutua esclusione fu presentata dal matematico olandese Dekker. Affronteremo lo sviluppo di Dijkstra dell'algoritmo di Dekker, una implementazione della mutua esclusione per due thread.

# Algoritmo di Dekker

La **prima versione** dell'**algoritmo di Dekker** è il seguente

```
VERSIONE 1

1 System:
2
3 int threadNumber = 1;
4
5 startThreads(); // initialize and launch threads
6
7 Thread T1:
8
9 void main() {
10
11     while ( !done )
12     {
13         while ( threadNumber == 2 );
14             // enterMutualExclusion
15
16             // critical section code
17
18             threadNumber = 2; // exitMutualExclusion
19
20             // code outside critical section
21     } // end outer while
22
23 } // end Thread T1
24

25 Thread T2:
26
27 void main() {
28
29     while ( !done )
30     {
31         while ( threadNumber == 1 );
32             // enterMutualExclusion
33
34             // critical section code
35
36             threadNumber = 1; // exitMutualExclusion
37
38             // code outside critical section
39     } // end outer while
40
41 } // end Thread T2
```

In questo algoritmo viene definita una variabile **threadNumber** e viene settata inizialmente ad 1. Successivamente vengono definiti i due thread, senza fare caso al potenziale ordine di schedulazione. In entrambi i thread vi è un **primo loop infinito** al cui interno vi è un ulteriore loop in cui nel primo thread abbiamo la condizione `threadNumber == 2`, e nel secondo abbiamo `threadNumber == 1`.

Nel primo thread, una volta terminato il codice della critical section cede il controllo al secondo thread uscendo dal loop settando `threadNumber = 2`.

| N.B. ci può essere ovviamente del codice eseguibile anche fuori dalla critical section

Il secondo thread adesso entra nel loop più interno, esegue il codice della propria critical section, alla fine della quale setta nuovamente `threadNumber = 1`, cedendo il controllo al primo thread e così via..

La caratteristica fondamentale di questa prima versione è che innanzitutto riesce ad implementare la mutua esclusione, inoltre utilizza una variabile per determinare quale thread essere eseguito. Vi sono però due problemi, il primo è noto come **busy waiting**: il thread usa il processore per non eseguire praticamente nessun lavoro (per controllare ripetutamente il valore di `threadNumber`); il secondo problema è che se un thread ha bisogno di entrare nella sua sezione critica più frequentemente dell'altro, il thread più veloce sarà vincolato ad operare alla velocità di quello più lento (questo è chiamato problema della **lockstep synchronization**).

La **seconda versione** è definita come segue:

```
VERSIONE 2

1 System:
2
3 boolean t1Inside = false;
4 boolean t2Inside = false;
5
6 startThreads(); // initialize and launch threads
7
8 Thread T1:
9
10 void main() {
11
12     while ( !done ) {
13
14         while ( t2Inside );
15             // enterMutualExclusion
16
17         t1Inside = true; // enterMutualExclusion
18
19         // critical section code
20
21         t1Inside = false; // exitMutualExclusion
22
23         // code outside critical section
24     } // end outer while
25
26 } // end Thread T1
27

28 Thread T2:
29
30 void main() {
31
32     while ( !done ) {
33
34         while ( t1Inside );
35             // enterMutualExclusion
36
37         t2Inside = true; // enterMutualExclusion
38
39         // critical section code
40
41         t2Inside = false; // exitMutualExclusion
42
43         // code outside critical section
44     } // end outer while
45
46 } // end Thread T2
```

Una prima differenza consiste nel **numero di variabili**: in questo caso abbiamo due booleani, che indicano se il rispettivo thread si trova o meno nella critical section. Inizialmente saranno entrambi settati a false. Il controllo per il loop più interno consisterà nel check della zona critica sull'altro thread (primo thread check su t2 e viceversa). Utilizzando due variabile per gestire l'accesso alle sezioni critiche (una per ogni thread), siamo in grado di eliminare la lockstep synchronization.

T1 sarà in grado di entrare continuamente nella sua sezione critica, tutte le volte che sarà necessario, mentre t2Inside è false. Inoltre, se T1 entra nella sua sezione critica e imposta t1Inside a true, allora T2 entrerà in busy waiting. Infine, T1 terminerà la sua sezione critica ed eseguirà il suo codice per uscire dalla mutua esclusione, impostando t1Inside a false.

Nonostante questa soluzione elimini il problema della lockstep synchronization, sfortunatamente **non garantisce la mutua esclusione**: un thread può essere anticipato mentre si aggiorna il valore del flag.

Introduciamo quindi la **terza versione**:

```
VERSIONE 3

1 System:
2
3 boolean t1WantsToEnter = false;
4 boolean t2WantsToEnter = false;
5
6 startThreads(); // initialize and launch threads
7
8 Thread T1:
9
10 void main()
11 {
12     while ( !done )
13     {
14         t1WantsToEnter = true;
15         // enterMutualExclusion
16         while ( t2WantsToEnter );
17         // enterMutualExclusion
18         // critical section code
19
20         t1WantsToEnter = false;
21         // exitMutualExclusion
22         // code outside critical section
23
24     } // end outer while
25
26 } // end Thread T1
27
28 Thread T2:
29
30 void main()
31 {
32     while ( !done )
33     {
34         t2WantsToEnter = true;
35         // enterMutualExclusion
36         while ( t1WantsToEnter );
37         // enterMutualExclusion
38         // critical section code
39
40         t2WantsToEnter = false;
41         // exitMutualExclusion
42         // code outside critical section
43
44     } // end outer while
45
46 } // end Thread T2
```

L'idea alla base è che una volta che un thread prova ad eseguire il ciclo while, deve essere assicurato che l'altro non possa superare il proprio. Si cerca di risolvere utilizzando due variabili che rappresentano la **volontà** di entrare nella critical section. Quindi, T1 indica di voler entrare nella sua sezione critica impostando t1WantsToEnter a true. Se t2WantsToEnter è false, allora T1 entra nella sua sezione critica e previene T2 dal farlo. Quindi la **mutua esclusione è garantita**, e sembra che siamo arrivati ad una soluzione corretta.

Se ogni thread imposta il suo flag prima di procedere al ciclo while, allora ogni thread troverà l'altro flag impostato ed eseguirà il proprio ciclo while per sempre. Questo è un esempio di due processi che vanno in **deadlock**.

Il deadlock è un blocco irrisolvibile in cui i thread attendono una condizione che non si avvererà mai

Un'alternativa è quella della **versione quattro**:

```
VERSIONE 4

1 System:
2
3 boolean t1WantsToEnter = false;
4 boolean t2WantsToEnter = false;
5
6 startThreads(); // initialize and launch threads
7
8 Thread T1:
9
10 void main()
11 {
12     while ( !done )
13     {
14         t1WantsToEnter = true;
15         // enterMutualExclusion
16         while ( t2WantsToEnter )
17             // enterMutualExclusion
18         {
19             t1WantsToEnter = false;
20             // enterMutualExclusion
21             // wait for random time
22             t1WantsToEnter = true;
23         } // end while
24         // critical section code
25         t1WantsToEnter = false;
26         // exitMutualExclusion
27         // code outside critical section
28         // code outside critical section
29     } // end outer while
30 } // end Thread T1
31
32 } // end Thread T2
33
34
35 Thread T2:
36
37 void main()
38 {
39     while ( !done )
40     {
41         t2WantsToEnter = true;
42         // enterMutualExclusion
43         while ( t1WantsToEnter )
44             // enterMutualExclusion
45         {
46             t2WantsToEnter = false;
47             // enterMutualExclusion
48             // wait for random time
49             t2WantsToEnter = true;
50         } // end while
51         // critical section code
52         t2WantsToEnter = false;
53         // exitMutualExclusion
54         // code outside critical section
55     } // end outer while
56 } // end Thread T2
57
58 } // end Thread T2
59
60 } // end Thread T2
```

Anche in questo caso abbiamo due variabili che esprimono la volontà di entrare nella critical region, ma quello che cambia è il loop più interno. All'interno del loop **si forza ogni thread** che sta eseguendo il suo ciclo ad impostare ripetutamente a false la sua flag per brevi periodi. Questo permette all'altro thread di procedere oltre il suo ciclo while con la sua flag impostate a true.

La quarta versione **garantisce la mutua esclusione e previene i deadlock**, ma permette ad **un altro problema** potenzialmente devastante di svilupparsi, e cioè il **posticipo indefinito**. Questa è una condizione simile al deadlock ma in qualche modo possiamo supporre che sotto certe condizioni è risolvibile (a differenza del deadlock). Se si dovesse verificare una fase di stallo, con l'utilizzo di questi doppi flag e con i timer la situazione dovrebbe risolversi, ma non si può determinare in quanto tempo.

## Vediamo adesso la soluzione definitiva

VERSIONE 5 - FINALE

```
1 System:  
2  
3 int favoredThread = 1;  
4 boolean t1WantsToEnter = false;  
5 boolean t2WantsToEnter = false;  
6  
7 startThreads(); // initialize and launch threads  
8  
9 Thread T1:  
10  
11 void main() {  
12     while ( !done )  
13     {  
14         t1WantsToEnter = true;  
15  
16         while ( t2WantsToEnter )  
17         {  
18             if ( favoredThread == 2 )  
19             {  
20                 t1WantsToEnter = false;  
21                 while ( favoredThread == 2 );  
22                 // busy wait  
23                 t1WantsToEnter = true;  
24             } // end if  
25         } // end while  
26  
27         // critical section code  
28  
29         favoredThread = 2;  
30         t1WantsToEnter = false;  
31  
32         // code outside critical section  
33  
34     } // end outer while  
35 } // end Thread T1  
  
39 Thread T2:  
40  
41 void main() {  
42     while ( !done )  
43     {  
44         t2WantsToEnter = true;  
45  
46         while ( t1WantsToEnter )  
47         {  
48             if ( favoredThread == 1 )  
49             {  
50                 t2WantsToEnter = false;  
51                 while ( favoredThread == 1 );  
52                 // busy wait  
53                 t2WantsToEnter = true;  
54             } // end if  
55         } // end while  
56  
57         // critical section code  
58  
59         favoredThread = 1;  
60         t2WantsToEnter = false;  
61  
62         // code outside critical section  
63  
64     } // end outer while  
65 } // end Thread T2
```

L'Algoritmo di Dekker usa ancora un flag per indicare il desiderio di un thread di entrare nella sua sezione critica, ma incorpora anche il concetto di un "thread favorito" che entrerà nella selezione critica in caso di "pareggio".

Supponiamo, che quando T1 esegue il test while, scopra che il flag di T2 è impostato su true. In questo caso, c'è una **contesa** tra i thread che tentano di entrare nelle rispettive sezioni critiche. Il thread T1 entra nel corpo del suo ciclo while, in cui esamina il valore della variabile favoredThread, che viene utilizzato per risolvere tali conflitti. Se il thread T1 è il thread preferito, T1 salta il corpo dell'if ed esegue ripetutamente il test while, in attesa che T2 imposti t2WantsToEnter su false, che, come vedremo, prima o poi accadrà. Se T1 determina che T2 è il thread preferito, T1 è forzato ad entrare nel corpo dell'istruzione if, dove imposta t1WantsToEnter su false, quindi rimane all'interno del ciclo finché T2 rimane il thread favorito. Impostando t1WantsToEnter a false, T1 consente a T2 di entrare nella propria sezione critica.

L'Algoritmo di Dekker **garantisce la mutua esclusione prevenendo deadlock e posticipi indefiniti**. Tuttavia, la prova di questa affermazione non è immediatamente evidente, a causa della natura complessa della mutua esclusione.

## Algoritmo di Peterson

Lo sviluppo dell'algoritmo di Dekker nel paragrafo precedente introduce alcuni problemi sottili che sorgono a causa della concorrenza e dell'asincronismo nei sistemi multiprogrammati.

```

1 System:
2
3 int favoredThread = 1;
4 boolean t1WantsToEnter=false;
5 boolean t2WantsToEnter = false;
6
7 startThreads(); //initialize and launch
8
9 Thread T1:
10
11 void main()
12 {
13     while ( !done )
14     {
15         t1WantsToEnter = true;
16         favoredThread = 2;
17
18         while ( t2WantsToEnter &&
19                 favoredThread == 2 );
20             // critical section code
21
22         t1WantsToEnter = false;
23
24         // code outside critical section
25     } // end while
26
27 }
28 } // end Thread T1
29
30 Thread T2:
31
32 void main()
33 {
34     while ( !done )
35     {
36         t2WantsToEnter = true;
37         favoredThread = 1;
38
39         while ( t1WantsToEnter &&
40                 favoredThread == 1 );
41             // critical section code
42
43         t2WantsToEnter = false;
44
45         // code outside critical section
46     } // end while
47
48 }
49 } // end Thread T2

```

All'inizio del while, il thread1 setta t1WantsToEnter a true e consente a T2 di entrare, rendendolo il thread favorito. T2 setta t2WantsToEnter a true, e rende T1 il thread favorito. Se dovesse avvenire una situazione come quella di dekker v4 o v3, ossia di possibile deadlock o posticipo indefinito, in Peterson non ci sarebbe alcun problema. Affinché si verifichi il deadlock, sia T1 che T2 devono attendere contemporaneamente in busy wait. Ciò non si verificherà, perché favoredThread è 1 o 2 e non viene modificato durante il ciclo while, il che significa che il test while avrà sempre esito negativo per un thread, consentendogli di entrare nella sua sezione critica. Affinché si verifichi un posticipo indefinito, un thread dovrebbe essere in grado di completare e reimmettersi continuamente nella sua sezione critica mentre l'altro thread è in attesa. Poiché ogni thread imposta il valore di favoredThread sul numero corrispondente all'altro thread prima di entrare nel ciclo while, l'algoritmo di Peterson assicura che i due thread alterneranno l'esecuzione delle loro sezioni critiche, il che significa che non può verificarsi un posticipo indefinito.

Questi algoritmi che abbiamo visto sono validi solamente per due thread

## Mutua esclusione per N thread: Algoritmo di bakery di Lamport

**Lamport** è stato il primo ad introdurre un algoritmo che permetta ai thread di entrare nelle sezioni critiche velocemente quando l'accesso alle sezioni critiche non è conteso (che è un caso comune). Questi algoritmi di mutua esclusione veloci tendono ad essere inefficienti quando la sezione critica è invece contesa.

L'algoritmo di Lamport fornisce una soluzione più semplice, presa in prestito da uno scenario comune nel mondo reale: l'attesa in coda ad un panificio. Inoltre, l'algoritmo di Lamport non richiede che qualche operazione avvenga atomicamente.

Vediamo lo pseudocodice:

```
1 System:  
2 // array that records which threads are taking a ticket  
3 boolean choosing[n];  
4  
5 // value of the ticket for each thread initialized to 0  
6 int ticket[n];  
7  
8 startThreads(); // initialize and launch all threads  
10  
11 Thread Tx;  
12  
13 void main()  
14 {  
15     x = threadNumber(); // store current thread number  
16  
17     while ( !done ) {  
18         // take a ticket  
19         choosing[x] = true; // begin ticket selection process  
20         ticket [x] = maxValue( ticket ) + 1;  
21         choosing[x] = false; // end ticket selection process  
22  
23         // wait for number to be called by comparing current  
24         // ticket value to other thread's ticket value  
25         for (int i = 0 ; i < n ; i++) {  
26             if ( i == x ) {  
27                 continue; // no need to check own ticket  
28             }  
29  
30             // busy wait while thread[i] is choosing  
31             while ( choosing[i] != false );  
32  
33             // busy wait until current ticket value is lowest  
34             while ( ticket[i] != 0 && ticket[i] < ticket[x] );  
35  
36             // tie-breaker code favors smaller thread number  
37             if ( ticket[i] == ticket[x] && i < x )  
38                 // loop until thread[i] leaves  
39                 // its critical section  
40                 while ( ticket[i] != 0 ); // busy wait  
41         } // end for  
42  
43         // critical section code  
44  
45         ticket[x] = 0; // exitMutualExclusion  
46  
47         // code outside critical section  
48     } // end while  
53 } // end Thread Tx
```

Nell'algoritmo di Lamport, ogni thread rappresenta un cliente che deve "prendere un ticket" per determinare quando un thread può entrare nella sua sezione critica. Quando un thread possiede un ticket con il più basso valore numerico, può entrare nella sua sezione critica. La mutua esclusione viene applicata resettando il valore del ticket del thread quando esso esce dalla sua sezione critica. Al contrario di un distributore di ticket del mondo reale, l'algoritmo di Lamport permette a più thread di ottenere lo stesso numero di ticket. Come vedremo, l'algoritmo include un meccanismo per risolvere i pareggi e assicurare che solamente un thread possa entrare nella sua sezione critica in un dato istante.

L'array di booleani choosing è di dimensione n; se il thread Tx sta attualmente selezionando un valore per il proprio ticket, allora choosing(x) sarà true. Altrimenti, choosing(x) sarà false.

Ogni thread partecipante all'algoritmo di bakery esegue gli stessi costrutti enterMutualExclusion (righe 19-44) ed exitMutualExclusion (riga 48). Quando un thread viene avviato, esegue la riga 15, che memorizza un valore intero nella variabile x per identificare univocamente il thread. Il thread utilizza questo valore per determinare le entry corrispondenti a sé stesso negli array choosing e ticket. Ogni thread "prende un ticket" eseguendo le righe di codice 19-22. La riga 20 indica che il thread corrente sta cercando di prendere un ticket impostando choosing(x) a true.

Il thread chiama il metodo maxValue (riga 21), che ritorna il più grande valore nell'array di interi ticket. Il thread quindi aggiunge uno a quel valore e lo memorizza come il proprio valore di ticket, ticket(x) (riga 21). Si noti che se ci sono diversi thread nel sistema, il metodo maxValue può impiegare un po' di tempo per essere eseguito. Dopo che il thread ha assegnato il valore del proprio ticket a ticket(x) (riga 21), il thread imposta choosing(x) a false (riga 22), indicando che non sta più selezionando un ticket. Si noti che quando un thread esce dalla sua sezione critica (riga 48), il valore del ticket viene impostato a zero, stando a significare che il valore del ticket del thread è diverso da zero solo se esso vuole entrare nella sua sezione critica.

Prima di entrare nella sua sezione critica, un thread deve eseguire il ciclo for (righe 26-44) che determina lo stato di tutti i thread nel sistema. Se il thread Ti, il thread da esaminare, e il thread Tx, il thread che sta eseguendo il ciclo for, sono identici (riga 28), Tx esegue l'istruzione continue (riga 30), che salta le istruzioni successive nel corpo del ciclo for e procede direttamente ad incrementare i alla riga 26. Altrimenti, Tx determina se Ti sta scegliendo un ticket (riga 34). Se Tx non aspetta fino a quando Ti non ha scelto il suo ticket prima di entrare nella sezione critica, la mutua esclusione potrebbe essere violata. Per capire perché, esaminiamo le altre due condizioni che vengono testate in ogni iterazione.

La riga 37 controlla se il thread corrente possiede un ticket con valore minore o uguale a quello del ticket del thread che si sta esaminando. tuttavia, due o più thread nel sistema possono ottenere un ticket con lo stesso valore. Per esempio, consideriamo un thread Ta che viene rimosso dal processore dopo che il metodo maxValue ritorna e prima che il thread assegna il nuovo valore a ticket(a) alla riga 21. Se il prossimo thread che viene eseguito chiama maxValue, il metodo ritornerà lo stesso valore che aveva ritornato a Ta. Di conseguenza, qualunque thread potrebbe ottenere lo stesso valore per i propri ticket. Pertanto, in caso di pareggio, la riga 40 indica che il thread con identificatore univoco più basso procede per primo.

Quanto Tx ha superato tutti i testi per ogni thread (righe 28-43), a Tx viene garantito accesso esclusivo alla sua sezione critica. Quando Tx esce dalla sua sezione critica, imposta il valore del suo ticket a 0 (riga 48) per indicare che non sta più eseguendo codice nella sua sezione critica né sta cercando di entrarvi.

## SOLUZIONI HARDWARE AL PROBLEMA DELLA MUTUA ESCLUSIONE

I progettisti di hardware tendono ad implementare meccanismi precedentemente gestiti da software per migliorare le prestazioni e ridurre i tempi di sviluppo. La soluzione più semplice per impedire la violazione della mutua esclusione sarebbe quella di implementare meccanismi di controllo direttamente in hardware.

## Disabilitare gli interrupt

Sfortunatamente, **disabilitare gli interrupt** pone dei **limiti** a cosa un software può fare all'interno della sua sezione critica. Per esempio, un thread che entra in un ciclo infinito nella sua sezione critica dopo aver disabilitato gli interrupt, non cederà mai il processore. Disabilitare gli interrupt non è una soluzione praticabile sui sistemi con più processori. Dopotutto, il suo scopo è di assicurare che i processi e i thread non vengano rimossi dal processore.

Disabilitare gli interrupt su un processore (o anche su tutti), non è sufficiente per applicare la mutua esclusione. In generale, i progettisti dei sistemi operativi evitano di disabilitare gli interrupt per fornire la mutua esclusione. Tuttavia, c'è un **insieme limitato di soluzioni** in cui è ottimale per il kernel disabilitare gli interrupt per del codice fidato, la cui esecuzione richiede un breve periodo di tempo.

## Istruzione test-and-set

L'istruzione **test-and-set** permette ad un thread di eseguire questa operazione atomicamente (cioè indivisibilmente). Queste operazioni vengono anche descritte come operazioni di memoria read-modify-write (RMW) atomiche, perché il processore legge un valore dalla memoria, lo modifica nei suoi registri e scrive il valore modificato in memoria senza interruzioni.

L'istruzione `testAndSet(a, b)` elimina la possibilità che il thread possa essere sospeso durante questo intervallo. Vediamo un semplice esempio:

```
testAndSet(a, b)
```

Per prima cosa l'istruzione legge il valore di `b`, che potrebbe essere o `true` o `false`. Poi, il valore viene copiato in `a`, e l'istruzione imposta il valore di `b` a `true`.

## Istruzione swap

Nonostante il concetto sia semplice, uno scambio di valori tra due variabili richiede tre istruzioni e la creazione di una variabile temporanea in molti linguaggi di programmazione di alto livello:

```
temp = a -> a = b -> b = temp
```

Siccome queste operazioni di scambio vengono eseguite regolarmente, molte architetture supportano una istruzione di `swap`, che permette a un thread di scambiare i valori di due variabili atomicamente. L'istruzione

```
swap(a,b)
```

funziona come segue: per prima cosa carica il valore di `b`, che può essere `true` o `false`, in un registro temporaneo. Poi, il valore di `a` viene copiato in `b` e il valore del registro temporaneo viene copiato in `a`.

# SEMAFORI

Un altro meccanismo che un sistema può fornire per implementare la mutua esclusione è il **semaforo**. Un semaforo contiene una **variabile protetta** il cui valore intero, una volta inizializzato, può essere acceduto e alterato da due sole operazioni, **P** e **V**. Un thread chiama l'operazione P (anche chiamata operazione di **wait**) quando vuole entrare nella sua sezione critica, e chiama l'operazione V (anche chiamata operazione **signal**) quando ne vuole uscire.

L'inizializzazione imposta il valore della variabile protetta per indicare che nessun thread è in esecuzione nella sua sezione critica. Esso crea anche una coda che memorizza riferimenti ai thread in attesa di entrare nella loro sezione critica protetta da quel semaforo.

P e V sono due operazioni indivisibili, e se più threads tentano di accedere a P(S) simultaneamente, l'implementazione dovrebbe garantire che venga processato un thread per volta; gli altri thread dovrebbero rimanere in attesa con la garanzia che non si manifesti il fenomeno di posticipo indefinito.

## Mutua esclusione con i semafori

Un **semaforo binario** permette ad un solo thread per volta di accedere alla sezione critica, utilizzando solo due operazioni:

- **Operazione di WAIT (P)**: se nessun thread è in attesa, permette al thread di accedere alla propria sezione critica e decrementa la variabile protetta (a 0 in questo caso); in caso contrario lo inserisce nella coda di attesa.
- **Operazione di SIGNAL (V)**: indica che un thread si trova all'esterno della sezione critica, e incrementa la variabile protetta (da 0 a 1); a questo punto, uno dei thread in attesa può entrare.

Lo pseudocodice è il seguente:

```
1 System:  
2  
3 // create semaphore and initialize value to 1  
4 Semaphore occupied = new Semaphore(1);  
5  
6 startThreads(); // initialize and launch both threads  
7  
8 Thread Tx:  
9  
10 void main()  
11 {  
12     while (!done)  
13     {  
14         P(occupied); // wait  
15         // critical section code  
16         V(occupied); // signal  
17         // code outside critical section  
18     } // end while  
19 } // Thread TX
```

Nel momento in cui modifico questa variabile invio una notifica a tutti i thread.

## Sincronizzazione dei thread con i semafori

I semafori possono anche essere utilizzati per **sincronizzare due o più thread concorrenti**. Per esempio, supponiamo che un thread, T1, voglia ricevere una notifica relativa al verificarsi di un particolare evento. Supponiamo che un altro thread, T2, sia in grado di accorgersi che questo evento è avvenuto.

Una prima implementazione di un semaforo in java è la seguente

```
import java.util.concurrent.Semaphore;

public class SimpleSemaphore {

    // due thread stampano dei messaggi, vogliamo che non siano
    // interrotti fino al termine della stampa: usiamo un lock

    public static void main(String args[]) throws Exception {
        // il primo parametro è il numero di permits: 1 = binario
        // ovvero semaphoro disponibile / non disponibile

        // il secondo parametro è fairness: true/false
        // true: politica FIFO per decidere quale dei thread
        //       in attesa deve accedere non appena il lock
        //       è disponibile
        // false: la politica viene decisa dalla JVM

        Semaphore sem = new Semaphore(1,true);

        Thread thread_A = new Thread(new SynchroPrint(sem, "message from A"));
        Thread thread_B = new Thread(new SynchroPrint(sem, "message from B"));

        thread_A.start();
        thread_B.start();

        thread_A.join();
        thread_B.join();
    }
}
```

Il semaforo, se usato in maniera corretta, garantisce la mutua esclusione; il problema è infatti usarlo correttamente. In questa classe stiamo considerando due thread (A e B) che stampano dei messaggi. Vogliamo fare in modo che questi thread non vengano interrotti prima che venga eseguita la stampa.

Per definire un semaforo binario si utilizza la semplice sintassi

```
Semaphore sem = new Semaphore(1,true);
```

Il primo parametro serve per definire che il semaforo è binario (0,1), il secondo indica il valore di **fairness**, che se posto a true indica che in questo semaforo si utilizza una politica FIFO (Se posto a false la politica viene gestito dallo scheduler della Virtual Machine).

```

class SynchroPrint extends Thread {
    Semaphore semaphore;
    String message;
    public SynchroPrint(Semaphore s, String m) {
        semaphore = s;
        message = m;
    }
    public void run() {
        try {
            semaphore.acquire(); // poi commentare per mostrare comportamento
            for(int i = 1; i <= 1000; i++) {
                System.out.println(message+": " + i);
                Thread.sleep(300);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        semaphore.release(); // poi commentare per mostrare comportamento
    }
}

```

Entrambi i thread condividono lo stesso metodo Run, e quello che vogliamo realizzare è che un thread stampi ininterrottamente tutti i messaggi prima che stampi il secondo thread. Questo avviene caricando in coda il thread (acquiere()) ed eliminandolo solamente al termine della stampa.

L'output sarà di questo tipo (troncando tutte le successive stampe):

```

java SimpleSemaphore
message from A: 1
message from A: 2
message from A: 3
message from A: 4
message from A: 5
message from A: 6
message from A: 7
message from A: 8
message from A: 9
message from A: 10

```

| Non stiamo vincolando A ad essere il primo thread ad essere schedulato.

# Lezione 19

Riprendiamo e completiamo il discorso cominciato ieri sui semafori. Abbiamo infatti iniziato a parlare di mutua esclusione e che è possibile implementarla utilizzando alcuni algoritmi (Dekker e Peterson, ad esempio) o alcuni software come i semafori.

A differenza degli algoritmi, l'implementazione con i semafori non prevede **busy wait**, il che è sicuramente un notevole vantaggio. Il primo semaforo che abbiamo visto è quello binario che prevede i due valori 0 e 1. Abbiamo poi visto una implementazione dei semafori in Java tramite l'oggetto Semaphore.

## Semaforo: produttore-consumatore

Implementiamo ora con l'uso dei semafori una relazione produttore-consumatore. Si utilizzano **due semafori, uno per il produttore e uno per il consumatore**, ciascun semaforo protegge l'accesso alla risorsa condivisa (sharedValue nel codice) da parte di ciascun thread. In particolare, ogni thread, prima di potere svolgere il proprio compito, deve attendere un certo evento, il consumatore che sia prodotto un valore (evento indicato dal semaforo del produttore valueProduced), il produttore che il valore prodotto in precedenza sia consumato (evento indicato dal semaforo del produttore valueConsumed).

Il produttore entra nella sezione critica per produrre il valore, il consumatore sarà bloccato in questo frangente. Il consumatore entrerà poi nella sezione critica per leggere il valore scritto dal produttore e quest'ultimo non potrà aggiornare il valore fino a quando il consumatore non avrà finito.

Vediamo ora il codice:

```
1 System:  
2 // semaphores that synchronize access to sharedValue  
3 Semaphore valueProduced = new Semaphore(0);  
4 Semaphore valueConsumed = new Semaphore(1);  
5 int sharedValue; // variable shared by producer and consumer  
6  
7 startThreads(); // initialize and launch both threads  
8  
9 Producer thread  
10  
11 void main()  
12 {  
13     int nextValueProduced; // variable to store value produced  
14  
15     while ( !done )  
16     {  
17         nextValueProduced = generateTheValue(); // produce value  
18         P( valueConsumed ); // wait until value is consumed  
19         sharedValue = nextValueProduced; // critical section  
20         V( valueProduced ); // signal that value has been produced  
21     } // end while  
22 } // end producer thread  
23  
24 Consumer thread  
25 void main()  
26 {  
27     int nextValue; // variable to store value consumed  
28  
29     while ( !done )  
30     {  
31         nextValue = sharedValue; // critical section  
32         P( valueProduced ); // wait until value is produced  
33         sharedValue = nextValueConsumed; // critical section  
34         V( valueConsumed ); // signal that value has been consumed  
35         processTheValue( nextValueConsumed ); // process the value  
36  
37     } // end while  
38  
39 } // end consumer thread  
40  
41 } // end consumer thread
```

Qui vediamo lo pseudocodice del produttore-consumatore implementato con i semafori: il produttore genera un nuovo valore (riga 17), quindi attende che venga consumato il valore sul semaforo valueConsumed. Ma il valore iniziale era stato settato volutamente ad 1 quindi il produttore entra nella sezione critica e assegna alla sharedValue il valore generato casualmente (riga 19). In seguito, il produttore effettua una operazione sul semaforo valueProduced

aumentandolo ad 1 (riga 20). Il consumatore è in attesa su quel semaforo (riga 34) e quando il producer ne aumenta il valore, il consumatore legge il valore prodotto (ovvero assegna il valore di sharedValue alla variabile locale nextValueConsumed, riga 35). A questo punto il consumer effettua un signal sul semaforo valueConsumed (riga 36), il che permette al produttore di entrare nella sezione critica e così via.

Analizziamo ora nello specifico il thread produttore: abbiamo una variabile per memorizzare il valore prodotto (nextValueProduced), che viene generato dentro il ciclo while. Il produttore dopo aver creato il valore dovrebbe poterlo scrivere nella variabile condivisa, ma affinché questo avvenga il buffer deve essere vuoto. La variabile (buffer) è vuota quando il consumatore ha letto o ha completato l'operazione di accesso alla variabile stessa. Per questo motivo il produttore per potere scrivere deve fare l'operazione P( valueConsumed ) per capire se il buffer è vuoto. Vediamo nello specifico cosa prevede questo controllo:

**P(S):**

**If  $S > 0$**

$S = S - 1$

**Else**

*The calling thread is placed in the semaphore's queue of waiting threads*

Se P vale 1, il thread ha il permesso di entrare, e quando lo fa decrementa il valore di S portandolo a 0 e bloccando l'accesso agli altri thread. Altrimenti, se S non è maggiore di 0 allora il thread viene messo in coda di attesa.

Una volta che il produttore avrà il permesso di entrare potrà settare il valore del buffer ( shardValue ) pari al valore generato precedentemente ( nextValueProduced ). Per far sì che il produttore scriva per primo nel buffer abbiamo inizializzato il semaforo del consumatore ad 1. Ora supponiamo che lo scheduler passi il controllo al consumatore, allora il consumatore applicherà un comportamento analogo a quello del producer. Anche lui effettua il P(wait) nell'altro semaforo per capire se può accedere o meno alla sezione critica. Ma dato che il semaforo del produttore è stato inizializzato a 0 e quindi il consumer entrerà in attesa. Nel mentre il produttore eseguirà la riga 20, notificando la sua uscita dalla sezione critica con la funzione V:

**V(S):**

**If any threads are waiting on S**  
**Resume the "next" waiting thread in**  
**the semaphore's queue**

**Else**

$S = S + 1$

Il produttore incrementerà il valore di valueProduced. Questo porterà il consumatore a svegliarsi dalla fase di wait, che ora potrà entrare nella sezione critica e leggere il valore condiviso. Fatto ciò il consumatore notificherà il produttore di aver letto effettuando l'operazione V( valueConsumed ) e così via. Per riassumere, i valori dei semafori valueProduced e valueConsumed si alterneranno come segue: 0, 1 → 1, 0 → 0, 1.. grazie all'utilizzo delle funzioni P(wait) e V(signal). In sostanza la

funzione di **P è di decrementare la variabile** passata come argomento, al contrario **V ha la funzione di aumentare la variabile** passata come argomento.

### Semafori contatori

Oltre ai semafori binari ne esistono anche di altre tipologie. Un esempio è quello dei **counting semaphores** che sono inizializzati con valori maggiori di 1 e possono essere utilizzati per accedere ad una stessa risorsa condivisa da più thread. Ogni operazione P decremente il semaforo di 1, indicando che un'altra risorsa è stata rimossa dalla zona condivisa ed è stata utilizzata da un altro thread. Ogni operazione V incrementa di 1 il semaforo indicando che un thread ha rilasciato una risorsa e che è ora utilizzabile da un altro thread.

Se un thread tenta di effettuare una operazione P quando il semaforo ha raggiunto il valore 0, il thread dovrà attendere prima che una risorsa sia restituita al pool tramite un'operazione V.

### Implementazione dei semafori

I semafori possono essere implementati o a **livello kernel** o a **livello applicativo**. E' chiaro che le implementazioni a livello applicativo sono più inefficienti in quanto fanno uso di loop e dunque non eliminano il problema dovuto al **busy waiting**.

Le implementazioni a livello kernel superano questo problema. Uno dei modi a livello kernel, per implementare i semafori, è quello di rimuovere interrupt ma questo non è sempre una buona soluzione in quanto potrebbero presentarsi **deadlock**.

# Lezione 20

---

## Programmazione concorrente

I semafori funzionano logicamente ma a livello pratico presentano delle criticità, ad esempio, nell'uso di P e V. I problemi di deadlock o di posticipazione indefinita si presentano infatti nel momento in cui i thread vengono schedulati in un preciso ordine tale che i thread stessi rimangano tutti in attesa e analizzare il codice non ci permette sempre di capire se si presenterà questo genere di problema. Inoltre il livello dei semafori è così basso che risulta difficile utilizzarli per la risoluzione di problemi complessi.

Si è quindi immaginato di creare dei servizi di più alto livello rispetto ai semafori per garantire la mutua esclusione.

### Monitor

Come sappiamo il semaforo consente la modifica, a turni, di una variabile condivisa, da parte di vari thread. Il **monitor** invece è una sorta di *scatola* che contiene sia dati che procedure, questi elementi risultano protetti all'interno di questa *scatola* e affinché si possano modificare dati o procedure bisogna aprire questa *scatola*.

In maniera formale definiamo il **monitor** come un oggetto che protegge sia dati che procedure. Nello specifico è un oggetto che contiene dati e procedure necessarie per effettuare l'allocazione di una, o un gruppo, di risorse condivise riusabili serialmente.

Per effettuare l'allocazione tramite monitor, un thread deve richiamare una routine d'ingresso nel monitor. Questa entry può essere però richiamata da un thread per volta infatti tutti i thread al di fuori del monitor non hanno alcun accesso alle informazioni che il monitor custodisce.

Così come i semafori, il monitor, prevede due operazioni di **wait** e **signal**. Nel momento in cui un thread prova ad entrare nel monitor e il monitor è libero, il thread acquisisce un **lock** sul monitor e vi entra. Viceversa se risulta che un altro thread si trova già al suo interno, allora il primo thread dovrà andare in wait al di fuori del monitor fin quando non verrà rilasciato il lock del monitor e avrà ricevuto un signal. Dato che esternamente non è possibile accedere a nessuna risorsa interna al monitor, si può affermare che questo meccanismo consente di realizzare il cosiddetto **information hiding**: le informazioni all'interno del monitor sono nascoste a tutti i thread all'esterno del monitor.

Per evitare la posticipazione indefinita, un monitor assegna una priorità più alta ai thread in attesa, piuttosto che ai nuovi arrivati.

### Variabili di condizione (Condition Variable)

Le due operazioni che nel semaforo erano P e V, nei monitor sono **wait** e **signal**. Queste operazioni vengono effettuate su una **variabile di condizione (conditionVariable)** che fa riferimento ad un evento che avviene al di fuori del monitor.

I monitor, infatti, implementano sia la mutua esclusione che la sincronizzazione tra thread. Un thread può infatti avere bisogno di attendere all'esterno di un monitor che, un altro thread interno al monitor stesso, svolga una precisa azione. Un thread che sta attendendo l'avvenire di un certo evento lo fa associandovi una specifica condition variable. Ad ogni operazione di wait e signal va assegnata una specifica condition variable.

Nei monitor abbiamo più variabili di condizione, ognuna delle quali rappresenta uno specifico evento. A differenza delle variabili tradizionali, le variabili di condizione sono caratterizzate dall'essere associate ad una coda usata per svegliare i thread che sono in attesa da più tempo. Inoltre, nonostante vengano chiamate variabili, queste ultime sono a tutti gli effetti degli oggetti.

Un thread che esegue una wait su una condition variable viene posizionato alla fine della coda e finché vi rimane è al di fuori del monitor in modo che un altro thread vi possa entrare e richiamare un signal. Un thread che richiama un signal su una condition variable particolare fa sì che un thread in attesa su quella condition variable sia rimosso dalla coda associata e rientri nel monitor. Prima che un thread possa entrare, o rientrare, nel monitor il thread che ha richiamato il signal deve uscire dal monitor stesso.

Il signal avviene secondo due modalità:

- **Signal and exit:** un thread dentro il monitor ha completato le sue operazioni e manda un signal uscendo dal monitor.
- **Signal and continue:** un thread all'interno del monitor invia un signal ma quest'ultimo è una notifica che serve a comunicare che presto il monitor potrebbe essere libero, ma il lock viene mantenuto fin quando il thread non esce. In pratica c'è una sorta di intervallo da quando viene mandato il signal a quando il monitor diventa effettivamente disponibile. Questa modalità viene utilizzata di default in java.

Un thread può uscire da un monitor mettendosi in attesa su una condition variable o completando l'esecuzione del codice protetto dal monitor. Un thread richiamato da un signal di tipo signal and continue deve attendere finché il thread che ha lanciato il signal non ne esca.

Vediamo ora un esempio di gestione di una variabile condivisa tra più thread tramite monitor:

```
1 // Fig. 6.1: Resource allocator monitor
2
3 // monitor initialization (performed only once)
4 boolean inUse = false; // simple state variable
5 Condition available; // condition variable
6
7 // request resource
8 monitorEntry void getResource()
9 {
10     if ( inUse ) // is resource in use?
11     {
12         wait( available ); // wait until available is signaled
13     } // end if
14
15     inUse = true; // indicate resource is now in use
16
17 } // end getResource
18
19 // return resource
20 monitorEntry void returnResource()
21 {
22     inUse = false; // indicate resource is not in use
23     signal( available ); // signal a waiting thread to proceed
24
25 } // end returnResource
```

Cominciamo vedendo una variabile booleana inUse settata di default a false che notifica la disponibilità del monitor, se è True la risorsa non è disponibile. Abbiamo poi la condition variable per gestire le code di thread. In questa condition variable si pongono in attesa i thread che trovano la risorsa occupata, finché un thread non la liberi comunicandolo poi con un opportuno signal.

La prima entry del monitor si chiama getResource: se inUse è uguale a true, il monitor è occupato, e quindi si effettua un wait sulla variabile di condizione. Al termine del wait, quando il thread riesce ad entrare nel monitor, va a settare inUse a true.

returnedResource è la entry per ritornare la risorsa, in questo caso prima di rilasciare la risorsa setto inUse a false ed eseguo un signal notificando ai thread in coda di accedere al monitor.

Questo esempio ricorda molto l'implementazione di un semaforo con P e V:

```
1 // Fig. 6.1: Resource allocator monitor
2
3 // monitor initialization (performed only once)
4 boolean inUse = false; // simple state variable
5 Condition available; // condition variable
6
7 // request resource
8 monitorEntry void getResource() similar to P(s)
9 {
10     if (inUse) // is resource in use?
11     {
12         wait(available); // wait until available is signaled
13     } // end if
14
15     inUse = true; // indicate resource is now in use
16
17 } // end getResource
18
19 // return resource
20 monitorEntry void returnResource() similar to V(s)
21 {
22     inUse = false; // indicate resource is not in use
23     signal(available); // signal a waiting thread to proceed
24
25 } // end returnResource
```

### Buffer circolare: produttore e consumatore con i monitor

Vediamo un esempio di implementazione di un buffer circolare (in termini pratici un array) in cui è possibile scrivere dati fino a riempirlo e ricominciare dall'inizio qualora si riempia. In questo esempio produttore e consumatore hanno velocità differenti, quindi per ridurre la latenza, facciamo in modo che produttore e consumatore non scrivano sulla stessa variabile. Se avessimo una sola variabile, con produttore e consumatore che vanno a velocità diverse, avremmo molti tempi morti e uno dei due thread rimarrebbe bloccato.

Con un buffer circolare il produttore può riempire il buffer circolare e il consumatore leggerà in maniera asincrona, sempre in ordine FIFO però. In questo esempio assumiamo che il produttore sia più veloce del consumatore. Se il produttore riempisse l'ultimo elemento dell'array, ripartirebbe dal primo a scrivere (che nel mentre sarà stato svuotato dal consumatore, anche se con tempi diversi).

Il produttore potrebbe trovare l'array pieno e in questo caso dovrà andare in wait, d'altra parte il consumatore potrebbe trovare degli elementi vuoti e in questo caso dovrebbe andare in wait. Vediamo ora lo pseudocodice:

```

3  char circularBuffer[] = new char[ BUFFER_SIZE ]; // buffer
4  int writerPosition = 0; // next slot to write to
5  int readerPosition = 0; // next slot to read from
6  int occupiedSlots = 0; // number of slots with data
7  Condition hasData; // condition variable
8  Condition hasSpace; // condition variable
9
10 // monitor entry called by producer to write data
11 monitorEntry void putChar( char slotData )
12 {
13     // wait on condition variable hasSpace if buffer is full
14     if ( occupiedSlots == BUFFER_SIZE )
15     {
16         wait( hasSpace ); // wait until hasSpace is signaled
17     } // end if
18
19     // write character to buffer
20     circularBuffer[ writerPosition ] = slotData;
21     ++occupiedSlots; // one more slot has data
22     writerPosition = (writerPosition + 1) % BUFFER_SIZE;
23     signal( hasData ); // signal that data is available
24 } // end putChar
25

```

Innanzitutto viene creato il buffer circolare e impostato ad una certa dimensione (BUFFER\_SIZE). Per gestire il comportamento asincrono di produttore e consumatore devo mantenere degli indici writerPosition, readerPosition che indicano rispettivamente in quale posizione del buffer l'elemento successivo sarà inserito dal produttore e da quale cella esso sarà rimosso dal consumatore. La variabile occupiedSlots serve invece a capire quante celle dell'array sono effettivamente piene (ovvero che il produttore vi ha scritto dei dati che non sono ancora stati letti dal consumatore). Creiamo inoltre due variabili di condizione a cui associamo due code: hasData rappresenta la presenza di dati nel buffer, hasSpace rappresenta la assenza di dati nel buffer.

La prima routine che implementiamo è quella per inserire un elemento nell'array. Chiamiamo questo metodo putChar che è mutuamente esclusivo con altri metodi. Se il numero di slot occupati è uguale alla dimensione dell'array allora l'array è pieno e si dovrà fare una wait sulla conditon variable hasSpace, ovvero il thread produttore aspetterà che si liberi un posto nell'array. Se invece c'è spazio nel buffer posso scrivere i dati nel buffer, poi incrementerò il numero di slot occupati nell'array e incrementerò di uno la posizione del writer, utiliziamo l'operatore modulo % per riportarci in prima posizione quando sfioriamo la grandezza del monitor. Dopo avere gestito la scrittura facciamo il signal sulla condition variable hasData, adesso il buffer contiene dei dati.

```

26 // monitor entry called by consumer to read data
27 monitorEntry void getChar( outputParameter slotData )
28 {
29     // wait on condition variable hasData if the buffer is empty
30     if ( occupiedSlots == 0 )
31     {
32         wait( hasData ); // wait until hasData is signaled
33     } // end if
34
35     // read character from buffer into output parameter slotData
36     slotData = circularBuffer[ readPosition ];
37     occupiedSlots--; // one fewer slots has data
38     readerPosition = (readerPosition + 1) % BUFFER_SIZE;
39     signal( hasSpace ); // signal that character has been read
40 } // end getChar

```

Vediamo ora il metodo getChar che ci serve a leggere i dati nel buffer. Se il numero di slot è occupati è pari a zero, effettuiamo un wait sulla condition variable hasData dato che il consumatore in questo caso non avrebbe nulla da leggere. Uscirò da questa zona di attesa solo se ricevo un signal su hasData (a valle dell'operazione putChar). Adesso si può procedere alla lettura: si leggono i dati secondo la readPosition, riduco la posizione degli slot occupati di 1, successivamente si incrementa la posizione di lettura sempre gestendo la dimensione del buffer con l'operatore modulo. Fatta la lettura eseguo un signal sulla condition variable hasSpace.

Questo meccanismo è molto simile al meccanismo implementato con i due semafori per produttore e consumatore. In questo caso però alle condition variable sono associate delle code di thread. Inoltre il vantaggio del buffer circolare è che permette a produttore e consumatore di non attendersi in maniera rigida l'un l'altro come accadeva con i semafori, riducendo così i tempi di attesa e migliorando in generale le prestazioni del sistema.

### Monitor: lettori e scrittori

Questo problema vede la presenza di **thread lettori** e **thread scrittori**. Uno specifico **reader** può entrare nel monitor solo se non ci sono thread writer che stanno scrivendo o se ci sono thread in coda per scrivere; inoltre nel momento in cui un reader ha completato la lettura, può comunicare ad un altro reader di continuare la lettura. Più reader possono leggere contemporaneamente dato che non modificano il contenuto dei dati.

I **writer** hanno invece accesso esclusivo a tutto quello che si può scrivere nel monitor e possono modificare i dati condivisi. Dato che i thread writer possono modificare il contenuto delle risorse condivise devono avere un accesso esclusivo a queste ultime.

Vediamo il codice:

```
1 // Fig. 6.3: Readers/writers problem
2
3 int readers = 0; // number of readers
4 boolean writeLock = false; // true if a writer is writing
5 Condition canWrite; // condition variable
6 Condition canRead; // condition variable
7
8 // monitor entry called before performing read
9 monitorEntry void beginRead()
10 {
11     // wait outside monitor if writer is currently writing or if
12     // writers are currently waiting to write
13     if ( writeLock || queue( canWrite ) )
14     {
15         wait( canRead ); // wait until reading is allowed
16     } // end if
17
18     ++readers; // there is another reader
19
20     signal( canRead ); // allow waiting readers to proceed
21 } // end beginRead
22
```

All'inizio troviamo una variabile che ci dice quanti sono i lettori che stanno leggendo. Troviamo poi una variabile booleana per notificare se uno scrittore sta scrivendo, d'altro canto ricordiamoci che un solo scrittore per volta può scrivere. Abbiamo poi le due condition variable.

La prima entry, beginRead, serve per leggere. La lettura viene negata quando un writer sta scrivendo, o se ci sono dei writer in attesa di potere scrivere (in coda). Se non si verifica neanche una tra le due condizioni allora il lettore può leggere e successivamente effettuare il signal sulla variabile canRead. Questo serve a mandare un signal ad un altro reader, che verrà svegliato e

potrà essere sbloccato per leggere a sua volta, così da ottenere una catena di lettura (**chain reaction**) che si concluderà nel momento in cui tutti i lettori saranno diventati attivi.

```
23 // monitor entry called after reading
24 monitorEntry void endRead()
25 {
26     --readers; // there are one fewer readers
27
28     // if no more readers are reading, allow a writer to write
29     if ( readers == 0 )
30     {
31         signal ( canWrite ); // allow a writer to proceed
32     } // end if
33
34 } // end endRead
35
36 // monitor entry called before performing write
37 monitorEntry void beginWrite()
38 {
39     // wait if readers are reading or if a writer is writing
40     if ( readers > 0 || writeLock )
41     {
42         wait( canWrite ); // wait until writing is allowed
43     } // end if
44 }
```

Nel momento in cui è finita l'operazione di lettura, ciascun reader andrà a decrementare il numero di reader. Se un reader è l'ultimo (`readers == 0`) ed è finita la catena di lettura, allora posso inviare un signal agli scrittori.

Gli scrittori non potranno scrivere fin quando ci saranno reader che leggono o un altro scrittore che scrive. In questo caso il writer manderà un wait in `canWrite`. Se non ci sono lettori che stanno leggendo o non ci sono altri scrittori che stanno scrivendo, allora lo scrittore potrà scrivere importando il `writeLock` su true:

```
45     writeLock = true; // lock out all readers and writers
46 } // end beginWrite
47
48 // monitor entry called after performing write
49 monitorEntry void endWrite()
50 {
51     writeLock = false; // release lock
52
53     // if a reader is waiting to enter, signal a reader
54     if ( queue( canRead ) )
55     {
56         signal( canRead ); // cascade in waiting readers
57     } // end if
58     else // signal a writer if no readers are waiting
59     {
60         signal( canWrite ); // one waiting writer can proceed
61     } // end else
62
63 } // end endWrite
```

Conclusa la lettura si imposta `writeLock` su false, successivamente si controlla se ci sono reader in attesa. Se è così si invia un signal su `canRead` permettendo la lettura ai reader, altrimenti si manda un signal su `canWrite` permettendo ad un altro writer di scrivere.

## Monitor in java

Java nasconde completamente l'implementazione dei monitor e ci consente di utilizzarli in maniera molto semplice. In java, i monitor, possono essere associati a qualsiasi oggetto noi creiamo, e per realizzare la mutua esclusione basta utilizzare la **keyword synchronized** davanti al metodo.

Dobbiamo quindi individuare l'oggetto da proteggere col monitor e i relativi metodi, per proteggerli dobbiamo fare in modo che i metodi in cui questi oggetti sono dichiarati venga definiti come synchronized.

Quando un thread prova ad eseguire un metodo protetto da un metodo synchronized, allora quel thread entrerà nella coda del monitor chiamata **entry set**. La mutua esclusione viene garantita permettendo ad un solo thread alla volta di accedere ad un oggetto o un metodo dichiarato synchronized.

Se non ci sono thread che vogliono accedere al monitor allora il primo thread che lo chiederà avrà accesso immediato, viceversa il thread entrerà in coda d'attesa fin quando il monitor non tornerà disponibile.

In java i monitor sono di tipo signal and continue.

In java le condition variable sono implicite quindi l'operazione di wait non avviene su una specifica condition variable ma su una variabile implicita. Il metodo wait permette al thread di rilasciare il lock sul monitor così da aspettare una certa condition variable implicita.

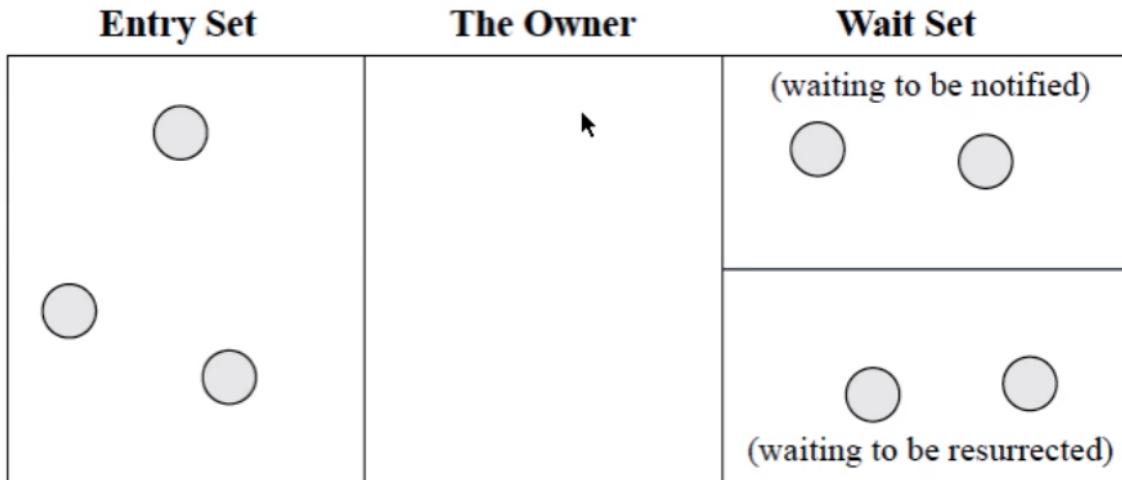
Dopo un wait il thread viene posizionato su una coda chiamata **wait set**, che è una coda di thread per cui è previsto che i thread stessi rientrino nel monitor. Per uscire dal wait set è necessario che i thread ricevano un signal da parte di un altro thread.

Dato che le condition variable sono implicite si possono verificare alcuni comportamenti particolari: un thread potrebbe ricevere un signal per rientrare in un monitor e contemporaneamente verificare che la condizione che attendeva per accedere non si è ancora verificata e in questo caso il thread dovrebbe tornare nello stato di wait. Java consente infatti di realizzare **due diversi tipi di signal**:

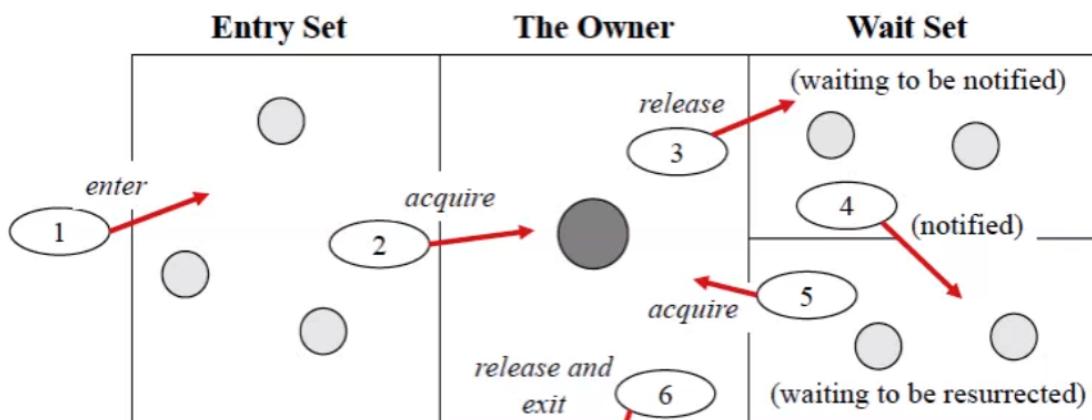
- **Notify**: corrisponde al classico signal. Notify sveglia un solo thread in wait set, quale di questi thread viene svegliato è deciso dalla IJVM, quindi noi non lo sappiamo.
- **NotifyAll**: se ci sono più thread in attesa di essere eseguiti utilizziamo notifyAll che riporta in uno stato di ready tutti i thread in wait set e sarà poi lo scheduler a determinare quali di questi thread mettere in stato di running.

# Lezione 21

Vediamo ora come è strutturato un monitor in java e i vari passaggi che si susseguono nella gestione di un thread:



NB: I pallini grigi rappresentano dei thread



1. Threads enter to acquire lock
2. Lock is acquired by one thread
3. Thread goes to waiting state if you call `wait()` method on the object; (6) otherwise it releases the lock and exits
4. `notify()` or `notifyAll()` move the thread to the notified state (runnable state)
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor

Tutte queste code sono gestite come code separate e vengono gestite dalla IJVM o dallo scheduler, in base al signal che viene lanciato (se è `notify` o `notifyAll`).

## Buffer sincronizzato: implementazione con i monitor

Riprendiamo ora l'esempio del buffer non sincronizzato, che avevamo già visto in passato, con produttore e consumatore (*vedi lezione 17*). Adesso andiamo ad implementare questo meccanismo produttore-consumatore in maniera sicura, ovvero sincronizzando i due thread.

Per fare la sincronizzazione implementeremo un monitor in maniera tale che il produttore sia in grado di scrivere e dovrà attendere che il consumatore legga prima di andare a scrivere di nuovo. Il produttore farà quindi:

- WAIT per aspettare che il consumatore legga dal buffer.
- NOTIFY per notificare il consumatore della avvenuta scrittura.

Il consumatore invece deve eseguire:

- WAIT per aspettare che il produttore scriva nel buffer un dato.
- NOTIFY per notificare il produttore della avvenuta lettura.

In questo specifico esercizio la protezione del monitor andrà implementata nei metodi get e set del buffer, in quanto il buffer rappresenta la risorsa condivisa che vogliamo proteggere. Per implementare questo comportamento definiremo i metodi come **synchronized**, in questa maniera i metodi non potranno essere eseguiti contemporaneamente. Ed è proprio questo ciò che vogliamo fare.

La cosa comoda di questo approccio è che i thread possono essere definiti come preferiamo (infatti in questo caso sono uguali a quelli dell'esempio della lezione 17), la cosa importante sarà solo capire quale risorsa andare a proteggere, in questo caso il buffer sincronizzato. Ciò ci permette di avere molto margine di movimento nella scrittura del codice in quanto l'operazione critica sta proprio nell'individuare le risorse condivise e nel definire i metodi di accesso a quest'ultime come synchronized.

Vediamo l'[esempio](#):

```
// SharedBufferTest creates producer and consumer threads.

public class SharedBufferTest_Synchronized
{
    public static void main( String [] args )
    {
        // create shared object used by threads
        Buffer sharedLocation = new SynchronizedBuffer();

        // create producer and consumer objects
        Producer producer = new Producer( sharedLocation );
        Consumer consumer = new Consumer( sharedLocation );

        producer.start(); // start producer thread
        consumer.start(); // start consumer thread

    } // end main
}

} // end class SharedBufferTest
```

La prima classe che vediamo è la classe SharedBufferTest\_Synchronized la quale contiene il main. All'inizio della classe troviamo la definizione del buffer (definito come SynchronizedBuffer) seguito dalla definizione e inizializzazione dei thread produttore e consumatore che lavoreranno sulla stessa variabile condivisa.

Vediamo ora come è definito l'oggetto SynchronizedBuffer:

```

public class SynchronizedBuffer implements Buffer
{
    private int buffer = -1; // shared by Producer and Consumer
    private int occupiedBuffers = 0; // new-SYNCH

    // parola chiave synchronized
    public synchronized void set( int value )
    {
        // il nome del thread che ha richiesto il metodo set()
        String name = Thread.currentThread().getName();

        // new-SYNCH: controllo se il buffer è pieno e aspetto
        while (occupiedBuffers == 1) {
            // wait
            try {
                showMessage("-- " + name + " tenta di scrivere.");
                showMessage("-- Buffer pieno... " + name + " waits..");
                wait();
            } // se il thread viene interrotto durante il waiting...
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // se siamo qui il buffer è vuoto (0) e posso scrivere
        buffer = value;

        // adesso il buffer è pieno (1) e non posso inserire nuovi valori
        ++occupiedBuffers;

        showMessage(name + " ha scritto " + buffer + " --> notify()");
        notify();
    } // end method set, rilascia il lock
}

```

Innanzitutto vediamo come il SynchronizedBuffer implementi l'interfaccia buffer a noi già nota (*vedi lezione 17*). È presente la variabile condivisa da produttore e consumatore di nome buffer inizializzata arbitrariamente a -1; troviamo poi un intero occupiedBuffer, che ci serve a capire se il buffer contiene dei dati p meno (se è pari a 0 vuol dire che il buffer è stato letto e svuotato e quindi un produttore dovrà scrivere, viceversa, se è pari a 1 il buffer è pieno ed un produttore vi ha scritto qualche dato che dovrà essere letto da un consumatore). Dopodiché la classe implementa il metodo set, dichiarato come metodo synchronized, in quanto dovrà essere eseguito in maniera mutuamente esclusiva rispetto a get (che sarà a sua volta implementato come metodo synchronized per il medesimo motivo). All'interno del metodo abbiamo un controllo (ciclo while) per verificare se il buffer sia libero o meno, nel caso in cui fosse pieno (occupiedBuffer = 1) dovremo aspettare che si liberi mandando il thread in wait. Il metodo wait() si inserisce in un blocco try catch per gestire eventuali eccezioni e/o errori che si potrebbero presentare.

Una volta che il buffer si è svuotato e il thread produttore ha ricevuto una notifica dal consumatore si può scrivere nel buffer, per farlo ci basta impostare il valore del buffer pari alla variabile value generata dal produttore. Poi avverrà l'incremento della variabile occupiedBuffer e il lancio di una notifica con il metodo notify() che serve a chiudere il metodo set.

Teniamo però a mente una cosa importante: essendo le condition variable implicite in java, nel momento in cui si lancia una notify, questa notifica arriverà a tutti i thread, sia consumatori che produttori. Sarà poi interpretata dalla VM che deciderà se svegliare un produttore o un consumatore. Nel caso in cui, ad esempio, un produttore, lanciando una notify, svegliasse un altro produttore, quest'ultimo non potrebbe accedere nuovamente al buffer in quanto occupiedBuffer sarebbe ancora ad 1 come valore. Soltanto quando la condizione del while non è più vera si può passare alle operazioni successive.

Vediamo ora il metodo get:

```

// return value from buffer
public synchronized int get()
{
    // il nome del thread che ha richiesto il metodo get()
    String name = Thread.currentThread().getName();

    // new-SYNCH: controllo se il buffer è vuoto e aspetto
    while (occupiedBuffers == 0 ) {
        // wait
        try {
            showMessage("-- " + name + " tenta di leggere.");
            showMessage("-- Buffer vuoto... " + name +" waits..");
            wait();
        } // se il thread viene interrotto durante il waiting...
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    // se sono qui il Buffer contiene un valore
    // decremento così che il Producer possa scrivere
    --occupiedBuffers;

    showMessage(name + " ha letto " + buffer + " --> notify()");
    notify();
    return buffer;
} // end method get

// metodo non synchronized, invocato da dentro il monitor
public void showMessage( String message ) {
    System.err.println( message );
} // end method showMessage

} // end class SynchronizedBuffer

```

---

Questo metodo è anch'esso, ovviamente, synchronized per garantire la mutua esecuzione. Anche qui abbiamo un controllo per verificare la presenza di dati o meno nel buffer. All'interno del while abbiamo il solito blocco try catch con il wait(). Dopo il while troviamo le operazioni da fare dentro il monitor, quindi nel caso del consumatore, la lettura, in maniera analoga a quanto avveniva prima. Vediamo un risultato di una esecuzione:

```

r-Synchronized$ java SharedBufferTest_Synchronized
Producer ha scritto 1 --> notify()
Consumer ha letto 1 --> notify()
Producer ha scritto 2 --> notify()
Consumer ha letto 2 --> notify()
-- Consumer tenta di leggere.
-- Buffer vuoto... Consumer waits..
Producer ha scritto 3 --> notify()
Consumer ha letto 3 --> notify()
-- Consumer tenta di leggere.
-- Buffer vuoto... Consumer waits..
Producer ha scritto 4 --> notify()
Consumer ha letto 4 --> notify()
Consumer read values totaling: 10.
Terminating Consumer.
Producer done producing.
Terminating Producer.

```

### Buffer circolare con i monitor

Vediamo un esempio di buffer circolare implementato con i monitor, tenendo conto che producer e consumer sono uguali a prima (*vedi lezione 17*):

```
// set up the producer and consumer threads and start them
public class CircularBufferTest
{
    public static void main ( String args[] )
    {
        // create shared object for threads
        CircularBuffer sharedLocation = new CircularBuffer();

        // display initial state of buffers in CircularBuffer
        System.err.println( sharedLocation.createStateOutput() );

        // create producer and consumer objects
        Producer producer = new Producer( sharedLocation );
        Consumer consumer = new Consumer( sharedLocation );

        producer.start();
        consumer.start();
    } // end main

} // end class CircularBufferTest
```

Nel main non troviamo nulla di nuovo: definizione di un oggetto CircularBuffer e creazione dei due thread produttore e consumatore.

Analizziamo ora la classe circularBuffer, anch'essa implementa l'interfaccia buffer:

```
public class CircularBuffer implements Buffer
{
    // each array element is a buffer
    private int buffers[] = { -1, -1, -1 };

    // occupiedBuffers maintains count of occupied buffers
    private int occupiedBuffers = 0;

    // variables that maintain read and write buffer locations
    private int readLocation = 0;
    private int writeLocation = 0;
```

Come abbiamo visto nelle precedenti lezioni un buffer circolare può essere implementato come array, proprio per questo motivo inizializziamo un array che servirà a rappresentare il buffer. Inoltre inizializziamo la variabile occupiedBuffers che serve a tenere traccia delle celle del buffer che sono occupate. Utilizziamo inoltre due variabili, readLocation e writeLocation, che servono come indici per il thread consumatore e il thread produttore.

Vediamo ora il metodo set:

```

// place value into buffer
public synchronized void set( int value )
{
    // get name of thread that called this method
    String name = Thread.currentThread().getName();

    // while buffer full, place thread in waiting state
    while ( occupiedBuffers == buffers.length ) {
        // output thread and buffer information, then wait
        try {
            System.err.println( "\nAll buffers full. " + name + " waits." );
            wait(); // wait until space is available
        } // end try

        // if waiting thread interrupted, print stack trace
        catch ( InterruptedException exception ) {
            exception.printStackTrace();
        } // end catch
    } // end while

    // place value in writeLocation of buffers
    buffers[ writeLocation ] = value;

    // output produced value
    System.err.println( "\n" + name + " writes " + buffers[ writeLocation ] + " " );

    // indicate that one more buffer is occupied
    ++occupiedBuffers;

    // update writeLocation for future write operation
    writeLocation = ( writeLocation + 1 ) % buffers.length;

    // display contents of shared buffers
    System.err.println( createStateOutput() );

    notify(); // return a waiting thread to ready state
} // end method set

```

La prima istruzione che incontriamo è il while che ci serve per implementare un controllo: infatti sarà possibile effettuare il metodo set, cioè la scrittura, solo quando ci sarà un posto libero nel buffer (occupiedBuffers == buffers.length indica infatti che il buffer è pieno e quindi il produttore entrerà in wait). Per evitare che il produttore scriva quando il buffer è pieno si utilizza un ciclo while sul quale avviene il controllo sulle celle del buffer; come al solito mettiamo un blocco try catch all'interno del while per la gestione di eventuali errori. Dopodiché si va a scrivere, nella posizione puntata dall'indice del thread produttore (writeLocation), il valore che quest'ultimo ha generato e si incrementa il numero di posti occupati nel buffer. Infine si aggiorna l'indice del thread produttore che passerà alla cella successiva ( o alla prima cella del buffer nel caso in cui abbia scritto nell'ultimo posto del buffer).

Fatte queste operazioni, che servono ad implementare il set, si effettua la notify così da svegliare un thread in wait.

Seguendo la stessa logica, ma speculare, implementiamo il get:

```

public synchronized int get()
{
    // get name of thread that called this method
    String name = Thread.currentThread().getName();

    // while buffer is empty, place thread in waiting state
    while ( occupiedBuffers == 0 ) {
        // output thread and buffer information, then wait
        try {
            System.err.println( "\nAll buffers empty. " + name + " waits." );
            wait(); // wait until buffer contains new data
        } // end try

        // if waiting thread interrupted, print stack trace
        catch ( InterruptedException exception ) {
            exception.printStackTrace();
        } // end catch
    } // end while

    // obtain value at current readLocation
    int readValue = buffers[ readLocation ];

    // output consumed value
    System.err.println( "\n" + name + " reads " + readValue + " " );

    // decrement occupied buffers value
    --occupiedBuffers;

    // update readLocation for future read operation
    readLocation = ( readLocation + 1 ) % buffers.length;
}

// display contents of shared buffers
System.err.println( createStateOutput() );

notify(); // return a waiting thread to ready state

return readValue;
} // end method get

```

Vediamo un risultato della esecuzione del programma:

### *Sample Output:*

```

(buffers occupied: 0)
buffers: -1   -1   -1
----- -----
          WR

All buffers empty. Consumer waits.

Producer writes 11
(buffers occupied: 1)
buffers: 11   -1   -1
----- -----
          R   W

```

```
Consumer reads 11
(buffers occupied: 0)
buffers: 11 -1 -1
-----
          WR

Producer writes 12
(buffers occupied: 1)
buffers: 11 12 -1
-----
          R   W

Producer writes 13
(buffers occupied: 2)
buffers: 11 12 13
-----
          W       R
```

### *Sample Output (Cont.).*

```
Consumer reads 12
(buffers occupied: 1)
buffers: 11 12 13
-----
          W       R

Producer writes 14
(buffers occupied: 2)
buffers: 14 12 13
-----
          W       R

Producer writes 15
(buffers occupied: 3)
buffers: 14 15 13
-----
          WR
```

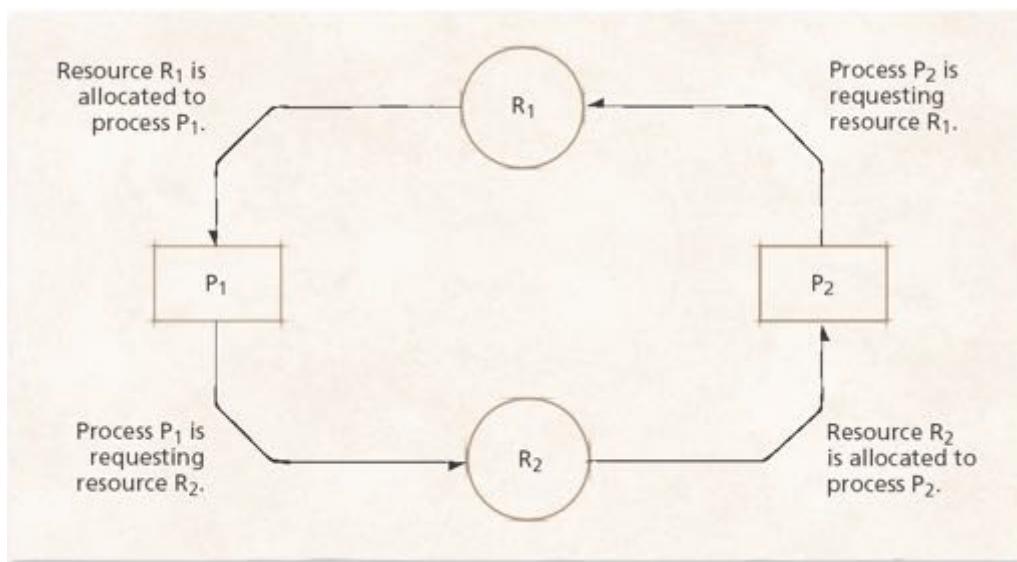
# Lezione 22

## Deadlock e posticipazione indefinita

Si possono verificare delle condizioni in cui i processi, nel tentativo di svolgere una operazione nella sezione critica, rimangano in attesa di un evento che non avverrà mai. Se questo evento non avviene il thread rimarrà bloccato perennemente. Questa condizione è detta **deadlock**. Diciamo che un sistema è in deadlock quando uno o più processi sono in deadlock.

Ci sono delle situazioni in cui si può però superare un deadlock.

I deadlock dipendono spesso dalla implementazione della mutua esclusione. Vediamo un grafo che mette in relazione processi e risorse:



Questo grafo mostra due processi rappresentati da rettangoli e due risorse rappresentate da cerchi. La freccia che collega una risorsa a un processo indica che la risorsa appartiene al processo o comunque gli è stata allocata, viceversa, la freccia che collega il processo alla risorsa indica che il processo necessita di quella risorsa per continuare la sua esecuzione.

In questo caso il processo P1 possiede la risorsa R1 e per continuare necessita di R2, mentre P2 possiede R2 ma per continuare necessita R1. Ciascun processo è in attesa che l'altro liberi una risorsa che non è stata liberata.

Questa situazione prende il nome di **attesa circolare**, ed è una delle tante situazioni di deadlock che possono manifestarsi.

Un altro problema che può nascere è la **posticipazione indefinita**, e dipende da come la risorsa condivisa e gestita e dallo scheduler. Una posticipazione indefinita avviene quando una determinata politica di scheduling va a favorire solo una particolare tipologia di thread, svantaggiandone di conseguenza altri. In questa maniera avremo dei thread che verranno sempre eseguiti ed altri che invece verranno eseguiti dopo lassi di tempo molto elevati o addirittura infiniti.

Un metodo per prevenire la posticipazione indefinita è chiamato **Aging**, ed è un metodo che serve a cambiare la priorità dei processi proporzionalmente al tempo che hanno passato in attesa nel sistema.

## Il problema dei 5 filosofi

È un tipico problema che chiarisce la differenza tra deadlock e posticipazione indefinita.

Abbiamo 5 filosofi seduti attorno al tavolo che alternano la fase in cui pensano a quella in cui mangiano. Nella fase in cui deve mangiare un filosofo deve accedere al piatto, tramite le posate, che nel tavolo sono 5 in totale. Tuttavia per mangiare un filosofo ha bisogno di due posate. Si deve trovare un modo per consentire a tutti i filosofi di mangiare, senza che vi sia deadlock o posticipazione indefinita.



Vediamo uno pseudocodice che riassume grossolanamente quali sono le due operazione che avvengono in questo problema:

```

1 void typicalPhilosopher()
2 {
3     while ( true )
4     {
5         think();
6         eat();
7     } // end while
8
9 } // end typicalPhilosopher

```

Ovviamente l'operazione che deve prevedere un controllo delle risorse è eat().

Una prima versione proposta è la seguente:

```

1 typicalPhilosopher()
2 {
3     while ( true )
4     {
5         think();
6
7         pickUpLeftFork();
8         pickUpRightFork();
9
10        eat();
11
12        putDownLeftFork();
13        putDownRightFork();
14    } // end while
15
16 } // end typicalPhilosopher

```



L'idea consiste nell'inserire, dopo la fase di think(), una fase in cui il filosofo prova a prendere la posata di sinistra e poi quella di destra, successivamente mangia e posa le forchette nello stesso ordine in cui le ha prese. Questa soluzione però causa un deadlock poiché, se tutti i filosofi prendessero la forchetta di sinistra, allora nessuno di loro potrebbe mangiare in quanto non sarebbero rimaste forchette a destra. Vediamo una possibile implementazione in java di questa versione del problema utilizzando i monitor: il monitor dovrà proteggere l'accesso alla risorsa condivisa, quindi in questo caso alle forchette che si utilizzano per mangiare appunto. Per fare ciò dobbiamo dichiarare come synchronized i metodi che utilizziamo per accedere alla risorsa, ovvero pickUpFork e putDownFork. Il thread dovrà provare ad eseguire il metodo per prendere la posata e contemporaneamente dovrà fare in modo che nessun'altro possa prendere la forchetta. Il codice è il seguente:

```
1 // Soluzione #1 al problema dei cinque filosofi (deadlock)
2 public class Main {
3     public static void main(String[] args) {
4         int numFilosofi = 5; //Quanti filosofi
5         Filosofo filosofi[] = new Filosofo[numFilosofi];
6         Bacchetta bacchette[] = new Bacchetta[numFilosofi];
7
8         // Inizializzo tutte le bacchette
9         for (int i = 0; i < 5; i++) {
10             bacchette[i] = new Bacchetta(i);
11         }
12
13         // Inizializzo tutti i filosofi
14         for (int i = 0; i < 5; i++) {
15             int sinistra, destra; // Quali bacchette questo filosofo può prendere?
16             +
17             // Calcoliamo le bacchette per questo filosofo
18             sinistra = i - 1;
19             if (sinistra < 0)
20                 sinistra = numFilosofi - 1;
21             destra = i;
22
23             // Creo il filosofo e lancio il thread
24             filosofi[i] = new Filosofo(i, bacchette[sinistra], bacchette[destra]);
25             filosofi[i].start();
26         }
27     }
28 }
```

Cominciamo dichiarando gli oggetti filosofi e bacchette (che sono array), inoltre, per ognuno dei filosofi andremo a dichiarare due variabili che specificano quali sono le bacchette che devono utilizzare (sinistra e destra). Per ciascun filosofo la bacchetta di sinistra sarà quella di indice  $i - 1$ , mentre la bacchetta di destra avrà indice  $i$ , questa operazione avviene dentro il primo for. Dopo questa operazione, nel secondo for, utiliziamo un if per assegnare ad un filosofo una bacchetta, inoltre si va a creare il thread filosofo di posto  $i$  che viene avviato.

Vediamo come è implementata la classe filosofo:

```

1 // La classe che implementa il filosofo
2 public class Filosofo extends Thread {
3     private int IDF; // L'ID di questo filosofo
4     private final int NSEC = 2; // Quanto tempo ha (al max) per pensare o mangiare
5     private Bacchetta bacchettaSin; // Bacchetta sinistra
6     private Bacchetta bacchettaDes; // Bacchetta sinistra
7
8
9     // Costruttore
10    public Filosofo(int ID, Bacchetta sin, Bacchetta des) {
11        IDF = ID; //Quale filosofo
12        this.setName("Filosofo #" + IDF); //Diamo un nome al filosofo
13        bacchettaSin = sin; // Oggetto condiviso: la bacchetta sinistra
14        bacchettaDes = des; // Oggetto condiviso: la bacchetta destra
15    }
16    // Metodo principale del thread...
17    public void run() {
18        for (;;) {
19            // Creiamo un loop infinito che modella la vita di un filosofo ;-
20            this.pensa();
21            this.prendiBacchette();
22            this.mangia();
23            this.rilasciaBacchette();
24        }
25    }
26}

```

Nel costruttore del filosofo passo l'id del filosofo e le posate che usa, di indici diversi. All'inizio abbiamo la dichiarazione delle variabili che servono per il costruttore, ovvero id del filosofo, posata sinistra e posata destra. Nel metodo run implementiamo il metodo pensa(), seguito dal metodo prendiBacchette(), mangia() e rilasciaBacchette().

Fatto ciò creiamo la classe pensa() che a livello pratico non farà nulla se non stampare a schermo un messaggio e mandare il thread per qualche secondo. Vediamolo:

```

18    // Il filosofo pensa per un tempo random tra 0 e NSEC secondi
19    private void pensa() {
20        System.out.println(this.getName() + " sta pensando...");
21        try {
22            Thread.sleep(Math.round(Math.random() * NSEC * 1000));
23        } catch (InterruptedException e) {
24            e.printStackTrace();
25        }
26    }

```

A questo punto possiamo vedere i metodi che abbiamo già notato all'interno del run(). Partiamo analizzando il metodo prendiBacchette():

```

29  // Il filosofo ha fame, e vuole prendere le bacchette
30  // Prima a sinistra, poi a destra
31  // Possibilità di DEADLOCK!
32  // Se volete simulare il deadlock ponete NSEC a 0, ricompilate ed eseguite
33  private void prendiBacchette() {
34      System.out.println(this.getName() + " ha fame...");
35
36      System.out.println(this.getName() + " sta cercando di prendere la bacchetta "
37                      + bacchettaSin.bacchettaNum + " alla sinistra");
38      // Prendi la bacchetta sinistra (o aspetta se è occupata)
39      bacchettaSin.prendi();
40      // Ho avuto la bacchetta
41      System.out.println(this.getName() + " ha preso la bacchetta "
42                      + bacchettaSin.bacchettaNum);
43
44      System.out.println(this.getName() + " sta cercando di prendere la bacchetta "
45                      + bacchettaDes.bacchettaNum + " alla destra");
46      // Prendi la bacchetta destra (o aspetta se è occupata)
47      bacchettaDes.prendi();
48      // Ho avuto la bacchetta
49      System.out.println(this.getName() + " ha preso la bacchetta "
50                      + bacchettaDes.bacchettaNum);
51  }

```

Nel metodo troviamo fondamentalmente due elementi: delle print per visualizzare a schermo dei messaggi e un ulteriore metodo prendi() che serve a prendere prima la posata sinistra e poi quella di destra, approfondiremo dopo l'utilità di questo metodo. Vediamo ora il metodo mangia(), praticamente uguale a pensa(), seguito dal metodo usato per rilasciare le bacchette rilasciaBacchette():

```

54  // Il filosofo mangia per un tempo random tra 0 e NSEC secondi
55  private void mangia() {
56      System.out.println(this.getName() + " sta mangiando...");
57      try {
58          Thread.sleep(Math.round(Math.random() * NSEC * 1000));
59      } catch (InterruptedException e) {
60          e.printStackTrace();
61      }
62  }
63
64  // Il filosofo rilascia le bacchette in suo possesso
65  private void rilasciaBacchette() {
66      System.out.println(this.getName() + " ha finito di mangiare.");
67      bacchettaSin.rilascia();
68      bacchettaDes.rilascia();
69  }

```

Il metodo per rilasciare le bacchette prevede un ulteriore metodo al suo interno di nome rilascia() il cui scopo è l'opposto del metodo prendi() visto dentro prendiBacchette(), ovvero per rilasciare singolarmente le posate. Vediamo ora come è fatta la classe bacchetta, la quale implementa il monitor:

```

1 // Classe che modella le bacchette presenti sul tavolo
2 // Questa è la risorsa condivisa - bisogna sincronizzare gli accessi
3 public class Bacchetta{
4     // Un bool di int per monitorare se la bacchetta è libera o meno
5     private boolean bacchettaLibera;
6     // Quale bacchetta?
7     public int bacchettaNum;
8
9
10    // Costruttore
11    public Bacchetta(int quale){
12        bacchettaNum = quale;
13        bacchettaLibera = true;
14    }
15

```

Le due variabili all'inizio indicano se la bacchetta è occupata o meno (bacchettaLibera), e inoltre quale è la bacchetta che è stata presa (bacchettaNum). Dopodiché c'è il metodo prendi(), ed è qui che andremo a implementare il monitor:

---

```

17 // Metodo sincronizzato per prendere la bacchetta
18 public synchronized void prendi(){
19     while (!bacchettaLibera){
20         // La bacchetta è occupata, dobbiamo aspettare
21         try{
22             System.out.println(Thread.currentThread().getName()
23             + " sta aspettando che si liberi la bacchetta " + bacchettaNum);
24             wait();
25         }
26         catch(InterruptedException e){
27             e.printStackTrace();
28         }
29     }
30     // Rendi questa bacchetta non disponibile
31     bacchettaLibera = false;
32 }
33
34
35 // Metodo sincronizzato per rilasciare le bacchette
36 public synchronized void rilascia(){
37     // Rendi questa bacchetta disponibile
38     bacchettaLibera = true;
39     // Notifco gli altri filosofi
40     notifyAll();
41 }
42 }

```

---

I due metodi rilascia() e prendi() sono dichiarati come synchronized poiché devono essere eseguiti in maniera mutuamente esclusiva, d'altronde, se ci pensiamo, la risorsa condivisa in questo problema sono proprio le bacchette. Nel metodo prendi prevediamo un ciclo che lasci il thread in wait quando la bacchetta scelta per mangiare è occupata, superato questo ciclo la bacchetta sarà presa dal filosofo e diventerà non disponibile (bacchettaLibera = false). Nel metodo rilascia viene resa disponibile la bacchetta (bacchettaLibera = true) e in seguito lanciata la notifyAll per svegliare uno degli altri thread.

Vediamo una esecuzione del codice:

```

Filosofo #0 sta pensando...
Filosofo #2 ha fame...
Filosofo #1 ha fame...
Filosofo #4 ha fame...
Filosofo #0 ha fame...
Filosofo #1 sta cercando di prendere la bacchetta 0 alla sinistra +
Filosofo #3 sta cercando di prendere la bacchetta 2 alla sinistra
Filosofo #2 sta cercando di prendere la bacchetta 1 alla sinistra
Filosofo #4 sta cercando di prendere la bacchetta 3 alla sinistra
Filosofo #0 sta cercando di prendere la bacchetta 4 alla sinistra
Filosofo #1 ha preso la bacchetta 0
Filosofo #1 sta cercando di prendere la bacchetta 1 alla destra
Filosofo #3 ha preso la bacchetta 2
Filosofo #2 ha preso la bacchetta 1
Filosofo #2 sta cercando di prendere la bacchetta 2 alla destra
Filosofo #3 sta cercando di prendere la bacchetta 3 alla destra
Filosofo #1 sta aspettando che si liberi la bacchetta 1
Filosofo #3 sta aspettando che si liberi la bacchetta 3
Filosofo #2 sta aspettando che si liberi la bacchetta 2
Filosofo #4 ha preso la bacchetta 3
Filosofo #4 sta cercando di prendere la bacchetta 4 alla destra
Filosofo #4 sta aspettando che si liberi la bacchetta 4
Filosofo #0 ha preso la bacchetta 4
Filosofo #0 sta cercando di prendere la bacchetta 0 alla destra
Filosofo #0 sta aspettando che si liberi la bacchetta 0

```

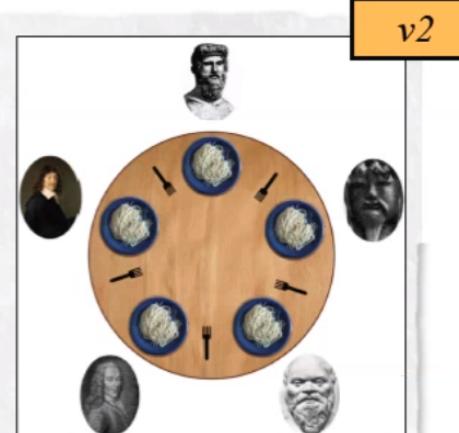
Notiamo come ad un certo punto della esecuzione il sistema si blocca poiché tutti i vari thread hanno occupato risorse che non libereranno mai nell'attesa di successive altre risorse che non potranno ottenere in quanto occupate dagli altri thread filosofi.

Vediamo la seconda versione del problema:

```

1 typicalPhilosopher()
2 {
3     while ( true )
4     {
5         think();
6
7         pickUpBothForksAtOnce(); +
8
9         eat();
10
11        putDownBothForksAtOnce();
12    } // end while
13
14 } // end typicalPhilosopher

```



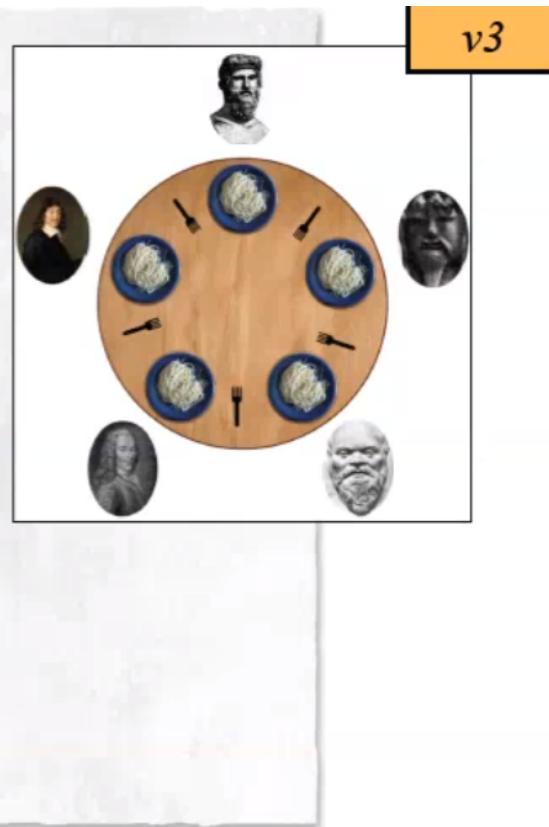
Questa soluzione prevede che i filosofi prendano entrambe le forchette contemporaneamente per evitare che altri filosofi le rubino prima. Ma questo comporta un problema, in quanto, al massimo, solo 2 filosofi su 5 riescono a prendere le risorse e a mangiare, mentre gli altri 3 subiranno una posticipazione indefinita. Non abbiamo infatti la garanzia che un filosofo, dopo aver mangiato, posa le bacchette e non le riprenda. Se dovesse posarle e riprendere subito dopo gli altri filosofi andrebbero incontro ad una posticipazione delle risorse.

Vediamo la soluzione 3:

```

1 typicalPhilosopher()
2 {
3     while ( true )
4     {
5         think();
6
7         while ( notHoldingBothForks )
8         {
9             pickUpLeftFork();
10
11            if ( rightForkNotAvailable )
12            {
13                putDownLeftFork();
14            } // end if
15            else
16            {
17                pickUpRightFork();
18            } // end while
19        } // end else
20
21        eat();
22
23        putDownLeftFork();
24        putDownRightFork();
25    } // end while
26
27 } // end typicalPhilosopher

```



v3

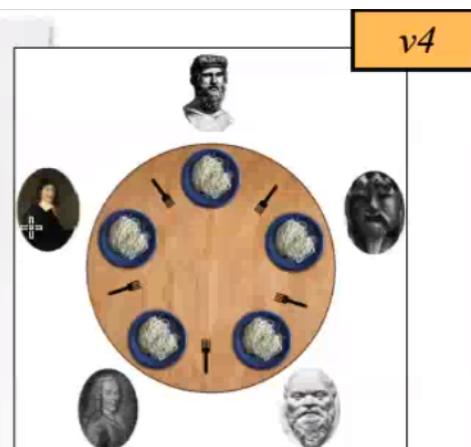
Un modo per risolvere il problema della soluzione 1 è quello di fare in modo che il filosofo prenda le forchette solo esclusivamente se entrambe sono libere, qualora una delle due fosse usata, allora il filosofo rilascerebbe l'altra forchetta rilasciando così una risorsa per renderla disponibile agli altri thread. Quello che potrebbe accadere è che un filosofo prenda la forchetta sinistra, si accorga che quella destra è occupata, e di conseguenza posi la forchetta sinistra. Ma questa situazione potrebbe ripetersi più e più volte e così il filosofo continuerebbe a prendere e lasciare la forchetta.

Vediamo infine l'ultima implementazione:

```

1 typicalPhilosopher()
2 {
3     while ( true )
4     {
5         think();
6
7         if ( philosopherID mod 2 == 0 )
8         {
9             pickUpLeftFork();
10            pickUpRightFork();
11
12            eat();
13
14            putDownLeftFork();
15            putDownRightFork();
16        } // end if
17        else
18        {
19            pickUpRightFork();
20            pickUpLeftFork();
21
22            eat();
23
24            putDownRightFork();
25            putDownLeftFork();
26        } // end else
27    } // end while
28
29 } // end typicalPhilosopher

```



v4

Questa versione prevede di definire una sorta di ordine nel flusso di esecuzione delle operazioni dei filosofi. In questo caso viene eliminata l'attesa circolare con un controllo: se sei un filosofo con indice pari devi provare a prendere prima la forchetta di sinistra e poi quella di destra, viceversa se sei un filosofo di indice dispari, prima provi a prendere prima una forchetta di destra

e poi quella di sinistra. In questa maniera si spezza l'attesa circolare, evitando così il deadlock. Tuttavia anche in questo caso si può verificare la posticipazione indefinita, che si ha quando alcuni filosofi ottengono una forchetta ma non riescono ad ottenere l'altra poiché contesa da un altro filosofo.

# Lezione 23

---

Per capire se un codice presenta possibilità di deadlock basta capire se si verificano le 4 condizioni necessarie:

- La presenza di **condizioni di mutua esclusione**. Il fatto che una risorsa possa essere bloccata da un processo automaticamente comporta la possibilità che si blocchino degli altri processi.
- La **condizione di wait-for**: si può verificare il caso in cui un processo, ottenuta una risorsa, per completare un task deve chiedere alcune risorse, che se non ottiene dovrà aspettare. Questo accentua la possibilità di deadlock.
- La **condizione di no-preemption**: nel momento in cui un processo ha ottenuto una risorsa e la mantiene per troppo tempo, il sistema operativo potrebbe forzare il processo a liberare la risorsa, ma se non c'è modo di forzare quel processo ad abbandonare quella risorsa allora si verificherà una no-preemption. Questo avviene spesso quando i processi mascherano gli interrupt.
- La **condizione di attesa circolare** (*vista nella lezione 22*): due o più processi si trovano in una situazione di attesa reciproca nella quale ogni processo sta aspettando una o più risorse allocate dai processi successivi.

Se un codice verifica queste condizioni, allora la possibilità che vi sia deadlock è alta. Prese tutte insieme queste 4 condizioni non sono solo necessarie, ma anche sufficienti per l'esistenza dei deadlock.

Di contro, per evitare che si verifichi deadlock, ci possiamo assicurare che nel nostro codice non vi siano queste 4 condizioni qua sopra (ad esclusione di quella di mutua esclusione che è necessaria per implementare il parallelismo).

Dividiamo lo studio del deadlock in 4 aree: **deadlock prevention**, **deadlock avoidance**, **deadlock detection** e **deadlock recovery**.

## Deadlock prevention

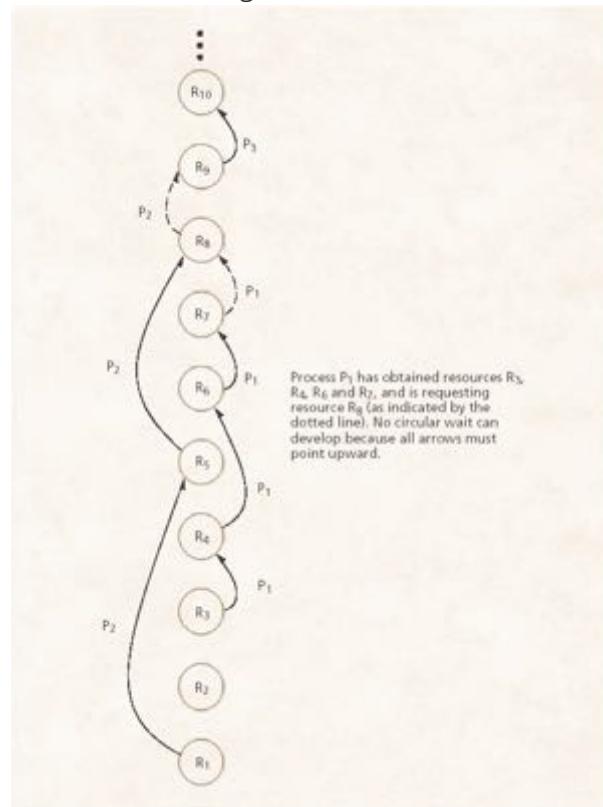
Per prevenire il deadlock possiamo andare a rimuovere alcune delle 4 condizioni viste sopra. La prima non può essere eliminata ma le altre 3 condizioni possono essere gestite mediante specifiche strategie, analizziamone una per una, caso per caso:

- Per prevenire il wait-for dobbiamo fare in modo che un processo chieda contemporaneamente (nello stesso istante) tutte le risorse prima di iniziare le sue operazioni. Questo significa che se tutte le risorse non sono disponibili, il processo non andrà a occupare risorse parziali. Il processo dovrà quindi verificare la disponibilità di tutte le risorse prima di andare ad occuparne altre.  
Questa soluzione diminuisce certamente la probabilità di deadlock, ma non elimina il problema della posticipazione indefinita in quanto i processi che richiedono meno risorse saranno eseguiti più frequentemente dei processi che richiedono un quantitativo di risorse maggiore.
- Per prevenire la condizione di no-preemption possiamo forzare i processi a liberare le risorse per evitare che le blocchino perennemente, nello specifico si richiede che, quando a un processo che detiene delle risorse venga negata la richiesta di risorse aggiuntive, esso debba rilasciare quelle che detiene e, se necessario, effettuare una seconda richiesta che comprenda anche quelle aggiuntive. Questo però può portare alla perdita del lavoro svolto da un processo, aggiungendo overhead totale in quanto questa operazione ha un costo sul sistema. Anche in questo caso abbiamo problemi di posticipazione indefinita in quanto se

un processo occupa poche risorse, è meno probabile che quelle risorse siano chieste da altri processi; anche in questo caso i processi con più risorse sono penalizzati.

- Per prevenire la attesa circolare possiamo etichettare le risorse e consentire ad un determinato processo di ottenere le risorse in ordine crescente seguendo un ordinamento lineare, cioè, dobbiamo assegnare un ID ad ogni risorsa, allora potremo dire al processo di fare richieste delle risorse rispettando l'ordine degli ID. Questo metodo però è poco adatto a sistemi particolarmente complessi che richiedono molte risorse, poiché il sistema dovrebbe prevedere la creazione e l'assegnamento degli ID delle nuove risorse.

Graficamente la soluzione potrebbe essere la seguente:



Il processo P1 ha ottenuto R3, R4, R6 ed R7. A questo punto sta chiedendo la risorsa R8 ma visto che tutte le frecce vanno verso l'alto non si può creare attesa circolare, Potranno quindi nascere condizioni di attesa, ma sicuramente non circolare.

### Deadlock avoidance

Possiamo utilizzare l'**algoritmo del banchiere**, proposto da dijkstra, per evitare il deadlock. Il concetto dell'algoritmo è simile al concetto di "prestito" classico delle banche. Vediamo come funziona:

L'idea è quella di dare all'OS un certo numero di risorse condivise  $t$ , condivise tra  $n$  processi. Ogni processo prima di essere eseguito può specificare quali sono le risorse di cui ha bisogno, allora l'OS accetta questa richiesta se la richiesta di risorse è inferiore al numero di risorse disponibili, se queste risorse non sono attualmente disponibili, allora il processo viene messo in attesa, ma l'algoritmo garantisce che questa attesa sia finita fino alla disponibilità delle risorse. Il sistema garantisce inoltre che una volta allocate le risorse ad un certo processo, questo processo le libererà entro un certo arco di tempo definito. Se si verificano tutte queste condizioni è garantito che il sistema non presenterà deadlock.

Questo algoritmo gestisce il sistema basandosi su due stati:

- **Safe state:** l'OS garantisce che tutti i processi sono in grado di completare il loro task in un tempo finito.

- **Unsafe state:** alla luce dei prestiti attualmente fatti, l'OS non garantisce che tutti i processi riescano a finire i loro task in tempo finito. Pure essendo in uno stato unsafe non è sicuro che vi sia deadlock, ma segnala la possibilità che il deadlock si verifichi.

Secondo questo criterio l'algoritmo prevede che le risorse vengano allocate solamente se allocando quelle risorse, lo stato del sistema rimanga safe.

Questo algoritmo, purtroppo, non è applicabile a tutti i sistemi, ma solamente a quelli molto statici (come gli embedded). Nei sistemi dinamici il numero di risorse varia continuamente e di conseguenza l'algoritmo non si potrà applicare.

Per studiare in maniera approfondita l'algoritmo dobbiamo analizzare come avviene la distribuzione delle risorse:

Dato un processo  $P_i$ , questo processo durante la sua esecuzione andrà a chiedere un certo numero di risorse. Il numero massimo di risorse che il processo chiederà durante la sua esecuzione verrà indicato con  $\max(P_i)$ , ad un certo punto il processo avrà ottenuto delle risorse, che saranno indicate con  $\text{loan}(P_i)$ , tuttavia per raggiungere il valore  $\max(P_i)$  necessario a completare l'esecuzione serviranno altre risorse che prendono il nome di  $\text{claim}(P_i)$  (che sarà uguale a  $\max - \text{loan}$ ). Allora in un certo istante il valore di risorse del sistema che sono ancora disponibili sarà dato da:

$$a = t - \sum_{n=i}^n \text{loan}(P_i)$$

La variabile  $a$  rappresenta il numero di risorse ancora disponibili nel sistema tenendo conto delle risorse che sono state prestate al processo  $P_i$ , ricordando che  $t$  è il numero di risorse totali a disposizione del sistema.

Vediamo un eSEMPIO numerico:

<i>Process</i>	<i>max(<math>P_i</math>) (maximum need)</i>	<i>loan(<math>P_i</math>) (current loan)</i>	<i>claim(<math>P_i</math>) (current claim)</i>
$P_1$	4	1	3
$P_2$	6	4	2
$P_3$	8	5	3
<i>Total resources, <math>t = 12</math></i>		<i>Available resources, <math>a = 2</math></i>	

Supponiamo di avere 3 processi e un totale di risorse di sistema pari a 12. Attualmente queste 12 risorse sono allocate in un certo modo che vediamo in figura. Il processo  $P_1$  ha bisogno di 4 risorse,  $P_2$  ha bisogno di 6 risorse e  $P_3$  ha bisogno di 8. Se vado a sommare le richieste di questi processi, il risultato, sarà maggiore delle risorse totali di sistema ( $18 > 12$ ). Decido di assegnare le risorse "a pezzi", a  $P_1$  assegno 1 risorsa, a  $P_2$  ne assegno 4, a  $P_3$  ne assegno 5; attualmente ho assegnato 10 risorse in totale e ne rimarranno disponibili 2 ( $t - \text{loan}(P_i)$ ).

Avendo due risorse disponibili devo capire se il sistema garantisce che tutti i processi siano eseguiti. Con due risorse disponibili si può completare il processo  $P_2$ , quindi assegno le risorse a  $P_2$  che si completa. Se  $P_2$  completa il suo task significa che si libereranno 6 risorse le quali torneranno disponibili.

Adesso le risorse a disposizione saranno 6, che basteranno per completare gli altri due processi ( $P_1$  e  $P_3$ ). In tutto questo arco di tempo il sistema sarà in stato safe perché potrà sempre garantire il completamento dei processi.

Altro eSEMPIO:

Process	$\max(P_i)$ (maximum need)	$\text{loan}(P_i)$ (current loan)	$\text{claim}(P_i)$ (current claim)
P <sub>1</sub>	10	8	2
P <sub>2</sub>	5	2	3
P <sub>3</sub>	3	1	2
Total resources, t, = 12	Available resources, a, = 1		

Stesso esempio di prima, ma cambiano le richieste dei singoli processi e la allocazione momentanea delle risorse. In questo caso, le risorse disponibili non basteranno a completare l'esecuzione di alcun processo. Quindi siamo in uno stato unsafe, ovvero uno stato che potrebbe portare il sistema in deadlock. Teniamo però a mente che solo alcune specifiche sequenze di eventi portano al deadlock.

Mentre nel primo esempio, a prescindere, ho risorse sufficienti, nel secondo caso invece si potrebbe verificare deadlock.

### Transizioni di stato

Lo stato del sistema comunque si può sempre alternare, con delle transizioni di stato. La politica di allocazione delle risorse deve considerare attentamente le richieste prima di soddisfarle, altrimenti un processo in uno stato safe rischia di entrare in uno unsafe.

Ad esempio se si passasse dal seguente stato:

Process	$\max(P_i)$ (maximum need)	$\text{loan}(P_i)$ (current loan)	$\text{claim}(P_i)$ (current claim)
P <sub>1</sub>	4	1	3
P <sub>2</sub>	6	4	2
P <sub>3</sub>	8	5	3
Total resources, t, = 12	Available resources, a, = 2		

A questo:

Process	$\max(P_i)$ (maximum need)	$\text{loan}(P_i)$ (current loan)	$\text{claim}(P_i)$ (current claim)
P <sub>1</sub>	4	1	3
P <sub>2</sub>	6	4	2
P <sub>3</sub>	8	6	2
Total resources, t, = 12	Available resources, a, = 1		

Avremmo una transizione di stato da safe ad unsafe. L'algoritmo deve quindi prevedere tutte le possibili combinazioni che portano a transizioni di stato. In questo caso sarebbe compito del sistema forzare P3 a mantenere il prestito di sole 5 risorse e non 6, poiché aumentando le risorse prestate, si entrerebbe in uno stato unsafe.

Scopo principale dell'algoritmo è quindi quello di rimanere il più possibile nello stato safe.

Notiamo che in questo algoritmo:

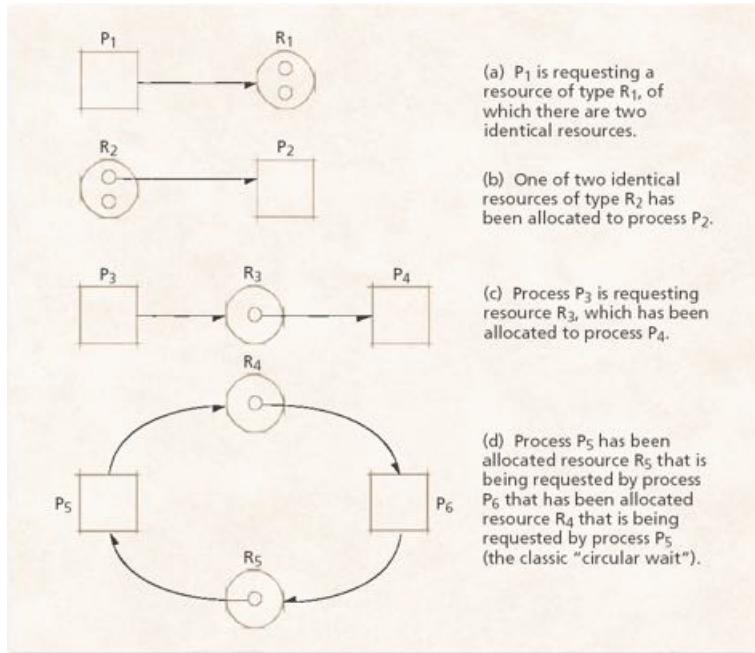
- La mutua esclusione ovviamente avviene.
- La condizione di wait-for è ammessa.
- La condizione di no-preemption è anch'essa ammessa.

Andando a posticipare l'esecuzione delle richieste unsafe si garantisce che il sistema rimanga in stato safe e che le richieste si possano completare in un certo arco di tempo. Ciò garantisce la assenza di deadlock. Rispetto alla strategia di deadlock prevention, questo algoritmo permette di eseguire dei processi senza causare deadlock, la strategia di prevenzione infatti va a valutare il presentarsi di certe condizioni, che se si verificano, impediscono l'esecuzione del processo;

viceversa, in questo caso, evitando a monte il problema, il processo viene eseguito sempre seguendo un certo ordine che elimina il deadlock. Abbiamo però dei contro dati dal fatto che i processi devono essere completati in un tempo finito che può essere eccessivamente lungo. Non abbiamo infatti garanzie che questo tempo sia sufficientemente piccolo.

### Deadlock detection

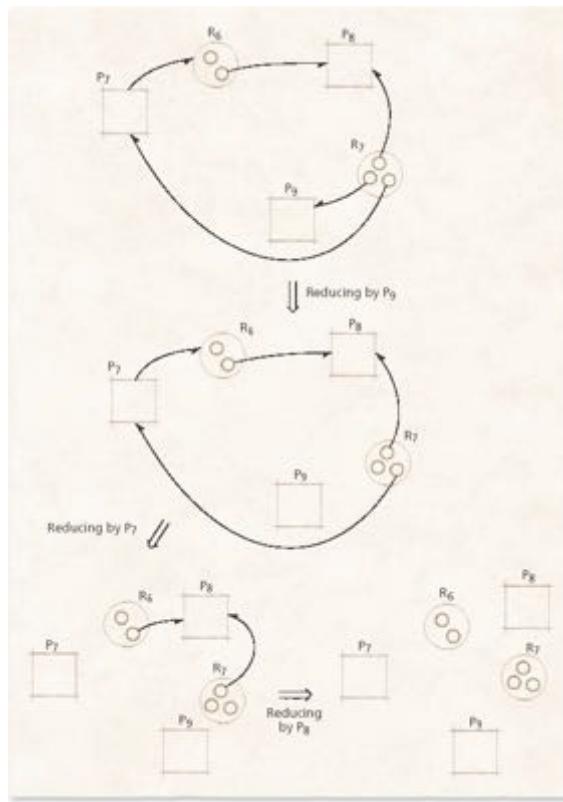
Questa strategia consiste nel capire se si è verificato un deadlock e identificare quali sono i processi e le risorse che hanno causato il deadlock. Per fare questo tipo di analisi si utilizzano dei grafi di allocazione delle risorse dove vengono rappresentati i processi, le risorse e le relative dipendenze. Ad esempio:



Per rappresentare questo grafo delle risorse si usa una notazione in cui i processi sono i quadrati e le risorse sono i cerchi al cui interno si trovano le istanze della risorsa stessa indicate come cerchi più piccoli. Analizziamo i vari esempi: nel primo disegno vediamo come il processo P1 stia richiedendo una delle due istanze della risorsa R1, viceversa, nel secondo disegno vediamo che una specifica istanza della risorsa R2 viene assegnata al processo P2, quindi non vi è alcun problema. Nel terzo disegno vediamo una situazione di potenziale deadlock poiché la risorsa R3 è assegnata a P4, ma anche P3 ne ha bisogno. Una situazione di questo tipo ci allerta su potenziali deadlock.

Il quarto disegno rappresenta la classica attesa circolare, situazione nella quale il deadlock è garantito.

Una utile tecnica per individuare i deadlock si basa sulla **riduzione dei grafi**, in cui si determinano i processi che possono completare la loro esecuzione e quelli che non possono. Se una richiesta di un processo può essere soddisfatta, si dice che il grafo può essere ridotto, viceversa non potrà essere ridotto. La riduzione dei grafi equivale a mostrare come sarebbe il grafo se al processo fosse permesso di terminare e restituire le risorse al sistema. Un grafo si riduce rimuovendo la freccia dalle risorse verso quel processo e le frecce del processo verso le risorse. Se un grafo è riducibile per tutti i suoi processi allora non ci sarà deadlock, altrimenti i processi non ridotti causeranno deadlock. Notiamo come né il 4o né il 3o disegno della precedente figura possano essere ridotti completamente, ecco perché presentano deadlock. Vediamo ora l'esempio di una riduzione di un grafo:



## Deadlock recovery

Una volta che il sistema entra in deadlock il nostro obiettivo diventa quello di uscire da questa situazione, per farlo dobbiamo distruggere uno o più delle quattro condizioni necessarie.

Il recupero dei deadlock rappresenta comunque un problema non facile da risolvere, in primo luogo, perché non sempre appare chiaro che il sistema sia in deadlock. In secondo luogo perché molti sistemi non prevedono la possibilità di sospendere indefiniteamente un processo, rimuoverlo dal sistema e ripristinarlo in un istante successivo senza perdere il lavoro. Infine il recupero del deadlock è complicato perché prevede l'utilizzo di tantissime risorse con conseguente aumento dell'overhead.

Nei sistemi moderni una possibilità di rimuovere il deadlock è quella di rimuovere forzatamente il processo che ha causato il deadlock reclamandone le risorse. Solitamente il sistema perde il lavoro fatto dal processo, ma gli altri processi potrebbero essere in grado di completarsi. Potrei quindi, eliminare il processo che ha causato il deadlock, o impedire che un thread blocchi alcune risorse mediante l'uso del signal KILL().

Esistono però delle alternative meno brusche e invadenti rispetto alla KILL(), per esempio, il **meccanismo di sospensione/ripristino** che permette al sistema di sospendere momentaneamente un processo, rimuovendo momentaneamente anche le sue risorse e, una volta che quel processo sia diventato safe, fare riprendere il processo senza perdere il lavoro. Questa operazione si implementa solitamente mediante i **checkpoint**: dei salvataggi di sistema che permettono al sistema, qualora venga riavviato, di riprendere la sua esecuzione da lì. Quest'ultima operazione rende il sistema molto robusto ai deadlock ma aumenta notevolmente l'overhead.

Ricordiamo però che l'importanza della stabilità del sistema varia da situazione a situazione. In sistemi critici (come il controllore di un aereo) è meglio avere alto overhead per rimuovere il deadlock, viceversa in sistemi non critici si può preferire un approccio contrario.

# Lezione 24

Oggi vediamo una serie di esempi che possono essere risolti mediante l'utilizzo dei monitor.

## Problema del barbiere

Nel negozio di un barbiere vi è una sala di attesa con un divano contenente al massimo 5 persone. Al negozio lavorano tre barbieri, ciascuno con la propria poltrona per il taglio di capelli. Se un cliente entra nel negozio quando il divano è pieno va via senza attendere il proprio turno. Altrimenti, si accomoda nel divano e attende di essere chiamato da un barbiere. Il barbiere serve per primo il cliente in attesa da più tempo, gli taglia i capelli, e provvede al pagamento e al rilascio della ricevuta.

Si implementi in java la sincronizzazione tra clienti e barbieri usando il costrutto monitor. Si discuta inoltre se la soluzione proposta può presentare posticipazione indefinita o deadlock, e se sì, discutere e realizzare eventuali modifiche per evitarli.

In questo specifico caso possiamo dire che il produttore è il cliente che occupa elementi e il consumatore è il barbiere che libera le risorse.

```
//Classe Main

public class Main{
    public static void main(String[] args){
        Sala sala = new Sala(); //Sala d'attesa

        //Creiamo e inizializziamo i barbieri
        int nBarbieri = 3;
        Barbiere barbieri[] = new Barbiere[nBarbieri];

        for(int i = 0; i < nBarbieri; i++){
            barbieri[i] = new Barbiere(sala, i);
            barbieri[i].start();
        }

        //Facciamo la stessa cosa per i clienti che saranno in numero maggiore
        int nClienti = 1000;
        Cliente clienti[] = new Cliente[nClienti];

        for(int i = 0; i < nClienti; i++){
            clienti[i] = new Cliente(sala, i);
            clienti[i].start();
            try{
                Thread.sleep( (int) (Math.random() * 501));
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }

    }

//Classe Barbiere
```

```

public class Barbiere extends Thread{
    private Sala sala;
    private int id;

    public Barbiere(Sala s, int i){
        super("Barbiere");
        sala = s;
        id = i;
    }

    public void run(){
        while(true){
            System.out.println("Barbiere " + id + " pronto per servire un
cliente");
            sala.serveCliente(id);

            try{
                Thread.sleep( (int) (Math.random() * 501));
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }
    }
}

//classe Cliente

public class Cliente extends Thread{

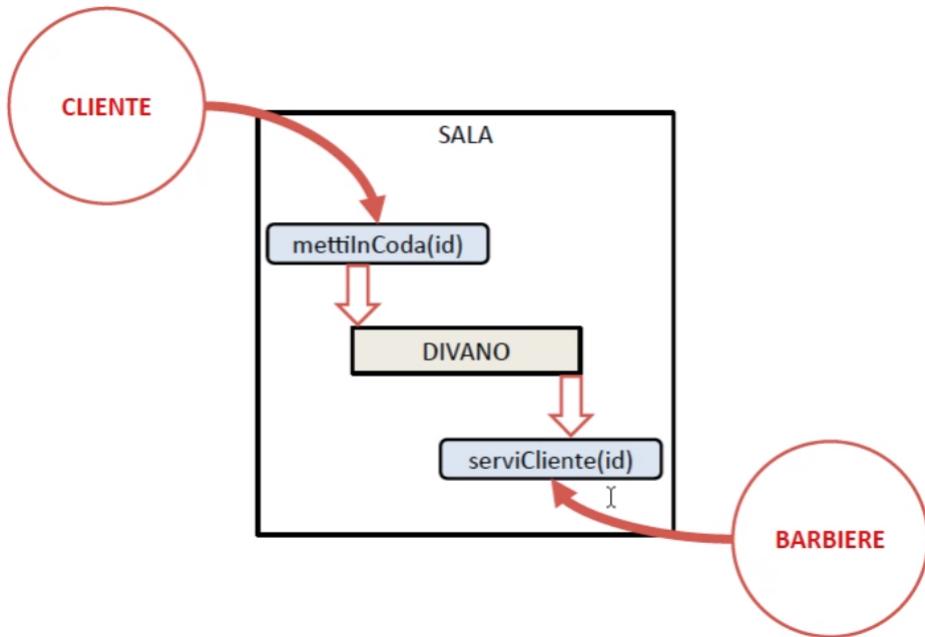
    private Sala sala;
    private int id;

    public Cliente(Sala s, int i){
        super("Cliente");
        sala = s;
        id = i;
    }

    public void run(){
        try{
            Thread.sleep((int) (Math.random() * 1000));
            sala.mettiInCoda(id)
        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }
}

```

La logica del programma segue quindi il seguente schema:



```

//Classe sala
import java.util.LinkedList;

public class Sala {

    //Utilizziamo una linked list perché permette gestione FIFO in maniera
    facile
    //divano è la variabile condivisa
    private int servito;
    private int NUM_POSTI = 5;
    private LinkedList<Integer> divano = new LinkedList<Integer>();

    //Il cliente non fa wait perché se trova il divano occupato se ne va, come
    detto nel testo
    public synchronized void mettiInCoda(int id){
        System.out.println("== arriva il cliente " + id);

        if(divano.size() <= NUM_POSTI){
            divano.add(id);
            notifyAll();
        }else{
            System.out.println("DIVANO PIENO " + divano + " - il cliente " + id
+ " va via");
        }
    }

    //Il barbiere invece esegue il wait in quanto deve aspettare che arrivino
    dei clienti
    //Il barbiere non effettua il notify in quanto i clienti non vanno in wait
    ma se ne vanno
    public synchronized void serveCliente(int id){
        while(divano.size() == 0){
            try{
                System.out.println("Barbiere " + id + "nessun cliente da
servire: wait");
                wait();
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }
    }
}

```

```

        }
    }

    System.out.println("Barbiere " + id + " ottiene il cliente " +
divano.getFirst();
    servito = divano.removeFirst();

    try{
        Thread.sleep((int) (Math.random() * 101));
    }catch(InterruptedException e){
        e.printStackTrace();
    }

    System.out.println("Barbiere " + id + " ha servito il cliente " +
servito);
}
}

```

Proviamo ora ad implementare una versione più complicata dello stesso problema:

Nel negozio di un barbiere vi è una sala d'attesa con un divano contenente al massimo 5 persone [e un'area dove i clienti possono attendere in piedi. Per motivi di sicurezza, all'interno del negozio possono sostare al massimo N persone contemporaneamente].  
Al negozio lavorano tre barbieri, ciascuno con la propria poltrona per il taglio dei capelli.  
Se un cliente entra nel negozio quando il divano è pieno va via senza attendere il proprio turno. Altrimenti, [dappressa si accomoda nell'area di attesa, poi] si accomoda nel divano e [infine] viene chiamato dal barbiere.  
Il barbiere serve per primo il cliente in attesa da più tempo, gli taglia i capelli, e provvede al pagamento e al rilascio della ricevuta. [Poiché vi è soltanto un registro di cassa, i clienti prima di andare via devono aspettare il proprio turno per pagare].  
Si implementi la sincronizzazione tra i clienti e i barbieri usando il costrutto Monitor. Si discuta inoltre se la soluzione proposta può presentare rinvio indefinito e/o deadlock, e se sì, discutere eventuali modifiche per evitarli.

```

//Classe Main

public class Main{

    public static void main(String[] args){

        Barbiere barbieri[] = new Barbiere[5];
        Cliente clienti[] = new Cliente[100];
        Sala sala = new Sala();

        for(int i = 0; i < 3; i++){
            barbieri[i] = new Barbiere(i, sala);
            barbieri[i].start();
        }

        for(int i = 0; i < 100; i++){

            try{
                //La sleep del thread serve a simulare l'arrivo dilazionato dei
                clienti
                Thread.sleep( (int) (Math.random() * 500));
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }
    }
}

```

```

        }

        clienti[i] = new Cliente(i, sala);
        clienti[i].start();

    }

}

//Classe Barbiere

public class Barbiere extends Thread{

    private int id;
    private Sala sala;

    public Barbiere(int id, Sala sala){
        this.id = id;
        this.sala = sala;
        System.out.println("Il barbiere " + id + " ha cominciato a lavorare");
    }

    //Il barbiere serve un cliente e quando finisce puo' farne pagare uno nella
    //coda di pagamento
    public void run(){

        while(true){

            System.out.println("Il barbiere " + id + " puo' servire un
cliente");
            sala.servicliente(this.id);

            try{
                Thread.sleep((int) (Math.random() * 3000));
            }catch(InterruptedException e){
                e.printStackTrace();
            }

            sala.effettuaPagamento(this.id);

        }
    }

}

//Classe Cliente

public class Cliente extends Thread{

    private int id;
    private Sala sala;

    public Cliente(int id, Sala sala){
        this.id = id;
        this.sala = sala;
    }
}

```

```

//I clienti arrivano con frequenza doppia rispetto alla frequenza con la
quale il barbiere serve i clienti
public void run(){
    System.out.println("Il cliente " + this.id + " e' entrato nel negozio");
    sala.mettiInAttesa(this.id);
}
}

//classe Sala

import java.util.LinkedList;

public class Sala{

    //sia la sala di attesa sia la coda di pagamento sono implementate come
    linked list poiche' sono delle code
    private LinkedList<Integer> divano;
    private LinkedList<Integer> sala_attesa;
    private LinkedList<Integer> coda_pagamento;
    private int cliente_servito;
    private int primo_cliente_attesa;

    public Sala(){
        divano = new LinkedList<Integer>();
        sala_attesa = new LinkedList<Integer>();
        coda_pagamento = new LinkedList<Integer>();
    }

    public synchronized void servisciCliente(int id){
        //Se il divano e' vuoto e non ci sono clienti il barbiere non puo'
        lavorare e va in wait
        while(divano.size()==0){

            try{
                System.out.println("Non ci sono clienti nel negozio, il barbiere
" + id + " non puo' lavorare");
                wait();
            }catch(InterruptedException e){
                e.printStackTrace();
            }
        }

        //Se il divano non e' vuoto il barbiere serve il primo cliente in coda
        sul divano che viene rimosso dal divano
        cliente_servito = divano.removeFirst();

        System.out.println("Il barbiere " + id + " ha servito il cliente " +
cliente_servito);

        //Dopo che il cliente e' stato eliminato dal divano il primo cliente in
        sala di attesa passa in coda sul divano, qualora esista
        if(sala_attesa.size()>0){
            primo_cliente_attesa = sala_attesa.removeFirst();
            divano.add(primo_cliente_attesa);
            System.out.println("Il cliente " + primo_cliente_attesa + " si e'
spostato nel divano");
        }
    }
}

```

```
    }

    //Dopo avere servito un cliente quest'ultimo si mette in coda per pagare
    coda_pagamento.add(cliente_servito);
}

public synchronized void mettiInAttesa(int id){

    //Se divano ha 5 posti occupati e sala di attesa ha 10 posti occupati il
    cliente esce
    //Se divano ha 5 posti occupati ma almeno un posto in sala di attesa il
    cliente si accomoda lì
    //Se il divano è libero il cliente si siede direttamente nel divano
    if((divano.size()>5) && (sala_attesa.size()>10)) {
        System.out.println("Il negozio è pieno e il cliente " + id + " se
ne va");
    }else if (divano.size()>5){
        System.out.println("Il cliente " + id + " non trova posto nel divano
e si mette in sala di attesa");
        sala_attesa.add(id);
    }else{
        System.out.println("Il cliente " + id + " trova posto nel divano e
si siede");
        divano.add(id);
        notifyAll();
    }
}

public void effettuaPagamento(int barbiere){
    //Il cliente è stato precedentemente servito e può ora pagare
    System.out.println("Il barbiere " + barbiere + " ha fatto pagare il
cliente " + coda_pagamento.removeFirst());
}
}
```

# Lezione 25

---

## Algoritmi di scheduling

L'obiettivo dello scheduling è determinare in un certo istante quale processo può andare in esecuzione nel processore. Quando un sistema può scegliere quale processo eseguire, deve avere una strategia per decidere quale processo deve essere eseguito in un dato istante. Questa strategia è chiamata **politica di schedulazione** del processore. Lo scheduler tipicamente fa in modo che questa esecuzione sia quanto più fluida possibile, ed in particolare gli **algoritmi di scheduling** servono a:

- Massimizzare il throughput (numero di processi completati per una certa unità di tempo). Massimizzare il throughput significa fare in modo che nell'unità di tempo vengano eseguiti più processi possibili, allora l'idea per farlo è favorire i processi più veloci.
- Minimizzare la latenza (tempo di attesa per un processo prima di essere eseguito).
- Prevenire la posticipazione indefinita. Ciò si fa con il meccanismo di **aging**, ovvero prevedere un meccanismo in cui, periodicamente, aumenta la priorità del processo più lento, che altrimenti non verrebbe mai eseguito.
- Garantire che il processo eseguito termini in un intervallo di tempo ben definito, o comunque entro una certa scadenza.
- Massimizzare l'utilizzo del processore.

Vi sono alcuni algoritmi che riescono a soddisfare la maggior parte dei criteri sopra elencati, anche se in determinati contesti può essere più urgente risolvere una sola delle problematiche di sopra piuttosto che altre.

## Livelli di scheduling

Lo scheduling può avvenire a più livelli:

- **Scheduling di alto livello:** questo tipo di scheduling è tipico dei primi OS. In questo scheduling si determina quale tra i vari job può essere ammesso alla fase successiva così da competere attivamente per le risorse del sistema; una volta ammessi i job hanno inizio e diventano processi di sistema. Questo tipo di scheduling determina inoltre i. **livello di multiprogrammazione** in quanto determina il numero di processi presenti nel sistema in un certo istante. In quanto le risorse di sistema sono limitate, è compito di questo livello di scheduling, proibire temporaneamente l'ingresso di nuovi job come processi di sistema per evitare la saturazione del sistema. Nei sistemi più moderni questo livello di scheduling non è presente.
- **Scheduling di livello intermedio:** dopo che la politica di schedulazione di alto livello ha ammesso un job (che potrebbe contenere uno o più processi) nel sistema, la politica di schedulazione di medio livello determina a quali processi dovrebbe essere concesso di competere per il processore. Questa politica risponde alle fluttuazioni a breve termine nel carico del sistema, sospendendo temporaneamente e poi riprendendo i processi per far funzionare in modo fluido il sistema. Questo livello rappresenta quindi una sorta di "cuscinetto" tra il livello alto e basso di scheduling.
- **Scheduling di basso livello:** a questo livello abbiamo la determinazione di quale processo, tra quelli attivi, andrà assegnato al processore. Sempre in questo livello si vanno a selezionare le priorità dei processi che poi verranno utilizzate per determinare quali processi eseguire prima. Questo scheduling viene effettuato dal **dispatcher**, che fisicamente gestisce lo scheduling dei processi, e di conseguenza sarà sempre caricato in memoria principale.

Nei sistemi reali attuali solitamente è presente solo lo scheduling di livello intermedio e basso.

Vediamo graficamente quando descritto sopra:

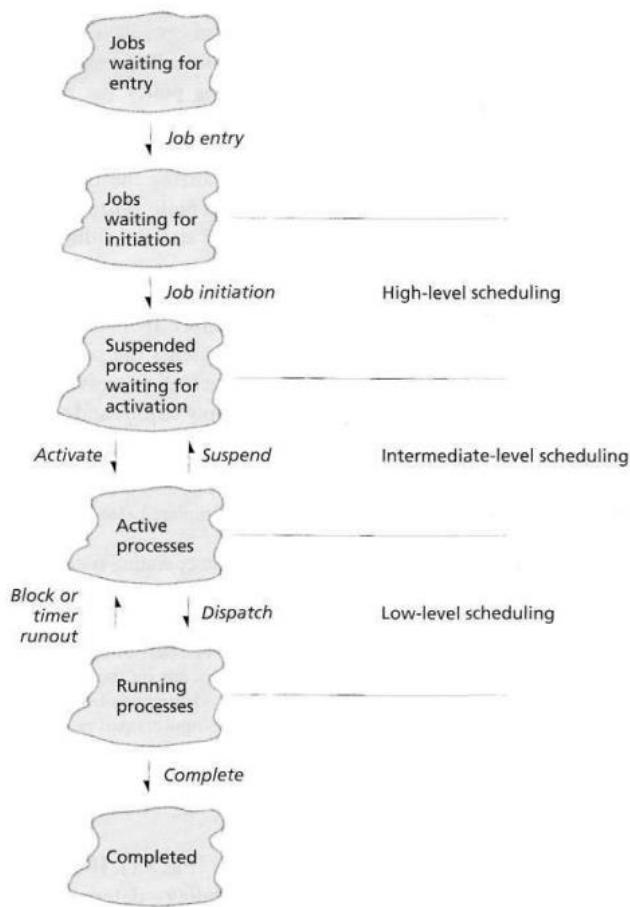


Figure 8.1 | Scheduling levels.

### Schedulazione Preemptive e Nonpreemptive

Un'altra caratteristica degli algoritmi di scheduling è data dalla possibilità che i processi possano eventualmente essere rimossi dal processore durante la loro esecuzione. Vediamo una distinzione tra i due tipi di processi:

- **Preemptive:** i processi di questa tipologia possono essere rimossi dal processore in cui sono attualmente assegnati. Potrebbero essere rimossi perché vi è necessità di assegnare quel processore ad un altro processo. Questo tipo di processi sono molto presenti nei sistemi interattivi; in ogni caso i processi di questo tipo rimangono caricati in memoria. Lo svantaggio nell'uso di questi processi è che avremo necessità di andare a memorizzare in memoria una immagine del processo utile per effettuare il context switching così da non perdere informazioni e dati elaborati fino a quel punto.
- **Nonpreemptive:** hanno la caratteristica di non potere essere rimossi dal processore fino al loro completamento. Lo svantaggio di questo approccio si ha, per esempio, nel caso in cui abbiamo un processo molto importante da eseguire, ma anche molto lento, e ciò porterebbe a bloccare gli altri processi in attesa. Una possibilità per sopperire a questa situazione è che il processo stesso rinunci volontariamente al controllo del processore in determinate situazioni. Il vantaggio di questo approccio è una maggiore semplicità di implementazione (in quanto non vi è complex switching), inoltre è possibile calcolare i tempi di esecuzione dei vari processi, o quantomeno stimarli.

### Priorità

Gli scheduler utilizzano spesso le **priorità** per determinare come schedulare ed effettuare il dispatching dei processi. Le priorità possono essere assegnate staticamente o cambiare dinamicamente e servono a quantificare l'importanza dei processi. Analizziamo nel dettaglio le priorità:

- **Priorità statiche:** valori fissi che vengono assegnati all'inizio a ciascun processo e non variano nel tempo. È un metodo facile da implementare e ha anche un basso overhead, tuttavia, essendo statico, per definizione, non si adatta molto bene a sistemi reali dinamici.
- **Priorità dinamiche:** consentono di rispondere in modo immediato a eventuali cambi di importanza all'interno del sistema. Si può quindi cambiare, all'evenienza, la priorità di un processo. Lo svantaggio è che le priorità dinamiche sono più complesse da implementare e hanno un maggiore overhead rispetto a quelli statici; il maggiore overhead viene però giustificato da una maggiore reattività del sistema.
- **Acquisto delle priorità:** l'utente paga per acquistare priorità e fare eseguire prima i suoi processi. Questa è una metodologia utilizzata spesso nei servizi cloud, l'utente che paga di più, ottiene delle prestazioni migliori ed esecuzioni più rapide di certi servizi.

## Obiettivi della schedulazione

Ci sono alcune caratteristiche che devono essere presenti in tutti gli algoritmi di scheduling:

- **Equità:** tutti i processi devono essere trattati in maniera equa, non ci devono essere preferenze tra due o più processi. Inoltre nessun processo deve essere posticipato in maniera indefinita.
- **Predicibilità:** i processi dovrebbero essere eseguiti sempre nello stesso tempo medio di esecuzione, così da rendere predicable il tempo di esecuzione dei processi stessi.
- **Scalabilità:** l'algoritmo di scheduling deve funzionare ed essere stabile sia per pochi processi sia per molti. Nel caso in cui ci sia troppo carico le prestazioni si devono degradare lentamente e non crollare di botto.

## Tipologie di processi

Gli algoritmi di scheduling devono tenere conto di due classificazioni dei processi. La prima è quella tra processi **Processor-Bound** e processi **I/O Bound**:

- **Processi Processor-bound:** processi che fanno un uso massiccio del processore. Sono processi che fanno molti calcoli e che tendono ad usare tutto il tempo del processore che il sistema gli alloca.
- **Processi IO Bound:** processi che usano molti dispositivi di IO ma poco il processore. Sono processi che necessitano di molte risorse a disposizione ma in compenso effettuano pochi calcoli mediante il processore.

La seconda classificazione è quella tra **processi Batch** e **processi interattivi**:

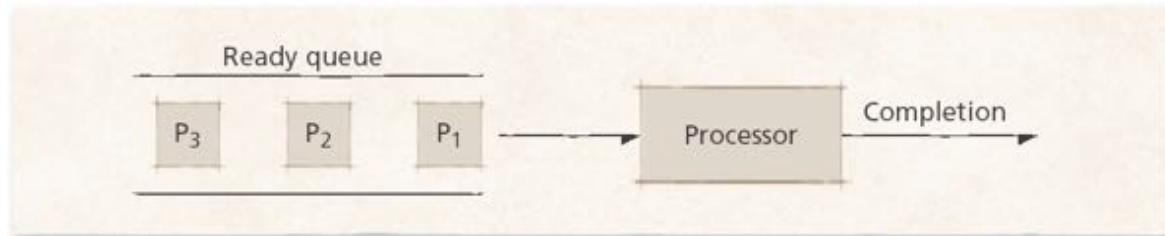
- **Processi Batch:** un processo batch contiene istruzioni che il sistema deve eseguire senza interagire con l'utente.
- **Processi interattivi:** un processo interattivo richiede di frequente input dall'utente. Il sistema dovrebbe assicurare dei buoni tempi di risposta a un processo interattivo, mentre in generale un processo batch può essere ragionevolmente ritardato.

L'obiettivo dello **scheduler** è di gestire tutte queste tipologie di processi, e per farlo deve decidere quando e per quanto tempo ogni processo sarà eseguito. Elenchiamo ora i principali algoritmi di scheduling:

## Algoritmo FIFO (First in First out)

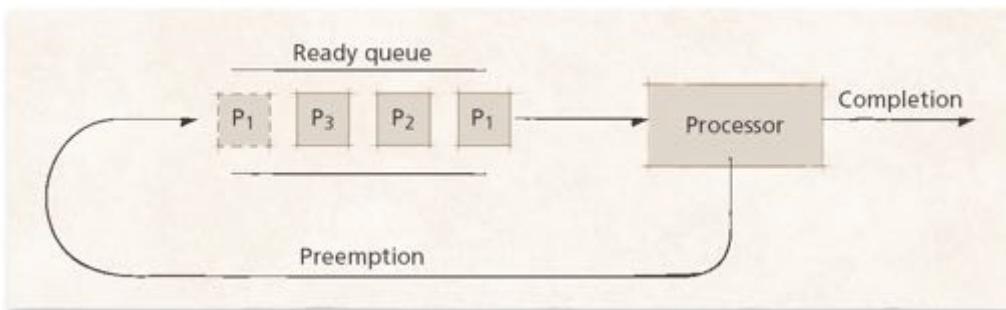
È lo schema più semplice, e si basa su una coda. Il primo processo che si mette in coda sarà il primo ad essere eseguito. Questo algoritmo è **nonpreemptible**, ovvero, una volta che il processo va in esecuzione, non potrà essere rimosso dal processore. Inoltre è **fair** poiché tutti i processi vengono eseguiti in ordine di arrivo, a prescindere dalla priorità, ma è anche **unfair** perché i processi con tempi di esecuzione più lunghi staranno più tempo all'interno del processore penalizzando quelli più corti (dal punto di vista del tempo di esecuzione) che aspetteranno la fine della esecuzione dei processi più pesanti.

Vediamolo meglio nella figura di seguito:



### Algoritmo Round-Robin (RR)

In questo algoritmo si utilizza il concetto di **quanto di tempo**. Il RR utilizza una **coda FIFO** per determinare quali processi eseguire, tuttavia a differenza dell'algoritmo FIFO, i processi non rimangono nel processore fino al completamento, ma hanno a disposizione una quantità di tempo limitata. Se il processo non viene completato in questo quanto di tempo viene rimosso dal processore e rimesso in coda alla fine. Affinché questo meccanismo funzioni è necessario che venga mantenuta una coda molto ampia di processi in attesa di essere eseguiti.



Di RR ne esistono più varianti, ad esempio:

- **Selfish RR (SRR)**: in questo algoritmo vengono gestite due diverse code, la prima è una **coda "nascosta**" dove i processi vengono inseriti fino a quando non arrivano ad una certa priorità (che aumenta col passare dei cicli). Arrivata alla soglia prefissata di priorità il processo passa alla **coda attiva** che è quella dei processi che effettivamente andranno ad essere eseguiti.

L'SRR utilizza quindi la tecnica di **aging** per aumentare gradualmente le priorità dei processi attraverso il tempo.

I processi nella coda attiva aumentano di priorità molto più lentamente rispetto ai processi nella coda "nascosta" (anche chiamata **coda holding**), ciò significa che i processi nella coda holding vi sosteranno di meno.

Ora, la priorità di un processo viene incrementata con un tasso a mentre è nella holding queue, e ad un tasso b, con  $b \leq a$ , nella active queue. Quando  $b < a$ , i processi nella holding queue invecchiano ad un tasso maggiore di quelli nella active queue, così che questi entreranno prima o poi nella active queue, contendendo per il processore, mentre se  $b = a$  non c'è differenza tra le due code e il SRR degenera in uno schema FIFO. Viceversa quando  $a >> b$  i processi che entrano nel sistema aspetteranno poco tempo, o addirittura nessuno, nella holding queue, e l'SRR degenererà così in round-robin.

Il problema generale dello schema RR è legato alla scelta del **quanto di tempo**. Il problema è determinare il quanto di tempo, in quanto sarà quest'ultimo a determinare le performance dell'algoritmo RR. Se sceglieremo un quanto elevato il RR potrebbe degenerare in un algoritmo FIFO in quanto qualsiasi processo potrebbe essere completato in un quanto. Di contro, se il quanto è molto piccolo, i processi potrebbero non essere mai completati in un unico quanto e ciò porterebbe a molti context switching che aumenterà di molto l'overhead del sistema.

Quindi, come possiamo scegliere il quanto di tempo  $q$ ?

In linea teorica dovrebbe essere scelto in maniera tale da completare in una unica esecuzione i processi **IO-bound**, ma allo stesso tempo al non completare in una unica esecuzione i processi **Batch** e **processor-bound**.

### Algoritmo Shortest-Process-First (SPF)

È un algoritmo **nonpreemptible** in cui si mandano in esecuzione i processi per i quali il tempo di esecuzione stimato è minore possibile. Piuttosto che rimuovere dal processore i processi che stanno per essere completati, vengono lasciati al processore così da essere completati. Ciò riduce il numero di processi totali in attesa di essere assegnati al processore (waiting process).

È quindi particolarmente importante effettuare delle stime affidabili per il tempo di esecuzione dei processi. SPF riduce i tempi di attesa media rispetto a FIFO. I tempi di attesa, tuttavia, hanno una maggiore varianza (cioè sono meno predibili) rispetto a FIFO, di conseguenza in sistemi molto statici può essere un buon approccio utilizzare questo algoritmo. Per massimizzare il throughput è sicuramente una delle migliori soluzioni in quanto termina molti processi in certi archi di tempo rispetto ad altri algoritmi.

### Algoritmo Highest-Response-Ratio-Next (HRRN)

L'algoritmo HRRN tiene conto sia del tempo di attesa sia di quello di esecuzione, sui cui si basa la priorità dei processi, infatti la priorità viene calcolata secondo la seguente formula:

$$\text{priority} = \frac{\text{timewaiting} + \text{servicetime}}{\text{servicetime}}$$

A numeratore si avvantaggiano i processi che hanno aspettato tanto e che hanno tempo di esecuzione non breve, viceversa, al denominatore si vanno ad avvantaggiare i processi più corti da eseguire.

Questo algoritmo non soffre di posticipazione indefinita (in quanto utilizza le priorità che è definita anche in funzione del tempo di attesa) ed è anche **nonpreemptive**.

### Algoritmo Shortest-Remaining-Time (SRT)

La schedulazione shortest-remaining-time (SRT) è la controparte con **preemptible** di SPF che tenta di aumentare il throughput dando servizio ai piccoli processi in arrivo. In SRT, lo scheduler seleziona il processo con il minor tempo di esecuzione rimanente stimato. In SPF, una volta che un processo inizia la sua esecuzione, continua fino a completare. In SRT, un processo appena arrivato con un basso tempo di esecuzione stimato prende il posto di un processo in esecuzione con maggior tempo di esecuzione stimato.

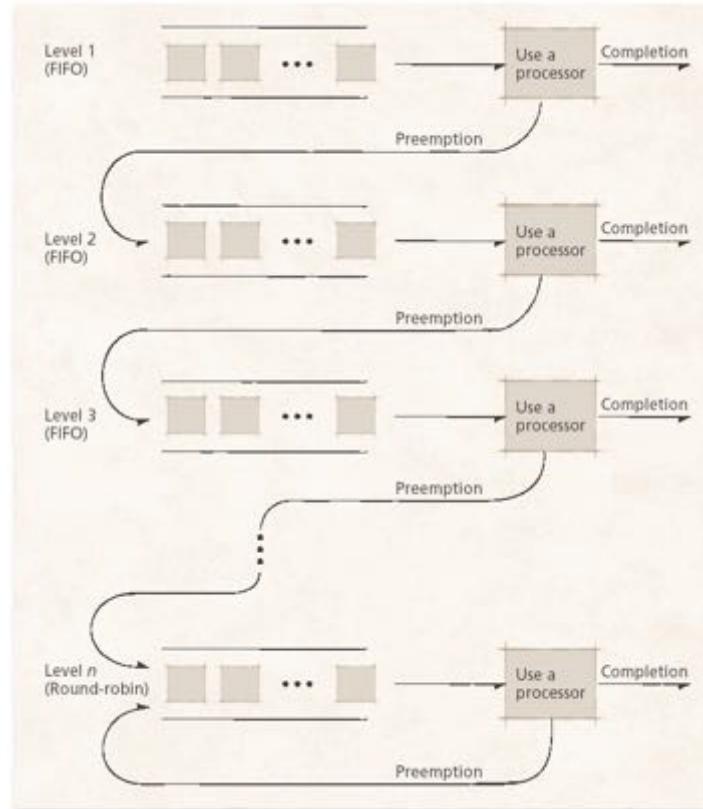
I nuovi processi in arrivo con basso tempo di esecuzione verranno eseguiti quasi immediatamente a discapito dei processi con medio e lungo tempo di esecuzione.

### Code multilivello a retroazione

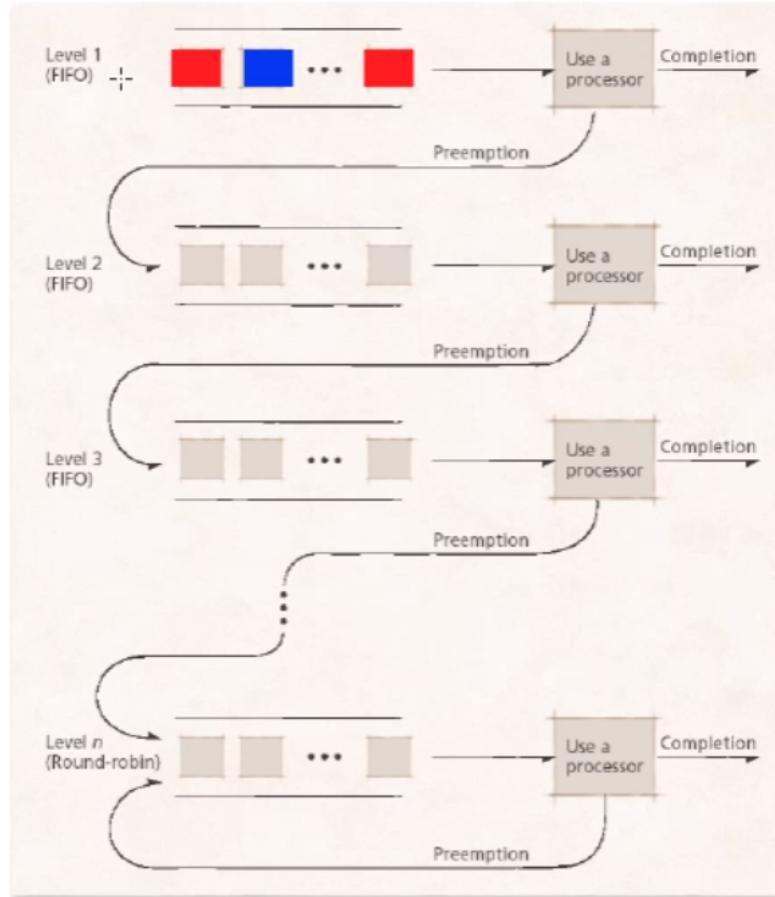
Non si ha una semplice coda di attesa bensì si prevede un meccanismo tale che un processo transiti da una coda all'altra, così da gestire sia processi IO bound sia processi Processor Bound, con l'idea che bisognerebbe fare in modo che i processi IO Bound siano eseguiti prima dei processor bound così da non aspettare tempi elevati.

Il modo più semplice di gestire i processi è di farli spostare in queste code via via che vengono eseguiti: quando un processo deve contendere il processore entra nella coda di più alto livello (la quale ha priorità maggiore), via via che il processo viene eseguito e non riesce a completare viene spostato in una coda sottostante (di livello minore). Ciò significa che i processi brevi rimarranno nelle code di alto livello, viceversa i processi più lenti si sposteranno via via nelle code di livello più basso.

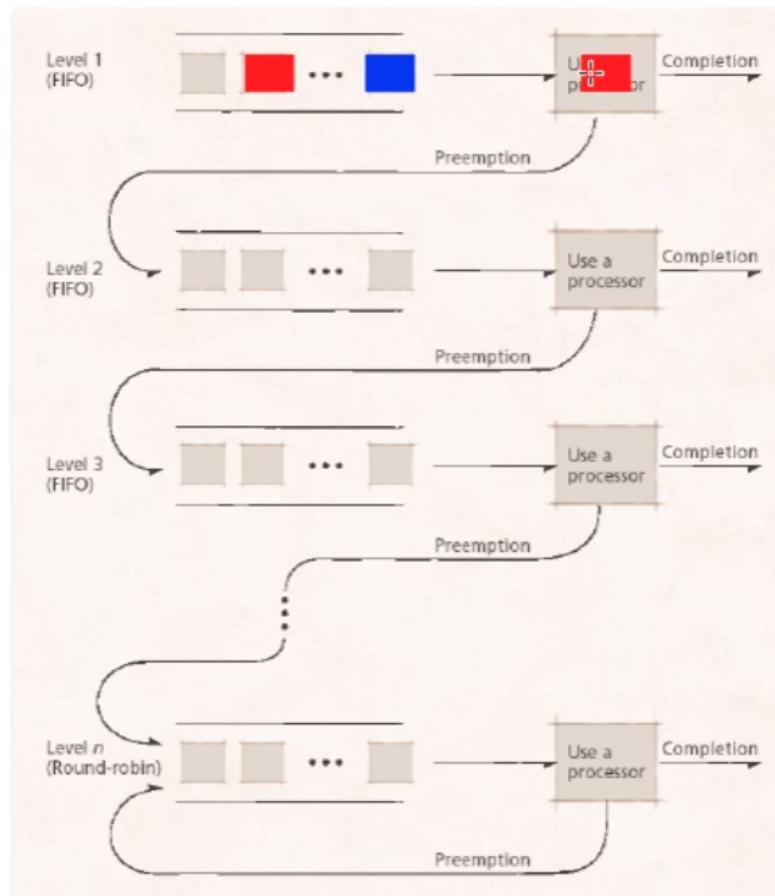
Tutti i processi vengono gestiti nelle relative code seguendo un meccanismo FIFO, la coda di ultimo livello (quella con meno priorità) invece usa una sorta di procedimento RR. Vediamolo graficamente:

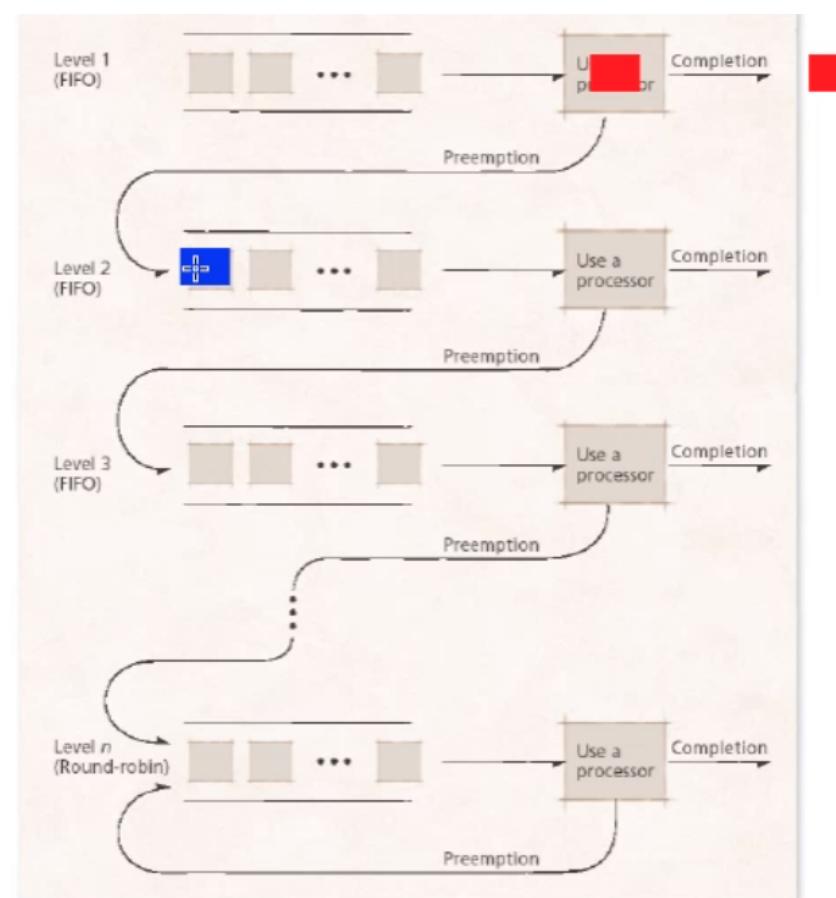
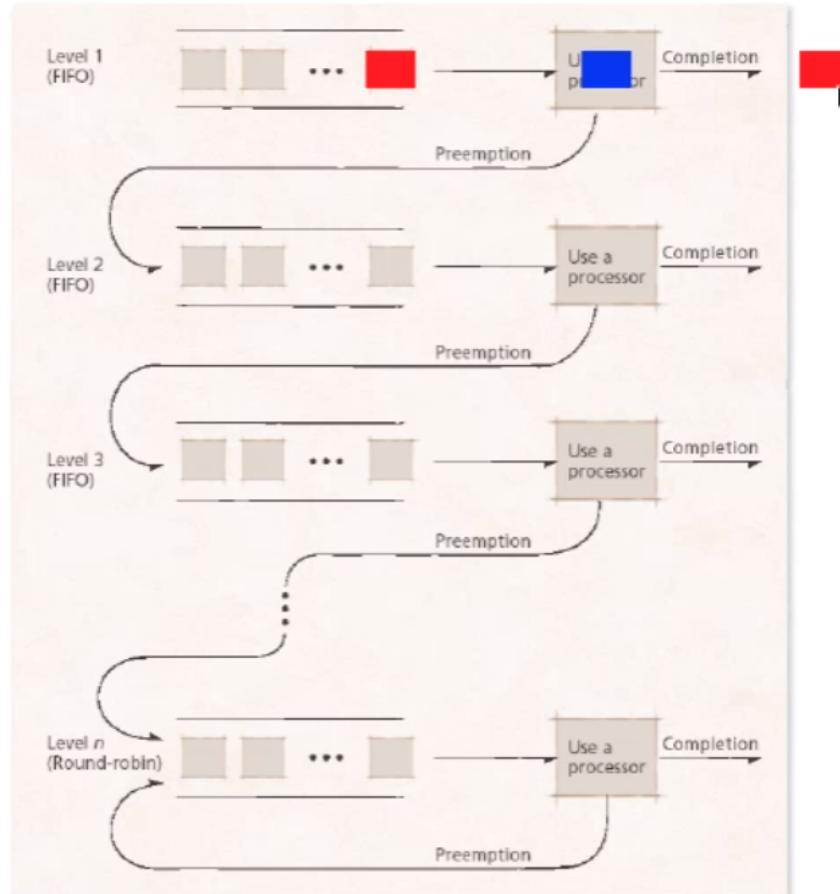


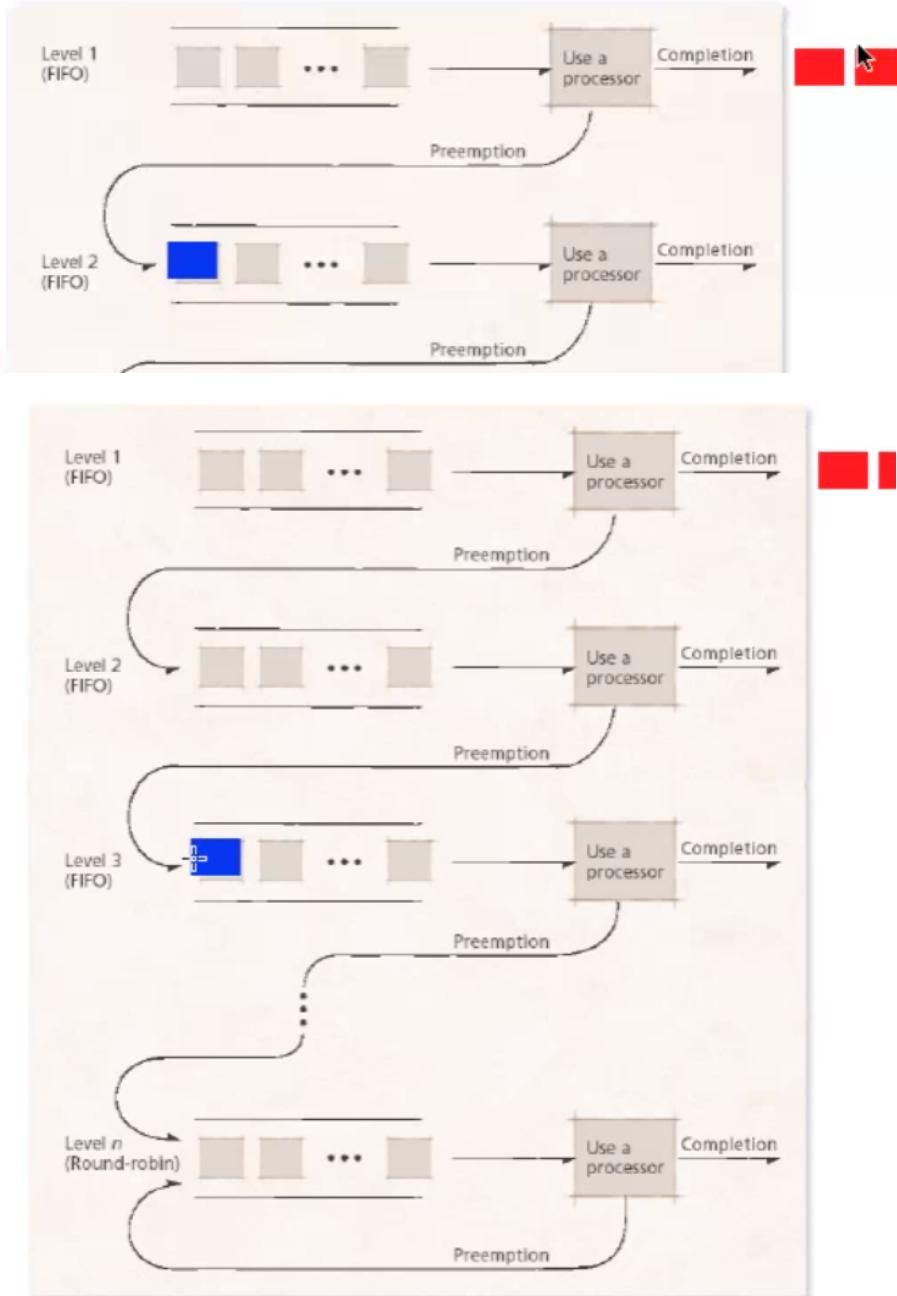
Supponiamo ora di avere processi più lunghi da eseguire (blu) e più corti da eseguire (rossi):



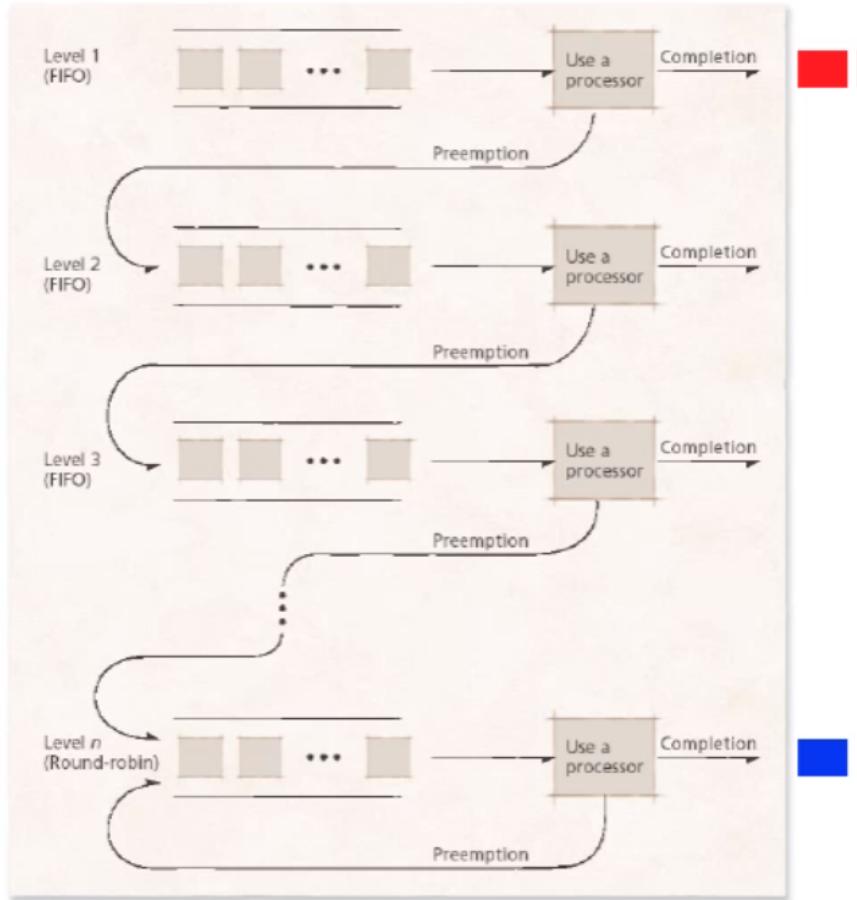
Al livello  $n$ -esimo un processo va in round robin, quindi se non dovesse essere completato si inserirà nuovamente nella coda di livello  $n$ . Inizialmente i processi sono tutti anelli la coda di primo livello e quindi lo scheduler li selezionerà in maniera FIFO:



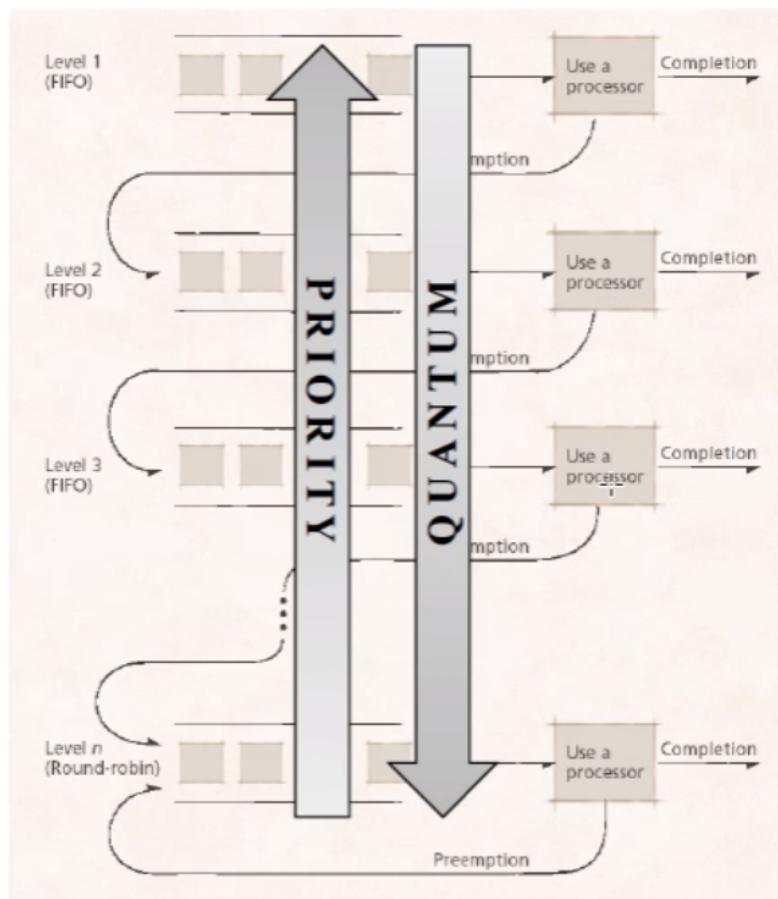




E così via fino a che il processo blu non arriverà all'ultimo livello dove verrà eseguito fino a quando non sarà completato:



In generale, più si è in alto di livello maggiore sarà la priorità di esecuzione, viceversa, più in basso saremo di livello, maggiore sarà il quanto di tempo assegnato ai processi:



Il vantaggio di questo tipo di algoritmo è che riesce a favorire una singola esecuzione (e completamento) dei processi più leggeri eliminando così il problema della posticipazione indefinita. Anche in questo algoritmo vengono eseguite delle stime dei tempi medi di esecuzione dei processi in maniera tale da mandare, se è necessario, i processori più lenti all'n-esimo livello evitando di farglieli attraversare tutti inutilmente (dato che non verrebbe comunque completata l'esecuzione).

# Lezione 26

## Organizzazione della memoria

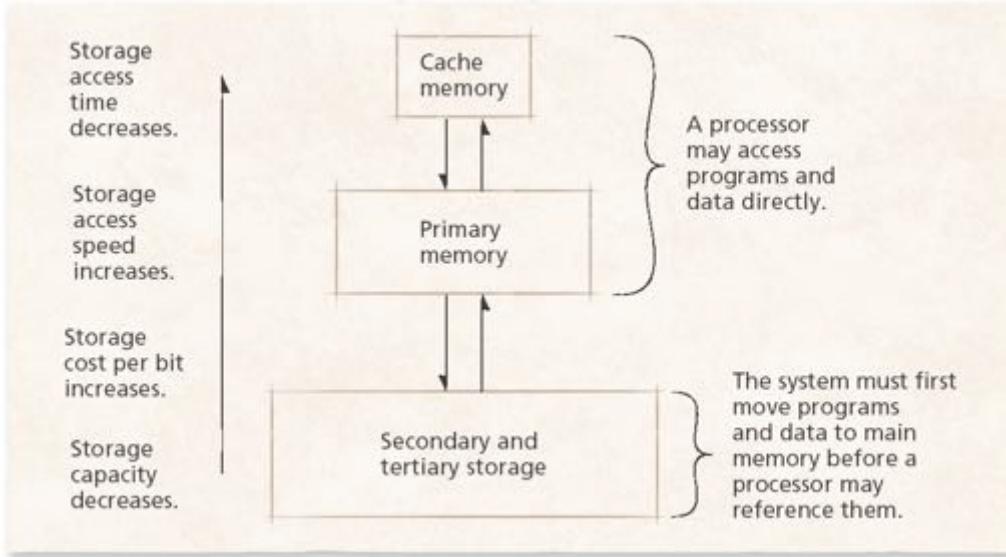
Ciò che vogliamo fare è capire come assegnare memoria ad un processo e come gestirla in generale. Nei sistemi moderni a ciascun processo viene assegnata una partizione di memoria che potrà essere allocata o in modo dinamico o in modo statico. Via via che i programmi diventano più complessi si assiste ad una tendenza ad esaurire molto rapidamente la quantità di memoria a disposizione. I sistemi operativi hanno dei requisiti minimi di memoria che sono andati ad aumentare via via col passare del tempo e con la complessità dei sistemi operativi stessi:

<i>Operating System</i>	<i>Release Date</i>	<i>Minimum Memory Requirement</i>	<i>Recommended Memory</i>
Windows 1.0	November 1985	256KB	
Windows 2.03	November 1987	320KB	
Windows 3.0	March 1990	896KB	1MB
Windows 3.1	April 1992	2.6MB	4MB
Windows 95	August 1995	8MB	16MB
Windows NT 4.0	August 1996	32MB	96MB
Windows 98	June 1998	24MB	64MB
Windows ME	September 2000	32MB	128MB
Windows 2000 Professional	February 2000	64MB	128MB
Windows XP Home	October 2001	64MB	128MB
Windows XP Professional	October 2001	128MB	256MB

Una problematica importante da affrontare è quella di capire se in main memory vada inserita solo un processo o più processi contemporaneamente. Ipotizzando che la main memory sia in grado di ospitare più processi contemporaneamente, un'altra problematica da risolvere è capire quanto spazio in memoria va riservato ad ogni processo, in particolare se le partizioni dei singoli processi devono avere la stessa dimensione o dimensioni diverse. Tutte queste considerazioni sono alla base di tutto ciò che abbiamo visto sino ad adesso.

Le operazioni di gestione della memoria vengono svolte da una componente specifica chiamata **memory manager** che si occuperà di scegliere quali processi caricare in memoria, in che posizione caricarli e quanta memoria dargli. Nel caso in cui la memoria si riempie, inoltre, sarà compito di questa componente quello di capire quali processi rimuovere dalla memoria per liberare spazio.

Ricordando come è strutturata la gerarchia della memoria:



## Strategie di gestione della memoria

Abbiamo più tipologie di strategie:

- **Strategie di fetching:** sono strategie che determinano quando spostare il prossimo pezzo di un programma o dato dalla memoria secondaria a quella principale. Queste strategie possono essere eseguite secondo due modalità, o **fetch-on-demand** o **anticipatory fetch**. La prima modalità consiste nel caricare i dati in memoria via via che il sistema lo richiede, la seconda soluzione invece si basa sul allocare della memoria sapendo preventivamente quali saranno le richieste del sistema. Ovviamente quest'ultima modalità ha un overhead maggiore.
- **Strategie di placement:** sono strategie atte a capire dove caricare in memoria (in quale posizione) i dati e i programmi. Esistono 3 diverse strategie: **First fit**, **Best Fit**, **Worst Fit**. La strategia **first fit** mette il processo nella prima partizione libera in memoria centrale grande abbastanza per contenerlo. Questa strategia può però lasciare dei frammenti, ovvero parti di memoria non utilizzata, piuttosto numerosi. La strategia **best fit** prevede che il nuovo processo venga inserito nella partizione con dimensione più vicina alla dimensione del processo. In questo modo, si tende a lasciare frammenti sempre più piccoli. L'ultima strategia, la **worst fit**, è invece meglio funzionante. Essa prevede che il nuovo processo venga allocato nella partizione di memoria più grande disponibile. Questo è giustificato dal fatto che nei frammenti lasciati, che saranno di grandi dimensioni, potranno essere allocati altri processi.
- **Strategie di sostituzione:** quando la memoria è troppo piena per far spazio ad un nuovo programma, il sistema deve rimuovere qualche programma o dato (o tutti) che risiede attualmente in memoria. Le strategie di sostituzione determinano cosa rimuovere.

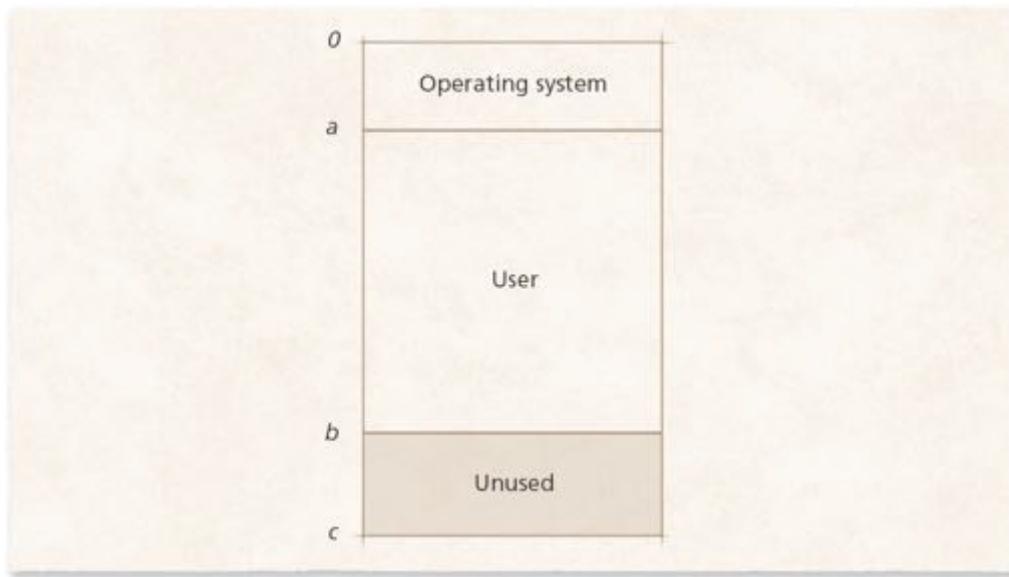
## Tipo di allocazione

La allocazione può avvenire secondo due modalità:

- **Allocazione contigua:** i programmi vengono allocati per intero in indirizzi di memoria adiacenti. Questo implica che ci debba essere abbastanza spazio contiguo per contenere il programma nella sua interezza.
- **Allocazione non contigua:** i programmi vengono spezzettati in **segmenti** di dimensione fissa o variabile che vengono poi caricati in aree di memoria non adiacenti. Tuttavia questa operazione non è semplicissima in quanto bisogna rappresentare la memoria e le aree libere che vi sono all'interno.

## Allocazione di memoria in sistemi singolo utente

I primi computer potevano essere utilizzati da una sola persona alla volta. Tutte le risorse della macchina erano dedicate a quell'utente il quale aveva il controllo dell'intero sistema e per questo motivo tutto lo spazio poteva essere utilizzato da questo utente:

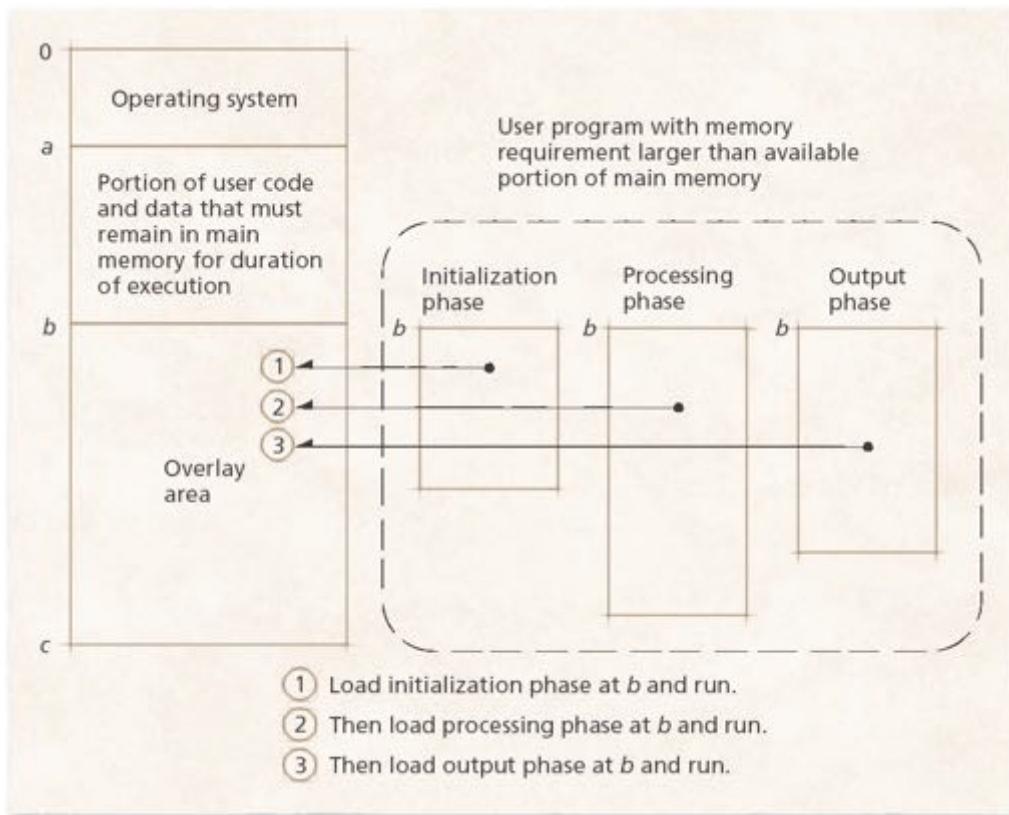


### Overlays

Per superare i limiti dati dalla allocazione contigua della memoria, si cominciarono ad utilizzare i cosiddetti **overlay**, i quali permettevano al sistema di eseguire programmi più grandi della memoria principale (ciò non era possibile con la allocazione contigua della memoria). Gli overlay basano il loro funzionamento su una tecnica di programmazione che prevede di suddividere il programma in sezioni in base ai vari task che vengono svolti, i task a loro volta sono divisi in sezioni logiche. Questi programmi sono divisi in sezioni logiche e quindi possiamo caricare in memoria le singole sezioni che ci servono piuttosto che l'intero programma.

Queste sezioni, inizialmente, venivano individuate manualmente dal programmatore ma questo rendeva il programma non modulare e pesante da realizzare anche per il programmatore. Ora invece questa operazione di individuazione delle sezioni del programma vengono fatte dal memory manager.

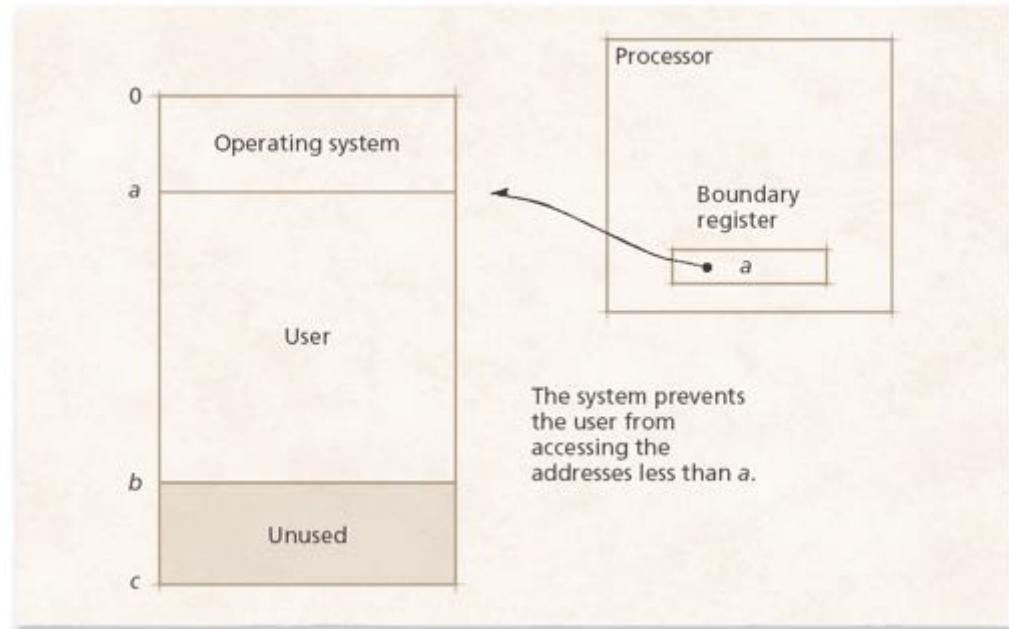
Immaginandolo graficamente:



Nel programma individuiamo 3 diverse sezioni che vengono caricate a turno nell'area di overlay.

### Boundary register

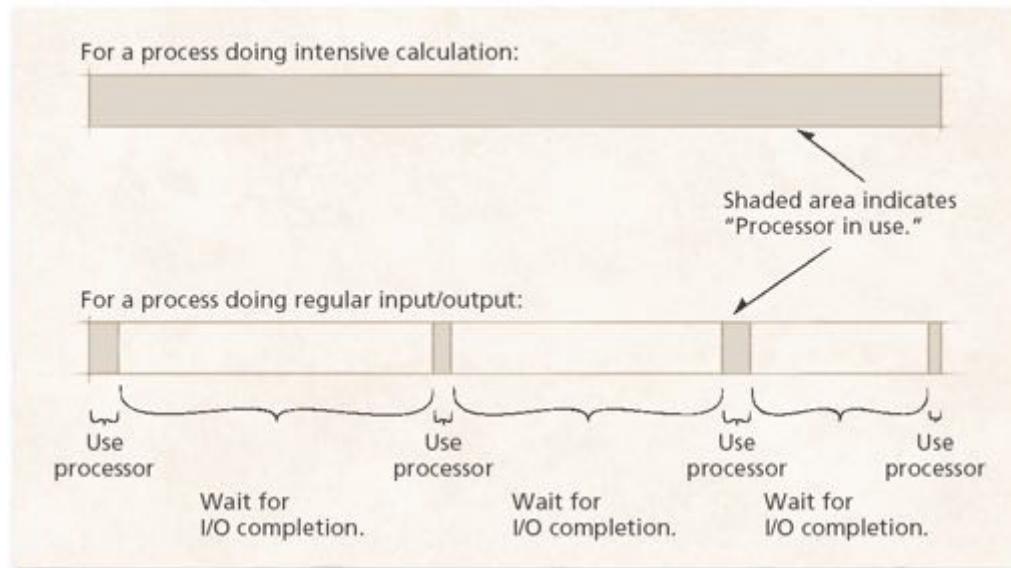
Sono dei registri che contengono indirizzi di memoria che possono essere utilizzati per andare a regolare l'accesso a determinate aree di memoria protette. Nello specifico si vuole evitare che i processi utente non vengano allocati nella zona di memoria riservata al sistema operativo:



### Sistemi multiprogrammati a partizione fissa

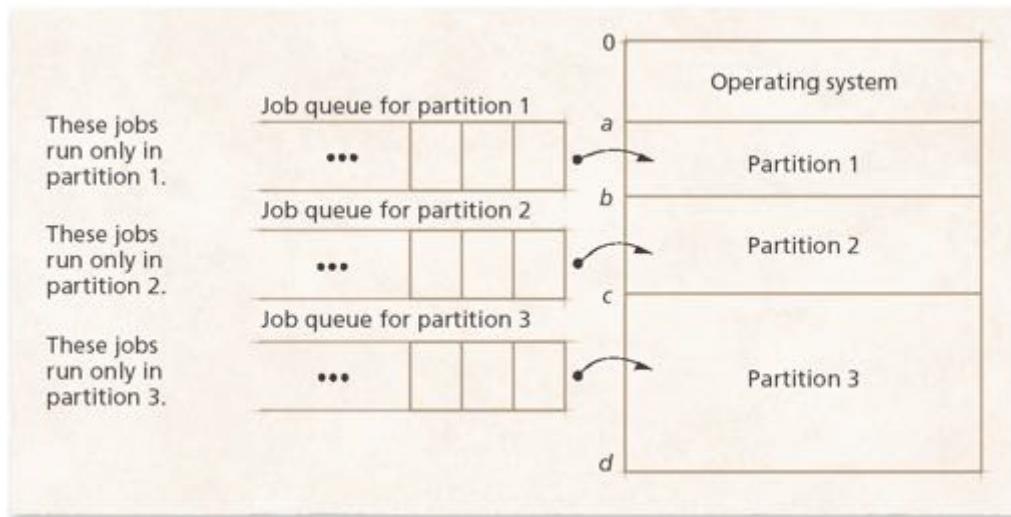
Analizziamo ora sistemi multiprogrammati a **partizione fissa**, ovvero sistemi con più utenti che possono accedere alle risorse del sistema. L'idea alla base di questi sistemi è di alternare continuamente processi processor bound e processi IO bound, nello specifico, il processo in attesa per l'I/O cedeva il processore se un altro era pronto ad effettuare calcoli. Quindi le operazioni di I/O e i calcoli del processore

potevano avvenire simultaneamente. Questo ha aumentato enormemente l'utilizzo del processore e il throughput dei sistemi.



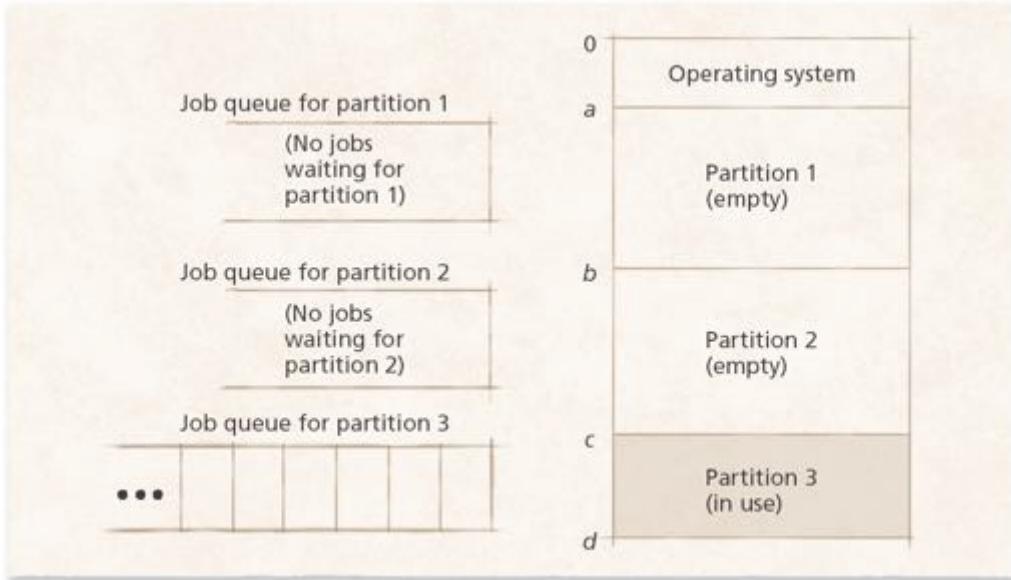
Ogni processo che deve essere eseguito riceve come destinazione un blocco di dimensione fissa. Il processore alterna i vari processi che devono essere eseguiti e in questo caso avremo più boundary register per evitare che i processi non si interferiscano a vicenda.

Per ciascuna delle partizioni presenti nel sistema esiste una coda di processi che vogliono utilizzare quella partizione:

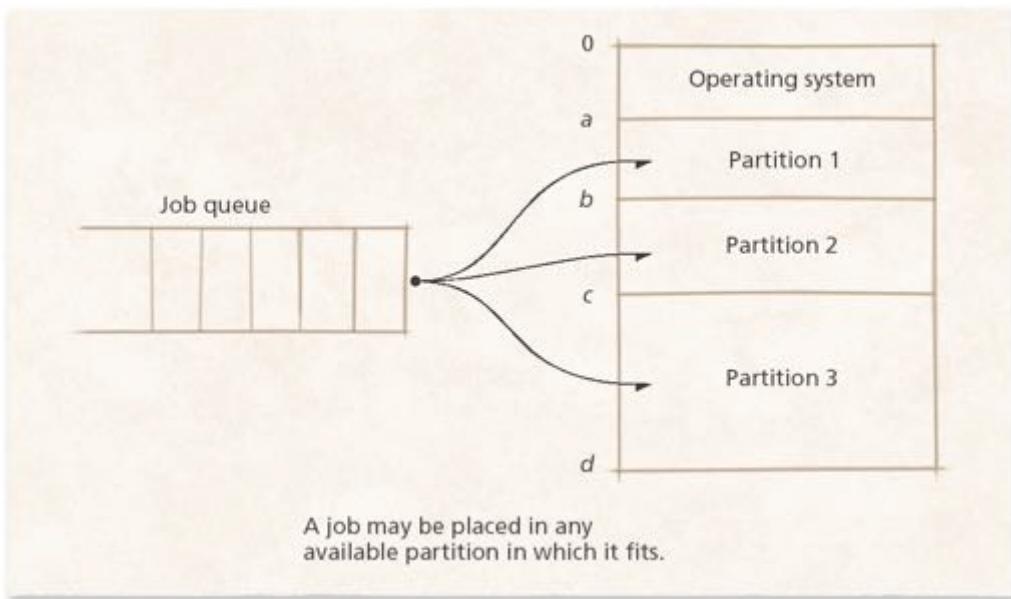


Secondo questo schema, il sistema divide la memoria principale in un certo numero di partizioni di dimensione fissa. Ogni partizione contiene un singolo job, e il sistema scambia il processore rapidamente tra i job per creare l'illusione della simultaneità. Questa tecnica permette al sistema di fornire semplici capaci di multiprogrammazione.

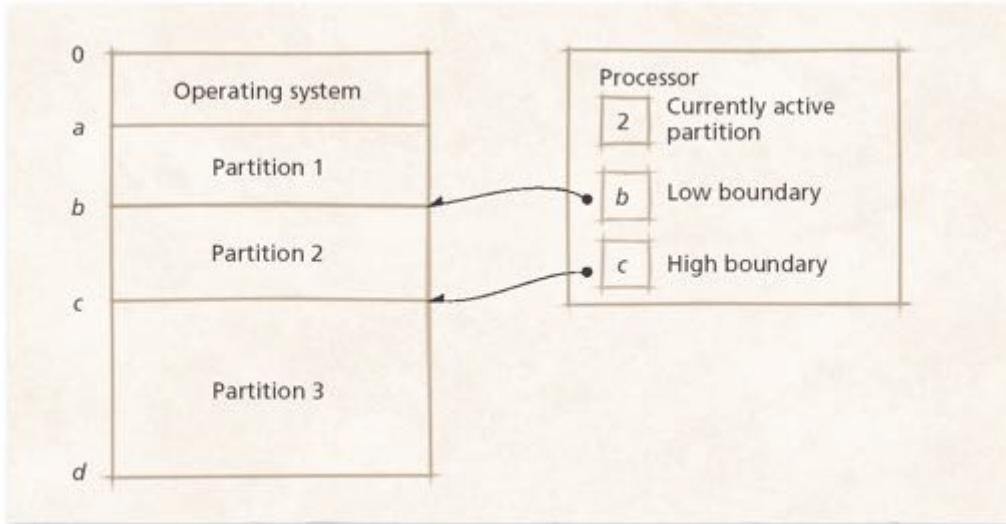
Il limite di questo approccio è che, se le 3 code crescono con 3 velocità differenti, ci potrebbero essere zone di memoria (*leggasi: partizioni*) sempre occupate e zone di memoria vuote:



Per superare il problema dello spreco di memoria, gli sviluppatori hanno creato dei compilatori, degli assemblatori e dei loader riallocanti. Questi strumenti producono un programma riallocabile che può essere eseguito in una qualunque partizione disponibile, grande abbastanza da contenerlo. Lo vediamo nel seguente scenario:



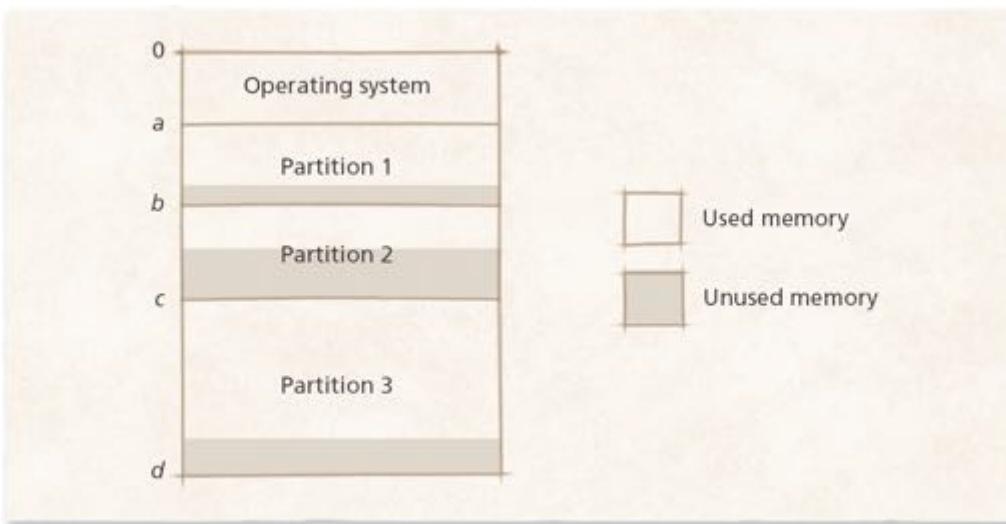
Man mano che l'organizzazione della memoria è diventata più complessa, i progettisti hanno dovuto migliorare i loro schemi di protezione. In un sistema a singolo utente, il sistema operativo deve essere protetto solamente dal processo dell'utente. In un sistema multiprogrammato, il sistema deve proteggere il sistema operativo da tutti i processi degli utenti e proteggere ogni processo da tutti gli altri. Nei sistemi multiprogrammati ad allocazione contigua la protezione viene spesso implementata con dei boundary register. Il sistema può limitare ogni partizione con due boundary register, low e high, chiamati anche base register e limit register:



Le partizioni hanno dimensioni diverse, ma non sono dinamiche in quanto la loro dimensione non varia ma rimane fissa sin da quando vengono create.

### Frammentazione

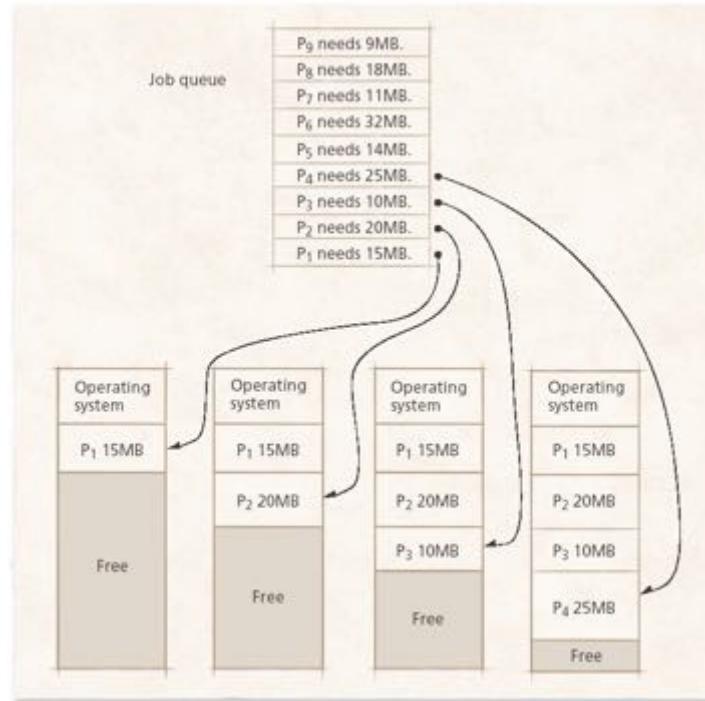
Per superare il problema dello spreco di memoria dovuto alle partizioni fisse si utilizza la **frammentazione**, esistono infatti aree di memoria che pur essendo disponibili non possono essere utilizzate in quanto troppo piccole per poter contenere determinati processi. Nello specifico si parla di **frammentazione interna**: nel caso di partizioni fisse, abbiamo una certa partizione all'interno della quale c'è una parte di memoria utilizzata e una parte non utilizzata che non potrà essere utilizzata da altri processi. Vediamolo graficamente:



In figura notiamo che le tre partizioni utente del sistema sono occupate, ma ogni programma è più piccolo della sua partizione corrispondente. Conseguentemente, il sistema potrebbe avere abbastanza spazio in memoria per eseguire un altro programma, ma non ha partizioni rimanenti in cui eseguirlo. Quindi, parte delle risorse di memoria del sistema vengono sprecate.

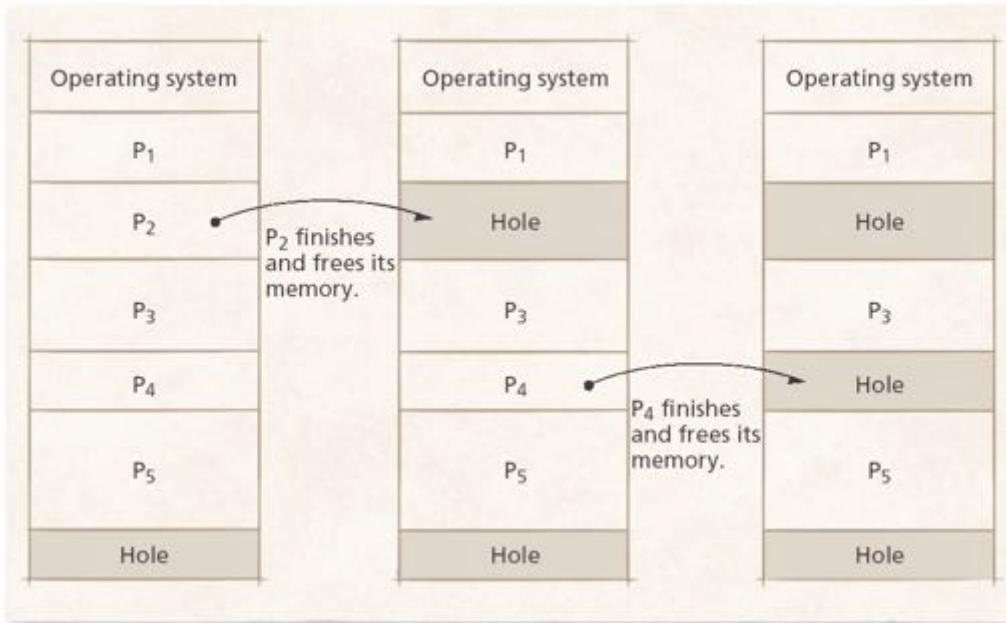
### Sistemi multiprogrammati a partizione variabili

Dato che le partizioni fisse hanno questi problemi immaginiamo delle **partizioni variabili**, dove non c'è alcun limite nelle aree di memoria e il sistema operativo andrà a caricare i processi nelle zone di memoria libere occupando la memoria di cui si ha strettamente bisogno senza occuparne in eccesso:



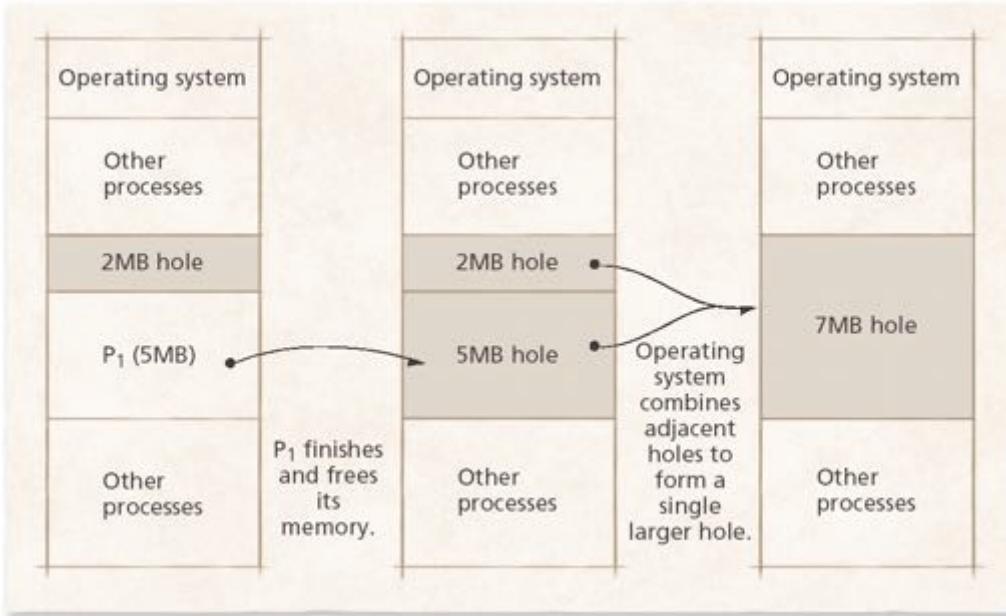
In questo caso non si presenta la partizione interna poiché la partizione di un processo è esattamente della stessa dimensione di un processo, tuttavia si presenta una situazione nella quale lo spazio libero rimasto alla fine della memoria principale non basti per contenere il prossimo processo in coda. Questa situazione prende il nome di **frammentazione esterna**, via via che un processo completa il proprio task, rimangono delle aree di memoria disponibili che però prese singolarmente non saranno sufficienti a contenere un nuovo processo. Magari la somma basterà a contenere il processo ma ricordiamo che stiamo considerando un sistema con allocazione esclusivamente contigua.

Vediamo con un esempio:

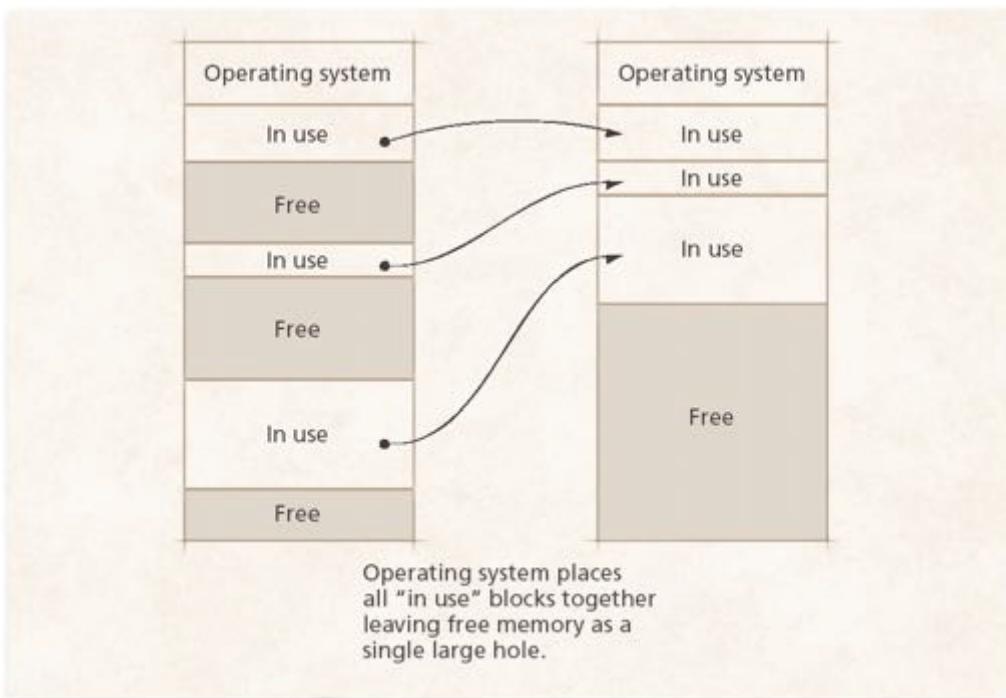


Gli spazi inutilizzati potranno contenere solo processi che occupano meno spazio rispetto alla loro dimensione.

Per compattare questi spazi utilizziamo una tecnica chiamata di **Coalescing (fusione)** che serve a combinare degli spazi vuoti adiacenti in memoria così da creare un blocco di memoria più ampio che può contenere un processo più grande. Vediamolo graficamente:



Un'altra tecnica che possiamo utilizzare è la **deframmentazione (o compattazione)** che serve a spostare e riorganizzare la memoria in maniera tale da compattare verso l'alto tutti i processi che usano la memoria lasciando così delle zone di memoria libere in fondo alla memoria che potranno essere utilizzate da nuovi processi:



Il problema di questa operazione è l'elevato costo di overhead.

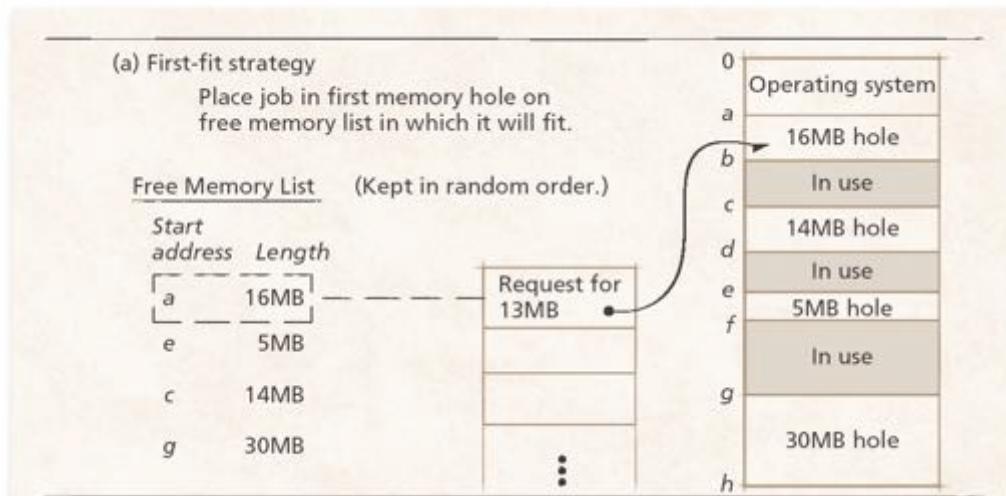
### Memory placement

In un sistema multiprogrammato a partizione variabile, il sistema deve spesso scegliere in quale spazio di memoria allocare un processo in arrivo. La strategia di allocazione di memoria del sistema determina dove collocare i programmi e i dati in arrivo all'interno della memoria principale. Esistono varie strategie per eseguire questo compito. Prima ne abbiamo citate tre che ora analizzeremo in dettaglio.

#### First-Fit

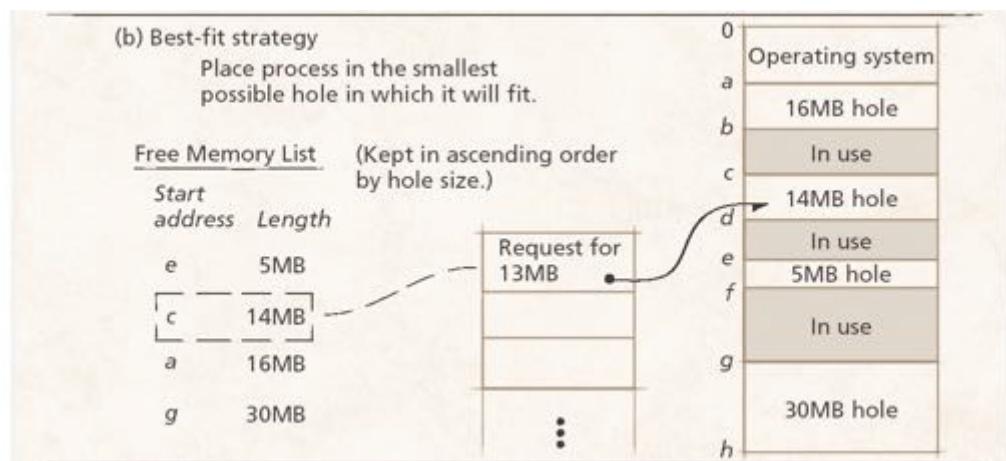
In questa strategia i processi vengono allocati nel primo spazio disponibile in memoria (di grandezza tale da contenere il processo). È una strategia con basso overhead e semplice da implementare.

Per implementarla si fa utilizzo di una **free memory list** che serve a tenere conto delle zone di memoria libere, il loro spazio e il loro indirizzo di memoria. Quando si dovrà caricare un programma in memoria il memory manager andrà a controllare la lista cercando il primo blocco di dimensione adeguata per allocare il processo:



### Best-Fit

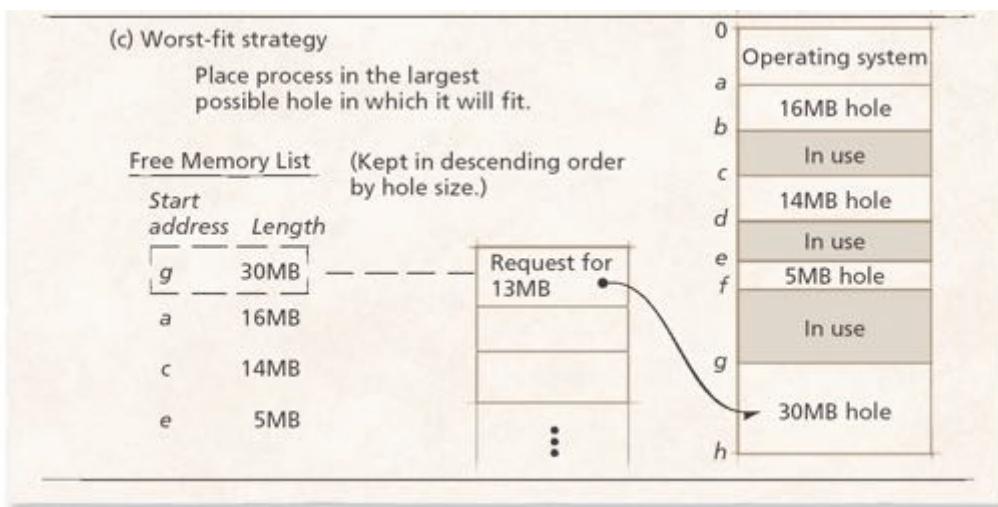
Anche in questo caso si usa una free memory list, ma rispetto al caso precedente la lista viene ordinata in modo crescente così da avere prima i blocchi di dimensioni minori e poi via via quelli di dimensione maggiore. Questo serve ad allocare i processi nelle prime zone disponibili che hanno lo spazio sufficiente a contenerli, quindi, nelle posizioni migliori (migliori per quanto riguarda lo spreco di spazio):



Il seguente algoritmo ha un overhead e un tempo di esecuzione maggiori rispetto alla strategia First-Fit. Inoltre, via via che gli spazi liberi di memoria si vanno a riempire si andrà a creare della frammentazione in quanto le dimensioni delle aree libere saranno sempre minori e inutilizzabili.

### Worst- Fit

La strategia opposta al Best-Fit prende il nome di Worst-Fit. Questa strategia fa uso sempre della free memory list, ordinata però in maniera decrescente, e consiste nel collocare un job in memoria in principale, nell'area in cui entra "peggior" (cioè nello spazio più largo). Il vantaggio è semplice: dopo che il processo viene collocato in questo largo spazio, lo spazio rimanente è spesso grande anch'esso e quindi è in grado di contenere un nuovo programma relativamente grande:



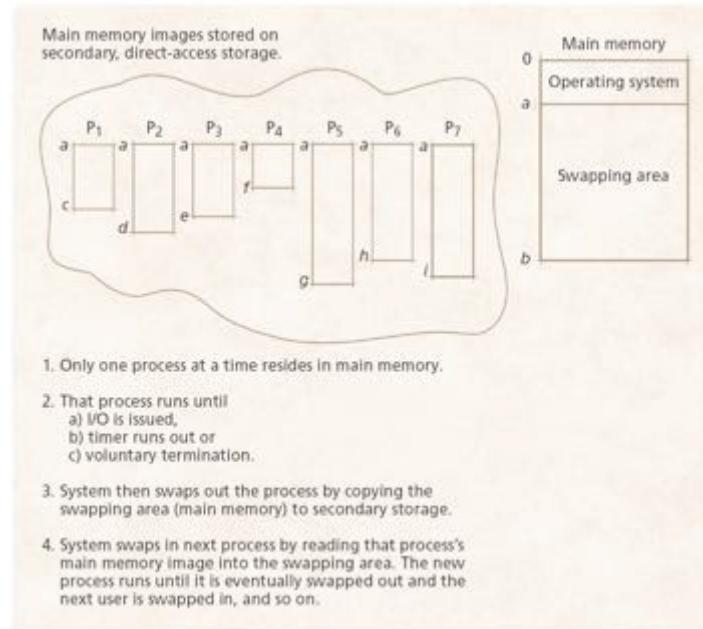
## Next-Fit

È una variante del First-Fit che però vede la memoria come una lista circolare, nel senso che una volta scandita tutta la lista per cercare uno spazio di memoria, se non lo trova, ricomincia dall'inizio della lista a cercare.

## Memory Swapping

Un'alternativa a questo schema è lo **swapping**, in cui un processo non deve necessariamente rimanere nella memoria principale durante la sua esecuzione.

In alcuni sistemi di swapping, solamente un processo occupa la memoria principale in un dato istante. Quel processo viene eseguito fino a quando non può più continuare (ad esempio perché deve aspettare per il completamento di un'operazione di I/O), e in quel momento cede sia il processore che la memoria principale al prossimo processo. L'idea è quindi quella di evitare che processi inattivi vadano ad occupare la memoria.



Avremo una area di swap nella quale andiamo a caricare i processi temporaneamente inattivi. In questa maniera si va a massimizzare la disponibilità di memoria per i processi attivi, ma di contro, l'overhead di questa operazione è notevole in quanto questa tecnica prevede molti context switching.

## Memoria Virtuale

Inizialmente i primi sistemi operativi avevano solo una memoria reale, col passare degli anni i sistemi operativi si sono evoluti introducendo la multiprogrammazione. Ora come ora, ai sistemi di memoria reale vengono affiancate le memorie virtuali.

Real		Real		Virtual	
Single-user dedicated systems	Real memory multiprogramming systems		Virtual memory multiprogramming systems		
	Fixed-partition multiprogramming	Variable-partition multiprogramming	Pure paging	Pure segmentation	Combined paging and segmentation
Absolute	Re-locatable				

Come abbiamo detto prima la memoria reale è di dimensione limitata e il numero, e la dimensione, di processi che si vogliono eseguire è sempre maggiore dello spazio necessario a contenerli tutti. Anche avendo tanti processori a disposizione, con una memoria limitata, non si potrebbero eseguire tutti i processi.

L'idea per risolvere il problema sta nell'eliminare dalla memoria i processi che non sono attualmente in esecuzione. Ma dove li andiamo a mettere tutti questi processi eliminati dalla memoria principale? Si crea uno spazio virtuale in cui potere allocare procedure e dati in modo tale che questo stesso spazio di indirizzamento sia molto più ampio dello spazio fisico realmente disponibile. Ciò crea al sistema l'illusione di avere molto più spazio a disposizione.

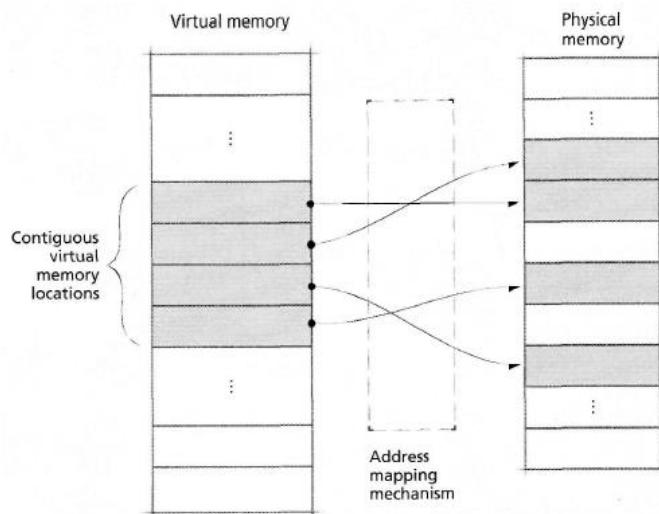
I sistemi di memoria virtuale forniscono ai processi l'illusione di avere più memoria di quella incorporata nel computer. Pertanto, ci sono due tipi di indirizzi nei sistemi di memoria virtuale: quelli a cui fanno riferimento i processi e quelli disponibili nella memoria principale. Gli indirizzi a cui fanno riferimento i processi sono chiamati **indirizzi virtuali**. Quelli disponibili nella memoria principale sono chiamati **indirizzi fisici** (o reali).

I processi quindi, in ogni istruzione, faranno riferimento a degli indirizzi virtuali che poi verranno tradotti in indirizzi fisici. Questa traduzione di indirizzi viene effettuata dalla **MMU (Memory management unit)** che associa rapidamente indirizzi virtuali a indirizzi reali. Avremo dunque due spazi di indirizzamento differenti:

- **Spazio di indirizzamento virtuale V**: l'intervallo di indirizzi virtuali a cui il processo può fare riferimento.
- **Spazio di indirizzamento reale R**: l'intervallo di indirizzi reali disponibili su un particolare sistema informatico.

Nei sistemi di memoria virtuale, normalmente si ha  $|V| >> |R|$ , cioè lo spazio di indirizzamento virtuale è molto più grande di quello reale).

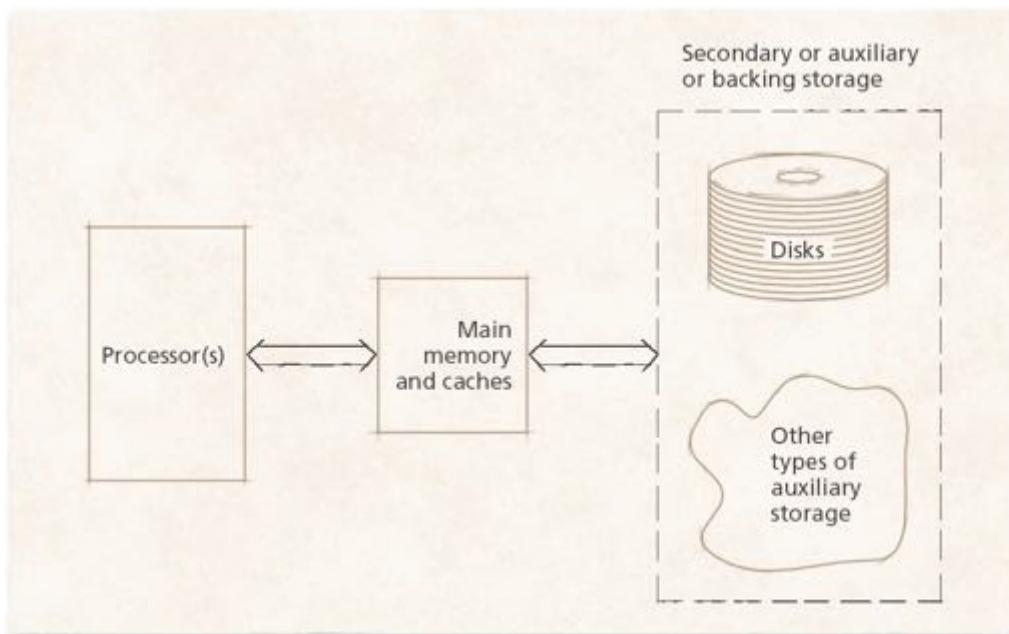
La MMU implementa una traduzione dinamica degli indirizzi chiamata **DAT (Dynamic address translation)**, che serve a convertire dinamicamente durante l'esecuzione di un programma tutti gli indirizzi virtuali in indirizzi reali. Inoltre, i sistemi che utilizzano una traduzione dinamica degli indirizzi hanno un altro vantaggio, possono infatti associare ad un certo processo degli indirizzi virtuali contigui che però corrispondono nella realtà a degli indirizzi reali non contigui nella memoria principale. Questo tipo di caratteristica prende il nome di **contiguità artificiale**, e nello specifico, il vantaggio sta nell'evitare strategie di placement nella memoria virtuale. Vediamo in figura questo concetto:



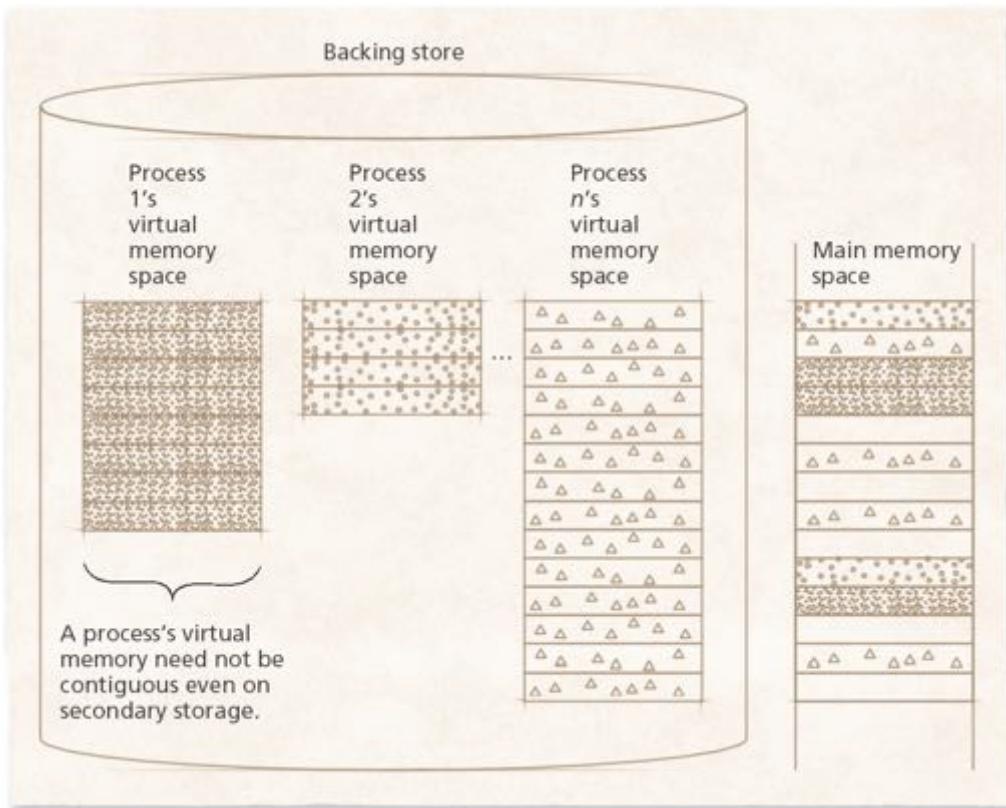
**Figure 10.5 | Artificial contiguity.**

### Mapping dei blocchi

Solitamente la dimensione di V deve essere molto maggiore della dimensione di R (indirizzi reali). Visto che V sarà più grande di R di una certa differenza, questa differenza includerà una quantità di indirizzi memorizzati all'esterno della memoria principale, nel caso dei sistemi attuali, solitamente questi indirizzi vengono caricati nella memoria secondaria:



Quando il sistema è pronto per eseguire un processo, il sistema carica il codice e i dati del processo dalla memoria secondaria alla memoria principale. Solo una piccola parte di questi deve essere subito nella memoria principale affinché il processo possa eseguire. La figura di seguito illustra un sistema di archiviazione di questo tipo in cui gli elementi degli spazi di memoria virtuale di vari processi sono stati collocati nella memoria principale:



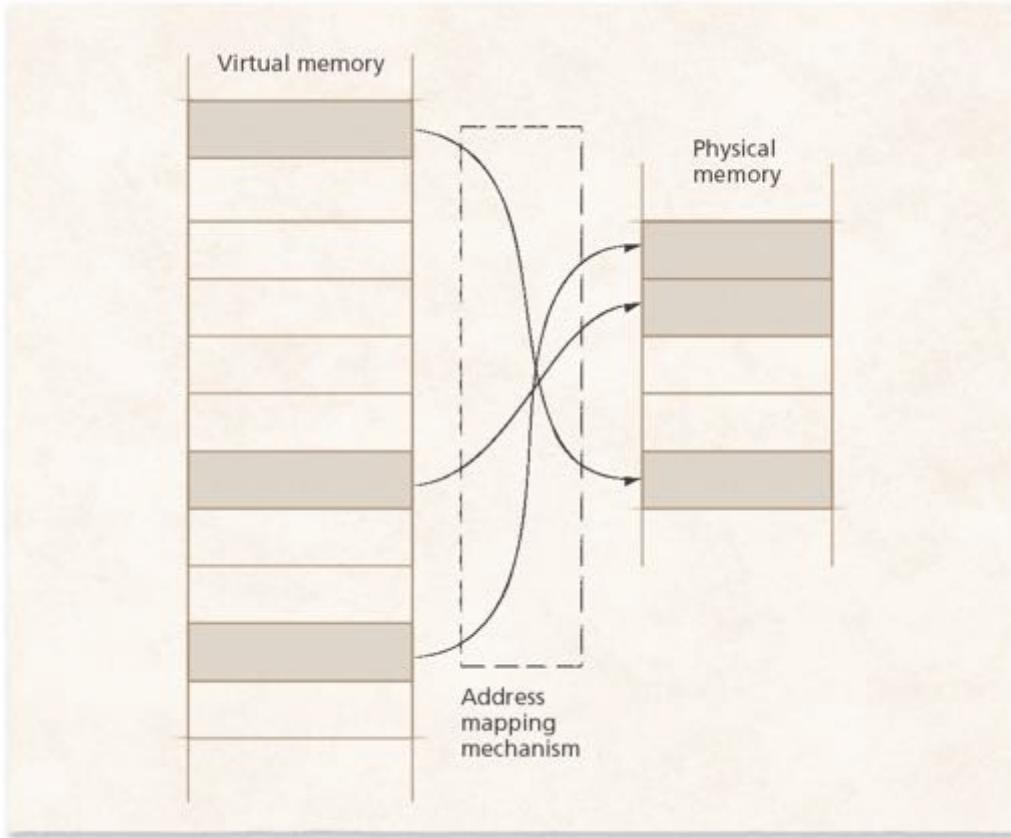
Come vediamo dalla figura, i processi nello spazio di indirizzamento virtuale, sono allocati in maniera contigua, ma nella memoria principale reale, sono in realtà allocati in maniera non contigua.

I meccanismi di traduzione degli indirizzi dinamici devono mantenere le **mappe di traduzione degli indirizzi** che indicano quali regioni dello spazio d'indirizzamento virtuale di un processo sono attualmente nella memoria principale e dove si trovano. Se questa mappatura dovesse contenere voci per ogni indirizzo in V, allora le informazioni di mappatura richiederebbero più spazio di quello disponibile nella memoria principale, quindi questo non è fattibile.

La soluzione più ampiamente implementata consiste nel raggruppare le informazioni in **blocchi**; il sistema tiene quindi traccia di dove nella memoria principale è stato posizionato ogni blocco di memoria virtuale. Maggiore è la dimensione media del blocco, minore è la quantità di informazioni di mappatura. I blocchi più grandi, tuttavia, possono portare a frammentazione interna e possono richiedere più tempo per il trasferimento tra la memoria secondaria e la memoria principale.

Vedremo nei prossimi paragrafi che quando i blocchi hanno dimensione fissa (o più dimensioni fisse), vengono chiamati **pagine** e l'organizzazione della memoria virtuale associata viene chiamata **paginazione**. Quando i blocchi possono essere di dimensioni diverse, vengono chiamati **segmenti** e l'organizzazione della memoria virtuale associata viene chiamata **segmentazione**.

Possiamo immaginare il mapping come una sorta di tabella che fa corrispondere ad indirizzi virtuali determinati indirizzi fisici e reali:



Ma come viene realizzato questo mapping? Sicuramente tradurre un indirizzo alla volta è una operazione molto onerosa per il sistema operativo in quanto gli indirizzi virtuali sono tanti, infatti ciò che si fa è prevedere una tecnica che ci permetta di raggruppare indirizzi e tradurre poi una singola volta il blocco intero di indirizzi virtuali. Blocchi di indirizzi virtuali vengono mappati in corrispondenti blocchi di indirizzi reali. È chiaro che, essendo gli indirizzi virtuali maggiori in numero di quelli reali, il mapping non sarà di tipo 1 a 1, ma bensì di tipo  $m$  ad  $n$  (con  $m$  indirizzi virtuali  $>$   $n$  indirizzi reali).

In generale, più grandi sono i blocchi, minori saranno le operazioni di mapping da eseguire, ma di contro, più i blocchi saranno grandi più è facile che si verifichi la frammentazione.

### Formato degli indirizzi virtuali

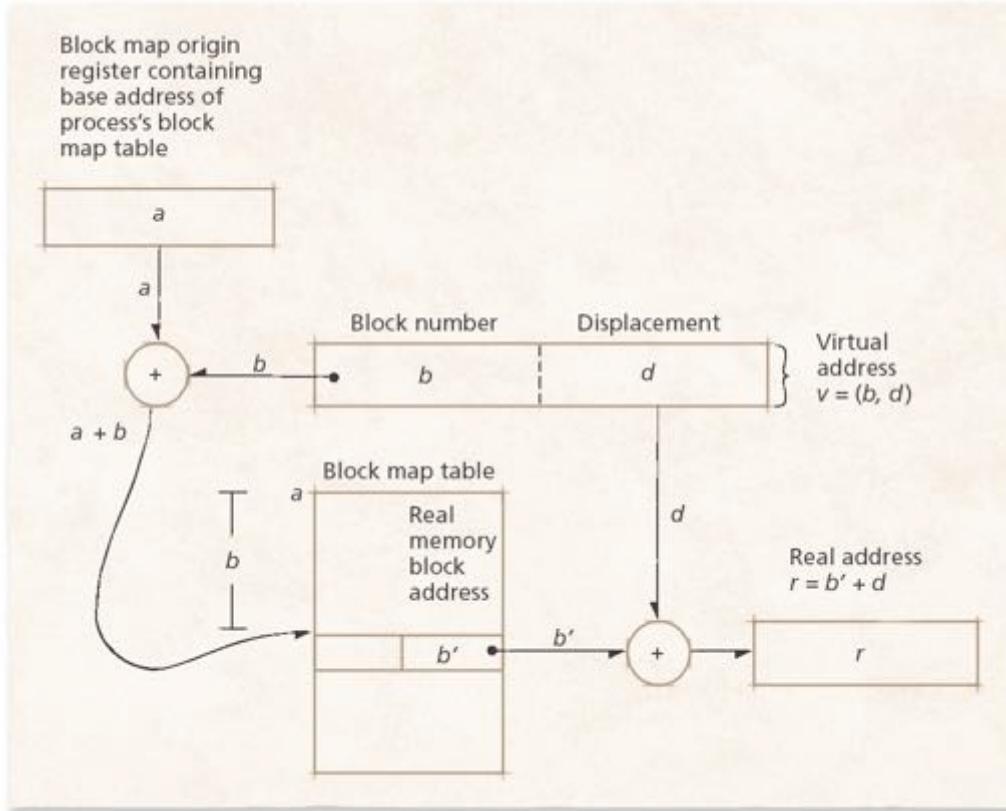
Un indirizzo virtuale è caratterizzato da due elementi che rappresentano un blocco e un offset a partire dall'inizio del blocco stesso:



$b$  è il numero di blocco dove il processo risiede mentre  $d$  è l'offset dall'inizio del blocco. L'obiettivo è di tradurre un indirizzo virtuale in un indirizzo reale, partendo da questi due elementi.

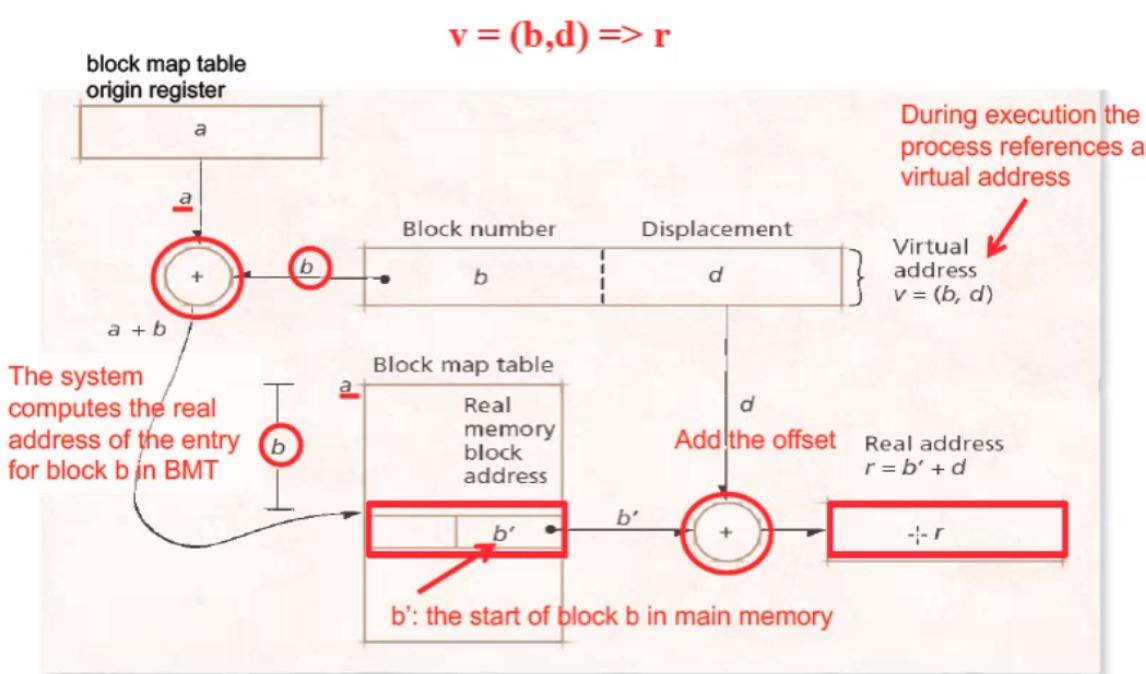
Per farlo il sistema mantiene in memoria principale una tabella di **block map** per ogni processo. La tabella contiene un record per ogni blocco del processo, i quali vengono mantenuti in ordine. Per andare a determinare l'indirizzo del processo nella **block map table** si usa un registro del processore chiamato **block map origin register**.

Vediamo un esempio:



Quando avviene il context switching l'OS va a recuperare la tabella che contiene il mapping per il processo corrente, in particolare andiamo a prendere dal **block map origin register** l'indirizzo  $a$  che contiene l'indirizzo della **block map table** per il processo corrente. Il sistema aggiunge il numero di blocco  $b$ , all'indirizzo di base  $a$ , della **block map table** del processo per formare l'indirizzo reale della entry per il blocco  $b$  nella tabella di mappatura dei blocchi. Il risultato della somma tra  $a$  e  $b$  sarà  $b'$  che contiene l'indirizzo di inizio del blocco  $b$  nella memoria principale, che stiamo cercando. Arrivati a questo punto si aggiunge quindi lo spostamento  $d$ , all'indirizzo iniziale del blocco  $b'$ , per formare l'indirizzo reale desiderato  $r$ .

Nella figura di seguito vediamo quando abbiamo descritto precedentemente:



a: indirizzo della block map table per il processo corrente

b: blocco dell'indirizzo cercato

d: offset dell'indirizzo cercato rispetto al blocco

Ricapitolando quindi: il sistema calcola l'indirizzo reale  $r$  a partire dall'indirizzo virtuale  $v = (b, d)$ , tramite l'equazione  $r = b' + d$ , dove  $b'$  è memorizzato nella cella della block map table situata all'indirizzo della memoria reale  $a + b$ .

# Lezione 27

---

L'ultima lezione abbiamo parlato di memorie reali e virtuali, in particolare ci siamo fermati alla traduzione degli indirizzi con mappatura a blocchi, riprendiamo ora continuando il discorso sulle memorie virtuali.

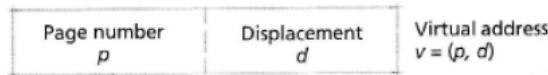
## Definizione dei blocchi

Per definire i blocchi dobbiamo innanzitutto decidere la dimensione dei blocchi, che può essere fissa o variabile. Nel caso in cui i blocchi siano di dimensione fissa prendono il nome di **pagine** e diremo che il sistema di gestione della memoria virtuale si basa sulla **paginazione**; l'alternativa vede l'uso di blocchi di dimensione variabile (che prendono il nome di **segmenti**) e la tecnica di gestione della memoria virtuale si chiamerà **segmentazione**.

Esiste una tecnica mista tra le due dove si utilizzano sia segmenti che pagine, in particolare ci sono pagine di dimensioni fisse raggruppate all'interno di segmenti di dimensione variabile. Lo vedremo alla fine della lezione.

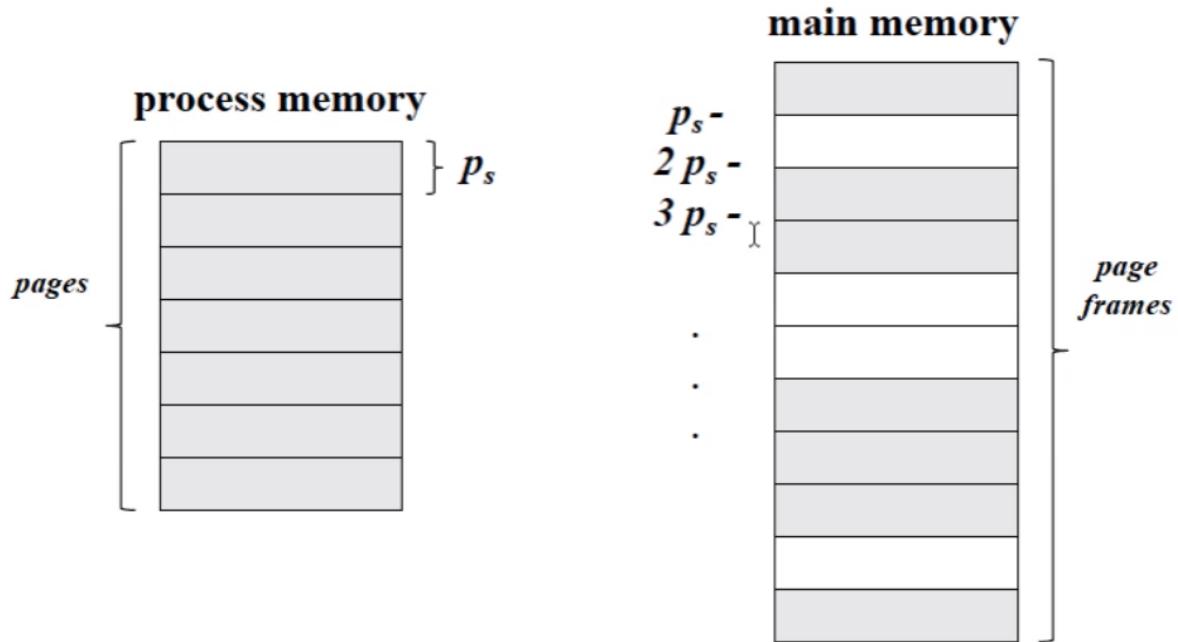
## Paginazione

Le pagine utilizzano blocchi di dimensione fissa, ciò significa che l'indirizzo virtuale sarà nella forma  $v=(p, d)$  dove  $p$  è il numero della pagina e  $d$  rimane sempre l'offset a partire dall'inizio della pagina:



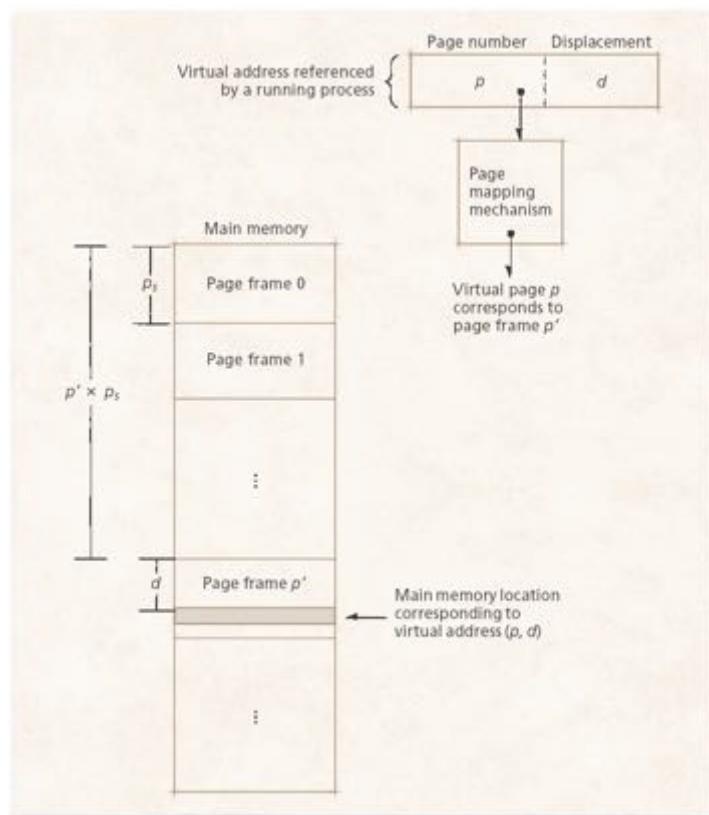
Affinché un processo sia in esecuzione è necessario che le pagine correnti che utilizza siano caricate nella memoria principale. Quando il sistema trasferisce una pagina dalla memoria secondaria alla memoria principale, colloca la pagina in un blocco di memoria principale, chiamato **page frame**, che ha le stesse dimensioni della pagina in entrata. C'è quindi una distinzione tra il blocco di memoria virtuale che sta nella memoria secondaria ed è chiamato semplicemente **pagina** e il blocco di memoria reale allocato in memoria principale chiamato **page frame**.

Tutti i page frame, come le pagine, hanno una dimensione prefissata  $p_s$ , di conseguenza potremo immaginare così la situazione in memoria:



Gli indirizzi dei page frame si possono quindi sempre calcolare come multipli del valore  $p_s$ . Quando si caricherà una pagina in memoria principale il sistema potrà posizionare una pagina in arrivo in qualsiasi page frame disponibile in quanto sono di dimensione prefissata e quindi tra un page frame e l'altro non cambia nulla.

Vediamo come avviene il processo di traduzione nel caso della paginazione:

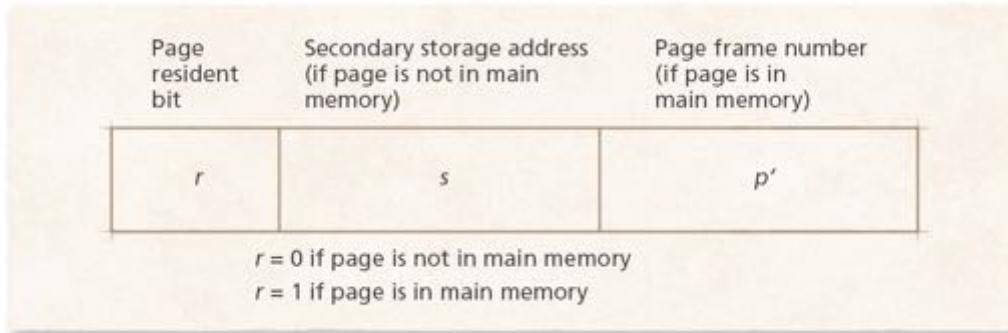


Un processo in esecuzione fa riferimento a un indirizzo di memoria virtuale  $v = (p, d)$ . Un meccanismo di mappatura della pagina, cerca la pagina  $p$  nella **Page Table** e determina che la pagina  $p$  si trova nel page frame  $p'$ . Notiamo che  $p'$  è un numero di page frame, non un indirizzo di memoria fisica. Supponendo che i page frame siano numerati  $\{0, 1, 2, \dots, n\}$ , l'indirizzo di memoria fisica in cui inizia il page frame  $p'$  è il prodotto di  $p'$  per la dimensione fissa della pagina,  $p' \times p_s$ . L'indirizzo di riferimento è formato aggiungendo l'offset  $d$ , all'indirizzo di memoria fisica al quale inizia il page frame  $p'$ . Quindi, l'indirizzo di memoria reale è  $r = (p' \times p_s) + d$ .

Un vantaggio di un sistema di memoria virtuale a pagine è che non tutte le pagine appartenenti a un processo devono risiedere nella memoria principale nello stesso momento: la memoria principale deve mantenere solo la pagina (o le pagine) in cui il processo sta facendo riferimento a un indirizzo (o indirizzi). Proprio per questo motivo nella memoria principale possono essere allocati più processi contemporaneamente.

Poiché la memoria principale normalmente non contiene tutte le pagine di un processo contemporaneamente, utilizziamo una **page table** che serve ad indicare se una pagina mappata risiede o meno nella memoria principale.

La page table è divisa in **PTE (Page Table Entry)**, le quali hanno la seguente struttura:

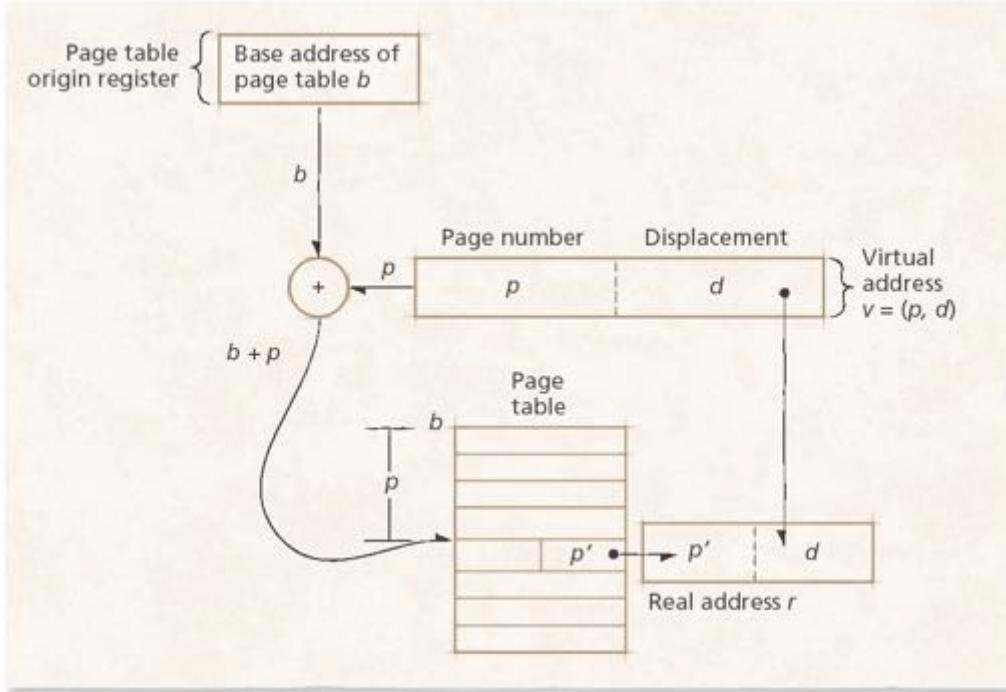


Un bit  $r$  (**resident bit**), è impostato su 0 se la pagina non è nella memoria principale e a 1 se lo è. Se la pagina non si trova nella memoria principale,  $s$  è il suo indirizzo di archiviazione secondario. Se la pagina è nella memoria principale, allora  $p'$  è il suo numero di frame.

Nel momento in cui ci accorgiamo che  $r$  è uguale a 0 per un certo processo che deve essere eseguito, allora dovremo caricare quella pagina dalla memoria secondaria in memoria principale, in particolare, quando un processo fa riferimento a una pagina che non si trova nella memoria principale, il processore genera un **page fault**, che richiama il sistema operativo per caricare la pagina mancante nella memoria principale dalla memoria secondaria. Questa operazione, quando capita, chiaramente comporta un rallentamento dovuto alla interruzione di un dato processo che non può proseguire in quanto gli manca una pagina per continuare la sua esecuzione.

### Traduzione degli indirizzi con paginazione a mappatura diretta

La prima possibilità che abbiamo è il **mapping diretto**, che è molto simile alla mappatura a blocchi che abbiamo visto nella precedente lezione.



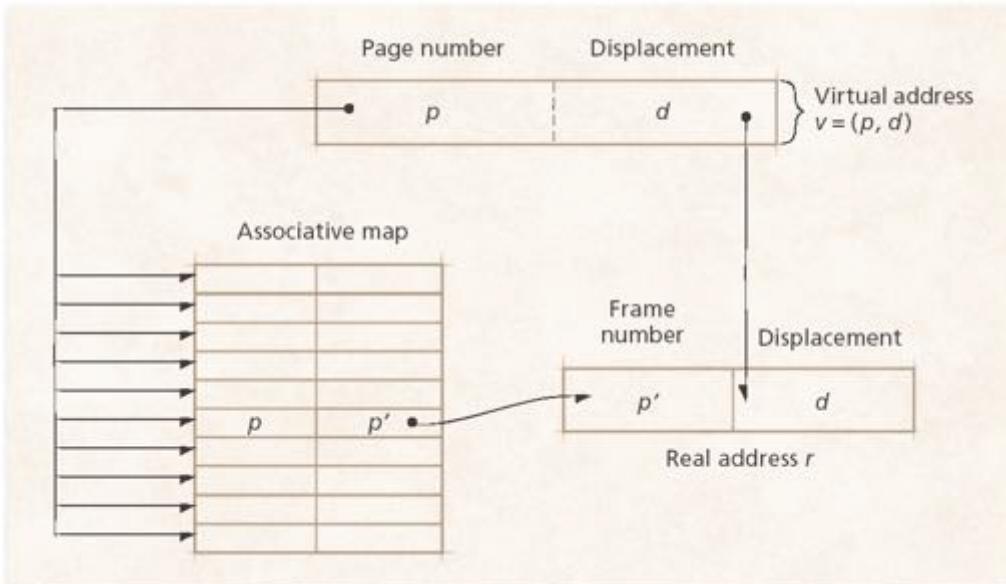
Un processo fa riferimento all'indirizzo virtuale  $v = (p, d)$ . Al momento del context switching, il sistema operativo carica l'indirizzo di memoria principale della page table di un processo nel **registro di origine della page table (page table origin register)**. Per determinare l'indirizzo di memoria principale che corrisponde all'indirizzo virtuale referenziato, il sistema prima aggiunge l'indirizzo di base della page table del processo  $b$ , al numero di pagina di riferimento,  $p$  (cioè  $p$  è l'indice nella page table). Questo risultato,  $b + p$ , è l'indirizzo di memoria principale della PTE per pagina  $p$ .

Questo PTE indica che la pagina virtuale  $p$  corrisponde al page frame  $p'$ . Il sistema quindi somma  $p'$  con l'offset  $d$  per formare l'indirizzo reale  $r$ .

Il problema di questo tipo di mapping è dovuto alla dimensione della Page Table, che deve mantenere una riga per ogni pagina di ogni processo. Allora una tecnica alternativa è l'utilizzo del mapping associativo.

### Traduzione degli indirizzi con paginazione a mappatura associativa

Questo metodo di traduzione fa uso della **memoria associativa** tramite la quale si possono effettuare ricerche simultanee in tutte le celle e che ha un tempo di lettura maggiore di un ordine di grandezza più veloce della memoria principale.

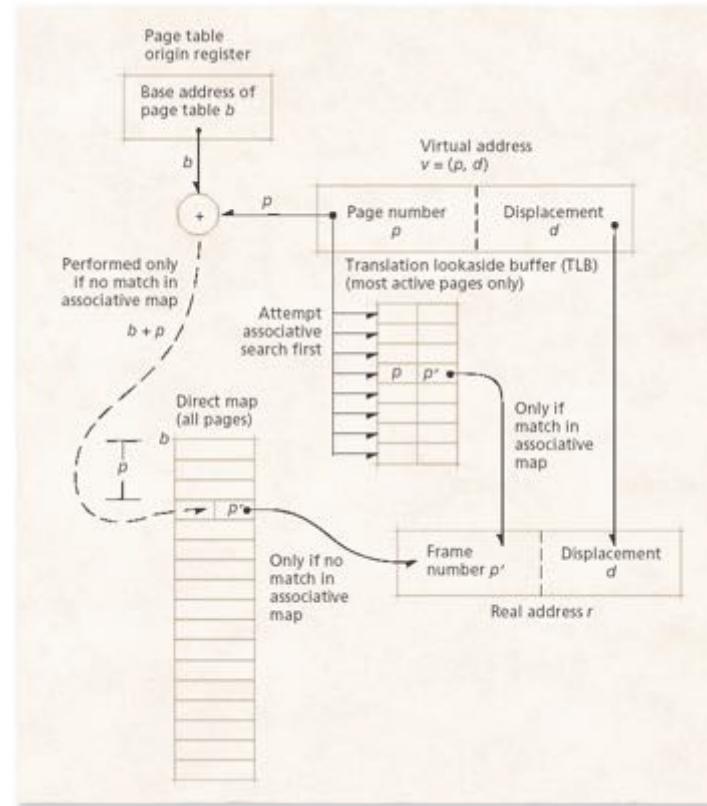


Un processo si riferisce all'indirizzo virtuale  $v = (p, d)$ . Ogni voce nella memoria associativa viene cercata simultaneamente per pagina  $p$  in tutta la tabella. La ricerca restituisce  $p'$  come il page frame corrispondente alla pagina  $p$ , e  $p'$  è sommato all'offset  $d$ , formando l'indirizzo reale  $r$ .

Il problema principale di questa tecnica è che la memoria associativa è molto costosa e quindi è difficile andare a realizzare memorie di grandi dimensioni come servirebbero a noi.

### Mapping diretto-associativo

È un approccio ibrido che utilizza sia il mapping diretto sia quello associativo. L'idea è di usare le page table del mapping diretto (di dimensioni elevate) per mantenere la maggior parte delle pagine, mentre altre pagine (in numero decisamente inferiore) vengono associate ad una memoria associativa di nome **translation lookaside buffer (TLB)**. Le pagine che andiamo a caricare nella TLB sono quelle che abbiamo utilizzato più recentemente poiché hanno più probabilità di essere utilizzate. Lo schema del seguente mapping è quello mostrato in figura:



L'idea è quella di cercare una entry direttamente dentro la TLB, se la trovo la carico in memoria principale, altrimenti mediante un meccanismo di traduzione la vado a cercare nella page table. Il meccanismo di mappatura della pagina tenta innanzitutto di trovare la pagina  $p$  nel TLB. Se il TLB contiene  $p$ , la ricerca restituisce  $p'$  come numero di frame corrispondente alla pagina virtuale  $p$ , e  $p'$  è concatenato con lo spostamento  $d$  per formare l'indirizzo reale,  $r$ . Nel caso in cui troviamo la pagina subito nella TLB riscontriamo un cosiddetto **TLB hit**.

E nel caso in cui non trovassimo la pagina cercata nel TLB?

Se il TLB non contiene una entry per la pagina  $p$  (cioè, il sistema riscontra un **TLB miss**), il sistema individua la entry della page table utilizzando una mappa diretta convenzionale nella memoria principale più lenta, cosa che aumenta il tempo di esecuzione. L'indirizzo nel page table origin register,  $b$ , viene aggiunto al numero di pagina,  $p$ , per individuare la entry appropriata per la pagina  $p$  nella page table con mappatura diretta nella memoria principale. La voce contiene  $p'$ , il page frame corrispondente alla pagina virtuale  $p$ . A questo punto  $p'$  viene concatenato con lo spiazzamento,  $d$ , per formare l'indirizzo reale,  $r$ . Il sistema dopo aver effettuato questa mappatura diretta posiziona la PTE appena trovata nel TLB in modo che i riferimenti alla pagina nel prossimo futuro possano essere tradotti rapidamente. Se il TLB è pieno, il sistema sostituisce

una entry (in genere quella con riferimenti meno recenti) per creare spazio per la entry per la pagina corrente.

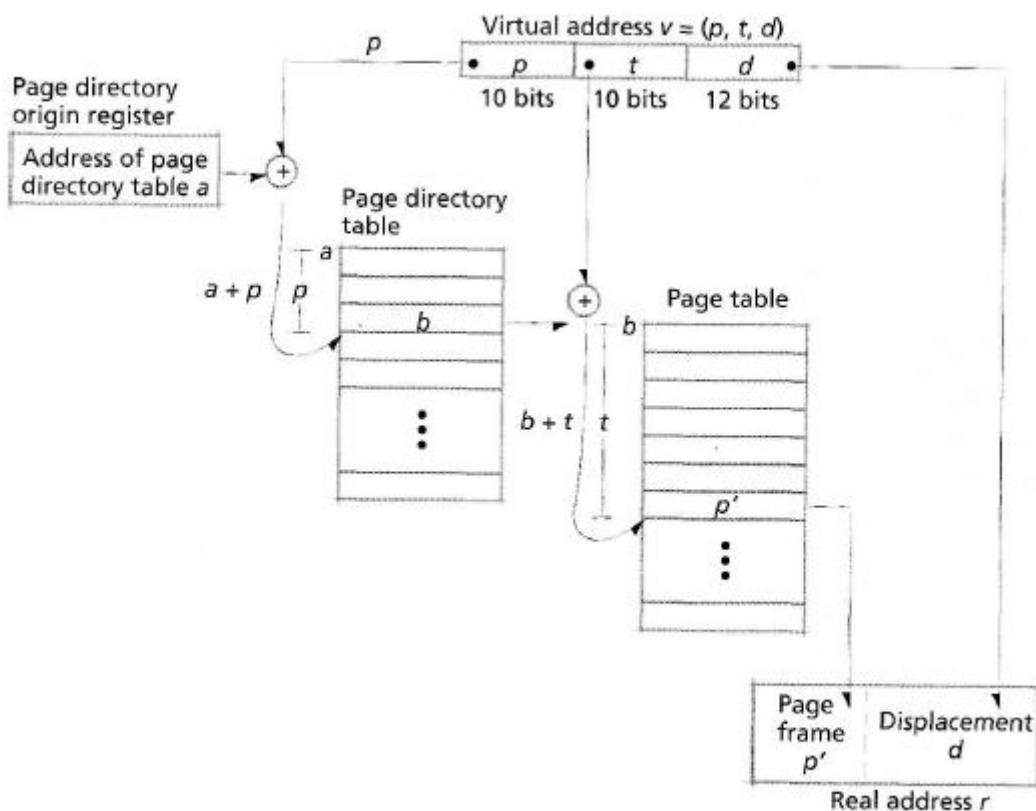
### Page Table multilivello

Una limitazione delle traduzioni degli indirizzi che utilizzano la mappatura diretta è che tutte le PTE di una page table devono essere nella mappa e memorizzate in ordine contiguo e sequenziale per numero di pagina e questo porta anche ad un notevole consumo di memoria.

Le **page table multilivello** (o gerarchiche) consentono al sistema di memorizzare in posizioni non contigue nella memoria principale quelle parti della page table di un processo utilizzate attivamente da questo. Le parti rimanenti possono essere create la prima volta che vengono utilizzate e spostate nella memoria secondaria quando cessano di essere utilizzate attivamente. Le page table multilivello vengono implementate creando una gerarchia in cui ogni livello contiene una tabella che memorizza i puntatori alle tabelle nel livello sottostante. Il livello più in basso è composto da tabelle contenenti la mappatura di ogni pagina ai rispettivi page frame.

Nel caso di pagine multilivello l'indirizzo virtuale si rappresenta come  $v = (p, t, d)$ , dove la coppia ordinata  $(p, t)$  è il numero di pagina e  $d$  è lo spiazzamento in quella pagina.

Per capire bene le pagine multilivello consideriamo un sistema che usa due livelli di page table e consideriamone la traduzione, che seguirà il seguente schema:



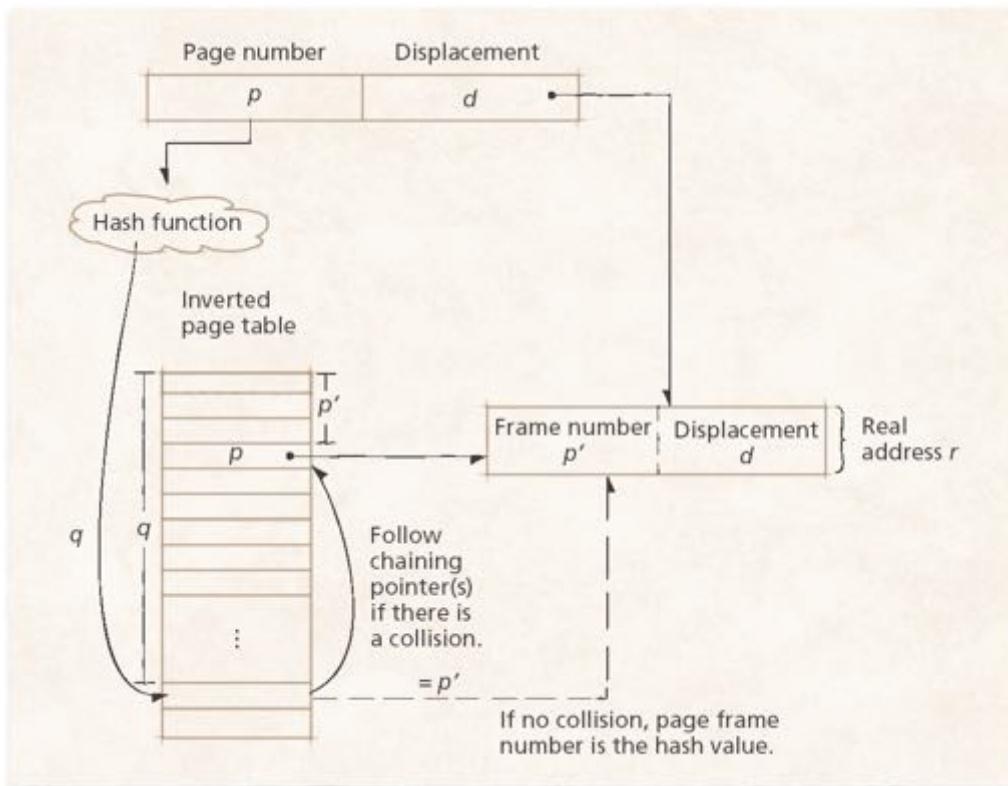
**Figure 10.15 | Multilevel page address translation.**

Il sistema aggiunge innanzitutto il valore di  $p$  all'indirizzo nella memoria principale dell'inizio della page directory,  $a$ , memorizzato nel registro di origine delle page directory. La entry in posizione  $a + p$  contiene l'indirizzo della page table corrispondente,  $b$ . Il sistema somma  $t$  e  $b$  per individuare la entry della page table che memorizza il numero di page frame,  $p'$ . Infine, il sistema forma il vero indirizzo  $r$  concatenando  $p'$  e lo spiazzamento,  $d$ .

### Page Table invertite

Le **page table invertite** memorizzano esattamente una PTE per ogni page frame nel sistema. Le page table invertite usano le **funzioni hash** per mappare le pagine virtuali su PTE. Poiché il dominio di una funzione di hash (ad esempio i numeri di una pagina virtuale di un processo) è generalmente più ampio dell'intervallo (ad esempio i numeri di page frame), più input possono generare lo stesso valore hash: queste situazioni sono chiamati **collisioni**. Per evitare che più elementi vengano sovrascritti l'un l'altro quando mappati sulla stessa cella della tabella hash, le page table invertite possono implementare una variante di un meccanismo di concatenamento per risolvere collisioni come segue: quando un valore hash mappa un elemento in una posizione occupata, al valore hash viene applicata una nuova funzione. Il valore risultante viene utilizzato come posizione nella tabella in cui deve essere inserito l'input. Per garantire che l'elemento possa essere trovato quando referenziato, un puntatore a questa posizione viene aggiunto alla voce nella cella corrispondente al valore hash originale dell'oggetto. Questo processo viene ripetuto ogni volta che si verifica una collisione.

Vediamo come avviene la traduzione mediante l'uso di page table invertite:

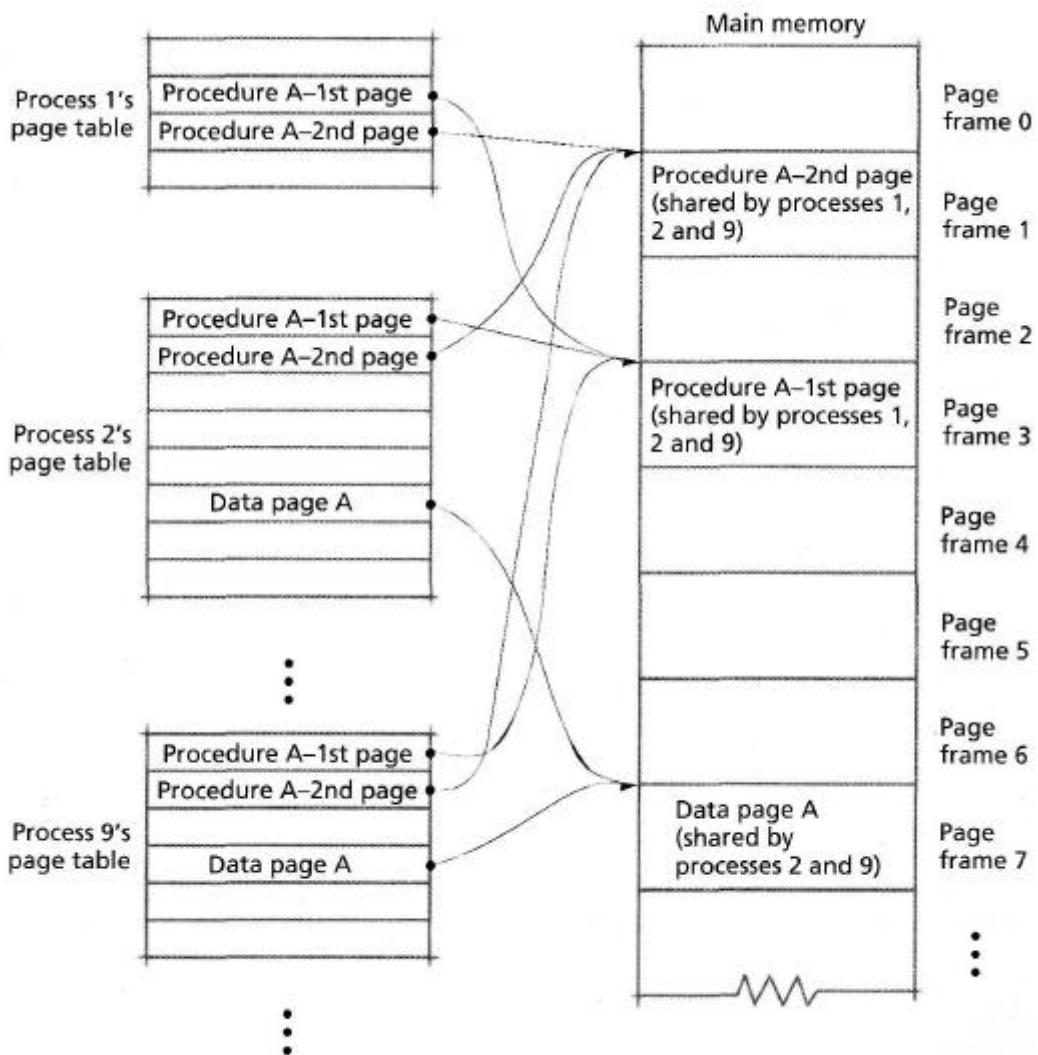


In un sistema di paginazione che implementa page table invertite, ciascun indirizzo virtuale è rappresentato dalla coppia ordinata  $v = (p, d)$ . Per individuare rapidamente l'entry della tabella hash corrispondente alla pagina virtuale  $p$ , il sistema applica una funzione hash a  $p$ , che produce un valore  $q$ . Se la  $q$ -esima cella nella page table invertita contiene  $p$ , l'indirizzo virtuale richiesto è nel page frame  $q$ . Se il numero di pagina nella  $q$ -esima cella della page table invertita non corrisponde a  $p$ , il sistema controlla il valore del puntatore di concatenamento a quella cella. Se il puntatore è nullo, la pagina non è in memoria, quindi il processore genera un page fault. Il sistema operativo può quindi recuperare la pagina dalla memoria secondaria. Altrimenti, c'è stata una collisione a quell'indice nella page table invertita, quindi l'entry della page table memorizza un puntatore alla entry successiva nella catena. Il sistema segue i puntatori della catena finché non trova una entry contenente  $p$  o finché la catena non termina, a quel punto il processore genera un page fault per indicare che la pagina non è presente.

### Condivisione in un sistema di paginazione

Nei sistemi attuali può capitare spesso che molti utenti utilizzino simultaneamente gli stessi programmi. Se il sistema assegnasse singole copie di questi programmi a ciascun utente, gran parte della memoria principale sarebbe sprecata. La soluzione ovvia è che il sistema condivida quelle pagine comuni ai singoli processi. Nasce quindi la necessità di identificare ogni pagina come **condivisibile** o **non condivisibile**.

Fatto ciò la condivisione delle pagine può essere implementata come segue:



*Figure 10.18 | Sharing in a pure paging system.*

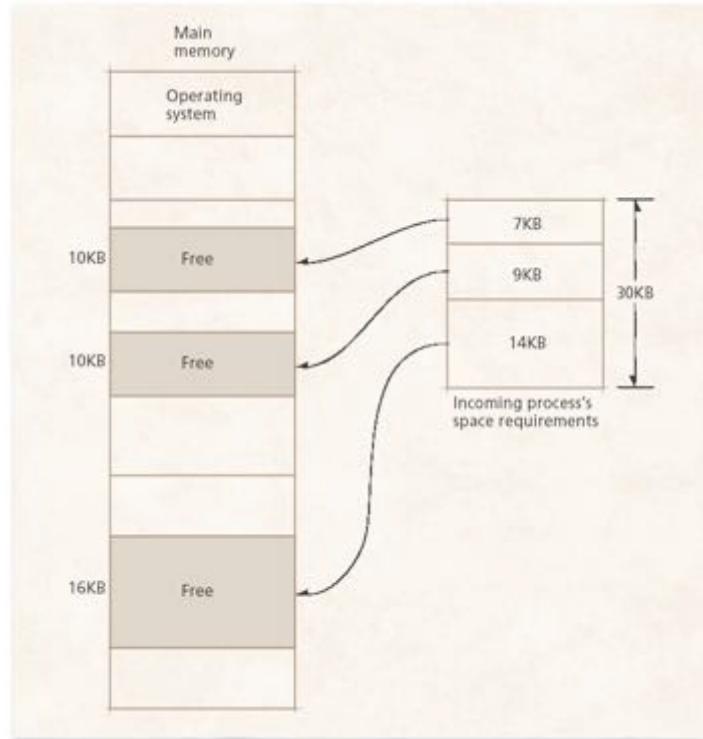
Se le voci della page table di diversi processi puntano allo stesso page frame, questo page frame viene condiviso da ciascuno dei processi. La condivisione riduce la quantità di memoria principale richiesta affinché un gruppo di processi possa funzionare in modo efficiente e può consentire a un determinato sistema di aumentare il livello di multiprogrammazione. Il problema che nasce da questo sistema di condivisione è dovuto al fatto che alcuni processi potrebbero interferire tra di loro.

Molti sistemi operativi utilizzano una tecnica chiamata **copy-on-write** per risolvere questo problema. Inizialmente, il sistema mantiene una copia di ciascuna pagina condivisa in memoria per i processi da P1 a Pn. Se P1 tenta di modificare una pagina di memoria condivisa, il sistema operativo crea una copia della pagina, applica la modifica e assegna la nuova copia per elaborare lo spazio degli indirizzi virtuali di P1. La copia non modificata della pagina rimane mappata allo spazio d'indirizzamento di tutti gli altri processi che condividono la pagina. Ciò garantisce che quando un processo modifica una pagina condivisa, nessun altro processo ne risenta.

## Segmentazione

Questa tecnica prevede l'uso di blocchi di dimensione non fissa, bensì differente, ogni segmento è composto da locazioni contigue; però, i segmenti non hanno bisogno di essere della stessa grandezza o di essere posti l'uno adiacente all'altro nella memoria principale. Il principale vantaggio della segmentazione è dato dal meccanismo di mapping, dove sarà possibile andare a dividere il blocco in tante piccole pagine.

Così come avviene con la paginazione se un processo fa riferimento alla memoria in un segmento che non risiede attualmente nella memoria principale, il sistema di memoria virtuale deve recuperare quel segmento dalla memoria secondaria.



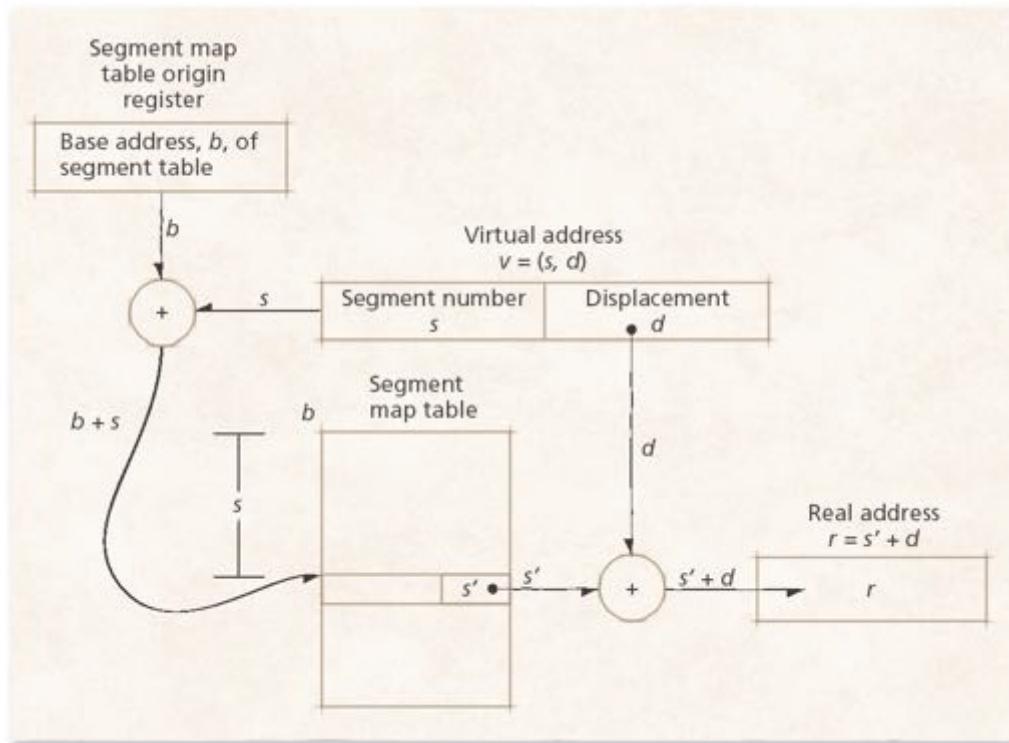
L'allocazione dei vari segmenti in posizione di memoria può avvenire in ordine sparso, come si vede nella figura di sopra. In particolare un **indirizzo di segmentazione della memoria virtuale** è una coppia ordinata  $v = (s, d)$ , dove  $s$  è il numero del segmento nella memoria virtuale nel quale risiede l'elemento referenziato, e  $d$  è lo spiazzamento all'interno del segmento  $s$  in cui si trova l'elemento referenziato:



Ciò che cambia rispetto alla paginazione è che in questo caso per trovare il segmento che cerchiamo non possiamo più cercare i multipli della dimensione del segmento, poiché per definizione ogni segmento ha una sua dimensione differente rispetto agli altri segmenti.

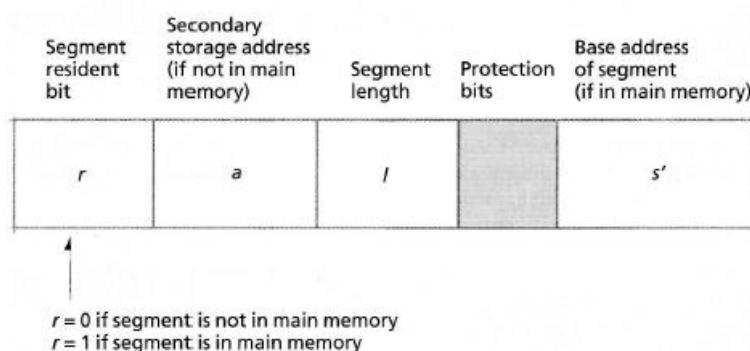
#### **Traduzione di indirizzi di segmentazione mediante mappatura diretta**

Un processo fa riferimento ad un indirizzo della memoria virtuale  $v = (s, d)$  per determinare dove risiede il segmento referenziato nella memoria principale. Il sistema aggiunge il numero di segmento,  $s$ , al valore dell'indirizzo di base della **segment map table**,  $b$ , collocato nel **registro di origine della segment map table**. Il valore risultante,  $b + s$ , è la collocazione della entry della segment map table. Vediamo sotto in figura cosa avviene:



Se il segmento risiede attualmente nella memoria principale, la entry contiene l'indirizzo iniziale del segmento della memoria principale,  $s'$ . Il sistema aggiunge lo spiazzamento  $d$  a questo indirizzo per formare l'indirizzo della memoria reale della locazione referenziata,  $r = s' + d$ . Non possiamo semplicemente concatenare  $d$  a  $s'$ , come in un sistema di paging puro, poiché i segmenti sono di grandezza variabile, ecco perché dobbiamo utilizzare, come nella paginazione, la **segment map table** per contenere le **STE (segment table entry)** che di danno le informazioni per la traduzione.

Anche nella segment map table quindi avremo delle **STE (segment table entry)** di struttura simile alle PTE(page table entry):



Un resident bit  $r$ , indica se il segmento è attualmente nella memoria principale o no. Se lo è, allora  $s'$  è l'indirizzo della memoria principale al quale inizia il segmento. Altrimenti  $a$  è l'indirizzo della memoria secondaria dal quale il segmento deve essere recuperato prima che il processo possa procedere. Tutti i riferimenti al segmento sono confrontati con la lunghezza del segmento  $l$ , per assicurare che essi cadano all'interno del range del segmento.

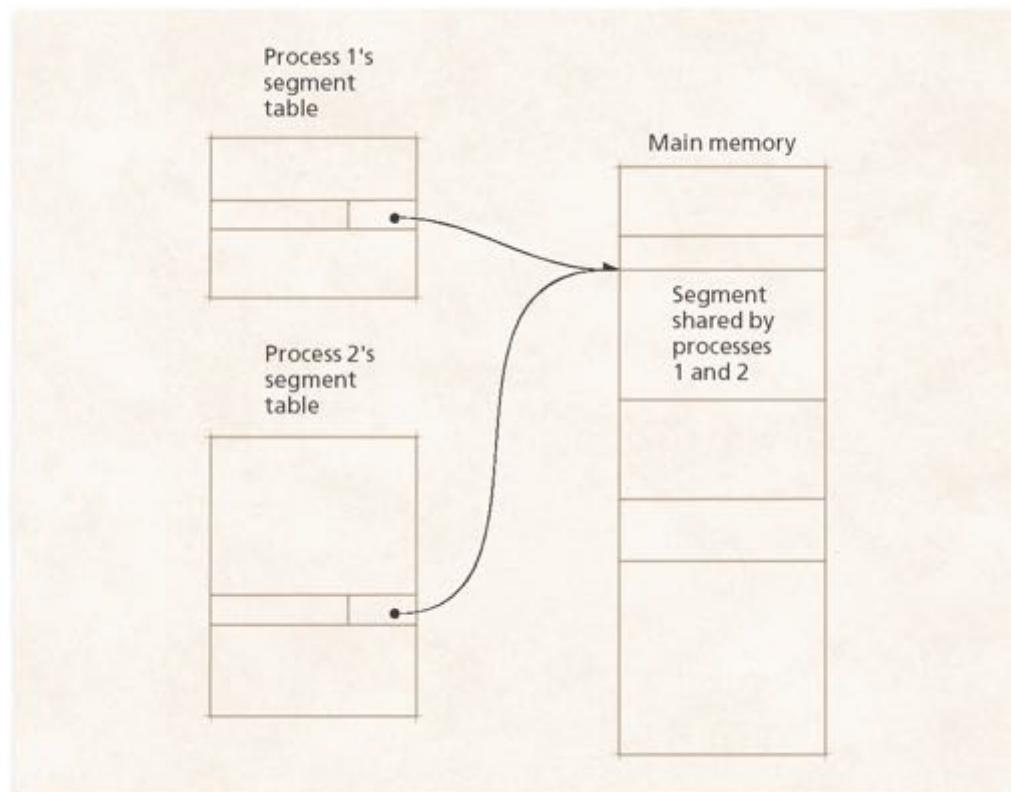
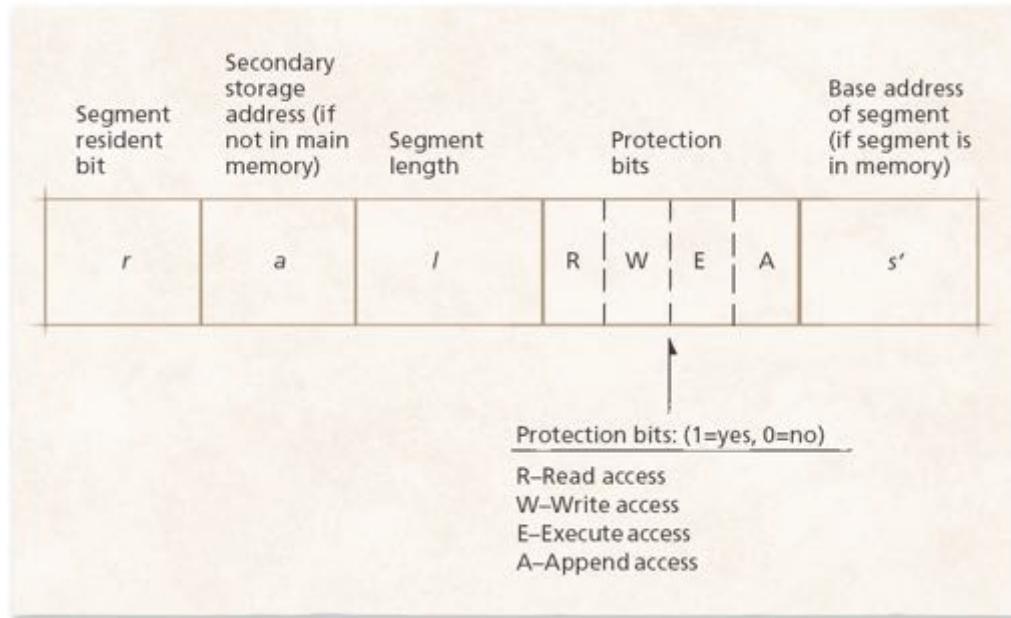
Durante la traduzione dinamica degli indirizzi, una volta che la entry della segment map table  $s$  è stata trovata, il resident bit,  $r$ , viene esaminato per determinare se il segmento è nella memoria principale. Se non lo è, allora viene generato un **missing-segment fault**, che provoca l'ottenimento del controllo da parte del sistema operativo e il caricamento del segmento referenziato che inizia dall'indirizzo a della memoria secondaria. Una volta che il segmento viene caricato, la traduzione degli indirizzi procede con il controllo che lo spiazzamento  $d$ , sia minore o uguale alla lunghezza del segmento,  $l$ . Se non è così, allora viene generato una eccezione di

**segment-overflow**, causando l'ottenimento del controllo da parte del sistema operativo e potenzialmente la terminazione del processo. Allora l'indirizzo di base,  $s'$ , del segmento nella memoria principale viene aggiunto allo spiazzamento  $d$ , per formare l'indirizzo della memoria reale  $r = s' + d$  che corrisponde all'indirizzo di memoria virtuale  $v = (s, d)$ .

Infine notiamo come ci sia una sezione, nella STE, denominata "protection bits". Questi bit servono a capire se l'operazione che il sistema sta cercando di eseguire è permessa o meno, in particolare serviranno per la condivisione della memoria che spieghiamo di seguito.

### Condivisione della memoria tramite segmentazione

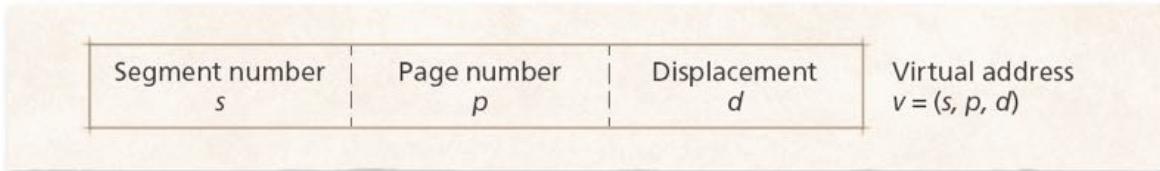
A volte può essere comodo condividere segmenti tra più processi. Sebbene la condivisione fornisca degli ovvi benefici, introduce anche certi rischi. Ad esempio, un processo potrebbe, intenzionalmente o inavvertitamente, compiere una operazione su un segmento che influenza negativamente gli altri processi che condividono quel segmento. A tale scopo vengono utilizzati i **protection bits** del STE che sono i seguenti:



## Sistemi di segmentazione-paginazione

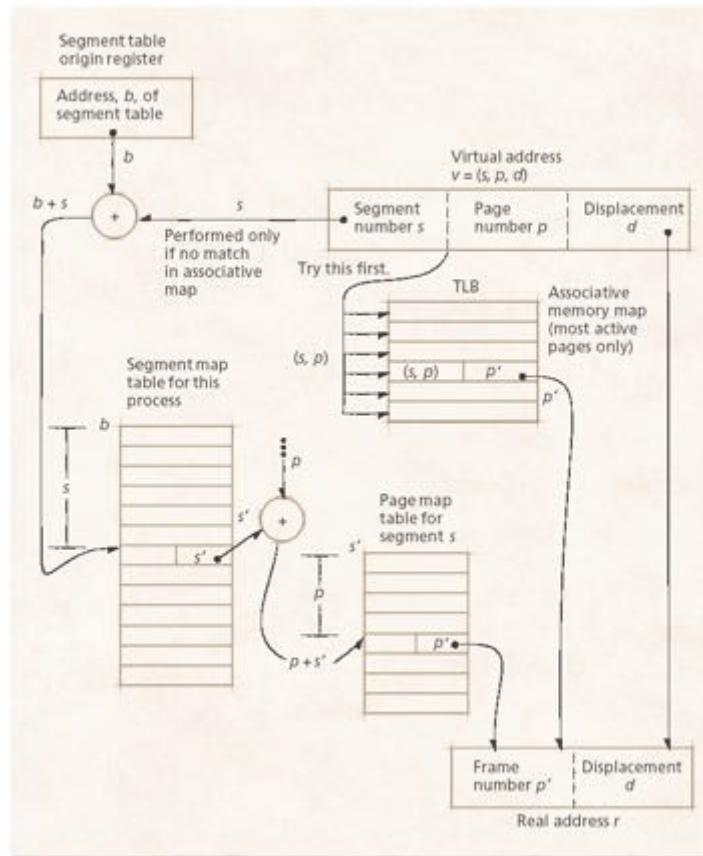
In informatica si tende spesso a creare delle soluzioni ibride che prendano i vantaggi di più tecniche. In un **sistema combinato di segmentazione/paginazione**, i segmenti contengono più pagine. Tutte le pagine di un segmento non devono necessariamente essere nella memoria principale contemporaneamente e le pagine di memoria virtuale che sono contigue nella memoria virtuale non devono essere contigue nella memoria principale. Sotto questo schema, un indirizzo di memoria virtuale è implementato come  $v = (s, p, d)$ , dove  $s$  è il numero del segmento,  $p$  il numero di pagina all'interno del segmento e  $d$  lo spostamento all'interno della pagina in cui l'elemento desiderato si trova.

Vediamo quale è la struttura di un indirizzo virtuale che utilizza sia segmentazione che paginazione:



## Traduzione in sistemi di segmentazione e paginazione

Vediamo lo schema di traduzione di questi schemi ibridi:



Un processo fa riferimento all'indirizzo virtuale  $v = (s, p, d)$ . Gli intervalli di referenziati più di recente hanno una entry nella mappa della memoria associativa (cioè il TLB). Il meccanismo di traduzione esegue una ricerca associativa per tentare di individuare  $(s, p)$ . Se il TLB contiene  $(s, p)$ , la ricerca restituisce  $p'$ , il page frame in cui si trova la pagina  $p$ . Questo valore è concatenato con lo spostamento,  $d$ , per formare l'indirizzo di memoria reale  $r$ .

Se il TLB non contiene una entry per  $(s, p)$ , il processore deve eseguire una mappatura diretta completa come segue. L'indirizzo di base,  $b$ , della tabella di mappatura di segmento (nella memoria principale) viene aggiunto al numero del segmento,  $s$ , per formare l'indirizzo,  $b + s$ . Questo indirizzo corrisponde alla posizione fisica della memoria della entry del segmento nella

tabella di mappatura di segmento. La entry della tabella di mappatura di segmento indica l'indirizzo di base,  $s'$ , della page table (nella memoria principale) per i segmenti  $s$ . Il processore aggiunge il numero di pagina,  $p$ , all'indirizzo di base,  $s'$ , per formare l'indirizzo della voce della page table per la pagina  $p$  del segmento  $s$ . Questa voce di tabella indica che  $p'$  è il numero del page frame corrispondente alla pagina virtuale  $p$ . Questo numero di frame,  $p'$ , è concatenato con lo spostamento,  $d$ , per formare l'indirizzo reale,  $r$ . La traduzione viene quindi caricata nel TLB.

Per usare questo tipo di sistema ibrido si fa uso di tabelle specifiche così strutturate:

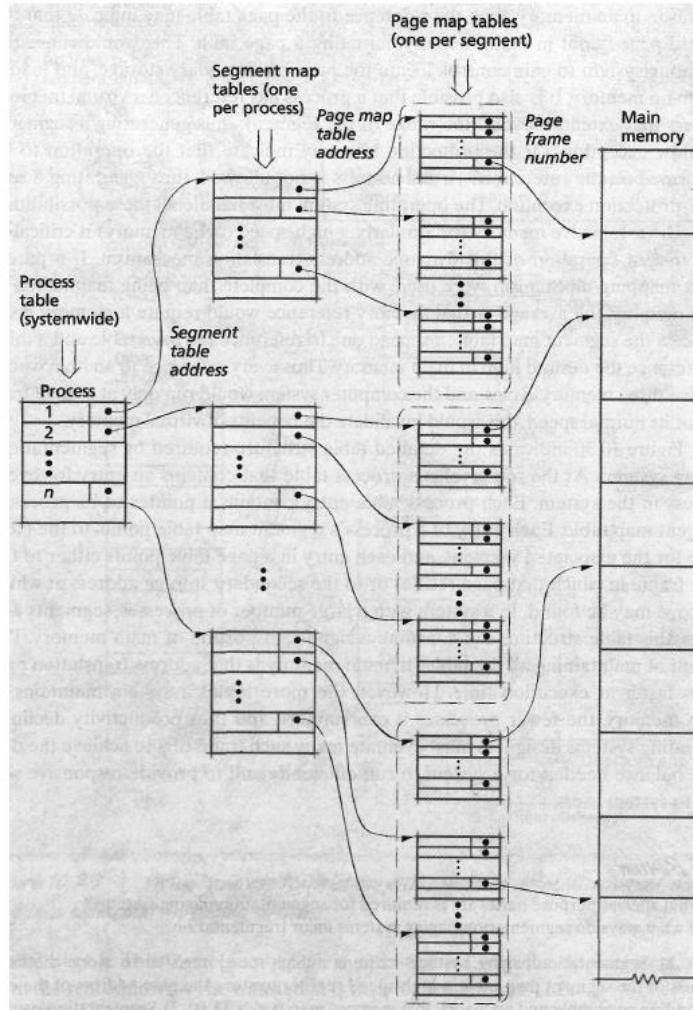


Figure 10.30 | Table structure for a segmentation/paging system.

Al livello più alto c'è una **tabella dei processi** che contiene una entry per ogni processo nel sistema. Ogni entry della tabella dei processi contiene un puntatore alla **tabella di mappatura dei segmenti del processo**. Ogni entry della tabella di mappatura dei segmenti di un processo punta alla **page map table per il segmento associato** e ogni entry in una page table punta al **page frame** in cui si trova la pagina o all'indirizzo di memoria secondario in cui è possibile trovare la pagina. Si cerca quindi di scomporre il problema iniziale organizzandolo in sotto problemi già studiati e più facili da implementare singolarmente.

# Lezione 28

---

## Gestione della memoria virtuale

Le strategie per la gestione della memoria virtuale si dividono in due categorie:

- **Strategie di fetch:** le **strategie di fetch** della memoria virtuale determinano quando una pagina o un segmento dovrebbero essere spostati dallo storage secondario alla memoria principale. Le strategie di **fetch su richiesta (demand fetch)** aspettano che un processo referenzi una pagina o un segmento prima di caricare quest'ultimi nella memoria principale. Le strategie di **anticipatory fetch** utilizzando euristiche per prevedere quali pagine o segmenti un processo referenzierà a breve: se la probabilità di referenziazione è alta e lo spazio è disponibile, allora il sistema carica la pagina o il segmento in memoria principale prima che il processo la referenzi esplicitamente, quindi migliorando le prestazioni quando avviene la referenziazione.
- **Strategie di sostituzione:** le **strategie di sostituzione** determinano quale pagina o segmento sostituire per fornire spazio ad una pagina o segmento in arrivo.

## Località

Il concetto di **località** è centrale alla maggior parte delle strategie di gestione della memoria. Il concetto di località è un concetto empirico, cioè derivato da esperienze, e nello specifico questo concetto ci dice che i processi tendono a localizzarsi in memoria secondo certi **pattern** (quindi non in maniera casuale).

Nei sistemi con paginazione, per esempio, si osserva che i processi tendono a favorire certi sottoinsiemi delle loro pagine, e che queste pagine tendono ad essere vicine le une con le altre in uno spazio di indirizzamento virtuale di un processo.

La località si manifestano sia nel tempo che nello spazio:

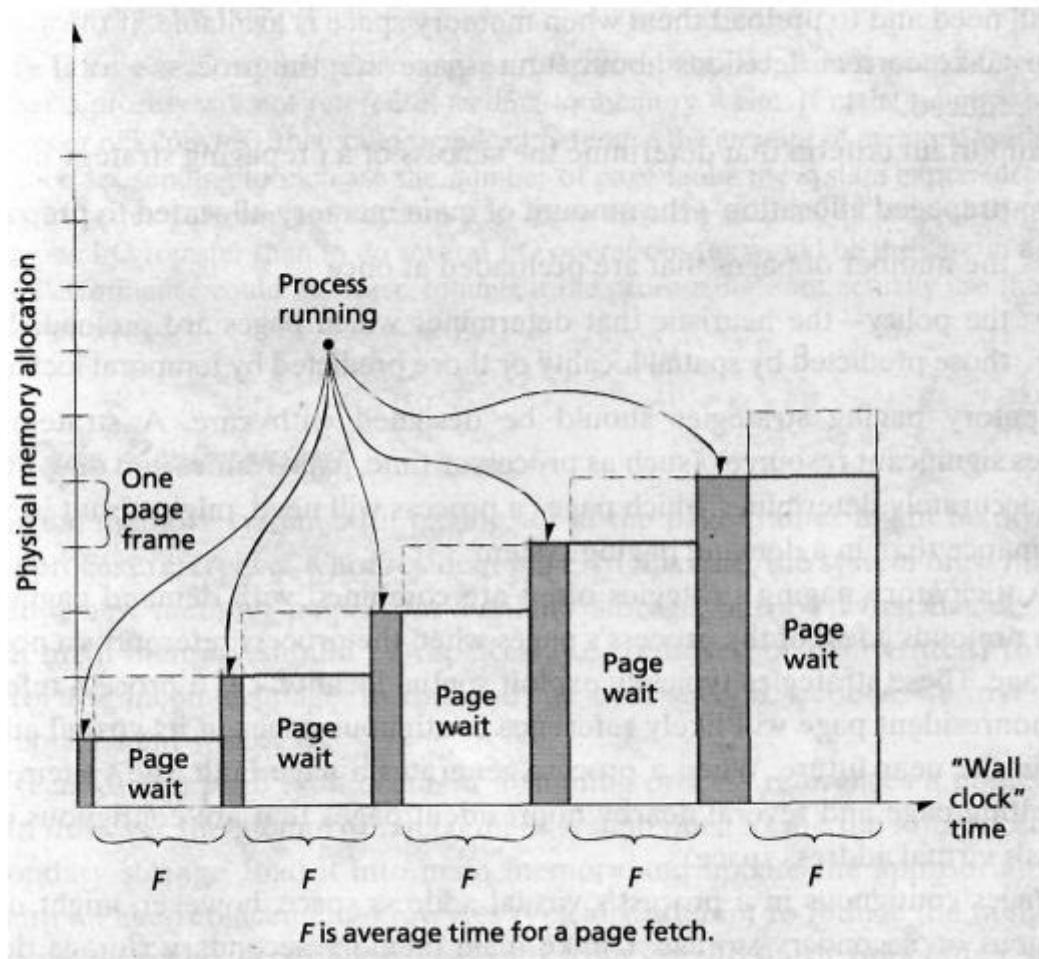
- **Località spaziale:** fa riferimento ad indirizzi vicini spazialmente. Un sistema favorisce pagine/segmenti vicini. Lo scorrimento degli array, l'esecuzione del codice in modo sequenziale e la tendenza di un programmare (o dei compilatori) di posizionare le definizioni di variabili correlate l'una vicino all'altra comportano tutte la località spaziale: tendono tutti a generare riferimenti di memoria raggruppati.
- **Località temporale:** fa riferimento alla vicinanza nel tempo. I cicli, le funzioni, le procedure e le variabili usate per il conteggio ed il totale comportano tutte la località temporale. In questi casi, è più probabile che le locazioni di memoria recentemente referenziate vengano referenziato di nuovo in un futuro prossimo.

## Demand Paging

Provando ad utilizzare i concetti di località spaziale e temporale sono nate tecniche che permettono al sistema di effettuare previsioni. Un modo di caricare le pagine senza fare previsioni è la **demand paging**. Quando un processo viene eseguito per la prima volta, il sistema carica nella memoria principale la pagina che contiene le sue prime istruzioni. Da quel momento in poi, il sistema carica una pagina dallo storage secondario nella memoria principale soltanto quando il processo referenzia esplicitamente la pagina.

Il paging on demand fa sì che la pagina venga caricata soltanto quando il processo lo richiede esplicitamente. Via via che questi trasferimenti vengono realizzati il processo rimane in attesa in cui il processo non può essere eseguito. Per valutare questa tecnica si guarda il valore di space-time product in cui si tiene in considerazione sia il tempo impiegato nel caricare le pagine, sia lo spazio usato per allocarle.

Vediamo un esempio:



**Figure 11.1 | Space-time product under demand paging.**

Nell'asse X abbiamo il tempo mentre nell'asse Y abbiamo lo spazio occupato. Inizialmente un processo viene eseguito (zona grigia) dopodiché il processo richiede un'altra pagina che non si trova in main memory quindi il processo dovrà attendere il caricamento (zona bianca). Successivamente si ripete l'operazione con un'altra pagina, e nuovamente partirà una attesa, e così via. Le aree grigie prendono il nome di **aree spazio-tempo in cui il processo è attivo**, mentre quelle bianche si chiamano **aree spazio-tempo in cui il processo è sospeso**. Il nostro obiettivo è diminuire il più possibile le aree bianche.

Affinché si ottenga un buon demand paging, allora il tempo di attesa deve essere il più piccolo possibile, quindi si usano delle tecniche di previsione per provare a ridurre questo tempo di attesa.

### Paging anticipato

Come abbiamo dimostrato nel paragrafo precedente, un modo per ridurre i tempi di attesa è quello di evitare i ritardi in un sistema che usa demand paging. Con l'**anticipatory paging** il sistema operativo prova a prevedere le pagine di cui un processo avrà bisogno e le carica preventivamente quando è disponibile spazio nella memoria principale. Se il sistema è in grado

di prendere delle decisioni corrette riguardo al futuro utilizzo delle pagine, il tempo di esecuzione totale del processo può essere ridotto.

In questa maniera si aumenta l'overhead del sistema ma potenzialmente può diminuire il tempo di attesa. In più abbiamo altri criteri importanti da tenere a mente nella strategia di paging anticipato:

- Allocazione prepaged: La quantità di memoria principale allocata al prepaging. Dobbiamo capire quanto deve essere grande questa area di memoria principale.
- Il numero di pagine che sono precaricate in una volta.
- L'euristica che determina quali pagine sono precaricate (ad esempio quelle predette usando il concetto di località spaziale o temporale).

Questi criteri, se scelti bene, possono determinare una scelta vincente per il paging.

### Sostituzione delle pagine

La sostituzione delle pagine ha a che fare con la scelta di quali pagine andare a rimuovere dalla memoria per aggiungerne delle altre. Un problema che si crea in generale con i sistemi di paging visti precedentemente è che prima o poi dovremo andare ad aggiungere una nuova pagina in memoria principale. Il memory manager allora va a individuare in memoria secondaria dove si trova la pagina e conseguentemente la va a caricare in memoria principale, questo avviene guardando le page table (o segment table).

Se la cella di memoria in cui andiamo a caricare la pagina è occupata allora dovremo spostare il contenuto in memoria secondaria facendo attenzione che la pagina non sia stata modificata. Se la pagina non ha subito modifiche si può tranquillamente eliminare, viceversa non posso solo eliminarla, ma devo andare prima ad aggiornare l'immagine di quella pagina in memoria secondaria. Per fare ciò si utilizza un bit di nome **dirty bit** che ci indica se una pagina è stata modificata o meno. Nel caso in cui sia stata modificata il sistema dovrà prima ricopiarla in memoria secondaria.

Inoltre, quando si valuta una strategia di **page-replacement**, la si confronta spesso con la cosiddetta strategia di page-replacement ottimale, che afferma che, per ottenere ottime prestazioni si dovrebbe rimpiazzare la pagina che sarà referenziata nuovamente, se mai, nel futuro più lontano possibile. Dobbiamo quindi stimare quali saranno le pagine che ci serviranno a breve. Di queste strategie di replacement ne esistono tante, e ognuna di esse ha pro e contro. Vediamole nel dettaglio.

### Strategie di page-replacement

Ogni strategia è caratterizzata dall'euristica che usa per selezionare una pagina da sostituire e dell'overhead che l'attuazione di questa euristica apporta.

#### Page replacement casuale (RAND)

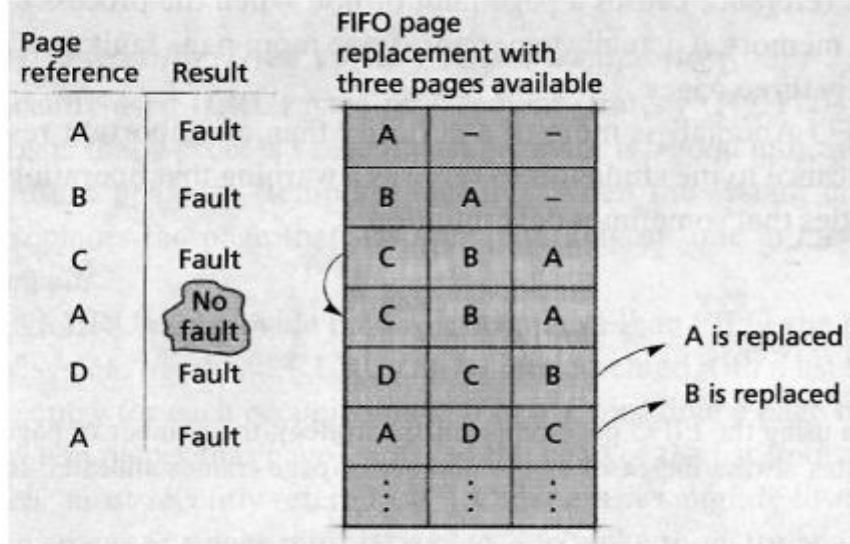
Il **Page replacement casuale (RAND)** è una strategia di page-replacement con basso overhead e facile da implementare. Con questa strategia, ogni pagina in memoria principale ha una eguale probabilità di essere selezionata per la sostituzione (*leggasi: sono equiprobabili*).

Il vantaggio è l'overhead bassissimo, lo svantaggio è che questa tecnica non tiene in conto i principi di località di cui abbiamo discusso prima. C'è quindi il rischio di sostituire delle pagine che ci serviranno nell'immediato futuro.

#### Page replacement FIFO

Strategia basata su una coda in cui si inseriscono le pagine in base all'istante di tempo in cui sono state caricate in memoria. Di conseguenza, viene sostituita la pagina che è stata più a lungo nel sistema. Il problema è che non tutte le pagine che sono da più tempo nel sistema sono inutili o rimuovibili. Anche in questa strategia non si considerano i concetti di località, bensì si considera solo l'età delle pagine. Il vantaggio è la semplicità di implementazione e il basso overhead.

Vediamola graficamente:



**Figure 11.2 | First-in-first-out (FIFO) page replacement.**

Abbiamo un processo che nell'ordine va a referenziare le pagine nella colonna all'estrema sinistra. La coda ha dimensione 3. All'inizio il processo farà riferimento alla pagina A che non sarà in memoria, si genera quindi un page fault e la pagina viene caricata in memoria principale. Si continua così per le successive pagine B e C che genereranno anche esse dei page fault. Appena verrà richiesta la pagina A, non si genererà più un page fault, perché la pagina si trova ancora in memoria (nell'ultimo posto della coda). Appena viene chiesta la pagina D, viene eliminata la pagina A, e viene caricata la pagina D. Il problema è che subito dopo però verrà richiesta A nuovamente e verrà generato un fault. Notiamo come, per sole 6 pagine, si sono generati 5 page fault.

#### Anomalia FIFO

Se andiamo ad aumentare la lunghezza della coda (quindi di page frame in memoria), in linea teorica potremmo pensare che, si dovrebbe ridurre il numero di page fault. Maggiore è il numero di page frame che alloco, minore è il numero di page fault. Tuttavia questa idea non viene correttamente riscontrata nella realtà, in quanto si crea un'anomalia dovuta ai pattern (dovuti al referenziamento delle pagine) che prende il nome di **Anomalia di Belady**. Capiamolo meglio con un esempio dove andiamo a confrontare una coda di 3 elementi con una di 4 elementi:

Page reference	Result	FIFO page replacement with three pages available			FIFO page replacement with four pages available		
		A	-	-	Fault	A	-
B	Fault	B	A	-	Fault	B	A
C	Fault	C	B	A	Fault	C	B
D	Fault	D	C	B	Fault	D	C
A	Fault	A	D	C	No fault	D	B
B	Fault	B	A	D	No fault	D	A
E	Fault	E	B	A	Fault	E	D
A	No fault	E	B	A	Fault	A	D
B	No fault	E	B	A	Fault	B	E
C	Fault	C	E	B	Fault	C	B
D	Fault	D	C	E	Fault	D	C
E	No fault	D	C	E	Fault	E	D

Three "no faults"

Two "no faults"

**Figure 11.3 | FIFO anomaly-page faults can increase with page frame allocation.**

Ho aumentato il numero di page frame, vediamo cosa succede se replichiamo due sequenze identiche. Notiamo che nella coda di 3 elementi si verificano 9 page fault, contro i 10 page fault della coda di 4 elementi. Quindi nel caso delle code FIFO pur aumentando le dimensioni di page frame non si ottiene una riduzione dei page fault, bensì il contrario.

### Page replacement Least-Recently-Used (LRU)

Invece di usare una coda FIFO si utilizza una tecnica che tiene conto del fatto che alcune pagine siano stati utilizzate più di recente rispetto che ad altre. La strategia di **page-replacement LRU** si affida sull'euristica della **località temporale** che afferma che un recente comportamento passato di un processo è un buon indicatore del comportamento nel prossimo futuro (**località temporale**). Quando il sistema deve sostituire una pagina, LRU sostituisce la pagina che ha è stata più a lungo in memoria principale senza essere referenziata.

Quindi si vanno a sostituire sempre le pagine meno utilizzate (e non le ultime in coda). In questa maniera si aumenta l'overhead in quanto ci servono dei contatori per tenere conto della frequenza di referenziamento delle pagine. Per ridurre la possibilità di page fault, in questo caso, si può mantenere una lista che mantenga tante pagine.

Vediamo un esempio:

Page reference	Result	LRU page replacement with three pages available		
A	Fault	A	-	-
B	Fault	B	A	-
C	Fault	C	B	A
B	No fault	B	C	A
B	No fault	B	C	A
A	No fault	A	B	C
D	Fault	D	A	B
A	No fault	A	D	B
B	No fault	B	A	D
F	Fault	F	B	A
B	No fault	B	F	A

**Figure 11.4 | Least-recently-used (LRU) page-replacement strategy.**

Si caricano le pagine A, B e C. Nel momento in cui si riutilizza B (riga 4) la pagina B passa in coda in quanto aumenta la sua frequenza di uso. Poi utilizzo A che passa in coda, come era accaduto con B. Appena si richiede la pagina B, per eliminare un elemento, si prende l'elemento nella testa della lista (ovvero quello usata meno recentemente), in questo caso C e così via.

Questa strategia tiene conto della località temporale, a differenza della coda FIFO, si comporta quindi meglio della FIFO, seppur richieda un overhead maggiore.

#### Page replacement Last-Frequently-Used (LFU)

La strategia di **page-replacement LFU** fa basare le decisioni sulla sostituzione su quanto intensivamente ciascuna pagina viene utilizzata. Con LFU, il sistema sostituisce la pagina che è stata utilizzata meno frequentemente o referenziata meno intensamente. Questa strategia è basata sull'euristica che afferma che una pagina che non viene referenziata intensivamente è poco probabile che venga referenziata in futuro. LFU può essere implementato utilizzando un contatore che è aggiornato ogni volta che la sua pagina corrispondente viene referenziata, ma questo causa un overhead significativo.

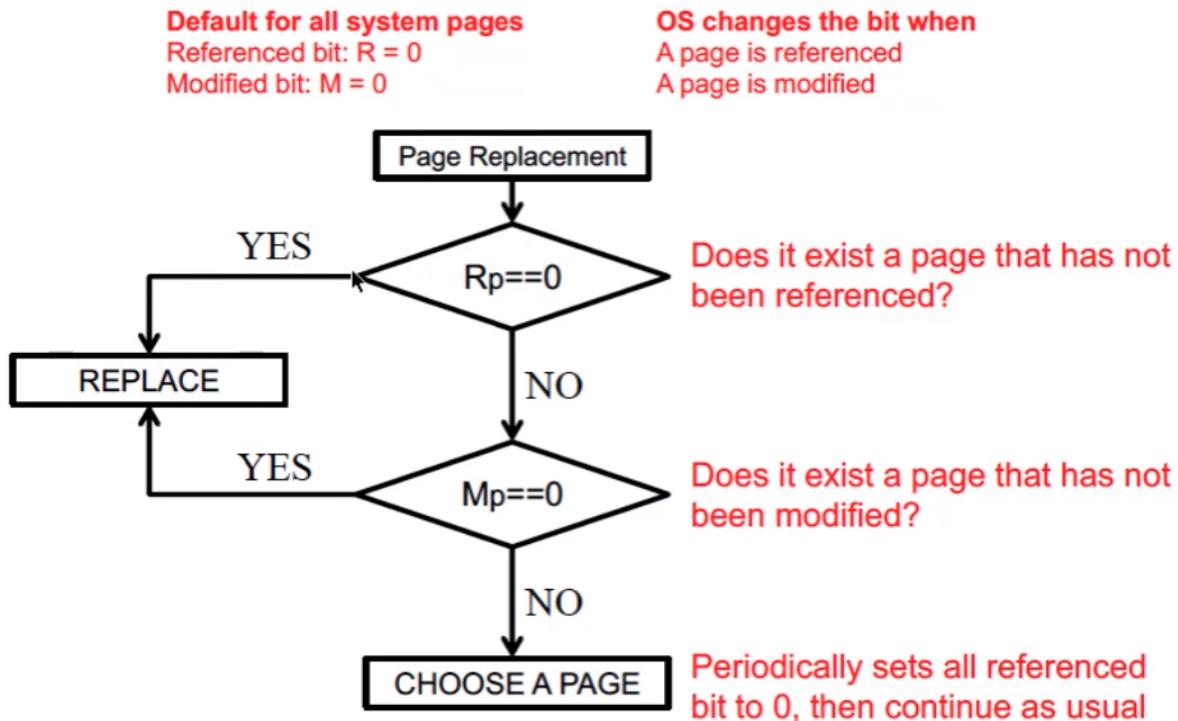
La strategia di page-replacement LFU, inoltre, può facilmente selezionare pagine non corrette per la sostituzione. Per esempio, la pagina usata meno frequentemente potrebbe essere la pagina portata in memoria principale più recentemente. Questa pagina è stata usata solo una volta. In questo caso, il meccanismo di page-replacement sostituisce la nuova pagina, quando in effetti la pagina potrebbe avere molta probabilità di essere usata immediatamente.

#### Page replacement Not-Used-Recently (NUR)

Uno schema popolare per approssimare LRU con un basso overhead è la strategia **NUR**. NUR è basato sull'idea che una pagina che non è stata utilizzata di recente ha poche probabilità di essere utilizzata in un futuro prossimo. La strategia NUR è implementata utilizzando i due seguenti bit hardware per ogni entry nella page table:

- **Referenced bit:** impostato a 0 se la pagina non è stata referenziato ed a 1 altrimenti.
- **Modified bit:** impostato a 0 se la pagina non è stata modificata ed a 1 altrimenti.

La strategia su cui si basa questa tecnica è la seguente:



Inizialmente, il sistema imposta i referenced bit di tutte le pagine a 0. Quando un processo referenzia una pagina, il sistema imposta il referenced bit di quella pagina ad 1. I modified bit su tutte le pagine sono inizialmente impostati a 0. Quando una pagina viene modificata, il sistema imposta il modified bit della pagina a 1. Quando il sistema deve sostituire una pagina, NUR prima prova a trovare una pagina che non è stata ancora referenziata. Se nessuna pagina soddisfa questo requisito, il sistema deve sostituire una pagina referenziata. In questo caso, NUR controlla il modified bit per determinare se la pagina è stata modificata. Se la pagina non è stata modificata, il sistema la seleziona per sostituirla. Altrimenti, il sistema deve sostituire una pagina che è stata modificata. Ricordiamo che sostituire una pagina modificata causa un ritardo sostanziale perché la pagina viene archiviata nello storage secondario per preservare i propri contenuti.

Una tecnica che è stata largamente implementata per evitare questo problema consiste nell'effettuare una sorta di refresh impostando periodicamente tutti i referenced bit a 0, e dopo continuare normalmente.

Se tutte le pagine sono state modificate e referenziate si sceglie una pagina randomica. Possiamo dividere le pagine in 4 gruppi:

<i>Group</i>	<i>Referenced</i>	<i>Modified</i>	<i>Description</i>
Group 1	<b>0</b>	<b>0</b>	Best choice to replace
Group 2	<b>0</b>	<b>1</b>	[Seems unrealistic]
Group 3	<b>1</b>	<b>0</b>	
Group 4	<b>1</b>	<b>1</b>	Worst choice to replace

**Figure 11.5 | Page types under NUR**

Le pagine con i gruppi con i numeri più bassi dovrebbe essere sostituiti per prime, e quelle con i gruppi con i numeri più alti per ultime. Le pagine dentro un gruppo sono selezionate casualmente per la sostituzione. Il gruppo due sembra rappresentare una situazione irrealistica: cioè, pagine che sono state modificata ma non referenziate. Questo avviene a causa del reset periodico dei referenced bit (ma non dei modified bit).

#### Variante FIFO: seconda chance e sostituzione a orologio

Il problema di FIFO è che può sostituire una pagina usata spesso che risiede però in memoria da tanto tempo. Si può evitare ciò utilizzando un bit di referenza (come quello del capitolo precedente) per ciascuna pagina e sostituendo quelle pagine con il suddetto bit posto a 0. Questa variante di FIFO prende il nome di **strategia di sostituzione con seconda chance**, e nello specifico funziona nella seguente maniera: si esamina il bit di referenza della pagina più vecchia, se è pari a 0, si seleziona quella pagina per la sostituzione. Se il bit è 1 si azzera il bit e si sposta la pagina alla fine della coda FIFO. Quindi la pagina verrà considerata come una "nuova arrivata" e man mano scorrerà la coda.

Ovviamente, anche in questo caso, una pagina modificata dovrà essere salvata nel dispositivo secondario di memorizzazione prima di essere sostituita.

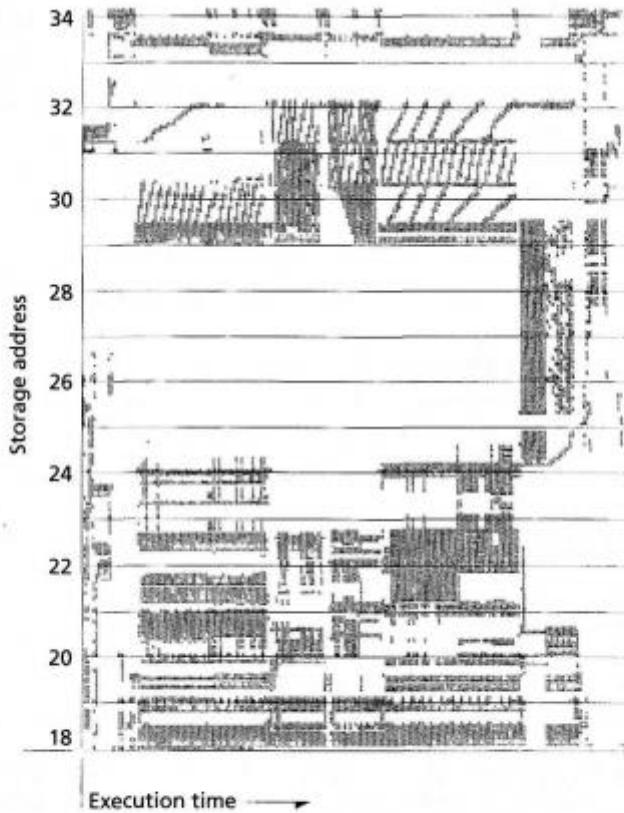
La **strategia di sostituzione a orologio** funziona esattamente come la strategia della seconda chance, con la differenza che la lista che utilizza non è una coda, bensì una lista circolare. Ogni volta che c'è un page fault un puntatore scorre la lista come le lancette di un orologio. Quando il bit di referenza di una pagina viene posto a 0, il puntatore viene spostato verso l'elemento successivo della lista. La seguente tecnica posiziona i nuovi arrivi nella prima pagina che incontra con il bit di referenza nullo.

#### Working set

Un programma può essere eseguito efficientemente anche se soltanto un piccolo sottoinsieme di pagine risiede nella memoria principale in un dato istante. Vogliamo dunque garantire che questo sottoinsieme di pagine sia sempre caricato in main memory. Se non riuscissimo a individuare questo sottoinsieme di pagine di fatto il processo andrebbe a rispondere alle precedenti tecniche di replacement (con conseguenti page fault).

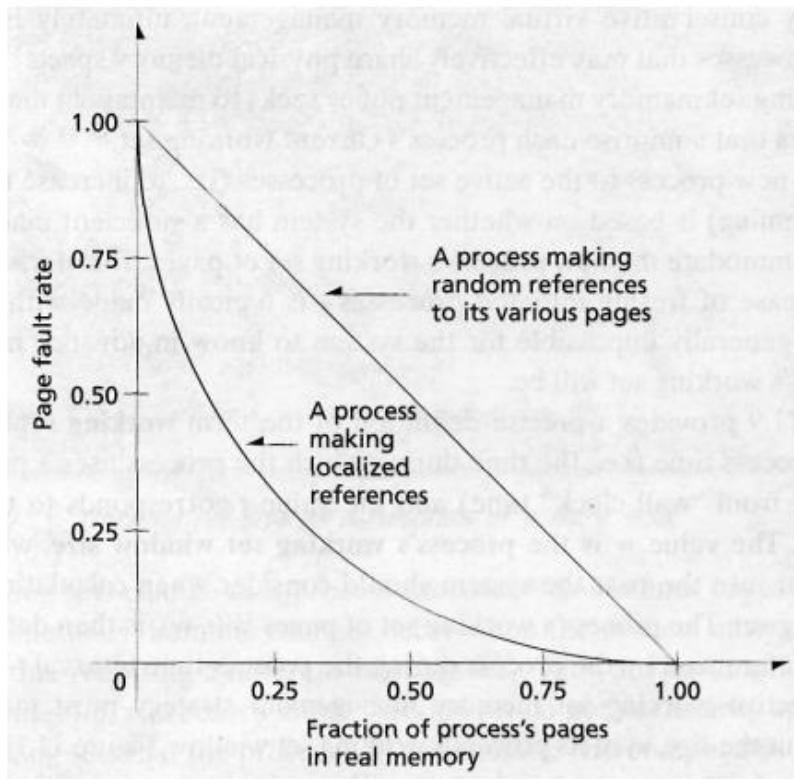
Questo insieme di pagine preferite si chiama **working set**, che riflette il principio di località, ed esistono delle euristiche per calcolarlo.

Vediamo di seguito un grafico di un pattern di referenziazione di memoria di un processo attraverso le sue pagine. La figura mostra in modo chiaro come questo processo tenda a favorire un sottoinsieme di pagine durante determinati intervalli di esecuzione:



**Figure 11.7 | Storage reference pattern exhibiting locality.** (Reprinted by permission from IBM Systems Journal. © 1971 by International Business Machines Corporation.)

Nel seguente grafico vediamo nell'asse x il tempo di esecuzione e nell'asse delle y degli indirizzi di memoria. Le zone più scure indicano la presenza di pagine, in un certo istante, in una certa locazione di memoria. Notiamo che ci sono una serie di macchie scure la cui configurazione si ripete nel tempo. Ci sono infatti degli indirizzi che sono sempre (o quasi) referenziati e altri che non lo sono quasi mai invece. La località spaziale si vede dalla posizione delle macchie scure rispetto all'asse y, viceversa quella temporale rispetto all'asse x.

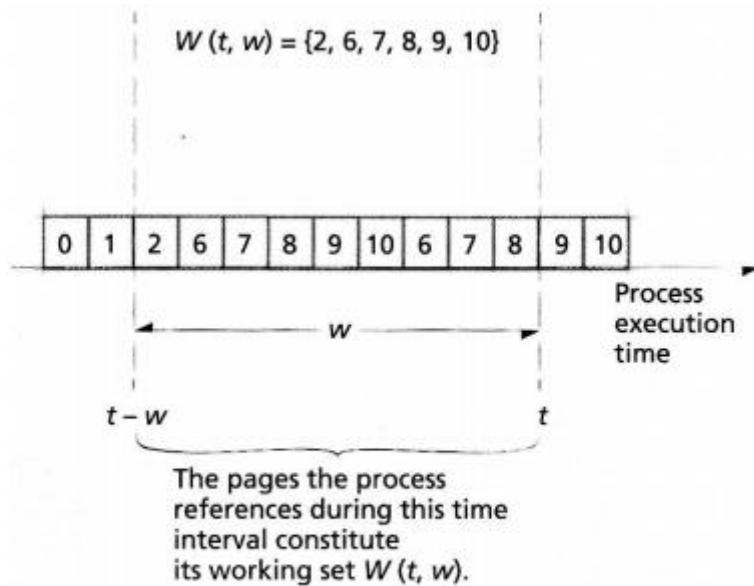


**Figure 11.8 | Dependency of page fault rate on amount of memory for a process's**

Quello che si vede da questo grafico è ciò che avviene via via che aumenta il numero di pagine di un certo processo che ospito in maniera principale. Per  $x=1$  tutte le pagine del processo sono ospitate in memoria principale, e così via. Man mano che il numero di page frame disponibile ai processi decresce, c'è un intervallo su cui non influenza in modo significativo la frequenza di page fault. Se tutte le pagine fossero caricate in memoria avrei 0 page fault, se nessuna pagina fosse caricata in memoria allora avrei sicuramente un page fault, ma se cariciamo esattamente la metà delle pagine di un processo, allora avrei la metà dei page fault. Visto che invece in uno scenario reale abbiamo un principio di località allora non avremo una retta, bensì una curva come quella che vediamo nel grafico. Allora in questo caso se caricassi il 10% delle pagine del processo, otterrei un page fault abbastanza alto, tuttavia, in corrispondenza del 50% delle pagine caricate, avremo un page fault molto basso.

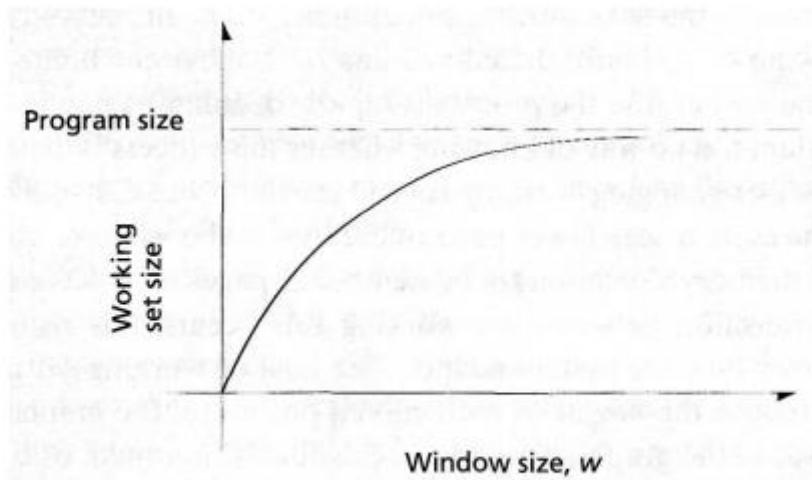
Il motivo per cui si ottiene un page fault molto alto è dovuto al fatto che non stiamo caricando abbastanza pagine da caratterizzare il working set del processo.

Quindi il working set è definito come un insieme di pagine che servono ad un determinato processo all'interno di un certo arco di tempo. Nello specifico il **working set** del processo  $W(t,w)$ , è definito come l'insieme di pagine referenziate dal processo durante l'intervallo di tempo del processo che si estende da  $t-w$  a  $t$ . Graficamente:



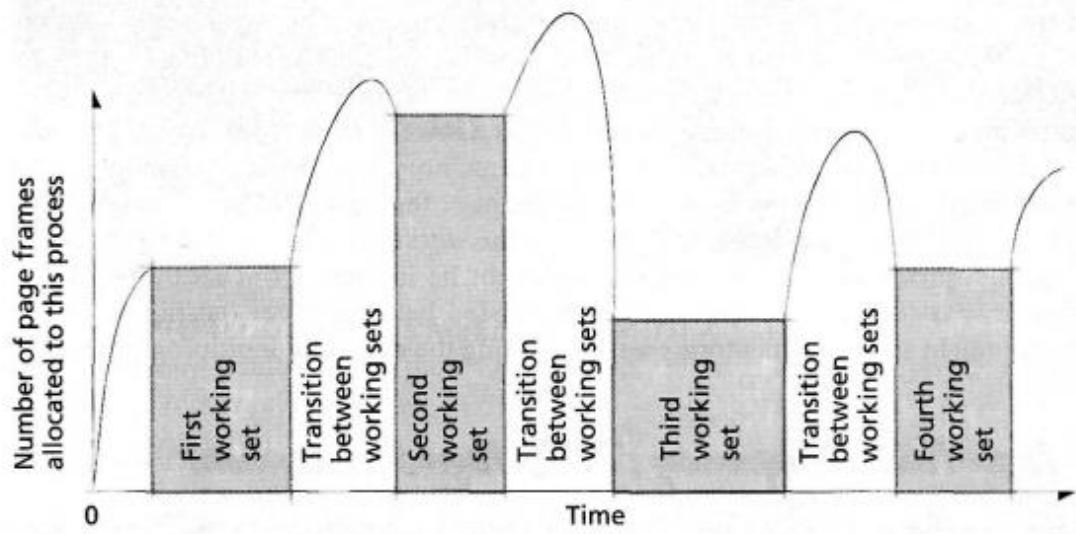
**Figure 11.9 | Definition of a process's working set of pages.**

In generale, se considerassimo un unico working set, via via che aumentiamo il valore di  $w$ , andiamo a considerare tutte le possibili pagine del processo, graficamente:



**Figure 11.10 | Working set size as a function of window size.**

I working set cambiano man mano che il processo viene eseguito. Qualche volta pagine vengono aggiunte o rimosse. A volte avvengono cambiamenti drastici quando il processo entra in una nuova fase (cioè, l'esecuzione richiede un working set differente). Quindi, qualsiasi assunzione sulla dimensione ed il contenuto del working set iniziale di un processo non si applica necessariamente ai working set successivi che il processo accumula. Immaginandolo graficamente come evoluzione temporale:



**Figure 11.11 | Main memory allocation under working set memory management.**

Avviene che inizialmente il processo inizia a richiedere pagine, dopo un po' il numero di page frame costituirà il primo working set che non varia per una certa quantità di tempo. Dopo un certo periodo il processo comincia a chiedere nuovamente pagine creando così il secondo working set. Successivamente il processo inizierà a referenziare alcune pagine e a scartarne altre ottenendo un terzo working set, e così via in maniera ciclica.

L'obiettivo diventa quindi quello di trovare una strategia che renda le curve di transizione da un working set all'altro, più basse possibili, in quanto queste curve rappresentano la richiesta delle pagine (quindi overhead del sistema).

Per fare ciò conviene utilizzare una strategia di fetching anticipato per sapere prima quali pagine andare a caricare.

### Page replacement Page-Fault-Frequency (PFF)

La frequenza di page fault viene utilizzata come indice della efficienza del sistema di paging. Nello specifico possiamo avere due casi limite:

- Un processo ha costantemente dei page fault, il che significa che il suo working set non è stato ancora caricato.
- Un processo non ha mai page fault, il che significa che la strategia è ottima, ma può anche significare che si è sovrastimata la dimensione del working set, aggiungendo in quest'ultimo pagine superflue che non servono effettivamente al processo.

L'idea di utilizzare il page fault come valore indicativo sulla efficienza del sistema è la seguente: andiamo a determinare dinamicamente quali sono le pagine che si devono trovare in memoria utilizzando come indicatore la frequenza di fault di queste ultime. L'algoritmo di **page-fault-frequency (PFF)** regola l'insieme di pagine di un processo residenti in memoria principale (cioè, quelle pagine che sono correntemente in memoria), basandosi sulla frequenza a cui un processo genera page fault. Alternativamente, PFF potrebbe regolare il suddetto insieme basandosi sul tempo tra page fault, chiamato **tempo interfault**.

Con PFF, quando un processo fa una richiesta che genera un page fault, la strategia calcola il tempo dall'ultimo page fault. Se questo tempo è più grande di un valore superiore di soglia, la pagina in arrivo diventa un membro dell'insieme di pagine residenti in memoria del processo.

### Page release

I processi, via via che continuano la loro esecuzione, possono non avere più bisogno di certe pagine caricate in memoria. Quindi si può fare in modo che un processo rilasci in maniera volontaria un page frame che non utilizza più. Questo elimina il periodo di attesa tra un working set e l'altro.

### Dimensioni delle pagine

Un'importante caratteristica dei sistemi con paginazione a memoria virtuale è la dimensione delle pagine de dei page frame che il sistema supporta.

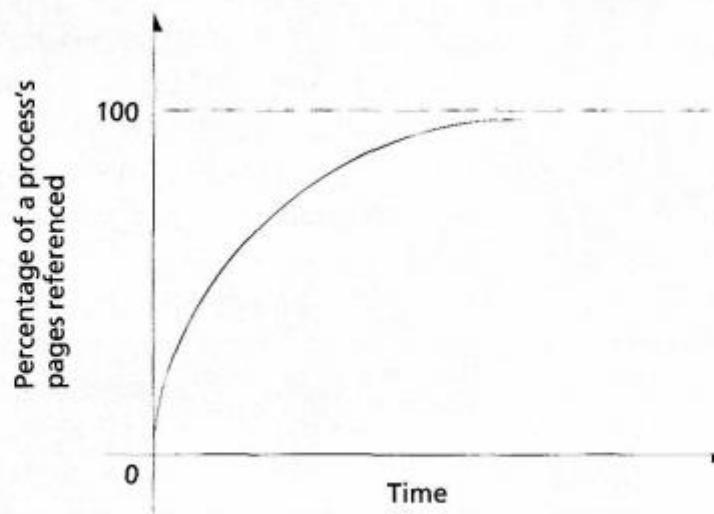
I sistemi basati sulla paginazione soffrono, come sappiamo, di frammentazione interna, ecco perché a volte si prediligono pagine di dimensione piccola. Inoltre avere pagine piccole aumenta la probabilità che l'insieme di pagine del working set siano di dimensione limitata, e quindi, si possano caricare in memoria.

Se invece usiamo pagine abbiamo meno entry nella page table, meno pagine, ma più frammentazione interna.

Esiste anche la possibilità di combinare segmentazione e paginazione che riducono notevolmente la frammentazione esterna.

### Comportamento programmi in presenza di paginazione

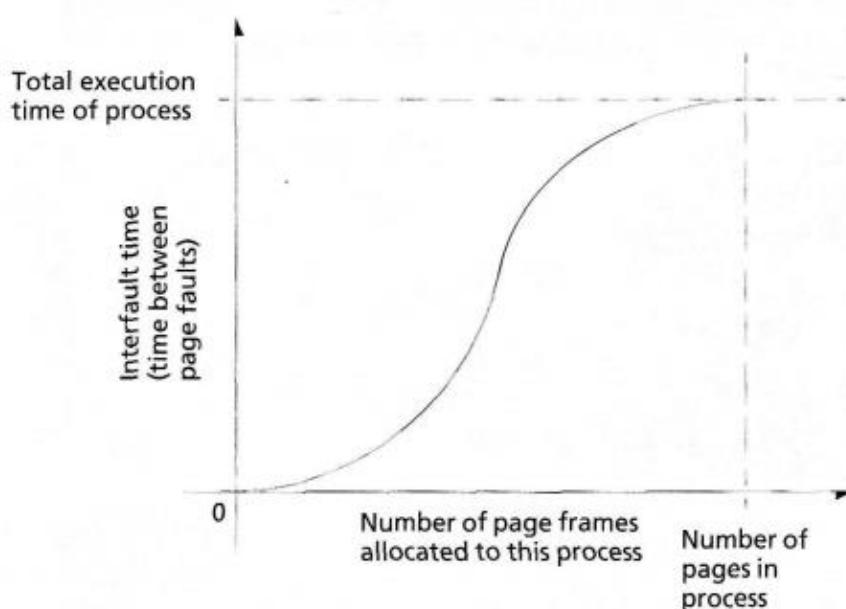
Studiamo ora due figure che ci permettono di capire come si comportano i processi in funzione delle pagine referenziate.



**Figure 11.14 | Percentage of a process's pages referenced with time.**

La figura mostra la percentuale di pagina che sono state referenziate di un ipotetico processo, cominciando dal momento in cui il processo comincia la sua esecuzione. La pendenza pronunciata iniziale indica che un processo tende a referenziare una porzione significativa di pagine immediatamente dopo l'inizio dell'esecuzione. Con il tempo, la pendenza diminuisce, ed il grafico tende asintoticamente al cento percento. Di certo alcuni processi referenziano il 100 percento delle loro pagine, ma il grafico ha la funzione di mostrare che molti processi potrebbero essere eseguiti a lungo senza fare ciò.

La figura di seguito mostra come il tempo interfault medio (cioè, il tempo tra i page fault) varia man mano che il numero di page frame allocati al processo aumenta. Il grafico è non decrescente ciò significa che più page frame ha il processo, più è lungo il tempo tra i page fault.



**Figure 11.15** | Dependency of interfault time on the number of page frames allocated to a process.

# Lezione 29

---

## Files

I **file** sono collezioni di dati che il sistema gestisce in modo unitario. Vengono memorizzati nelle unità di storage secondario ma nel momento in cui ci serve accedere ad uno specifico file allora esso andrà caricato in memoria principale.

I diversi OS hanno varie tecniche di gestione dei file, a partire dalla organizzazione dei file nel disco, fino ad arrivare all'interfaccia di visualizzazione dei file.

Per capire cosa contiene un file dobbiamo fare riferimento alla gerarchia dei dati. Le informazioni vengono archiviate nei computer in base a una gerarchia di dati. Il livello più basso della gerarchia di dati è composto da bit. Il livello successivo nella gerarchia dei dati è costituito da pattern a lunghezza fissa di bit come byte, caratteri e parole. Quando si fa riferimento alla memoria, un byte è tipicamente di 8 bit. Una parola è il numero di bit su cui un processore può operare contemporaneamente.

I bit si possono quindi raggruppare in parole, ma anche in byte e caratteri. Gruppi di caratteri rappresentano dei **campi**, un insieme di campi rappresenta un **record**, e un **file** è visto come un insieme di record collegati.

Tutti i file che si trovano nel sistema devono essere gestiti mediante una entità specifica: il **file system**. Per ogni file system ci sono delle specifiche regole per l'accesso e la manipolazione dei file.

Il termine **volume** indica un'unità di archiviazione dei dati che può contenere più file.

Sui file si possono eseguire più operazioni:

- Apertura: aprire il file per predisporre ad operazioni successive.
- Chiusura: chiudere il file per evitare modifiche su quel file.
- Creazione
- Distruzione
- Copiatura
- Rinominare
- Listare: corrisponde alla operazione di lettura.

Al di là delle operazioni di base, tra le caratteristiche che ci sono nella gestione dei file, vi sono alcune informazioni:

- Dimensione: la quantità di dati memorizzati nel file.
- Posizione: la posizione del file (cioè il percorso a quel file).
- Accessibilità: restrizioni imposte all'accesso ai dati del file.
- Tipo: come vengono utilizzati i dati del file. Ad esempio, un file eseguibile contiene istruzioni macchina per un processo. Un file di dati può specificare l'applicazione utilizzata per accedere ai suoi dati.
- Volatilità: la frequenza con cui inserimenti e cancellazioni vengono apportati a un file.
- Attività: definiscono lo storico delle varie operazioni eseguite su un certo file.

## File System

Un **file system** organizza i file e gestisce l'accesso ai dati. I file system sono responsabili per:

- **Gestione dei file:** fornisce i meccanismi per archiviare i file, fare riferimento ad essi, condividerli e proteggerli. Uno dei problemi legati alla gestione dei file è infatti quello legato alla sicurezza, che dipende in gran parte dal file system che stiamo utilizzando.
- **Gestione memoria ausiliaria:** alloca lo spazio per i file su dispositivi di archiviazione secondaria o terziaria.
- **Meccanismi di integrità dei file:** garantiscono che le informazioni memorizzate in un file non siano corrotte e i dati siano integri. Quando l'integrità del file è assicurata, i file contengono solo le informazioni che devono avere.
- **Metodi di accesso:** modalità di accesso ai dati memorizzati. Questo si fa controllando i vari permessi di accesso dei file.

Il file system, inoltre, deve creare un livello di astrazione che aiuti gli utenti ad accedere al file in modo indipendente dal dispositivo. Per questo motivo il file system il file system ci offre una interfaccia tramite la quale possiamo associare ad ogni file un **nome simbolico** (esempio: *myfile.txt*), anche se a basso livello un file corrisponde a blocchi allocati dei dispositivi di memoria secondaria. Questa operazione deve essere indipendente rispetto al device, nel senso che se prendo un file e lo copio su un altro disco, ciò che io sposto apparentemente è il nome simbolico che rappresenta il file, ma ciò che il sistema deve copiare è il contenuto dei blocchi di memoria di quel file.

Per evitare perdite accidentali o distruzione dannosa delle informazioni, i file system dovrebbero anche fornire funzionalità di **backup** che facilitino la creazione di copie ridondanti di dati e funzionalità di ripristino che consentano agli utenti di ripristinare i dati persi o danneggiati.

Alcuni file system offrono inoltre anche degli **strumenti di cifratura**, in questo caso per accedere ai contenuti del disco, vi sarà bisogno di una chiave di cifratura. In questi sistemi si va solitamente a cifrare l'intero disco, ciò porterà un notevole overhead, ma allo stesso tempo si garantirà sicurezza nei dati.

## Directory

Per organizzare e individuare rapidamente i file, i file system utilizzano le **directory**, che contengono i nomi e le posizioni degli altri file nel file system. Le proprietà delle directory sono riportate qui di seguito:

<i>Directory Field</i>	<i>Description</i>
Name	Character string representing the file's name.
Location	Physical block or logical location of the file in the file system (i.e., a pathname).
Size	Number of bytes consumed by the file.
Type	Description of the file's purpose (e.g., data file or directory file).
Access time	Time the file was last accessed.
Modified time	Time the file was last modified.
Creation time	Time the file was created.

## File System a livello singolo

Nei primi sistemi non c'era l'esigenza di avere delle directory. L'organizzazione più semplice di un file system è una struttura di directory a **livello singolo**. In questa implementazione, il file system memorizza tutti i suoi file utilizzando una singola directory. In un file system a livello singolo, non è possibile che due file abbiano lo stesso nome.

Questo approccio è sicuramente semplice, ma non utile dal punto di vista della gestione delle risorse per via di alcune limitazioni (ad esempio non possono esserci file con lo stesso nome).

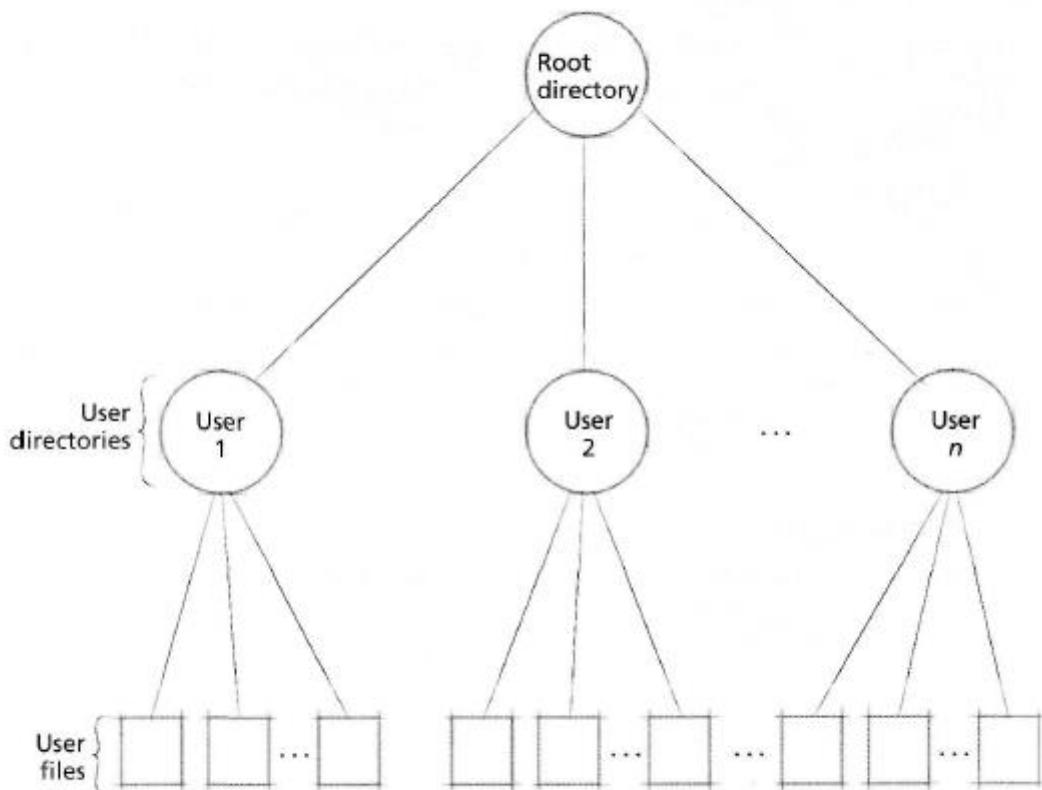
Inoltre, se abbiamo tutto su una directory, la ricerca di un file avverrà in maniera lineare scandendo tutto il disco e peggiorando le prestazioni e i tempi di attesa.

### File System gerarchico

Sono i file system attuali i quali sono composti da una gerarchia ad albero. Vi è un nodo principale (radice) da cui si diramano tutte le altre directory.

I nomi dei file devono essere univoci solo all'interno di una determinata directory utente. In tali file system gerarchicamente strutturati, ciascuna directory può contenere diverse sottodirectory ma non più di una directory padre. Il nome di un file viene solitamente formato come nome di percorso dalla directory root al file. Questo meccanismo prende il nome di **pathname**.

Vediamolo graficamente:



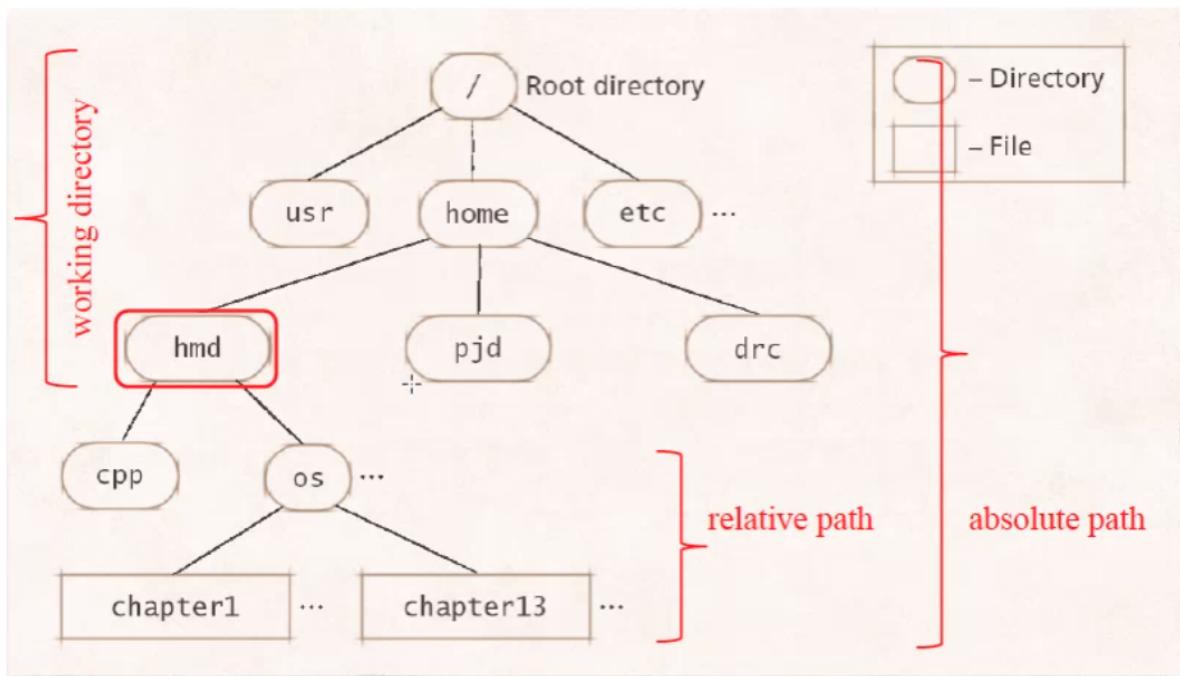
**Figure 13.2 | Two-level hierarchical file system.**

In questo tipo di file system ogni utente ha il suo spazio di indirizzamento specifico evitando le modifiche indesiderate dei file altrui.

### Working directory

Le working directory consentono di rappresentare i percorsi con dei **path relativi**. La working directory (rappresentata dalla directory entry ".") nei file system basati su Windows e UNIX consente agli utenti di specificare un percorso che non inizia nella directory principale. Quando un file system incontra un percorso relativo, forma il **percorso assoluto** (cioè, il percorso che inizia alla radice) concatenando la directory di lavoro e il percorso relativo. Il file system attraversa quindi la struttura della directory per individuare il file richiesto.

Vediamo quanto descritto graficamente:



## Link

I **link** sono degli elementi che consentono di fare riferimento a file o ad altre directory che si trovano in percorsi diversi rispetto alla posizione in cui il link è stato creato (un esempio classico sono i collegamenti dei programmi che utilizziamo giornalmente al pc).

Esistono due diverse tipologie di link:

- **Soft link:** sono dei collegamenti che puntano al nome di un altro file del sistema (cioè puntano alla posizione logica di un file).
- **Hard link:** link che non contiene il percorso del file, ma contiene un riferimento al numero di blocco in memoria fisica su cui quel file è memorizzato.

Vediamo un [esempio](#):

Directory	
Name	Location
foo	467
bar	843
:	:
foo_hard	467
foo_soft	./foo

Soft link

Hard link to foo

*Figure 13.4 | Links in a file system.*

Creando il soft link di foo vado a specificare il pathname del file di destinazione (vedi freccia che punta al file di nome foo). Quindi è un link al nome del file. Nel caso dell'hard link, si copia il riferimento al blocco di memoria e lo salvo nell'hard link. Vediamo le problematiche che possono nascere:

- Dato che il soft link punta alla posizione logica (il nome) del file, allora se un utente sposta il file di origine in una directory diversa o rinomina il file, eventuali soft link a tale file non sono più validi.

più validi.

- Dato che l'hard link memorizza il numero del blocco di memoria e non il nome simbolico, allora nel momento in cui il file di origine dovesse cambiare posizione in memoria allora l'hard link non sarebbe più valido. Questo problema può sorgere, ad esempio, nelle operazioni di deframmentazione.

Vediamo ora un esempio sulla BASH:

```
marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 8
-rw-r--r-- 1 marco marco 12 giu  2 14:57 file1.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2.txt
marco@marco-VirtualBox:~/Scrivania/link$ cat file1.txt
contenuto 1
marco@marco-VirtualBox:~/Scrivania/link$ cat file2.txt
contenuto 2
```

Per creare un link utilizzo il **comando ln**, che senza parametri va a creare un hard link. Questo comando quindi fa sì che nella tabella vista precedente si crei una nuova riga contenente l'hard link del file e il numero del blocco di memoria.

Utilizzando il **parametro -s** si crea invece un soft link:

```
marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 8
-rw-r--r-- 1 marco marco 12 giu  2 14:57 file1.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2.txt
marco@marco-VirtualBox:~/Scrivania/link$ cat file1.txt
contenuto 1
marco@marco-VirtualBox:~/Scrivania/link$ cat file2.txt
contenuto 2
marco@marco-VirtualBox:~/Scrivania/link$ ln file1.txt file1_hlink
marco@marco-VirtualBox:~/Scrivania/link$ ln -s file2.txt file2_slink
```

Quindi avremo creato un hard link su file1.txt e un soft link sul file2.txt.

A questo punto se vado ad aggiornare il contenuto della directory col **comando ls -l** ottengo:

```
marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 8
-rw-r--r-- 1 marco marco 12 giu  2 14:57 file1.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2.txt
marco@marco-VirtualBox:~/Scrivania/link$ cat file1.txt
contenuto 1
marco@marco-VirtualBox:~/Scrivania/link$ cat file2.txt
contenuto 2
marco@marco-VirtualBox:~/Scrivania/link$ ln file1.txt file1_hlink
marco@marco-VirtualBox:~/Scrivania/link$ ln -s file2.txt file2_slink
marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 12
-rw-r--r-- 2 marco marco 12 giu  2 14:57 file1_hlink
-rw-r--r-- 2 marco marco 12 giu  2 14:57 file1.txt
lrwxrwxrwx 1 marco marco  9 giu  2 14:59 file2_slink -> file2.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2.txt
```

Notiamo che il numero di link (seconda colonna della riga di un file) è stato aumentato rispetto a prima. Sia file1.txt sia file1\_hlink sono caratterizzati da due hard link. Nel caso del file2 notiamo come bash metta in corrispondenza il soft link del file2 col nome simbolico del file stesso.

Ora, andiamo a rinominare il file1.txt, che chiameremo file1\_new.txt:

```
marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 8
-rw-r--r-- 1 marco marco 12 giu  2 14:57 file1.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2.txt
marco@marco-VirtualBox:~/Scrivania/link$ cat file1.txt
contenuto 1
marco@marco-VirtualBox:~/Scrivania/link$ cat file2.txt
contenuto 2
marco@marco-VirtualBox:~/Scrivania/link$ ln file1.txt file1_hlink
marco@marco-VirtualBox:~/Scrivania/link$ ln -s file2.txt file2_slink
marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 12
-rw-r--r-- 2 marco marco 12 giu  2 14:57 file1_hlink
-rw-r--r-- 2 marco marco 12 giu  2 14:57 file1.txt
lrwxrwxrwx 1 marco marco  9 giu  2 14:59 file2_slink -> file2.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2.txt
marco@marco-VirtualBox:~/Scrivania/link$ mv file1.txt file1_new.txt
```

Ora accedo al contenuto dell'hard link file1\_hlink:

```
marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 8
-rw-r--r-- 1 marco marco 12 giu  2 14:57 file1.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2.txt
marco@marco-VirtualBox:~/Scrivania/link$ cat file1.txt
contenuto 1
marco@marco-VirtualBox:~/Scrivania/link$ cat file2.txt
contenuto 2
marco@marco-VirtualBox:~/Scrivania/link$ ln file1.txt file1_hlink
marco@marco-VirtualBox:~/Scrivania/link$ ln -s file2.txt file2_slink
marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 12
-rw-r--r-- 2 marco marco 12 giu  2 14:57 file1_hlink
-rw-r--r-- 2 marco marco 12 giu  2 14:57 file1.txt
lrwxrwxrwx 1 marco marco  9 giu  2 14:59 file2_slink -> file2.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2.txt
marco@marco-VirtualBox:~/Scrivania/link$ mv file1.txt file1_new.txt
marco@marco-VirtualBox:~/Scrivania/link$ cat file1_hlink
contenuto 1
```

E noteremo che il contenuto è rimasto lo stesso perché il nome è cambiato, ma la posizione in memoria del file1 no. Se faccio la stessa cosa col file2, vediamo cosa succede:

```

marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 8
-rw-r--r-- 1 marco marco 12 giu  2 14:57 file1.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2.txt
marco@marco-VirtualBox:~/Scrivania/link$ cat file1.txt
contenuto 1
marco@marco-VirtualBox:~/Scrivania/link$ cat file2.txt
contenuto 2
marco@marco-VirtualBox:~/Scrivania/link$ ln file1.txt file1_hlink
marco@marco-VirtualBox:~/Scrivania/link$ ln -s file2.txt file2_slink
marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 12
-rw-r--r-- 2 marco marco 12 giu  2 14:57 file1_hlink
-rw-r--r-- 2 marco marco 12 giu  2 14:57 file1.txt
lrwxrwxrwx 1 marco marco  9 giu  2 14:59 file2_slink -> file2.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2.txt
marco@marco-VirtualBox:~/Scrivania/link$ mv file1.txt file1_new.txt
marco@marco-VirtualBox:~/Scrivania/link$ cat file1_hlink
contenuto 1
marco@marco-VirtualBox:~/Scrivania/link$ mv file2.txt file2_new.txt
marco@marco-VirtualBox:~/Scrivania/link$ cat file2_slink
cat: file2_slink: File o directory non esistente

```

In questo caso la modifica del nome simbolico di file2 ha corrotto il link. Infine vado a rieseguire il comando ls:

```

marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 8
-rw-r--r-- 1 marco marco 12 giu  2 14:57 file1.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2.txt
marco@marco-VirtualBox:~/Scrivania/link$ cat file1.txt
contenuto 1
marco@marco-VirtualBox:~/Scrivania/link$ cat file2.txt
contenuto 2
marco@marco-VirtualBox:~/Scrivania/link$ ln file1.txt file1_hlink
marco@marco-VirtualBox:~/Scrivania/link$ ln -s file2.txt file2_slink
marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 12
-rw-r--r-- 2 marco marco 12 giu  2 14:57 file1_hlink
-rw-r--r-- 2 marco marco 12 giu  2 14:57 file1.txt
lrwxrwxrwx 1 marco marco  9 giu  2 14:59 file2_slink -> file2.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2.txt
marco@marco-VirtualBox:~/Scrivania/link$ mv file1.txt file1_new.txt
marco@marco-VirtualBox:~/Scrivania/link$ cat file1_hlink
contenuto 1
marco@marco-VirtualBox:~/Scrivania/link$ mv file2.txt file2_new.txt
marco@marco-VirtualBox:~/Scrivania/link$ cat file2_slink
cat: file2_slink: File o directory non esistente
marco@marco-VirtualBox:~/Scrivania/link$ ls -l
totale 12
-rw-r--r-- 2 marco marco 12 giu  2 14:57 file1_hlink
-rw-r--r-- 2 marco marco 12 giu  2 14:57 file1_new.txt
-rw-r--r-- 1 marco marco 12 giu  2 14:58 file2_new.txt
lrwxrwxrwx 1 marco marco  9 giu  2 14:59 file2_slink -> file2.txt

```

Il soft link del file 2 è ora visualizzato in rosso in quanto il link non è più valido (a seguito della ridenominazione del file). Tutto questo esempio è utile per chiarire la differenza tra hard e soft link.

## Metadati

La maggior parte dei file system memorizza dati diversi dai dati utente e dalle directory, come le posizioni dei blocchi liberi di un dispositivo di memorizzazione (per garantire che i nuovi dati di file non sovrascrivano i blocchi utilizzati) e l'ora in cui un file è stato modificato per l'ultima volta (per fini contabili). Questa informazione, chiamata **metadato**, protegge l'integrità del file system.

Con l'operazione preliminare di **formattazione** si consente la scrittura di metadati. Durante la formattazione vengono creati dei **superblocchi** che servono ad archiviare informazioni che proteggono l'integrità del file system.

Il superblocco contiene alcune informazioni utili:

- Un identificativo del file system che identifica in modo univoco il tipo di file system.
- Il numero di blocchi nel file system.
- La posizione dei blocchi liberi del dispositivo di archiviazione.

Per ridurre il rischio di perdita di dati, la maggior parte dei file system distribuisce copie ridondanti del superblocco su tutto il dispositivo di archiviazione. Pertanto, il file system può utilizzare copie ridondanti del superblocco per determinare se il superblocco primario è danneggiato e, in tal caso, sostituirlo.

### **File descriptor**

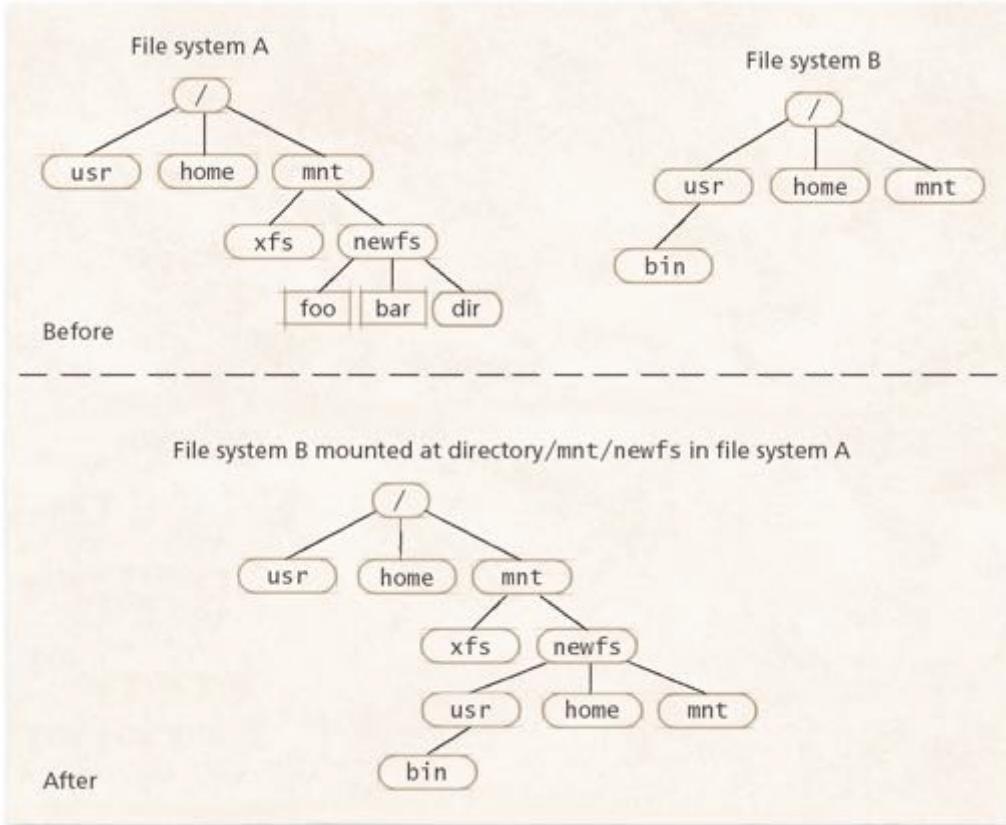
In molti sistemi, l'operazione di apertura del file restituisce un descrittore di file, un numero intero non negativo che indica nella tabella dei file aperti. Da questo punto in poi, l'accesso al file è diretto attraverso il **file descriptor**.

I file descriptor sono dei valori contenuti in una tabella chiamata **Open File Table** la quale contiene la lista di file aperti in un certo istante.

### **Mounting**

I sistemi operativi offrono la possibilità di effettuare il **mount** di più file system. Il **montaggio** combina più file system in uno spazio dei nomi, un insieme di file che possono essere identificati da un singolo file system. Lo spazio dei nomi unificato consente agli utenti di accedere ai dati da posizioni diverse come se tutti i file si trovassero all'interno del file system nativo.

Questa operazione, avviene, per esempio, quando collegiamo un disco esterno (con il proprio file system) al nostro pc. Quando facciamo ciò il nostro sistema operativo visualizza gli elementi interni al disco esterno come se quest'ultimo fosse una periferica. Per accedere agli elementi di questa periferica il sistema operativo crea una directory di mount interna al nostro os facendola corrispondere alla root directory del file system della periferica esterna. Vediamolo graficamente:



Si va a sostituire la root del file system esterno B con una directory locale.

Lo svantaggio di questo meccanismo è che lo user non potrà creare degli hard link tra i due file system, mentre i soft link saranno invece ammessi.

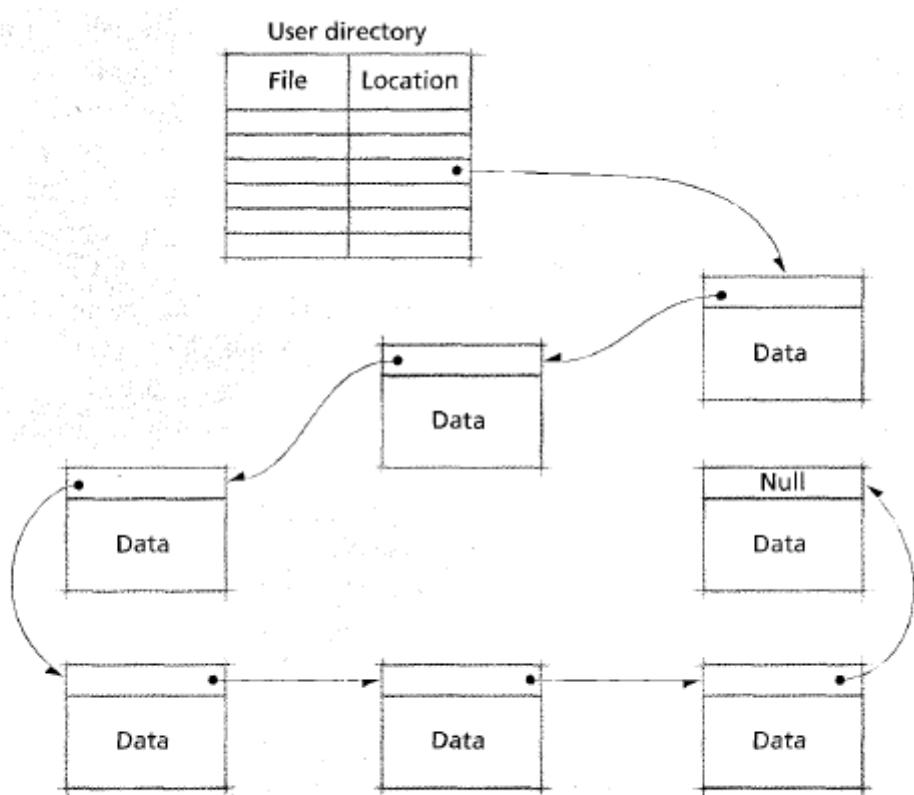
### Allocazione dei file

L'**allocazione dei file** determina il modo in cui andare a scrivere i file in relazione allo spazio disponibile. Anche in questa allocazione (come in quella della memoria) esiste la **allocazione contigua**, che però non viene utilizzata spesso, in quanto i file spesso cambiano dimensione. Tuttavia l'allocazione contigua è uno strumento semplice e utile che viene utilizzato per la scrittura di CD e DVD dove i dati scritti non verranno più modificati. Il vantaggio è che i record sono adiacenti e quindi la lettura dei record è abbastanza veloce.

Per quanto riguarda l'**allocazione non contigua** il problema sta nell'andare a determinare, di volta in volta, la posizione dei file. Per farlo ci sono più soluzioni. Vediamo nel dettaglio le **tecniche di allocazione non contigua dei file**:

#### Allocazione di file tramite linked list

Un approccio all'allocazione di file non contigua consiste nell'implementare una **linked list sector-based**. In questo schema, ogni directory entry punta al primo settore di un file su un dispositivo di archiviazione a testa mobile come un disco rigido. La porzione di dati di un settore memorizza il contenuto del file; la parte puntatore memorizza un puntatore al prossimo settore del file.



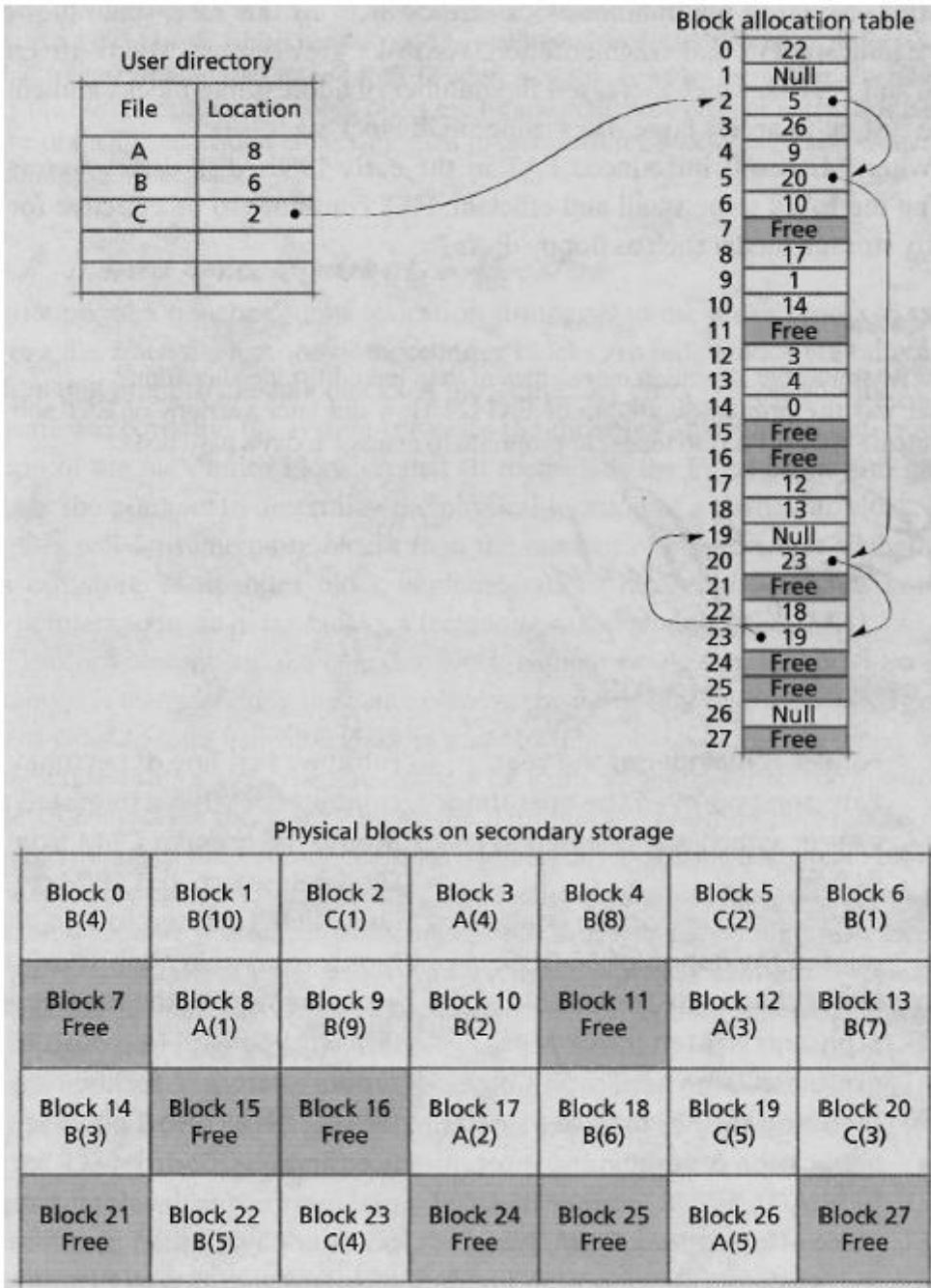
*Figure 13.6 | Noncontiguous file allocation using a linked list.*

Uno schema utilizzato per gestire lo storage secondario in modo più efficiente e ridurre l'overhead dovuto all'attraversamento dei file è chiamato **allocazione a blocchi**. In questo schema, invece di allocare i singoli settori, vengono allocati blocchi di settori contigui. Il sistema tenta di allocare nuovi blocchi in un file scegliendo blocchi liberi il più vicino possibile ai blocchi di dati di file esistenti. In questa maniera si può gestire dinamicamente l'aumento delle dimensioni dei file. I blocchi che comprendono un file contengono ciascuno due parti: un blocco dati e un puntatore al blocco successivo.

Le dimensioni del blocco possono influire in modo significativo sulle prestazioni del file system. Se i blocchi sono suddivisi tra file, le dimensioni dei blocchi di grandi dimensioni possono comportare una notevole quantità di frammentazione interna. Le grandi dimensioni dei blocchi, tuttavia, riducono il numero di operazioni di I / O necessarie per accedere ai dati dei file. Piccole dimensioni dei blocchi possono causare la diffusione dei dati dei file su più blocchi, che tendono ad essere dispersi sul disco. In generale i valori di blocchi comunemente utilizzati variano da 1 a 8kB.

### Allocazione di file tramite tabelle

L'**allocazione di file tabulare** non contigua memorizza i puntatori ai blocchi di file in modo contiguo nelle tabelle per ridurre il numero di lunghe ricerche richieste per accedere a un particolare record. Per fare ciò si utilizza una **tavella di allocazione dei blocchi** che serve a determinare la sequenza di blocchi che compongono un file. Vediamo un esempio per capirlo meglio:



**Figure 13.7 | Tabular noncontiguous file allocation.**

Le voci della directory indicano il primo blocco di un file. Ad esempio, il primo blocco del file C 2. Il numero di blocco corrente viene utilizzato come indice nella tabella di allocazione dei blocchi per determinare la posizione del blocco successivo. Pertanto, il valore del prossimo numero di blocco del file C viene memorizzato nella posizione 2 nella tabella di allocazione dei blocchi. In questo caso, il prossimo blocco del file C è 5, e così via. La voce della tabella di allocazione dei blocchi per l'ultimo blocco di un file memorizza il valore null.

Questa struttura è abbastanza semplice tuttavia per costruire un file bisogna seguire molti puntatori nella tabella di allocazione dei blocchi. Un esempio di file system che utilizza questa struttura tabellare è il FAT32 di Microsoft. Il numero dopo FAT indica il numero di bit per ogni riga della tabella di allocazione dei blocchi.

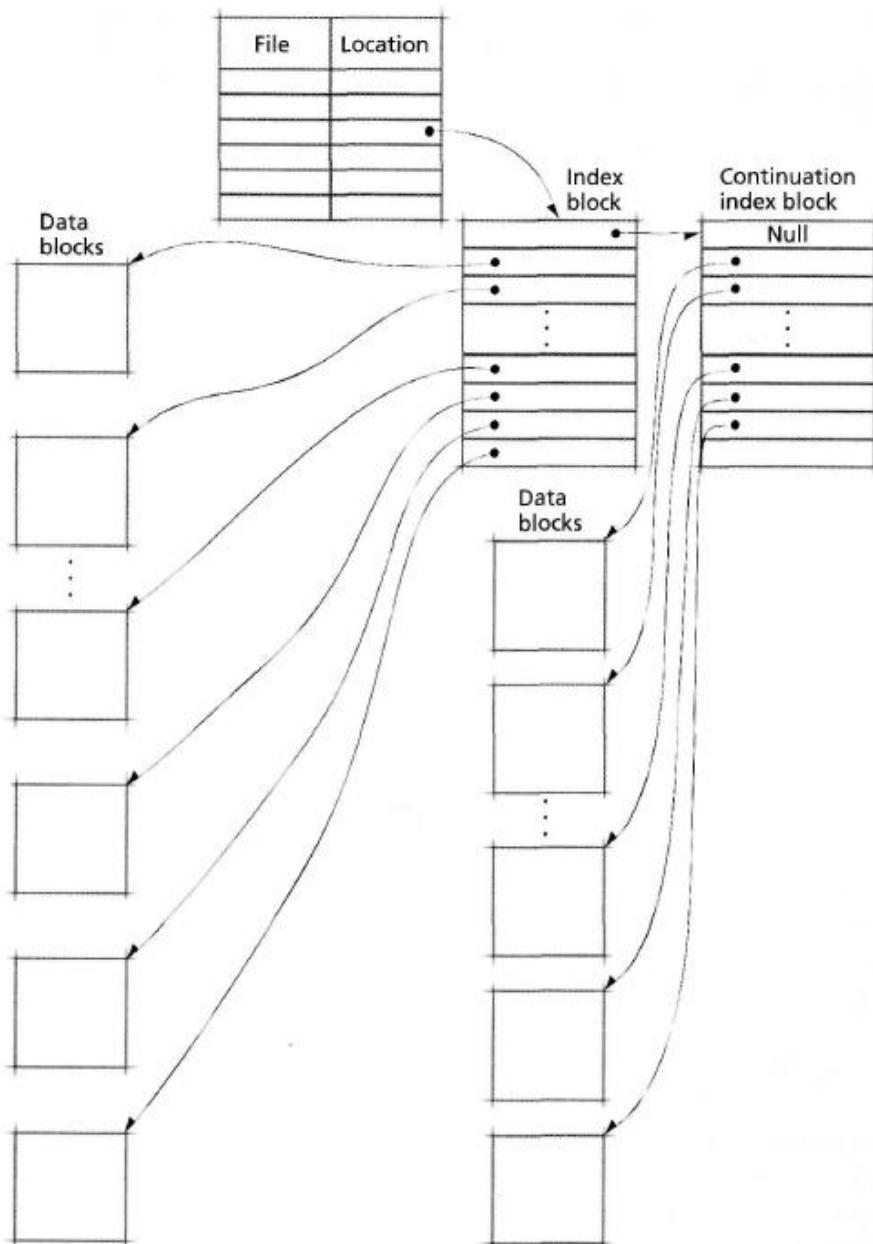
#### Allocazione di file indicizzata

Un'altra strategia di allocazione non contigua è quella di utilizzare degli **index block** per puntare ai dati in un file. Ogni file ha uno o più index block. Un index block contiene un **elenco di puntatori** che puntano ai blocchi contenenti i dati del file (**in termini semplici: un index block è un insieme di puntatori che puntano ai blocchi dati del file**).

La directory entry di un file punta al suo index block. Per individuare un record, il file system attraversa la struttura della directory per determinare la posizione dell'index block del file su disco. Quindi carica l'index block in memoria e utilizza i puntatori per determinare la posizione fisica di un particolare blocco.

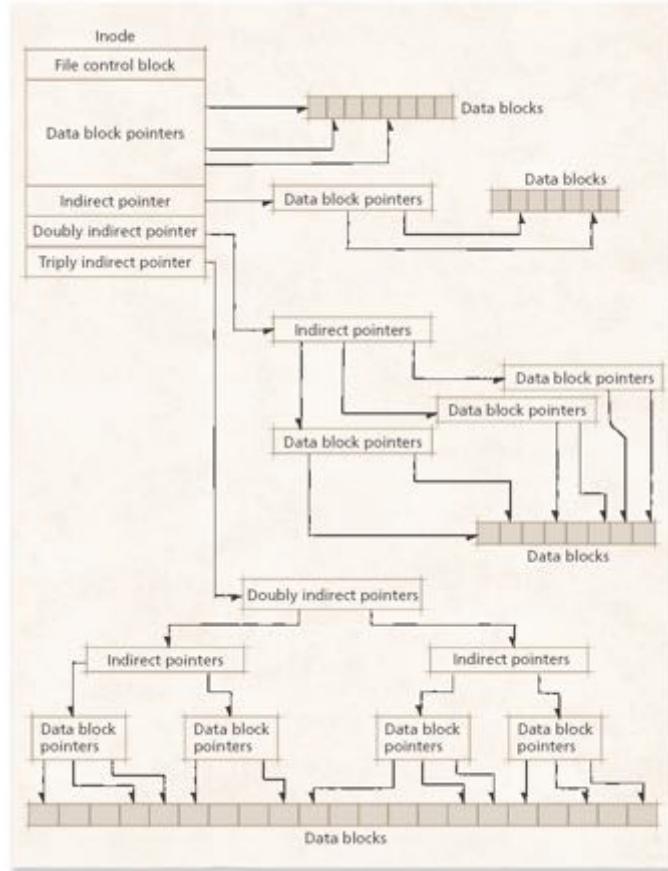
Per evitare di appesantire troppo la struttura degli index block si può suddividere l'index block di un solo file in più index block.

Vediamolo graficamente:



*Figure 13.8 | Index block chaining.*

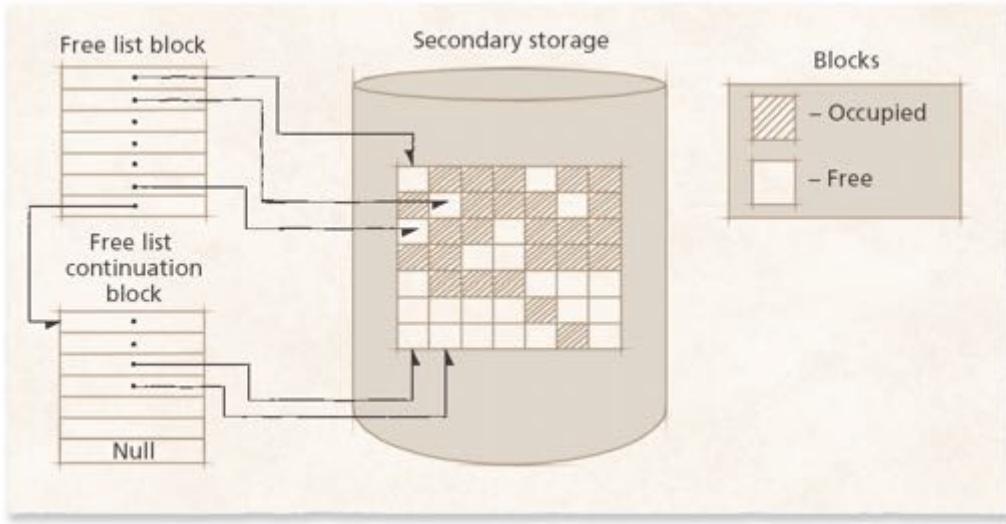
Nei sistemi UNIX gli index block prendono il nome di **inode** i quali contengono attributi relativi ai file (proprietario, dimensione, ecc..) e dei puntatori ad altri inode di continuazione, chiamati **blocchi indiretti**. Le strutture di inode supportano fino a tre livelli di blocchi indiretti, come vediamo in figura:



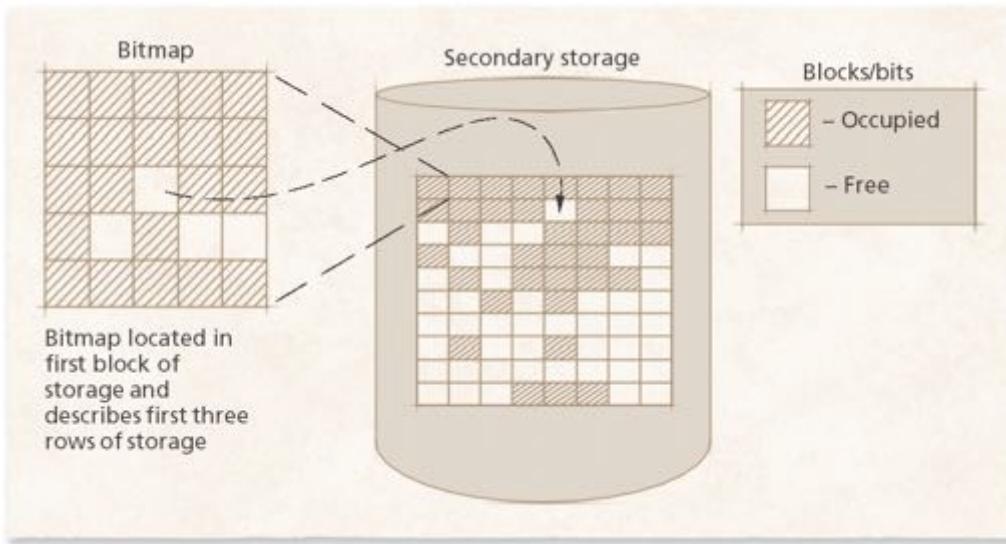
Il primo blocco indiretto punta a blocchi di dati; questi blocchi di dati sono singolarmente indiretti. Il secondo blocco indiretto contiene puntatori che fanno riferimento solo ad altri blocchi indiretti. Questi blocchi indiretti puntano a blocchi di dati che sono doppiamente indiretti. Il terzo blocco indiretto punta solo ad altri blocchi indiretti che puntano solo a blocchi più indiretti che puntano a blocchi di dati; questi blocchi di dati sono triplamente indiretti. In questa maniera si può indirizzare qualsiasi indirizzo seguendo, al più, 4 puntatori.

### Gestione dello spazio libero

Man mano che i file crescono e si restringono, i file system conservano la posizione dei blocchi disponibili per memorizzare nuovi dati (cioè i blocchi liberi). Un file system può usare una **free list** per tenere traccia dello spazio libero. La free list è un linked list di blocchi contenenti le posizioni dei blocchi liberi. L'ultima voce di un blocco di free list memorizza un puntatore al successivo blocco di questa; l'ultima voce dell'ultimo blocco della free list memorizza un puntatore nullo per indicare che non ci sono altri blocchi della free list. Quando il sistema deve allocare un nuovo blocco a un file, trova l'indirizzo di un blocco libero nella free list, scrive i nuovi dati nel blocco libero trovato e rimuove la voce di quel blocco dalla free list. Vediamolo graficamente:



Un altro metodo comune di gestione dello spazio libero è la **bitmap**. Una bitmap contiene un bit per ogni blocco nel file system, dove il bit  $i$ -esimo corrisponde al blocco  $i$ -esimo nel file system. In un'implementazione, un bit nella bitmap è 1 quando il blocco corrispondente è in uso e 0 quando non lo è. Vediamo la bitmap graficamente:



Uno svantaggio delle bitmap è che il file system potrebbe dover cercare l'intera bitmap per trovare un blocco libero, con conseguente overhead in fase di esecuzione.

### Controllo di accesso ai file

I file, spesso e volentieri, contengono dati sensibili, di conseguenza i file system dovrebbero includere meccanismi per controllare l'accesso degli utenti ai dati. Esistono varie tecniche per implementare il controllo dell'accesso ai file.

### Matrice di controllo degli accessi

Un modo per controllare l'accesso ai file consiste nel creare una **matrice di controllo degli accessi** bidimensionale che elenca tutti gli utenti e tutti i file nel sistema. La cella  $(i, j)$  è 1 se l'utente  $i$  è autorizzato ad accedere al file  $j$ ; altrimenti  $(i, j) = 0$ .

User \ File	1	2	3	4	5	6	7	8	9	10
1	1	1	0	0	0	0	0	0	0	0
2	0	0	1	0	1	0	0	0	0	0
3	0	1	0	1	0	1	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	1	1	1	1	1
6	0	0	0	0	0	1	1	0	0	0
7	1	0	0	0	0	0	0	0	0	1
8	1	0	0	0	0	0	0	0	0	0
9	1	1	1	1	0	0	0	0	1	1
10	1	1	0	0	1	1	0	0	0	1

### Controllo dell'accesso basato su classi di utenti

Una tecnica che richiede molto meno spazio rispetto all'utilizzo di una matrice di controllo degli accessi è il controllo dell'accesso basato su varie **classi di utenti**. Uno schema di classificazione di accesso ai file comune è:

- **Proprietario**: normalmente questo è l'utente che ha creato il file. Il proprietario ha accesso illimitato al file e in genere può modificare i permessi dei file.
- **Utente specificato**: il proprietario specifica che un altro utente può utilizzare il file.
- **Gruppo (o progetto)**: gli utenti sono spesso membri di un gruppo che lavora su un particolare progetto. In questo caso, a tutti i membri del gruppo può essere concesso l'accesso ai rispettivi file relativi al progetto.
- **Pubblico**: la maggior parte dei sistemi consente di designare un file come pubblico in modo che possa essere accessibile da qualsiasi membro della comunità di utenti del sistema. Per impostazione predefinita, i diritti di accesso pubblico in genere consentono agli utenti di leggere o eseguire un file, ma non di scriverlo.

### Tecniche di accesso ai dati

In molti sistemi, diversi processi potrebbero richiedere dati da vari file sparsi sul dispositivo di archiviazione, portando a molte ricerche su disco. Invece di rispondere istantaneamente alle richieste di I/O dell'utente immediatamente, il sistema operativo può utilizzare diverse tecniche per migliorare le prestazioni. I sistemi operativi odierni forniscono generalmente molti metodi di accesso. Questi sono talvolta raggruppati come **metodi di accesso con coda** e **metodi di accesso di base**.

- **Metodi di accesso con coda**: vengono utilizzati quando è possibile anticipare la sequenza in cui i record devono essere elaborati, ad esempio l'accesso sequenziale e sequenziale indicizzato. I metodi con coda eseguono il **buffering anticipatorio** e la pianificazione delle operazioni di I/O. Tali metodi cercano di avere il record successivo disponibile per l'elaborazione non appena il record precedente è stato elaborato. Più di un record alla volta viene mantenuto nella memoria principale; ciò consente di sovrapporre le operazioni di elaborazione e I/O, migliorando le prestazioni.
- **Metodi di accesso di base**: vengono normalmente utilizzati quando la sequenza in cui i record devono essere elaborati non può essere anticipata.

### Backup e ripristino dei dati

La maggior parte dei sistemi implementa tecniche di **backup** per archiviare copie ridondanti di informazioni, e **tecniche di ripristino** che consentono al sistema di ripristinare i dati dopo un errore di sistema. Le strategie di backup e ripristino possono anche proteggere i sistemi da eventi generati dall'utente, come la cancellazione involontaria di dati importanti. L'esecuzione di backup periodici è la tecnica più comune utilizzata per prevenire la perdita di dati.

Esistono vari tipi di backup:

- **Backup fisici:** i backup fisici duplicano i dati di un dispositivo di archiviazione a livello di bit. In alcuni casi, il sistema copia solo blocchi di dati allocati. I backup fisici sono semplici da implementare, ma non memorizzano informazioni sulla struttura logica del file system.
- **Backup logici:** un backup logico memorizza i dati del file system e la sua struttura logica. Pertanto, i backup logici controllano la struttura delle directory per determinare quali file devono essere sottoposti a backup, quindi scrivono questi file su un dispositivo di backup. Poiché i backup logici possono leggere solo i dati esposti dal file system, possono omettere informazioni quali file nascosti e metadati copiati da un backup fisico durante la copia di ciascun bit sul dispositivo di archiviazione del file system.
- **Backup incrementali:** i backup incrementali sono backup logici che memorizzano solo i dati del file system modificati dopo il backup precedente.

### Integrità dei dati

Se si verifica un errore di sistema durante un'operazione di scrittura, i dati del file system possono essere lasciati in uno stato inconsistente. La registrazione basata sulle transazioni riduce il rischio di perdita di dati utilizzando **transazioni atomiche**, che eseguono un gruppo di operazioni nella loro interezza o non lo fanno affatto. Se si verifica un errore che impedisce il completamento di una transazione, viene eseguito il **rollback** riportando il sistema allo stato in cui si trovava prima dell'inizio della transazione.

Le transazioni atomiche possono essere implementate registrando il risultato di ciascuna operazione in un file di log invece di modificare i dati esistenti. Una volta completata la transazione, viene eseguita la registrazione registrando un valore speciale nel registro. A un certo momento nel futuro, il registro viene trasferito nella memoria permanente. Se il sistema non riesce prima del completamento della transazione, tutte le operazioni registrate dopo la precedente transazione confermata vengono ignorate. Quando il sistema si ripristina, legge il file di log e lo confronta con i dati nel file system per determinare lo stato del file system all'ultimo punto di commit.

Per ridurre il tempo impiegato per le transazioni di rielaborazione nel registro, la maggior parte dei sistemi basati su transazioni mantiene i **punti di controllo** che puntano all'ultima transazione che è stata trasferita alla memoria permanente.

Il **paging shadow** implementa le transazioni atomiche scrivendo i dati modificati in un blocco libero invece del blocco originale. Una volta che la transazione è stata eseguita, il file system aggiorna i suoi metadati in modo che facciano riferimento al nuovo blocco e rilasci il vecchio blocco o la **shadow page** come spazio libero.

### File System Log-Structured (LFS)

La registrazione delle transazioni e il paging delle ombre impediscono ai dati del file system di entrare in uno stato incoerente, ma non garantiscono necessariamente che il file system stesso si trovi in uno stato coerente. Un **file system log-strutturato(LFS)**, esegue le operazioni del file system come transazioni registrate.