DIPARTIMENTO
DI INFORMATICA
SAPIENZA
UNIVERSITÀ DI ROMA

Sistemi Operativi I

Authors

Emanuele D'Agostino
Giuseppe Borracci

GitHub

Rurik-D
GiusTMP

AA 22/23

Indice

Indice

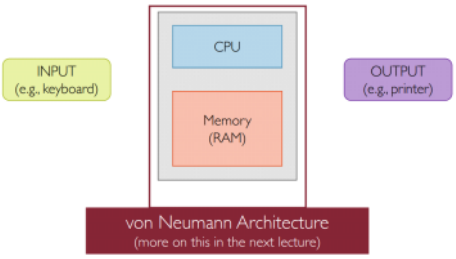
0 Introduzione

0.1 Von Neumann Architecture

L'espressione *architettura di Von Neumann* (o *macchina di Von Neumann*) è riferita allo schema progettuale pensato dal matematico ungherese John Von Neumann intorno alla metà del Novecento. Il computer, così come lo intendiamo attualmente, è stato realizzato grazie a tale schema. I concetti di base dell'architettura di Von Neumann sono inclusi anche nei moderni microprocessori

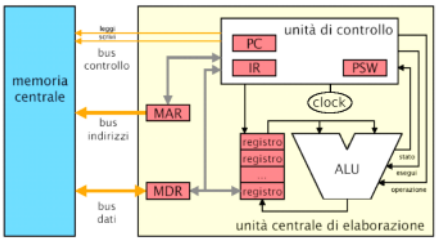
Lo schema si basa su 4 componenti fondamentali:

- **CPU**
- **RAM**
- **Interfaccia di I/O** (Input/Output)
- **Bus di sistema**



CPU

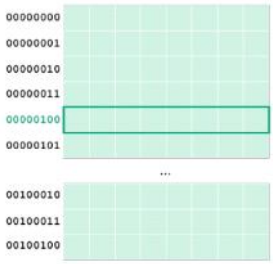
La *Central Processing Unit* è l'unità centrale di elaborazione. La CPU esegue le istruzioni di calcolo e di controllo, coordinando le altre componenti nell'esecuzione delle istruzioni stesse.



Tra le componenti principali della **CPU** troviamo:

- **ALU** (*Arithmetic Logic Unit*), che esegue i calcoli aritmetici e logici per l'esecuzione delle operazioni.
- **CU** (*Control Unit*), è l'unità di controllo che regola il funzionamento delle altre componenti della CPU.

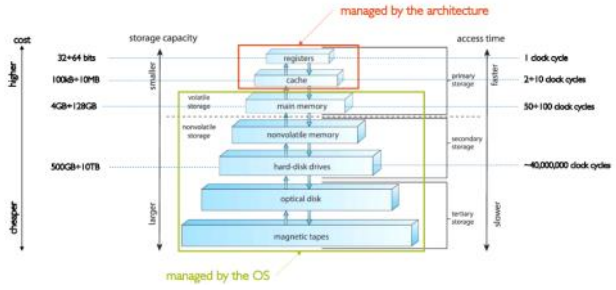
Memoria



Essenzialmente la memoria è una sequenza di celle organizzate in gruppi da 8 bit, 1 Byte, o multipli di esso; ogni cella (Byte) ha il proprio indirizzo.
La CPU legge e scrive continuamente in memoria tramite l'utilizzo degli indirizzi. La più piccola unità indirizzabile è generalmente il Byte.

Distinguiamo:

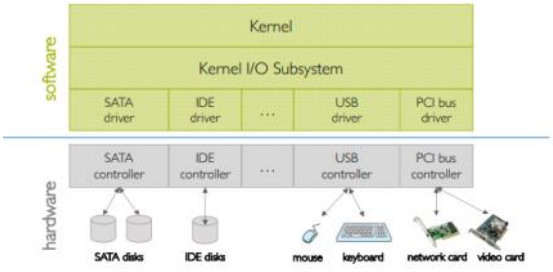
- **Memoria principale** - essa è composta essenzialmente dalla memoria **RAM** (*Random Access Memory*), e in secondo luogo anche dalla cache (memoria di alta qualità che si pone tra la RAM e la CPU) e dai registri (contenuti all'interno della CPU).
  La RAM è però una memoria volatile, per il salvataggio permanente dei dati si utilizza quindi la memoria di massa).
- **Memoria secondaria** - memoria di qualità peggiore rispetto alla RAM, ma che consente l'immagazzinamento persistente di grandi quantità di dati. Tra queste memorie troviamo gli **Hard Disk** e le **SSD**.



INPUT/OUTPUT

Ogni dispositivo I/O è composto da 2 parti:

- Il **dispositivo stesso**, con cui l'utente interagisce
- Il **device controller**, uno o più chip che controllano un insieme di dispositivi fisici



Possono essere categorizzati in: memorizzazione, comunicazione, interfacce-utente ecc...
Il sistema operativo comunica con i device controller utilizzando uno specifico device driver.

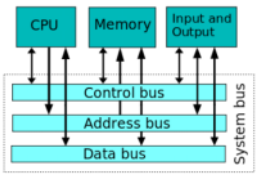
Ogni dispositivo ha un numero di registri dedicati con cui comunicare:

- Registri di stato - forniscono informazioni di stato all CPU riguardo il dispositivo (es. inattivo, pronto per l'input, occupato, errore, transazione completata)
- Registri di configurazione/controllo - usati dalla CPU per configurare e controllare il dispositivo
- Registri dati - utilizzati per leggere/scrivere dati sul dispositivo

BUS

Sono i canali di comunicazione che connettono tra loro i componenti dell'architettura.

Nell'architettura di Von Neumann possiamo distinguere tre tipi diversi di **bus**:

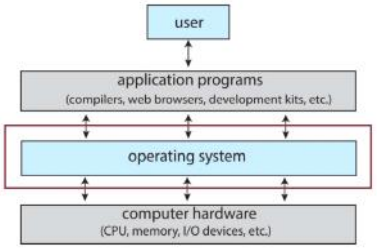


- **Bus di controllo** - abilita la memoria in lettura o in scrittura
- **Bus di indirizzo** - individua gli indirizzi in memoria
- **Bus dati** - consente il passaggio dei dati da leggere/da scrivere

Altri bus sono stati aggiunti per gestire il traffico CPU-memoria e il traffico I/O (es. PCI, SATA, USB ecc...)

0.2 Sistemi operativi

Un **sistema operativo** è un software di base composto da più sottosistemi e componenti software, come il **kernel**, lo **scheduler**, il **file system**, il **gestore della memoria**, il gestore delle periferiche, l'interfaccia utente e via dicendo.
Lo scopo ultimo è quello di fornire servizi agli utenti e alle applicazioni tramite lo sfruttamento dell'hardware.
Cambiamenti strutturali dell'hardware possono incidere sulla progettazione del sistema operativo.



0.3 API

API, acronimo di Application Programming Interface (interfaccia di programmazione delle applicazioni), indica un insieme di definizioni e protocolli per la creazione e l'integrazione di applicazioni software. Un'API è un intermediario software grazie al quale due applicazioni possono comunicare tra loro.

0.4 Istruzioni della CPU

La CPU esegue (almeno) i 3 step seguenti per ogni istruzione:

- **Fetch**: Caricamento di un'istruzione dalla memoria (posizione indicata dal Program Counter (PC))
- **Decode**: Riconoscimento dell'istruzione (la CU attiva le parti funzionali necessarie)
- **Execute**: Esecuzione dell'istruzione (in genere effettuata mediante la ALU)

Le istruzioni sono scritte in **linguaggio macchina**, che definisce
l'insieme di istruzioni elementari che la CPU può eseguire direttamente. È rappresentato il sistema binario, dove ogni istruzione è codificata in una sequenza di bit (il bit è la più piccola
unità dell'informatica). Una word è l'unità di dati con cui la CPU
può operare direttamente (ad oggi spazia tra i 32 e i 64 bit).

Simbolo	Equivalente in Byte	Equivalente nell'unità precedente	Nome dell'unità di misura
1 b	1/8 byte		Binary Digit (bit)
1 B	1 Byte	-	Byte
1 KB	1 024 B	1 024 Byte	Kilobyte
1 MB	1 048 576 B	1024 KB	Megabyte
1 GB	1 073 741 824 B	1024 MB	Gigabyte
1 TB	1 099 511 627 776 B	1024 GB	Terabyte
1 PB	1 125 899 906 842 624 B	1024 TB	Petabyte
1 EB	1 152 921 504 606 846 976 B	1024 PB	Exabyte
1 ZB	1 180 591 620 717 411 303 424 B	1024 EB	Zettabyte
1 YB	1 208 925 819 614 629 174 706 176 B	1024 ZB	Yottabyte

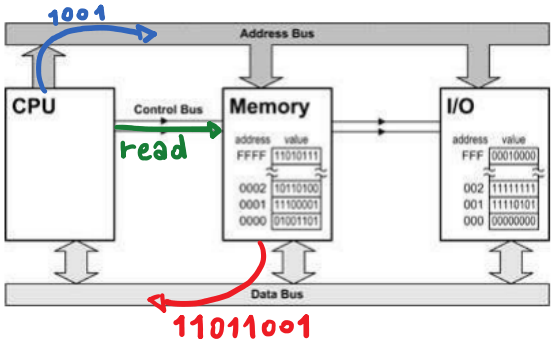
Le istruzioni in linguaggio macchina sono composte essenzialmente da due parti:

- Un **operatore** (op code)
- Zero o più **operandi** che rappresentano i registri interni della CPU o gli indirizzi in memoria

0.5 Indirizzamento tramite bus di sistema

Come la CPU fa riferimento ad un indirizzo in memoria:

- Inserisce l'indirizzo di un byte in memoria nell'**address bus**
- Attiva il segnale **read** sul **control bus**
- Eventualmente, la RAM risponde con il contenuto della memoria nel **data bus**



0.6 Port-mapped VS Memory mapped

La CPU può comunicare con i device controller in 2 modi:

- **Port-mapped I/O** - ingressi/uscite mappati su porte
- **Memory-mapped I/O** - ingressi/uscite mappati in memoria

Il Port-mapped I/O usa una speciale classe di istruzioni specifiche per l'esecuzione dell'input/output. I dispositivi di I/O hanno uno spazio indirizzi separato da quello della memoria, ottenuto tramite un extra "I/O" pin sull'interfaccia fisica della CPU oppure tramite un bus dedicato.

Il Memory-mapped I/O usa lo stesso bus per indirizzare sia la memoria che i dispositivi di I/O, e le stesse istruzioni della CPU utilizzate per leggere e scrivere. La memoria è utilizzata anche per accedere ai dispositivi di I/O. Per poter alloggiare i dispositivi di I/O, aree di spazio indirizzabile dalla CPU devono essere riservate per l'I/O invece che essere usate per la memoria.

Il vantaggio principale dell'uso del port-mapped I/O si ha su CPU con limitate capacità di indirizzamento. Dato che il port-mapped I/O separa l'accesso a ingressi/uscite dagli accessi alla memoria, tutto lo spazio di indirizzi può essere usato per la memoria.

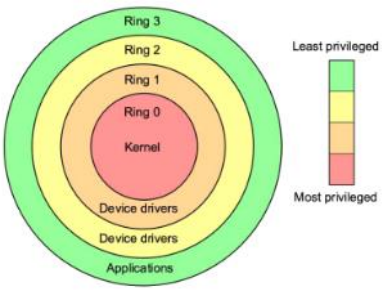
Il vantaggio dell'uso del Memory-mapped I/O è che, non utilizzando una complessità ulteriore necessaria per il port-mapped I/O, la CPU richiede meno logica interna ed è di conseguenza più economica, veloce e facile da costruire; questo segue il principio del RISC (Reduced Instruction Set Computing). Dato che CPU con architetture a 16 bit stanno diventando obsolete e vengono rimpiazzate con architetture a 32 o 64 bit, riservare spazio nella mappatura di memoria dei dispositivi I/O non è più un problema.

0.7 Protezione

I computer agiscono in due modalità diverse che si possono alternare durante l'esecuzione dei programmi:

- **Kernel mode**
  Modalità che consente di agire senza limitazioni, il sistema operativo può eseguire qualsiasi tipo di operazione, comprese quelle privilegiate
- **User mode**
  Modalità limitata per i programmi utente, serve a evitare il danneggiamento involontario (o volontario) di dati essenziali per il funzionamento del sistema operativo.
  L'utente non può quindi, ad esempio, accedere agli indirizzi I/O direttamente, modificare il contenuto della memoria principale, fermare la macchina, passare alla kernel mode, ecc...

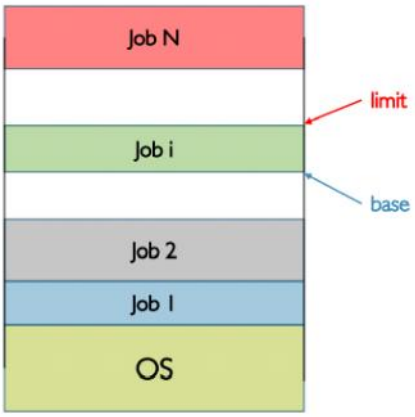
L'hardware sottostante deve essere progettato per supportare almeno kernel E user mode. Sono infatti possibili anche soluzioni più articolate, è il caso dei
protection rings, supportati per esempio dai processori Intel, dove esistono 4
livelli principali di protezione che vanno da 3 (user mode) a 0 (kernel) mode.
Intel supporta anche livelli più bassi (-1, -2, -3), ma a quel punto non è più competenza del sistema operativo (windows, ad esempio, lavora solo sui livelli 0 e 3).



L'architettura deve inoltre fornire supporto al sistema operativo per la protezione dei proteggere i programmi utente gli uni dagli altri.

La tecnica più semplice è quella di utilizzare **2 registri dedicati**:

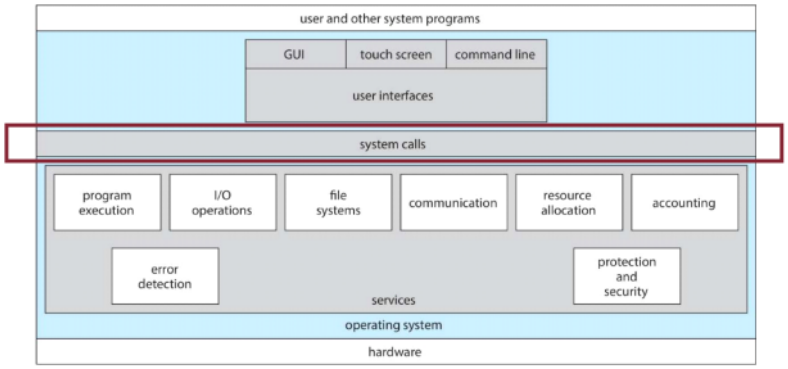
- base
  Contiene l'indirizzo di memoria valido dove il processo può iniziare ad operare
- limit
  Contiene l'ultimo indirizzo di memoria valido



Il SO carica i registri base e limit all'avvio del programma.
La CPU controlla che ogni indirizzo di memoria referenziato dal programma utente ricada tra i valori base e limit.

0.8 Servizi del sistema operativo

Un sistema operativo fornisce un'ambiente per l'esecuzione dei programmi, rendendo disponibili dei servizi ai programmi e agli utenti di tali programmi. Possiamo però riassumere alcune classi di servizi comuni alla maggior parte dei sistemi operativi con il seguente schema:



Di seguito un set di servizi generalmente forniti dai sistemi operativi:

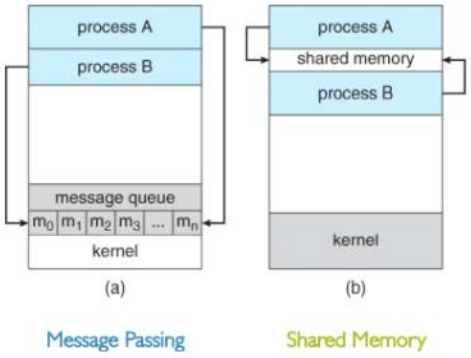
- **User Interface (UI)**
  La maggior parte dei sistemi operativi è fornita di un'interfaccia utente. Essa può avere varie forme, generalmente viene usata la **Graphical User Interface (GUI)**. In questo caso l'interfaccia è un sistema di finestre con un puntatore (gestito dal mouse) che serve a direzionare l'I/O.
  I sistemi mobile offrono generalmente invece la **Touch-Screen Interface**, consentendo all'utente di selezionare con il semplice tocco delle dita.
  Un'altra opzione è la **Command-Line Interface (CLI)**, che usa comandi in formato di testo e un metodo per confermarli (generalmente inseriti tramite tastiera).
- **Program execution**
  Il sistema deve essere in grado di caricare un programma nella memoria e di eseguirlo. Il programma deve essere in grado di terminare la sua esecuzione normalmente o anormalmente (indicando un errore).
- **Operazioni I/O**
  I programmi potrebbero richiedere operazioni di I/O, il sistema operativo deve quindi fornire un modo per eseguire queste operazioni. L'I/O generalmente non viene gestito direttamente dall'utente per questioni di protezione ed efficienza.
- **Manipolazione del file-system**
  Un "file-system", indica informalmente un meccanismo con il quale i file sono posizionati e organizzati su dispositivi di archiviazione di dati, ad esempio una SSD, o su dispositivi remoti tramite protocolli di rete. Il sistema operativo deve fornire la possibilità di accedere al file-system e di scrivere, leggere, eliminare, creare e cercare i file.

- **Communication**

  In parecchie circostanze capita che un processo necessiti di comunicare con un altro processo (sullo stesso si sistema o su sistemi diversi interconnessi da un network).

  Esistono due modelli comuni per l'implementazione della comunicazione tra processi:

  - **Message passing**
  - **Shared memory**



Il modello **message passing** i processi comunicanti possono scambiarsi messaggi l'un l'altro per trasferire informazioni tramite una mail-box comune. Prima che la comunicazione possa avvenire deve essere aperta una connessione. Il nome dell'altro comunicante deve essere conosciuto (ogni processo ha un **process name** tradotto in un ID) e i processi devono essere sullo stesso sistema o su sistemi diversi connessi da una **communication network**.
È il metodo più semplice e facile, particolarmente usato per le comunicazioni inter-computer, generalmente appropriato per piccole quantità di dati.

Il modello **shared memory** usa shared_memory_create() e shared_memory_attach() per creare e ottenere l'accesso a regioni di memoria possedute da un altro processo. Ricordiamo che il sistema operativo tenta di prevenire che un processo possa accedere alla memoria di un altro processo. La shared memory richiede quindi che i processi coinvolti siano d'accordo sulla rimozione di tali restrizioni. Lo scambio di informazioni avviene mediante la lettura e scrittura, da parte dei processi coinvolti, della stessa area di memoria. È responsabilità di tali processi di evitare di modificare contemporaneamente gli stessi dati.
È la soluzione migliore nel caso della condivisione di grandi quantità di dati. Ideale soprattutto quando i processi necessitano principalmente di leggere i dati piuttosto che scriverli.

- **Gestione degli errori**

  L'OS necessita di individuare e correggere errori costantemente. Gli errori possono verificarsi nella CPU o nella memoria, nei dispositivi I/O o nei programmi utente. Per ogni tipo di errore l'OS dovrebbe intraprendere le operazioni appropriate per assicurare la corretta e consistente computazione.

0.9 Interruzioni

INTERRUPT

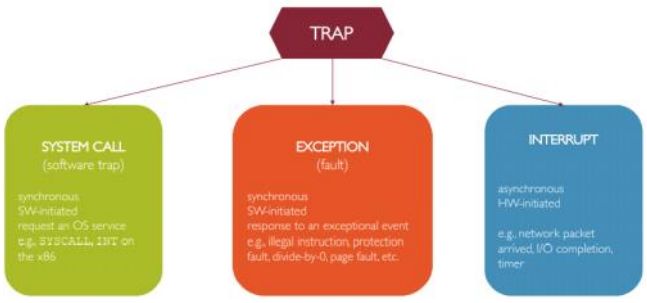
Termine solitamente riservato alle interruzioni hardware. Sono interruzioni del controllo del programma causate da eventi hardware esterni alla CPU.
Possono derivare ad esempio da chip timer, periferiche, porte I/O, unità disco ecc..

EXCEPTION

È un interrupt software, che può essere identificato come una routine speciale dal gestore. Un'eccezione si verifica nel caso di una condizione "eccezionale" durante l'esecuzione di un programma.
Possono esser causati ad esempio da divisioni per zero o l'esecuzione di un codice operativo illegale.

TRAP

Ci riferiamo ad una 'trap' come qualsiasi evento che causa l'interruzione di un processo con il conseguente passaggio da user-mode a kernel-mode.



0.10 System calls

I programmi utente possono richiedere al SO l'esecuzione di operazioni privilegiate (che altrimenti non potrebbero essere eseguite) in modalità kernel, delegando quindi l'esecuzione dell'operazione. Queste richieste vengono effettuate mediante l'utilizzo delle **system calls**.
L'utente potrebbe ad esempio necessitare della scrittura di dati su un file memorizzato su un disco o di inviare dati tramite l'interfaccia di rete.

Abbiamo 6 tipi di system calls:

- **Process control**

  Utilizzate per la gestione del flusso di esecuzione dei processi.

  Include: end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, allocate/free memory.

- **File management**

  Utilizzate per ottenere l'accesso alle tipiche operazioni sui file.

  Include: create file, delete file, open, close, read, write, reposition, get file attributes, set file attributes.

- **Device management**

  Utilizzate per richiedere o gestire le risorse hardware necessarie, come potenza di calcolo o spazio di archiviazione.

  Include: request device, release device, read, write, reposition, get/set device attributes, logically attach or detach devices

- **Information maintenance**

  Utilizzare per scambiare o richiedere le informazioni, consentono di conservare l'integrità dei dati.

  Include: get/set time, date, system data/process, file, device attributes.

- **Communications**

  Consente un'interazione regolare tra l'OS e i vari programmi utente.
  Le system call possono operare su entrambi i modelli precedentemente analizzati:
  - **Message passing**
  - **Shared memory**

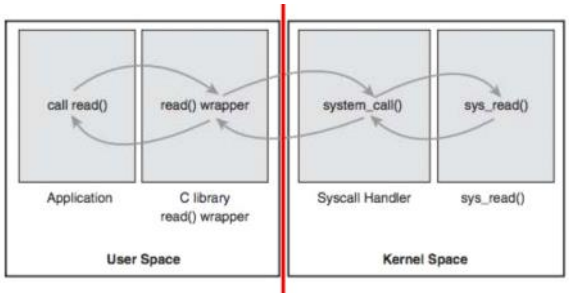
  Include: create/delete communication connection, send/receive messages, transfer status information, attach/detach remote devices.

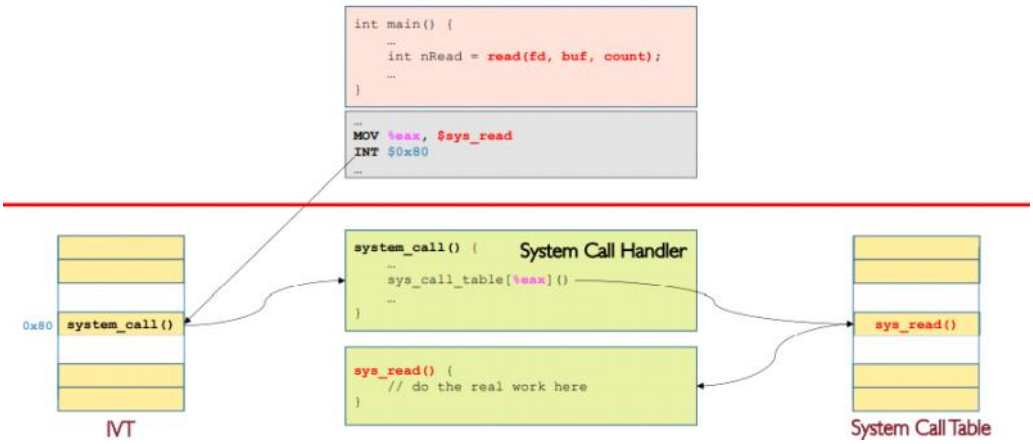
- **Protection**

  Fornisce meccanismi per controllare quali utenti/processi hanno accesso a quali risorse di sistema. Le system calls consentono di regolare tali meccanismi secondo necessità. Agli utenti non privilegiati possono esser concessi temporaneamente permessi di accesso elevati in casi particolari.

0.11 System call Handler

Durante la chiamata di una system call il chiamante non ha bisogno di sapere come essa sia implementata. Il chiamante deve soltanto obbedire alla API, che conosce gli argomenti di input, e l'output atteso dal sistema operativo. La maggior parte dei dettagli della chiamata sono quindi nascosti dalla API.





La trap causata dall'invocazione della chiamata di sistema fa passare la CPU dalla modalità utente a quella kernel.

Il system call handler è responsabile di:

- Salvare lo stato della computazione in modalità user-mode in registri dedicati
- Trovare e saltare alla routine corretta per quella "trap" (es. **sys_read()**)
- Ripristino della stato del programma in user-mode quando la routine è terminata (es. l'istruzione privilegiata **IRET**)

0.12 Memoria virtuale

È un'astrazione (dell'attuale memoria principale fisica).Dà ad ogni processo l'illusione che la memoria fisica sia solo uno spazio di indirizzi contigui (spazio di indirizzi virtuali) e permette di eseguire programmi senza che vengano caricati interamente nella memoria principale.

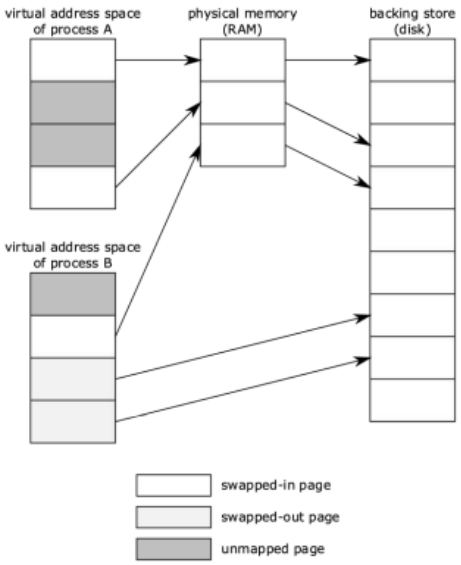
E' Implementata sia in HW (MMU) che in SW (OS):

- MMU(L'**unità di gestione della memoria**, indica una classe di componenti **hardware** che gestisce le richieste di accesso alla **memoria** generate dalla **CPU**.) è responsabile della traduzione degli indirizzi virtuali in indirizzi fisici.
- OS è responsabile della gestione degli spazi di indirizzi virtuali

Su un sistema a 64 bit la CPU è in grado di indirizzare 2^{64} byte = 16 exbibyte (EiB),quindi lo spazio degli indirizzi virtuali va da 0 a 2^{64}-1.Questo significa che la memoria virtuale è circa un miliardo di volte più grande della memoria principale.

Lo spazio degli indirizzi virtuali è suddiviso in blocchi contigui della stessa dimensione (ad esempio, 4KiB), chiamati **pagine**. Le pagine che non sono caricate nella memoria principale vengono memorizzate su disco.

L'**MMU** associa gli indirizzi virtuali a quelli fisici tramite una tabella di pagine gestita dall'OS e utilizza una cache denominata **Translation Look-aside Buffer (TLB)** con "mappature recenti" per ricerche più rapide.
Il sistema operativo deve sapere quali pagine sono caricate nella memoria principale e quali su disco.



0.13 OS Design and Implementation

La struttura interna dei diversi sistemi operativi può variare notevolmente.
È fondamentale separare le politiche dai meccanismi:

- **Politica**: *cosa* sarà fatto
- **Meccanismo**: *come* farlo

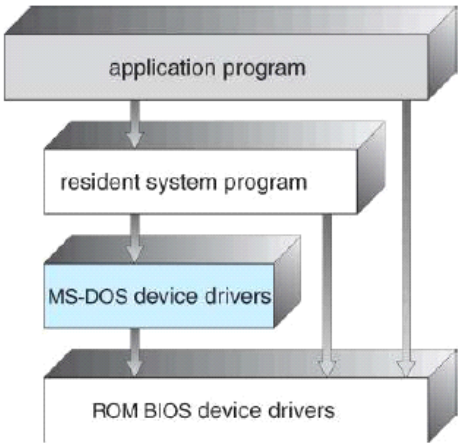
Il disaccoppiamento della logica della politica dal meccanismo sottostante, è un principio di progettazione generale nell'informatica, in quanto migliora il sistema dandogli:

- **Flessibilità**: l'aggiunta e la modifica delle politiche possono essere facilmente supportate
- **Riutilizzabilità**: i meccanismi esistenti possono essere riutilizzati per l'attuazione di nuove politiche
- **Stabilità**: l'aggiunta di una nuova policy non destabilizza necessariamente il sistema

Le modifiche alle politiche possono essere facilmente regolate senza riscrivere il codice.

Il sistema operativo dovrebbe essere suddiviso in sottosistemi separati, ciascuno con compiti, input, output e caratteristiche prestazionali accuratamente definiti.
Vari modi per strutturare un sistema operativo:

- Semplice: MS-DOS
- Complesso: UNIX
- A strati: MULTICS
- Microkernel: MACH



STRUTTURA SEMPLICE (MS-DOS)

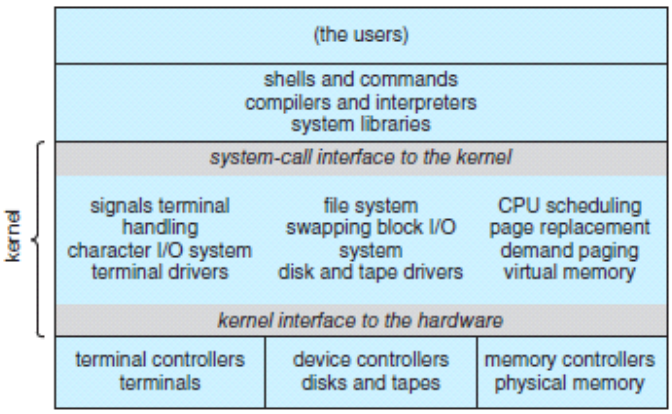
Nessun sottosistema modulare! **E nessuna separazione tra modalità utente e modalità kernel.**

PRO: facile da implementare
CONTRO: rigidità, sicurezza

KERNEL MONOLITICO TRADIZIONALE (UNIX)

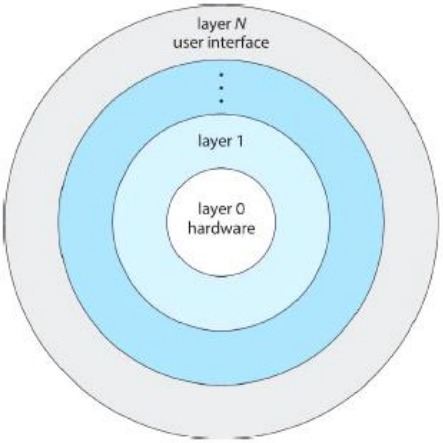
Consiste di due parti separabili: il **kernel** e i **programmi di sistema**. Il kernel è ulteriormente separato in una serie di interfacce e driver delle periferiche. Possiamo vedere il sistema operativo UNIX come **a strati**. Come si può vedere dall'immagine, tutto quello che c'è sotto le chiamate di sistema e sopra l'hardware, è il kernel. Il kernel fornisce lo scheduling della CPU, la gestione della memoria e altre operazioni di sistema tramite le chiamate di sistema. Insomma, è un enorme quantità di funzionalità da combinare in **unico spazio di indirizzi.** La maggior parte dei moderni sistemi operativi sono una variante di questa tradizionale struttura monolitica.

PRO: efficienza, facile da implementare
CONTRO: rigidità, sicurezza



STRUTTURA A STRATI (MULTICS)

Tale sistema è diviso in **componenti separate** e più piccole che hanno funzionalità specifiche e limitate. Tutti questi componenti insieme costituiscono il kernel. Il vantaggio di questo approccio modulare è che i cambiamenti in un componente **influiscono** solo su quello stesso componente. Il sistema operativo è suddiviso in N livelli (HW = strato 0). Ogni **layer** L utilizza le funzionalità implementate dal layer **L-1** per offrire nuove funzionalità al layer **L+1**. Questo approccio **semplifica il debug** e la **verifica del sistema**.

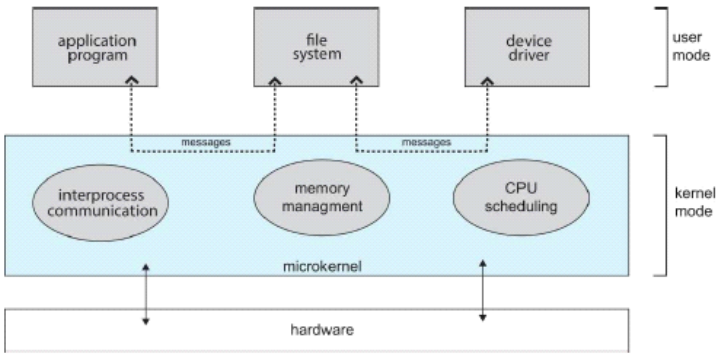


PRO: modularità, portabilità, facilità di debug
CONTRO: communication overhead, extra copy

STRUTTURA DEL MICROKERNEL (MACH)

Questo metodo struttura il sistema operativo **rimuovendo tutti i componenti non essenziali** dal kernel, e implementandoli a livello utente in programmi che risiedono in spazi di indirizzi separati.
Il risultato è un kernel più piccolo. I microkernel forniscono **processi minimi**, gestione della memoria e una struttura di comunicazione. La funzione principale del microkernel è **fornire la comunicazione** tra il programma client e i vari servizi che sono anche in esecuzione nello spazio utente. La comunicazione viene fornita attraverso il **"message passing"**. Ad esempio, se il client desidera accedere a un file, deve interagire con il file server. Il client e il servizio non interagiscono mai direttamente. Piuttosto, comunicano indirettamente scambiando messaggi con il microkernel. Uno dei vantaggi dell'approccio del microkernel è che **semplifica l'estensione del sistema operativo**. Tutti i nuovi servizi vengono aggiunti allo spazio utente e di conseguenza non richiedono la modifica del kernel. Il microkernel fornisce anche maggiore sicurezza e affidabilità, poiché la maggior parte dei servizi è in esecuzione come processi utente, piuttosto che kernel.

Se un servizio fallisce, il resto del sistema operativo rimane intatto. Le prestazioni dei microkernel possono risentirne a causa dell'aumento del sovraccarico (overhead) delle funzioni di sistema. Quando due servizi a livello utente devono comunicare, i messaggi devono essere copiati tra i servizi, che risiedono in spazi di indirizzi separati. Inoltre, il sistema operativo potrebbe dover passare da un processo all'altro per scambiare i messaggi.



MODULI KERNEL CARICABILI (LKMs)

Forse la migliore metodologia attuale per la progettazione del sistema operativo. Qui, il kernel ha una serie di componenti principali e può collegare servizi aggiuntivi tramite **moduli**, sia all'avvio che durante l'esecuzione. Il collegamento dinamico dei servizi è preferibile all'aggiunta di nuove funzionalità direttamente nel kernel, che richiederebbe la ricompilazione del kernel ogni volta che viene apportata una modifica. per esempio, potremmo creare algoritmi di scheduling della CPU e di gestione della memoria direttamente nel kernel e quindi aggiungere il supporto per diversi file system tramite **moduli caricabili**.
Il risultato assomiglia a un sistema a strati in quanto ogni sezione del kernel ha interfacce definite e protette; ma è più flessibile di un sistema a strati, perché qualsiasi modulo può chiamare qualsiasi altro modulo. E' simile anche all'approccio del microkernel perché il modulo principale ha solo funzioni di base e sa come caricare e comunicare con altri moduli; ma è più efficiente, perché i moduli non hanno bisogno di invocare il **"message passing"** per comunicare.

HYBRID SYSTEMS

In pratica, pochissimi sistemi operativi adottano un'unica struttura rigorosamente definita.
Invece, combinano strutture diverse, dando vita a sistemi ibridi che risolvono problemi di prestazioni, sicurezza e usabilità.

1 Gestione dei Processi

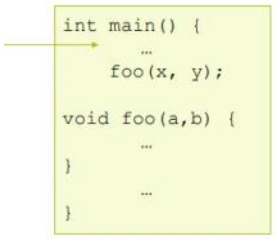
1.1 Stack

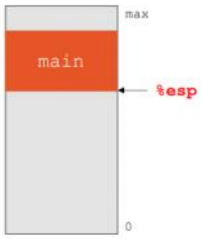
È una struttura dati di tipo **LIFO** *(Last-In-First-Out)*, dove l'ultimo elemento entrato sarà il primo ad essere rimosso. Lo spazio occupato dallo stack cresce verso il basso.

Nello **stack** sono definite **2 operazioni**:

- **Push :** per aggiungere elementi sulla pila
- **Pop** : per rimuovere elementi dalla pila

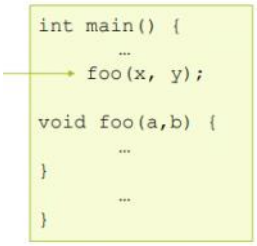
Esiste un registro dedicato (ad esempio, **esp**) il cui contenuto è l'indirizzo, nella memoria principale, della parte superiore dello stack (**%esp** è il suo contenuto).

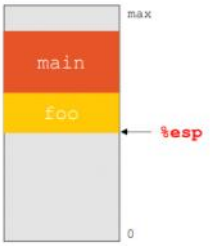




Ogni funzione utilizza una porzione dello stack, chiamata **stack frame**. In ogni momento, possono esistere simultaneamente più stack frame, a causa di diverse chiamate di funzioni annidate, ma **solo uno è attivo**.
Per ogni funzione Lo stack frame è diviso in **3 parti**:

1) parametri della funzione + indirizzo di ritorno
2) back-pointer al precedente stack frame
3) variabili locali

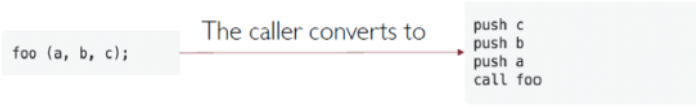




La **prima parte** è impostata dal chiamante(**caller**), mentre la **seconda** e la **terza** sono impostate dal chiamato(**callee**).

PARAMETRI DELLA FUNZIONE + INDIRIZZO DI RITORNO

Quando un oggetto viene "spinto" nella pila, la pila si incrementa, ma il valore del registro **esp** viene decrementato (ad esempio, di 4 byte nelle macchine a 32 bit), in quanto lo stack decresce verso l'heap, e l'elemento viene copiato nella posizione di memoria da esso indicata. L'istruzione di **chiamata** inserirà implicitamente nello stack l'indirizzo di ritorno.



1.2 Processi

Un **programma** è un file eseguibile che risiede nella memoria persistente (ad es. disco) in cui è contenuto l'insieme di istruzioni necessarie per eseguire un lavoro specifico. Ad esempio, il programma **ls** (comando di Linux per elencare file e cartelle di una repository) è un file eseguibile memorizzato in **/bin/ls** sul disco.
Un **processo** è una particolare istanza di un programma quando viene caricato nella memoria principale.
Ad esempio, **più istanze** del programma **ls** equivalgono a **più processi** che eseguono lo stesso programma.

Un processo è **l'astrazione** del sistema operativo di un programma in esecuzione (**unità di esecuzione**).
Il processo è **dinamico**, mentre il programma è **statico**. Diversi processi possono eseguire lo stesso programma (ad esempio, più istanze di Google Chrome) ma **ognuno ha il proprio stato**. Un processo esegue un'istruzione alla volta, in **sequenza**.

Il sistema operativo fornisce generalmente **la stessa** quantità di spazio di indirizzi virtuali a ciascun processo.
Lo spazio degli indirizzi virtuali è **un'astrazione** dello spazio di indirizzamento della memoria fisica.
L'intervallo di indirizzi virtuali validi che può essere assegnato ad un processo **dipende dalla macchina**. Ad esempio, su un'architettura a 32 bit, gli indirizzi virtuali vanno da 0 a 2^{32} -1 (ad eccezione di alcuni indirizzi riservati al kernel del sistema operativo).

Sebbene le sezioni dello stack e dell'heap crescano **l'una verso l'altra,** il sistema operativo deve garantire che **non si sovrappongano** l'una all'altra.

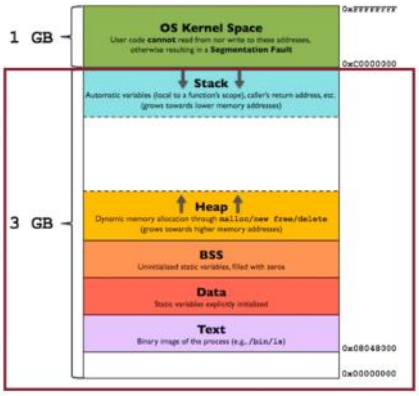
Text: contiene istruzioni eseguibili

Data: variabili globali e statiche (inizializzate)

BSS: variabile globale e statica (non inizializzata o inizializzata a 0)

Stack: Struttura LIFO utilizzata per memorizzare tutti i dati necessari a una chiamata di funzione (stack frame)

Heap: utilizzato per l'allocazione dinamica



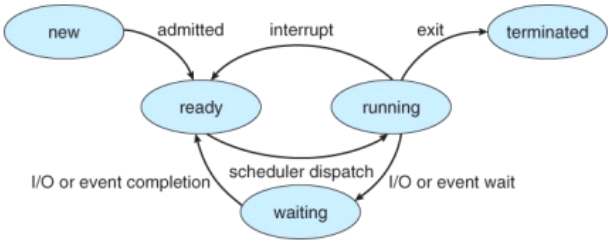
1.3 STATO DI ESECUZIONE DEI PROCESSI

Un processo può trovarsi in uno dei seguenti **5 stati**:

- **New:** Il sistema operativo ha inizializzato lo stato del processo
- **Ready:** Il processo è pronto per essere eseguito ma in attesa di essere pianificato sulla CPU
- **Running:** Il processo sta eseguendo istruzioni sulla CPU
- **Waiting:** Il processo è sospeso in attesa che una risorsa sia disponibile o che un evento si completi/si verifichi (ad esempio, input da tastiera, accesso al disco, timer, ecc.).
- **Terminated:** Il processo è terminato e il sistema operativo può distruggerlo

Durante l'esecuzione, il processo passa da uno stato all'altro in base a:

- **Azioni del programma** (ad es. chiamate di sistema)
- **Azioni del sistema operativo** (ad es. Scheduling)
- **Azioni esterne** (ad es. Interrupts)



La maggior parte delle chiamate di sistema (ad esempio quelle di I/O) sono **bloccanti**. Il processo chiamante (**user space**) non può fare nulla fino al **ritorno della chiamata di sistema**.

Il **sistema operativo** (**kernel space**):

- imposta il processo corrente in uno **stato di attesa** (ovvero, in attesa del ritorno della chiamata di sistema)
- **schedula un processo diverso** che è pronto, per evitare che la CPU sia inattiva.

Una volta che la chiamata di sistema ritorna, il processo precedentemente bloccato è pronto per essere nuovamente schedulato per l'esecuzione.
NOTA: non è bloccato l'intero sistema, lo è solo il processo che ha richiesto la chiamata!

1.4 PCB

È la **struttura dati principale** utilizzata dal sistema operativo **per tenere traccia** di qualsiasi processo. Il PCB tiene traccia dello **stato di esecuzione** e della **locazione di un processo**. Il sistema operativo **alloca** un nuovo PCB quando viene creato un processo e lo inserisce in una **coda**, mentre lo **dealloca** appena il processo associato termina.

Il **PCB** contiene quanto segue:

- **Stato del processo:** (ready, running, waiting, ecc.)
- **Numero di processo:** (ovvero un identificatore univoco)
- Program Counter (**PC**) + Stack Pointer (**SP**) + registri generici
- **Informazioni di scheduling della CPU**: priorità e puntatori alle code
- **Informazioni sulla gestione della memoria**: tabelle delle pagine (page tables)
- **Informazioni sull'account**: tempo utilizzato dalla CPU dell'utente e del kernel
- **Stato I/O**: elenco dei file aperti

1.5 OPERAZIONI SUI PROCESSI

Nella maggior parte dei sistemi operativi i processi vengono eseguiti in concomitanza, possono inoltre essere **creati e eliminati dinamicamente**. I sistemi devono quindi fornire sistemi di creazione e terminazione dei processi.

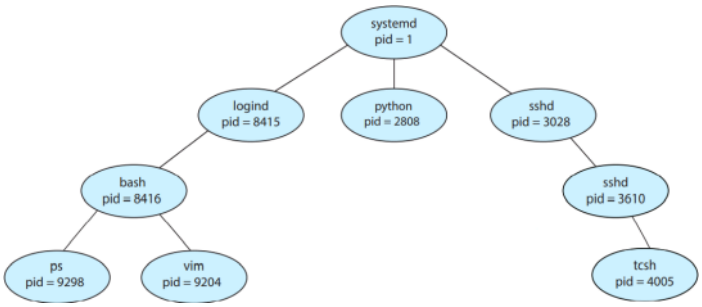
CREAZIONE

Durante l'esecuzione, un processo può creare parecchi nuovi processi. Un processo che crea un nuovo processo è chiamato **processo padre**, mentre il processo creato è chiamato **processo figlio**; ogni nuovo processo può crearne altri, formando così un **albero di processi**.

La maggior parte dei sistemi operativi identifica un processo tramite un identificatore unico chiamato **pid** *(process identifie)*, che tipicamente è rappresentato da un intero. Il **pid** fornisce quindi un valore unico per ogni processo, esso può essere usato come indice di accesso a vari attributi del processo da parte del kernel o di altri processi. Ogni processo ha inoltre un **ppid** *(parent pid)* che identifica il pid del processo padre di quel processo.

Tipicamente nei sistemi UNIX il **processo scheduler** è chiamato **sched** e gli viene assegnato **pid 0**. La prima cosa che lo scheduler fa all'avvio (**boot**) del programma è di lanciare il **processo init**, al quale viene assegnato **pid 1** e che si occuperà della creazione di tutti i processi necessari all'avvio della macchina.

Di fianco viene illustrato un tipico albero di processi per il SO Linux (anche se nel gergo di Linux si usa il termine **task** invece di process). Inizialmente le distribuzioni di Linux usavano il processo **init** per l'avvio del sistema, ma le distribuzioni più recenti lo hanno rimpiazzato con systemd, il quale, seppur molto simile a init, è molto più flessibile e può offrire molti più servizi. Nell'esempio notiamo i processi figli di systemd logind, responsabile della gestione dei client che accedono direttamente al sistema, e sshd, responsabile della gestione dei client che si connettono al sistema tramite ssh *(secure shell)*.



Su UNIX e Linux possiamo trovare il comando ps utilizzato per elencare i processi. Ad esempio, con il comando ps -el possiamo elencare tutti i processi attivi nel sistema. Tramite il comando pstree verrà invece mostrato l'albero di tutti i processi attivi nel sistema (come nella figura sopra).

Generalmente, quando un processo crea un processo figlio, questo nuovo processo richiede delle risorse per il suo funzionamento (CPU time, memoria, file, dispositivi I/O). Il processo figlio può ottenere queste risorse direttamente dal SO, oppure può essere limitato ad attingere ad un sottoinsieme di risorse del processo padre. Nel secondo caso il processo padre può partizionare le sue risorse tra i suoi figli, oppure condividere alcune risorse tra alcuni di loro. Ad ogni modo il restringimento delle risorse di un processo figlio ad un sottoinsieme delle risorse del padre impedisce a qualsiasi processo di sovraccaricare il sistema generando troppi processi figli.

Quando viene generato un nuovo processo, ci sono due possibilità per l'esecuzione:

- Il padre prosegue la sua esecuzione in concomitanza con il figlio
- Il padre aspetta fin quando alcuni o tutti i suoi figli hanno terminato (utilizzando rispettivamente le sys call **waitpid()** e **wait()**)

Ci sono inoltre due possibilità per l'address space del nuovo processo:

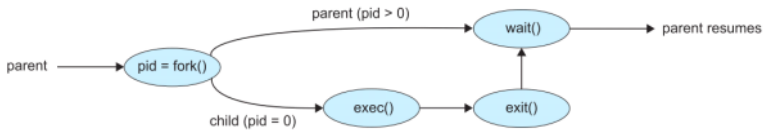
- Il processo figlio è un duplicato del padre (ha lo stesso programma e gli stessi dati del padre)
  - Ognuno avrà il proprio PCB, inclusi program counter, registri e pid
  - Su UNIX questo comportamento viene implementato tramite la sys call **fork()**
- Il processo figlio ha un nuovo programma nel suo address space (diverso da quello del padre)
  - Questo comportamento è implementato su Windows tramite la sys call **spawn()**
  - UNIX lo implementa come secondo passaggio della fork, utilizzando la sys call **exec()** che carica il file binario nella memoria, distruggendo l'immagine del vecchio programma

Analizziamo ora un programma in C implementato su un sistema operativo UNIX

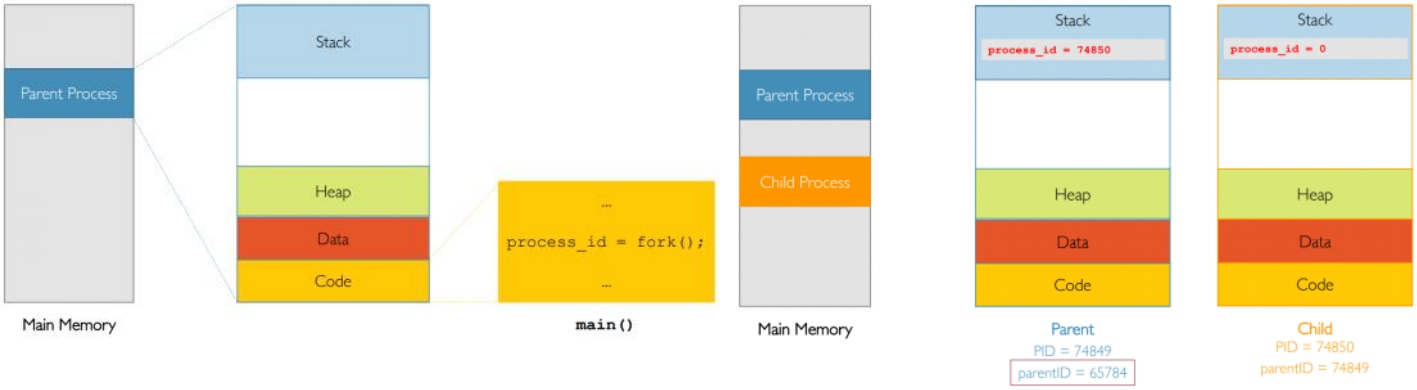
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

All'interno della funzione main viene inizializzata la variabile pid, la quale viene assegnata al valore di ritorno della fork. Viene così generato un secondo processo che esegue lo stesso programma del padre, con l'unica differenza che il padre avrà la variabile pid uguale al pid del figlio, mentre la variabile pid del processo figlio è uguale a 0. Il processo figlio riprenderà l'esecuzione del programma dalla fork che lo ha generato.
Viene fatto un controllo per escludere errori, in caso negativo, se pid è uguale a 0, quindi siamo nel processo figlio, viene eseguito l'execlp, che caricherà un nuovo programma nella memoria e il processo figlio inizierà ad eseguirlo (cessando l'esecuzione del vecchio programma).
Il processo padre ricadrà nell'else, in quanto la sua variabile pid è maggiore di 0, e attenderà la fine dell'esecuzione di tutti i processi figli.

L'esecuzione del programma viene riassunta con il seguente schema



Infine ecco cosa avviene in memoria principale alla chiamata della sys call fork()



Ricapitolando, come abbiamo potuto vedere i sistemi UNIX fanno uso di due system call per la creazione di nuovi processi. Generalmente all'avvio del sistema operativo UNIX, viene anche avviato un processo shell. Ogni comando che viene immesso nella shell crea un nuovo processo figlio, il cui padre è la shell stessa. UNIX inoltre permette di aggiungere il carattere **&** alla fine del comando, ordinando al sistema di eseguire il programma in concomitanza con la shell stessa.

System Call per la creazione di processi su UNIX:

- **fork**
  Genera un nuovo processo figlio con lo stesso programma del padre
- **execlp**
  Rimpiazza il programma del processo corrente con il nome del programma in input (execlp è una versione di exec)
- **wait / waitpid**
  Aspetta che tutti/alcuni programmi terminino l'esecuzione
- **sleep**
  Sospende l'esecuzione per un certo ammontare di secondi

TERMINAZIONE

Un processo termina quando finisce di eseguire la sua ultima istruzione, chiedendo all'OS di essere eliminato mediante la system call **exit()**. A questo punto il processo potrebbe ritornare un valore al padre (generalmente 0 se l'esecuzione è terminata senza errori, altri valori in caso di errore), ma solo se il processo padre sta aspettando la terminazione del figlio tramite l'utilizzo della system call **wait()**. Tale system call ritorna il pid del figlio, così che il padre possa sapere quale dei suoi figli è stato terminato.

Quando un processo termina, tutte le sue risorse vengono deallocate dall'OS. Ad ogni modo, la sua voce nella process table rimane fin quando il padre invoca wait(), perché la process table contiene lo stato di uscita del processo figlio.
I processi che hanno terminato, ma per i quali il padre non ha ancora invocato wait() vengono chiamati **processi zombie**. Generalmente questo status dura poco, una volta che il processo padre invoca wait(), il pid del processo zombie e la sua voce nella process table vengono rilasciati.
Se il processo padre termina la sua esecuzione non invocando mai wait(), il processo figlio che stava cercando di terminare, viene chiamato **processo orfano**. Tradizionalmente i sistemi UNIX gestiscono questi scenari **assegnando come padre al processo orfano il processo init** (o systemd in molti sistemi Linux). Init periodicamente invoca wait(), consentendo lo stato di exit a tutti i processi orfani.

I processi possono essere terminati dall'OS per vari motivi, come l'impossibilità del sistema di consegnare le risorse necessarie o in risposta di un comando **kill()**, o di un interrupt di processo non gestito.

Un processo può causare la terminazione di un altro processo tramite apposita system call. Generalmente **un processo può terminare solo un processo figlio** (altrimenti un programma utente potrebbe terminare arbitrariamente qualsiasi processo).

Un processo padre può eliminare un processo figlio per vari motivi:

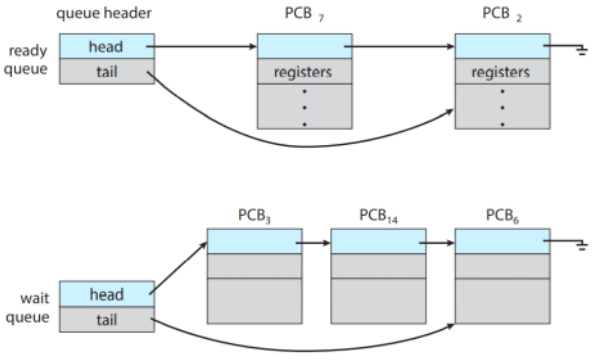
- Il figlio ha superato l'utilizzo di alcune risorse (il padre deve però avere un meccanismo per controllare lo stato del figlio)
- Il compito assegnato al figlio non è più richiesto
- Il padre ha terminato l'esecuzione del programma e il sistema operativo non permette al figlio di continuare se il processo padre ha terminato.

Come appena accennato, alcuni sistemi operativi non consentono ai processi figlio di esistere se i processi padre sono terminati. In tali sistemi operativi il fenomeno di terminazione di tutti i figli di un processo terminato, viene definito **cascading termination**.

SCHEDULING

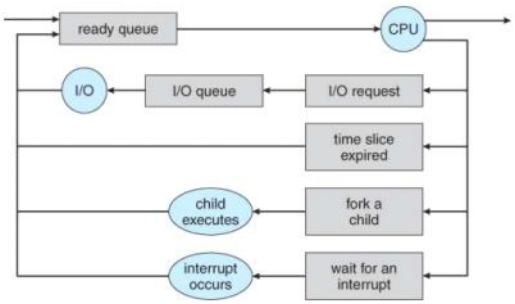
Il processo scheduler è responsabile del trasferimento di un processo da uno stato all'altro, con l'obbiettivo di **tenere sempre la CPU occupata**. L'obbiettivo del **time sharing** è di far passare continuamente i singoli core della CPU da un processo all'altro, così frequentemente che l'utente possa interagire con ogni programma con **tempi accettabili di risposta**. Questi obbiettivi vanno però in **conflitto**, in quanto ogni volta che lo scheduler interviene per scambiare i processi, viene impiegato tempo sulla CPU per farlo, tempo che viene quindi "perso" dal fare qualsiasi lavoro produttivo.

Lo scheduler dovrà quindi selezionare i processi liberi per l'esecuzione di un programma su un core (**ogni core può eseguire un solo processo per volta**). Se ci sono più processi che core, i processi in eccesso dovranno attendere fin quando un core si liberi e possano quindi essere rischedulati. L'OS mantiene i PCB di tutti i processi in stato di coda (c'è una coda per ognuno dei 5 stati). Tipicamente c'è una coda per ogni dispositivo I/O (dove i processi attendono che un dispositivo diventi accessibile per la lettura e la scrittura dei dati).



Quando un processo viene spostato da uno stato all'altro, il suo PCB viene scollegato dalla coda corrente e collegato in quella nuova. Il sistema operativo può usare diverse politiche per gestire ogni coda di stato.

La coda di esecuzione è vincolata dal numero di core disponibili sul sistema. Le altre code sono fondamentalmente illimitate, in quanto non vi è alcun limite teorico sul numero di processi.



La maggior parte dei processi possono essere descritti come **I/O-bound process** (processi legati principalmente alla comunicazione con i dispositivi I/O) o **CPU-bound process** (processi legati principalmente alla computazione).
Un sistema di scheduling efficiente dovrebbe selezionare il giusto mix di CPU-bound processes e I/O bound processes.

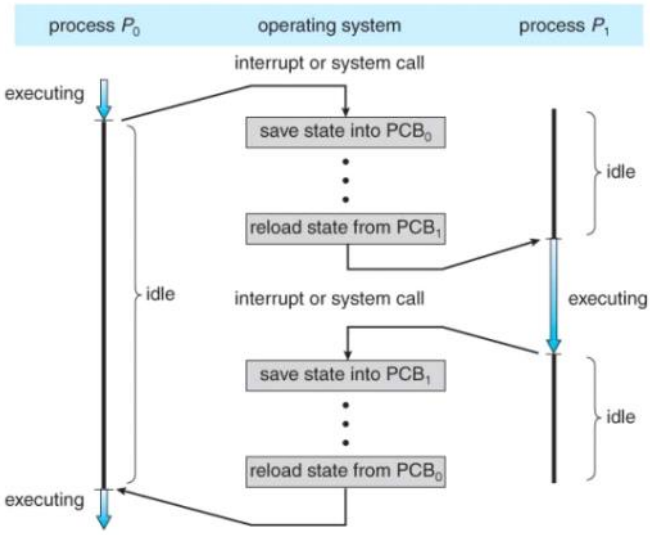
Abbiamo tre tipi principali di scheduler che possono essere implementati sulla macchina:

- **Long-term scheduler** → eseguito infrequentemente e tipico dei sistemi batch, o altri sistemi molto pesanti.
- **Medium-term scheduler** → quando i caricamenti del sistema diventano grandi, questo scheduler permette lavori più piccoli e veloci di finire rapidamente e pulire il sistema
- **Short-term scheduler** → eseguito molto frequentemente, deve sostituire molto rapidamente un processo all'interno della CPU con un altro.

Il **context switch** è la procedura usata dalla CPU per sospendere l'esecuzione di un processo al fine di avviarne un altro. È un'**operazione ad alto costo** in quanto bisogna salvare tutti i dati dello stato interno del processo bloccato (PC, SP, registri, ecc...) al suo PCB, successivamente bisogna caricare dal PCB del nuovo processo tutti i dati necessari alla sua esecuzione.
Il context switch viene generato a causa di una trap (system calls, exceptions o interrupts). Quando arriva una trap, la CPU deve:

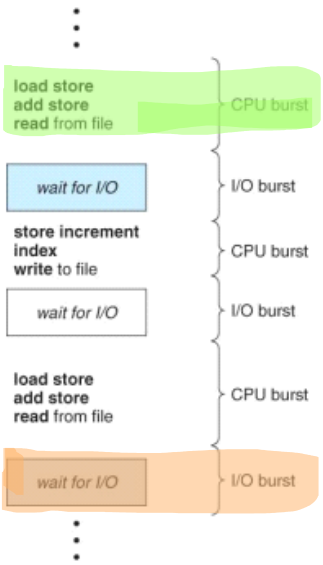
- Eseguire un salvataggio dello stato del processo attualmente in esecuzione
- Passare alla modalità kernel per gestire l'interrupt
- Eseguire un ripristino dello stato del processo interrotto

Gli I/O-bound process eventualmente subiscono context switch in caso di richieste di I/O. I CPU-bound process, invece, potrebbero non sollevare mai una richiesta di I/O. Per evitare quindi che i CPU-bound process intasino la CPU, i context switch vengono generati anche tramite HW con un timer interrupts.
Tra un context switch ed un altro può quindi passare al massimo un periodo di tempo limitato (per assicurare il **time sharing**), questo periodo di tempo viene detto **time slice** (o **time quantum**).
Bisogna trovare un compromesso nella lunghezza del time slice (time slice più corti aumentano la velocità di risposta, mentre time slice più lunghi riducono il tempo di inutilizzo della CPU).



Generalmente il time slice va dai 10 ai 100 ms, mentre un context switch richiede 10 µs.

1.6 CPU scheduling

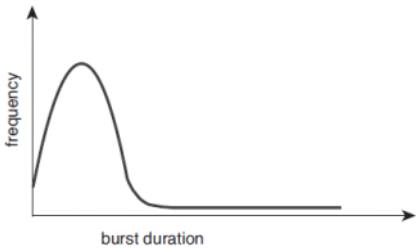


Per scheduling si intende l'**insieme delle Policy per stabilire quale processo eseguire sulla CPU**.

Un sistema di scheduling consente a un processo di utilizzare la CPU mentre un altro è in attesa di I/O, massimizzando così l'utilizzo del sistema (evitando quindi sprechi di tempo). L'obiettivo è quello di rendere il sistema più "efficiente" ed "equo" possibile.

L'esecuzione del processo inizia con un **burst della CPU**(esegue calcoli), questo è seguito da un **burst di I/O**(attesa del trasferimento dei dati all'interno o all'esterno del sistema), seguito da un altro burst della CPU, quindi da un altro burst di I/O e così via.

Le durate dei **burst della CPU variano notevolmente** da processo a processo e da computer a computer. Tendono ad avere una curva di frequenza simile a quella mostrata in figura.



Un programma legato all'**I/O** ha in genere molti **brevi burst** della CPU. Un programma legato alla CPU potrebbe avere alcuni **lunghi picchi di burst** della CPU.

PREEMPTIVE VS NON-PREEMPTIVE

Ogni volta che la CPU diventa inattiva, seleziona un altro processo dalla coda "ready" per l'esecuzione del prossimo.
Le **decisioni di scheduling** della CPU avvengono in una delle **4 condizioni**:

1. Quando un processo passa dallo stato di **running** allo stato di **waiting** (per una richiesta di I/O o un'invocazione di una chiamata di sistema di wait)
2. Quando un processo passa dallo stato di **running** allo stato **ready** (in risposta a un'interruzione)
3. Quando un processo passa dallo stato di **waiting** allo stato **ready** (dopo il completamento dell'I/O o un ritorno dalla wait)
4. Quando un processo viene **creato** o **terminato**

Per le situazioni 1 e 4, non c'è scelta in termini di scheduling. Un nuovo processo (se ne esiste uno nella coda "ready") deve essere selezionato per l'esecuzione perché la CPU deve sempre rimanere occupata. C'è scelta invece per le situazioni 2 e 3, perché la CPU può selezionare di nuovo lo stesso processo oppure ne può scegliere un altro.

Quando lo scheduling avviene nella condizione 1 e 4, diciamo che lo scheduling è **non-preemptive** o **cooperative**. Altrimenti diciamo che è **pre-emptive**.
Nello scheduling **non-preemptive**, una volta che la CPU è stata assegnata a un processo, il processo **mantiene la CPU fino a quando non termina** o passa allo stato di waiting come si può ben vedere nei punti 1 e 4.

Nello scheduling **preemptive**, la **CPU può lasciare un processo prima che termini** come si può vedere nei punti 2 e 3.

	Windows	Mac	UNIX-like
Non-preemptive	up to Win 3.x	up to Mac OS 9.x	-
Preemptive	since Win 95	since Mac OS X	since forever

Tutti i moderni sistemi operativi inclusi Windows, macOS, Linux e UNIX utilizzano algoritmi di scheduling preemptive.

Sfortunatamente, lo **scheduling preemptive** può portare a condizioni di **"race" quando i dati sono condivisi tra diversi processi**. Consideriamo il caso di due processi che condividono i dati. Mentre un processo sta aggiornando i dati, viene **preempted (anticipato)** in modo che il secondo processo possa essere eseguito. Il secondo processo tenta quindi di leggere i dati, che però si trovano in uno **stato incoerente**.

Inoltre questo tipo di scheduling, influisce anche sulla progettazione del kernel del sistema operativo. Durante l'elaborazione di una system call, il kernel può essere occupato con un'attività per conto di un processo. Tali attività possono comportare la **modifica** di dati importanti del kernel (ad esempio, le code di I/O). Cosa succede se il processo viene anticipato nel mezzo di questi cambiamenti e il kernel (o il driver del dispositivo) deve leggere o modificare la stessa struttura? il caos.

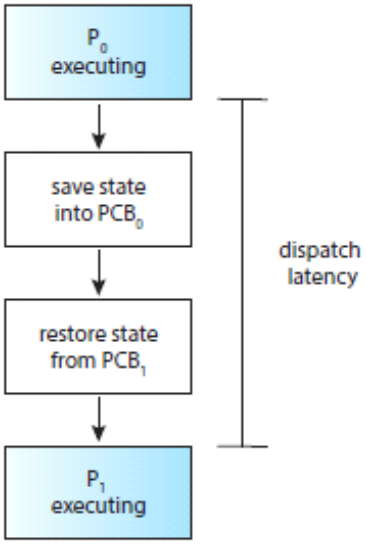
Un kernel **non-preemptive** attenderà il completamento di una chiamata di sistema o per il blocco di un processo durante l'attesa del completamento dell'I/O prima di eseguire un **context switch**. Questo schema assicura che la struttura del kernel **sia semplice**, poiché il kernel **non anticiperà** un processo mentre le strutture dati del kernel sono in uno **stato incoerente**. Sfortunatamente, questo modello di esecuzione del kernel non è adatto per supportare l'elaborazione in tempo reale(real-time computing), in cui le attività devono completare l'esecuzione entro un determinato lasso di tempo.

Un kernel **preemptive** richiede meccanismi come i mutex locks per prevenire condizioni di "race" (competizione) durante l'accesso a strutture di dati condivise del kernel. La maggior parte dei sistemi operativi moderni sono **completamente preemptive** quando viene eseguita la **modalità kernel**.

Poiché gli interrupt possono, per definizione, verificarsi in qualsiasi momento e poiché non possono sempre essere ignorati dal kernel, le **sezioni di codice interessate dagli interrupt devono essere protette dall'uso simultaneo**. Per fare in modo che queste sezioni di codice non siano accessibili contemporaneamente da più processi, vengono **disabilitati gli interrupt all'inizio** e vengono **riattivati alla fine**.

È **importante notare** che le sezioni di codice che disabilitano gli interrupt **non si verificano molto spesso** e in genere contengono **poche istruzioni**.

IL DISPATCHER



È il modulo che **dà il controllo della CPU al processo** selezionato dallo scheduler. Le sue funzioni includono:

- Switching context
- Passaggio alla user mode
- Salto alla locazione corretta nel programma appena caricato

Il **dispatcher** dovrebbe essere **il più veloce possibile**, poiché viene **richiamato durante ogni context switch**. Il tempo impiegato dal dispatcher per arrestare un processo e avviarne un altro in esecuzione è noto come dispatch latency.

A livello di sistema, **il numero di context switch** può essere ottenuto utilizzando il comando `vmstat` disponibile sui sistemi Linux.
Possiamo anche utilizzare il file system `/proc` per determinare il **numero di context switch per un dato processo**. Per esempio il contenuto del file `/proc/2166/stato`, dà come output il numero di context switch (volontari e non volontari) nel corso della durata del processi.

DEFINIZIONI UTILI

- **Arrival Time**: momento in cui il processo arriva nella coda pronta
- **Completion Time**: tempo impiegato da un processo per completare la sua esecuzione
- **Burst Time**: tempo richiesto da un processo alla CPU per l'esecuzione
- **Turnaround Time**: differenza di tempo tra il Completion Time e l'Arrival Time
- **Waiting Time**: differenza di tempo tra il Turnaround Time e il Burst Time

CRITERI/METRICHE DI SCHEDULING

Esistono diversi criteri da considerare quando si tenta di selezionare l'algoritmo di scheduling "migliore", tra cui:

- **CPU utilization**

  Intuitivamente, la percentuale di tempo in cui la CPU è occupata. Idealmente la CPU dovrebbe essere occupata il 100% del tempo. Su un sistema reale l'utilizzo della CPU dovrebbe variare dal 40% (lightly loaded) al 90% (heavily loaded).

- **Throughput**

  Il numero di processi completati in un'unità di tempo. Può variare da 10 al secondo a 1 all'ora.

- **Tournaround time**

  Tempo necessario per il completamento di un particolare processo, dall'avvio al completamento. Include tutto il tempo di attesa (waiting time).

- **Waiting time**

  Quanto tempo i processi trascorrono nella coda "ready" aspettando il proprio turno per andare nella CPU. Da **non confondere** con lo stato di attesa!. I processi nello stato di attesa non sono sotto il controllo dello scheduler della CPU (semplicemente non sono pronti per essere eseguiti).

- **Response time**

  Tempo che intercorre tra la presentazione di una richiesta e la produzione della prima risposta. È il tempo necessario per iniziare a rispondere, non il tempo necessario per emettere la risposta.

Idealmente, bisognerebbe scegliere uno scheduler che **ottimizzi** tutte le metriche contemporaneamente. Generalmente è **impossibile** ed è necessario un compromesso! L'idea è quella di scegliere un algoritmo di scheduling che soddisfi una determinata **policy**.

POLITICHE DI SCHEDULING

- **Ridurre al minimo il tempo medio di risposta (average response time)**
  Fornire l'output all'utente il più rapidamente possibile.
- **Ridurre al minimo il tempo massimo di risposta(maximum response time)**
  Legato al caso peggiore.
- **Ridurre al minimo la varianza del tempo di risposta(variance of response time)**
  È più importante minimizzare la varianza nel tempo di risposta piuttosto che minimizzare il tempo medio di risposta.
  Un sistema con tempi di risposta ragionevoli è più desiderato rispetto a un sistema mediamente più veloce ma molto variabile.
- **Massimizzare il throughput**
  Riduzione al minimo dell'overhead (context switching del sistema operativo).
- **Ridurre al minimo il waiting time**
  Assegna a ogni processo la stessa quantità di tempo sulla CPU. Potrebbe aumentare il tempo medio di risposta .

Sebbene la maggior parte delle architetture CPU moderne disponga di più core di elaborazione, descriviamo i prossimi algoritmi di scheduling nel contesto di un solo core di elaborazione disponibile, cioè una singola CPU che ha un singolo core di elaborazione (il sistema è in grado di eseguire un solo processo alla volta).

1.7 Scheduling algorithm: First-Come-First-Serve

Iniziamo con l'algoritmo di scheduling concettualmente più semplice.
Il **first-come-first-serve** si basa sul concetto che il processo che richiede prima la CPU, viene eseguito per primo.
L'implementazione della politica dell'**FCFS** è facilmente gestita con una coda **FIFO**. Quando un processo entra nella **ready queue**, il suo **PCB** viene legato all'estremità della coda. Quando la CPU è libera viene eseguito il processo alla testa della coda (il primo ad aver richiesto la CPU), tale processo viene poi rimosso dalla coda.

ESEMPIO

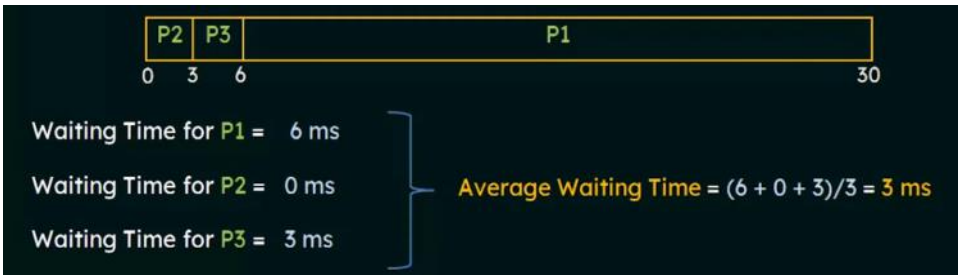
Consideriamo il seguente set di processi arrivati tutti al momento 0

Process	Burst Time (ms)
P1	24
P2	3
P3	3

Se i processi arrivano nell'ordine P1, P2, P3 la CPU li eseguirà esattamente nell'ordine di arrivo

In questo caso i processi avranno i seguenti tempi di attesa:

Possiamo però facilmente notare come, solo cambiando l'ordine di esecuzione dei processi, l'Average Waiting Time si possa drasticamente ridurre

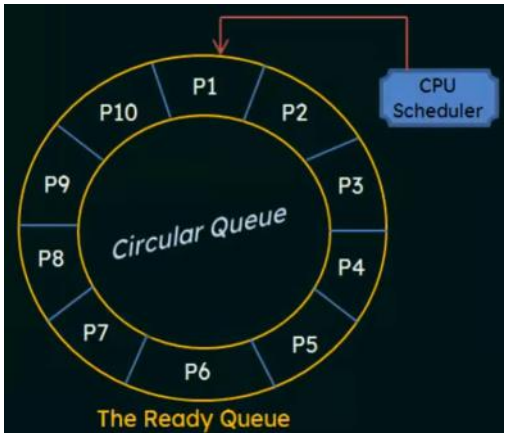


Notiamo quindi le seguenti formule:

$$\boldsymbol{Turn\ Around\ Time = Completion\ time - Arrival\ time}$$
$$\boldsymbol{Waiting\ Time = Turn\ Around\ Time - Burst\ time}$$

L'**FCFS è nonpreemptive**, ciò significa che quando un processo entra nella CPU quest'ultima resterà occupata fino alla terminazione del processo. Ciò causa particolari problemi ai sistemi che usano il time-sharing, dove è importante che la CPU faccia esegua costantemente il context switch.

1.8 Scheduling algorithm: Round-Robin



Il **round-robin** è progettato specialmente per i sistemi time-sharing, è simile all'FCFS, con l'unica differenza che l'**RR è preemptive**, consentendo così di passare da un processo all'altro anche se il processo in corso non è ancora terminato. Viene utilizzato un time slice (già trattato qui), generalmente compreso tra i 10 e i 100 ms, che attiva il context switch ad ogni scadere del tempo, facendo passare in esecuzione il processo successivo. Nel round-robin la coda di esecuzione è trattata come una **coda circolare**.

Una volta che il processo entra nella CPU, possono accadere 2 cose:

- Il processo ha un **CPU burst minore** del time slice
  - Il processo, una volta terminato, libererà la CPU prima dello scadere del tempo
  - Lo scheduler procederà quindi al prossimo processo in coda
- Il processo ha un **CPU burst maggiore** del time slice
  - Allo scadere del timer verrà generato un interrupt
  - Verrà eseguito un context switch con il processo successivo mentre il processo corrente verrà messo alla fine della coda

Consideriamo ad esempio un time slice di 4 ms, con i processi P1, P2, P3 che arrivano in sequenza

Process ID	Burst Time
P1	24
P2	3
P3	3

Gantt Chart:

0	4	7	10	14	18	22	26	30
P1	P2	P3	P1	P1	P1	P1	P1	

Method 1

Turn Around time = Completion time - Arrival time

Waiting time = Turn Around time - Burst time

Process ID	Completion Time	Turnaround Time	Waiting Time
P1	30	30 - 0 = 30	30 - 24 = 6
P2	7	7 - 0 = 7	7 - 3 = 4
P3	10	10 - 0 = 10	10 - 3 = 7

Method 2

Waiting time = Last Start Time - Arrival Time - (Preemption x Time Quantum)

Process ID	Waiting Time
P1	26 - 0 - (5x4) = 6
P2	4 - 0 - (0x4) = 4
P3	7 - 0 - (0x4) = 7

1.9 Scheduling algorithm: Shortest-Job-First

Questo algoritmo associa ad ogni processo la lunghezza del proprio CPU burst corrente. Quando la CPU è libera le viene assegnato il processo con il minor CPU burst corrente. Se più processi hanno lo stesso CPU burst, viene utilizzato l'algoritmo FCFS.
L'SJF è sia preemptive che non preemptive.

Consideriamo il seguente esempio

Process ID	Arrival Time	Burst Time
P1	0	~~8~~ 7
P2	1	4
P3	2	9
P4	3	5

Gantt Chart:

0	1	5	10	17	26
P1	P2	P4	P1	P3	

Waiting Time for P1 = (10 - 1 - 0) = 9 ms

Waiting Time for P2 = (1 - 0 - 1) = 0 ms

Waiting Time for P3 = (17 - 0 - 2) = 15 ms

Waiting Time for P4 = (5 - 0 - 3) = 2 ms

Average Waiting Time

= (9 + 0 + 15 + 2)/4 = 6.5 ms

Possiamo notare che il Waiting Time si può calcolare con la seguente formula

$$Waiting\ Time = Total\ Waiting\ Time\ - num.di\ ms\ eseguiti\ dal\ processo\ - Arrival\ Time$$

1.10 Scheduling algorithm: Priority-Scheduling

L'algoritmo **SJF è un caso speciale dell'algoritmo di Priority-Scheduling**. A ogni processo è associata una priorità e la CPU è assegnata a un processo che ha la massima priorità. Le **priorità possono essere definite internamente o esternamente**. Le priorità definite **internamente** utilizzano una o più quantità misurabili per calcolare la priorità di un processo. Ad esempio, i limiti di tempo, i requisiti di memoria, il numero di i file aperti ecc. Le priorità **esterne** sono stabilite da criteri al di fuori del sistema operativo, come l'importanza del processo, il tipo e l'importo dei fondi pagati per l'utilizzo del computer ed altri fattori.

I **processi con pari priorità** sono schedulati con l'algoritmo **FCFS**. Un algoritmo SJF è semplicemente un algoritmo di priorità in cui la priorità (p) è l'inverso del prossimo CPU burst (previsto). Maggiore è il burst della CPU, minore è la priorità e viceversa. Il Priority-scheduling può essere sia **preemptive** che **non-preemptive**. Un algoritmo di **preemptive priority scheduling** anticiperà(preempt) la CPU se la priorità del nuovo processo arrivato è più alta del processo attuale. Un algoritmo di **non preemptive priority scheduling**, metterà semplicemente il nuovo processo in testa alla coda "ready".

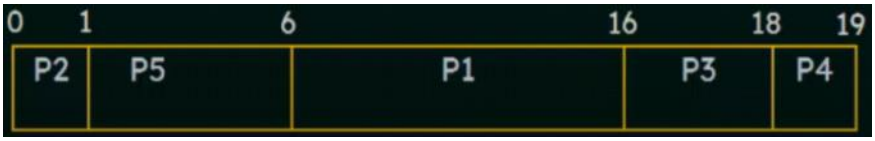
ESEMPIO

Consideriamo i seguenti processi e assumiamo che arrivino tutti **al tempo 0**,nell'ordine P1, P2, P3, P4, P5 con l'unità di misura del CPU burst in **millisecondi**:

Process ID	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

***Ha priorità maggiore chi ha il numero più basso nella colonna "Priority"**

scheduleremo questi processi in base al seguente **diagramma di Gantt**:



Il **waiting time** di ogni processo è facile da ricavare, infatti avremo:

- Waiting time per **P1 = 6ms**
- Waiting time per **P2 = 0ms**
- Waiting time per **P3 = 16ms**
- Waiting time per **P2 = 18ms**
- Waiting time per **P2 = 1ms**

Da qui possiamo ricavarci l'**average waiting time** che è semplicemente la **media dei waiting time** dei singoli processi:

$$Average\ waiting\ time = (6 + 0 + 16 + 18 + 1) / 5 = 8,2\ ms$$

PROBLEMA DEL PRIORITY SCHEDULING

Un grosso problema con questo algoritmo è "**l'indefinite blocking**" o "**starvation**", ovvero quando un processo con **bassa priorità**, pronto per la CPU, può attendere di essere eseguito **indefinitamente**. Un processo pronto per essere eseguito ma in attesa della CPU può essere considerato **bloccato**.

SOLUZIONE AL PROBLEMA

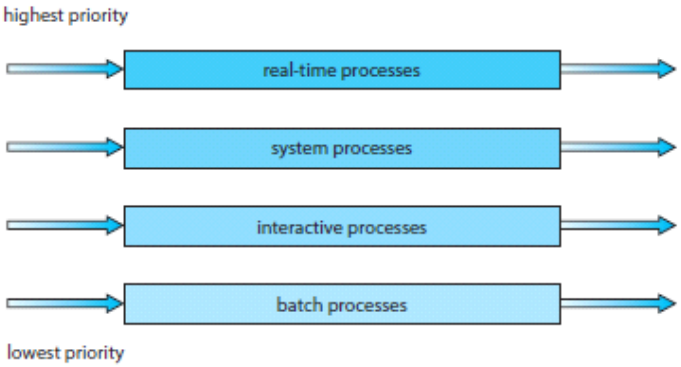
Una soluzione al problema dell'**indefinite blocking** dei processi a bassa priorità è l'**aging(invecchiamento)**. L'aging comporta l'**aumento graduale della priorità** dei processi che attendono nel sistema per lungo tempo. Ad esempio, se le priorità vanno da 127 (basso) a 0 (alto), potremmo periodicamente (diciamo ogni secondo) aumentare la priorità di un processo in attesa di 1. Alla fine, anche un processo con una priorità iniziale di 127 avrebbe la **massima priorità** nel sistema e verrebbe eseguito.

1.11 Scheduling algorithm: Multilevel-Queue

Con il priority e il round-robin scheduling, tutti i processi possono essere inseriti in un'unica coda e lo scheduler seleziona quindi il processo da eseguire con la priorità più alta. Un **Multilevel Queue Scheduling** può anche essere utilizzato per **suddividere i processi in diverse code separate** in base al tipo di processo.

ESEMPIO

Ad esempio, viene effettuata una divisione comune tra **foreground (interattivo) processes**, ovvero i processi in primo piano, e **background (batch) processes**, ovvero i processi in background. Questi due tipi di processi hanno requisiti di tempo di risposta diversi e quindi possono avere esigenze di scheduling diverse. È possibile utilizzare **code separate** per i processi in primo piano e in background e **ciascuna coda potrebbe avere il proprio algoritmo** di scheduling.



Inoltre, deve esserci uno **scheduling tra le code**, che è comunemente implementata con un **fixed-priority preemptive scheduling**. Ad esempio, la coda dei **real time processes** può avere priorità assoluta sulla coda degli **interactive processes**.

Ora prendiamo in esempio l'immagine qui di fianco: Ogni coda ha **priorità assoluta** rispetto alle code con priorità inferiore. Nessun processo nella coda batch processes, ad esempio, potrebbe essere eseguito a meno che tutte le altre code sopra (con priorità superiore) non siano tutte **vuote**. Se un interactive process entra nella coda dei processi pronti mentre è in esecuzione un batch process, il batch process verrebbe **preempted**(**anticipato**) per eseguire l'interactive process che ha priorità più alta.

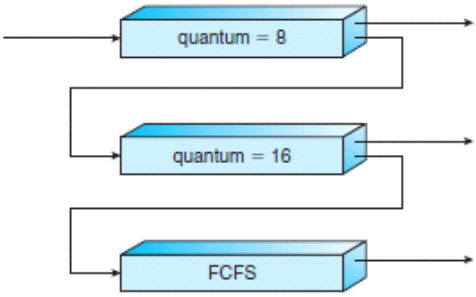
Un'altra possibile schedulazione tra le code è il **time-slicing**. Qui, ogni coda ottiene una certa porzione del tempo della CPU, che può quindi gestire tra i suoi vari processi.

1.12 Scheduling algorithm: Multilevel-Feedback-Queue

Il **Multilevel Feedback Queue** può essere considerato un evoluzione del MLQ, in quanto consente a un processo di **spostarsi** tra le code. L'idea è di separare i processi secondo le caratteristiche dei loro **CPU burst**. Se un processo utilizza troppo tempo della CPU, verrà **spostato** in una **coda con priorità inferiore**. Inoltre, un processo che attende troppo a lungo in una coda con priorità inferiore può essere **spostato** in una **coda con priorità più alta**. Questa forma di **aging** previene la **starvation**.

ESEMPIO

Ad esempio, si consideri uno scheduler di code di feedback multilivello con tre code, numerate da 0 a 2 (dall'alto verso il basso) come in figura. Lo scheduler esegue prima tutti i processi nella **coda 0**. Solo quando la coda 0 **è vuota** eseguirà i processi nella **coda 1**. Allo stesso modo, i processi nella coda 2 verranno eseguiti solo se le code 0 e 1 **sono vuote** e così via. A un processo nella coda 0 viene assegnato un **time quantum** di 8 millisecondi. Se non termina entro questo tempo, viene spostato in coda nella coda 1.



Se la coda 0 è vuota, al processo in testa alla coda 1 è dato un **time quantum** di 16 millisecondi. Se non viene completato, viene **anticipato(preempted)** e viene inserito nella coda 2. I processi nella coda 2 vengono eseguiti su base FCFS ma vengono eseguiti solo quando le code 0 e 1 **sono vuote**.

In generale, uno scheduler multilevel feedback queue **è definito dai seguenti parametri**:

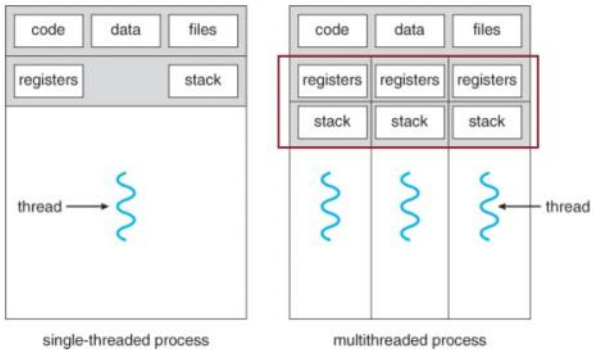
- Il numero di code
- L'algoritmo di scheduling per ogni coda
- Il metodo per determinare quando aggiornare un processo a una coda con priorità più alta/bassa
- Il metodo per determinare in quale coda un processo entrerà in base al servizio di cui necessita

1.13 Thread

Prima di dare una definizione di thread, va chiarito un punto. I processi (o **heavyweight process**) sono istanze di un programma caricate in memoria composte da spazio di indirizzamento, codice da eseguire, dati, file, risorse etc. I processi però **non eseguono direttamente il codice**, questo compito **spetta** invece **ai thread**.

Un **thread** (o **lightweight process**, da non confondersi con l'LWP di cui si parlerà più avanti) è un **flusso di esecuzione indipendente** all'interno di un processo.

I processi tradizionali hanno un solo thread di controllo (definiti **single-thread**), è presente quindi un solo PC e una sola sequenza di istruzioni che possono essere effettuate in un dato momento.
Un processo **può avere più thread** che vengono eseguiti concorrentemente all'interno della CPU; il numero di thread che possono essere eseguiti contemporaneamente all'interno della CPU **dipende dal numero di core** presenti.



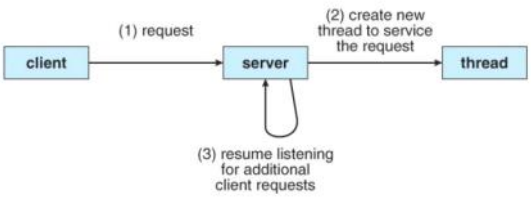
I thread di un processo condividono tra loro spazio di indirizzamento, sezione di codice, dati, file e altre risorse.
Non sono richieste system calls ai thread per comunicare tra di loro (in quanto condividono già lo stesso spazio di memoria), ciò rende la comunicazione **inter-thread** molto più semplice della comunicazione **inter-process**. Questo implica che anche i context-switch saranno **molto più rapidi tra i thread** piuttosto che tra i processi (sia per la quantità di informazioni molto ridotta, che per l'agevole comunicazione tra i thread).

Ogni thread è inoltre dotato di un **TCB (thread control block)**, che contiene specifiche informazioni per il funzionamento del thread, come ad esempio:

- **Thread identifier** (*tid*) → id unico assegnato ad ogni thread
- **Stack pointer** → punta allo stack del thread nel processo
- **Program counter** → punta all'istruzione corrente del programma eseguita dal thread
- **Stato del thread** (running, ready, waiting, new, terminated)
- **Registri associati**
- **Puntatore al PCB** del processo contenitore

Nella programmazione moderna i thread sono estremamente utili dovunque siano presenti **attività multiple** che possono essere eseguite indipendentemente dalle altre. Ciò è particolarmente vero quando un'attività potrebbe interrompersi e si desidera di continuare l'esecuzione delle altre **senza doverle interrompere**.

Thread multipli consentono a multiple richieste **di essere soddisfatte contemporaneamente**, senza esser costretti a gestire le richieste sequenzialmente, o senza dover usare una `fork` per creare un nuovo processo per ogni richiesta in arrivo.

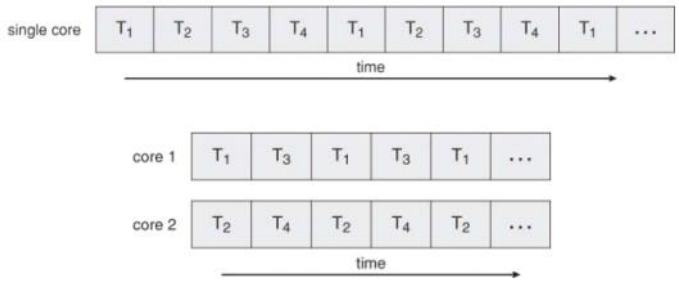


Ricapitoliamo quindi i **4 principali benefici della programmazione multi-thread**:

- **Reattività** → un thread può fornire risposte rapide mentre altri thread sono bloccati o rallentati da calcoli complessi
- **Condivisione delle risorse** → i thread condividono codice, dati e spazio di indirizzo
- **Economia** → creare e gestire threads (e i context-switch tra di loro) è molto più veloce che eseguire le stesse operazioni sui processi
- **Scalabilità** → nelle architetture multi-core, un processo single-thread può essere eseguito su un singolo core, mentre un processo multi-thread può essere diviso tra tutti i processori/core disponibili

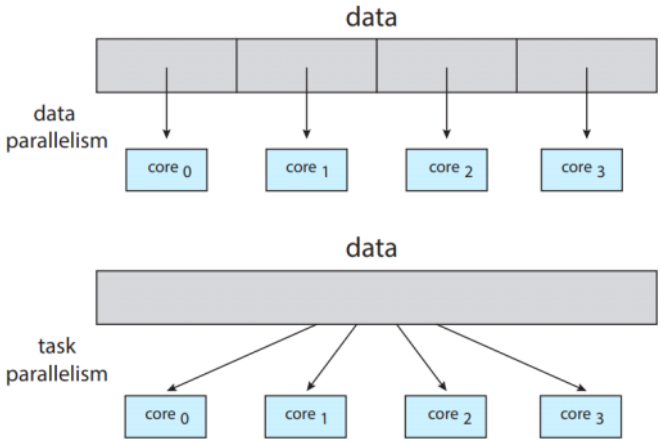
PROGRAMMAZIONE MULTI-CORE

Se avviassimo un'applicazione multi-thread su un processore single-core, non ne avremmo un grande beneficio in quanto i thread verrebbero **eseguiti sequenzialmente**. La programmazione multi-thread acquista di senso solamente quando applicata su processori multi-core, dove i thread vengono **divisi e eseguiti in tutti i core liberi**, **consentendo un reale parallelismo**.



I processori multi-core richiedono nuovi algoritmi di scheduling, per fare un uso migliore dei multipli core liberi.

I recenti processori sono sviluppati per supportare più thread simultanei per core nell'hardware, ad esempi con l'[hyper-threading] di Intel (dove ogni core fisico **appare come due processori** al sistema operativo, consentendo lo scheduling concorrente di **due thread per core fisico**).



In generale abbiamo due tipi di parallelismo:

- **Data parallelism** - Si concentra sulla distribuzione di sottoinsiemi degli stessi dati tra tutti i core liberi, eseguendo la stessa operazione su dati diversi.

  Ad esempio, se volessimo sommare il contenuto di un array di N elementi, su un sistema single-core un thread semplicemente sommerebbe i valori da [0] a [N - 1]. In un sistema dual-core, un thread A, in esecuzione nel core 0, sommerebbe i valori da [0] a [N/2 - 1], mentre il thread B, in esecuzione nel core 1, sommerebbe i valori da [N/2] a [N - 1].

- **Task parallelism** - Implica la distribuzione non di dati ma di compiti (threads) tra i multipli core. Ogni thread esegue un'operazione unica e indipendente

Ad ogni modo, data e task parallelism **non sono mutualmente esclusivi**, un'applicazione può infatti usare un ibrido di queste due strategie.

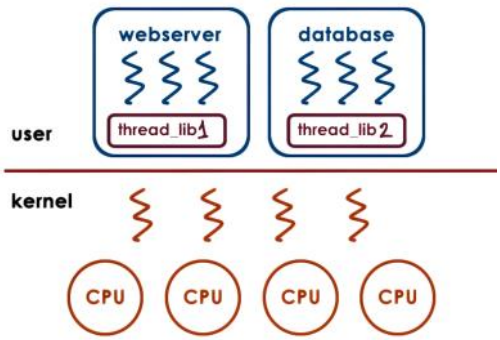
I thread possono essere supportarti in 2 modi:

- Livello kernel → **kernel thread**

  Gestiti direttamente dal kernel del sistema operativo

- Livello utente → **user thread**

  Gestiti nello spazio utente dalle **user-level thread library**, senza l'intervento dell'OS



KERNEL THREADS

È la **più piccola unità di esecuzione** che può essere schedulata dal sistema operativo: **la gestione dei thread viene eseguita dal kernel** (generalmente l'OS fornisce system calls per la gestione e la creazione di questi thread dallo spazio utente). Più kernel thread dello stesso processo possono essere schedulati su core diversi. I thread del kernel **sono in genere più lenti** da creare e gestire rispetto ai thread utente. Quando un kernel thread si blocca, il kernel può schedularne un altro per lo stesso processo.

USER THREADS

I thread a livello utente **sono piccoli e molto più veloci** dei thread a livello di kernel. Sono supportati direttamente dal sistema operativo. I thread a livello utente vengono implementati dagli utenti e **il kernel non è a conoscenza** dell'esistenza di questi thread, li gestisce come se fossero processi a thread singolo. Sono gestiti interamente dal **run-time system** (user-level thread library, che possono cambiare da un processo all'altro).
Gli user thread non possono beneficiare del multiprocessing, inoltre se uno user thread effettua un'operazione bloccante **l'intero processo viene fermato**.

1.14 Modelli multi-thread

Gli user-thread devono essere mappati ai kernel thread in uno dei seguenti modi:

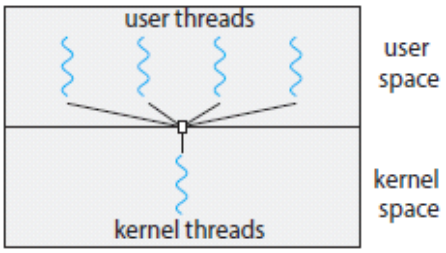
- **Many-to-One**
- **One-to-One**
- **Many-to-Many**
- **Two-level**

MANY TO ONE MODEL

Il modello **many to one** associa **molti user thread a un solo kernel thread**. La gestione dei thread viene eseguita dalla libreria dei thread nello user-space, quindi è efficiente. Tuttavia, l'intero processo si bloccherà se uno user thread effettua una chiamata di sistema bloccante.

Inoltre, poiché solo un thread alla volta può accedere al kernel, su sistemi multicore **non possono essere eseguiti più user thread in parallelo**.

Tuttavia, pochissimi sistemi continuano a utilizzare questo modello a causa della sua incapacità di sfruttare più core di elaborazione, che ora sono diventati standard sulla maggior parte dei sistemi informatici

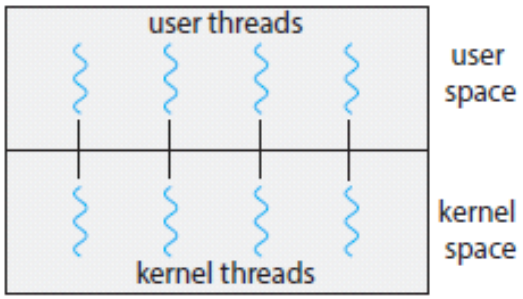


ONE TO ONE MODEL

Il modello **one-to-one** associa **ogni user thread a singoli kernel thread** (rapporto 1:1). Fornisce più concorrenza rispetto al modello many to one **consentendo l'esecuzione di un altro user thread** quando uno user thread effettua una system call bloccante. Consente inoltre l'**esecuzione in parallelo di più user thread** su sistemi multi-core.

L'unico inconveniente di questo modello è che la creazione di uno user thread richiede la creazione del kernel thread corrispondente, e un numero elevato di kernel thread può gravare sulle prestazioni del sistema. Per ovviare a questo, la maggior parte delle implementazioni di questo modello pone un **limite al numero di thread** che possono essere creati.

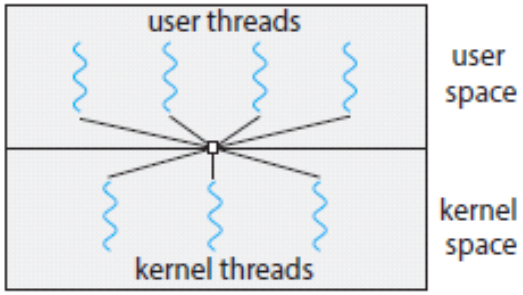
Linux, insieme alla famiglia di sistemi operativi Windows, implementa il modello one-to-one.



MANY TO MANY MODEL

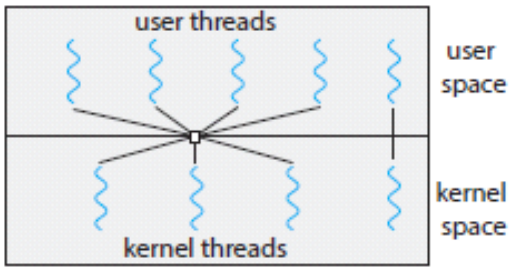
Nel modello **many to many, molti user thread vengono associati ad un numero inferiore o uguale di kernel thread** tramite un'unica via di comunicazione. Il numero di kernel thread dipende dall'uso specifico di una particolare applicazione e da quanti core ha la macchina.

Gli sviluppatori possono creare tutti gli user thread necessari e i corrispondenti kernel thread possono essere eseguiti in **parallelo** su un multiprocessore. Inoltre, quando un thread esegue una chiamata di sistema bloccante, il kernel **può schedulare l'esecuzione di un altro thread**.



TWO LEVEL MODEL

Il modello **two level**, è una variante del modello many to many. Associa gli user thread ad un numero inferiore o uguale di kernel thread allo stesso modo di come avviene nel many to many model, **ma consente anche** di associare uno user thread ad un kernel thread, come si può ben notare dall'immagine qui di fianco.



1.15 Thread libraries

Una **thread library**, fornisce ai programmatori un'**API per la creazione e la gestione dei thread**.
Esistono due modi principali per implementare una thread library:

1. Il primo approccio consiste nel fornire una libreria **interamente nello user space senza sostegno del kernel**. Tutto il codice e le strutture dati per la libreria **esistono nello user space**. Ciò significa che l'invocazione di una funzione nella libreria **risulta in una chiamata di funzione locale nello user space e non in una system call**.
2. Il secondo approccio consiste nell'implementare una **libreria a livello di kernel** supportata direttamente dal sistema operativo. In questo caso, il codice e le strutture dati per la libreria **esistono nel kernel space**. Invocare una funzione nell'API per la libreria in genere **si traduce in una system call**.

Ci sono 3 thread library principali in uso oggi:

- **POSIX Phtreads**: può essere fornito come libreria utente o kernel, come estensione dello standard POSIX
- **Win32 threads**: fornito come libreria a livello di kernel su sistemi Windows
- **Java threads**: l'implementazione dei thread si basa su qualsiasi sistema operativo e hardware su cui è in esecuzione la JVM, ad esempio Pthread o Win32

PTHREADS: ESEMPIO DI SOMMA

Pthreads si riferisce allo **standard POSIX (IEEE 1003.1c**) che definisce un'API per la creazione e la sincronizzazione dei thread.

Questa è una specifica per il comportamento del thread, **non un'implementazione**. I progettisti di sistemi operativi possono implementare la specifica in qualsiasi modo desiderino.

Il programma C mostrato nella figura mostra l'API Pthreads di base per la costruzione di un **programma multi-thread** che calcola, in un thread separato, le somme che vanno da 1 a un numero intero non negativo fornito in input.
In un programma **Pthreads**, i thread separati iniziano l'esecuzione in una funzione specificata, questa è la funzione `runner()`.

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

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Quando questo programma inizia, il thread di controllo inizia nel `main()`. Dopo qualche inizializzazione, `main()` crea un secondo thread che prende il controllo della funzione `runner()`. Entrambi i thread **condividono il dato globale** sum.

Tutti i programmi Pthreads **devono includere il file di intestazione** `pthread.h`. L'istruzione `pthread_t tid` dichiara l'identificatore per il thread che creeremo. Ogni thread ha una serie di attributi, tra cui la dimensione dello stack e le informazioni di scheduling. La dichiarazione `pthread_attr_tattr` rappresenta gli attributi per il thread.

Impostiamo gli attributi nella chiamata di funzione `pthread_attr_init(&attr)`. Poiché non abbiamo impostato esplicitamente alcun attributo, utilizziamo gli attributi predefiniti forniti. In seguito viene creato un thread separato con la chiamata alla funzione `pthread_create()`. Oltre a passare l'identificatore del thread e gli attributi per il thread, passiamo anche il nome della funzione in cui inizierà l'esecuzione del nuovo thread, in questo caso la funzione `runner()`. Infine, passiamo il parametro intero che viene fornito sulla riga di comando, `argv[1]`. A questo punto, il programma ha due thread: il thread iniziale (o genitore) in `main()` e il thread figlio che esegue l'operazione di sommatoria nella funzione `runner()`.

Questo programma segue la strategia thread create/join, per cui dopo aver creato il thread figlio, il thread genitore attenderà il termine del thread figlio chiamando la funzione `pthread_join()`. Il thread figlio terminerà dopo aver chiamato la funzione `pthread_exit()`. Una volta che il thread figlio ha eseguito il `return`, il thread padre restituirà il valore di somma condiviso con la funzione `printf()`.

Un metodo semplice per attendere su più thread utilizzando la funzione `pthread_join()`, come mostrato nell'immagine di fianco, consiste nel racchiudere l'operazione all'interno di un semplice ciclo for.

```c
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

JAVA THREADS

I **thread in Java** possono essere creati in 2 modi:

- Estendendo la classe thread
- implementando l'interfaccia Runnable

Entrambe le soluzioni richiedono l'override del metodo run()
NOTA: Java non supporta l'ereditarietà multipla! Se la tua classe estende la classe Thread, non può estendere nessun'altra classe. In una situazione del genere, **è preferibile implementare Runnable**

1.16 Thread pool

Creare nuovi thread piuttosto che nuovi processi è sicuramente meno costoso. Tuttavia, potrebbero verificarsi alcuni problemi importanti:

- Il primo problema riguarda la **quantità di tempo necessaria** per creare il thread, insieme al fatto che il thread verrà scartato una volta completato il suo lavoro.
- La seconda questione è più problematica. Se consentiamo a ogni richiesta di essere eseguita su un nuovo thread, non abbiamo posto un limite al numero di thread che possono rimanere attivi contemporaneamente nel sistema. I thread illimitati potrebbero **esaurire le risorse di sistema**, come il tempo della CPU o la memoria. Una soluzione a questo problema consiste nell'utilizzare un **pool di thread**.

L'idea generale alla base di un pool di thread è creare un certo numero di thread all'avvio e inserirli in un **pool**, dove si "siedono" e aspettano il lavoro.

Quando ad esempio un server riceve una richiesta, anziché creare un thread, **invia una richiesta al pool** di thread e riprende il lavoro in attesa di ulteriori richieste. Se c'è un thread disponibile nel pool, **viene risvegliato** e la richiesta viene gestita immediatamente. Se il pool non contiene thread disponibili, l'attività **viene accodata** fino a quando ne diventa libero uno. Una volta che un thread ha completato il suo servizio, **ritorna nel pool e attende** altro lavoro. I pool di thread funzionano bene quando le attività inviate al pool possono essere eseguite in modo asincrono.

I pool di thread **offrono questi vantaggi**:

- Servire una richiesta con un thread esistente è spesso **più veloce** che attendere la creazione di un thread.
- Un pool di thread **limita il numero di thread esistenti**. Questo è particolarmente importante su sistemi che non possono supportare un gran numero di thread concorrenti.

PROBLEMI LEGATI AL THREADING:

Q: Se un thread in un programma chiama `fork()`, il nuovo processo duplica tutti i thread o il nuovo processo è a thread singolo?

- **A1**: Dipende dal sistema
- **A2**: Alcuni sistemi UNIX hanno scelto di avere due versioni di `fork()`, una che duplica tutti i thread e un'altra che duplica solo il thread che ha richiamato la system call `fork()`.
- **A3**: Se un thread richiama la system call `exec()`, il programma specificato nel parametro `exec()` sostituirà l'intero processo, inclusi tutti i thread.

Nei sistemi UNIX viene utilizzato un **segnale** per notificare a un processo che si è verificato un particolare evento. Un segnale può essere ricevuto in modo **sincrono o asincrono**, a seconda della sorgente e del motivo dell'evento segnalato.

Tutti i segnali, sincroni o asincroni, **seguono lo stesso schema**:

- Un segnale è generato dal **verificarsi di un particolare evento**.
- Il segnale viene **inviato a un processo**.
- Una volta consegnato, il segnale **deve essere gestito**.

Un segnale può essere gestito da uno dei **due possibili handlers**:

- Un gestore di segnale **predefinito**
- Un gestore di segnale **definito dall'utente**

Ogni segnale ha un **signal handler** predefinito che il kernel esegue quando gestisce quel segnale. Questa azione predefinita può essere sovrascritta da uno user-define(utente gestore) che viene chiamato per gestire il segnale.

I segnali vengono gestiti in modi diversi. Alcuni segnali **possono essere ignorati**, mentre altri (ad esempio un accesso illegale alla memoria) vengono gestiti **terminando il programma**.

La gestione dei segnali nei programmi a thread singolo è semplice: i segnali vengono **sempre inviati** a un processo. Tuttavia, fornire segnali nei programmi multithread è più complicato, perché un processo può avere diversi thread.

Q: Dove, allora, dovrebbe essere inviato un segnale?
In generale, esistono le seguenti opzioni:

- **A1**: Invia il segnale al thread a cui si applica il segnale.
- **A2**: Invia il segnale a ogni thread nel processo.
- **A3**: Invia il segnale a determinati thread nel processo.
- **A4**: Viene assegnato un thread specifico per ricevere tutti i segnali del processo.

UNIX consente ai singoli thread di indicare quali segnali stanno accettando e quali stanno ignorando. Fornisce 2 chiamate di sistema separate per fornire segnali a processi/thread, rispettivamente:

- `kill(pid, signal)`
- `pthread_kill(tid, signal)`

CONTENTION SCOPE

Una distinzione tra user-level thread e kernel-level thread risiede nel modo in cui sono **schedulati**. Sui sistemi che implementano i modelli many to one e many to many, la libreria di thread mappa l'esecuzione degli user thread su un **LWP** (lightweight process, ovvero un processore virtuale) disponibile. Questo schema è noto come **process-contention scope (PCS)**, poiché la competizione per la CPU avviene tra thread appartenenti allo stesso processo. Il kernel quando si verificano determinati eventi comunica con la libreria di thread a livello utente tramite un **upcall**. L'upcall è gestita nella libreria dei thread da un **upcall handler**. L'upcall fornisce anche un **nuovo LWP** all'upcall handler, che può quindi essere **riutilizzato** per riprogrammare lo user thread che sta per essere bloccato. Il sistema operativo emetterà un upcall anche quando un thread si sblocca, in modo che la libreria dei thread possa apportare le modifiche appropriate. Se il thread del kernel si blocca, l'LWP si blocca, il che significa che viene bloccato anche lo user thread.

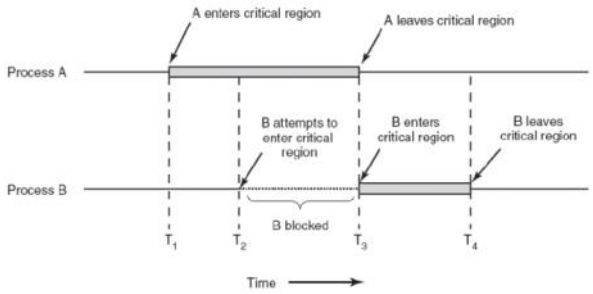
Per decidere quale thread a livello di kernel pianificare su una CPU, il kernel utilizza il **system contention scope** (**SCS**). Con la pianificazione SCS, la competizione per la CPU avviene tra **tutti i thread del sistema**. I sistemi che utilizzano il one to one model, ad esempio Windows e Linux, pianificano i thread utilizzando solo **SCS**.

2 Sincronizzazione tra Processi/Thread

Abbiamo già accennato al fatto che i processi/thread possono cooperare tra loro per raggiungere un compito comune. Tuttavia, la cooperazione può richiedere la **sincronizzazione tra i thread** a causa della presenza delle cosiddette **critical sections** (sezioni critiche).

Consideriamo il seguente scenario del mondo reale, che coinvolge 2 coinquilini: Bob e Carla

Time	Bob	Carla
5:00pm	Arrive home	
5:05pm	Look in the fridge → No milk!	
5:10pm	Leave home for the grocery	
5:20pm		Arrive home
5:25pm	Arrive at the grocery	Look in the fridge → No milk!
5:30pm	Buy milk	Leave home for the grocery
5:45pm	Arrive home, put the milk in the fridge	Arrive at the grocery
5:50pm		Buy milk
6:05pm		Arrive home, put the milk in the fridge
6:05pm	Oh f*%#k!	Oh f*%#k!



Nell'esempio Bob e Carla rappresentano 2 processi/thread. Teoricamente **dovrebbero cooperare** per raggiungere un compito comune (ad esempio, comprare del latte). In pratica, però, potrebbero incorrere in situazioni spiacevoli (ad esempio comprare troppo latte!).

IMPLEMENTAZIONI DELLA SINCRONIZZAZIONE

Per la sincronizzazione abbiamo bisogno di "strumenti" appropriati (**costrutti primitivi**) forniti dai linguaggi di programmazione utilizzati come **blocchi atomici** (la cui esecuzione non può essere interrotta a metà). Analizzeremo i seguenti strumenti:

- **Lock**: In ogni momento, solo un processo detiene un lock, esegue la sua sezione critica e infine rilascia il lock
- **Semaphore**: Una generalizzazione dei lock
- **Monitor**: Per connettere i dati condivisi alle primitive di sincronizzazione

OBIETTIVI DELLA SINCRONIZZAZIONE

Qualsiasi soluzione di sincronizzazione al problema della sezione critica deve soddisfare le seguenti **3 proprietà**:

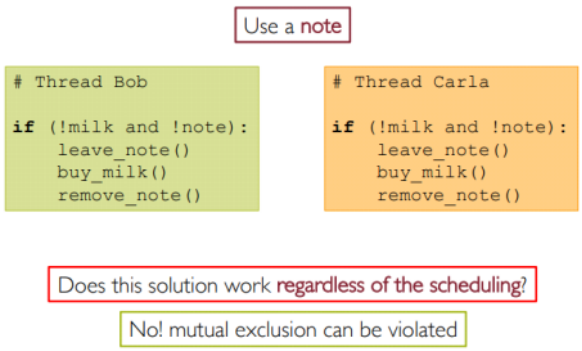
- **Mutua esclusione**: Solo un processo/thread alla volta può trovarsi nella sua sezione critica!
- **Progresso**: Se nessun processo è nella sua sezione critica, e uno o più processi vogliono accedervi, allora ognuno di questi deve essere in grado di entrarci.
- **Attesa limitata**: Esiste un limite al numero di volte in cui altri processi possono accedere alle loro sezioni critiche dopo che un processo abbia fatto richiesta per entrare nella sua sezione critica e prima che tale richiesta venga accolta.

Nell'esempio del latte:

- Garantire la **mutua esclusione** significa che non verrà acquistato più latte di quello necessario (ovvero, solo uno tra Bob e Carla comprerà il latte se necessario)
- Garantire il **progresso** significa che qualcuno dovrebbe comprare del latte (cioè, l'opzione in cui sia Bob che Carla non fanno nulla è sicuramente sicura ma non è desiderata)
- Garantire l'**attesa limitata** significa che eventualmente Bob e Carla entreranno nella loro sezione critica

Tornando all'esempio del latte, analizziamo 3 possibili soluzioni.

ESEMPIO 1

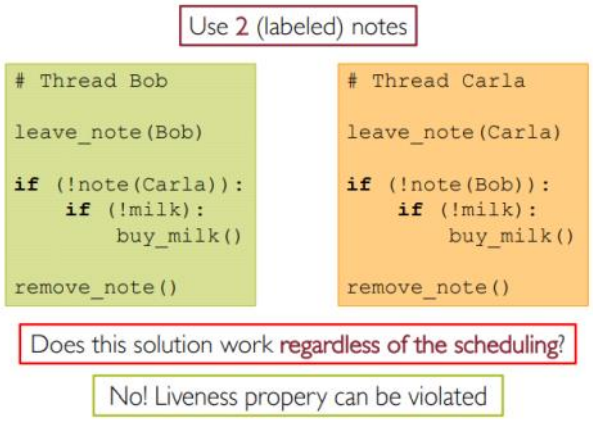


Questa soluzione funziona?
No! La **mutua esclusione può essere violata**.

Nel caso in cui il Thread Bob entri nell'`if`, ma venga interrotto prima di eseguire la funzione `leave_note()`, passando il controllo al Thread Carla, tale thread entrerà nell'`if` (siccome Thread Bob non ha, ancora, lasciato nessuna nota e il latte manca) eseguendo la sezione critica. Quando il controllo tornerà al Thread Bob, che riprenderà la sua esecuzione da dentro l'`if` (e non controllerà quindi se il latte sia stato comprato) eseguirà anch'esso la sezione critica.

Questo porterà ad un acquisto doppio del latte: **la mutua esclusione viene violata**.

ESEMPIO 2

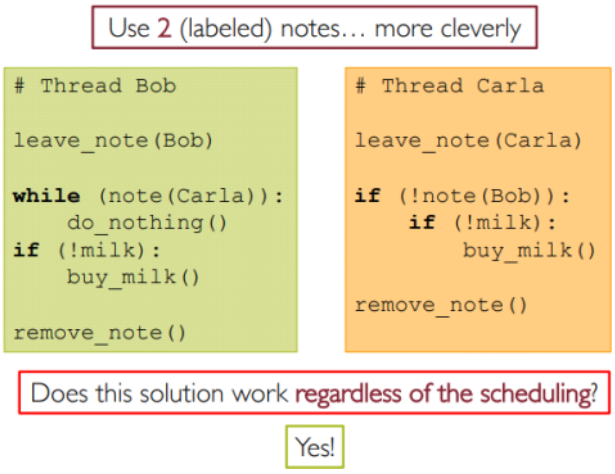


Questa soluzione funziona?
No! Il **progresso** può essere violato.

Se Thread Bob esegue `leave_note()` ma si ferma sul primo `if` senza eseguirlo, passando il controllo al Thread Carla, quest'ultimo eseguirà `leave_note()`, ma trovando la nota lasciata dal Thread Bob, non eseguirà il codice dentro l'`if`. A questo punto, se il controllo tornasse al Thread Bob prima che Thread Carla abbia rimosso la nota, Thread Bob non eseguirebbe il codice dentro l'`if`, in quanto la nota di Carla è ancora presente, passando direttamente ad eseguire `remove_note()`.

In questo caso nessuno dei due thread esegue la sezione critica: **la proprietà del progresso viene violata**.

ESEMPIO 3



Questa soluzione funziona?
Si! Tutte le proprietà sono rispettate.

Analizziamo il "caso peggiore":
Thread Bob esegue `leave_note()`, il controllo passa al Thread Carla che esegue `leave_note()`, a questo punto se il controllo tornasse al Thread Bob, quest'ultimo entrerebbe nel while, non facendo nulla fin quando (prima o poi) il controllo ritorni al Thread Carla, che eseguirebbe la sezione critica terminando l'esecuzione correttamente e rimuovendo la nota (consentendo al Thread Bob di uscire e terminare l'esecuzione correttamente).

In questo caso, e in qualunque altro, **nessuna proprietà viene violata**.

Possiamo quindi dire che la terza soluzione è una soluzione valida?
Non proprio, principalmente per 3 motivi:

- **Troppo complicata**: non è intuitivo capire che funzioni correttamente
- **Asimmetrica**: i thread Bob e Carla sono diversi (aggiungere thread rovinerà ancora di più le cose!)
- **Busy waiting**: il thread Bob sta occupando la CPU senza fare nulla per un tempo indeterminato

2.1 MUTEX LOCKS

Iniziamo con una delle soluzioni ad alto livello più semplice per la risoluzione del problema della sezione critica: la **mutex lock** (**mutex** sta per **mutual exclusion**).
Serve infatti per garantire la **mutua esclusione** ai dati condivisi ed **evitare le situazioni di race**.

Regole per l'utilizzo della classe lock:

- **Acquisire** sempre la lock prima di accedere ai dati condivisi
- **Solo un thread per volta può acquisire la lock**, gli altri aspetteranno
- **Rilasciare** sempre la lock dopo aver terminato con i dati condivisi

La classe lock a tal proposito deve definire 2 funzioni atomiche:

- `Lock.acquire()`
  se la lock è libera viene acquisita, altrimenti il thread si mette in attesa
- `Lock.release()`
  rilascia la lock e riattiva qualsiasi thread in attesa

```
# Thread Bob

Lock.acquire()

if (!milk):
    buy_milk()

Lock.release()
```

```
# Thread Carla

Lock.acquire()

if (!milk):
    buy_milk()

Lock.release()
```

L'implementazione di queste funzioni varia a seconda che si stia operando su un sistema single-core o su un sistema multi-core.

IMPLEMENTAZIONE CON DISABILITAZIONE DEGLI INTERRUPTS

In un sistema single-core possiamo risolvere il problema semplicemente disabilitando gli interrupts all'interno delle funzioni.

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value; // 0=FREE, 1=BUSY
  private Queue q;

  Lock() {
    // lock is initially FREE
    this.value = 0;
    this.q = null;
  }
}
```

We need both `acquire` and `release` being implemented as system calls

```
public void acquire(Thread t) {
  disable_interrupts();
  if(this.value) { // lock is held by someone
    q.push(t); // add t to waiting queue
    t.sleep(); // put t to sleep
  }
  else {
    this.value = 1;
  }
  enable_interrupts();
}
```

```
public void release() {
  disable_interrupts();
  if(!q.is_empty()) {
    t = q.pop(); // extract a waiting thread from q
    push_onto_ready_queue(t); // put t on ready queue
  }
  else {
    this.value = 0;
  }
  enable_interrupts();
}
```

Analizziamo questa implementazione:

- All'inizio della classe `Lock` vengono definiti metodi e attributi, mentre nel costruttore vengono inizializzati gli attributi (lock libera e coda vuota).
- Nell'`acquire` viene controllato che la Lock sia libera, se lo è viene occupata, altrimenti si aggiunge in coda il thread che ha chiamato la funzione e lo si manda in attesa.
- Nella `release` si controlla se la coda è vuota, se lo è `value` viene resettato (la lock è libera), altrimenti viene estratto un thread dalla coda dei thread in attesa e lo si manda in esecuzione.

In entrambe le funzioni gli interrupt vengono disattivati all'inizio e riattivati alla fine tramite system call.

Tale implementazione è però **inattuabile su sistemi multi-core**: la disabilitazione degli interrupts richiede tempo per essere attivata (il messaggio deve passare tra tutti i core). Il passaggio di messaggi ritarda l'ingresso nella sezione critica, diminuendo notevolmente le prestazioni del sistema. Infine bisogna considerare anche l'effetto sul clock di un sistema, se il clock viene mantenuto aggiornato dagli interrupt.

IMPLEMENTAZIONE CON TEST&SET

Passiamo ora ad un'implementazione valida su sistemi multi-core tramite l'istruzione atomica `test&set()` (utilizzata dalla maggior parte delle architetture). Questa istruzione prende in input un valore, lo legge (restituendolo come output al chiamante) e setta ad 1 il valore in memoria.

```
Class Lock {
  public void acquire();
  public void release();
  private int value;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

```
public void acquire() {
  while(test&set(this.value) == 1) {
    // while busy do nothing
  }
}
```

```
public void release() {
  this.value = 0;
}
```

In questa implementazione abbiamo 2 casi possibili:

1. Se la **lock è libera** (`value=0`), `test&set(value)` leggerà 0, setterà `value` ad 1 e ritornerà 0. La lock è ora occupata, l'espressione booleana nel while è `False` e l'`acquire` termina.
2. Se la **lock è occupata** (`value=1`), `test&set(value)` leggerà 1, setterà `value` ad 1 e ritornerà 1. La lock è ancora occupata, l'espressione booleana nel while è `True` e l'`acquire` continua in loop fin quando non viene eseguita una `release`.

Tale implementazione lascia però spazio a due importanti problemi:

- **Busy waiting**
  Tempo computazionale sprecato nel while per un periodo indeterminato
- **Iniquità** (nell'ordine di esecuzione dei thread)
  Poiché non esiste una coda in cui i thread attendono il rilascio del blocco

Vediamo quindi un altro approccio che ci permette di risolvere, seppur parzialmente, i problemi sopra citati.

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value;
  private int guard;
  private Queue q;

  Lock() {
    // lock is initially free
    this.value = 0;
  }
}
```

```
public void acquire(Thread t) {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(this.value) {
    q.push(t);
    t.sleep_and_reset_guard_to_0();
  }
  else {
    this.value = 1;
    this.guard = 0;
  }
}
```

```
public void release() {
  while(test&set(this.guard) == 1) {
    // while busy do nothing
  }
  if(!q.is_empty()) {
    t = q.pop();
    push_onto_ready_queue(t);
  }
  else {
    this.value = 0;
  }
  this.guard = 0;
}
```

In questa implementazione, a differenza della prima, aggiungiamo due parametri: `Guard` e `Queue`. Supponiamo che un thread(T_1) entri nell'`acquire()`,T_1 leggerà il valore 0, imposterà la `Guard` a 1 e ritornerà 0. Il valore booleano restituito dal `while` sarà `False`, quindi T_1 passerà direttamente al controllo dell'`if`. L'`if` darà come valore booleano `False` perché `value` all'inizio è uguale a 0. Supponiamo ora che T_1 venga interrotto e un altro thread(T_2) provi ad acquisire la lock. T_2 leggerà il valore 1, imposterà la `Guard` a 1 e ritornerà 1. A questo punto il valore booleano del while ritornerà `True` e quindi T_2 entrerà nel `while` causando un attesa (**busy waiting**). Ora supponiamo che T_2 venga interrotto e che T_1 riprenda il controllo della CPU. T_1 ripartirà dall'`else`, e imposterà il valore di `value` a 1 e quello di `Guard` a 0. Supponiamo ora che T_2 riprenda il controllo. Nel momento della lettura del valore all'interno del `while`, il valore di `Guard` letto, ora sarà 0, di conseguenza imposterà il valore di `Guard` a 1 e ritornerà 0, uscendo così dall'attesa(**busy waiting**). Dopodiché, passerà all'istruzione `if`, che restituirà il valore `True`, dato che ora `value` è uguale a 1, e T_2 verrà inserito in una coda di attesa e verrà messo a "dormire". T_2 verrà "svegliato" solo quando T1 eseguirà l'istruzione `push_onto_ready_queue()` della funzione `release()`, che servirà a "svegliare" T_2 per metterlo nella ready queue.

Questa implementazione è più efficace della prima, perché a differenza della prima, T_2 rimane in attesa fino a quando T_1 non completa la funzione di `acquire()`, dopodiché può uscire dal `while` e può entrare in una coda, smettendo di occupare la CPU inutilmente. Nella prima implementazione invece T_2 deve attendere fino a quando T_1 non esegue tutta la funzione di `release()`, quindi fino a quando non rilascia totalmente la lock.

In conclusione, abbiamo 2 problemi principali con la disabilitazione degli interrupt:

- **Sovraccarico** dovuto al bisogno di invocare il kernel
- **Irrealizzabile** con architetture multiprocessore

E 2 problemi principali con le istruzioni atomiche:

- **Busy waiting**
- **Iniquità** in quanto non esiste una coda in cui i thread attendono il rilascio della lock

2.2 SEMAFORI

Un semaforo S **è una variabile intera** a cui, a parte l'inizializzazione, si accede solo tramite due operazioni atomiche standard: `wait()` e `signal()`.

L'implementazione di `wait()` è la seguente:

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

L'implementazione di `signal()` è la seguente:

```
signal(S) {
    S++;
}
```

Tutte le modifiche al valore intero del semaforo nelle operazioni `wait()` e `signal()` **devono essere eseguite atomicamente** (senza interruzione). Cioè, quando un processo modifica il valore del semaforo, nessun altro processo può modificare contemporaneamente lo stesso valore. Inoltre, nel caso di `wait(S)`, il test del valore intero di S ($S \leq 0$), nonché la sua eventuale modifica (S--), devono essere eseguiti senza interruzione.

USO DEL SEMAFORO

I sistemi operativi spesso distinguono tra **counting** e **binary semaphores**. Il valore di **un counting semaphore può variare su un dominio** senza restrizioni. Il valore di u**n semaforo binario può variare solo tra 0 e 1**.

Pertanto, i **binary semaphore** si comportano in modo simile ai **mutex locks**. Infatti, su sistemi che non forniscono mutex lock, per fornire la mutua esclusione possono essere usati i binary semaphore.
I **counting semaphore** possono essere utilizzati per controllare l'accesso a una data risorsa costituita da un **numero finito di istanze**. Il semaforo è inizializzato al numero di risorse disponibili. Ogni processo che desidera utilizzare una risorsa esegue un'operazione `wait()` sul semaforo (**decrementando** così il counter). Quando un processo rilascia una risorsa, esegue un'operazione `signal()` (**incrementando** il counter). Quando il conteggio va a 0, significa che tutte le risorse vengono utilizzate. Successivamente, i processi che desiderano utilizzare una risorsa **si bloccheranno** finché il counter non diventa maggiore di 0.

Possiamo usare i semafori anche per risolvere vari problemi di sincronizzazione. Ad esempio, consideriamo due processi in esecuzione contemporaneamente: P_1 con un'istruzione S_1 e P_2 con un'istruzione S_2. Supponiamo di richiedere che S2 venga eseguito solo dopo che S_1 è stato completato. Possiamo implementare facilmente questo schema lasciando che P_1 e P_2 condividano una sincronizzazione del semaforo comune, inizializzata a 0. Nel processo P_1, inseriamo le istruzioni:

```
S1;
signal(synch);
```

Nel processo P_2, inseriamo le istruzioni:

```
wait(synch);
S2;
```

Poiché **synch** è inizializzato a 0, P_2 eseguirà S_2 solo dopo che P_1 ha richiamato `signal(synch)`, ovvero dopo che l'istruzione S_1 è stata eseguita.

IMPLEMENTAZIONE DEL SEMAFORO

Ricordiamo che l'implementazione dei mutex lock provoca **busy waiting**. Le operazioni del semaforo `wait()` e `signal()` appena descritte presentano lo stesso problema. Per ovviare a questo problema, possiamo modificare le due operazioni come segue: Quando un processo esegue l'operazione `wait()` e scopre che il valore del semaforo non è positivo, **deve attendere**. Tuttavia, invece di andare in busy waiting, il processo **può sospendersi**. L'operazione di sospensione **inserisce un processo in una coda** di attesa associata al semaforo e lo stato del processo passa allo **stato di waiting**. Quindi il controllo viene trasferito allo scheduler della CPU, che seleziona un altro processo da eseguire. Un processo sospeso, in attesa su un semaforo S, dovrebbe essere riavviato quando qualche altro processo esegue la funzione di `signal()`. Il processo viene riavviato dalla funzione `wakeup()`, **che modifica lo stato dal processo** da waiting a ready. Il processo viene quindi inserito nella coda ready.

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Definiamo l'implementazione come segue:
Ogni semaforo **ha un valore intero e una lista di processi**.
Quando un processo deve attendere su un semaforo, viene aggiunto alla lista dei processi. L'operazione `signal()` rimuove un processo dalla lista dei processi in attesa e lo risveglia.

Ora, l'operazione `wait()` può essere definita come:

```
wait(semaphore *S) {
    S -> value--;
    if (S -> value < 0) {
        add this process to S -> list;
        sleep();
}
```

e l'operazione `signal()` può essere definita come:

```
signal(semaphore *S) {
    S -> value++;
    if (S -> value <= 0) {
        remove a process P from S -> list;
        wakeup(P);
}
```

L'operazione `sleep()` **sospende il processo** che la invoca. L'operazione `wakeup(P)` **riprende l'esecuzione di un processo** sospeso P. Queste due operazioni sono fornite dal sistema operativo come system call di base.

Se il valore di un semaforo è negativo, **la sua grandezza è il numero di processi in attesa** su quel semaforo (per esempio, se il valore è -5, significa che ci sono 5 processi in attesa). Un modo per aggiungere e rimuovere processi dalla lista in modo da garantire un'attesa limitata, è utilizzare una coda FIFO, dove il semaforo contiene sia il puntatore alla testa che quello alla coda.

Nei processori a singolo core, possiamo risolvere il problema della mutua esclusione, **disabilitando gli interrupts durante il tempo in cui** `wait()` **e** `signal()` **sono in esecuzione**. Viene eseguito solo il processo attualmente in esecuzione fino a quando gli interrupt non vengono abilitati e lo scheduler può riprendere il controllo.

In un ambiente multicore, **gli interrupt devono essere disabilitati su ogni core**. In caso contrario, le istruzioni provenienti da processi diversi (in esecuzione su core diversi) potrebbero essere alternati in modo arbitrario.

Disabilitare gli interrupt su ogni core può essere un compito difficile e può ridurre drasticamente le prestazioni. Pertanto, i sistemi SMP(Sistema multiprocessore simmetrico) devono fornire tecniche alternative, come il `compare_and_swap()` o spinlocks, per garantire che `wait()` e `signal()` vengano eseguiti atomicamente.
Con questa definizione delle operazioni `wait()` e `signal()`, **non abbiamo eliminato completamente il busy waiting**.

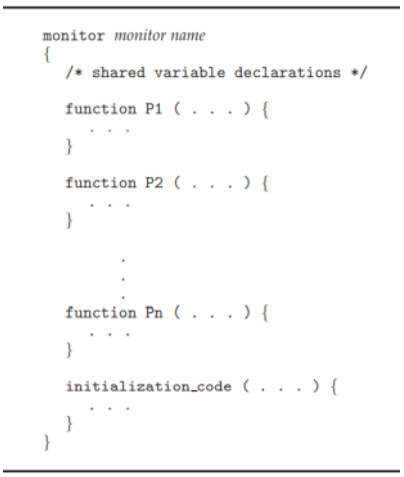
Piuttosto, **abbiamo spostato il busy waiting dalla entry section alla critical section** dei programmi. Inoltre, abbiamo limitato il busy waiting alle critical section delle operazioni `wait()` e `signal()`, e **queste sezioni sono brevi** (se codificate correttamente, non dovrebbero essere più di una decina di istruzioni). Pertanto, la critical section non è quasi mai occupata e il busy waiting si verifica raramente, e quindi solo per un breve periodo.

2.3 MONITOR

Per comprendere cosa sia un monitor dobbiamo introdurre il concetto di **ADT** (***Abstract Data Type***).
Un **ADT è un'astrazione di una struttura dati**, definita tramite insieme di funzioni per **incapsulare e manipolare** tali dati. Tali dati sono indipendenti da qualsiasi specifica implementazione dell'ADT.

> Ad esempio, uno stack astratto, che è una struttura LIFO, può essere definito tramite 3 funzioni: *push*, per inserire dati alla cima; *pop*, per togliere dati dalla cima; *peek*, per accedere alla cima senza rimuovere i dati. In tal modo lo stack viene definito semplicemente definendo le funzioni che utilizza.

Un **monitor è un ADT** che include un set di operazioni di alto livello con lo scopo di regolare l'accesso ai dati condivisi. Il monitor è anche conosciuto come uno **strumento di sincronizzazione**: Java, C++ e C# sono alcuni esempi di linguaggi di programmazione che consentono l'utilizzo dei monitor. I processi operanti fuori dal monitor non possono accedere alle sue variabili interne, se non tramite la chiamata delle funzioni del monitor.



Il costrutto monitor garantisce che sia attivo un solo processo alla volta all'interno del monitor. Di conseguenza, il programmatore non ha bisogno di scrivere i vincoli di sincronizzazione esplicitamente.

Il costrutto monitor, come definito fin ora, non è però abbastanza potente per modellare alcuni schemi di sincronizzazione. Per questo scopo dobbiamo definire meccanismi di sincronizzazione aggiuntivi. Questi meccanismi sono forniti dal costrutto `condition`. Un programmatore che ha bisogno di scrivere uno schema di sincronizzazione su misura, può definire una o più variabili di tipo `condition`.

```
condition x, y;
```

Le uniche operazioni che possono essere invocate su una variabile `condition`, sono `wait()`, `signal()` e `broadcast()`. L'operazione

```
x.wait();
```

significa che il processo che invoca questa operazione sarà sospeso fin quando un altro thread non invochi

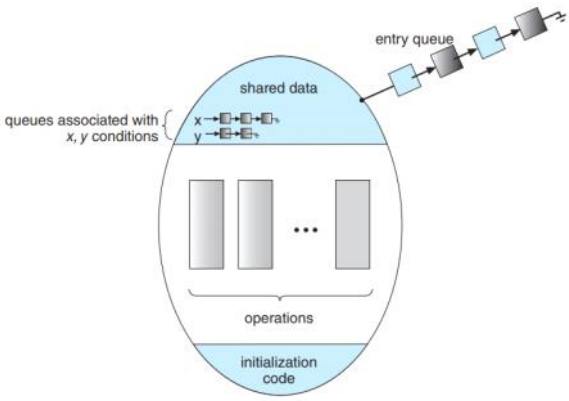
```
x.signal();
```

che riattiva un thread in stato di attesa. Se non vi sono thread in attesa, l'operazione `signal()` non ha alcun effetto. Da non confondere questa operazione con quella dei semafori, la quale influenza sempre lo stato del semaforo.

L'operazione `broadcast()` semplicemente riattiva tutti i thread in stato di attesa.

STRUTTURA DI UN MONITOR

Come possiamo vedere nello schema di fianco, un monitor consta essenzialmente di 4 parti:



- **Shared data**
  Dove vengono dichiarate le variabili `condition` condivise tra i processi
- **Operations**
  Funzioni che definiscono le operazioni da eseguire sui dati condivisi
- **Initialization code**
  Dove vengono inizializzate le variabili di cui si ha bisogno
- **Entry queue**
  Coda di thread in attesa del proprio turno per eseguire le operazioni (metodi) del monitor

Supponiamo adesso che, quando l'operazione `x.signal()` è invocata da un processo P, esista un processo sospeso Q associato con la condizione `x`. Chiaramente se il processo sospeso Q riprende l'esecuzione, il processo segnalante P deve andare in attesa (altrimenti sia P che Q sarebbero attivi simultaneamente nel monitor).

Abbiamo quindi 2 possibilità dopo la chiamata `x.signal()` da parte di P:

- **Signal and wait**
  P attende che Q lasci il monitor o che si metta in attesa per un'altra condizione
- **Signal and continue**
  Q attende che P lasci il monitor o che si metta in attesa per un'altra condizione

Esiste anche un "compromesso" tra queste due scelte: quando un thread P esegue l'operazione `signal()`, lascia immediatamente il monitor, in tal modo Q viene attivato subito.

2.4 DEADLOCK

In un ambiente di multiprogrammazione, **diversi thread possono competere per un numero finito di risorse**. Un thread richiede risorse, se le risorse non sono disponibili in quel momento, il thread entra in uno stato di attesa.
A volte, un thread in attesa non può più cambiare stato, perché le risorse che ha richiesto sono conservate da altri thread in attesa. Questa situazione è chiamata **deadlock**.

> Forse il miglior esempio di una situazione di deadlock può essere tratto da una legge approvata dalla legislatura del Kansas all'inizio del XX secolo che dice:
>
> *"Quando due treni si avvicinano a un incrocio, entrambi devono fermarsi completamente e nessuno dei due ricomincerà finché l'altro non se ne sarà andato"*

ESEMPIO PRATICO:

Thread A

```
printer.wait();
disk.wait();
// copy from disk to printer
printer.signal();
disk.signal();
```

Thread B

```
disk.wait();
printer.wait();
// copy from disk to printer
printer.signal();
disk.signal();
```

Supponiamo che il thread A parta per primo. Fa l'`acquire()` della printer e poi avviene un context switch. A questo punto arriva il thread B che fa l'`acquire()` del disco e avviene un context switch. Il controllo ritorna al thread A, esegue `disk.wait()` ma si blocca. Il controllo ritorna a B, che esegue `printer.wait()`, ma anch'esso si blocca. Tutto questo perché A aspetta che B rilasci disk, mentre B aspetta che A rilasci printer.

I sistemi operativi in genere non forniscono funzionalità di prevenzione dei deadlock e rimane ai programmatori la responsabilità di garantire che progettino programmi privi di deadlock .

Un sistema è costituito da un numero finito di risorse da distribuire tra un numero di thread concorrenti. **Le risorse possono essere suddivise in più tipi** (o classi), **ciascuno costituito da un certo numero di istanze identiche**.

> I cicli della CPU, i file e i dispositivi I/O (come le interfacce di rete e le unità DVD) sono esempi di tipi di risorse.
> Se un sistema ha quattro CPU, il tipo di risorsa CPU ha quattro istanze.

Se un thread richiede un'istanza di un tipo di risorsa, l'allocazione di qualsiasi istanza di quel tipo può soddisfare la richiesta.

2.5 DEADLOCK CHARACTERIZATION

Una situazione di "deadlock" può verificarsi se sono presenti contemporaneamente le seguenti quattro condizioni:

- **Mutual exclusion**: solo **un thread alla volta può utilizzare la risorsa**. Se un altro thread richiede quella risorsa, il thread richiedente deve essere ritardato fino a quando la risorsa non sia stata rilasciata.
- **Hold and wait**: Un thread deve **contenere almeno una risorsa e attendere di acquisire risorse aggiuntive** che sono attualmente detenute da altri thread.
- **No preemption**: Le **risorse non possono essere anticipate**; ovvero, una risorsa può essere rilasciata solo volontariamente dal thread che la detiene, dopo che quel thread abbia completato il suo compito.
- **Circular wait**: Deve esistere un insieme $\{T_0, T_1, ..., T_n\}$ di thread in attesa, tale che T_0 sia in attesa di una risorsa detenuta da T_1, T_1 sia in attesa di una risorsa detenuta da T_2, ..., T_{n-1} sia in attesa di una risorsa detenuta da T_n, e T_n sia in attesa di una risorsa detenuta da T_0.

La condizione di circular wait implica la condizione hold and wait, quindi le quattro condizioni non sono completamente indipendenti.

2.6 RESOURCE ALLOCATION GRAPH

I deadlock possono essere descritti più precisamente in termini di un grafo diretto chiamato "system resource-allocation graph". Questo grafo consiste in un insieme di **vertici V** e un insieme di **archi E**.

L'**insieme dei vertici V** è partizionato in **due diversi tipi di nodi**: $T = \{T_1, T_2, ..., T_n\}$, l'insieme costituito da tutti i **thread attivi** nel sistema, e $R = \{R_1, R_2, ..., R_m\}$, l'insieme costituito da tutti i **tipi di risorse** nel sistema.

Un arco diretto dal thread T_i al tipo di risorsa R_j è indicato da $T_i \rightarrow R_j$: il thread T_i ha richiesto un'istanza del tipo di risorsa R_j ed è attualmente in attesa di tale risorsa.
Un arco diretto dal tipo di risorsa R_j al thread T_i è denotato da $R_j \rightarrow Ti$: un'istanza del tipo di risorsa R_j è stata allocata al thread T_i.

Un arco orientato $T_i \rightarrow R_j$ è detto "**request edge**" (arco di richiesta); un arco orientato $R_j \rightarrow T_i$ è detto "**assignment edge**" (arco di assegnazione). Rappresentiamo ogni thread T_i come un cerchio e ogni tipo di risorsa R_j come un rettangolo.

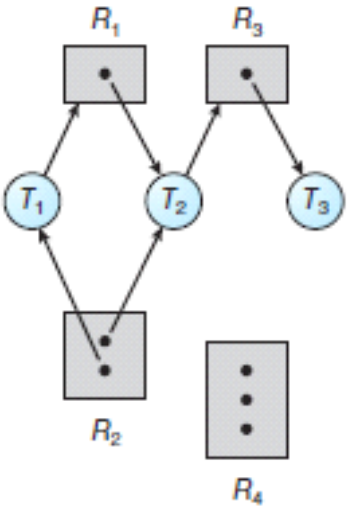
Poiché il tipo di risorsa R_j può avere più di un'istanza, rappresentiamo ciascuna di queste istanze come un punto all'interno del rettangolo. Quando il thread T_i richiede un'istanza del tipo di risorsa R_j, nel resource-allocation graph viene inserito un request edge.

Quando questa **richiesta può essere soddisfatta**, il **request edge viene trasformato in un assignment edge**. Quando il thread non ha più bisogno di accedere alla risorsa, rilascia la risorsa. Di conseguenza, viene eliminato l'assignment edge.

Il resource-allocation graph mostrato nella figura, rappresenta la seguente situazione:

Gli insiemi T, R ed E:

- $T = \{T_1, T_2, T_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$



Istanze di risorse:

- Un'istanza del tipo di risorsa R_1
- Due istanze del tipo di risorsa R_2
- Un'istanza del tipo di risorsa R_3
- Tre istanze del tipo di risorsa R_4

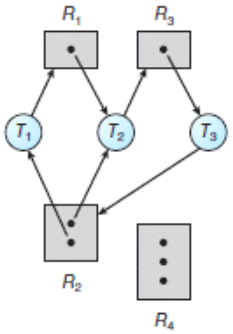
Stati del thread:

- Il thread T_1 contiene un'istanza del tipo di risorsa R_2 ed è in attesa di un'istanza del tipo di risorsa R_1.
- Il thread T_2 contiene un'istanza del tipo di risorsa R_1 e un'istanza del tipo di risorsa R_2,ed è in attesa di un'istanza del tipo di R_3.
- Il thread T_3 contiene un'istanza di R_3.

Data la definizione di **resource-allocation graph**, si può dimostrare che, se il grafo non contiene cicli, nessun thread nel sistema è in "deadlock". Se il grafo contiene un ciclo, **potrebbe** avvenire un deadlock.

Se **ogni tipo di risorsa ha solo un'istanza**, un **ciclo implica che sia presente un deadlock**.
Se **ogni tipo di risorsa ha diverse istanze**, un **ciclo non implica necessariamente che sia presente un deadlock**. In quest'ultimo caso, un ciclo nel grafo è una condizione necessaria ma non sufficiente per l'esistenza di un deadlock.

Per illustrare questo concetto, torniamo al resource-allocation graph rappresentato in figura:
Supponiamo che il thread T3 richieda un'istanza del tipo di risorsa R2. Poiché nessuna istanza della risorsa T2 è attualmente disponibile, aggiungiamo un request edge T3 → R2.



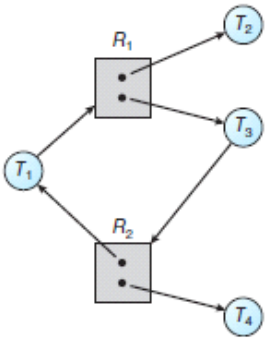
A questo punto nel sistema esistono due cicli minimi:

- $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
- $T2 \rightarrow R3 \rightarrow T3 \rightarrow R2 \rightarrow T2$

I thread T_1, T_2 e T_3 sono in deadlock. Il thread T_2 è in attesa della risorsa R_3, che è tenuta dal thread T_3. Il thread T_3 è in attesa che il thread T_1 o il thread T_2 rilasci la risorsa R_2. Inoltre, il thread T_1 è in attesa che il thread T_2 rilasci la risorsa R_1.

Consideriamo ora il resource-allocation graph in figura qui di fianco.
In questo esempio, abbiamo un ciclo:

- $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$



Tuttavia, non vi è alcuna situazione di deadlock, in quanto possiamo osservare che il thread T_4 può rilasciare la sua istanza del tipo di risorsa R_2. Quella risorsa può quindi essere assegnata a T_3, interrompendo il ciclo. In sintesi, se un resource-allocation graph non ha un ciclo, il sistema ***non*** è in uno stato di deadlock. Se c'è un ciclo, allora il **sistema potrebbe trovarsi in uno stato di deadlock**.

In generale, possiamo affrontare il problema del deadlock in tre modi:

- Possiamo ignorare del tutto il problema e fingere che i deadlock non si verifichino mai nel sistema.
- Possiamo utilizzare un protocollo per prevenire o evitare i deadlock, assicurandoci che il sistema non entri mai in uno stato di deadlock.
- Possiamo consentire al sistema di entrare in uno stato di deadlock, rilevarlo e ripristinarlo.

La prima soluzione è quella utilizzata dalla maggior parte dei sistemi operativi, inclusi Linux e Windows. Per garantire che i deadlock non si verifichino mai, il sistema può utilizzare uno schema **deadlock prevention** o uno schema di **deadlock avoidance.**

2.7 DEADLOCK PREVENTION

Come abbiamo detto perché si verifichi un deadlock, devono sussistere **4 condizioni**. Assicurandoci che almeno una di queste condizioni non possa verificarsi, possiamo prevenire il verificarsi di un deadlock. Elaboriamo questo approccio esaminando ciascuna delle quattro condizioni necessarie separatamente.

MUTUAL EXCLUSION

La condizione di mutua esclusione deve valere. Cioè, **almeno una risorsa deve essere non condivisibile**. Le risorse condivisibili non richiedono un accesso che si escluda a vicenda e quindi non possono essere coinvolte in un deadlock.

> I file read-only sono un buon esempio di risorsa condivisibile. Se più thread tentano di aprire contemporaneamente un file di sola lettura, è possibile concedere loro l'accesso simultaneo al file.

Un thread non deve mai attendere una risorsa condivisibile. In generale, tuttavia, non possiamo prevenire i deadlock negando la condizione di mutua esclusione, perché alcune risorse sono non condivisibili. Ad esempio, un mutex lock non può essere condiviso contemporaneamente da più thread.

HOLD AND WAIT

Per garantire che la condizione hold-and-wait non si verifichi mai nel sistema, dobbiamo garantire che, ogni volta che un thread richieda una risorsa, esso non contenga altre risorse.

Un primo metodo che possiamo usare, **consiste nel far richiedere e allocare tutte le risorse richieste da un thread prima della sua esecuzione**. Questo metodo è impraticabile per la maggior parte delle applicazioni per la natura dinamica della richiesta di risorse.

Un secondo metodo, consiste nel **consentire l'allocazione delle risorse ad un thread, solo se il thread non ha già altre risorse allocate**. Un thread può richiedere alcune risorse e utilizzare, prima però che possa richiederne altre, deve rilasciare tutte le risorse attualmente allocate.

Entrambi questi metodi comportano 2 principali svantaggi:

1. **Basso utilizzo delle risorse**. Le risorse possono essere allocate ma non utilizzate per un lungo periodo.

   > Ad esempio, ad un thread può essere assegnato un mutex lock per la sua intera esecuzione, ma richiederlo solo per una breve durata.

2. **Starvation**. Un thread che ha bisogno di diverse risorse molto richieste, **potrebbe dover attendere indefinitamente**, perché almeno una delle risorse di cui ha bisogno è sempre allocata a qualche altro thread.

NO PREEMPTION

La terza condizione necessaria per i deadlock è che non ci sia preemption delle risorse che sono già state assegnate.

Per garantire che questa condizione non valga, possiamo utilizzare il seguente protocollo: Se un thread che già detiene risorse, richiede un'altra risorsa che non può essere allocata immediatamente, tutte le risorse attualmente in possesso del thread vengono preempted (**rilasciate)**.
Le risorse preempted vengono aggiunte all'elenco delle risorse per le quali il thread è in attesa. Il thread verrà riavviato solo quando potrà recuperare le sue vecchie risorse, oltre a quelle nuove che sta richiedendo.

In alternativa, se un thread richiede alcune risorse, prima controlliamo se sono disponibili. Se lo sono, le assegniamo. In caso contrario, controlliamo se sono allocate a qualche altro thread in attesa di risorse aggiuntive. In tal caso, facciamo un preempt delle risorse desiderate dal thread in attesa e le allochiamo al thread richiedente.
Se le risorse non sono né disponibili né trattenute da un thread in attesa, il thread richiedente deve attendere. Mentre è in attesa, alcune delle sue risorse potrebbero essere preempted, ma solo se un altro thread le richiede.
Un thread può essere riavviato solo quando gli vengono assegnate le nuove risorse che sta richiedendo e recupera tutte le risorse che sono state preempted mentre era in attesa. Questo protocollo viene spesso applicato a risorse il cui stato può essere facilmente salvato e ripristinato successivamente, come i registri della CPU e le transazioni di un database.

Questo protocollo in genere non può essere applicato a risorse come mutex lock e semafori, ovvero il tipo di risorse in cui si verificano più comunemente i deadlock.

CIRCULAR WAIT

Le tre opzioni presentate finora per la prevenzione dei deadlock sono poco pratiche nella maggior parte delle situazioni. Tuttavia, la quarta e ultima condizione per il deadlock, ovvero la circular wait, presenta un'opportunità per invalidare tale condizione. Un modo per garantire che questa condizione non sia mai valida è imporre un ordinamento di tutti i tipi di risorse e far sì che **ogni thread richieda risorse in un ordine di enumerazione crescente**.

Per illustrare, supponiamo che $R = \{R_1, R_2, \dots, R_m\}$ sia l'insieme dei tipi di risorsa. Assegniamo a ciascun tipo di risorsa un numero intero univoco, che ci consente di confrontare due risorse e di determinare se una precede l'altra nel nostro ordinamento.

Formalmente, definiamo una funzione $F: R \to N$, dove N è l'insieme di numeri naturali.

Un thread può quindi inizialmente richiedere un'istanza di una risorsa, ad esempio R_i; successivamente il thread può richiedere un'istanza della risorsa $\boldsymbol{R_j}$**se e solo se** $\boldsymbol{F(R_j) > F(R_i)}$. In alternativa, possiamo richiedere che un thread che richiede un'istanza della risorsa R_j abbia rilasciato qualsiasi risorsa R_i tale che $F(R_i) \geq F(R_j)$. Si noti inoltre che se sono necessarie più istanze dello stesso tipo di risorsa, ed è necessario emettere un'unica richiesta per tutte. Se vengono utilizzati questi due protocolli, la condizione di attesa circolare non può essere mantenuta.

Possiamo dimostrare questo fatto supponendo che esista un'attesa circolare (dimostrazione per assurdo). Sia $\{T_0, T_1, \dots, T_n\}$ l'insieme dei thread coinvolti nell'attesa circolare dove T_i è in attesa di una risorsa R_i, che è detenuta dal thread $T_{(i+1)\,\%\,n}$ (Il modulo aritmetico viene utilizzato sugli indici, in modo che T_n sia in attesa di una risorsa R_n detenuta da T_0). Quindi, poiché il thread T_{i+1} contiene la risorsa R_i mentre richiede la risorsa R_{i+1}, dobbiamo avere $F(R_i) < F(R_{i+1})$ per ogni i. Ma questa condizione significa che $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$. Per transitività, $F(R_0) < F(R_0)$ il che è impossibile. Pertanto, non può esserci alcuna attesa circolare.

Tieni presente che lo sviluppo di un ordinamento, o gerarchia, non previene di per sé il deadlock. Spetta agli sviluppatori scrivere programmi che seguano l'ordine.

2.8 DEADLOCK DETECTION

Se un sistema non utilizza né un algoritmo di prevenzione né un algoritmo di avoidance, può verificarsi una situazione di deadlock. In questo caso, il sistema può fornire:

- Un algoritmo che esamina lo stato del sistema per determinare se si è verificato un deadlock
- Un algoritmo per riprendersi da un deadlock

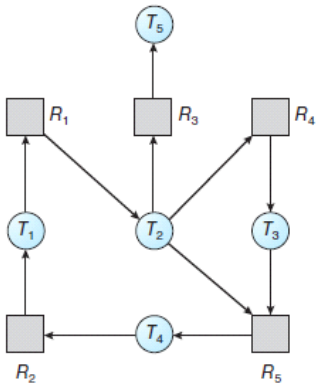
Successivamente, discuteremo questi due requisiti in quanto riguardano sistemi con una sola istanza di ciascun tipo di risorsa, così come sistemi con diverse istanze di ciascun tipo di risorsa.

SINGOLA ISTANZA DI OGNI TIPO DI RISORSA

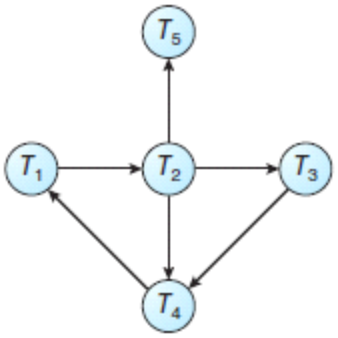
Se tutte le risorse hanno una sola istanza, allora possiamo definire un algoritmo di rilevamento dei deadlock che utilizza una variante del resource-allocation graph, chiamato grafo di attesa (**wait-for graph**).

Otteniamo questo grafo dal resource-allocation graph rimuovendo i nodi delle risorse e gli archi appropriati. Più precisamente, un arco da T_i a T_j nel grafo di attesa implica che il thread T_i sta aspettando che il thread T_j rilasci una risorsa di cui T_i ha bisogno.

Un arco $T_i \rightarrow T_j$ in un grafo di attesa esiste **se e solo se** il corrispondente resource-allocation graph contiene due archi $T_i \rightarrow R_q$ e $R_q \rightarrow T_j$ per qualche risorsa R_q.
Nelle due figure qui sotto possiamo vedere un resource-allocation graph e il corrispondente wait-for graph.



Resource-allocation graph



Corresponding wait-for graph

Come prima, nel sistema esiste un deadlock se e solo se il grafo di attesa contiene un ciclo. Per rilevare i deadlock, il sistema deve mantenere il wait-for graph e invocare periodicamente un algoritmo che cerchi un ciclo nel grafo. Un algoritmo per rilevare un ciclo in un grafo richiede $O(n^2)$ operazioni, dove n è il numero di vertici nel grafo.

Lo schema del wait-for graph non è applicabile a un sistema di allocazione delle risorse con più istanze per tipo di risorsa. L'algoritmo utilizza diverse strutture di dati variabili nel tempo che sono simili a quelle utilizzate nel Banker's Algorithm.

Quando dovremmo quindi invocare l'algoritmo di rilevamento?
La risposta dipende da due fattori:

- Con che frequenza è probabile che si verifichi un deadlock?
- Quanti thread saranno interessati dal deadlock quando si verificherà?

Se i deadlock si verificano frequentemente, l'algoritmo di detection deve essere richiamato frequentemente. Le risorse allocate ai thread in deadlock saranno inattive fino a quando il deadlock non sarà risolto. Inoltre, il numero di thread coinvolti nel ciclo di deadlock può aumentare.

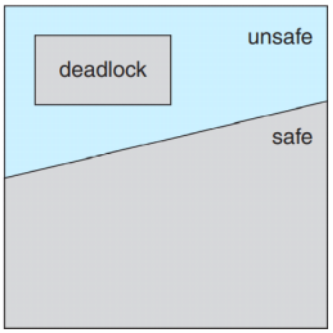
2.9 DEADLOCK AVOIDANCE

Come abbiamo visto, gli algoritmi di prevenzione della deadlock agiscono **limitando il modo in cui le richieste possano esser fatte**. Tali limiti assicurano che **almeno una** delle condizioni necessarie per la deadlock non si verifichi. Aggirare il problema in questo modo può però portare a degli effetti collaterali e dei rallentamenti nel sistema.

Un metodo alternativo per evitare le deadlocks è quello di richiedere informazioni addizionali su come le risorse debbano essere richieste.

> Ad esempio, in un sistema con risorse R_1 ed R_2, il sistema, prima di rilasciare le risorse, potrebbe aver bisogno di sapere che il thread T_1 richieda prima l'accesso ad R_1 e successivamente ad R_2, mentre che il thread T_2 acceda prima a R_2 e poi a R_1.

Esistono vari algoritmi che usano questo approccio, il più semplice e utile richiede che **ogni thread dichiari a priori il massimo numero di risorse per ogni tipo, al quale potrebbe aver bisogno di accedere**. Tramite queste informazioni è possibile definire un algoritmo che non entrerà mai in uno stato di deadlock.
Un algoritmo di evitamento quindi esamina dinamicamente lo stato di allocazione delle risorse, per assicurare che non capiti mai una condizione di "attesa-circolare".



SAFE STATE

Uno stato è **sicuro** se il sistema può allocare risorse ad ogni thread (fino al suo massimo) in qualche ordine e evitando la deadlock. Formalmente possiamo dire che:

> *Un sistema è in uno stato sicuro se e soltanto se esiste un **sequenza sicura**.*

Una sequenza di thread $< T_1,\ T_2, \dots, T_n >$ viene definita **sicura** per lo stato di allocazione corrente, se per ogni T_i, le richieste di risorse che T_i può ancora fare possono essere soddisfatte **dalle risorse al momento accessibili più le risorse detenute da tutti i thread T_j con $j < i$**. In questa soluzione se le risorse che T_i richiede non sono immediantamente accessibili, allora T_i **può attendere** fin quando T_j abbia finito. Quando tutti i processi precedenti a T_i hanno finito, T_i può ottentere tutte le risorse richieste, completare i suoi task, ritornare le risorse allocate e terminare l'esecuzione. Quando T_i termina, **T_{i+1} può eventualmente ottenere le sue risorse richieste** (l'algoritmo può anche decidere di concedergliele in un secondo momento).
Se non esistono sequenze sicure lo stato viene detto **unsafe**, ciò però non vuol dire necessariamente che si andrà a finire in una deadlock (**uno stato non sicuro potrebbe portare ad una deadlock**).

> Consideriamo un sistema con 12 risorse e 3 thread (T_0, T_1, T_2). T_0 potrebbe richiedere fino 10 risorse, T_1 potrebbe richiederne un massimo di 4 e T_2 potrebbe richiederne fino a 9. Supponiamo ora che al tempo t_0: T_0 sta utilizzando 5 risorse mentre T_1 e T_2 ne stanno utilizzando 2. Abbiamo quindi 9 risorse occupate e 3 risorse disponibili al tempo t_0.

	Max. Needs	*Allocated t_0*	*Available t_0*	*Allocated t_1*	*Available t_1*	*Allocated t_2*	*Available t_2*
T_0	10	5		5		0	
T_1	4	2		0		0	
T_2	9	2		2		2	
			3		5		10

> Al tempo t_0 il sistema è in uno stato sicuro: la sequenza <T_1, T_0, T_2> infatti soddisfa la condizione di sicurezza. T_1 può infatti allocare immediatamente tutte le risorse che gli servono, finendo il suo compito e rilasciandole. Abbiamo quindi adesso 5 risorse libere, il thread T_0 può terminare tranquillamente la sua esecuzione allocando tutte le risorse che gli servono (fino ad un massimo 10) e liberandole al termine del suo compito. Adesso T_2 può accedere fino a 7 risorse tra le 10 accessibili (contando le 2 già occupate), terminando anch'esso la sua esecuzione e liberando tutte le risorse occupate.

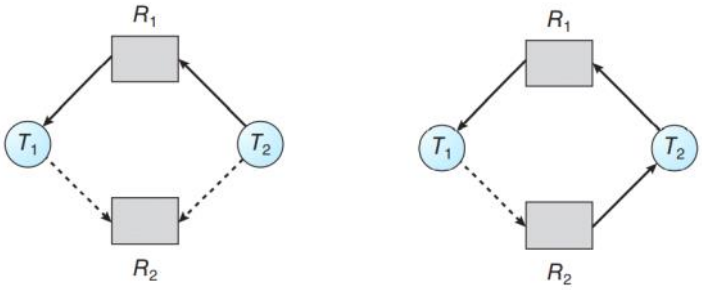
Supponiamo ora che al tempo t_1, dopo l'esecuzione di T_1, T_2 alloca un'altra risorsa, abbiamo ora 4 risorse disponibili, e non siamo in uno stato sicuro. Se T_0 richiedesse meno di 5 risorse, oltre a quelle già occupate, non ci sarebbero problemi, in caso contrario dovrebbe aspettare. In fine se T2 richiedesse, ad esempio 6 risorse, dovrebbe anch'esso aspettare, generando una deadlock.

Ora che abbiamo chiarito il concetto di safe state, procediamo con la descrizione di alcuni algoritmi che seguono lo schema introdotto.

RESOURCE-ALLOCATION-GRAPH ALGORITHM

Se abbiamo un sistema di allocazione delle risorse con solo un'istanza per ogni tipo di risorsa, possiamo usare una variante del resource-allocation-graph, di cui abbiamo già parlato, che aggiunge un nuovo tipo di arco (edge): il **claim edge**. Il claim edge $T_i \rightarrow R_j$ indica che il thread T_i potrebbe richiedere l'accesso alla risorsa R_j in futuro. La direzione di tale arco è la stessa del request edge, ma viene **rappresentato da una freccia tratteggiata**. Quando T_i richiede la risorsa R_j, **il claim edge viene convertito in request edge** e, quando una risorsa viene liberata, **l'assignement edge viene convertito in claim edge**.
Ovviamente per funzionare vuol dire che tutte le risorse che potrebbero servire a ciascun thread **devono essere dichiarate a priori**.

Supponiamo adesso che il thread T_i richieda la risorsa R_j: tale risorsa sarà garantita soltatanto se, convertendo il request edge in un assignment edge, non si creino cicli nel resource-allocation-graph. Controlliamo la sicurezza dello stato con un **algoritmo di cycle-detection** che, in questo schema, richiede un numero di operazioni nell'ordine di $\boldsymbol{n^2}$ con $n = numero\ di\ thread\ nel\ sistema$.
Se non sono presenti cicli allora l'allocazione della risorsa lascerà il sistema in uno stato sicuro, in caso contrario, per evitare di entrare in uno stato non sicuro, il thread che ha eseguito la richiesta **dovrà attendere** prima di poter accedere alla risorsa.



Nell'esempio sopra possiamo vedere come l'assegnamento di R_2 a T_2 metta il sistema in uno stato di rischio, cosa che non sarebbe accaduta facendo accedere prima alla risorsa il thread T_1.

BANKER'S ALGORITHM

Il resource-allocation-graph non può però essere applicato a sistemi di allocazione delle risorse con multiple istanze per ogni tipo di risorsa. Andiamo quindi ad analizzare un algoritmo meno efficiente del precedente, ma molto più flessibile e generico: il **banker's algorithm**.

In quest'algoritmo, quando un nuovo thread entra nel sistema, **deve dichiarare il massimo numero di istanze per tipo di risorsa** di cui potrebbe aver bisogno. Questi valori ovviamente **non devono eccedere il massimo numero di istanze** di risorse disponibili nel sistema.

Quando un thread richiede quindi un set di risorse, l'algoritmo deve determinare se l'allocazione di tali risorse lascerà il sistema in uno stato sicuro: in caso positivo, il thread potrà accedere alle risorse; in caso negativo il thread dovrà attendere fin quando qualche altro thread rilasci abbastanza risorse.

Al fine di poter effettuare tali controlli, abbiamo bisogno delle seguenti strutture dati (dove $\boldsymbol{n}$ **è il numero di thread del sistema** ed $\boldsymbol{m}$ **è il numero di tipi di risorse**):

- **Available**
  Vettore di lunghezza m che indica il numero di risorse libere per ogni tipo.

$$\boldsymbol{Available[j] = numero\ di\ risorse\ di\ tipo\ R_j}$$

- **Max**
  Matrice $n * m$ che definisce il massimo numero di risorse per tipo che ciascun thread T_i potrebbe richiedere.

  $$\boldsymbol{Max[i][j] = massimo\ numero\ di\ risorse\ di\ tipo\ R_j\ che\ T_i\ può\ richiedere}$$

- **Allocation**
  Matrice $n * m$ che definisce il numero di risorse per ogni tipo attualmente allocate per ogni thread.

  $$\boldsymbol{Allocation[i][j] = numero\ di\ risorse\ di\ tipo\ R_j\ allocate\ dal\ thread\ T_i}$$

- **Need**
  Matrice $n * m$ che indica il rimanente numero di risorse di cui ciascun thread potrebbe necessitare.

  $$\boldsymbol{Need[i][j] = numero\ di\ risorse\ di\ tipo\ R_j\ che\ T_i\ potrebbe\ ancora\ richiedere}$$

Notiamo che

$$\boldsymbol{Need = Max - Allocation}$$

Prima di procedere, definiamo alcune notazioni per semplificare i passaggi:

- Dati due vettori V_1 e V_2, diremo che $V_1 \leq V_2 \Leftrightarrow V_1[i] \leq V_2[i]\ \forall\ i = 1, 2, \dots, n$.
- Tratteremo ogni riga delle matrici come vettori, e ci riferiremo ad essi come, ad esempio $Allocation_i$ o $Need_i$.

Il banker's algorithm si divide in 2 passaggi, che risolveremo con relativi algoritmi:

- **Safety Algorithm**
  Per definire se un sistema sia o meno in uno stato sicuro.
- **Resource-Request Algorithm**
  Per definire quali richieste possano essere soddisfatte in modo sicuro.

BANKER'S ALGORITHM: SAFETY ALGORITHM

Passaggi:

I. Dato il vettore $Finish$, di lunghezza m, inizializzato come
   $Finish[i] = False \qquad \forall\ i = 1, 2, \dots, n - 1$

II. Troviamo un indice i tale che:
   a. $Finish[i] == False$
   b. $Need_i \leq Available$

III. Aggiorniamo le variabili
   $Available\ +=\ Allocation_i$
   $Finish[i] = True$

IV. Se $Finish[i] == True \quad \forall\ i$:
   Il sistema è sicuro
   Altrimenti:
   Torna al punto II

BANKER'S ALGORITHM: RESOURCE-REQUEST ALGORITHM

Data la matrice $Request$ dove $Request_i$ definisce il numero di risorse per tipo R_j che il thread T_i sta richiedendo.

Quando una richiesta viene effettuata, vanno fatti i seguenti controlli:

I. Se $Request_i \leq Need_i$:
   vai al passaggio II
   Altrimenti:
   solleva una condizione di errore (il thread ha superato il massimo di risorse richiedibili)

II. Se $Request_i \leq Available_i$:
   vai al passaggio III
   Altrimenti:
   T_i deve aspettare fin quando le risorse non siano disponibili

III. Si assume che il sistema abbia allocato le risorse richieste al thread T_i, vengono quindi modificate le seguenti variabili:
   $Available \ -= \ Request_i$
   $Allocation_i \ += \ Request_i$
   $Need_i \ -= \ Request_i$

   Se lo stato risultante è sicuro, la transazione è completata e T_i alloca le sue risorse. Altrimenti T_i deve aspettare che la rich
   iesta possa essere soddisfatta, viene ripristinato lo stato di allocazione precedente.

BANKER'S ALGORITHM: ESEMPIO

Prendiamo come esempio il seguente stato computazionale di un sistema con 5 thread.

	Max	Allocation		Need
	A B C D	A B C D		A B C D
P0	0 0 1 2	0 0 1 2	P0	
P1	1 7 5 0	1 0 0 0	P1	Available
P2	2 3 5 6	1 3 5 4	P2	A B C D
P3	0 6 5 2	0 6 3 2	P3	1 5 2 0
P4	0 6 5 6	0 0 1 4	P4	

Prima di tutto troviamo i vettori di $Need$ con l'equazione $\boldsymbol{Need = Max \ - Allocation}$.

	Max	Allocation		Need	
	A B C D	A B C D		A B C D	
P0	0 0 1 2	0 0 1 2	P0	0 0 0 0	Available
P1	1 7 5 0	1 0 0 0	P1	0 7 5 0	A B C D
P2	2 3 5 6	1 3 5 4	P2	1 0 0 2	1 5 2 0
P3	0 6 5 2	0 6 3 2	P3	0 0 2 0	
P4	0 6 5 6	0 0 1 4	P4	0 6 4 2	

Adesso, per controllare di essere in uno stato sicuro, applichiamo il safety algorithm: per ogni thread controlliamo se in quel dato momento sia possibile allocare le risorse richieste ($Need \leq Available$), altrimenti andiamo al thread successivo, tornando al thread in attesa solo quando ci saranno le risorse disponibili per poterlo eseguire.

NB!
Si dà per assunto che un thread dopo che venga eseguito rilasci tutte le sue risorse, aggiornando la variabile $Available$ (nella realtà il processo successivo attenderebbe comunque la fine del precedente per iniziare, se quest'ultimo sta ancora finendo di usare delle risorse necessarie alla sua eseuzione)

	Max	Allocation	Need
	A B C D	A B C D	A B C D
P0	0 0 1 2	0 0 1 2	0 0 0 0
P1	1 7 5 0	1 0 0 0	0 7 5 0
P2	2 3 5 6	1 3 5 4	1 0 0 2
P3	0 6 5 2	0 6 3 2	0 0 2 0
P4	0 6 5 6	0 0 1 4	0 6 4 2

Available
A B C D
1 5 2 0

Eseguiamo dunque il safety:

I. Current thread = P0
   Need = 0 0 0 0
   Available = 1 5 2 0
   Need ≤ Available ⇒ SI
   Available += Allocation ⇒ 1 5 3 2

II. Current thread = P1
   Need = 0 7 5 0
   Available = 1 5 3 2
   Need ≤ Available? ⇒ NO

III. Current thread = P2
   Need = 1 0 0 2
   Available = 1 5 3 2
   Need ≤ Available? ⇒ SI
   Available += Allocation ⇒ 2 8 8 6

IV. Current thread = P3
   Need = 0 0 2 0
   Available = 2 8 8 6
   Need ≤ Available? ⇒ SI
   Available += Allocation ⇒ 2 14 11 8

V. Current thread = P4
   Need = 0 6 4 2
   Available = 2 14 11 8
   Need ≤ Available? ⇒ SI
   Available += Allocation ⇒ 2 14 12 12

VI. Current thread = P1
   Need = 0 7 5 0
   Available = 2 14 12 12
   Need ≤ Available? ⇒ SI
   Available += Allocation ⇒ 3 14 12 12

Abbiamo quindi trovato che la sequenza <P0, P2, P3, P4, P1> è una sequenza sicura, stiamo quindi in uno stato sicuro.

Adesso, supponiamo di avere un vettore $Request_1 = 0\ 1\ 2\ 0$, che ci indica set di risorse aggiuntive che sta richiedendo il thread P1. Eseguiamo il Resource-Request Algorithm.

Valutiamo innanzitutto se la richiesta è legale

$$Request_1 \leq Need_1 \Rightarrow 0\ 1\ 2\ 0 \leq 0\ 7\ 5\ 0$$

A questo punto controlliamo se la richiesta può essere soddisfatta immediatamente o se bisogna attendere.

$$Request_1 \leq Available \Rightarrow 0\ 1\ 2\ 0 \leq 1\ 5\ 2\ 0$$

La risorsa può essere assegnata subito, aggiorniamo quindi $Available$ e rieseguiamo il safety algorithm per valutare se siamo ancora in uno stato sicuro.

$$Available -= Request_1 \Rightarrow 1\ 4\ 0\ 0$$

	Max				Allocation				Need				Available			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	P0	0	0	0	0	1	4	0	0
P1	1	7	5	0	1	1	2	0	P1	0	6	3	0				
P2	2	3	5	6	1	3	5	4	P2	1	0	0	2				
P3	0	6	5	2	0	6	3	2	P3	0	0	2	0				
P4	0	6	5	6	0	0	1	4	P4	0	6	4	2				

Eseguiamo nuovamente il safety:

I. Current thread = P0
   Need = 0 0 0 0
   Available = 1 4 0 0
   Need ≤ Available ⇒ SI
   Available += Allocation ⇒ 1 4 1 2

II. Current thread = P1
   Need = 0 6 3 0
   Available = 1 4 1 2
   Need ≤ Available? ⇒ NO

III. Current thread = P2
   Need = 1 0 0 2
   Available = 1 4 1 2
   Need ≤ Available? ⇒ SI
   Available += Allocation ⇒ 2 7 6 6

IV. Current thread = P3
   Need = 0 0 2 0
   Available = 2 7 6 6
   Need ≤ Available? ⇒ SI
   Available += Allocation ⇒ 2 13 9 8

V. Current thread = P4
   Need = 0 6 4 2
   Available = 2 13 9 8
   Need ≤ Available? ⇒ SI
   Available += Allocation ⇒ 2 13 10 12

VI. Current thread = P1
   Need = 0 6 3 0
   Available = 2 13 10 12
   Need ≤ Available? ⇒ SI
   Available += Allocation ⇒ 3 14 12 12

A questo punto possiamo affermare che nonostante l'assegnamento di $Request_1$, siamo comunque in uno stato sicuro, in quanto esiste la sequenza sicura <P0, P2, P3, P4, P1>.

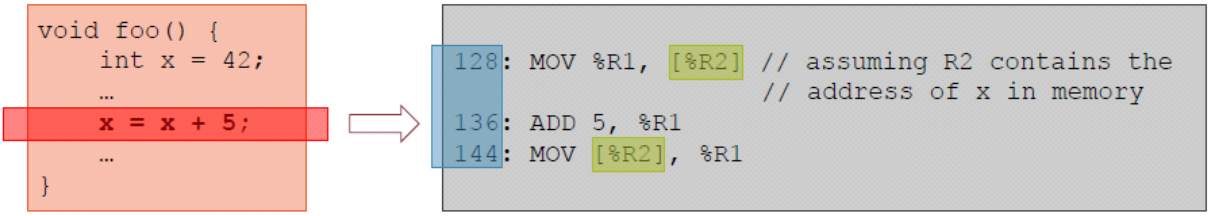
3 Gestione della memoria

Possiamo considerare la memoria come un **grande array di byte, ciascuno con il proprio indirizzo**. La CPU fa il fetch delle istruzioni dalla memoria in base al valore del program counter. Queste istruzioni possono causare un caricamento aggiuntivo e l'archiviazione in indirizzi di memoria specifici.

> Un ciclo di esecuzione di istruzioni tipico, ad esempio, prima fa il fetch di un'istruzione dalla memoria. L'istruzione viene quindi decodificata e può causare il fetch degli operandi dalla memoria. Dopo che l'istruzione è stata eseguita , i risultati possono essere memorizzati nuovamente.

L'unità di memoria vede solo un flusso di indirizzi di memoria e non sa come vengono generati o a cosa servono.

ESEMPIO PRATICO



1. Esegue il fetch dell'istruzione all'indirizzo `128`
2. Esegue l'istruzione di load dall'indirizzo `[%R2]` (ad esempio, 1234)
3. Esegue il fetch dell'istruzione all'indirizzo `136`
4. Esegue l'istruzione di addizione (nessun riferimento alla memoria)
5. Esegue il fetch dell'istruzione all'indirizzo `144`
6. Esegue l'istruzione store all'indirizzo `[%R2]` (1234)

La memoria principale e i registri integrati in ciascun core di elaborazione sono l'unica memoria generica a cui la CPU può accedere direttamente.

Esistono istruzioni che accettano indirizzi di memoria come argomenti, ma nessuna che accetta indirizzi del disco. Pertanto, qualsiasi istruzione in esecuzione e qualsiasi dato utilizzato dalle istruzioni deve trovarsi in uno di questi dispositivi di archiviazione ad accesso diretto. Se i dati non sono in memoria, devono essere spostati lì prima che la CPU possa operare su di essi.

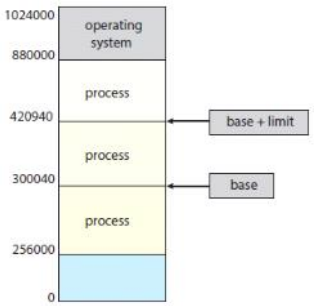
Per il corretto funzionamento del sistema, dobbiamo **proteggere il sistema operativo dall'accesso da parte dei processi utente**, quindi dobbiamo **proteggere i processi utente l'uno dall'altro**. Questa protezione deve essere fornita dall'hardware, perché il sistema operativo di solito non interviene tra la CPU e i suoi accessi alla memoria.

Una possibile implementazione potrebbe essere la seguente:
Per prima cosa dobbiamo assicurarci che ogni processo abbia uno spazio di memoria separato. In questo modo possiamo proteggere i processi l'uno dall'altro, ed è fondamentale per l'esecuzione simultanea dei processi.

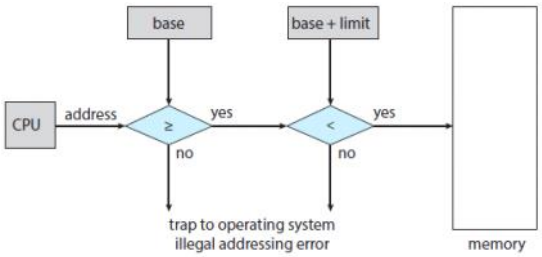
Per separare gli spazi di memoria, dobbiamo essere capaci di determinare l'intervallo di indirizzi legali a cui il processo può accedere e di garantire che il processo possa accedere solo a questi indirizzi legali.

Possiamo fornire questa protezione utilizzando **due registri**, solitamente **una base e un limite**, come illustrato in figura. Il **base register** contiene l'indirizzo di memoria fisica legale più piccolo; il **limit register** contiene la dimensione dell'intervallo.

> Ad esempio, se il base register contiene 300040 e il limit register contiene 120900, il programma può accedere legalmente a tutti gli indirizzi da 300040 a 420939 (compreso).



La protezione dello spazio di memoria si ottiene facendo in modo che l'hardware della CPU confronti ogni indirizzo generato in modalità utente con i registri.



Qualsiasi tentativo da parte di un programma in esecuzione in modalità utente di accedere alla memoria del sistema operativo o alla memoria di altri utenti, provoca una trap per il sistema operativo, che tratta il tentativo come un errore irreversibile (**fatal error**). Questo schema previene che un programma in modalità utente modifichi (accidentalmente o deliberatamente) di modificare il codice o le strutture dati del sistema operativo o di un altro utente.

I registri base e limit possono essere caricati solo dal sistema operativo, che utilizza un istruzione privilegiata. Il sistema operativo, in esecuzione in modalità kernel, ha accesso illimitato sia alla memoria del sistema operativo che alla memoria degli utenti.

Questa disposizione consente al sistema operativo di caricare i programmi utente nella memoria degli utenti, di fare un **dump** (scaricarli) **in caso di errori**, di accedere e modificare i parametri delle chiamate di sistema, di eseguire operazioni di I/O da e verso la memoria dell'utente, e di fornire molti altri servizi.

ADDRESS BINDING

Prima di parlare di address binding è necessario dare delle definizioni:

- **Relocatable object file**
  L'output di un compilatore i cui i contenuti possono essere caricati in qualsiasi posizione della memoria fisica.
- **Linker**
  Un servizio di sistema che combina dei relocatable object files in un singolo file binario eseguibile.
- **Loader**
  Un servizio di sistema che carica un file binario eseguibile in memoria, dove può essere eseguito su un core della CPU. Durante la fase di loading, possono essere inclusi anche altri object file o librerie, come la libreria standard C o math.
- **Relocation**
  Un'attività associata al linking e al loading che assegna gli indirizzi finali alle parti del programma e regola il codice e i dati nel programma in modo che corrispondano a tali indirizzi.

Di solito, un programma risiede su un disco come file binario eseguibile. Per essere eseguito, l'istanza del programma (il processo) deve essere caricata in memoria, dove diventa idonea per l'esecuzione su una CPU disponibile. Mentre il processo viene eseguito, la CPU accede alle istruzioni e ai dati dalla memoria. Alla fine, il processo termina e la sua memoria viene ripristinata per essere utilizzata da altri processi.

La maggior parte dei sistemi consente a un processo utente di risiedere in qualsiasi parte della memoria fisica. Pertanto, sebbene lo spazio degli indirizzi possa iniziare da 00000, non è necessario che il primo indirizzo del processo utente sia 00000.

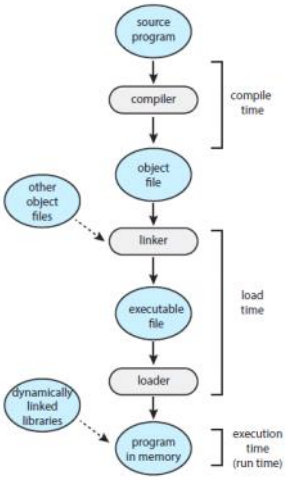
Nella maggior parte dei casi, un programma utente passa attraverso diversi passaggi prima di essere eseguito. Gli indirizzi possono essere rappresentati in modi diversi durante questi passaggi. Gli indirizzi nel programma sono generalmente simbolici (come la variabile "count" ad esempio).

Un compilatore in genere associa questi indirizzi simbolici a indirizzi rilocabili. Il linker o il loader a loro volta associano gli indirizzi rilocabili agli indirizzi assoluti (come ad esempio "74014").

L'associazione di istruzioni e dati agli indirizzi di memoria può essere eseguita in qualsiasi fase del percorso:

- **Compile time**: Se sappiamo dove risiederà in memoria il processo in fase di compilazione, allora può essere generato il cosiddetto codice assoluto (absolute code). Per esempio se sappiamo che un processo risiederà a partire dalla posizione R, il codice generato dal compilatore inizierà in quella posizione e poi si estenderà da lì. Se la posizione di partenza cambia, allora sarà necessario ricompilare il codice.

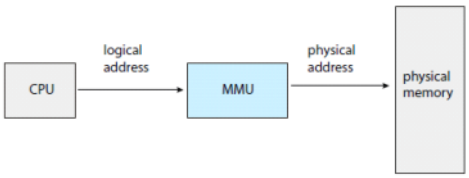
- **Load time**: Se al momento della compilazione non è noto dove risiederà il processo in memoria, il compilatore deve generare codice rilocabile (Codice con collegamenti agli indirizzi di memoria che vengono modificati a loading time per riflettere la posizione del codice nella memoria principale). In questo caso l'associazione è ritardata fino al load time. Se l'indirizzo di partenza cambia, è sufficiente ricaricare lo user code per incorporare questo valore modificato.



- **Execution time**: Se un processo può essere spostato da un segmento di memoria ad un altro durante la fase di execution, allora l'associazione deve essere ritardata fino alla fase di execution time. Affinché questo schema funzioni, deve essere disponibile un hardware speciale chiamato MMU come verrà discusso più avanti. La maggior parte dei sistemi operativi usa questo metodo.

3.1 LOGICAL VERSUS PHYSICAL ADDRESS SPACE

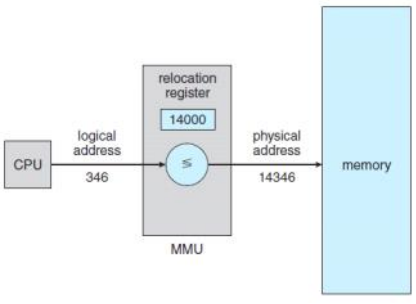
Il registro degli indirizzi di memoria è comunemente indicato come indirizzo fisico. A tempo di compilazione e a tempo di caricamento vengono generati indirizzi logici e fisici identici. Tuttavia, lo schema di associazione degli indirizzi in fase di esecuzione comporta indirizzi logici e fisici diversi. In questo caso, di solito ci riferiamo all'indirizzo logico come indirizzo virtuale. Utilizziamo l'indirizzo logico e l'indirizzo virtuale in modo intercambiabile.



L'insieme di tutti gli indirizzi logici generati da un programma è uno **spazio di indirizzi logici**. L'insieme di tutti gli indirizzi fisici corrispondenti a questi indirizzi logici è uno **spazio di indirizzi fisici**. Pertanto, in fase di esecuzione nello schema di associazione degli indirizzi, gli spazi degli indirizzi logici e fisici differiscono.

La mappatura dagli indirizzi virtuali a quelli fisici viene eseguita in fase di esecuzione da un dispositivo hardware chiamato memory-management unit (MMU).

Illustriamo questa mappatura con un semplice schema che è una generalizzazione dello schema del base register descritto precedentemente. Il **base register ora è chiamato relocation register**. Come si può vedere nella figura qui di fianco, ogni volta che viene generato un indirizzo da un processo utente, prima che questo indirizzo venga inviato in memoria, viene aggiunto al valore che contiene il relocation register. Per esempio un accesso alla posizione 346 è mappato alla posizione 14346.



Il programma utente non accede mai agli indirizzi fisici. Il programma può creare un puntatore alla posizione 346, memorizzarlo, manipolarlo e confrontarlo con altri indirizzi. Il **programma utente si occupa degli indirizzi logici**, mentre l'**MMU converte gli indirizzi logici in indirizzi fisici**.

Ora abbiamo due diversi tipi di indirizzi: indirizzi logici (che vanno da 0 a max) e indirizzi fisici (che vanno da R + 0 a R + max per un valore base R).

Per una corretta gestione della memoria è necessario separare gli indirizzi logici dagli indirizzi fisici.

3.2 DYNAMIC LOADING

Finora è stato necessario che l'intero programma e tutti i dati di un processo fossero nella memoria fisica affinché il processo potesse essere eseguito. La dimensione di un processo è stata quindi limitata alla dimensione della memoria fisica.

Per ottenere un migliore utilizzo dello spazio di memoria, possiamo utilizzare il **dynamic loading**. Con il dynamic loading, una routine **non viene caricata finché non viene chiamata**. Tutte le routine sono conservate su disco in un formato di caricamento rilocabile. Il programma principale viene caricato in memoria ed eseguito.

Quando una routine deve chiamare un'altra routine, la routine chiamante verifica innanzitutto se l'altra routine è stata caricata. In caso contrario, viene chiamato il relocatable linking loader per caricare la routine desiderata in memoria e aggiornare le tabelle degli indirizzi del programma. Dopodiché il controllo viene passato alla routine appena caricata.

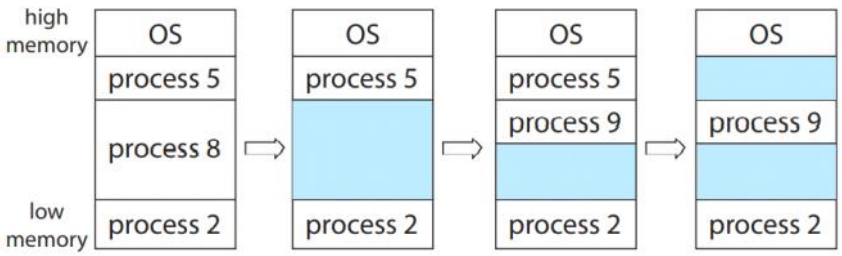
Il vantaggio del dynamic loading è che una routine viene caricata solo quando è necessaria. Questo metodo è particolarmente utile quando sono necessarie grandi quantità di codice per gestire casi che si verificano raramente, come le error routines. In una situazione simile, anche se la dimensione totale del programma può essere grande, la parte utilizzata (e quindi caricata) può essere molto più piccola.

Il dynamic loading non richiede un supporto speciale da parte del sistema operativo, ma è responsabilità degli utenti progettare i propri programmi in modo da trarre vantaggio da tale metodo. I sistemi operativi possono tuttavia aiutare il programmatore fornendo librerie di routine per implementare il dynamic loading.

3.3 CONTIGUOUS MEMORY ALLOCATION

La memoria principale deve accogliere, come abbiamo già visto, sia il sistema operativo che i vari processi utente, proprio per questo la memoria è generalmente **divisa in due macro-aree** per sopperire a questa necessità. Nella maggior parte dei sistemi operativi (come Linux e Windows), il processo del sistema operativo viene caricato nella parte alta della memoria principale, analizzeremo quindi esclusivamente questo caso.

Il sistema lavora quasi sempre con molti processi diversi caricati in memoria allo stesso momento. Dobbiamo quindi capire come allocare la memoria disponibile ai programmi che stanno aspettando di essere caricati in memoria. Nell' **allocazione di memoria contigua** (per "contiguo" si intende "prossimo/vicino"), **ogni processo è contenuto in una singola sezione di memoria** che è contigua alla sezione contente il processo successivo.



Quando nuovi processi entrano nel sistema, il sistema operativo tiene conto dei requisiti di memoria di ogni processo e della quantità di spazio di memoria disponibile, al fine di determinare quale processo viene allocato e dove. Una volta che ad un processo viene allocato lo spazio, viene caricato in memoria, dove potrà competere per la CPU. Quando un processo termina, **rilascia il suo spazio di memoria occupato**, il quale sarà poi allocato ad un altro processo dal sistema operativo.

Quando non c'è sufficiente memoria libera ci sono varie opzioni, analizziamone un paio:

1. **Rifiutare il processo** e fornire un appropriato messaggio d'errore.
2. **Piazzare i processi in una coda d'attesa**, dalla quale, quando la memoria verrà liberata, il sistema operativo determinerà quale processo potrà essere caricato in memoria.

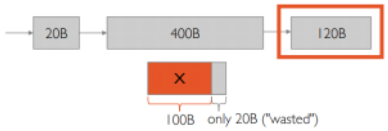
In genere, come menzionato, i blocchi di memoria disponibile comprendono un insieme di ***buchi*** di varie dimensioni sparpagliati in giro per la memoria. Quando arriva un nuovo processo, **il sistema scansiona l'insieme alla ricerca di uno spazio abbastanza grande per le esigenze del processo**. Se lo spazio è troppo grande viene diviso in due parti: una allocata al processo entrante, l'altra ritornata all'insieme dei buchi. Quando un processo termina, **rilascia i suoi blocchi di memoria** che vengono inseriti nuovamente nell'insieme dei buchi. Se il nuovo buco è adiacente ad altri, **i buchi adiacenti vengono uniti per formarne uno più grande**. Questa procedura è una particolare istanza del generale **dynamic storage-allocation problem**, che si preoccupa di come soddisfare un richiesta di grandezza n da una lista di buchi liberi.
Ci sono molte soluzioni a questo problema, ma vediamo le più comuni:

- **First-Fit**

  Viene allocato **il primo spazio** abbastanza grande. (generalmente la soluzione più veloce)

- **Best-Fit**

  Viene allocato **il più piccolo spazio** abbastanza grande.



- **Worst-Fit**

  Viene allocato **lo spazio più grande**. Potrebbe sembrare controproducente, ma un approccio di questo tipo aumenta la probabilità che la porzione rimanente sarà utilizzabile per soddisfare le future richieste.

FRAMMENTAZIONE

È il problema che si verifica **quando possiamo caricare in memoria meno processi di quanti la memoria ne supporterebbe al suo massimo utilizzo**. Ciò è dovuto a spazi liberi troppo piccoli per caricare un processo, o a spazi "sprecati" per altri processi.

Abbiamo infatti 2 tipi di frammentazione:

- **Frammentazione esterna**

  Si verifica **quando c'è abbastanza spazio di memoria totale per soddisfare una richiesta, ma gli spazi disponibili non sono contigui**. Problema molto comune nelle politiche di allocazione della memoria First-Fit e Worst-Fit.

- **Frammentazione interna**

  Si verifica **quando un processo richiede un certo spazio di memoria e gliene viene assegnato uno leggermente più grande per evitare che il sistema debba tenere conto di un buco di memoria troppo piccolo da essere rilevante**. Questa frammentazione è molto comune quando la memoria viene divisa in blocchi di dimensioni fisse e ai processi viene assegnato un blocco in base alla loro dimensione (ciò significa che quasi mai quel blocco sarà esattamente della dimensione richiesta dal processo).

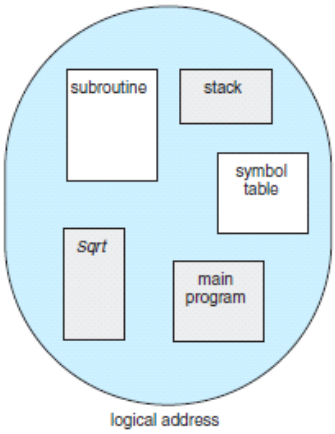
Una soluzione per la frammentazione esterna è quella della **compattazione** (il cui obbiettivo finale è quello di **riorganizzare il contenuto della memoria al fine di unificare tutti gli spazi liberi di memoria separati in un unico grande blocco di memoria libera**). La compattazione non è però sempre possibile se la relocation è statica e viene eseguita a tempo di assemblamento o a tempo di caricamento. La compattazione è quindi **possibile solo se la relocation è dinamica e viene effettuata a tempo di esecuzione**. L'algoritmo di compattazione più semplice è quello di **partire da un estremo e spostare ogni blocco al limite della memoria libera**. Un altro metodo di risoluzione è quello del **paging**, che **permette allo spazio di indirizzamento logico di essere non-contiguo**. Questo permette ai processi di essere allocati dovunque ci sia memoria libera (tale metodo verrà affrontato in seguito).

3.4 SEGMENTATION

La visione della memoria che ha un utente o un programmatore non è la stessa della memoria fisica. Trattare la memoria in termini di proprietà fisiche infatti è scomodo sia per il sistema operativo che per il programmatore. L'hardware deve fornire un **meccanismo tale che quello che viene visto dal programmatore venga mappato sulla memoria fisica**. La **segmentazione** fornisce questa possibilità.

La maggior parte dei programmatori preferisce vedere la memoria come una raccolta di segmenti di dimensioni variabili, senza alcun ordinamento. La memoria va invece considerata come un array lineare di byte, alcuni contenti informazioni altri dati.

Il programmatore immagina un programma come un insieme di metodi, procedure o funzioni che può anche includere varie strutture dati come array, oggetti, stack, variabili ecc.



Ciascuno di questi elementi è indicato per nome. Il programmatore parla di "lo stack", "la libreria math" ecc. , senza preoccuparsi di quali indirizzi in memoria occupino questi elementi.

I segmenti variano in lunghezza, ed essa è definita dal suo scopo nel programma. Gli elementi all'interno sono identificati dal loro offset dall'inizio del segmento.

La **segmentazione** è uno **schema di gestione della memoria che supporta questa visione** della memoria da parte del programmatore.

Uno spazio di indirizzi logici è una raccolta di segmenti. Ogni segmento ha un **nome** e una **lunghezza**. Gli **indirizzi specificano** sia il **nome del segmento** che l'**offset all'interno del segmento**.

Per semplicità di implementazione, i segmenti sono numerati e sono indicati da un numero di segmento, piuttosto che da un nome. Pertanto, un indirizzo logico è costituito da due tuple:

```
<segment-number, offset>.
```

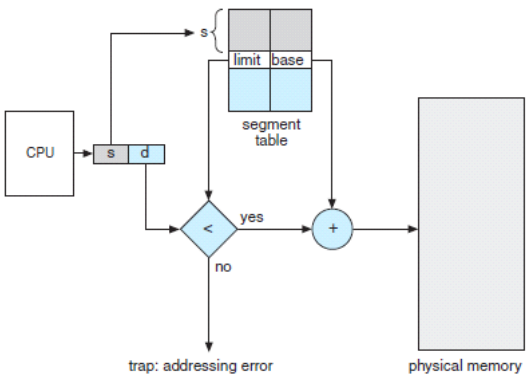
Normalmente, quando un programma viene compilato, il compilatore costruisce automaticamente i segmenti che riflettono il programma di input. Un compilatore C potrebbe creare segmenti separati nel seguente modo:

1. Il codice
2. Le variabili globali
3. L'heap, da cui viene allocata la memoria
4. Gli stack usati da ogni thread
5. La libreria standard C

Alle librerie collegate durante la fase di compilazione, potrebbero essere assegnati segmenti separati. Il loader prende tutti questi segmenti e assegna ad ognuno il proprio numero di segmento.

SEGMENTATION HARDWARE

Sebbene il programmatore possa ora fare riferimento agli oggetti nel programma tramite un indirizzo bidimensionale, la memoria fisica effettiva è ancora, ovviamente, una sequenza unidimensionale di byte. Pertanto, dobbiamo definire un'**implementazione per mappare indirizzi bidimensionali definiti dall'utente in indirizzi fisici unidimensionali**.



Questa mappatura è effettuata da una **segment table**. Ogni voce nella tabella dei segmenti ha un **segment base** e un **segment limit**. Il base segment contiene l'**indirizzo fisico iniziale** in cui il segmento risiede in memoria, mentre il segment limit specifica la **lunghezza del segmento**.

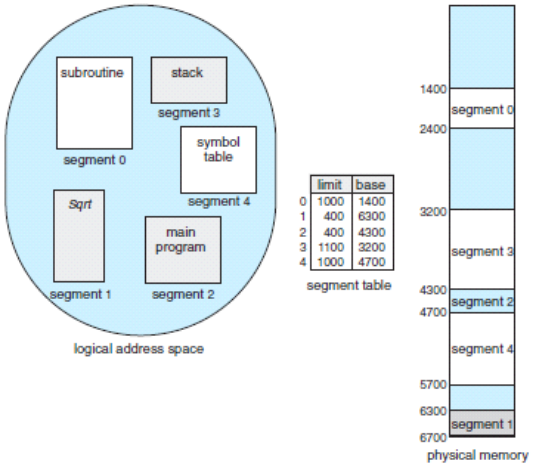
Nella figura qui di fianco è illustrato l'uso della segment table. Un indirizzo logico consiste di due parti: un numero di segmento s, e un offset in quel segmento d.

Il **numero del segmento** viene utilizzato come **indice della tabella dei segmenti**. **L'offset *d*** dell'indirizzo logico deve essere compreso tra 0 e il segment limit. In caso contrario, si genera una trap nel sistema operativo (tentativo di indirizzamento logico oltre la fine del segmento).

Quando un offset è legale, viene aggiunto alla base del segmento per produrre l'indirizzo nella memoria fisica del byte desiderato. La tabella dei segmenti è quindi essenzialmente un array di coppie di base-limit registers.

> Ad esempio si consideri la situazione mostrata nella figura. Abbiamo cinque segmenti numerati *da 0 a 4*. I segmenti sono memorizzati nella memoria fisica come mostrato.

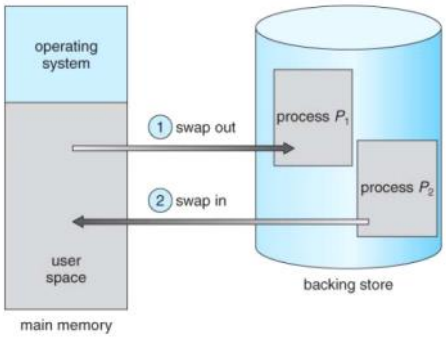
La segment table ha una voce separata per ogni segmento, che fornisce l'indirizzo iniziale del segmento nella memoria fisica (o base) e la lunghezza di quel segmento (o limite).

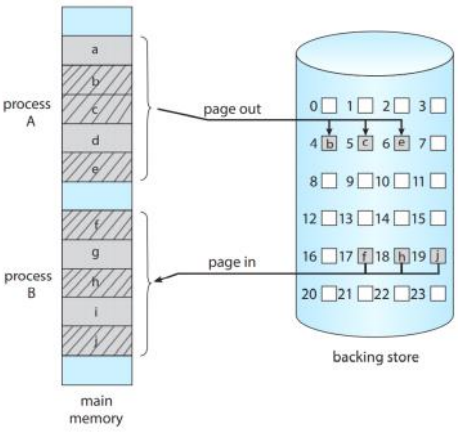


> Ad esempio, il segmento *2* è lungo *400 byte* e inizia alla posizione *4300*. Pertanto, un riferimento al byte *53* del segmento *2* viene mappato sulla posizione *4300 + 53 = 4353*.
> Un riferimento al segmento *3, byte 852*, viene mappato a *3200 (la base del segmento 3) + 852 = 4052*.
>
> Un riferimento al byte *1222* del segmento *0* comporterebbe una trap per il sistema operativo, poiché questo segmento è lungo solo *1.000 byte*.

3.5 SWAPPING

Le istruzioni dei processi, e i dati sul quale operano, devono essere caricati in memoria per essere eseguiti. Ad ogni modo, un processo, o una sua porzione, può essere temporaneamente spostata fuori dalla memoria in un **backing store** (memoria di supporto), per poi essere successivamente riallocata nella memoria principale per continuare l'esecuzione del processo. Questa azione **viene chiamata swapping**.
Un processo ha bisogno di risiedere fisicamente in memoria solo se la CPU sta eseguendo le sue istruzioni e accedendo ai suoi dati. Se un processo viene bloccato non c'è infatti bisogno che resti in memoria. In sostanza **lo swapping consente alla memoria occupata dai processi di eccedere al reale spazio fisico nella memoria principale**.





Lo **standard swapping** comporta lo **spostamento dell'intero processo tra la memoria principale e il backing store** (generalmente implementato con una memoria rapida secondaria). Lo standard swapping veniva utilizzato nei tradizionali sistemi UNIX, ma generalmente **non viene più usato** nei sistemi operativi moderni. La maggior parte dei sistemi operativi, inclusi Linux e Windows, utilizza una variante dello swapping, **nel quale il processo viene suddiviso in pagine (sezioni del processo)**, e lo swap viene eseguito sulle pagine del processo invece che sull'intero processo.

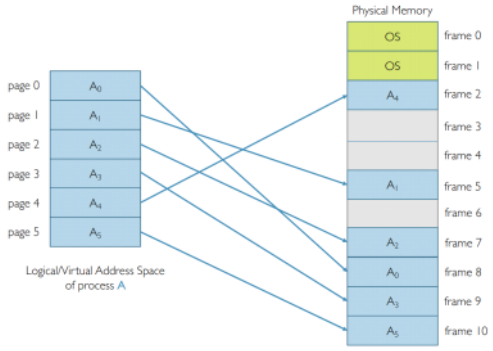
3.6 PAGING

Fin ora abbiamo parlato principalmente di schemi di gestione della memoria i quali richiedevano che l'indirizzo fisico di un processo fosse contiguo. Introdurremo adesso il **paging**, uno schema di gestione della memoria **che permette allo spazio di indirizzamento fisico di un processo di essere non contiguo**. Il paging **evita la frammentazione esterna e il conseguente bisogno di compattazione**.

Il paging offre difatti numerosi vantaggi e, proprio per questo, viene utilizzato sulla maggior parte dei sistemi, dai grandi server ai dispositivi mobili. l'implementazione del paging avviene attraverso la cooperazione tra sistema operativo e hardware.

IMPLEMENTAZIONE

L'implementazione di base richiede la divisione della memoria fisica **in blocchi di dimensione fissata**, chiamati **frame**, e la divisione della memoria logica **in blocchi delle stesse dimensioni**, chiamati **pagine**. Quando un processo deve essere eseguito le sue pagine vengono caricate nei frame liberi dalla loro sorgente (che sia un file system o un backing store).



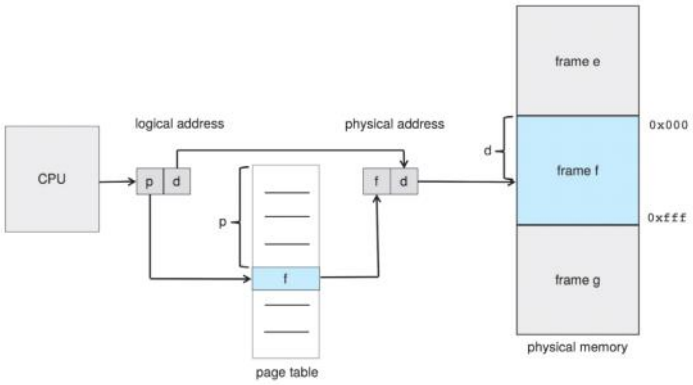
Il sistema operativo ha 2 principali responsabilità:

- **Mappatura** tra pagine logiche e frame fisici
- **Traduzione** da indirizzi logici a indirizzi fisici

Ogni indirizzo logico generato dalla CPU è diviso in 2 parti: **page number** (numero della pagina in cui risiede l'indirizzo) e **page offset** (posizione rispetto all'inizio della pagina). Per assicurare il corretto (ed efficiente funzionamento), l'OS utilizza uno strumento chiamato **page table**. La page è contenuta nel PCB di ogni processo e mappa le pagine del processo nei frame in memoria.

Di seguito i passaggi eseguiti dalla MMU per tradurre gli indirizzi generati dalla CPU in indirizzi fisici:

a. Viene estratto il numero di pagina *p* e usato come un indice nella page table
b. Viene estratto il corrispettivo numero del frame *f* dalla page table
c. Si rimpiazza il numero di pagina *p* nell'indirizzo logico con il numero del frame *f* (l'offset *d* viene mantenuto invariato)



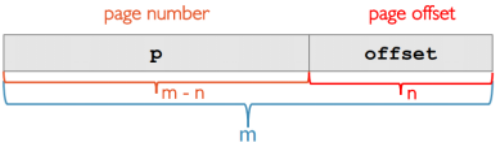
Supponiamo di avere 50B di memoria libera per i processi utente. Assumiamo di star usando pagine (frame) di grandezza S = 10B. Ogni processo può generare indirizzi virtuali nel range [0, 49]. Supponiamo che un processo generi un indirizzo virtuale x = 27.

$$p = \left\lfloor \frac{x}{s} \right\rfloor = \left\lfloor \frac{27}{10} \right\rfloor = 2 \qquad d = x \,\%\, s = 27 \,\%\, 10 = 7$$

La traduzione degli indirizzi richiede quindi le operazioni divisione intera e modulo algebrico.

La grandezza e il numero delle pagine (e dei frame) è determinata dall'architettura. Generalmente la grandezza è una potenza di 2 che va dai 4KB a 1GB per pagina. La potenza di 2 rende la traduzione da virtuale a fisico semplice e priva di divisioni intere e moduli.
Diciamo che un indirizzo virtuale è formato da m bit (quindi lo spazio di indirizzamento è lungo 2^m e va da 0 a $2^m - 1$).

Assumiamo di avere pagine di grandezza 2^n con $n < m$. **Gli $m - n$ bit più alti** dell'indirizzo virtuale indicano il **numero di pagina**. Gli n **bit più bassi** rappresentano l'**offset**.

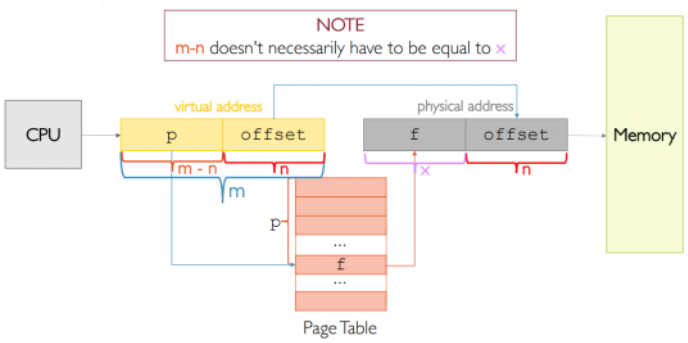


Supponiamo di avere una memoria virtuale e una memoria fisica, entrambe di grandezza M = 1024B (1 KiB).
Assumiamo che la grandezza delle nostre pagine/frame sia S=16B.

D1: Quanti bit ci servono per un indirizzo fisico/virtuale (assumendo di avere un indirizzamento al byte)?
R1: Servono 10 bit per indirizzare 1024B.

D2: Quanto è grande la page table?
R2: T = M / S = 1024 byte di memoria / 16 byte per pagina = $2^{10} / 2^4 = 64$ pagine.

D3: Quanti bit ci servono rispettivamente per rappresentare il page number e il page offset?
R3: I nostri indirizzi fisici sono di 10bit.
Abbiamo bisogno di $n = 4$ *bit* per rappresentare l'offset se ogni pagina/frame è grande 16B.
Servono $m\text{-}n = 6$ *bit* per rappresentare il numero di pagina, in quanto abbiamo 64 pagine.



TLB

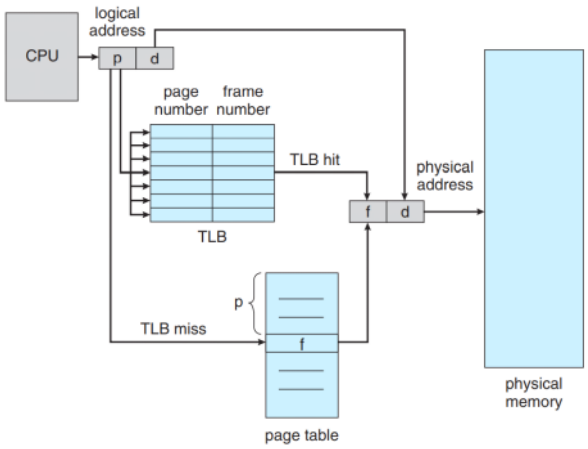
Ogni volta che un processo fa riferimento ad un indirizzo in memoria virtuale, l'indirizzo virtuale deve essere tradotto in un indirizzo fisico, attraverso (ovviamente) l'utilizzo della CPU.

Dove converrebbe quindi memorizzare la page table per avere una buona efficienza?

a. **Registri**:
   Molto veloce ma anche molto costoso e limitante.
b. **Memoria Principale**:
   Grande capacità ma relativamente lenta (ogni traduzione in memoria richiede un ulteriore accesso alla memoria).
c. Utilizziamo una via di mezzo:
   Translation Look-aside Buffer (**TLB**)

Prima di definire cos'è una TLB, rivediamo il concetto di **cache**: una cache è una **memoria piccola e veloce** che fa da **intermediario tra la CPU e la memoria principale**. In questa memoria vengono memorizzati gli indirizzi utilizzati più recentemente o più di frequente (a seconda della politica), così da ridurre il numero di accessi alla memoria.

Essenzialmente la TLB **è una cache fully-associative** che memorizza i **numeri delle pagine** (key, o tag) e i numeri dei frame (values). Quando un indirizzo logico viene generato dalla CPU, l'MMU prima controlla se il numero di pagina è presente nella TLB. Se il numero di pagina è presente abbiamo una TLB hit, il frame della pagina è quindi immediatamente accessibile.
Se il numero di pagine non è presente abbiamo una TLB miss, dovremo quindi cercare il frame nella page table. Dopo una miss, la coppia chiave/valore viene aggiunta alla TLB rimuovendo un'altra coppia, tale aggiornamento è gestito da una politica che dipende dall'implementazione. Ovviamente più la TLB è grande più la possibilità di una hit è alta, diminuendo il costo medio degli accessi alla memoria.



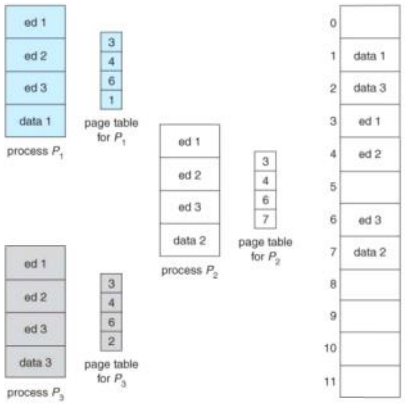
PROTEZIONE

La protezione della memoria viene mantenuta tramite l'utilizzo di **bit di protezione** associati con ciascun frame. Normalmente questi bit sono **mantenuti nella page table**.
Un bit può essere utilizzato per definire se una pagina sia di tipo read only o read/write. Un tentativo di scrittura da parte di un processo su una pagina read only genererà una trap nel sistema operativo (memory-protection-violation).
Può essere utilizzato un bit addizionale, detto **valid/invalid bit**, per specificare se la pagina associata è nello spazio di indirizzamento logico ed è una pagina legale (valida). Il sistema operativo setta questi bit per consentire/negare l'accesso alle pagine.

PAGINE CONDIVISE

Un grande vantaggio del paging è la facilità con cui si può condividere il codice, in quanto la memoria non necessita più di essere contigua. La condivisione può essere effettuata semplicemente duplicando le voci di pagina di diversi processi agli stessi frame (vale sia per il codice che per i dati).

Di fianco abbiamo 3 processi utente che stanno usando lo stesso editor *ed*. Solo una singola copia di *ed* è attualmente caricata in memoria.



3.7 STRUTTURA DI UNA PAGE TABLE

In questo paragrafo esploreremo alcune delle tecniche più comuni per strutturare la page table, tra cui lo hierarchical paging, le hashed page tables e le inverted page tables.

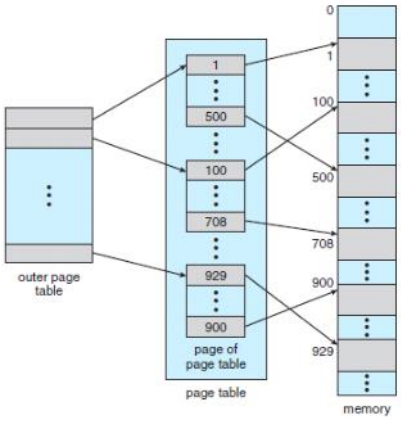
HIERARCHICAL PAGING

La maggior parte dei sistemi informatici moderni supporta un ampio spazio di indirizzi logici (da 2^{32} a 2^{64}). In un tale ambiente, la page table stessa diventa eccessivamente grande.

Ad esempio, si consideri un sistema con uno spazio di indirizzi logici a 32 bit. Se la dimensione della pagina in un tale sistema è di 4 KB (2^{12}), una page table può contenere fino a 1 milione di voci ($2^{32}/2^{12}$). Supponendo che ogni voce sia composta da 4 byte, ogni processo potrebbe richiedere fino a 4 MB di spazio di indirizzi fisici solo per la page table.

Banalmente, non vorremmo allocare la page table in modo contiguo nella memoria principale. Una semplice soluzione a questo problema è di dividere la page table in parti più piccole. Possiamo realizzare questa divisione in diversi modi. Un modo consiste nell'utilizzare il **two-level paging algorithm**, in cui viene paginata anche la page table stessa (come si può vedere in figura).

Ad esempio, si consideri nuovamente il sistema con uno spazio di indirizzi logici a 32 bit e una dimensione della pagina di 4 KB. Un indirizzo logico è suddiviso in un page number composto da 20 bit e un page offset composto da 12 bit.



Poiché impaginiamo la tabella delle pagine, il numero di pagina è ulteriormente suddiviso in un numero di pagina a 10 bit e un offset di pagina a 10 bit. Pertanto, un indirizzo logico è il seguente:

page number		page offset
p_1	p_2	d
10	10	12

Dove p1 è l'indice all'interno della outer page table e p2 è lo offset all'interno della page table interna. Poiché la conversione degli indirizzi funziona dalla tabella delle pagine esterne verso l'interno, questo schema è noto anche come forward-mapped page table. Per le architetture a 64 bit, le hierarchical page tables sono generalmente considerate inappropriate.

HASHED PAGE TABLES

Un approccio comune per la gestione di spazi di indirizzi superiori a 32 bit consiste nell'utilizzare una **hashed page table**. Con il valore **hash** che è il **numero di pagina virtuale**. Ogni voce nella tabella hash contiene una **linked list** di elementi che hanno l'hash nella stessa posizione (per gestire le collisioni).

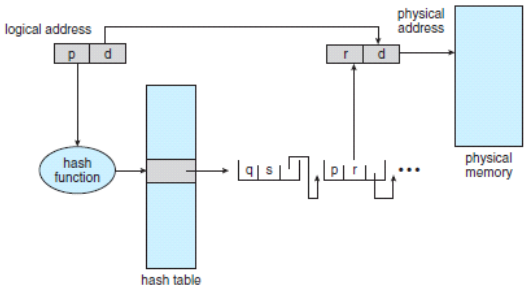
Ogni elemento è composto da tre campi:

- Il numero della pagina virtuale
- il valore della page frame mappata
- un puntatore all'elemento successivo nella linked list

L'algoritmo funziona come segue: il numero della pagina virtuale nell'indirizzo virtuale viene sottoposto a hash nella tabella hash. Il numero della pagina virtuale viene confrontato con il primo campo del primo elemento della linked list. Se c'è corrispondenza, il page frame (campo 2) viene utilizzato per formare l'indirizzo fisico. Se non c'è corrispondenza, vengono verificate le voci successive della linked list per trovare il valore della pagina virtuale corrispondente. Questo schema è visualizzato nell'immagine qui sotto

È stata proposta una variazione di questo schema utile per spazi di indirizzi a 64 bit. Questa variazione utilizza delle clustered page tables, che sono simili alle hashed page tables tranne per il fatto che ogni voce nella tabella hash fa riferimento a diverse pagine (ad esempio 16) anziché una singola pagina.

Quindi una singola page-table può mappare più physical-page. Le clustered page tables sono utili per spazi di indirizzi sparsi, in cui i riferimenti di memoria non sono contigui, ma sparsi in tutto lo spazio degli indirizzi.

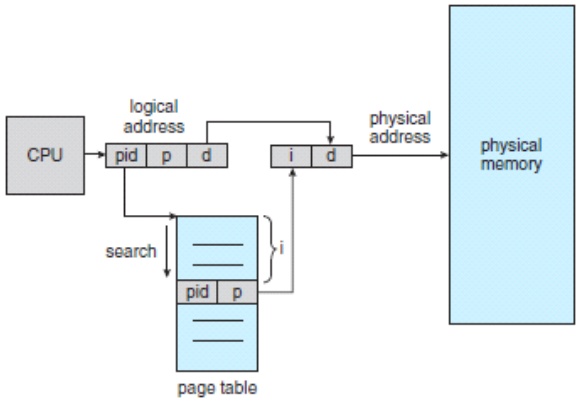


INVERTED PAGE TABLES

Di solito, ogni processo ha una tabella delle pagine associata. La tabella delle pagine ha una voce per ogni pagina utilizzata dal processo. Questa rappresentazione è naturale, poiché elabora le pagine di riferimento attraverso gli indirizzi virtuali delle pagine.

Il sistema operativo deve quindi tradurre questo riferimento in un indirizzo di memoria fisica. Poiché la tabella è ordinata per indirizzo virtuale, il sistema operativo è in grado di calcolare dove si trova nella tabella la voce dell'indirizzo fisico associato e di utilizzare direttamente tale valore.

Uno degli svantaggi di questo metodo è che ogni page table può essere composta da milioni di voci. Queste tabelle possono consumare grandi quantità di memoria fisica solo per tenere traccia di come viene utilizzata altra memoria fisica.



Per risolvere questo problema, possiamo utilizzare un'**inverted page table**.

Una inverted page table ha una voce di memoria per ogni pagina reale (o frame). Ogni voce è costituita dall'indirizzo virtuale della pagina memorizzata in quella posizione di memoria reale, con informazioni sul processo che possiede la pagina. Pertanto, nel sistema è presente solo una tabella di pagine e ha una sola voce per ogni pagina di memoria fisica.

In figura sono mostrate le operazioni di un inverted page table. Le inverted page table richiedono che in ciascuna voce della tabella delle pagine sia memorizzato un identificatore dello spazio degli indirizzi, poiché la tabella di solito contiene diversi spazi degli indirizzi diversi che mappano la memoria fisica.

Memorizzare questo identificatore garantisce che una logical page di un processo particolare sia mappata alla corrispondente physical page.

Per illustrare questo metodo, descriviamo una versione semplificata dell'inverted page table utilizzata in IBM RT. IBM è stata la prima grande azienda a utilizzare le inverted page table.

Per IBM RT, ogni indirizzo virtuale nel sistema è costituito da una tripla:

<process-id, page-number, offset>.

Ogni voce della inverted page table è una coppia <process-id, page-number>, dove il process-id assume il ruolo dell'identificatore dello spazio di indirizzi. Quando si verifica un riferimento alla memoria, parte dell'indirizzo virtuale, costituito da <process-id, page-number>, viene presentato al sottosistema della memoria.

Viene quindi cercata un inverted page table per una corrispondenza. Se viene trovata una corrispondenza, ad esempio alla voce i, viene generato l'indirizzo fisico <i, offset>. Se non viene trovata alcuna corrispondenza, è stato tentato un accesso illegale all'indirizzo.

Sebbene questo schema riduca la quantità di memoria necessaria per archiviare ogni page table, aumenta la quantità di tempo necessaria per cercare nella tabella quando si verifica un riferimento di pagina.

Poiché l'inverted page table è ordinata per indirizzo fisico, ma le ricerche si verificano sugli indirizzi virtuali, prima di trovare una corrispondenza potrebbe essere necessario eseguire la ricerca nell'intera tabella. Questa ricerca richiederebbe troppo tempo.

Per alleviare questo problema, usiamo una tabella hash, per limitare la ricerca a una o al massimo poche voci della page table. Questo metodo non può essere utilizzato con inverted page tables; poiché esiste una sola voce di pagina virtuale per ogni pagina fisica, una pagina fisica non può avere due (o più) indirizzi virtuali condivisi.

Una tecnica semplice per risolvere questo problema consiste nel consentire alla page table di contenere solo una mappatura di un indirizzo virtuale all'indirizzo fisico condiviso. Ciò significa che i riferimenti a indirizzi virtuali che non sono mappati provocano errori di pagina.

3.8 VIRTUAL MEMORY

Abbiamo discusso varie strategie di gestione della memoria utilizzate nei sistemi informatici. Tutte queste strategie hanno lo stesso obiettivo: consentire la multiprogrammazione mantenendo attivi simultaneamente molti processi in memoria. Tuttavia, queste strategie tendono a richiedere che un processo sia interamente caricato in memoria prima che possa essere eseguito.
La **virtual memory** è una tecnica che consente l'**esecuzione di processi che non sono completamente in memoria**. Uno dei principali vantaggi di questo schema è che **i programmi possono essere più grandi della memoria fisica**.

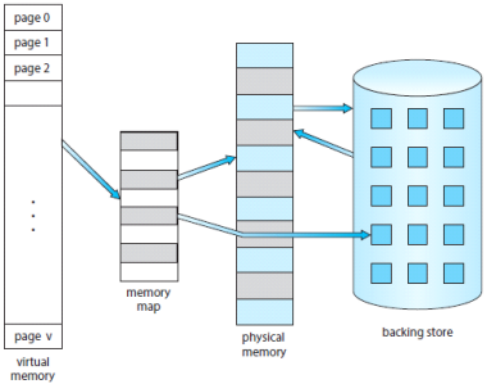
La memoria virtuale consente inoltre ai processi di condividere file e librerie e di implementare la memoria condivisa. Inoltre, fornisce un meccanismo efficiente per la creazione di processi. La memoria virtuale non è però facile da implementare e può ridurre le prestazioni se utilizzata inappropriatamente.
Il requisito che le istruzioni per essere eseguite debbano trovarsi in memoria fisica è comunque necessario, ma non è necessario aver caricato in memoria l'intero programma (nella maggior parte dei casi se ne utilizza solo una parte). Ad esempio, consideriamo quanto segue:

- I programmi hanno spesso codice per gestire condizioni insolite di errore. Poiché questi errori si verificano raramente, nella pratica questo codice non viene quasi mai eseguito.
- Agli array, agli elenchi e alle tabelle viene spesso allocata più memoria di quella di cui hanno effettivamente bisogno.
- Alcune opzioni e caratteristiche di un programma potrebbero essere utilizzate raramente.

Anche nei casi in cui è necessario l'intero programma, potrebbe non essere necessario tutto contemporaneamente. La possibilità di eseguire un programma che è solo parzialmente in memoria conferisce molti vantaggi:

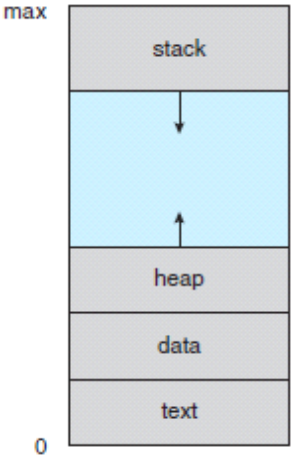
- Programmi non più vincolati alla quantità di memoria fisica (illusione di una memoria "illimitata").
- Esecuzione di più programmi in contemporanea, con conseguente aumento delle prestazioni.
- Necessario meno I/O per caricare o scambiare porzioni di programmi in memoria.

L'esecuzione di un programma che non è interamente in memoria va quindi a vantaggio sia del sistema che dei suoi utenti.



Lo spazio degli indirizzi virtuali di un processo si riferisce alla vista logica (o virtuale) di come un processo è archiviato in memoria.
In genere questa visione prevede che un processo inizi da un certo indirizzo logico ed esista nella memoria contigua. Ricordiamo, però, che la memoria fisica è organizzata in frame e che i frame assegnati a un processo potrebbero non essere contigui. Spetta quindi all'unità di gestione della memoria (MMU) mappare in memoria le pagine ai frame della memoria fisica.

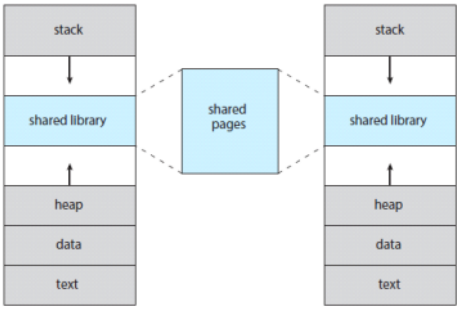
L'ampio spazio libero tra l'heap (che tramite allocazione dinamica cresce verso lo stack) e lo stack (che decresce verso l'heap tramite chiamate di funzioni consecutive) fa parte dello spazio degli indirizzi virtuali, ma richiede pagine fisiche solo quando heap o stack crescono.



Gli spazi di indirizzi virtuali che includono buchi sono noti come **spazi di indirizzi sparsi**. L'uso di uno spazio di indirizzi sparso è vantaggioso perché i buchi possono essere riempiti man mano che i segmenti dello stack o dell'heap crescono o se desideriamo collegare dinamicamente le librerie (o possibilmente altri oggetti condivisi) durante l'esecuzione del programma.

La memoria virtuale consente inoltre ai file e alla memoria di essere condivisi da due o più processi attraverso il page sharing, di conseguenza:

- Le librerie di sistema possono essere condivise da diversi processi tramite la mappatura dell'oggetto condiviso in uno spazio di indirizzi virtuale. Sebbene ogni processo consideri le librerie come parte del proprio spazio di indirizzamento virtuale, le pagine effettive in cui risiedono le librerie nella memoria fisica sono condivise da tutti i processi. In genere, una libreria viene mappata nello spazio di ciascun processo ad esso collegato in read-only.



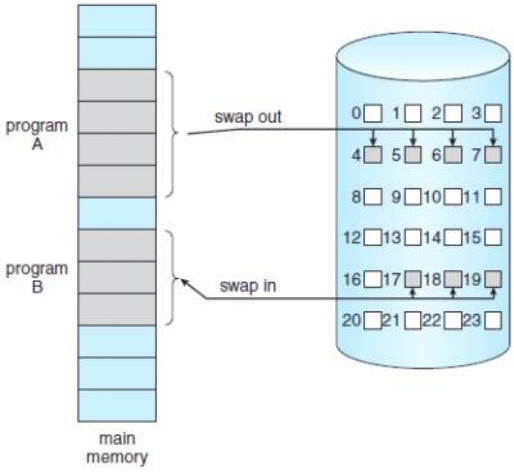
- La memoria virtuale consente a un processo di creare una regione di memoria che può condividere con un altro processo.
- Durante la creazione del processo con la chiamata di sistema `fork()`,le pagine possono essere condivise, questo velocizza la creazione del processo.

Tratteremo ora l'implementazione della virtual memory tramite **demand paging.**

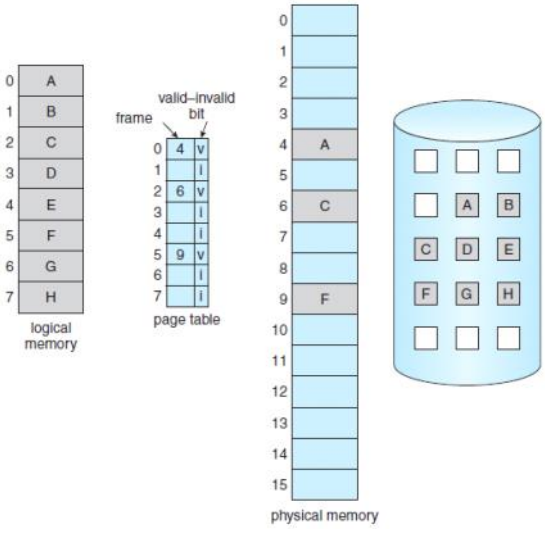
3.9 DEMAND PAGING

Si consideri come un programma eseguibile possa essere caricato dal disco alla memoria. Una strategia alternativa al caricamento dell'intero programma consiste nel caricare le pagine solo quando sono richieste. Questa tecnica è nota come **demand-paging** ed è comunemente utilizzata nei sistemi di memoria virtuale. Le pagine a cui non si accede mai non vengono quindi mai caricate nella memoria fisica.

Un sistema di demand-paging è simile a un sistema di paging con swapping in cui i processi risiedono nella memoria secondaria (solitamente su un disco).
Quando vogliamo eseguire un processo, lo trasferiamo in memoria. Piuttosto che scambiare l'intero processo in memoria però, usiamo un **lazy swapper**, che è un metodo di swapping dove solo le pagine che sono richieste vengono caricate in memoria. Nel contesto di un sistema di demand-paging, l'uso del termine "swapper" è tecnicamente errato. Uno swapper manipola interi processi, mentre un **pager** si occupa delle singole pagine di un processo. Quindi, in connessione con il demand paging, usiamo "pager" piuttosto che "swapper".



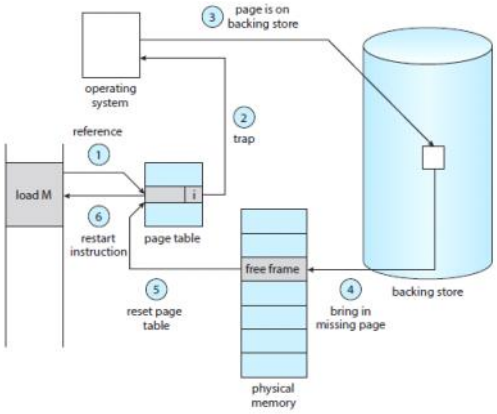
In un sistema di demand-paging, come abbiamo detto, durante l'esecuzione di un processo alcune pagine sono in memoria e altre nella memoria secondaria, a seconda delle pagine necessarie. Abbiamo però bisogno di un supporto hardware per distinguere tra le due: utilizziamo il valid-invalid bit. Quando questo bit è impostato su "**valid**", la pagina associata è **legale ed è in memoria**. Se il bit è impostato su "**invalid**", la pagina **non è valida** (ovvero non si trova nello spazio degli indirizzi logici del processo) oppure è **valida ma è attualmente sul disco** (si noti che contrassegnare una pagina come non valida non avrà alcun effetto se il processo non tenta mai di accedervi).



Ma cosa succede se il processo tenta di accedere a una pagina che non è stata portata in memoria? **L'accesso a una pagina contrassegnata come invalid provoca un page fault.**

L'hardware di paging, traducendo l'indirizzo attraverso la page table, noterà che il bit impostato è invalid, e causerà una trap al sistema operativo.

La procedura per gestire questa page fault è semplice:



1. Controlliamo una tabella interna (di solito conservata con il PCB) per questo processo per determinare se il riferimento è un accesso alla memoria valid o invalid.
2. Se il riferimento è invalid, interrompiamo il processo. Se è valid ma non abbiamo ancora inserito quella pagina, ora la inseriamo.
3. Troviamo un frame libero.
4. Scheduliamo un'operazione di archiviazione secondaria per leggere la pagina desiderata nel frame appena allocato.
5. Quando la lettura della memoria è completa, modifichiamo l'internal table conservata con il processo e la page table, per indicare che ora la pagina è in memoria.
6. Riavviamo l'istruzione che è stata interrotta dalla trap: il processo ora può accedere alla pagina.

Nel caso estremo, possiamo avviare l'esecuzione di un processo senza pagine in memoria. Quando il sistema operativo imposta il puntatore alla prima istruzione del processo, che si trova su una pagina che non è inserita in memoria, il processo genera immediatamente un errore. Dopo che questa pagina è stata portata in memoria, il processo continua l'esecuzione, generando errori se necessario finché ogni pagina di cui ha bisogno è in memoria. A quel punto, può essere eseguito senza più errori.

PERFORMANCE DEL DEMAND PAGING

Il demand paging può influire in modo significativo sulle prestazioni di un sistema informatico. Calcoliamo il tempo di accesso effettivo per una demand-paged memory. Supponiamo che il **tempo di accesso alla memoria**, indicato con $\boldsymbol{ma}$, sia di $10\ nanosecondi$.

Finché non abbiamo page faults, il tempo di accesso effettivo è uguale al tempo di accesso alla memoria. Se invece si verifica un page fault, dobbiamo prima leggere la relativa pagina dalla memoria secondaria e quindi accedere alla word desiderata.
Sia $\boldsymbol{p}$ **la probabilità di un errore di pagina** ($0 \leq p \leq 1$). Ci aspetteremmo che p sia vicino a zero, cioè ci aspetteremmo di avere solo pochi page fault. Il tempo di accesso effettivo è allora:

$$\boldsymbol{tempo\ di\ accesso\ effettivo} = (\mathbf{1} - \boldsymbol{p}) \times \boldsymbol{ma} + \boldsymbol{p} \times \boldsymbol{page\ fault\ time}.$$

Per calcolare il tempo di accesso effettivo, dobbiamo sapere quanto tempo è necessario per correggere un page fault. Un page fault provoca la seguente catena di eventi:

1. Genera una trap al sistema operativo.
2. Salva i registri utente e lo stato del processo.
3. Determina che l'interruzione era una page fault.
4. Verifica che il riferimento alla pagina sia legale e si determina la posizione della pagina sul disco.
5. Invia una lettura dal disco a un frame libero:
    a. Attende in coda per questo dispositivo finché la richiesta di lettura non viene soddisfatta.
    b. Attende la ricerca del dispositivo e/o il tempo di latenza.
    c. Inizia il trasferimento della pagina in un frame libero.
6. Durante l'attesa, si può assegnare la CPU a un altro utente.
7. Riceve un interrupt dal sottosistema di I/O del disco (I/O completato).
8. Salva i registri e lo stato del processo per l'altro utente (se viene eseguito il passaggio 6).
9. Determina che l'interruzione proveniva dal disco.
10. Corregge la page table e altre tabelle per mostrare che la pagina desiderata è ora in memoria.
11. Attende che la CPU venga nuovamente assegnata a questo processo.

12. Ripristina i registri utente, lo stato del processo e la nuova tabella delle pagine, e riprende l'istruzione interrotta.

Tutti questi passaggi non sono sempre necessari.

ESEMPIO DI CALCOLO DELLE PERFORMANCE

Dati:

$$t_{MA} = tempo\ di\ accesso\ alla\ memoria\ fisica$$

$$t_{FAULT} = \ tempo\ per\ gestire\ una\ page\ fault$$

$$p \in [0,1] = probabilità\ della\ page\ fault$$

$$t_{ACCESS} = tempo\ effettivo\ per\ ogni\ riferimento\ in\ memoria$$

$$\boldsymbol{t_{ACCESS} = (1-p) * tMA + p * tFAULT}$$

assumiamo che :

$$t_{MA} = 100\ nsec\ e\ t_{FAULT} = 20\ msec = 20,000,000\ nsec$$

$$t_{ACCESS} = (1-p) * 100 + p * 20,000,000$$

come si può notare, tutto dipende da p. Se per esempio 1 accesso ogni 1000 provoca una page fault, il tempo di accesso aumenta solo di $100\ nsec$ fino a $\sim 20.1\ microsec$.

E se volessimo che l'access time sia al massimo il 10% più lento dell'accesso alla memoria di base? Dobbiamo risolvere per p la seguente equazione:

$$1.1 * 100 = (1-p) * 100 + p * 20,000,000 =$$

$$1.1 * 100 = 100 - 100p + 20,000,000p =$$

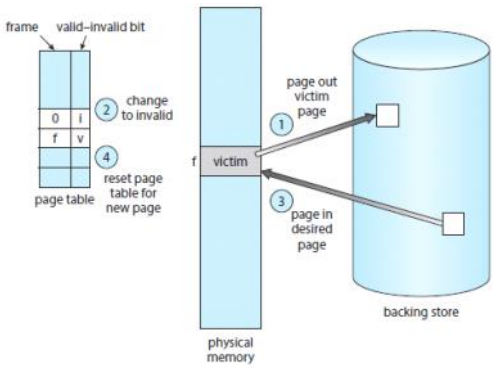
$$19,999,900p = 110 - 100 =$$

$$p = \frac{10}{19,999,900} = \frac{1}{1,999,990} \approx 0{,}0000005 = 5 * 10^{-7}$$

Per raggiungere questo obiettivo, possiamo tollerare al massimo 1 page fault ogni circa 2 milioni di accessi!

PAGE REPLACEMENT

Il page replacement adotta il seguente approccio: se nessun frame è libero, ne troviamo uno che non è attualmente in uso e lo liberiamo. Possiamo liberare un frame scrivendone il contenuto per fare lo swap e aggiornando la page table (e tutte le altre tabelle) per indicare che la pagina non è più in memoria. Possiamo ora caricare nel frame libero la pagina per la quale il processo è andato in errore.



Modifichiamo la routine di correzione del page-fault per includere la sostituzione della pagina:

1. Trova la posizione della pagina desiderata nella memoria secondaria.
2. Trova un frame libero:
   a. Se c'è un frame libero, usalo.
   b. Se non è presente alcun frame libero, utilizzare un algoritmo di sostituzione della pagina per selezionare un **victim frame**.
   c. Se necessario scrivi il victim frame (il frame rimosso dalla memoria principale) nella memoria secondaria, modificando di conseguenza le page e le frame table.
3. Leggi la pagina desiderata nel frame appena liberato, modificando le page e le frame table.
4. Continua il processo dal punto in cui si è verificato il page fault.

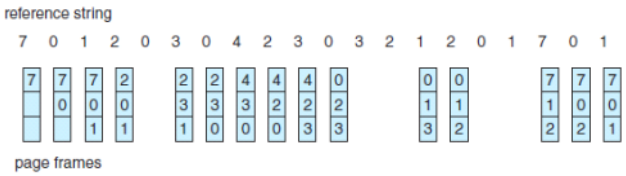
Si noti che, se non ci sono frame liberi, sono **necessari due trasferimenti** di pagina (uno per il **page-out** e uno per il **page-in**). **Questa situazione raddoppia il tempo di correzione di un page-fault** e aumenta di conseguenza il tempo di accesso effettivo. Possiamo ridurre questo sovraccarico utilizzando un **modify bit** (o **dirty bit**). Quando viene utilizzato questo schema, a ogni pagina o frame è associato un modify bit.

Il modify bit viene settato dall'hardware ogni volta che viene modificato un qualsiasi byte nella pagina, indicando che la pagina è stata modificata. Quando selezioniamo una pagina per la sostituzione, esaminiamo il suo modify bit. Se il bit è asserito, sappiamo che la pagina è stata modificata da quando è stata letta dalla memoria secondaria. In questo caso, dobbiamo scrivere la pagina nella memoria.

Se invece il modify bit non è asserito, la pagina non è stata modificata da quando è stata letta in memoria. In questo caso non è necessario scrivere la pagina in memoria. Questa tecnica è usata anche nelle pagine read-only, tali pagine non possono essere modificate, possono quindi essere scartate quando lo si desidera. Questo schema può ridurre il tempo necessario per correggere un page fault, poiché riduce della metà il tempo di I/O se la pagina non è stata modificata.

3.10 FIFO PAGE REPLACEMENT

La politica di page replacement più semplice è la FIFO (first-in-first-out): quando una pagina deve essere sostituita, viene scelta la pagina più vecchia.
Un algoritmo di sostituzione FIFO può associare ad ogni pagina l'orario in cui quella pagina è stata portata in memoria. Tale pratica non è strettamente necessaria: possiamo creare una coda. Rimuoviamo la prima pagina della coda (la più vecchia) e inseriamo la nuova pagina in coda.



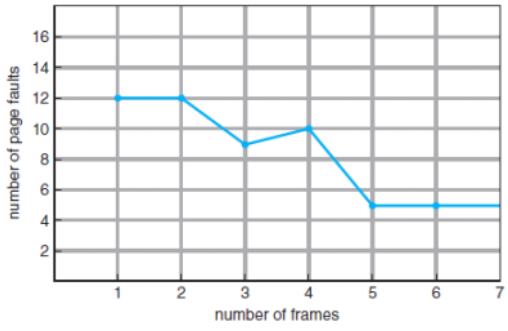
Per la nostra reference string, i nostri tre frame sono inizialmente vuoti. I primi tre riferimenti (7, 0, 1) causano page fault e vengono portati in questi frame vuoti. Il riferimento successivo (2) sostituisce la pagina 7: è stata inserita per prima quindi è la più vecchia. Poiché 0 è il riferimento successivo ed è già stato inserito in memoria, non abbiamo page fault. Il riferimento 3 sostituisce la pagina 0, essendo ora la pagina più vecchia. Questo processo continua per il resto dei riferimenti. Nell'immagine sono mostrati tutti i passaggi. In totale ci sono 15 page fault.

Gli algoritmi FIFO sono facili da capire e programmare. Tuttavia, le loro prestazioni non sono sempre buone. Da un lato, la pagina sostituita potrebbe essere un modulo di inizializzazione utilizzato molto tempo fa e non più necessario. D'altra parte, potrebbe contenere una variabile molto utilizzata che è stata inizializzata in anticipo ed è in uso costante. Si noti che, anche se per la sostituzione selezioniamo una pagina che è in uso attivo, tutto funziona ancora correttamente. Dopo aver sostituito una pagina attiva con una nuova, si verifica un errore quasi immediatamente per recuperare la pagina attiva. Per riportare in memoria la pagina attiva deve essere sostituita qualche altra pagina. Pertanto, una scelta di sostituzione errata aumenta il tasso di page faults e rallenta l'esecuzione del processo.

Per illustrare i possibili problemi con un algoritmo di sostituzione della pagina FIFO, consideriamo la seguente stringa di riferimento:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

La Figura mostra la curva dei page fault per questa stringa di riferimento rispetto al numero di frame disponibili.
Si noti che il numero di errori per quattro frame (dieci) è maggiore del numero di errori per tre frame (nove)!



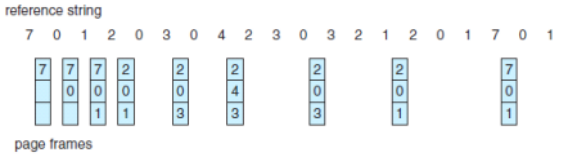
Questo risultato inaspettato è noto come **anomalia di Belady**: per alcuni algoritmi di page replacement, il tasso di page faults può aumentare all'aumentare del numero di frame allocati.
Ci aspetteremmo che dare più memoria a un processo ne migliori le prestazioni. I ricercatori hanno notato che questa ipotesi non era sempre vera. Di conseguenza è stata scoperta l'anomalia di Belady.

3.11 OPTIMAL PAGE REPLACEMENT

Un risultato della scoperta dell'anomalia di Belady è stata la ricerca di un algoritmo page-replacement ottimale, l'algoritmo che ha il tasso di errore di pagina più basso di tutti gli algoritmi e non soffrirà mai dell'anomalia di Belady. Tale algoritmo è stato chiamato **OPT** o **MIN**. Ed è semplicemente questo:

Sostituisci la pagina che non verrà utilizzata per il periodo di tempo più lungo.

Questo algoritmo di page replacement garantisce il tasso di page fault più basso possibile per un numero fisso di frame.



Nella nostra reference string, l'optimal page replacement produrrebbe nove page faults, come mostrato in figura. I primi tre riferimenti causano fault che riempiono i tre frame vuoti. Il riferimento alla pagina 2 sostituisce la pagina 7, che non sarà utilizzata fino al 18° riferimento, mentre la pagina 0 verrà utilizzata al 5° e la pagina 1 al 14°. Il riferimento alla pagina 3 sostituisce la pagina 1, che sarà l'ultima delle tre pagine in memoria a cui si farà nuovamente riferimento.

Con solo 9 page fault, l'optimal replacement si rivela nettamente migliore di un algoritmo FIFO, che risolve la stessa situazione con 15 page fault. (Se ignoriamo i primi tre fault, che tutti gli algoritmi devono subire, l'optimal replacement è due volte migliore della sostituzione FIFO).
Nessun algoritmo può elaborare questa stringa di riferimento in tre frame con meno di nove errori.

L'algoritmo di optimal page replacement è però difficile da implementare, poiché richiede una conoscenza futura della stringa di riferimento. L'optimal page replacement viene difatti utilizzato principalmente per studi comparativi.

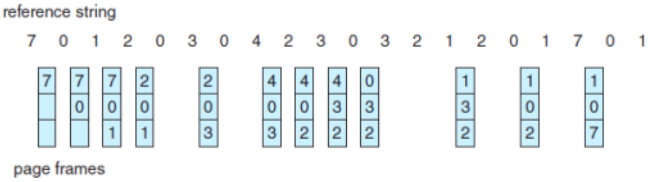
3.12 LRU PAGE REPLACEMENT

Se l'optimal algorithm non è implementabile, è possibile farne un'approssimazione. La distinzione fondamentale tra gli algoritmi FIFO e OPT è che l'algoritmo FIFO utilizza il momento in cui una pagina è stata portata in memoria, mentre l'algoritmo OPT utilizza il tempo in cui una pagina deve essere utilizzata.

Se usiamo il passato recente come approssimazione del futuro prossimo, allora possiamo s**ostituire la pagina che non è stata utilizzata per il periodo di tempo più lungo**. Questo approccio è chiamato **least recently used (LRU)**. L' LRU associa a ciascuna pagina l'orario dell'ultimo utilizzo. Quando una pagina deve essere sostituita, l'algoritmo LRU sceglie la pagina che non è stata utilizzata per il periodo di tempo più lungo.

Possiamo pensare a questa strategia come all'OPT, ma guardando indietro nel tempo, piuttosto che in avanti.

In figura è mostrato il risultato dell'applicazione della sostituzione LRU alla nostra reference string. L'algoritmo LRU produce dodici errori.



Si noti che i primi cinque errori sono gli stessi di quelli dell' OPT. Quando si verifica il riferimento alla pagina 4, tuttavia, la sostituzione LRU vede che, dei tre frame in memoria, la pagina 2 è quella utilizzata meno di recente. Pertanto, l'algoritmo LRU sostituisce la pagina 2, non sapendo che la pagina 2 sta per essere utilizzata. Quando poi restituisce un page fault per la pagina 2, l'algoritmo LRU sostituisce la pagina 3, poiché, delle tre pagine in memoria è la meno utilizzata di recente. Nonostante questi problemi, con dodici errori, la sostituzione di LRU è molto meglio della sostituzione FIFO con quindici. La politica LRU viene spesso utilizzata come page replacement algorithm ed è considerata buona.

Il problema principale è come implementare la sostituzione LRU. Un page-replacement algorithm LRU può richiedere una notevole assistenza hardware. Il problema è determinare un ordine per i frame definito dall'ora dell'ultimo utilizzo.

Vediamo due implementazioni:

- **Counters**

  Nel caso più semplice, associamo ad ogni entry della tabella delle pagine un campo time-of-use e aggiungiamo alla CPU un clock o logical timer. Il clock viene incrementato per ogni riferimento di memoria.

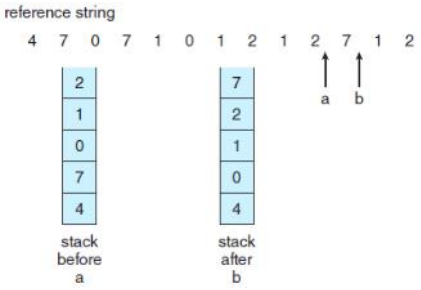
  Ogni volta che viene fatto un riferimento a una pagina, il contenuto del registro del clock viene copiato nel campo del time of use di quella pagina nella page table. In questo modo abbiamo sempre il “tempo” dell'ultimo riferimento ad ogni pagina.

  Gli orari devono essere mantenuti anche quando le page table vengono modificate.

- **Stack**

  Un altro approccio all'implementazione dell' LRU consiste nel mantenere uno stack di page number. Ogni volta che si fa riferimento a una pagina, questa viene rimossa dallo stack e messa in cima.

  In questo modo, la pagina utilizzata più di recente è sempre in cima allo stack e la pagina utilizzata meno di recente è sempre in fondo.



OPT e LRU appartengono a una classe di algoritmi di page replacement, chiamati **algoritmi stack**, che non possono mai mostrare l'anomalia di Belady. Un algoritmo stack è un algoritmo per il quale si può dimostrare che l'insieme di pagine in memoria per n frame è sempre un sottoinsieme dell'insieme di pagine che ci sarebbero in memoria con $n + 1$ frame.

3.13 LRU-APPROXIMATION PAGE REPLACEMENT

Non molti computer forniscono supporto hardware sufficiente per implementare un algoritmo LRU. In effetti, alcuni sistemi non forniscono alcun supporto hardware e devono essere utilizzati altri algoritmi di sostituzione della pagina. Molti sistemi forniscono invece un aiuto sotto forma di un **reference bit**, il quale è settato dall'hardware ogni volta che si fa riferimento a quella pagina (una lettura o una scrittura su qualsiasi byte nella pagina).

Inizialmente, il sistema operativo azzera tutti i bit. Durante l'esecuzione di un processo, il bit associato a ciascuna pagina a cui viene fatto riferimento viene impostato (a 1) dall' hardware. Dopo un po' di tempo, possiamo determinare quali pagine sono state utilizzate e quali non sono state utilizzate esaminando i reference bit, anche se non conosciamo l'ordine di utilizzo. Questa informazione è la base per molti page-replacement algorithm che approssimano la sostituzione LRU.

Possiamo ottenere ulteriori informazioni sull'ordinamento registrando i reference bit a intervalli regolari: per ogni pagina, possiamo conservare in memoria 8 bit (un byte). A intervalli regolari (ad esempio, ogni 100 millisecondi), un timer interrupt trasferisce il controllo al sistema operativo, il quale sposta il reference bit di ogni pagina nel bit di ordine superiore del suo byte, spostando gli altri bit a destra di 1 e scartando il bit di ordine inferiore. Questi shift registers a 8 bit contengono la cronologia dell'utilizzo della pagina per gli ultimi otto periodi di tempo.

> Se ad esempio, lo shift register contiene 00000000, la pagina non è stata utilizzata per otto periodi di tempo. Una pagina utilizzata almeno una volta in ogni periodo ha un shift register value di 11111111. Una pagina con un register value cronologico di 11000100 è stata utilizzata più recentemente di una con un valore di 01110111.

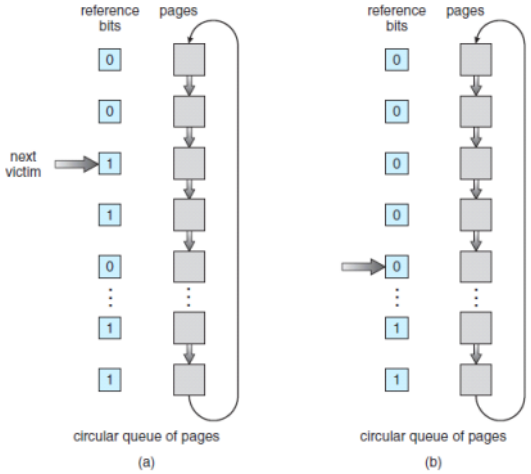
Se interpretiamo questi byte come numeri interi senza segno, la pagina con il numero più basso è l'LRU page e può essere sostituita. Tuttavia, non è garantito che i numeri siano univoci. Possiamo sostituire tutte le pagine con il valore più piccolo o utilizzare il metodo FIFO per scegliere tra di esse.
Il numero di history-bits inclusi nello shift register può variare a seconda dell'hardware disponibile, ed è scelto per rendere l'aggiornamento il più veloce possibile.

Il numero di bit può anche essere ridotto a zero, lasciando solo il bit di riferimento stesso. Questo algoritmo è chiamato **second chance page-replacement algorithm**.

SECOND-CHANCE ALGORITHM

L'algoritmo di base del **second-chance replacement** (o clock algorithm) è un algoritmo FIFO. Quando una pagina è stata selezionata ne ispezioniamo però il bit di riferimento. **Se il valore è 0, procediamo a sostituire questa pagina; ma se il bit di riferimento è impostato a 1, diamo alla pagina una seconda possibilità e passiamo alla selezione della successiva FIFO page**.
Quando una pagina ottiene una seconda possibilità, il suo bit di riferimento viene resettato e il suo arrival time viene reimpostato al current time. Pertanto, una pagina a cui viene data una seconda possibilità non verrà sostituita fino a quando tutte le altre pagine non saranno state sostituite (o date una seconda possibilità). Inoltre, se una pagina viene utilizzata abbastanza spesso da mantenere impostato il suo bit di riferimento, non verrà mai sostituita.

Un modo per implementare l'algoritmo second-chance è usare una coda circolare. Un puntatore indica quale pagina deve essere sostituita successivamente. Quando è necessario un frame, il puntatore avanza finché non trova una pagina con un reference bit a 0. Man mano che avanza, cancella i reference bit. Una volta trovata una victim page, la pagina viene sostituita e la nuova pagina viene inserita nella coda circolare in quella posizione.
Se tutti i bit sono settati, il puntatore scorre l'intera coda, dando a ciascuna pagina una seconda possibilità. Prima di selezionare la pagina successiva per la sostituzione, cancella tutti i reference bit. In tal caso, il second-chance degenera in un FIFO.



ENHANCED SECOND-CHANCE ALGORITHM

Possiamo migliorare il second-chance considerando reference e modify bit come una coppia ordinata. Con questi due bit, abbiamo quattro possibili classi:

- (0, 0) né usata né modificata di recente, è pagina migliore da sostituire
- (0, 1) non usato di recente ma modificato, non altrettanto buono, perché la pagina dovrà essere scritta prima di essere sostituita
- (1, 0) usata di recente ma pulita, probabilmente verrà riutilizzata presto
- (1, 1) utilizzata e modificata di recente, probabilmente verrà riutilizzata presto e la pagina dovrà essere scritta nella memoria secondaria prima di poter essere sostituita

Ogni pagina è in una di queste quattro classi. Quando è richiesto un page-replacement, usiamo lo stesso schema del second-chance ma, invece di esaminare se una pagina ha il bit di riferimento asserito, esaminiamo la classe a cui appartiene quella pagina. Sostituiamo la prima pagina incontrata nella classe non vuota più bassa. Si noti che prima di trovare una pagina da sostituire potremmo dover scansionare la coda circolare più volte. La differenza principale tra questo algoritmo e l'algoritmo di clock è che qui diamo la preferenza a quelle pagine che sono state modificate così da ridurre il numero di I/O richiesti.

3.14 TRASHING

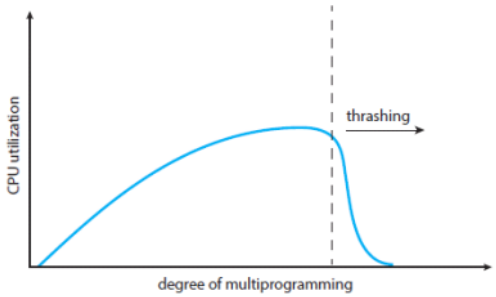
Si consideri cosa accade se un processo non dispone di "abbastanza" frame, ovvero non dispone del numero minimo di frame necessari per supportare le pagine nel working set. Il processo andrà rapidamente in page fault. A questo punto, deve sostituire qualche pagina. Tuttavia, poiché tutte le sue pagine sono in uso attivo, deve sostituire una pagina che sarà nuovamente necessaria dopo poco tempo. Di conseguenza, va continuamente in fault sostituendo pagine che dovrà poi riportare immediatamente. Questa alta attività di paging è chiamata **thrashing**. Un processo è in thrashing se **impiega più tempo a eseguire il paging che l'esecuzione**, ciò provoca ovviamente gravi problemi di prestazioni.

CAUSE DEL TRASHING

Si consideri lo scenario seguente: Il sistema operativo monitora l'utilizzo della CPU. Se l'utilizzo della CPU è troppo basso, aumentiamo il grado di multiprogrammazione introducendo un nuovo processo nel sistema, e utilizziamo un algoritmo globale di sostituzione della pagina che sostituisce le pagine indipendentemente dal processo a cui appartengono. Supponiamo ora che un processo entri in una nuova fase della sua esecuzione e necessiti di più frame. Inizia a generare fault e a sottrarre frame ad altri processi. Per fare lo swap in e out della pagina, questi processi di fault devono utilizzare il dispositivo di paging. Mentre si mettono in coda per il paging device, la coda dei ready si svuota. Man mano che i processi attendono il dispositivo di paging, l'utilizzo della CPU diminuisce. Lo scheduler della CPU vede la diminuzione dell'utilizzo della CPU e di conseguenza aumenta il grado di multiprogrammazione. Il nuovo processo tenta di avviarsi prelevando frame dai processi in esecuzione, causando più page faults e una coda più lunga per il dispositivo di paging. Di conseguenza, l'utilizzo della CPU diminuisce ulteriormente e lo scheduler della CPU cerca di aumentare ulteriormente il grado di multiprogrammazione.

Si è verificato un thrashing e il throughput del sistema è crollato. Il tasso di page fault aumenta enormemente e, di conseguenza, aumenta anche il tempo effettivo di accesso alla memoria. Non viene svolto alcun lavoro, in quanto i processi trascorrono tutto il loro tempo ad eseguire paging.

Questo fenomeno è illustrato nella Figura di fianco, in cui l'utilizzo della CPU è rappresentato graficamente rispetto al grado di multiprogrammazione. Con l'aumentare del grado di multiprogrammazione, aumenta anche l'utilizzo della CPU, fino a raggiungere il massimo.
Se il grado di multiprogrammazione aumenta ulteriormente, si verifica il thrashing e l'utilizzo della CPU diminuisce bruscamente. Per fermare il thrashing e aumentare l'utilizzo della CPU, dobbiamo diminuire il grado di multiprogrammazione.



Possiamo limitare gli effetti del thrashing utilizzando un **local replacement algorithm** (o priority replacement algorithm), il quale richiede che ciascun processo scelga solo dal proprio set di frame allocati.
Se un processo inizia il thrashing, non può rubare frame da un altro processo e causare il thrash anche di quest'ultimo.
Il problema non è, tuttavia, del tutto risolto: se i processi sono in thrashing, saranno in coda per il dispositivo di paging per la maggior parte del tempo. A causa della coda più lunga per il paging device, il tempo medio di servizio per un page faults aumenterà. Quindi, il tempo effettivo di accesso aumenterà anche per un processo che non è in thrashing. Per prevenire il thrashing, dobbiamo fornire a un processo tutti i frame di cui ha bisogno.
Una strategia è quindi quella di osservare quanti frame sta effettivamente utilizzando un processo. Questo approccio definisce il **modello di località** dell'esecuzione del processo. Una località è un gruppo di pagine utilizzate insieme attivamente. Un programma in esecuzione è generalmente composto da località diverse, che possono sovrapporsi. Il modello di località afferma che, mentre un processo viene eseguito, si sposta da una località all'altra.

> Ad esempio, quando viene chiamata una funzione, essa definisce una nuova località. In questa località, vengono fatti riferimenti di memoria alle istruzioni della chiamata di funzione, alle sue variabili locali e a un sottoinsieme delle variabili globali. Quando usciamo dalla funzione, il processo lascia questa località, poiché le variabili locali e le istruzioni della funzione non sono più in uso attivo.

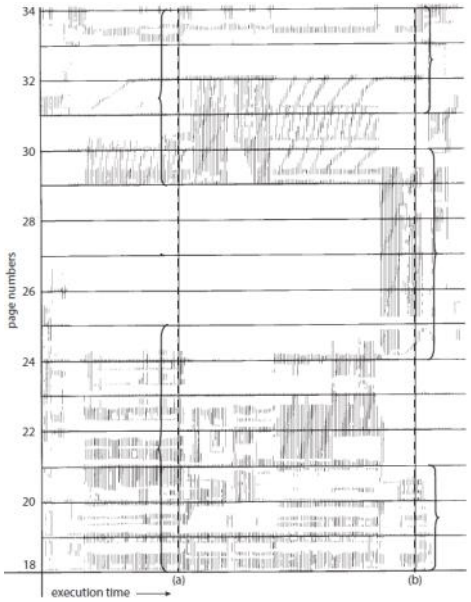
La Figura illustra il concetto di località e come la località di un processo cambia nel tempo.
Al tempo (a), la località è l'insieme delle pagine

$\{18, 19, 20, 21, 22, 23, 24, 29, 30, 33\}$.

Al tempo (b), la località cambia in

$\{18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33\}$.

Notare la sovrapposizione, poiché alcune pagine (es. $18, 19, 20$) fanno parte di entrambe le località.



Supponiamo di assegnare abbastanza frame a un processo per adattarsi alla sua attuale località. Andrà in fault per le pagine nella sua località finché tutte queste pagine non saranno in memoria; poi, non andrà più in faults fino a quando non cambierà località.

Se non allochiamo frame sufficienti per adattarsi alle dimensioni della località corrente, il processo andrà in tilt, poiché non può tenere in memoria tutte le pagine che sta utilizzando attivamente.

FORMULE

Dati:

$$m = numero\ di\ frame\ disponibili$$

$$n = numero\ di\ processi$$

$$S_i = dimensione\ dell'i-esimo\ processo$$

$$S = \sum_{i=1}^{n} S_i = dimensione\ totale\ di\ tutti\ i\ processi$$

$$\boldsymbol{Equal\ Allocation/Replacement} = \frac{\boldsymbol{m}}{\boldsymbol{n}}$$

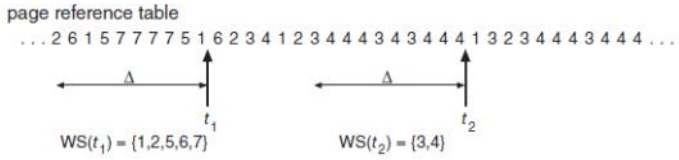
$$\boldsymbol{Proportional\ Allocation/Replacement} = \frac{\boldsymbol{m} * \boldsymbol{S_i}}{\boldsymbol{S}}$$

Le variazioni sulla proportional allocation potrebbero considerare la priorità del processo piuttosto che solo la loro dimensione. Poiché le allocazioni fluttuano nel tempo, anche m (i processi devono essere swappati o non possono essere avviati se non ci sono abbastanza frame).

WORKING-SET MODEL

Il modello del working set si basa sul presupposto della località. Questo modello utilizza un parametro Δ, per definire la working-set window. L'idea è di esaminare i riferimenti di pagina nell'unità di tempo Δ più recenti. l'insieme di pagine nei riferimenti Δ di pagina più recenti è il working set.

Se una pagina è in uso attivo, sarà nel working set. Se non viene più utilizzata, dopo il suo ultimo riferimento, verrà eliminata dall'unità di tempo Δ del working set.



Pertanto, il working set è un'approssimazione della località del programma.

> Ad esempio, data la sequenza di riferimenti di memoria mostrata nella Figura, se Δ = 10, allora il working set all'istante t1 è *{1, 2, 5, 6, 7}*. Al tempo t2, il working set è cambiato in *{3, 4}*.

La precisione del working set dipende dalla selezione di Δ. Se Δ è troppo piccolo, non comprenderà l'intera località; se Δ è troppo grande, può sovrapporsi a più località. Per assurdo, se Δ è infinito, il working set è l'insieme delle pagine toccate durante l'esecuzione del processo. La proprietà più importante del working set, quindi, è la sua dimensione. La dimensione del working set WSS_i, per ogni processo nel sistema, può essere considerata come

$$D = \sum WSS_i$$

dove D è la domanda totale di frame. Ogni processo utilizza attivamente le pagine nel proprio working set. Pertanto, il **processo i necessita di WSS_i frame**. Se la domanda totale è maggiore del numero totale di frame disponibili ($D > m$), si verificherà il thrashing, perché alcuni processi non avranno abbastanza frame. Una volta selezionato Δ, l'uso del modello del working set è semplice. Il sistema operativo monitora il working set di ciascun processo e alloca a quel working set un numero sufficiente di frame per fornirgli la dimensione del working set.

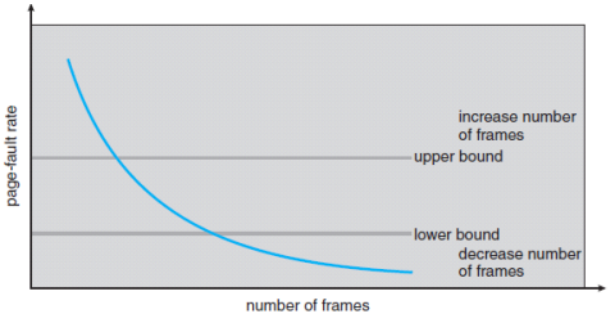
Se ci sono abbastanza frame aggiuntivi, è possibile avviare un altro processo. Se la somma delle dimensioni del working set aumenta, superando il numero totale di frame disponibili, il sistema operativo seleziona un processo da sospendere. Le pagine del processo vengono written out (swappate) e i suoi frame vengono riallocati ad altri processi. Il processo sospeso può essere riavviato successivamente. Questa strategia del working set impedisce il thrashing mantenendo il grado di multiprogrammazione il più alto possibile, pertanto ottimizza l'utilizzo della CPU. La difficoltà di questo modello è tenere traccia del working set, in quanto la finestra del working set è mobile. Una pagina è nel working set se vi si fa riferimento in qualsiasi punto della finestra del working set. Possiamo approssimare il modello del working set con un interrupt timer a intervallo fisso e un bit di riferimento. Ad esempio, supponiamo che Δ sia uguale a 10.000 riferimenti e che possiamo causare un'interruzione del timer ogni 5.000 riferimenti.

Quando riceviamo un interrupt del timer, per ogni pagina, copiamo e cancelliamo i valori dei bit di riferimento. Pertanto, se si verifica un page fault, possiamo esaminare il bit di riferimento corrente e i due bit in memoria per determinare se una pagina è stata utilizzata negli ultimi 10.000-15.000 riferimenti.
Se è stata utilizzata, almeno uno di questi bit sarà attivo. Se non è stata utilizzata, questi bit saranno disattivati. Le pagine con almeno un bit attivo saranno considerate nel working set.

FREQUENZA DI PAGE-FAULT

Il modello del working set ha successo e la conoscenza del working set può essere utile per la preparazione del paging, ma sembra un modo goffo per controllare il thrashing. Una strategia che utilizza la page-fault frequency (PFF) adotta un approccio più diretto.

Il problema è come prevenire il thrashing. Il thrashing ha un alto tasso di page fault. Pertanto, vogliamo controllare il tasso di page fault. Quando è troppo alto, sappiamo che il processo ha bisogno di più frame.
Al contrario, se il tasso di page fault è troppo basso, il processo potrebbe avere troppi frame. Possiamo quindi stabilire limiti superiori e inferiori al tasso di page fault desiderato.



Se il tasso di page fault supera il limite superiore, assegniamo un altro frame al processo. Se il tasso di page fault scende al di sotto del limite inferiore, rimuoviamo un frame dal processo. Pertanto, per prevenire il thrashing, possiamo misurare e controllare direttamente il tasso di page fault. Se il tasso di page fault aumenta e non sono disponibili frame liberi, dobbiamo selezionare un processo e swapparlo nel backing store. I frame liberati vengono quindi distribuiti ai processi che hanno un'elevata percentuale di page fault.

3.15 ALLOCATING KERNEL MEMORY

Quando un processo utente in esecuzione richiede memoria aggiuntiva, le pagine vengono allocate nei frame presenti nell'elenco di frame liberi gestito dal kernel. Questo elenco viene generalmente popolato utilizzando un algoritmo di page replacement come quelli discussi in precedenza, e molto probabilmente contiene pagine libere sparse nella memoria fisica.

Ricordiamo che se un processo utente richiede un singolo byte di memoria, accadrà una frammentazione interna, poiché al processo verrà concesso un intero page frame.
La memoria del kernel viene spesso allocata da un pool di memoria libera, che è diverso dall'elenco utilizzato per soddisfare i normali processi in modalità utente.

Ci sono due ragioni principali per questo:

1. Il kernel richiede memoria per strutture dati di varie dimensioni, alcune delle quali hanno dimensioni inferiori a una pagina. Di conseguenza, il kernel deve utilizzare la memoria in modo conservativo e tentare di ridurre al minimo gli sprechi dovuti alla frammentazione. Ciò è particolarmente importante perché molti sistemi operativi non sottopongono al sistema di paging il codice o i dati del kernel.
2. Le pagine allocate ai processi in modalità utente non devono necessariamente trovarsi in una memoria fisica contigua. Tuttavia, alcuni dispositivi hardware interagiscono direttamente con la memoria fisica senza il vantaggio di un'interfaccia di memoria virtuale e di conseguenza potrebbero richiedere memoria che risiede in pagine fisicamente contigue.

Esaminiamo due **strategie per la gestione della memoria libera assegnata ai processi del kernel**: il **buddy system** e la **slab allocation**.

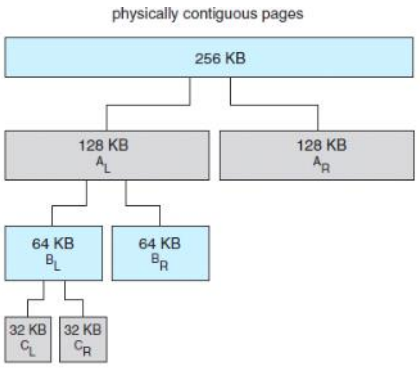
BUDDY SYSTEM

Il buddy system alloca la memoria in segmenti di dimensione fissa che contengono pagine contigue. La memoria viene allocata da questo segmento utilizzando un power-of-2 allocator, che soddisfa le richieste in unità dimensionate come una potenza di 2 (4 KB, 8 KB, 16 KB e così via).

Una richiesta in un unità non opportunamente dimensionata viene arrotondata alla successiva potenza di 2 più alta. Ad esempio, una richiesta di 11 KB viene soddisfatta con un segmento di 16 KB.

Consideriamo un semplice esempio:

> Supponiamo che la dimensione di un segmento sia inizialmente di 256 KB e che il kernel richieda 21 KB di memoria. Il segmento è inizialmente diviso in due **buddies** che chiameremo AL e AR ciascuno di 128 KB di dimensione. Uno di questi buddies viene ulteriormente suddiviso in due buddies da 64 KB, BL e BR. Tuttavia, la potenza di 2 più alta di 21KB è 32 KB, quindi BL o BR sono nuovamente divisi in due buddies da 32 KB, CL e CR. Uno di questi buddies viene utilizzato per soddisfare la richiesta di 21 KB. Questo schema è illustrato nella Figura, dove CL è il segmento assegnato alla richiesta di 21 KB.



Un vantaggio del buddy system è la rapidità con cui i buddies adiacenti, utilizzando una tecnica nota come coalescenza, possono essere combinati per formare segmenti più grandi.

Nella Figura qui di fianco, per esempio, quando il kernel rilascia l'unità C_L che era allocata, il sistema può unire C_L e C_R in un segmento di 64 KB. Questo segmento, B_L, può a sua volta essere unito al suo compagno B_R per formare un segmento di 128 KB. In definitiva, possiamo finire con il segmento originale da 256 KB.

Lo svantaggio del buddy system è che è molto probabile che l'arrotondamento per eccesso alla successiva potenza di 2 provochi la frammentazione all'interno dei segmenti allocati.

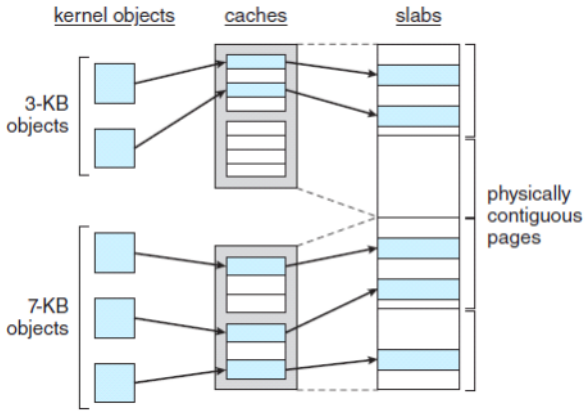
Ad esempio, una richiesta di 33 KB può essere soddisfatta solo da un segmento di 64 KB. Non possiamo infatti garantire che meno del 50% dell'unità assegnata andrà sprecata a causa della frammentazione interna.

SLAB ALLOCATION

La seconda strategia per l'allocazione della memoria del kernel è nota come **slab allocation**. Una lastra (slab) è composta da una o più pagine fisicamente contigue. Una cache consiste in una o più slab ed esiste una singola cache per ogni struttura dati del kernel. ad esempio, una cache separata per la struttura dati che rappresenta i process descriptors, una cache separata per gli object file, una cache separata per i semafori e così via.

Ogni cache è popolata da oggetti che sono istanze della struttura dati del kernel rappresentata dalla cache. Ad esempio, la cache che rappresenta i semafori memorizza istanze di oggetti semaforo, la cache che rappresenta i process descriptors memorizza istanze di oggetti process descriptors e così via.

La relazione tra slab, cache e oggetti è mostrata nella Figura qui di fianco. La figura mostra due oggetti del kernel di 3 KB di dimensione e tre oggetti di 7 KB di dimensione, ciascuno memorizzato in una cache separata.



L'algoritmo di slab allocation utilizza le cache per archiviare gli oggetti del kernel. Quando viene creata una cache, un numero di oggetti, inizialmente contrassegnati come liberi, viene allocato alla cache.

Il numero di oggetti nella cache dipende dalla dimensione della slab associata. Ad esempio, uno slab da 12 KB (costituito da tre pagine contigue da 4 KB) potrebbe memorizzare sei oggetti da 2 KB.

Inizialmente, nella cache, tutti gli oggetti nella cache sono contrassegnati come liberi. Quando è necessario un nuovo oggetto per una struttura dati del kernel, l'allocator, per soddisfare la richiesta, può assegnare qualsiasi oggetto libero dalla cache. l'oggetto assegnato dalla cache è contrassegnato come `used`.

Consideriamo uno scenario in cui il kernel richiede memoria dallo slab allocator per un oggetto che rappresenta un process descriptor. Nei sistemi Linux, un process descriptor è del tipo `struct task_struct`, che richiede circa 1,7 KB di memoria.

Quando il kernel di Linux crea una nuova attività, richiede la memoria necessaria per l'oggetto `struct task_struct` dalla sua cache. La cache soddisferà la richiesta utilizzando un oggetto `struct task_struct` che è già stato assegnato in una slab ed è contrassegnato come libero.

In Linux, una slab può trovarsi in uno dei tre possibili stati:

- **Full**: Tutti gli oggetti nella slab sono contrassegnati come utilizzati.
- **Empty**: Tutti gli oggetti nella slab sono contrassegnati come liberi.
- **Partial**: La slab è composta sia da oggetti usati che liberi.

Lo slab allocator tenta prima di soddisfare la richiesta con un oggetto libero che risiede in un partial slab. Se non esiste, viene assegnato un oggetto libero da un empty slab. Se non sono disponibili empty slab, viene allocata una nuova slab dalle pagine fisiche contigue e viene assegnata a una cache; la memoria per l'oggetto viene allocata da questa slab.

Lo slab allocator offre due vantaggi principali:

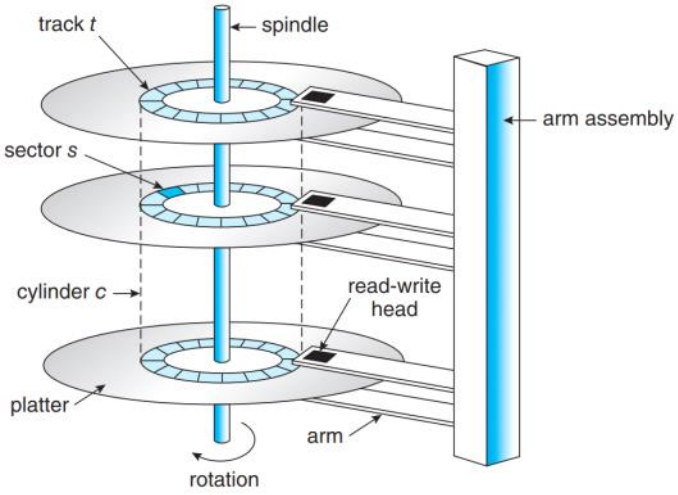
- Nessuna memoria viene sprecata a causa della frammentazione. La frammentazione non è un problema perché ogni struttura dati del kernel ha una cache associata e ogni cache è costituita da una o più slab suddivise in blocchi delle dimensioni degli oggetti rappresentati. Pertanto, quando il kernel richiede memoria per un oggetto, lo slab allocator restituisce l'esatta quantità di memoria richiesta per rappresentare l'oggetto.
- Le richieste di memoria possono essere soddisfatte rapidamente. Lo schema di slab allocation è quindi particolarmente efficace per la gestione della memoria quando gli oggetti vengono frequentemente allocati e deallocati, come spesso accade con le richieste dal kernel. L'atto di allocare e rilasciare la memoria può richiedere molto tempo. Tuttavia, gli oggetti vengono creati in anticipo e quindi possono essere assegnati rapidamente dalla cache. Inoltre, quando il kernel ha finito con un oggetto e lo rilascia, questo viene contrassegnato come libero e riportato nella sua cache, rendendolo così immediatamente disponibile per le successive richieste del kernel.

4 Gestione dei sistemi I/O

4.1 STRUTTURA DEI DISPOSITIVI DI ARCHIVIAZIONE DI MASSA

Il principale sistema di mass-storage nei computer moderni è la memoria secondaria, generalmente fornita tramite HDD (*hard disk drives*) e i dispostivi NVM *(nonvolatile memory).* Alcuni sistemi utilizzano anche una più lenta e più grande memoria terziaria, che generalmente consiste in nastri magnetici, dischi ottici o archiviazione cloud.

HDD



L'HDD è un dispositivo di archiviazione di massa che utilizza dei dischi magnetici per memorizzare dei dati. Il suo funzionamento è piuttosto semplice: ogni HDD è composto da più dischi piatti, simili a CD, posti uno sopra l'altro, entrambe le superfici del disco sono rivestite di un materiale magnetico. Le informazioni vengono memorizzate magneticamente sui dischi e vengono lette mediante il rilevamento della sequenza magnetica.
Le **testine** di lettura/scrittura "volano" appena sopra la superficie di ogni disco e sono attaccate a dei braccetti che si muovono all'unisono.
La superficie di ogni piatto è divisa in **tracce** circolari che sono suddivise in **settori**. Il settore 0 è il primo settore del cilindro più esterno. L'insieme di tracce di ogni disco ad una data altezza dei braccetti definisce un **cilindro**. Ogni HDD può contenere migliaia di cilindri ed ogni traccia può contenere centinaia di settori. I settori hanno una dimensione fissata e sono l'unità di trasferimento più piccola. La grandezza dei settori fino a circa il 2010 è comunemente stata di 512 byte, dal 2010 in poi si è iniziato ad usare settori di 4KB.
Il motore dei dischi li fa girare a velocità molto alte (tra i 60 e i 250 giri al secondo), specificato in **RPM** (giri al minuto), generalmente si hanno tra i 5400 e i 15000 RPM. Molti hard disk fermano il motore quando non sono usati, facendo girare i dischi solo quando richieste operazioni di I/O. La velocità di rotazione influenza il tempo di trasferimento. Un altro aspetto che influenza le performance è il **positioning time**, o **random-access time**, che consiste sostanzialmente in 2 parti: **seek time** (*tempo di posizionamento*) cioè il tempo necessario per muovere i braccetti sul cilindro desiderato; **rotational latency** (latenza di rotazione) cioè il tempo richiesto perché un determinato settore ruoti fino a trovarsi sotto la testina.
Tipicamente gli HHD possono trasferire centinaia di megabyte al secondo, velocità definita dal **transfer rate**, con tempi di posizionamento di vari millisecondi (ore in termini di computazionali). Per migliorare le performance sono posizionati nei drive controller dei DRAM buffer.
Possiamo quindi affermare la seguente regola:

$$Data\ Transfer\ Time = seek\ time + rotational\ delay + transfer\ rate$$

Le testine, come già detto, "volano" su un piccolissimo cuscino d'aria (o di elio), ad una distanza dal disco misurabile nell'ordine dei micron. Esiste la possibilità, seppur bassa, che la testina vada in contatto con la superficie del disco. I dischi sono protetti da un sottile strato protettivo, che però in alcuni casi può essere "bucato" dalla testina, questo incidente viene chiamato **head crash**. L'head crash generalmente non può essere riparato. Per ridurre questo rischio, quando l'HDD è spento le testine vengono "parcheggiate" fuori dal disco, o in un'area dove non sono presenti dati.

Gli hard drive possono essere rimovibili come i floppy disk, alcuni possono essere anche rimossi mentre il computer è in esecuzione. i dispositivi di archiviazione secondari comunicano con il computer tramite bus di sistema o bus di I/O, alcuni dei bus più comuni includono:

- Enhanced Integrated Drive Electronics (EIDE)
- Advanced Technology Attachment (ATA) and Serial ATA (SATA)
- Universal Serial Bus (USB)
- Fiber Channel (FC)
- Small Computer Systems Interface (SCSI)

I trasferimenti dati su bus sono effettuati da appositi elaboratori elettronici chiamati **controller**. L'**host controller** è il controller lato computer del bus, mentre il **device controller** è costruito su ogni dispositivo di archiviazione.

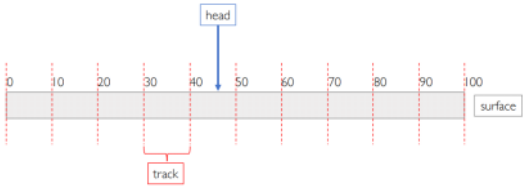
4.2 SCHEDULING DEL DISCO

A causa delle basse prestazioni dei dispositivi di archiviazione di massa, il sistema operativo interviene per aumentarne l'efficienza. L'idea è quella di riordinare l'ordine originale di arrivo delle richieste, in modo tale da ridurre la lunghezza e il numero di ricerche sul disco (cercheremo quindi di ridurre il seek time).

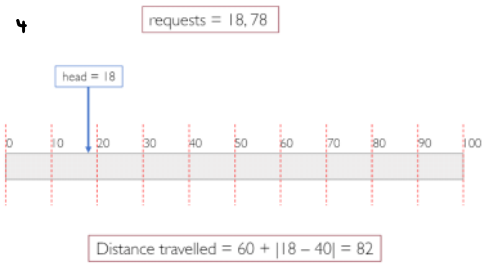
FCFS

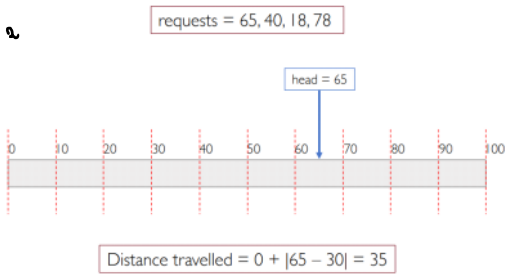
Iniziamo con l'algoritmo più banale, serviamo le richieste nell'ordine di arrivo, analizziamo un caso pratico.

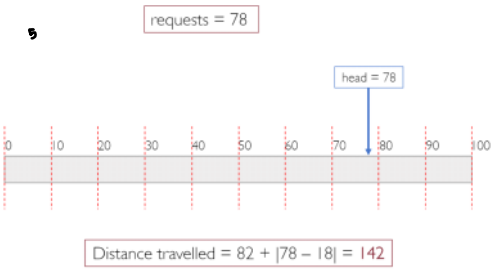
Dato il seguente disco

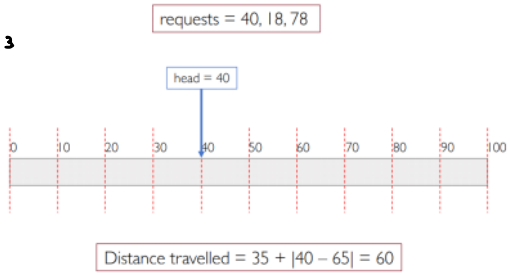


Supponiamo che arrivino delle richieste in ordine delle seguenti tracce: 65, 40, 18, 78.
Analizziamo il movimento della testina con l'algoritmo FCFS



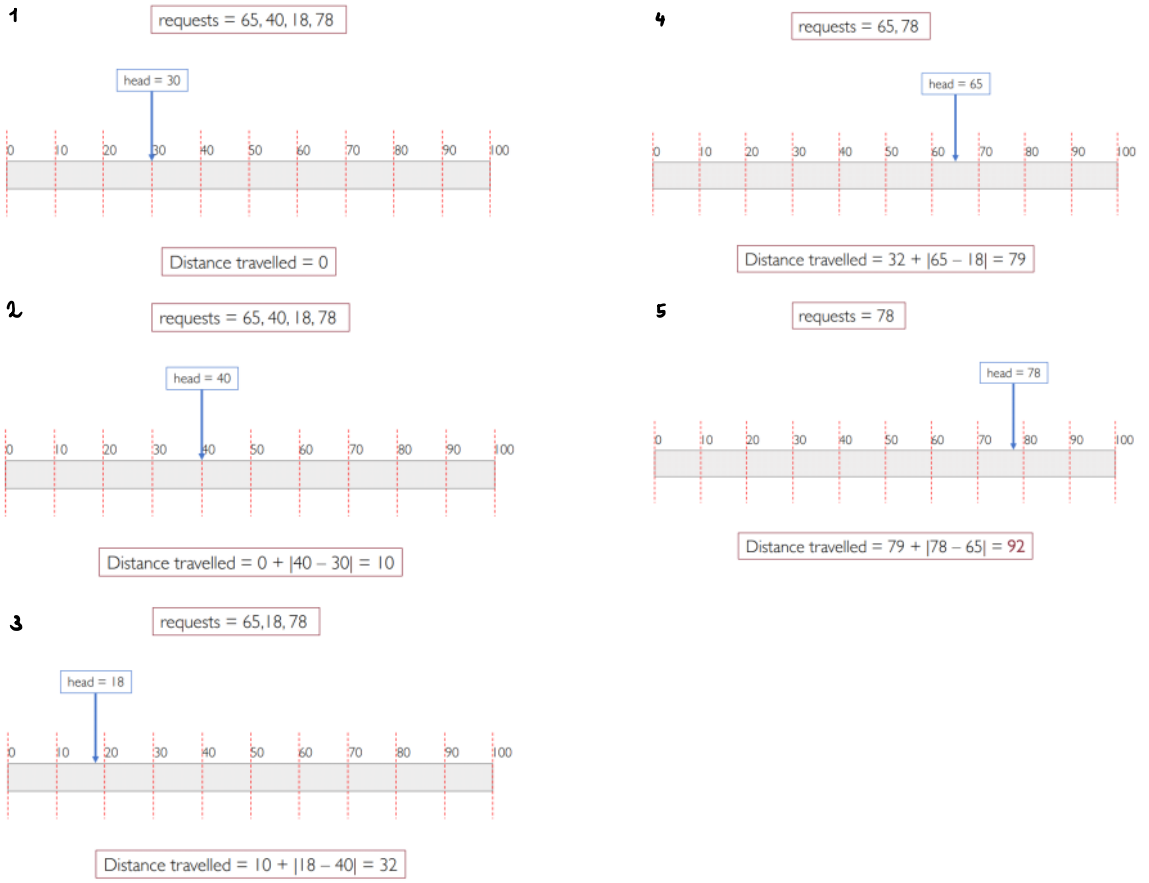






SSTF

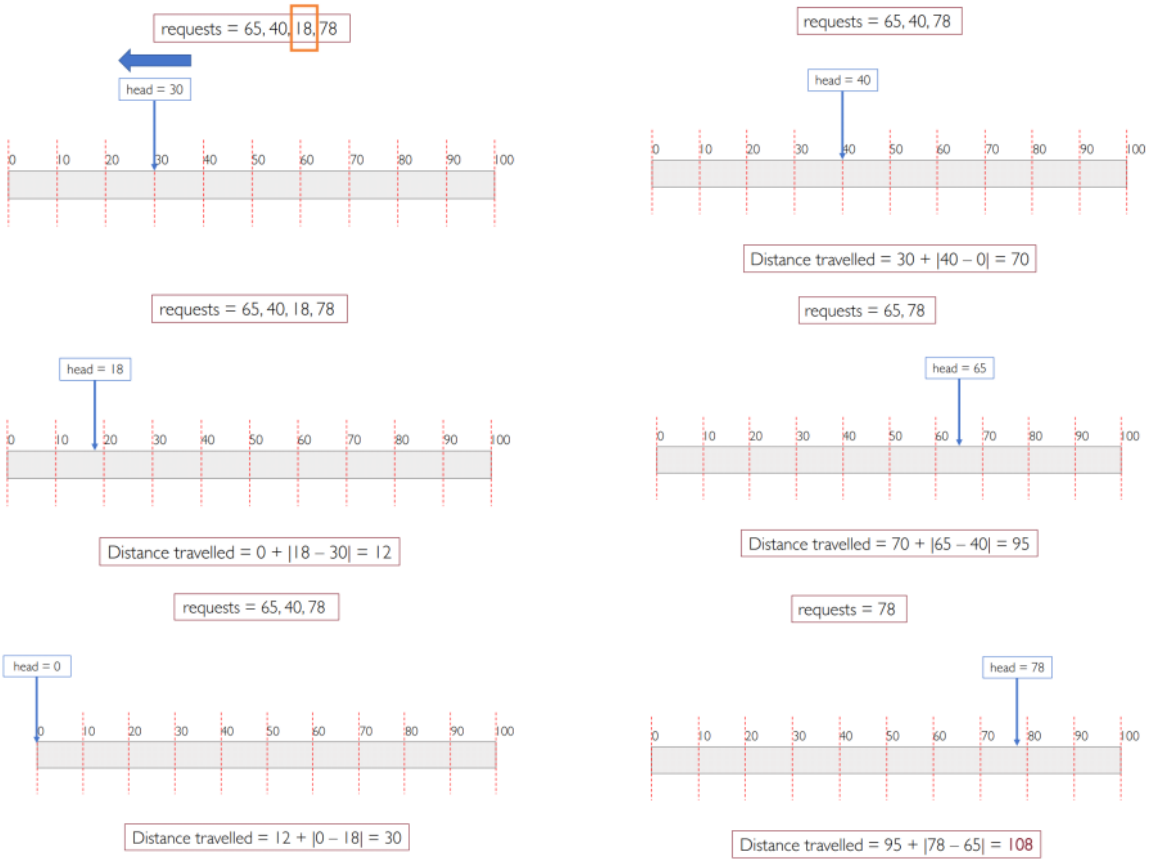
Questo algoritmo si basa sul concetto di far muovere la testina alla traccia più vicina



Non è un algoritmo complessivamente ottimale in quanto può causare starvation (se una richiesta è troppo lontana dalla posizione della testina)

SCAN

In questo algoritmo la testina si muove continuamente avanti e indietro (dall'inizio alla fine poi vice versa) e le richieste sono soddisfatte al passaggio della testina.



Una semplice ottimizzazione di questo algoritmo può essere quella di non dover per forza ogni volta arrivare agli estremi, ma fermarsi alla richiesta più vicina all'estremo dove la testina si sta dirigendo (nell'esempio precedente, la testina poteva fermarsi a 18 senza dover arrivare a 0, in questo modo la distanza percorsa sarebbe scesa a 72).

L'algoritmo SCAN non soffre di possibile starvation come invece ne può soffrire SSTF, anche se il seek time dell'SSTF è generalmente minore. La debolezza dello SCAN sono le richieste di tracce appena visitate, che potrebbero dover aspettare molto.

C-SCAN

In questo algoritmo la testina esegue uno "SCAN circolare": una volta raggiunta un'estremità, la testina si sposta immediatamente all'altra estremità eseguendo un nuovo SCAN nello stesso senso.

Similmente allo SCAN, possiamo applicare una miglioria anche al C-SCAN: non c'è bisogno che la testina arrivi per forza ad un estremo e riparta dall'altro, la testina può fermarsi alla richiesta più vicina ad un estremo e ripartire dalla richiesta più vicina all'altro estremo (la distanza percorsa verrebbe abbassata da 190 a 110).

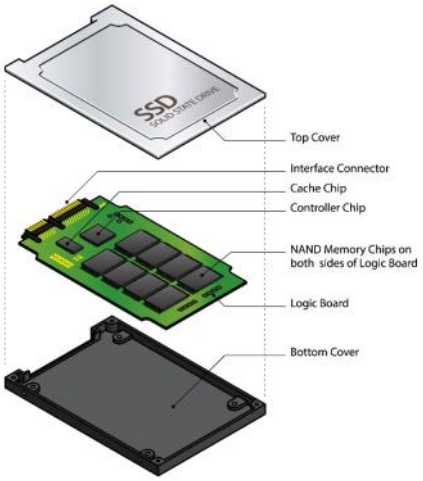
Tutti gli algoritmi sopra esposti vengono implementati direttamente sul disk controller (gli hard disk vengono forniti con uno di questi algoritmi già installato).

MIGLIORAMENTO DELLE PRESTAZIONI

L'allocazione contigua dei file sui blocchi del disco ha senso solo se il sistema operativo può reagire a una richiesta del disco ed emettere la successiva prima che il disco giri sul blocco successivo. Ad oggi le CPU sono abbastanza veloci da soddisfare subito le richieste e consentire l'allocazione contigua dei blocchi (un tempo però l'allocazione dei blocchi contigui veniva effettuata a 2/3 blocchi di distanza, in modo da dare il tempo alla CPU di formulare e mandare i dati della richiesta).
Un modo invece per ridurre il numero di ricerche effettuate sul disco, è quello di leggere i blocchi dal disco prima delle richieste del processo, archiviandoli in un buffer (cache) del disk controller (vengono ad esempio caricati tanti blocchi della traccia che si sta leggendo, anche se non esplicitamente richiesti).

4.3 SSD

Gli algoritmi appena analizzati vengono esclusivamente applicati ai dispositivi di archiviazione meccanici basati su dischi piatti, come le HDD. Le SSD (Solid State Drive) fanno invece parte dei dispositivi NVM che non richiedono dischi e testine di lettura/scrittura, difatti usano generalmente algoritmi FCFS.
Le SSD sono dispositivi di archiviazione di massa in costante crescita nell'utilizzo quotidiano. Sono generalmente composte da un controller, una cache e una serie di chip semiconduttori flash NAND utilizzati per l'archiviazione dei dati protetti da una batteria. Non avendo componenti meccaniche le SSD sono nettamente più veloci de tradizionali hard drive.
L'accesso ai blocchi avviene referenziando direttamente i numeri dei blocchi.

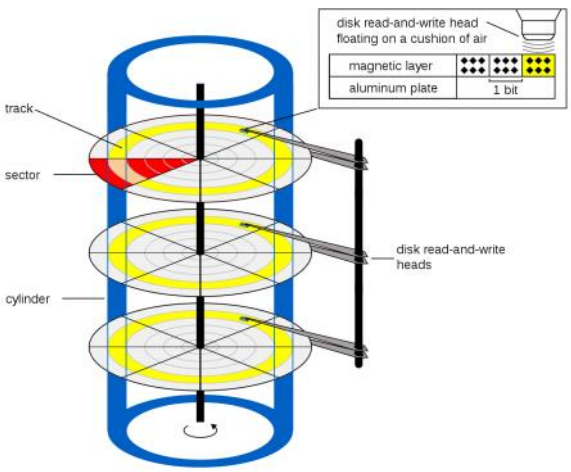


Le operazioni di lettura sono molto veloci, quelle di scrittura sono però più lente in quanto i dati non possono essere direttamente sovrascritti, necessitano invece di essere cancellati con un ciclo prima di poter essere riscritti. Le SSD hanno un ciclo di scrittura limitato (non si possono riscrivere dati all'infinito), per mantenere il bilanciamento delle scritture, ogni blocco ha un contatore delle riscritture effettuate. Una volta finite tutte le riscritture disponibili non se ne possono fare altre (problema che affligge anche le HDD).
Le SSD sono più costose degli hard drive e potrebbero avere una vita più breve. Le SSD sono specialmente utili come cache ad alta velocità per le informazioni degli hard disk che devono essere accedute rapidamente.
Le SSD sono anche usate nei laptop per renderli più piccoli, leggeri e veloci. Essendo però la SSD molto più veloce dei tradizionali hard disk, il throughtput dell'I/O bus potrebbe essere un fattore limitante della velocità. Alcune SSD vengono quindi collegate direttamente al system PCI bus.

4.4 DISK MANAGEMENT

FORMATTAZIONE

Prima che un disco possa essere usato necessita di essere formattato a basso livello. Ciò significa stabilire l'inizio (**header**) e la fine (**trailer**) di ogni settore. Header e trailer determinano gli estremi del settore e contengono i codici di correzione degli errori (ECC), per scopi di rilevamento e correzione. L'ECC viene eseguito ad ogni lettura/scrittura del disco, se viene rilevato un danno riparabile ai dati, viene risolto dallo stesso disk controller.
Una volta che il disco è formattato, il passo successivo è quello di partizionarlo in una o più partizioni. Questa operazione va eseguita anche se il disco viene usato come un'unica grande partizione.



Dopo la partizione va formattato logicamente il file system.

BOOT BLOCK

La ROM del computer contiene un **bootstrap program** (il primo programma ad essere eseguito all'avvio del sistema, indipendente dall'OS), con il codice appena necessario per trovare il primo settore sul primo hard drive sul primo controller. Questo carica il settore nella memoria e trasferisce il controllo ad esso. Il primo settore è conosciuto come **Master Boot Record (MBR)**, contiene una piccola quantità di codice e la **partition table**. La partition table ci dice come il disco sia logicamente partizionato e quale partizione sia attualmente attiva/attivabile. A questo punto il boot program guarda le partizioni attive alla ricerca di un sistema operativo. Una volta trovato il kernel, viene caricato nella memoria e il controllo viene trasferito al sistema operativo, il quale inizializza tutte le strutture dati importanti del kernel e i servizi di sistema.

4.5 NASTRI MAGNETICI

Il **nastro magnetico** è un supporto di memorizzazione a memoria magnetica che consiste in una sottile striscia in materiale plastico, rivestita di un materiale trattato con polarizzazione magnetica.
Sono usati principalmente per i backup. L'accesso ad un particolare punto può essere molto lento (non vi è accesso randomico/diretto, solo sequenziale).
La capacità dei nastri magnetici va dai 20 ai 200GB, al giorno d'oggi sono completamente rimpiazzati dai dischi.

4.6 ADVANCED TOPICS

Di seguito sono esposti gli ultimi argomenti trattati durante corso.

RAID STRUCTURE

Il RAID (Redundant Array of Independent Disks - *insieme ridondante di dischi economici*), è una tecnica di installazione raggruppata di diversi dischi rigidi economici in un computer (o collegati ad esso), che fa sì che gli stessi nel sistema appaiano e siano utilizzabili come se fossero un unico volume di memorizzazione, invece di doverne utilizzare uno o due molto più costosi.
I principali scopi del RAID sono: aumentare le performance, rendere il sistema resiliente alla perdita di uno o più dischi e poterli rimpiazzare senza interrompere il servizio.

DISK FAILURE

Definiamo "fallimento del disco" qualunque problema che renda impossibile l'accesso ai dati (come ad esempio un head crash)
I sistemi moderni possono richiedere l'utilizzo di parecchi dischi, basti pensare ai grandi server. Sorge però un problema: più dischi sono presenti in un sistema e maggiore sarà la probabilità che uno di essi abbia problemi in un dato momento. Infatti, incrementando il numero di dischi in un sistema, viene diminuito il **Mean Time To Failure** (MTTF) del sistema, cioè il tempo medio della comparsa di fallimenti all'interno del sistema.
Il fallimento di un singolo disco in un dato giorno è un evento abbastanza raro (circa 0.025% di possibilità). Ciò non è più così infrequente se lavoriamo con 4000 dischi (circa un fallimento al giorno), o con 400'000 dischi (fino a 100 fallimenti al giorno).

MIRRORING

Utilizziamo il **mirroring** (copiare gli stessi dati su dischi multipli) per aumentare la resilienza dei dati ai fallimenti. I dati non verranno infatti persi fin quando tutte le copie di tali dati non verranno perse contemporaneamente. Questa possibilità è molto più bassa di quella del fallimento di un singolo disco.

Il mirroring migliora anche le performance, particolarmente con le operazioni di lettura: ogni blocco di dati è duplicato su multipli dischi, e la lettura può essere effettuata da ogni copia disponibile. Dischi multipli possono leggere differenti blocchi allo stesso momento. Anche la scrittura può essere resa più veloce attraverso attenti algoritmi di scheduling, ma è effettivamente più complessa nella pratica.
Un altro modo di migliorare il tempo di accesso al disco è tramite lo **stripping**. Ciò significa diffondere i dati su più dischi che possono essere acceduti simultaneamente. Gli stripped disk sono logicamente visti come una singola unità.

In sintesi:

- Il mirroring migliora la resilienza dei dati, ma aumenta lo spreco di spazio
- Lo stripping migliora le performance, ma non aumenta la resilienza

Si può utilizzare un numero di differenti schemi combinati cerca di bilanciare tra i due:

- RAID level 0 - stripping only
- RAID level 1 - mirroring only
  .
  .
  .
- RAID level 6 - stripping + mirroring + parity bit