



Sistemi Operativi

Appunti del corso di Sistemi Operativi

▼ Lezione del 03/10

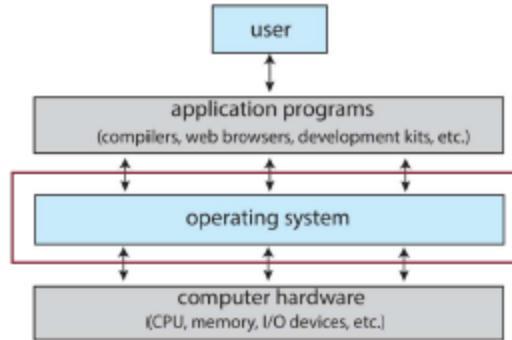
Introduzione generale

Cos'è il Sistema Operativo?

Un **sistema operativo** è un software che è alla base delle attività del calcolatore che è in grado di gestire le risorse hardware e software della macchina.

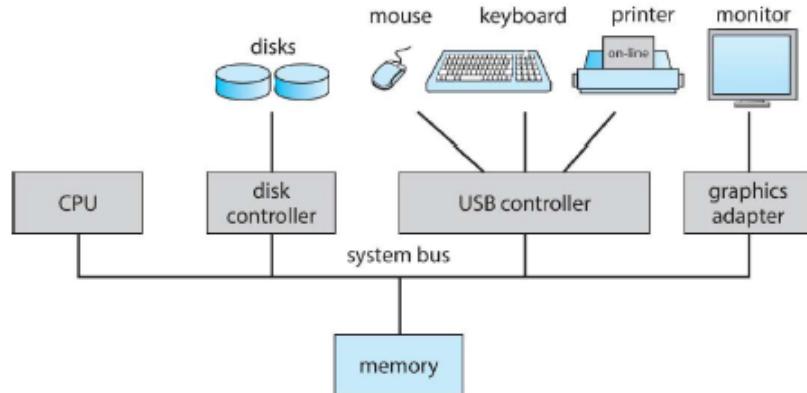
Un sistema operativo è costituito da diversi componenti, alcuni di questi sono simili tra tutti gli OS:

- **Kernel**, ossia il "cuore" del sistema operativo, il quale è sempre attivo, è possibile implementarlo come Micro-Kernel o Macro-Kernel.
- **Programmi di sistema**, ossia ogni altro software del sistema operativo
Tra i vari compiti svolti da un OS, in particolare troviamo:
 - **Gestione** delle risorse fisiche e logiche utilizzate dai vari software.
 - **Virtualizzazione** delle risorse, dando l'"illusione" ad ogni software di aver a disposizione "infinite risorse".
 - **Interfacciamento** tra Hardware e Software, fornendo un insieme di servizi comuni, ossia delle API, che permettono ai software e agli utenti di interagire con le risorse hardware senza doverle controllare in modo diretto.



Un'immagine semplificata che spiega il ruolo del sistema operativo.

Un computer è costituito da diverse parti



Le principali parti di un calcolatore sono:

- **CPU:** il processore che svolge la computazione dei programmi.
- **Memoria Principale:** Immagazzina i dati e le istruzioni per la CPU, è divisa tra la CPU e i sistemi di I/O.
- **Sistemi di I/O:** Monitor, mouse tastiera ecc.. associati a specifici device di controllo.
- **Sistem Bus:** ossia un mezzo di comunicazione tra CPU, memoria e dispositivi di I/O, composto a sua volta da:

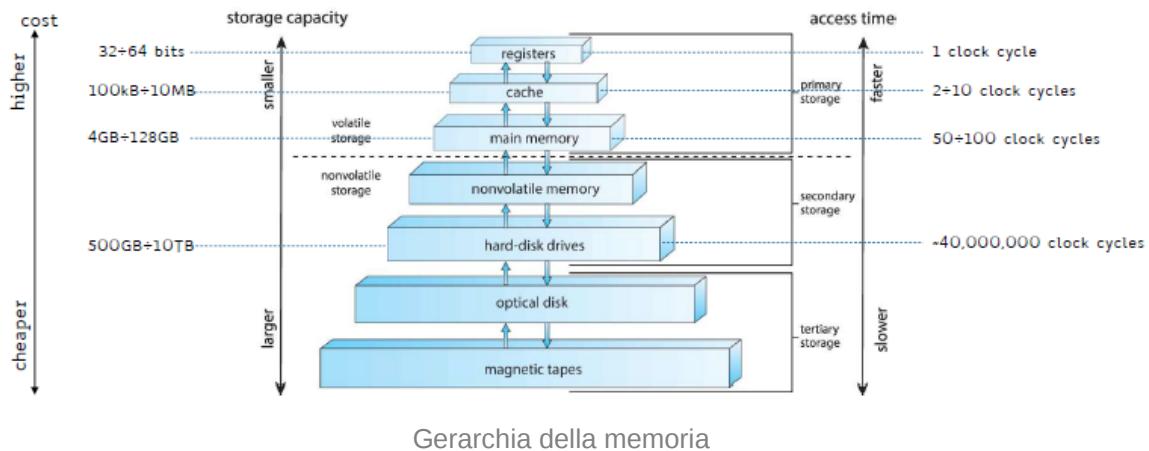
o -

La CPU

La CPU è un componente **fondamentale** di tutti i calcolatori, essa esegue le istruzioni descritte in linguaggio macchina (sequenze di bit), dove ogni **word** corrisponde ad una singola istruzione appartenente ad un insieme di istruzioni elementari.

Ogni CPU può essere formata da un singolo processore o da più di uno (**sistemi multicore**).

La Memoria



La memoria principale è essenzialmente una sequenza di celle, ogni cella è organizzata in gruppi di 8 bit (1 byte), ognuna di esse ha il proprio **indirizzo di memoria**.

E' fondamentale per la CPU e i sistemi di I/O poiché essi possono leggere o scrivere dati in essa.

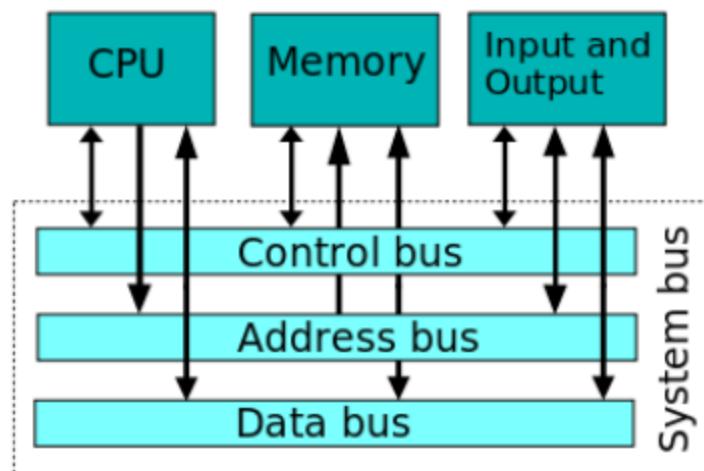
Computer Buses

Come già definito prima esistono diversi tipi di Bus.

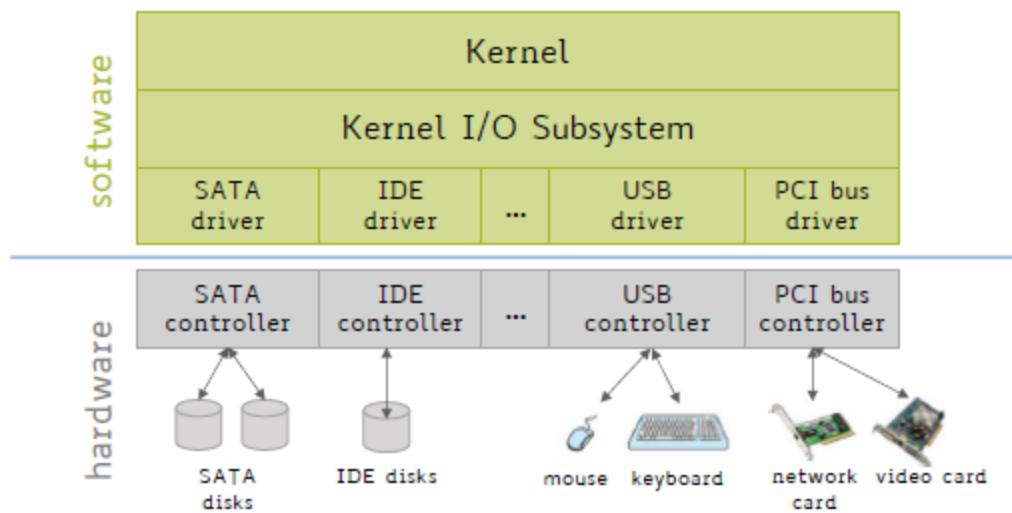
Inizialmente nei primi calcolatori vi era solo un Bus al quale erano collegati tutti i device e i sistemi del computer.

Ad oggi usiamo combinazioni di tre tipi di Bus definiti nel **System Bus**:

- **Data Bus**, il quale trasporta i dati effettivi
- **Address Bus**, il quale determina dove i dati debbano essere inviati
- **Control Bus**, il quale indica quale operazione deve essere effettuata



Sistemi di I/O



Ogni dispositivo di **I/O** è composto da due parti: il **dispositivo fisico** in se e il

device controller, ossia un insieme di piccoli chip che lo gestisce. Il sistema operativo

interagisce con tali dispositivi tramite dei driver, ossia dei particolari software adattati per ogni dispositivo che permettono di "tradurre" le richieste dell'OS.

Ogni device controller possiede una serie di registri dedicati con cui comunicare:

- Registri di stato, i quali forniscono informazioni alla CPU riguardo il dispositivo
- Registri di controllo, i quali vengono utilizzati dalla CPU per configurare e controllare il dispositivo
- Registri dati, i quali vengono utilizzati per lo scambio di dati con il dispositivo.

Per poter comunicare con i dispositivi di I/O, la CPU deve essere in grado di poter distinguere tra indirizzi della memoria principale ed indirizzi rappresentanti i dispositivi

stessi. A tal fine, il control bus è dotato di una linea speciale detta

M#IO in grado di

stabilire se la CPU desideri comunicare con la memoria principale o con i dispositivi

di
I/O, funziona come uno switch.

La CPU può dialogare con i device di controllo in due modi:

Comunicazione Port-Mapped

Nella comunicazione **Port-mapped**, i riferimenti ai controller vengono effettuati utilizzando uno spazio di indirizzi **aggiuntivo** dedicati solo all'accesso ai dispositivi di I/O, necessitando l'implementazione di **istruzioni aggiuntive** del tipo IN ed OUT che vadano a specificare se l'indirizzo sia riferito alla memoria o ad un dispositivo I/O.

```
MOV DX, 1234h //carica nel registro DX il valore 0x1234
MOV AL, [DX]   //carica nel registro DX il valore contenuto
                //nell'indirizzo 0x1234 della memoria
IN AL, DX      //carica nel register DX il valore letto dal
                //dispositivo connesso alla porta 0x1234
```

Comunicazione Memory-Mapped

Nella comunicazione **Memory-mapped** I/O, parte degli indirizzi della memoria viene "sprecato" per poter essere utilizzato per effettuare i riferimenti ai dispositivi I/O, **non necessitando istruzioni aggiuntive** e permettendo alla CPU di trattare i registri dei control device come se fossero dei suoi registri aggiuntivi

Gestione attività di I/O

Per eseguire le attività di I/O, vengono utilizzate due modalità di gestione:

- **Polling**, dove la CPU controlla periodicamente lo stato attuale dell'attività di I/O

- **Interrupt driven**, dove la CPU riceve un segnale di interrupt dal controller una volta che la task I/O viene completata

Tali operazioni di I/O possono essere svolte direttamente dalla CPU, la quale si occuperà

del trasferimento dei dati, oppure da un

Direct Memory Access Controller (DMA Controller)

, al quale viene delegato tale compito.

Solitamente, un DMA Controller viene utilizzato in combinazione alla modalità di gestione interrupt driven al fine di rimuovere ogni compito di gestione delle attività I/O dalla CPU.

Nella comunicazione Port-mapped, i riferimenti ai controller vengono effettuati utilizzando

uno spazio di indirizzi aggiuntivo dedicati solo all'accesso ai dispositivi di I/O, necessitando l'implementazione di istruzioni aggiuntive del tipo IN ed OUT che vadano a specificare se l'indirizzo sia riferito alla memoria o ad un dispositivo I/O

▼ Lezione del 05/10

Modern Computer System

Tra le varie tipologie di Sistemi Moderni abbiamo i **sistemi multi-programmabili** in grado di mantenere caricati in memoria più task contemporaneamente.

In questi sistemi, l'OS si occupava dell'organizzazione dell'esecuzione dei programmi, detto **job scheduling**, delle operazioni di I/O e della protezione della memoria, impedendo ai vari job di accedere ad aree utilizzate da altri.

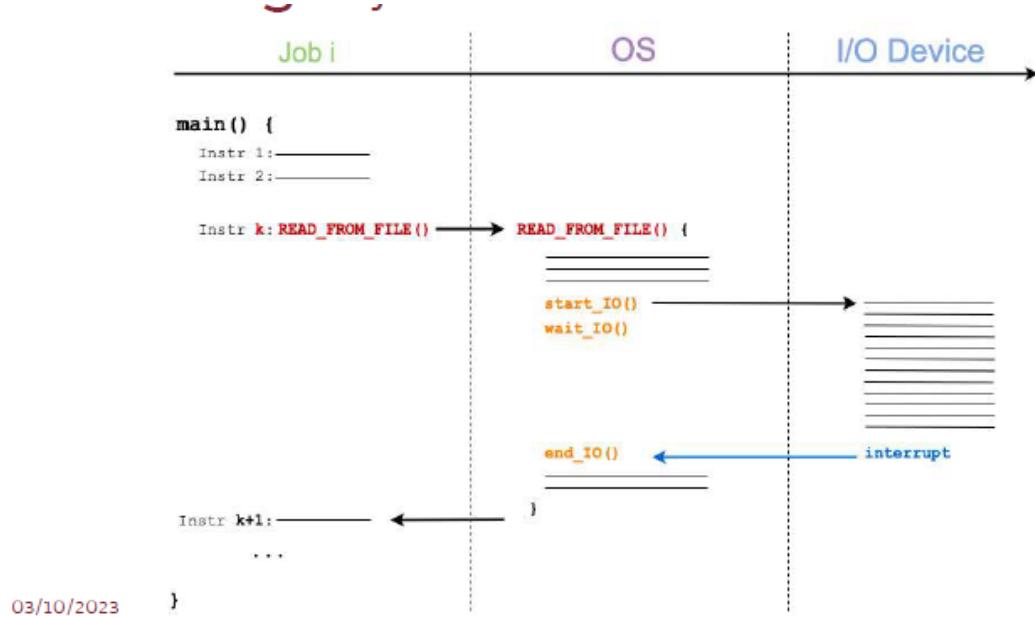
L'unica problematica di tali sistemi risultò essere la necessità di mantenere la CPU **inattiva** durante l'esecuzione di operazioni di I/O bloccanti.



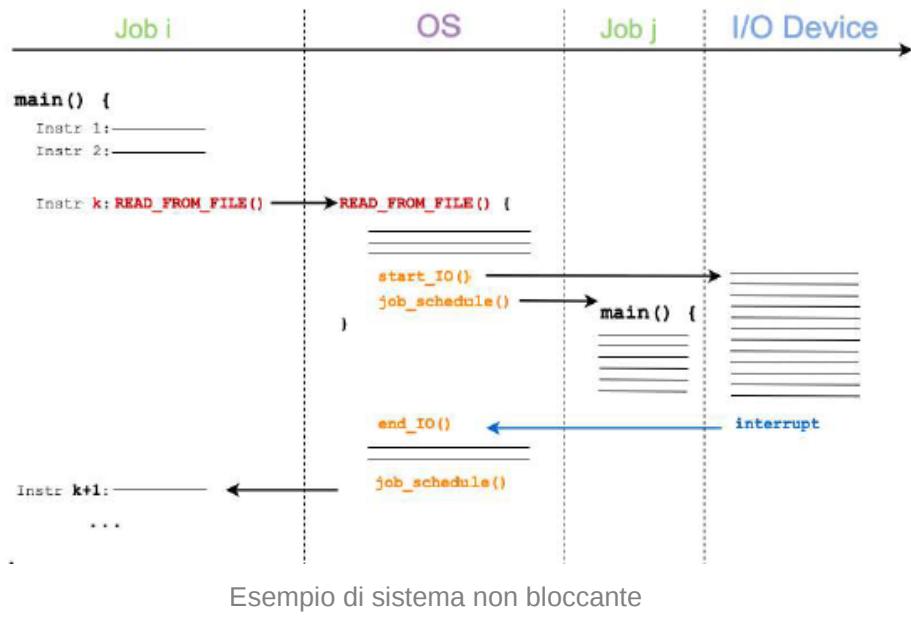
Per **sistema bloccante** si intende un sistema nel quale la CPU resta inattiva in un determinato arco temporale nel quale vengono svolte attività di I/O dalla DMA e viene “risvegliata” solo con un segnale di interruzione, questo limita notevolmente le performance della CPU.



I sistemi **non bloccanti** sono sistemi nei quali la CPU mentre viene effettuata un’operazione di I/O avvia una **task successiva** che però viene interrotta non appena alla CPU arriva il segnale di interruzione.



Esempio di sistema bloccante

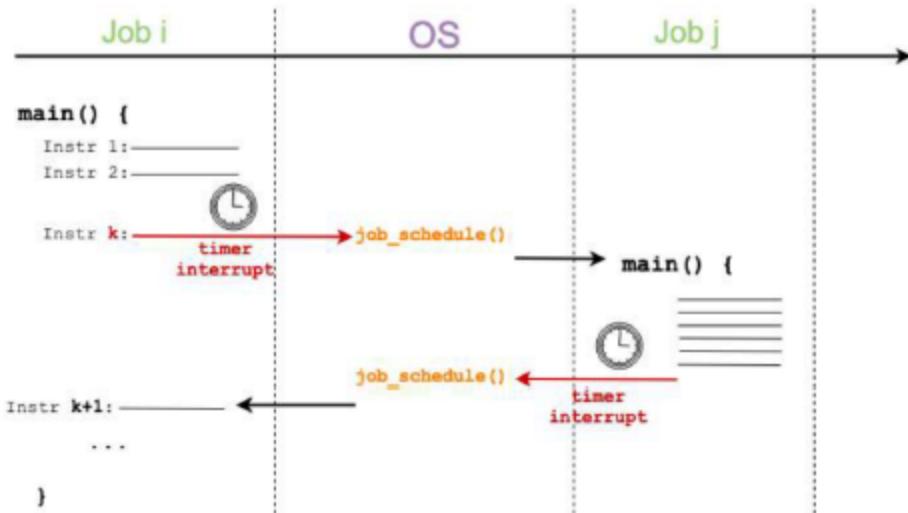


Sistemi time-sharing

Viene utilizzato un'**interruzione programmata** a tempo per alternare la CPU tra i vari job, dando l'**illusione** di esecuzione parallela (**pseudo-parallelismo**).

In particolare, è famoso il sistema operativo ideato da Ken Thompson e Dennis Ritchie, ossia UNIX, precursore di sistemi operativi come MacOS e le varie distribuzioni di Linux.

E' importante far notare come la scelta dell'intervallo di tempo tra una task e l'altra sia una **decisione** di design **molto importante**, poiché quando la CPU interrompe una task ne immagazzina lo stato fino a quel momento, poi procede a caricare lo stato della prossima task al momento precedente del cambio, scegliere quindi un tempo troppo piccolo farebbe sì che il tempo tra il salvataggio della task corrente e il caricamento della prossima lasci pochissimo tempo all'effettivo svolgimento dell'operazione.



Esempio di sistema time-sharing

Servizi del sistema operativo

Numerose funzionalità architetturali sono gestite dal SO.

| OS Service | HW Support |
|-------------------------|--|
| Protection and Security | Kernel/user mode, protected instructions, base/limit registers |
| System calls | Trap instructions and interrupt vectors |
| Exception handling | Trap instructions and interrupt vectors |
| I/O operations | Trap instructions, interrupt vectors, and memory mapping |
| Scheduling | Timer |
| Synchronization | Atomic instructions |
| Virtual memory | Translation Look-aside Buffer (TLB) |

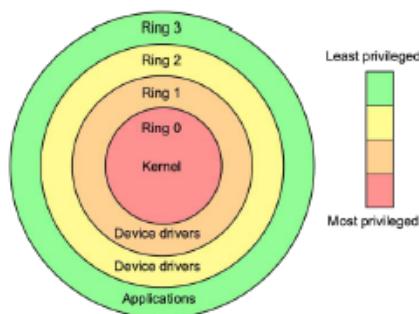
Alla base della divisione delle mansioni del sistema operativo vi è la distinzione tra **istruzioni privilegiate** e non, le istruzioni privilegiate sono quelle istruzioni sensibili che è possibile effettuare solo tramite **Kernel**.

Kernel vs. User Mode

La CPU può essere impostata in:

- **Kernel mode**, ossia in modalità senza alcuna restrizione, permettendo l'esecuzione di qualsiasi istruzione (utilizzata dal sistema operativo).
- **User mode**, dove non è possibile:
 - Accedere agli indirizzi riservati ai dispositivi di I/O
 - Manipolare il contenuto della memoria principale
 - Arrestare il sistema
 - Passare alla Kernel Mode

Per poter impostare una delle due modalità, viene utilizzato un **bit speciale** salvato in un registro protetto: se impostato su **0** la CPU sarà in **Kernel mode**, mentre se impostato su **1** la CPU sarà in **User mode**. Nelle architetture più moderne, vengono implementati più livelli di protezione, detti **protection rings**. Ogni ring aggiunge restrizioni sulle istruzioni eseguibili dalla CPU.

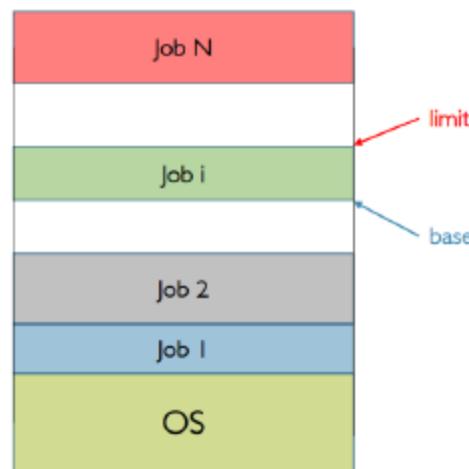


Protezione della memoria

Oltre alle protezioni imposte sulle istruzioni eseguite, è necessario imporre delle **protezioni** anche sulla **memoria**, in modo da proteggere reciprocamente gli spazi di memoria riservati ai singoli programmi degli utenti e in modo da proteggere gli spazi di memoria riservati all'OS dai programmi utente.

La tecnica più semplice per ottenere tale protezione è l'utilizzo di **due registri** dedicati, i quali vengono caricati dall'OS all'avvio di un programma l'OS, per poi controllare che ogni indirizzo richiesto durante l'esecuzione del programma ricada nell'**intervallo specificato**.

- **Registro base**, contenente l'indirizzo iniziale dello spazio utilizzabile
- **Registro limite**, contenente l'indirizzo finale dello spazio utilizzabile



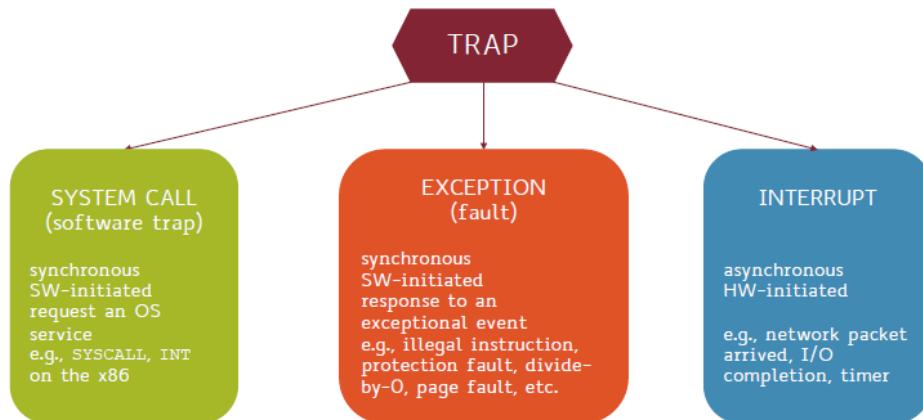
System Calls

Definiamo come **trap di sistema** un qualsiasi evento che richiede il passaggio **da user mode a kernel mode**.

In particolare, individuiamo tre tipi di trap:

- **System calls**, ossia la richiesta di un servizio dell'OS, svolte in modo sincrono e innescate dai software.

- **Exception**, ossia la gestione di errori dovuti ad eventi inattesi, svolte in modo sincrono e innescate dai software.
- **Interrupt**, ossia il completamento di una richiesta in attesa, svolte in modo asincrono e innescate dall'hardware.



Riassunto delle trap di sistema.



Definizione

Le

System calls sono delle procedure messe a disposizione dal sistema operativo per permettere agli altri software di poter accedere ai servizi messi a disposizione dal sistema operativo, in particolare l'esecuzione di alcune istruzioni privilegiate (ad esempio l'accesso ad un dispositivo I/O).

Le System call ricadono in **sei categorie**:

- **Process control**, include procedure come end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, allocate and free memory.

- **File management**, include procedure come create file, delete file, open, close, read, write, reposition, get and set file attributes.
- **Device management**, include procedure come request device, release device, read, write, reposition, get and set device attributes, logically attach or detach devices.
- **Information maintenance**, include procedure come get and set time, get and set date, get and set system data, get and set process, get and set file information, get and set device attributes.
- **Communications**, include procedure come create and delete communication connection, send and receive messages, transfer status information, attach or detach remote devices. Esistono 2 modelli diversi di comunicazione:
 - **Message passing:**
Questo approccio è più semplice (particolarmente per le comunicazioni tra computer) ed è appropriato per piccole quantità di dati.
 - **Shared memory:**
Questo approccio è più veloce e, in generale, è la scelta migliore quando si devono condividere grandi quantità di dati. È l'ideale quando la maggior parte dei processi ha bisogno di leggere dati anziché scriverli.

Quando un programma utente richiede l'esecuzione di una syscall tramite un'API fornita dal sistema operativo, tale richiesta viene convertita in un'istruzione di interrupt, richiamando l'**Interrupt Vector Table** (IVT) al momento della sua esecuzione, tramite cui verrà attivato il **System Call Handler**, il quale contiene una **System Call Table** tramite cui vengono mappate le syscall ai propri codici di esecuzione, per poi infine eseguire la syscall richiesta.

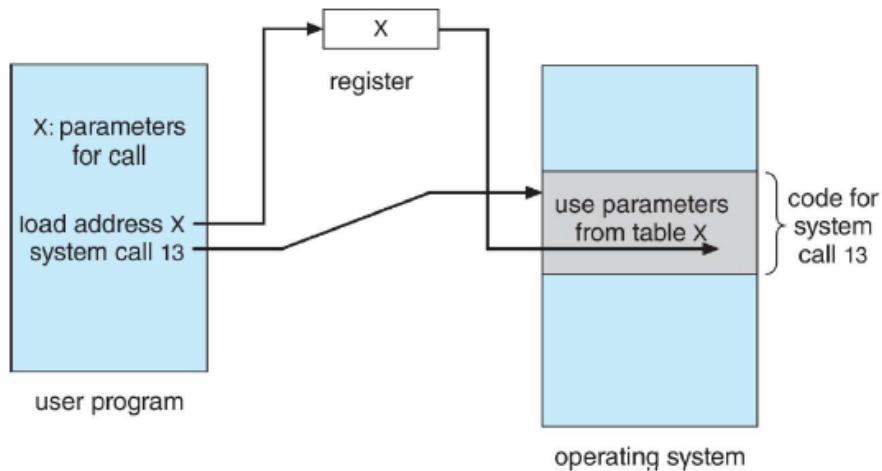
E' molto lento andare a ricercare il codice nella System Call Table perché richiede 2 accessi alla memoria.

Passaggio di parametri

Spesso è necessario fornire al sistema operativo **più informazioni** oltre all'identificatore della chiamata di sistema desiderata. Ci sono **3 metodi** utilizzati per passare i parametri al sistema operativo:

1. **Conservare i parametri nei registri** (potrebbero esserci più parametri che registri).
2. **Conservare i parametri in un blocco o tabella** in una zona di memoria dedicata, e l'indirizzo del blocco viene passato come parametro in un registro (usato in Linux e Solaris).
3. **I parametri vengono inseriti nello stack dal programma** e rimossi dallo stack dal sistema operativo (più complesso a causa degli spazi di indirizzamento differenti).

I metodi di blocco e stack non limitano il numero o la lunghezza dei parametri che vengono passati.



Passaggio di parametri via tabella.

Scheduling, Sincronizzazione e Virtualizzazione

Per poter gestire lo scheduling delle operazioni, viene implementato un **timer** nell'hardware, il quale genera un interrupt dopo una breve quantità di tempo, permettendo al **CPU scheduler** di prendere il controllo e decidere quale sia la prossima

operazione da eseguire, impedendo che la CPU venga monopolizzata da un job "egoista".

Inoltre, poiché le interrupt sono **asincrone**, esse potrebbero essere innescate in qualsiasi momento, interferendo con i programmi in esecuzione. Il sistema operativo, quindi, deve essere in grado di **sincronizzare le operazioni** di processi concorrenti e cooperanti.

Affinché ciò sia possibile, è necessario che l'hardware si assicuri che piccole sequenze di istruzioni vengano eseguite in modo **atomico**, ossia senza che esse possano essere interrotte in alcun modo.

La

virtualizzazione della memoria è un'astrazione logica della memoria fisica, dando ad ogni processo l'illusione di una memoria fisica strutturata come uno spazio di indirizzi contiguo, permettendo inoltre l'esecuzione di programmi senza la necessità che essi siano completamente caricati nella vera memoria principale (restando ovviamente caricati completamente nella memoria virtuale).

Design dell'OS e implementazione

Ogni sistema operativo dovrebbe essere progettato secondo una partizione in sottosistemi, ognuno dei quali possiede specifiche attività, input, output e performance.

Ovviamente esistono diversi pro e contro per ogni implementazione e decidere cosa implementare e come farlo è alla base del lavoro del **designer di OS**.

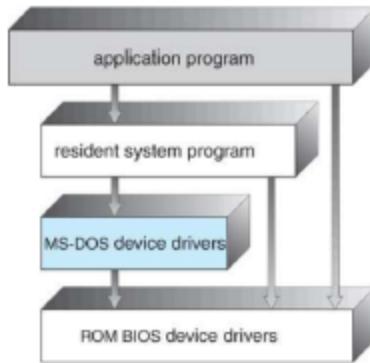
Una decisione importante è scegliere quanto rendere facile l'esperienza all'utente vs. quanto rendere facile l'implementazione al designer.

È fondamentale separare le politiche dai meccanismi.

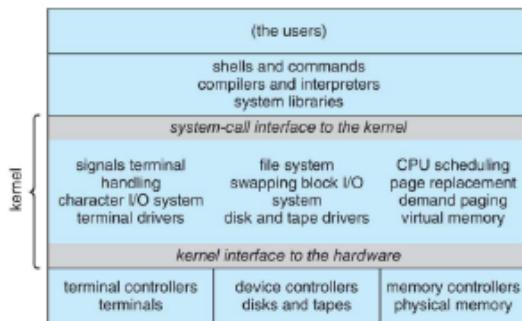
- **Policy:** cosa va fatto
- **Mechanism:** come va fatto

Esempi di strutture di sistemi operativi

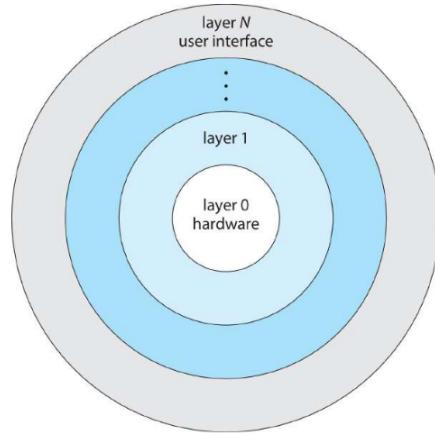
- **Struttura semplice**, dove non vi è alcun sottosistema e non vi è separazione tra kernel e user mode (esempio: il sistema MS-DOS). Semplice da implementare ma estremamente insicuro e poco rigido.



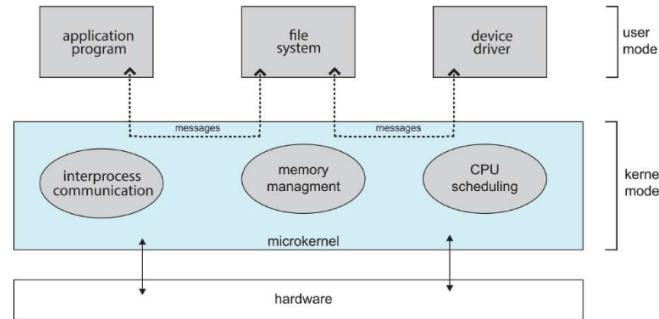
- **Struttura a Kernel Monolitico**, dove l'intero sistema operativo opera in kernel mode e solo i software utente lavorano in user mode (esempio: il sistema UNIX). Semplice da implementare ed efficiente, ma ancora poco sicuro e rigido.



- **Struttura a Livelli**, dove l'OS è suddiviso in N livelli ed ogni livello L usa funzionalità implementate dal livello L – 1 ed espone nuove funzionalità al livello L + 1.
 - Per via della struttura a livelli, il sistema è molto modulare, portabile e semplice da debuggare, rendendo tuttavia più complessa per la comunicazione tra di essi.



- **Struttura a Microkernel**, dove il kernel contiene solo le funzionalità di base, mentre tutte le altre funzionalità dell'OS e i programmi utente vengono eseguiti in user mode. Tale struttura porta ad una maggiore sicurezza, affidabilità ed estensibilità, ma anche ad un'efficienza ridotta.



- **Struttura a Moduli del Kernel caricabili (LKM)**, dove l'OS utilizza dei moduli tramite cui accedere alle funzionalità del kernel.
- **Sistema a Kernel Ibrido**, dove viene utilizzato un approccio intermedio al kernel monolitico e al microkernel, ottenendo i vantaggi di entrambi gli approcci.

Gestione dei processi

Programmi e Processi



Definizione

Un **programma** è un file eseguibile salvato in memoria secondaria. Contiene l'insieme di istruzioni necessarie a svolgere un compito richiesto.

Un

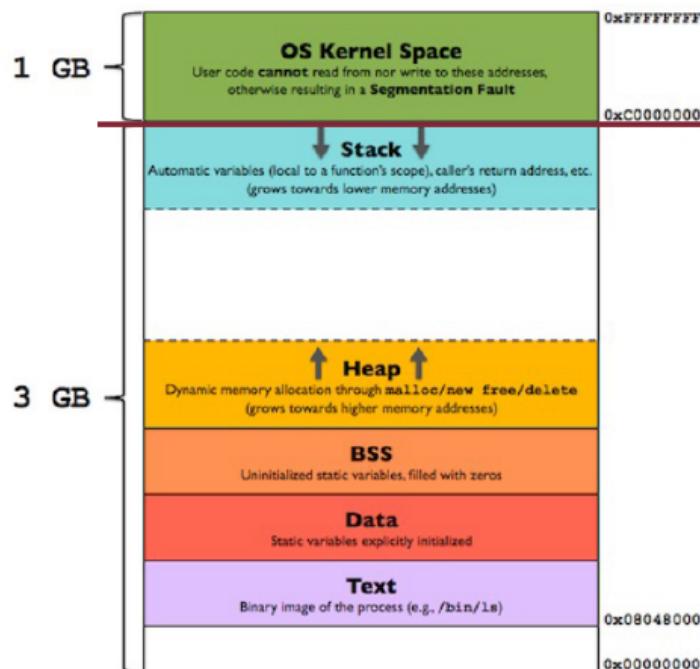
processo è una particolare istanza attiva di un programma caricata in memoria principale, eseguendo sequenzialmente le istruzioni descritte nel programma stesso. Più processi potrebbero corrispondere allo stesso programma, tuttavia ognuno di essi possiede un proprio stato attuale.

Il sistema operativo fornisce la stessa quantità di **virtual address space** ad ogni processo,

la quale è suddivisa in più sezioni:

-
- **Text**, contenente le istruzioni dell'eseguibile
-
- **Data**, contenente i dati statici già inizializzati all'avvio
-
- **BSS**, contenente i dati statici non inizializzati all'avvio
-
- **Stack**, contenente tutti i dati utilizzati da ogni funzione, ossia i vari stack frame
-
- **Heap**, utilizzato per allocazioni dinamiche, ossia di dimensione non fissa
-
- **Spazio libero**, interposto tra stack ed heap, utilizzato da entrambi per "espandersi"

Di seguito vi è un esempio di suddivisione del virtual address space di un'architettura a 32bit:



Esempio:

- Consideriamo il seguente programma:

```

int w = 42;
int x = 0;
float y;

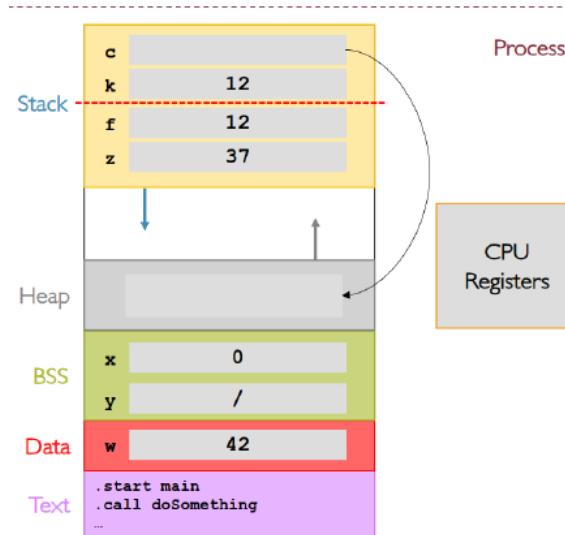
void doSomething(int f) {
    int z = 37;
    z += f;
    ...
}

int main() {
    char* c = malloc(128);
    int k = 12;
    doSomething(k);
    ...
}

```

- La variabile w sarà salvata all'interno della sezione *Data*, poiché essa è statica ed è già definita all'avvio
- Le variabili x e y verranno salvate all'interno della sezione *BSS*, poiché entrambe statiche ma non definite all'avvio (di default, il valore di una variabile di tipo int è già 0, dunque l'assegnamento iniziale inserito verrebbe semplicemente ignorato dal compilatore).
- Le istruzioni presenti nelle funzioni main() e doSomething() vengono inserite nella sezione *Text*.
- Gli stack frame delle funzioni main() e doSomething() vengono salvati nello *Stack*.

In particolare, la variabile char* c è un puntatore salvato nello Stack facente riferimento ad una zona dell'Heap di 128 bit, corrispondente quindi a 16 caratteri.



Stack

Nello Stack metto le variabili di funzioni, nel caso di una funzione ricorsiva se impostato male il caso base e quindi la funzione non viene fermata la memoria

virtuale a disposizione della CPU si esaurisce → Stack Overflow.

La memoria a disposizione dello Stack, inoltre, si espande verso il basso

Lo stack frame per ogni funzione è diviso in 3 parti:

- parametri di funzione + indirizzo di ritorno
- back-pointer allo stack frame precedente
- variabili locali

C'è un problema però:

Il puntatore ESP viene sempre aggiornato man mano che lo stack cresce.

È difficile per il chiamante accedere ai parametri effettivi senza un riferimento fisso nello stack.

Soluzione:

Invece di utilizzare un singolo puntatore alla cima dello stack (**esp**)

Usa un ulteriore puntatore alla base dello stack (**ebp**)

Lascia che **esp** sia libero di cambiare tra diverse chiamate di funzione, mantenendo **ebp** fisso all'interno di ciascun frame dello stack.

Il chiamante salva prima il puntatore alla base del frame corrente dello stack (**%ebp**) nello stack, dopodiché setta il nuovo frame-pointer base al valore corrente di **esp**

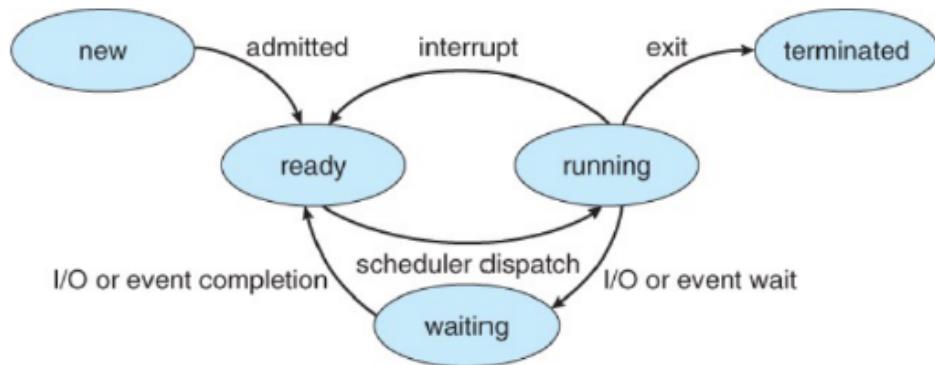
Stato dei processi



Definizione

In qualsiasi momento, uno processo può assumere uno dei seguenti stati:

-
- **New**, ossia il processo è appena stato avviato
-
- **Ready**, il processo è pronto ad essere eseguito ma è in attesa di essere schedulato nella CPU
-
- **Running**, le istruzioni del processo stanno venendo eseguite dalla CPU
-
- **Waiting**, il processo viene sospeso in attesa che una risorsa necessaria sia disponibile o che un evento sia completato o avvenga (ad esempio input da tastiera, accesso al disco, ...)
-
- **Terminated**, il processo ha completato la sua esecuzione e l'OS può eliminarlo



La maggior parte delle syscalls è **bloccante**:

- Dal lato **user space**, il processo chiamante non può svolgere operazioni finché la syscall termina la richiesta

- Dal lato **kernel space**, l'OS imposta il processo corrente nello stato waiting e schedula il primo processo in stato di ready presente nella **state queue**
 - Una volta che la **syscall** termina, il precedente processo bloccato passa in stato di ready, rimanendo in attesa di essere schedulato nuovamente
- Solo il processo che ha effettuato la chiamata viene bloccato, non l'intero sistema operativo

Process Control Block PCB



Il Process Control Block (PCB) è la struttura dati principale utilizzata dall'OS per tenere traccia dello stato e della posizione in memoria di un processo.

All'avvio di un processo, l'OS alloca un nuovo PCB, aggiungendolo alla **state queue**, per poi de-allocarlo una volta che il processo associato è terminato.

Ogni PCB è composto da:

- **Process state**, contenente lo stato del processo
- **Process number**, contenente un'identificatore univoco per il processo
- **Program counter**, Stack Pointer e registri vari relativi al processo in esecuzione
- **Informazioni per lo scheduling**, contenente l'indice di priorità e il puntatore alla state queue
- **Informazioni per la gestione della memoria e contabilità**, contenenti le page table relative al processo e il tempo in user e kernel mode del processo
- **Status I/O**, contenente la lista dei file aperti dal processo

▼ Lezione del 17/10

Creazione dei processi

Processo Padre e Figlio



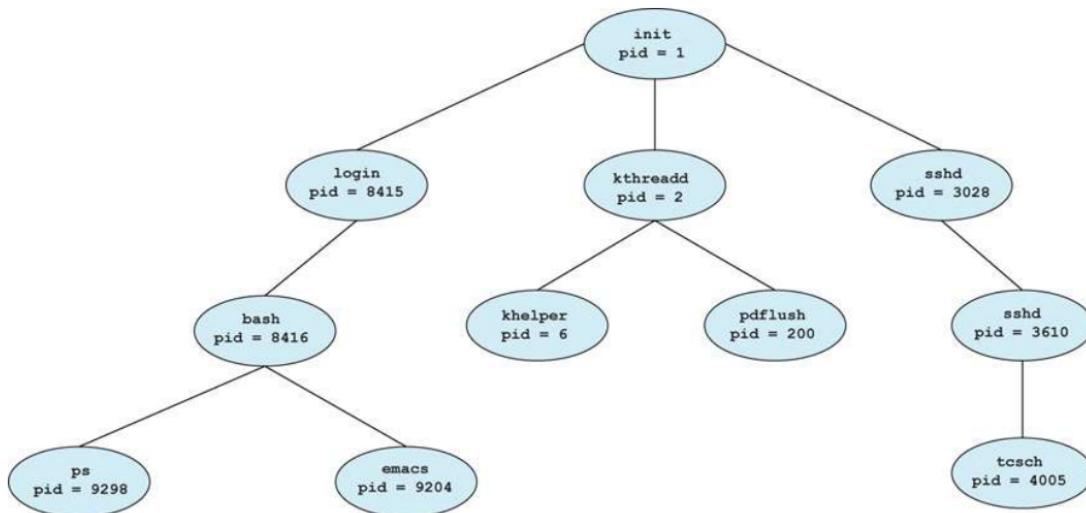
Un processo può creare altri processi tramite specifiche syscalls. Il processo creatore viene detto **processo padre**, mentre il processo creato viene detto **processo figlio**. Ad ogni processo è associato un **process identifier (PID)**, che lo identifica univocamente, ed un **parent process identifier (PPID)**, che identifica univocamente il suo processo padre.

Processo Sched e Init



In un tipico sistema UNIX, il processo scheduler è chiamato *sched* e il suo PID è **0**. La prima operazione eseguita dal processo *sched* all'avvio del sistema operativo è avviare il processo *init*, avente PID pari a **1**. Il processo *init* si occupa di avviare tutti i **processi daemon**, ossia processi eseguiti in background, e i processi relativi ai login degli utenti, diventando quindi l'antenato di tutti i processi che verranno avviati in seguito.

Esempio:



Fork e Spawn dei processi

Per poter creare i processi figli vengono utilizzate le seguenti syscall:

- **fork()** (solo su UNIX), dove il figlio creato è una copia esatta del padre, condividendo con esso le stesse risorse ed ognuno avente il proprio PCB
- **spawn()** (solo su Windows), dove il figlio creato è un processo legato ad un programma diverso da quello del padre e avente uno spazio d'indirizzamento diverso, dunque con istruzioni, dati e PCB diversi dal padre.

Nei sistemi UNIX-like, viene utilizzata la syscall **exec()** a seguito della chiamata fork() per poter ottenere lo stesso effetto della chiamata spawn() di Windows.

In particolare, la syscall

exec() rimpiazza completamente il processo precedente, evitando di riprendere l'esecuzione del precedente una volta completato il processo avviato dalla syscall.

Poiché utilizzando la syscall fork() i due processi condividono lo stesso codice e le stesse risorse, nel processo **padre** essa ritornerà un **PID maggiore di 0**, corrispondente al PID effettivo del figlio, mentre nel processo **figlio**, essa ritornerà **0 come PID**, corrispondente al PID di sched

Una volta creato un processo figlio, il processo padre ha due opzioni:

- Attendere che il processo figlio termini l'esecuzione utilizzando la syscall **wait()**
- Continuare la sua esecuzione in parallelo con il figlio senza essere bloccato

Esempi:

- Esempio 1- Consideriamo il seguente codice:

```

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

int main(){
    pid_t pid;

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

    if (pid < 0) {
        /* if the returned PID is < 0, an error occurred */

        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) {
        /* execute child process code */

        execlp("/bin/ls", "ls", NULL);
    }
    else {
        /* execute parent process code */

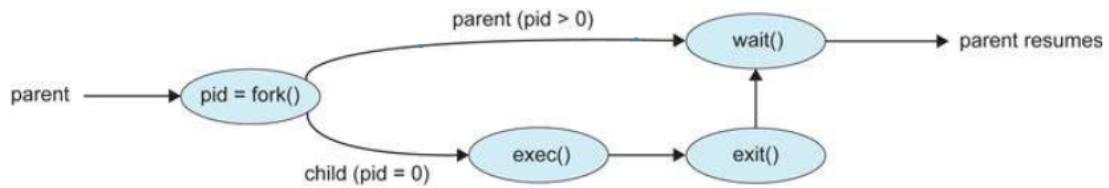
        ...

        /* wait for child to terminate */
        wait(NULL)
        printf("Child terminated");

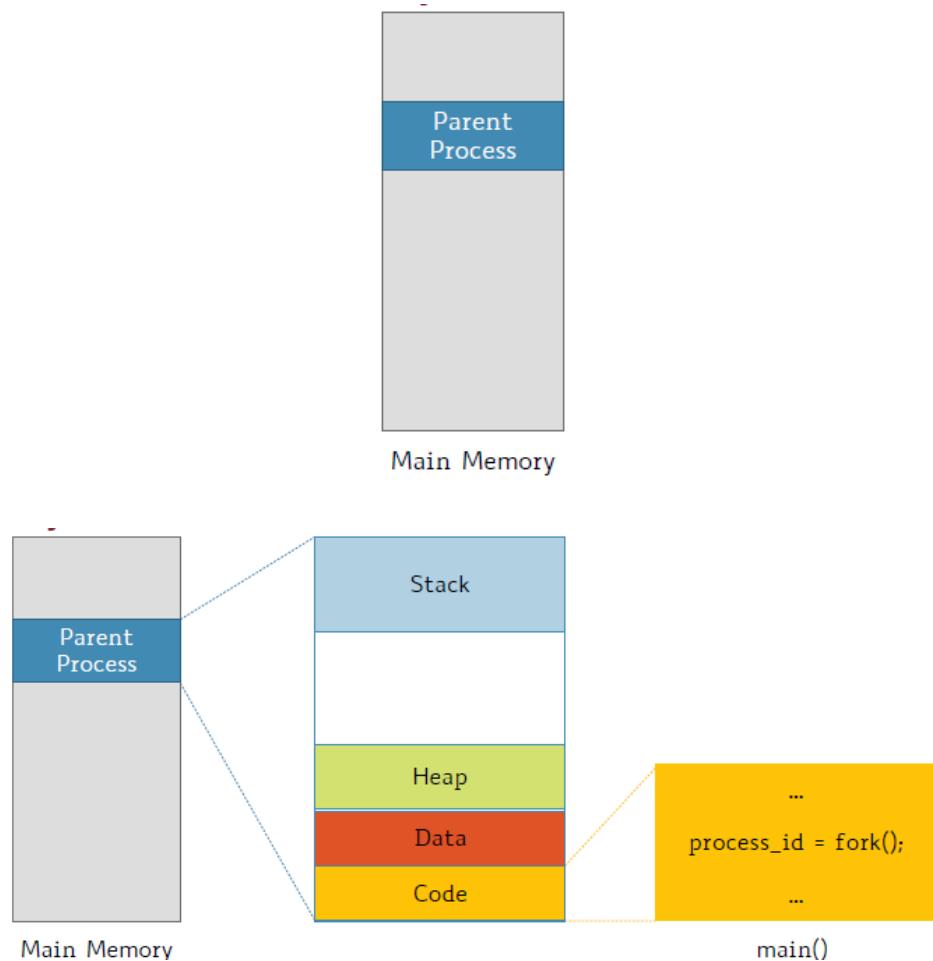
        exit(0);
    }
}

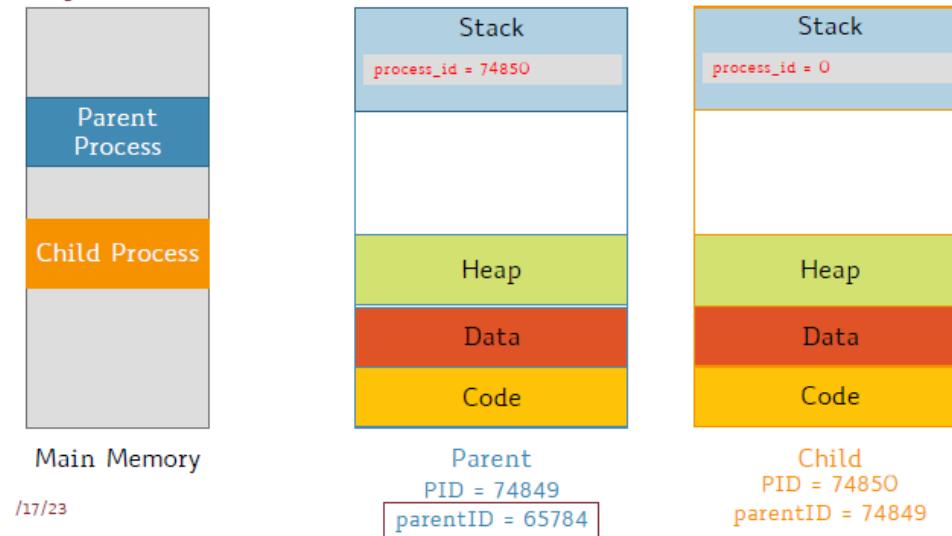
```

L'albero decisionale del programma si sviluppa così:



- Esempio 2:





- Esempio 3- Codice d'esempio:

```

1 #include <iostream>
2 #include <unistd.h>
3
4 using namespace std;
5
6 int main() {
7
8     cout << "Current process ID is: " << getpid() << endl;
9     cout << "\nCurrent parent's process ID is: " << getppid() << endl;
10
11    int pid;
12    pid = fork();
13    // once the fork() system call returns,
14    // both the parent and the child processes will resume from this point onward
15
16    if (pid == 0) // child
17        cout << "\nThis is the child process with process ID = "
18        << getpid() << endl;
19        cout << "\nThis is the child process with parent's process ID = "
20        << getppid() << endl;
21    }
22    else { // parent
23        sleep(1); // to ensure the child process finishes before the parent
24
25        cout << "\nThis is the parent process with process ID = "
26        << getpid() << endl;
27        cout << "\nThis is the parent process with parent's process ID = "
28        << getppid() << endl;
29    }
30
31    return 0;
32 }
```

- Esempio 4- Consideriamo il seguente codice:

```

int pid = fork();

if(pid == 0) {      // A's child (B)
    pid = fork();

    if(pid == 0) {      // B's child (C)
        ...
        execlp(...);
    }
    else { // B
        ...
    }
}
else { // A
    ...
}

```

La gerarchia dei processi corrisponde a:



- Esempio 5- Consideriamo il seguente codice:

```

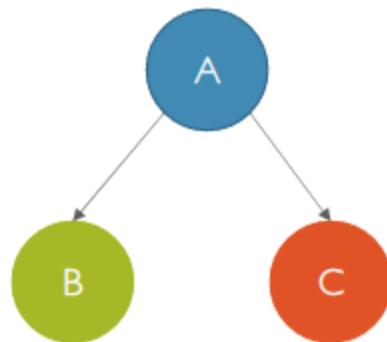
int pid = fork();

if(pid == 0) {      // A's child (B)
    ...
    execlp(...);
}
else { // A
    pid = fork();

    if(pid == 0) {      // A's child (C)
        ...
        execlp(...);
    }
}

```

La gerarchia dei processi generata corrisponde a:



- Esempio 6- Consideriamo il seguente codice:

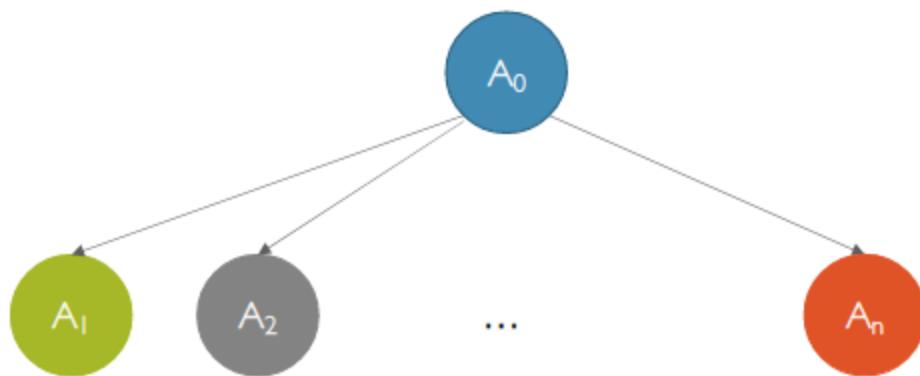
```

for(int i=0;i<n;i++) {
    if(fork() == 0) {           // A0's child
        ...
        execlp(...);
    }
}

// back in the parent A0
// wait for all children to terminate
for(int i=0;i<n;i++) {
    wait(NULL);
}

```

La gerarchia dei processi generata corrisponde a:



Recap System Calls

- **Fork:** avvia un nuovo processo figlio come copia esatta del genitore.
- **Execlp:** sostituisce il programma del processo corrente con il programma specificato come input.
- **Sleep:** sospende l'esecuzione per un determinato numero di secondi.
- **Wait/Waitpid:** attende che un qualsiasi processo/un processo specifico completi l'esecuzione.

Terminazione dei processi

Ogni processo può richiedere la sua **terminazione immediata** eseguendo la syscall `exit()`, la quale, tipicamente, ritorna un intero. L'intero ritornato viene passato al processo padre nel caso in cui esso stia eseguendo la syscall `wait()`.

I processi possono anche essere terminati immediatamente dal sistema operativo stesso, ad esempio nel caso in cui sia necessario liberare risorse o nel caso in cui venga eseguito un comando `kill`. Inoltre, un padre potrebbe uccidere un suo figlio se il compito ad esso assegnato non è più necessario.

Quando un processo termina, tutte le sue **risorse** di sistema vengono **liberate** (i dati vengono de-allocate, i file aperti vengono chiusi, ...) e viene ritornato lo **stato di processo terminato**, il **tempo di esecuzione** e un **codice di esito dell'esecuzione** (solitamente, viene ritornato 0 se l'esecuzione viene completata senza problemi, altrimenti viene ritornato un valore diverso da 0), i quali verranno passati al processo padre nel caso in cui esso stia eseguendo la syscall `wait()`

Processo Orfano e processo Zombie



Se un processo padre esegue la syscall `exit()` e un suo figlio non è ancora terminato, quest'ultimo viene detto **processo orfano**, venendo solitamente ereditato da `init`, il quale poi si occuperà di ucciderlo tramite `kill`. Se invece un processo figlio tenta di terminare ma il suo processo padre non sta eseguendo la syscall `wait()`, allora tale processo viene detto **processo zombie**, il quale eventualmente verrà ereditato da `init` per poi essere ucciso.

Scheduling dei processi

Process State Queue



Per poter gestire tutti i PCB dei vari processi attivi, il sistema operativo è dotato di **queue**, ossia code all'interno delle quali essi vengono inseriti. In particolare, vengono utilizzate **5 code per gestire gli stati dei processi** (una per ogni stato) e **una coda per ogni dispositivo I/O**.

Quando l'OS modifica lo stato di un processo, il PCB associato viene spostato da una coda all'altra, a seconda delle politiche di gestione dell'OS stesso.

La quantità di PCB che possono essere inseriti all'interno della **running queue** è strettamente legato al numero di core utilizzabili dall'OS, mentre per tutte le altre queue non vi è alcuna restrizione.

Schedulers

1. Un **scheduler a lungo termine** funziona raramente ed è tipico di un sistema batch o di un sistema molto gravemente sovraccaricato."
2. Uno **scheduler a breve termine** funziona molto frequentemente (circa ogni 100 millisecondi) e deve scambiare rapidamente un processo fuori dalla CPU e sostituirlo con un altro.
3. Quando i carichi di sistema diventano elevati, uno **scheduler a medio termine** consente ai lavori più piccoli e più veloci di completarsi rapidamente e di liberare il sistema.
4. Un sistema di pianificazione efficiente selezionerà una buona combinazione di processi legati alla CPU e **processi legati all'I/O**.

Context Switch

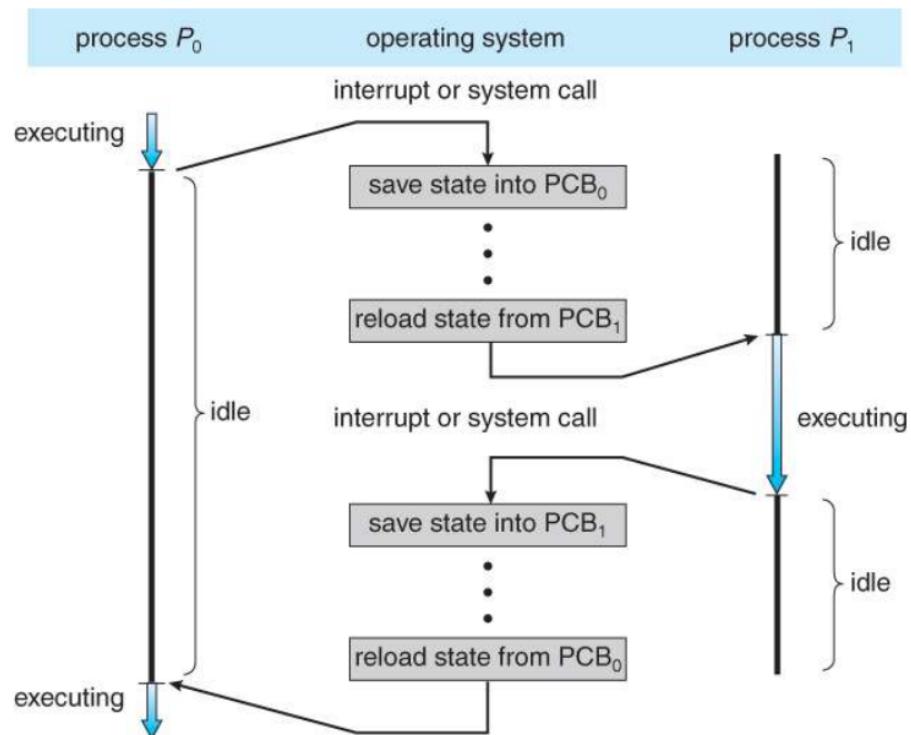


Il **context switch** è la procedura utilizzata dalla CPU per sospendere il processo attualmente in esecuzione per poter eseguirne un altro in stato di ready.

Il context switch viene eseguito

quando viene attivata una trap di sistema, dunque quando viene eseguita una syscall, viene gestita un'eccezione o eseguito un interrupt hardware, oppure una volta **superato il time slice**, ossia la massima quantità di tempo attesa dalla CPU prima di eseguire un context switch.

Poiché per passare da un processo all'altro è necessario salvare lo stato attuale del processo nel suo PCB per poi caricare lo stato salvato nel PCB del processo da eseguire, l'operazione di context switch risulta essere un'operazione molto **costosa**. Un time slice minore implica una reattività massimizzata, sottraendo tuttavia tempo all'esecuzione dei processi per via del tempo impiegato per completare un context switch, richiedendo quindi un **compromesso** tra i due



Comunicazione tra processi

Processi indipendenti e cooperanti



Un processo può essere **indipendente**, ossia ininfluente e non influenzato da altri processi, oppure **cooperante**, ossia influente o influenzato da altri al fine di svolgere un compito comune. I processi cooperanti possono comunicare tra loro tramite una zona di **memoria condivisa** oppure tramite lo **scambio di messaggi**.

Memoria Condivisa

- Più veloce una volta inizializzata, poiché non richiede syscall
- Più complessa da inizializzare e da usare tra processi distribuiti su più computer
- Preferibile per lo scambio di grosse quantità di informazioni tra processi sullo stesso computer
- La memoria da condividere è inizialmente all'interno dell'address space di un particolare processo, richiedendo di eseguire alcune syscall per poter rendere accessibile la memoria ad altri processi, i quali eseguiranno delle syscall per poter connettere tale memoria condivisa al loro spazio.

Scambio di messaggi

- Più lento poiché richiedente una syscall per ogni scambio
- Più semplice da inizializzare e da usare tra processi su più computer
- Preferibile quando la quantità e la frequenza delle informazioni da trasferire sono basse o quando più computer sono coinvolti

- Affinché sia utilizzabile, è necessario che l'OS sia provvisto di syscall per poter permettere ai processi di scambiare e ricevere messaggi, richiedendo che venga preventivamente stabilito un canale di comunicazione tra i due processi cooperanti

Nello scambio di messaggi è necessario:

- Scegliere l'utilizzo di una **comunicazione diretta**, dove il mittente deve sapere preventivamente il nome del processo destinatario, il quale a sua volta necessita di sapere il nome del mittente, oppure una **comunicazione indiretta**, tramite l'uso di mailbox o porte.
- Scegliere se utilizzare o meno una queue per i messaggi:
 - **Zero capacity**, dove i messaggi non possono essere salvati in una queue, dunque il mittente rimane bloccato finché il destinatario non riceve il messaggio
 - **Bounded capacity**, dove vi è una queue di capienza predefinita, permettendo ai mittenti di non bloccarsi a meno che la queue non sia piena
 - **Unbounded capacity**, dove la queue ha teoricamente capienza infinita, implicando che i mittenti non debbano mai bloccarsi

▼ Lezione del 19/10

Scheduler della CPU

CPU burst e I/O burst



Definiamo come **CPU burst** lo stato in cui un processo è eseguito dalla CPU, mentre definiamo come **I/O burst** lo stato in cui un processo è in attesa che i dati vengano trasferiti dentro o fuori dal sistema.
In generale, ogni processo alterna costantemente tra CPU burst e I/O burst.

Scheduler della CPU



Lo **scheduler della CPU** è un processo che, a seconda di una policy di scheduling stabilita, si occupa di selezionare un processo dalla ready queue da eseguire in ogni momento in cui la CPU è inattiva.
La struttura dati utilizzata per la ready queue e l'algoritmo usato per selezionare il processo successivo è basato su una queue FIFO (First In, First Out)

Esistono 2 tipi di Scheduling:

- Scheduling a **lungo termine**: Come il sistema operativo determina il livello di multiprogrammazione.
- Scheduling a **breve termine**: Come il sistema operativo seleziona un processo da eseguire dalla coda dei processi pronti

Le **decisioni di scheduling della CPU** vengono prese in una delle seguenti **quattro** condizioni:

1. Quando un processo passa **dal running state al waiting state** (ad esempio quando viene effettuata una richiesta I/O oppure viene invocata la syscall wait())
2. Quando un processo passa **dal running state al ready state** (ad esempio in risposta ad un interrupt)

3. Quando un processo passa **dal waiting state al ready state** (ad esempio quando viene completata una richiesta I/O oppure viene terminata una syscall wait())
4. Quando un processo viene **creato o terminato**

Scheduling Preemptive e Non-Preemptive



Uno scheduling viene detto **non-preemptive (non-preventivo)** se lo scheduler non ha alcuna scelta se non il selezionare un nuovo processo (ad esempio le condizioni 1 e 4).

Una volta che un processo viene avviato, esso continua ad essere eseguito finché esso non decide volontariamente di bloccarsi oppure finché esso non termina.

Uno scheduling viene detto

preemptive (preventivo) se lo scheduler deve scegliere se continuare ad eseguire il processo attuale oppure selezionarne uno nuovo (ad esempio le condizioni 2 e 3).

Attenzione: uno scheduler preemptive viene comunque attivato dalle condizioni 1 e 4, poiché in tali casi deve comunque essere selezionato un processo.

Lo scheduling preemptive potrebbe causare **problematiche** nel caso in cui esso avvenga mentre il kernel è occupato ad effettuare una syscall (ad esempio l'aggiornamento di strutture dati critiche per il kernel) o nel caso in cui due processi condividono dati (uno dei due potrebbe essere interrotto nel mezzo dell'aggiornamento di una struttura dati condivisa tra i due).

Le contromisure applicabili per prevenire tali problematiche sono:

- Far aspettare il processo finché la syscall del kernel non viene completata o bloccata

prima di procedere con lo scheduling.

- Disabilitare gli interrupt prima di entrare nella sezione di codice critica, per poi riabilitarle subito dopo.

Dispatcher



Il **dispatcher** corrisponde al modulo che cede il controllo della CPU al processo selezionato dallo scheduler.

Le sue funzioni includono l'esecuzione del **context switch**, l'esecuzione del **passaggio alla user mode** e il saltare alla posizione corretta nel programma appena caricato.

Tempistiche dello scheduling



- **Arrival time:** l'istante di arrivo del processo nella ready queue
-
- **Start time:** l'istante in cui la CPU esegue la prima istruzione di un processo
-
- **Completion time:** l'istante in cui il processo completa la sua esecuzione
-
- **Burst time:** il tempo richiesto da un processo per l'esecuzione della CPU
-
- **Turnaround time:** il tempo trascorso tra il completion time e l'arrival time
($T_{turnaround} = T_{completion} - T_{arrival}$)
-
- **Waiting time:** la differenza di tempo tra il turnaround time e il burst time
($T_{waiting} = T_{turnaround} - T_{burst}$)

Attenzione: il tempo di attesa per l'I/O non viene considerato, poiché i processi in waiting state non sono sotto il controllo dello scheduler della CPU, dunque esso dovrà essere sottratto dal waiting time

Durante la scelta dell'algoritmo di scheduling è necessario considerare più criteri, in particolare:

- La **massimizzazione dell'utilizzo della CPU**, ossia la percentuale di tempo in cui la CPU è occupata (idealmente 100%, su sistemi concreti è sufficiente che sia tra il 40% e il 90%).
- La **massimizzazione del throughput**, ossia il numero di processi completati in una determinata unità di tempo.
- La **minimizzazione del turnaround time**
- La **minimizzazione del waiting time**
- La **minimizzazione del response time**, ossia il tempo impiegato dall'emissione del comando all'inizio della risposta alla richiesta

Per le politiche di scheduling assumiamo che:

- Vi è un singolo processo per utente
- I processi sono indipendenti tra di loro, dunque non vi è comunicazione
- Ogni processo è costituito da un singolo thread

Job, Processo e Thread



Un **job** è l'unità generica di esecuzione della CPU, differenziandosi in **processi** e **thread**, ossia una sotto-istanza di un processo facente parte di una sua suddivisione in più thread

Algoritmi di Scheduling

Metodo - First Come First Served (FCFS)



L'algoritmo **First Come First Served (FCFS)** è un algoritmo di scheduling **nonpreemptive** dove, appena creati, i job vengono inseriti in una queue FIFO e lo scheduler esegue il primo job della queue fino al suo completamento, per poi procedere con il job successivo.
Lo scheduler prende il controllo della CPU solo nel caso in cui il processo in esecuzione richieda un'operazione di I/O oppure termina la sua esecuzione.

Pro:

-Estremamente semplice da implementare

Contro:

-
Un job potrebbe essere eseguito indefinitamente (ad esempio finché esso non si blocca)

-
Il waiting time medio è molto variabile poiché job con pochi CPU burst potrebbero essere in coda dopo job con molti CPU burst

-
Può crearsi un **effetto convoglio**, dovuto alla sovrapposizione tra CPU e I/O
poiché i processi strettamente legati alla CPU forzeranno i processi strettamente legati all'I/O ad aspettare

Esempi:

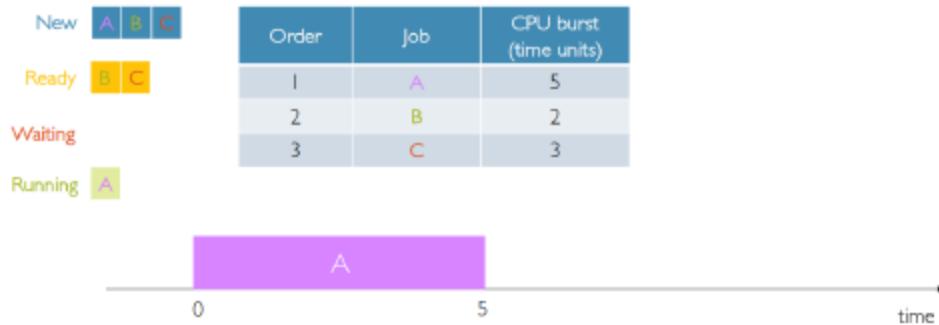
1. • Consideriamo la seguente coda di processi:

| New | A | B | C |
|-------|-----|---------------------------|---|
| Order | job | CPU burst (time units) | |
| 1 | A | 5 | |
| 2 | B | 2 | |
| 3 | C | 3 | |

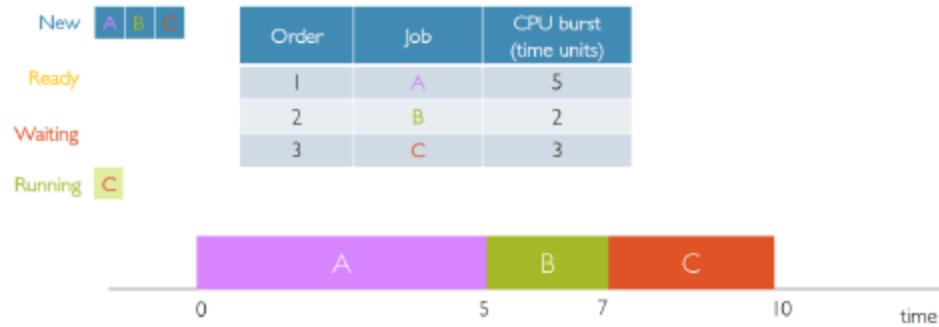
- I processi vengono creati allo stesso istante, dunque l'**arrival time** per tutti i processi è l'istante 0. Successivamente, tutti e tre i processi vengono messi nella ready queue

| New | A | B | C |
|---------|---|---|---|
| Ready | A | B | C |
| Waiting | | | |
| Running | | | |

- Di seguito, lo scheduling FCFS seleziona il primo processo della ready queue, ossia il processo A, spostandolo in stato di ready e completando la sua esecuzione



- Una volta terminato il processo A, lo scheduler ripeterà le stesse operazioni fino a che tutti i processi non saranno terminati



- Una volta terminati tutti i processi, calcoliamo il waiting time medio:

– Per il processo A si ha:

$$T_{\text{waiting}} = T_{\text{turnaround}} - T_{\text{burst}} = T_{\text{completion}} - T_{\text{arrival}} - T_{\text{burst}} = 5 - 0 - 5 = 0$$

– Per il processo B si ha:

$$T_{\text{waiting}} = T_{\text{turnaround}} - T_{\text{burst}} = T_{\text{completion}} - T_{\text{arrival}} - T_{\text{burst}} = 7 - 0 - 2 = 5$$

– Per il processo C si ha:

$$T_{\text{waiting}} = T_{\text{turnaround}} - T_{\text{burst}} = T_{\text{completion}} - T_{\text{arrival}} - T_{\text{burst}} = 10 - 0 - 3 = 7$$

– Il waiting time medio, quindi, sarà:

$$\overline{T}_{\text{waiting}} = \frac{1}{n} \sum_{i=0}^n T_i^{\text{waiting}} = \frac{0 + 5 + 7}{3} = 4$$

2. • Nel seguente scenario, utilizzando sempre uno scheduling FCFS, si ha:

New [B | C | A]

| Order | Job | CPU burst (time units) |
|-------|-----|---------------------------|
| 1 | B | 2 |
| 2 | C | 3 |
| 3 | A | 5 |



- Il waiting time medio, quindi, sarà:

$$\bar{T}_{\text{waiting}} = \frac{1}{n} \sum_{i=0}^n T_i^{\text{waiting}} = \frac{5 + 0 + 2}{3} \approx 2.3$$

3. • Consideriamo il seguente scenario:

New [A | B | C]

Ready [A | B | C]

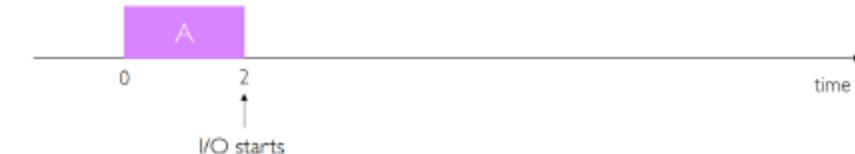
Waiting

Running [A]

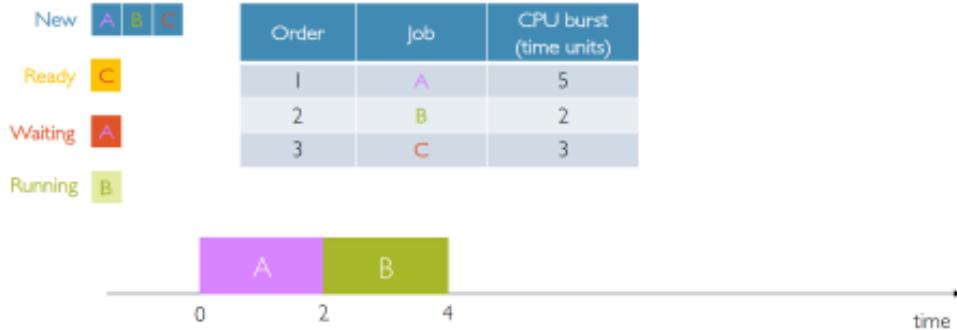
| Order | Job | CPU burst (time units) |
|-------|-----|---------------------------|
| 1 | A | 5 |
| 2 | B | 2 |
| 3 | C | 3 |

- Supponiamo che dopo 2 unità temporali il processo A esegua una richiesta I/O, entrando quindi in stato di waiting, la quale verrà completata dopo un istante.

Attenzione: poiché il tempo in attesa I/O non viene calcolato nel waiting time, sarà necessario sottrarre un istante dal waiting time del processo A.

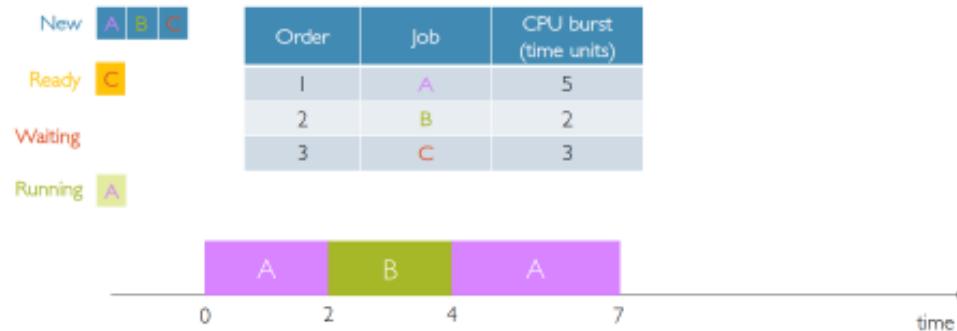


- Una volta entrato in stato di waiting, lo scheduler selezionerà il processo seguente nella coda, ossia il processo B, portandolo a termine

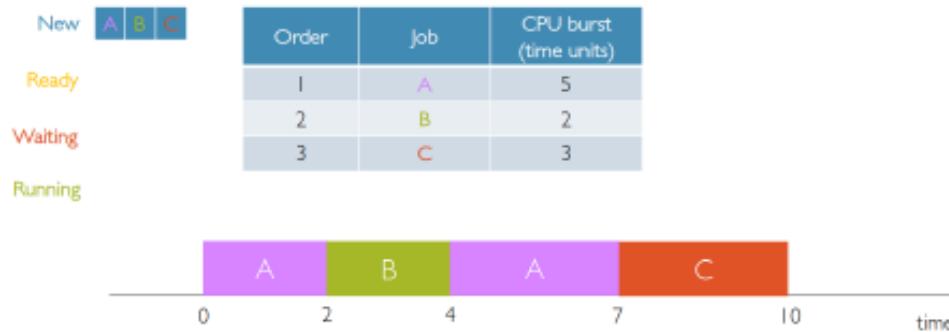


- Una volta terminata l'esecuzione del processo B, il processo A verrà selezionato per riprendere l'esecuzione.

Attenzione: viene selezionato il processo A e non il processo C poiché l'algoritmo FCFS utilizza una queue basata sull'arrival time dei processi



- Infine, il waiting time medio sarà:



$$\bar{T}_{\text{waiting}} = \frac{1}{n} \sum_{i=0}^n T_i^{\text{waiting}} = \frac{1+2+7}{3} \approx 3.3$$

Metodo - Round Robin (RR)



L'algoritmo **Round Robin (RR)** è un algoritmo di scheduling **preemptive** dove viene selezionato **il primo processo presente in ready queue**, con l'aggiunta di un timer che funga da limite al CPU burst dei job, detto time slice (o time quantum):

- Ogni volta che un job prende controllo della CPU, il time slice viene resettato
 - Se il job termina prima che il time slice scada, allora lo scheduler selezionerà il primo processo in ready queue
 - Se il time slice scade prima che il job termini, allora lo scheduler sposterà il job in esecuzione alla fine della ready queue, per poi selezionare il primo processo presente in tale queue

Per via della natura del RR, la ready queue viene gestita come una coda circolare, distribuendo il tempo di utilizzo della CPU equamente tra tutti i processi.

Pro:

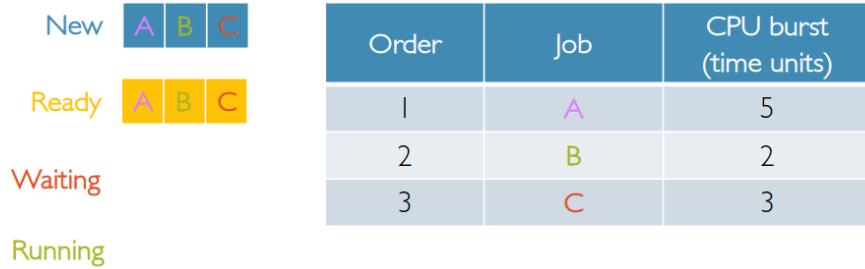
- Più equo rispetto all'FCFS, poiché viene fornita la stessa quantità di tempo di utilizzo della CPU ad ogni job

Contro:

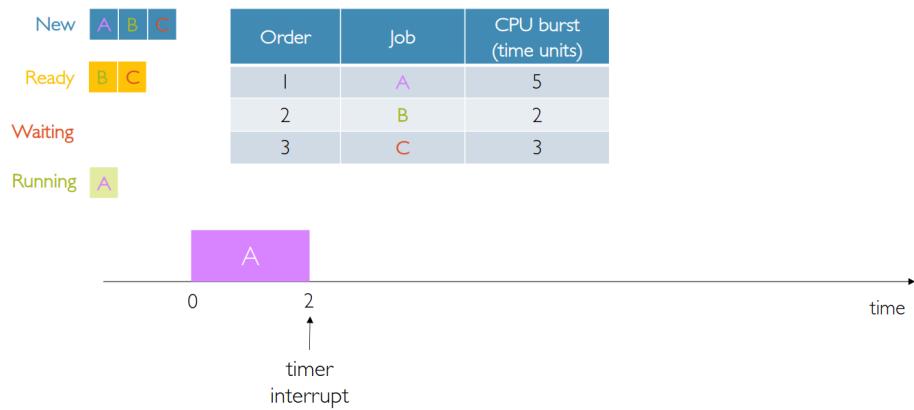
- Un time slice troppo ampio rischia di far degenerare un RR in un FCFS
- Un time slice troppo stretto rischia di generare un elevato numero di context switch, diminuendo l'uso effettivo della CPU

Esempio:

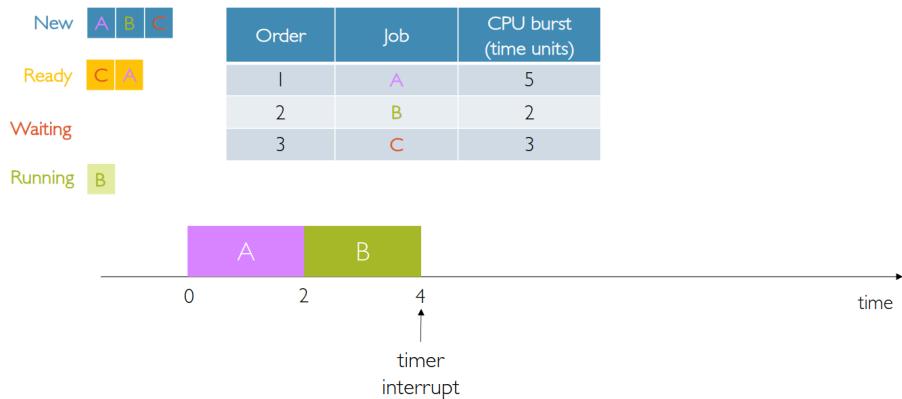
- Consideriamo la seguente coda dei processi utilizzando un algoritmo RR con un time slice di 2 e un context switch trascurabile:



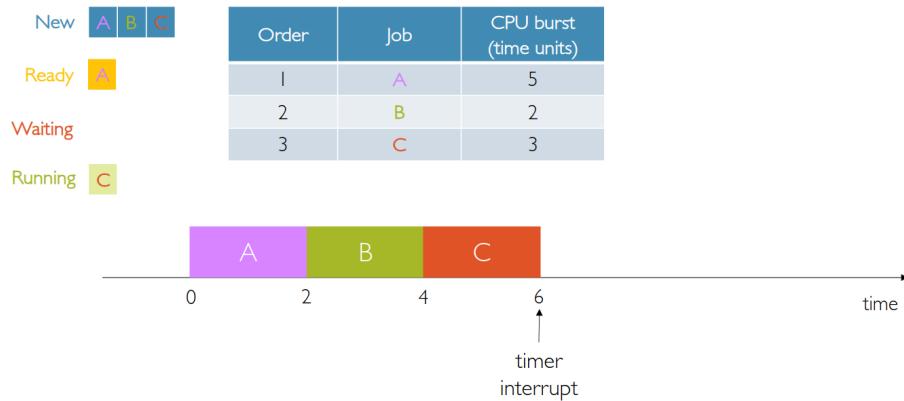
- Il primo processo a prendere il controllo della CPU è il processo A, venendo bloccato ed aggiunto alla ready queue dopo 2 unità temporali per via del time slice



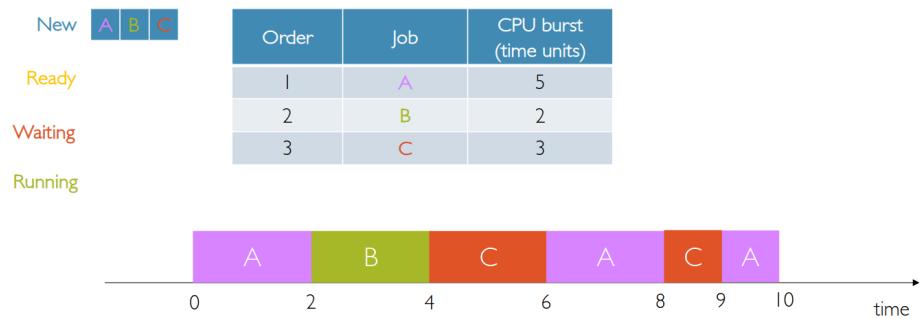
- Successivamente, il processo B prende controllo della CPU, venendo anch'esso bloccato dopo 2 unità temporali. In questo caso, tuttavia, il processo B non verrà aggiunto alla fine della ready queue poiché la sua esecuzione è terminata



- Di seguito, il processo C prenderà il controllo, venendo bloccato dopo 2 tempi



- Infine, vengono eseguiti i processi A e C finché essi non verranno completati



- Il waiting time medio, quindi, sarà:

$$\bar{T}_{\text{waiting}} = \frac{1}{n} \sum_{i=0}^n T_i^{\text{waiting}} = \frac{5 + 2 + 6}{3} \approx 4.3$$

FCFS vs RR

Confrontando il turnaround time e il waiting time medio tra uno scheduler FCFS e uno scheduler RR, il primo sembra essere a primo occhio più performante del secondo.

Tuttavia, considerando la varianza tra i waiting time di ogni processo, notiamo come il RR risulta essere

più equo, fornendo ad ogni processo la stessa quantità di tempo di utilizzo della CPU.

| Job | CPU burst | turnaround time | | waiting time | |
|------|-----------|-----------------|-----|--------------|-----|
| | | FCFS | RR | FCFS | RR |
| A | 100 | 100 | 496 | 0 | 396 |
| B | 100 | 200 | 497 | 100 | 397 |
| C | 100 | 300 | 498 | 200 | 398 |
| D | 100 | 400 | 499 | 300 | 399 |
| E | 100 | 500 | 500 | 400 | 400 |
| Avg. | | 300 | 498 | 200 | 398 |

Metodo - Shortest Job First (SJF)



L'algoritmo **Shorted Job First (SJF)** è un algoritmo di scheduling basato sull'esecuzione prioritaria del job con la **quantità minore stimata di CPU burst**.

L'algoritmo SJF può essere:

- **non-preemptive**, dove ogni job che ottiene il controllo della CPU viene eseguito fino al suo completamento
- **preemptive**, dove ogni volta che un **nuovo job** arriva nella ready queue e il suo CPU burst stimato è minore di quello rimanente del job attualmente in esecuzione, tale job prende controllo della CPU.
Questa versione del SJF viene anche detta
Shortest Remaining Time First (SRTF)

Pro:

- Ottimale nel caso in cui l'obiettivo sia minimizzare il waiting time medio
- Funzionale con scheduler sia non-preemptive sia preemptive

Contro:

- È quasi impossibile stimare esattamente il tempo di esecuzione della CPU di un job
 - I processi strettamente legati alla CPU attivi da molto tempo potrebbero andare in *starvation*

Esempi:

- 1) Consideriamo la seguente coda di processi gestita da uno scheduler SJF nonpreemptive. In tal caso, si ha che:

| Job | CPU burst (time units) |
|-----|---------------------------|
| A | 6 |
| B | 8 |
| C | 7 |
| D | 3 |



$$T_{avg. waiting} = \frac{3 + 16 + 9 + 0}{4} = 7$$

- 2) Consideriamo la seguente coda di processi gestita da uno scheduler SJF preemptive,
ossia uno scheduler SRTF

| Job | Arrival time | CPU burst (time units) |
|-----|--------------|---------------------------|
| A | 0 | 8 |
| B | 1 | 4 |
| C | 2 | 9 |
| D | 3 | 5 |

0 →

- Il primo job ad essere eseguito è il processo A, poiché il primo ad essere creato ed inserito nella ready queue.

| Job | Arrival time | CPU burst (time units) |
|-----|--------------|---------------------------|
| A | 0 | 8 |
| B | 1 | 4 |
| C | 2 | 9 |
| D | 3 | 5 |



- Una volta che il processo B viene creato, il suo CPU burst stimato, ossia 4, viene comparato con il CPU burst rimanente del processo A, ossia 7. Poiché $4 < 7$, allora il processo B prende controllo della CPU.

Analogamente, nell'istante in cui il processo C viene creato, il suo CPU burst stimato, ossia 9, viene comparato con quello rimanente del processo A, ossia 6, e quello del processo B, ossia 3. Siccome $3 < 7 < 9$, allora il processo B mantiene il controllo della CPU.

Lo stesso ragionamento viene effettuato anche dopo la creazione del processo D, dove il processo B mantiene il controllo ($2 < 7 < 5 < 9$). Di conseguenza, il processo B verrà eseguito fino al suo completamento.

| Job | Arrival time | CPU burst (time units) |
|-----|--------------|---------------------------|
| A | 0 | 8 |
| B | 1 | 4 |
| C | 2 | 9 |
| D | 3 | 5 |



- Una volta completato il processo B, verrà eseguito il processo avente il CPU burst stimato minore:
 - Il processo A ha un CPU burst rimanente pari a 7
 - Il processo C ha un CPU burst rimanente pari a 9
 - Il processo D ha un CPU burst rimanente pari a 5

Di conseguenza, verrà eseguito il processo D fino al suo completamento

| Job | Arrival time | CPU burst (time units) |
|-----|--------------|---------------------------|
| A | 0 | 8 |
| B | 1 | 4 |
| C | 2 | 9 |
| D | 3 | 5 |



- Analogamente al processo D, verranno eseguiti il processo A e il processo C, ognuno di essi fino al loro completamento (poiché, nel frattempo, nessun altro processo viene inserito nella ready queue, dunque non viene mai attivato l'algoritmo di scheduling)

| Job | Arrival time | CPU burst (time units) |
|-----|--------------|---------------------------|
| A | 0 | 8 |
| B | 1 | 4 |
| C | 2 | 9 |
| D | 3 | 5 |



$$T_{avg. waiting} = \frac{(17 - 0 - 8) + (5 - 1 - 4) + (26 - 2 - 9) + (10 - 3 - 5)}{4} = 6.5$$

Confronto tra FCFS, RR e SJF

| Job | CPU burst | turnaround time | | | waiting time | | |
|------|-----------|-----------------|-----|-----|--------------|-----|-----|
| | | FCFS | RR | SJF | FCFS | RR | SJF |
| A | 50 | 50 | 150 | 150 | 0 | 100 | 100 |
| B | 40 | 90 | 140 | 100 | 50 | 100 | 60 |
| C | 30 | 120 | 120 | 60 | 90 | 90 | 30 |
| D | 20 | 140 | 90 | 30 | 120 | 70 | 10 |
| E | 10 | 150 | 50 | 10 | 140 | 40 | 0 |
| Avg. | | 110 | 110 | 70 | 80 | 80 | 40 |

Metodo - Priority Scheduling



Gli algoritmi basati sul **priority scheduling** sono algoritmi di scheduling dove ad ogni job viene assegnato valore, il quale determina la sua priorità. Il job con priorità maggiore viene schedulato per primo. Le priorità possono essere assegnate in due modi:

-

Internamente, ossia assegnate dall'OS in base a determinati criteri (ad esempio

il CPU burst medio, il rateo di attività tra CPU e I/O, risorse utilizzate, ...)

-

Esteriormente, ossia assegnate dall'utente in base all'importanza del job o di altri criteri

Il priority scheduling può essere sia **non-preemptive** sia **preemptive**.

Contro:

-

Starvation (o bloccaggio infinito), dove un job di bassa priorità rimane in attesa perenne poiché altri job hanno sempre una priorità maggiore. Tali job potrebbero essere eseguiti eventualmente quando il carico del sistema è minore o dopo che il sistema stesso vada in crash, venga spento o venga riavviato

Come contromisura alla starvation viene utilizzato

L'aging, ossia l'incremento della priorità di un job in base al suo tempo in attesa, finché essi non verranno eventualmente schedulati

L'algoritmo SJF è un algoritmo con **priority scheduling** dove la priorità è dettata dal CPU burst minore.

Metodo - Multilevel Queue (MLQ)



L'algoritmo **Multilevel Queue (MLQ)** è un algoritmo di scheduling basato sull'utilizzo di queue multiple separate tra loro, ognuna per ogni **categoria** di job, dove ogni queue utilizza l'algoritmo di scheduling più appropriato per una determinata categoria di job.

Una volta inserito all'interno di una queue, **nessun job può essere spostato in un'altra queue**.

Le due opzioni più comuni per l'implementazione di tale algoritmo sono:

- **String priority**, dove nessun job all'interno di una queue di priorità più bassa viene eseguito finché esiste almeno un job delle queue di priorità più alta
- **Round robin**, dove ogni queue utilizza un proprio time slice, il quale aumenta esponenzialmente al diminuire della priorità della coda

Metodo - Multilevel Feedback Queue (MLFQ)



L'algoritmo **Multilevel Feedback Queue (MLFQ)** segue la stessa idea dell'algoritmo MLQ, con l'aggiunta della **possibilità per ogni job di essere spostato da una queue all'altra**.

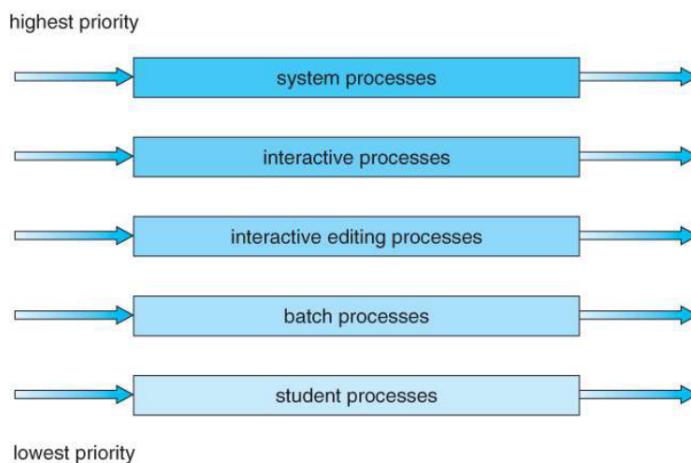
Lo spostamento di un job può rivelarsi necessario quando:

- Un job passa dall'utilizzare molto la CPU all'utilizzare molto l'I/O e viceversa
- Un job è in stato di starving, venendo spostato per breve tempo in una queue a priorità più alta

Per gestire gli spostamenti, l'algoritmo utilizza le seguenti regole:

- Inizialmente ad ogni job viene assegnata la **priorità più alta**
- Se il time slice di un job **scade**, allora quest'ultimo viene spostato nella queue con un livello di priorità **inferiore** rispetto a quella precedente
- Se il time slice di un job **non scade**, allora quest'ultimo viene spostato nella queue con un livello di priorità **superiore** rispetto a quella precedente
- La priorità di processi strettamente legati alla CPU viene diminuita rapidamente
- La priorità di processi strettamente legati all'I/O rimarrà alta

Esempio di suddivisione tramite MLQ e MLFQ:



Esempi:

1. • Consideriamo la seguente coda di processi gestita da un algoritmo MLFQ con 3 queue e strict priority.

New [A | B | C]

| Order | Job | CPU burst (time units) |
|-------|-----|---------------------------|
| 1 | A | 30 |
| 2 | B | 20 |
| 3 | C | 10 |

| Queue | Time Slice (time units) | Jobs |
|-------|-------------------------|---|
| 1 | 1 | A ¹ ₁ , B ¹ ₂ , C ¹ ₃ |
| 2 | 2 | |
| 3 | 4 | |

dove indichiamo l'avanzamento di ogni processo all'avanzare del tempo con la notazione $JOB_{total\ time\ elapsed}^{execution\ time}$

- Poiché il time slice per tutti e tre i processi è scaduto, ognuno di essi viene spostato nella queue a priorità inferiore

New [A | B | C]

| Order | Job | CPU burst (time units) |
|-------|-----|---------------------------|
| 1 | A | 30 |
| 2 | B | 20 |
| 3 | C | 10 |

| Queue | Time Slice (time units) | Jobs |
|-------|-------------------------|---|
| 1 | 1 | A ¹ ₁ , B ¹ ₂ , C ¹ ₃ |
| 2 | 2 | A ³ ₅ , B ³ ₇ , C ³ ₉ |
| 3 | 4 | |

- Analogamente, a prima, anche in questo caso il time slice per tutti e tre scade, dunque vengono spostati alla queue inferiore

New [A | B | C]

| Order | Job | CPU burst (time units) |
|-------|-----|---------------------------|
| 1 | A | 30 |
| 2 | B | 20 |
| 3 | C | 10 |

| Queue | Time Slice (time units) | Jobs |
|-------|-------------------------|--|
| 1 | 1 | A ¹ ₁ , B ¹ ₂ , C ¹ ₃ |
| 2 | 2 | A ³ ₅ , B ³ ₇ , C ³ ₉ |
| 3 | 4 | A ⁷ ₁₃ , B ⁷ ₁₇ , C ⁷ ₂₁ |

- Una volta raggiunta la queue a priorità più bassa, il time slice di tutti e tre i processi scadrà sempre fino al loro completamento

| New | A | B | C |
|-------|-----|---------------------------|---|
| Order | Job | CPU burst (time units) | |
| 1 | A | 30 | |
| 2 | B | 20 | |
| 3 | C | 10 | |

| Queue | Time Slice (time units) | Jobs |
|-------|-------------------------|---|
| 1 | 1 | A ¹ ₁ , B ¹ ₂ , C ¹ ₃ |
| 2 | 2 | A ³ ₅ , B ³ ₇ , C ³ ₉ |
| 3 | 4 | A ⁷ ₁₃ , B ⁷ ₁₇ , C ⁷ ₂₁ A ¹¹ ₂₅ , B ¹¹ ₂₉ , C ¹⁰ ₃₂ |

Metodo - Lottery scheduling



L'algoritmo **lottery scheduling** è un algoritmo di scheduling basato sulla **casualità**:

1. Ad ogni job vengono assegnati un determinato numero di biglietti. I job attivi da poco riceveranno più biglietti, mentre quelli attivi da molto ne riceveranno di meno. Per evitare la starvation, ad ogni job viene assegnato almeno un biglietto
2. Ogni volta che un time slice scade viene determinato un biglietto vincitore. Il processo che detiene tale biglietto prenderà il controllo della CPU.
3. Successivamente, il procedimento viene ripetuto

Per via della **legge dei grandi numeri**, all'aumentare del numero di estrazioni effettuate il tempo di utilizzo della CPU di ogni processo tenderà a raggiungere la media

▼ Lezione del 2/11

Thread e Multi-threading

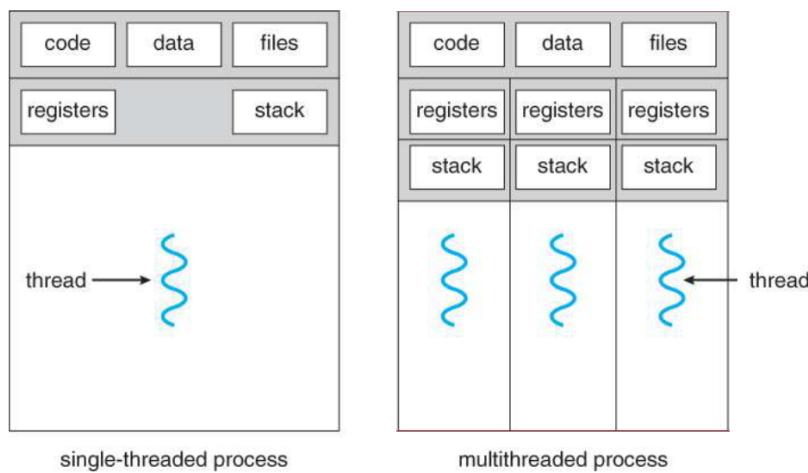
Thread



Ogni processo può essere costituito da **uno o più thread**, ognuno composto da un proprio program counter, un proprio stack, un proprio set di registri e un proprio ID.

Ogni processo definisce le **risorse "globali"** (ossia lo spazio d'indirizzamento, le istruzioni da eseguire, i dati, le risorse) mentre ogni suo thread definisce un **singolo stream** di esecuzione all'interno del processo stesso.

Poiché lo spazio d'indirizzamento del processo è **condiviso tra tutti i suoi thread**, nessuna syscall è richiesta per far cooperare i thread tra di loro, risultando in una comunicazione più semplice rispetto allo scambio di messaggi e alla memoria condivisa.



L'utilizzo dei thread risulta essere particolarmente utile nel caso in cui un processo deve svolgere **più attività indipendenti l'una dall'altra**. In particolare, nel caso in cui un'attività richieda che il processo si **blocchi**, utilizzando più thread viene

concesso alle altre attività di continuare ad essere svolte senza dover aspettare l’attività bloccata.

Teoricamente, ogni attività svolta da un software potrebbe essere svolta da un processo single-thread comunicante con gli altri. Tuttavia, l’utilizzo dei thread per la suddivisione delle attività svolte da un processo risulta essere la scelta migliore poiché:

- La comunicazione tra thread risulta essere estremamente più veloce di quella tra processi
- Il context switch tra thread risulta essere estremamente più veloce rispetto a quello tra processi

In definitiva, possiamo riassumere i **vantaggi dei processi multi-thread in:**

- **Reattività**, poiché un processo potrebbe fornire una risposta in modo rapido mentre tutti gli altri thread sono bloccati o rallentati a causa dell’intenso uso della CPU
- **Condivisione delle risorse "globali"**
- **Economicità**, poiché creare e gestire thread è più veloce rispetto alla creazione e gestione dei processi
- **Scalabilità**, poiché un processo single-thread può essere eseguito su un singolo core, mentre ogni thread di un processo multi-thread può essere suddiviso tra tutti i core disponibili della CPU (solo per architetture multi-core)

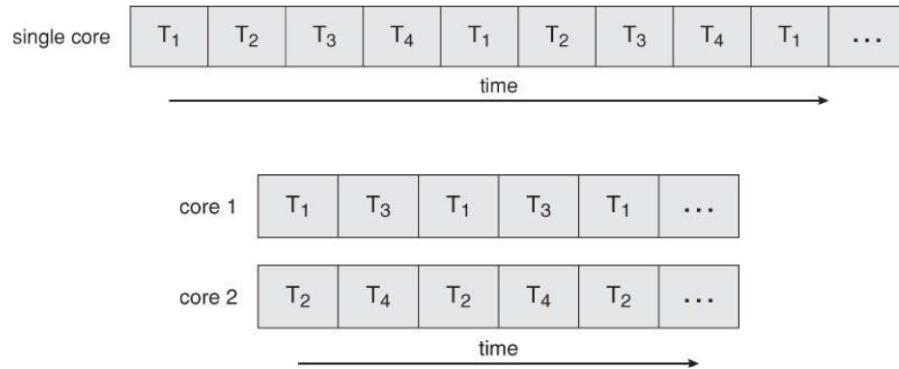
Concorrenza e Parallelismo



L'utilizzo di un'architettura basata su processori multi-core permette un **vero parallelismo tra processi**.

Spesso, le CPU moderne sono anche dotate di **hyperthreading**, dove ogni singolo core fisico appare come due core logici all'OS, permettendo lo **scheduling concorrente** di due processi per ogni core.

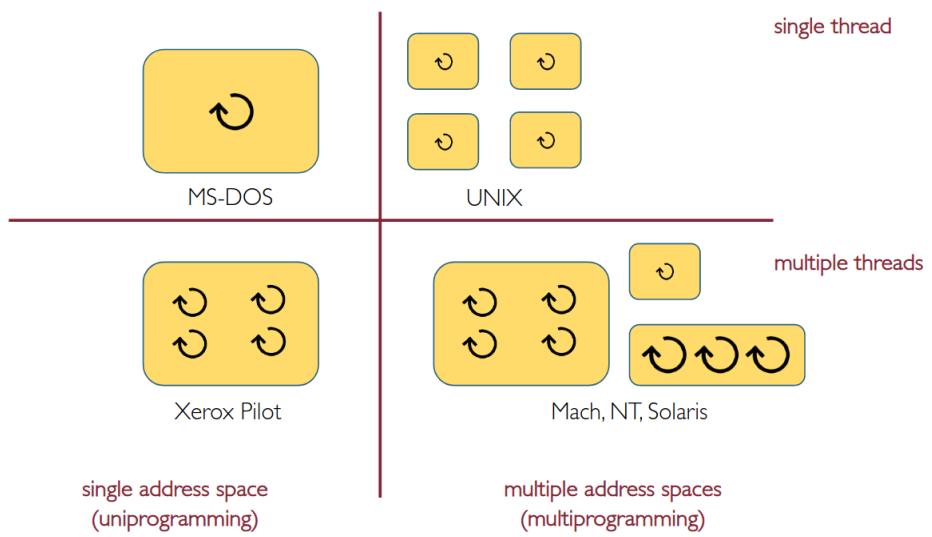
In definitiva, si parla di **concorrenza** quando vengono eseguiti processi multi-thread su CPU singlecore, mentre si parla di **parallelismo** quando vengono eseguiti processi multi-thread su CPU multicore



Tipi di parallelismo

- **Data parallelism** suddivide i dati tra più core (thread) e esegue la stessa operazione su ciascun frammento dei dati.
- **Task parallelism** suddivide le diverse attività da eseguire tra i diversi core e le esegue contemporaneamente.

Classificazione degli OS:



Kernel thread



Un **kernel thread** è l'unità di esecuzione più piccola schedulabile dall'OS, il quale è responsabile per il supporto e la gestione di tutti i kernel thread attivi, fornendo delle syscall per poterli creare e gestire dall'user space.

Ogni kernel thread è dotato di un **Thread Control Block (TCB)**.

PRO:

- Il kernel è a conoscenza di tutti i kernel thread avviati
- Lo scheduler potrebbe decidere di cedere più tempo di esecuzione della CPU ad un processo costituito da un numero elevato di thread
- Utile per applicazioni con blocchi frequenti
- Passare da un thread all'altro è più veloce rispetto al passare da un processo all'altro

CONTRO:

- Rende il kernel estremamente più complesso
- Lento ed inefficiente, poiché è necessario effettuare chiamate al kernel
- Nonostante il context switch sia più leggero, la sua gestione tramite il kernel non è consigliata

User thread



Gli **user thread** sono gestiti interamente dal sistema di run-time tramite **librerie per la gestione dei thread**, i quali vengono gestiti come se fossero dei processi singlethreaded.

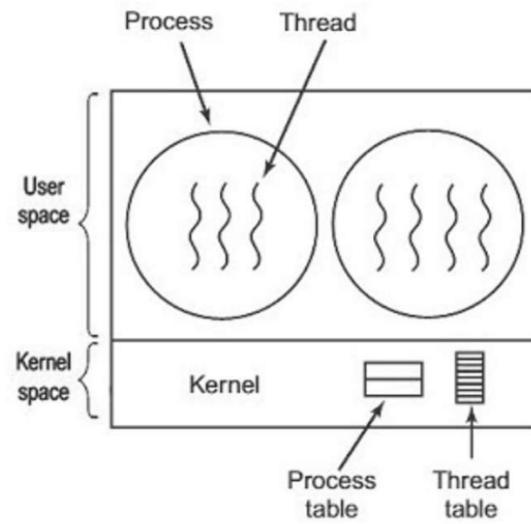
Idealmente, le attività svolte da tali thread dovrebbero essere veloci come chiamate a funzioni.

PRO:

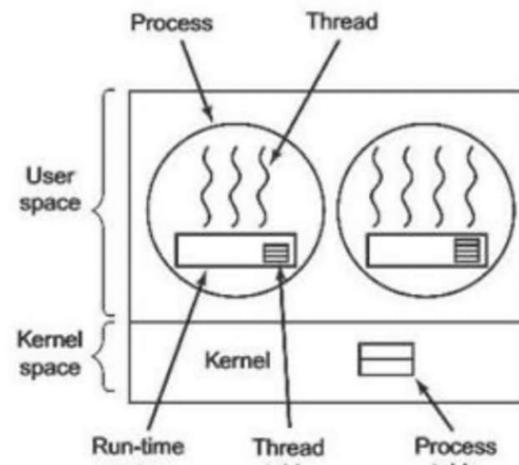
- Molto veloci e leggeri, le politiche di scheduling sono più flessibili
- Può essere implementato da OS che non supportano il threading in modo nativo
- Nessuna syscall o context switch necessario

Contro:

- Il kernel non è a conoscenza degli user thread attivi
- Le decisioni prese dallo scheduler spesso sono inefficienti
- Mancanza di coordinazione tra kernel e thread (ad esempio, un processo con 100 thread potrebbe competere per un time slice con un processo con un solothread)
- Richiede necessariamente che tutte le syscall dell'OS siano non bloccanti, poiché altrimenti tutti i thread in un processo dovrebbero attendere



Kernel Thread

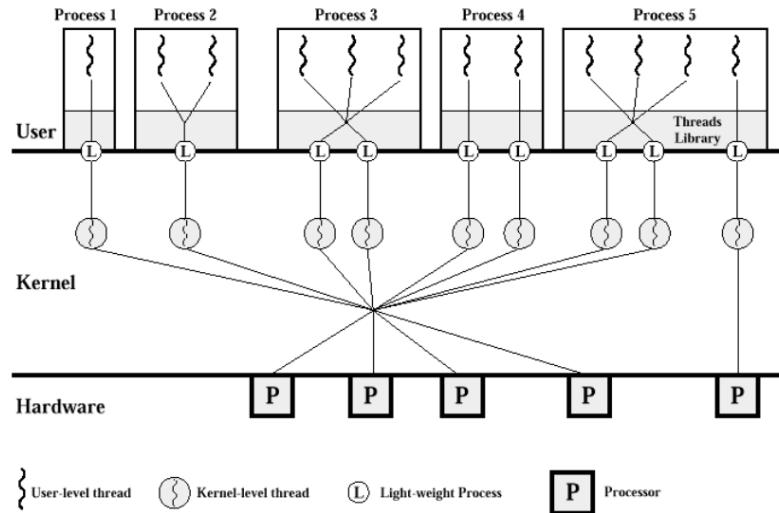


User Thread

Lightweight Process



Un **lightweight process (LWP)** è un processore **virtuale** attivo nella user space contenente un **singolo kernel thread**, mentre **multipli user thread** gestiti tramite una libreria per i thread vengono posti su uno o più LWP, i quali assumono un ruolo di "tramite" tra le due tipologie di thread

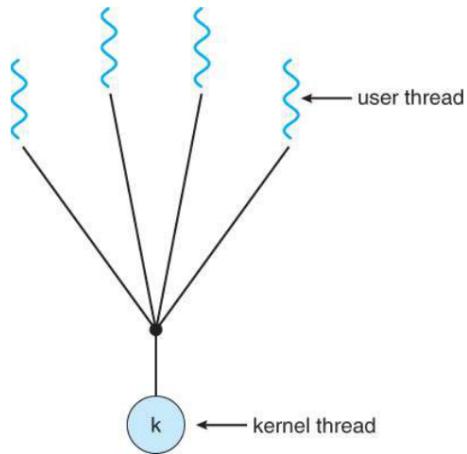


Modelli di multi-threading

Per gestire il **multithreading**, vengono utilizzati più modelli:

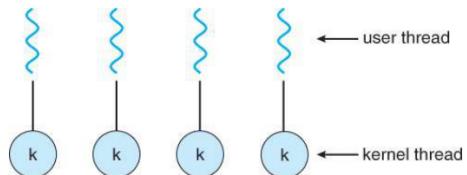
Modello Many-to-One

- Più user thread vengono mappati ad un singolo kernel thread
- Ogni processo può eseguire **un solo user thread per volta** poiché vi è un singolo kernel thread associato, il quale può operare su un singolo core, dunque i processi con più user-thread non possono essere suddivisi tra più core
- Se viene effettuata una syscall bloccante, **l'intero processo viene bloccato** anche nel caso in cui gli altri user-thread potrebbero continuare l'esecuzione.
- **Puramente basato sul livello user**



Modello One-to-One

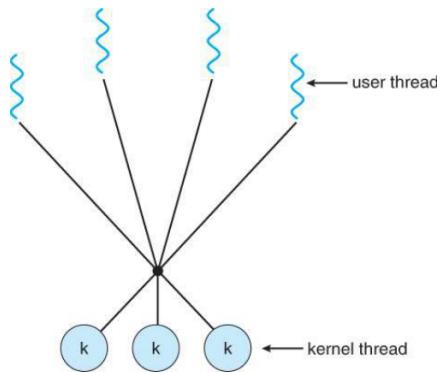
- Ogni singolo user thread viene mappato ad un singolo kernel thread
- Ogni processo può eseguire **più user thread per volta** poiché vi sono più kernel thread associati, i quali possono essere suddivisi tra più core
- Se viene effettuata una syscall bloccante, **l'intero processo non viene bloccato**
- È richiesto un kernel molto più complesso, portando il sistema ad essere rallentato
- La maggior parte delle implementazioni fissano **un limite al numero di thread** che possono essere creati
- **Puramente basato sul livello kernel**



Modello Many-to-Many

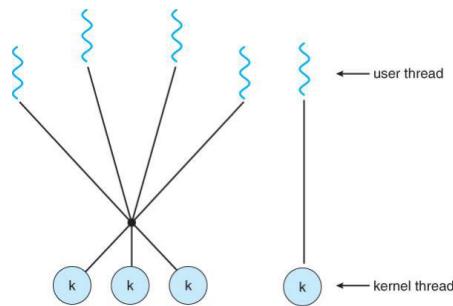
- Più user thread vengono mappati a più kernel thread

- Il numero di kernel thread può essere minore o uguale al numero di user thread
- I processi possono essere **suddivisi tra più core**
- Le syscall bloccanti **non bloccano l'intero processo**
- Il numero di thread creabili **non è limitato**



Modello Two-Level

- Variante del modello Many-to-Many, dove alcuni thread vengono gestiti tramite il modello **One-to-One**
- Le politiche di scheduling risultano **più flessibili**



Thread Pools

Idea:

- Un numero specifico di thread viene creato quando il processo inizia.
- Questi thread vengono collocati nel "pool" in attesa di un lavoro da svolgere.
- Quando il server web riceve una richiesta, viene risvegliato un thread dal pool.
- Il thread lavoratore elabora la richiesta e ritorna al pool una volta terminato.
- Se non sono disponibili thread nel pool, il server attende fino a quando uno sarà disponibile

Benefits

- Gestire una richiesta con un thread esistente è più veloce rispetto all'attesa per la creazione di un thread.
- Un pool di thread limita il numero di thread esistenti in un dato momento.
- Separare il compito da eseguire dalla creazione meccanica del compito ci consente di utilizzare diverse strategie per l'esecuzione del compito.
- Esempio: il compito potrebbe essere pianificato per essere eseguito dopo un ritardo temporale o eseguito periodicamente.

Osservazione

Se un thread effettua una syscall fork() o exec(), **l'intero processo potrebbe essere copiato** oppure potrebbe essere generato **un nuovo processo single-thread** contenente una copia del singolo thread.

Tale problematica viene gestita in uno dei seguenti modi:

1. È strettamente dipendente dalla progettazione dell'OS
2. Se il nuovo processo viene eseguito subito, allora non c'è necessità di copiare anche gli altri thread. Altrimenti, l'intero processo dovrebbe essere copiato.
3. Molte versioni di UNIX forniscono più versioni delle syscall fork() ed exec() per gestire casi specifici

Nel caso in cui un processo multi-thread riceva un segnale, è importante gestire quale thread sia il destinatario di tale segnale.

Tale problematica viene gestita in uno dei seguenti modi:

1. Il segnale viene inviato al thread che lo necessita
2. Il segnale viene inviato ad ogni thread del processo
3. Il segnale viene inviato ad alcuni thread specifici del processo
4. Viene scelto un thread specifico che vada a gestire tutti i segnali ricevuti

Scheduling dei thread



Per essere schedulato ed eseguito dalla CPU, ogni thread deve contendere con gli altri. I thread contendenti vengono gestiti dal **contention scope**, il quale si suddivide in due tipologie:

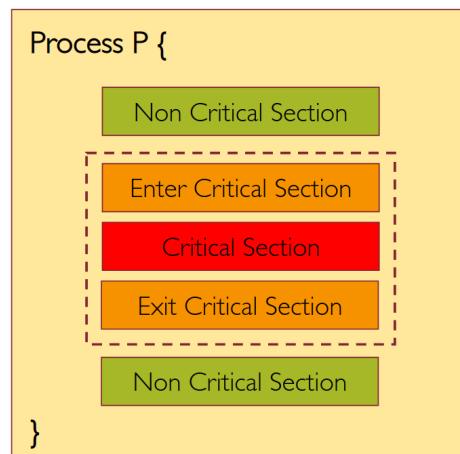
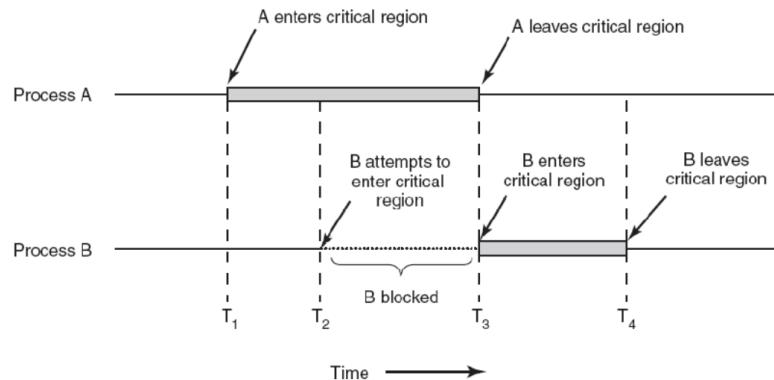
- **Process Contention Scope (PCS)**, dove la competizione avviene tra thread appartenenti allo stesso processo. Affinché sia implementabile, è necessario che il sistema utilizzi un modello Many-to-Many o Many-to-One
- **System Contention Scope (SCS)**, dove la competizione avviene tra ogni thread di ogni processo. Viene implementato in sistemi basati sul modello One-to- One

Tramite il lightweight process (LWP), il kernel è in grado di comunicare con le librerie per i thread nel momento in cui un determinato evento si verifica attraverso delle **upcall**, le quali vengono gestite da un **upcall handler** presente all'interno della libreria stessa. Ogni upcall fornisce un nuovo LWP tramite cui l'upcall handler verrà eseguito.

▼ Lezione del 09/11

Sincronizzazione tra processi e thread

Abbiamo già accennato come i processi o i thread possano cooperare tra di loro per svolgere un'attività comune. Tuttavia, affinché la cooperazione sia possibile è richiesta una **sincronizzazione** tra di essi per via della presenza di **settori critici**. Per evitare che tali settori generino problemi, vengono utilizzate delle primitive che si assicurino che **un solo thread** possa lavorare sul settore stesso.



Proprietà della sincronizzazione



Ogni **soluzione al problema del settore critico** tramite la sincronizzazione deve soddisfare tre proprietà:

- **Mutua esclusione**, ossia solo un processo/thread per volta può utilizzare il settore critico
- **Liveness**, ossia la possibilità per ogni processo di accedere al settore critico nel caso in cui nessun altro processo vi sia già
- **Attesa limitata**, ossia la possibilità per ogni processo richiedente l'accesso al settore critico di potervi accedere eventualmente, limitando il numero di processi che vi accederanno prima di esso.

Per poter sincronizzare i processi/thread tra di loro, i linguaggi di programmazione forniscono delle primitive atomiche basate su una delle seguenti **tre soluzioni**:

- La sincronizzazione avviene tramite un **lock**, dove prima di accedere ad un settore critico un processo acquisisce tale lock, per poi rilasciarlo una volta uscito dal settore critico
- La sincronizzazione avviene tramite un **semaforo**, una generalizzazione del lock
- La sincronizzazione avviene tramite un **monitor**, il quale connette dei dati condivisi alle primitive di sincronizzazione

Lock



Un **lock** fornisce la mutua esclusione tra processi utilizzando due primitive:

- **Lock.acquire()**, dove il processo rimane in attesa che il lock sia libero, per poi acquisirlo
- **Lock.release()**, dove il processo rilascia il lock precedentemente acquisito, inviando un segnale a tutti i processi che attualmente stanno eseguendo la primitiva acquire()

Per poter utilizzare correttamente un lock, è necessario seguire tre regole:

1. È sempre necessario acquisire il lock **prima** di accedere a dati condivisi
2. È sempre necessario rilasciare il lock **dopo** aver finito di utilizzare i dati condivisi
3. Il lock deve essere **inizialmente libero**
4. **Solo un processo per volta** può acquisire il lock

Poiché lo scheduler della CPU prende il controllo solo a seguito di **eventi interni**, ossia quando il thread in esecuzione lascia il controllo della CPU, oppure a seguito di **eventi esterni**, per evitare problemi all'interno dei settori critici è necessario **impedire lo scheduling** durante il rilascio o l'acquisizione di un lock. Per implementare le primitive ad alto livello relative alla sincronizzazione, quindi, è necessario del **supporto hardware al basso livello** che possa rendere tali operazioni atomiche, in particolare tramite la **disabilitazione degli interrupt** o l'uso di **istruzioni atomiche**.

Definizione - Operazione Atomica



Un'operazione viene detta **atomica** se essa **non può essere interrotta** in alcun modo, impedendo quindi che possa avvenire un context switch durante la sua esecuzione.

Metodo - Disabilitazione degli interrupt



La prima soluzione a supporto hardware per poter implementare la sincronizzazione, corrisponde alla **disabilitazione degli interrupt** mentre un **thread acquisisce o rilascia un lock**

```
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;  
    }  
}
```

```
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();  
}
```

Metodo - Istruzioni atomiche



La **seconda soluzione a supporto hardware** per poter implementare la sincronizzazione, corrisponde all'uso di un'**istruzione atomica** in grado di leggere un valore dalla memoria e modificarlo in un singolo istante.

Nella maggior parte delle architetture, viene utilizzata l'istruzione **test&set**:

- Nel caso in cui il lock sia libero (dunque il suo valore associato è 0), l'istruzione

test&set(value) leggerà 0 per poi settare tale valore ad 1, restituendo il valore 0 letto (

acquisizione del lock)

- Nel caso in cui il lock sia occupato (dunque il suo valore associato è 1), l'istruzione

test&set(value) leggerà 1 per poi settare tale valore ad 1, restituendo il valore 1 letto (**attesa del lock**)

```
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;  
}
```

Problematiche

L'implementazione del lock tramite **disabilitazione degli interrupt** presenta alcune **problematiche**:

- La necessità dell'invocazione del kernel rende il tutto più complesso
- Inutilizzabile in sistemi multi-core

Analogamente, anche l'implementazione del lock tramite **istruzioni atomiche** presenta alcune

problematiche:

- Ogni thread che non ha acquisito il lock è in attesa che esso si liberi, dunque la CPU non sta realmente svolgendo delle attività
- Una volta che il lock è libero, tutti i thread in attesa si contenderanno il lock, dunque non è possibile determinare quale thread otterrà il controllo

Nonostante il tempo in attesa della CPU sia un problema irrisolvibile nel caso delle istruzioni atomiche, esso può essere **minimizzato**. Inoltre, l'aggiunta di una **queue** permette di gestire in modo deterministico quale thread vada ad ottenere il lock a seguito del suo rilascio:

```
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;
}
```

▼ Lezione del 14/11

Semafori



Un **semaforo** è una generalizzazione del lock dotata di una propria **queue** ed una **variabile** sulla quale vengono svolte due primitive:

- *wait()* (o *P()*), la quale decrementa la variabile e inserisce il processo nella queue, in attesa che il semaforo sia aperto. Se quando la primitiva viene invocata
 - il semaforo è aperto, allora il processo continua l'esecuzione.
- *signal()* (o *V()*), la quale incrementa la variabile e permette ad un altro thread
 - di accedere al settore critico. Se un thread è in attesa nella coda, allora esso viene sbloccato. Altrimenti, se non ci sono thread nella coda, il segnale viene ricordato per il thread successivo.

I semafori si differenziano in due tipi:

- **Semafori binari, detti Mutex**, utilizzati per gestire una singola risorsa condivisa,
dove il valore iniziale della variabile associata corrisponde a 1. Di conseguenza,
il suo funzionamento coincide con quello di un lock.
- **Semafori di conteggio**, utilizzato per gestire risorse multiple condivise,
dove
il valore iniziale della variabile associata corrisponde al numero di risorse stesso.
Un processo può accedere ad una risorsa finché almeno una di esse è disponibile

```
// Semaphore S

S.wait(); // wait until S is available

<critical section>

S.signal(); notify other processes that S is open
```

```
Class Semaphore {
    public void wait(Thread t);
    public void signal();
    private int value;
    private int guard;
    private Queue q;

    Semaphore(int val) {
        // initialize semaphore
        // with val and empty queue
        this.value = val;
        this.q = null;
    }
}
```

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

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

Esempi:

1. Considerando i seguenti due processi A e B, un possibile sviluppo dell'esecuzione è il seguente:



| s (value) | Queue | A | B |
|---------------|-------|---------------|---------------|
| 2 | ∅ | ready to exec | ready to exec |
| A: S.wait() | 1 | ∅ | ready to exec |
| B: S.wait() | 0 | ∅ | ready to exec |
| A: S.wait() | -1 | A | blocked |
| B: S.signal() | 0 | ∅ | ready to exec |
| A: S.signal() | 1 | ∅ | ready to exec |
| A: S.signal() | 2 | ∅ | ready to exec |

2. Consideriamo i due seguenti processi Producer e Consumer:

Producer Process:

```

while (true)
{
    /* produce an item in nextProduced */
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}

```

Consumer Process:

```

while (true)
{
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in nextConsumed */
}

```

- I due processi condividono un buffer comune. La variabile counter tiene traccia del numero di elementi attualmente nel buffer.
- In tale caso, potrebbe crearsi una race condition: in assenza di sincronizzazione, i due processi potrebbero lavorare contemporaneamente sulla variabile
- Ad esempio, ipotizziamo che inizialmente si abbia counter = 5. In tal caso, una possibile esecuzione del programma potrebbe essere la seguente

Producer:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

Consumer:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Interleaving:

| | | | | |
|---------|-----------------|---------|-------------------------------|--------------------------|
| T_0 : | <i>producer</i> | execute | $register_1 = \text{counter}$ | { $register_1 = 5$ } |
| T_1 : | <i>producer</i> | execute | $register_1 = register_1 + 1$ | { $register_1 = 6$ } |
| T_2 : | <i>consumer</i> | execute | $register_2 = \text{counter}$ | { $register_2 = 5$ } |
| T_3 : | <i>consumer</i> | execute | $register_2 = register_2 - 1$ | { $register_2 = 4$ } |
| T_4 : | <i>producer</i> | execute | $\text{counter} = register_1$ | { $\text{counter} = 6$ } |
| T_5 : | <i>consumer</i> | execute | $\text{counter} = register_2$ | { $\text{counter} = 4$ } |

Monitor



Un **monitor** è un **costrutto** dei linguaggi di programmazione in grado di **controllare l'accesso ai dati condivisi**. In particolare, tramite tale costrutto, il codice relativo alla sincronizzazione viene inserito direttamente dal compilatore e imposto dal runtime stesso del linguaggio utilizzato. I processi che tentano di accedere al monitor vengono messi all'interno di una **queue**.

Formalmente, un monitor è costituito da un **lock**, utilizzato per assicurare la mutua esclusione sui dati condivisi e che solo un thread per volta sia attivo all'interno del monitor stesso, e zero o più **variabili di condizione**, utilizzate per gestire gli accessi concorrenti ai dati condivisi.

Ogni variabile di condizione supporta tre operazioni:

- *wait()*, dove il lock viene liberato e il thread viene messo a dormire nella queue
- *signal* (o *notify()*), dove un processo della queue, se esistente, viene svegliato,
altrimenti non viene effettuata alcuna operazione
- *broadcast* (o *notifyAll()*), dove tutti i thread vengono svegliati
Durante lo svolgimento di un'operazione su variabili condizionali, è necessario che il lock rimanga **chiuso**

Esempio:

```

class Queue {
    ...
    private ArrayList<Item> data;
    ...

    public void synchronized add(Item i) {
        data.add(i);
        notify();
    }

    public Item synchronized remove() {
        while (data.isEmpty()) {
            wait(); // give up the lock and sleep
        }
        Item i = data.remove(0);
        return i;
    }
}

```

Nonostante le similitudini, le variabili condizionali **non sono semafori**, poiché:

- L'accesso al monitor è controllato da un lock aggiuntivo
- Nei monitor, *wait()* blocca il thread chiamante e rilascia il lock, dunque è necessario che il thread stia già accedendo al monitor. Inoltre, se nessun thread è nella queue, allora il segnale inviato da *signal()* viene perso.
- Nei semafori, invece, *wait()* si occupa solo di bloccare il thread nella queue, mentre il segnale inviato da *signal()* viene preservato per il prossimo thread

Comunemente, i monitor vengono implementati secondo due stili:

- Lo **stile Mesa**, dove il thread segnalante inserisce un thread in attesa nella ready queue mentre il primo continua l'esecuzione all'interno del monitor, e dove la condizione, la quale deve essere verificata continuamente, non deve essere necessariamente verificata quando il thread in attesa viene nuovamente eseguito

```

class Queue {
    ...
    private ArrayList<Item> data;
    ...

    public void synchronized add(Item i) {
        data.add(i);
        notify();
    }

    public Item synchronized remove() {
        while (data.isEmpty()) {
            wait(); // give up the lock and sleep
        }
        Item i = data.remove(0);
        return i;
    }
}

```

- Lo **stile Hoare**, dove il thread segnalante viene trasformato immediatamente in un thread in attesa e dove la condizione anticipata dal thread in attesa è garantita quando esso viene eseguito

```

class Queue {
    ...
    private ArrayList<Item> data;
    ...

    public void synchronized add(Item i) {
        data.add(i);
        notify();
    }

    public Item synchronized remove() {
        if (data.isEmpty()) {
            wait(); // give up the lock and sleep
        }
        Item i = data.remove(0);
        return i;
    }
}

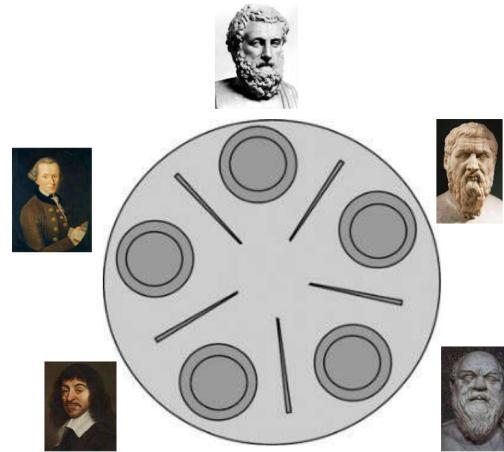
```

▼ Lezione del 16/11

Deadlock

Problema della cena dei filosofi

Ci sono 5 filosofi seduti ad un tavolo rotondo, ognuno provvisto di una bacchetta di legno alla propria sinistra, per un totale di 5 bacchette. Per poter mangiare, ogni filosofo necessita di due bacchette. L'obiettivo è far sì che ogni filosofo mangi.



Deadlock



Definiamo come **deadlock** una situazione in cui due o più processi sono in attesa di un evento che può essere generato da loro stessi, creando un blocco **perenne**.

Esempio:

- Consideriamo la seguente situazione

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();
```

- Le risorse printer e disk sono regolate da un mutex, dunque esiste un'unica risorsa per entrambi i tipi

- Il processo A esegue l'istruzione `printer.wait()` per acquisire la risorsa, decrementandone il mutex ed eseguendo il context switch, cedendo quindi il controllo sulla CPU al processo B
- Successivamente, in modo analogo, il processo B acquisirà la risorsa disk, effettuando quindi il context switch e cedendo il controllo ad A
- A questo punto, il processo A eseguirà l'istruzione `disk.wait()`, richiedendo di acquisire la risorsa disk. Tuttavia, poiché il mutex associato è impostato a 0, dunque non vi sono più risorse di tipo disk disponibili, il processo A rimarrà in attesa che la risorsa venga liberata, cedendo il controllo della CPU al processo B
- Tuttavia, una volta eseguita l'istruzione `printer.wait()`, anche il processo B rimarrà in attesa che la risorsa printer venga liberata
- Infine, quindi, va a crearsi una situazione dove il processo A è in attesa che il processo B liberi disk, il quale tuttavia è in attesa che il processo A liberi printer, creando quindi un **deadlock**

Deadlock e Starvation



Nonostante siano termini legati tra loro, è necessario distinguere

deadlock e starvation:

la

starvation si verifica nel caso in cui un thread attende indefinitivamente che una risorsa venga liberata, tuttavia, a differenza del *deadlock*, gli altri processi stanno continuando la loro esecuzione usufruendo di tale risorsa. Nel caso del deadlock, quindi, **l'intero sistema rimane bloccato.**

Consideriamo ora quindi delle possibili soluzioni al problema dei filosofi a cena:

- Esiste una **prima** soluzione banale al problema: possiamo utilizzare un lock globale che permette ad un singolo filosofo per volta di poter prendere in

mano due bacchette. In tal modo, tuttavia, l'assenza di concorrenza tra i filosofi implica che essi non possano mangiare contemporaneamente

- Una **seconda** soluzione prevede l'uso di un semaforo:

```
Semaphore chopsticks[5];

while(True) {
    chopsticks[i].wait();           // wait on the left chopstick
    chopsticks[(i+1)%5].wait();    // wait on the right chopstick

    eat();

    chopsticks[i].signal();        // signal on the left chopstick
    chopsticks[(i+1)%5].signal();  // signal on the right chopstick

    think();
}
```

Tuttavia, nel caso in cui ogni filosofo prenda in mano la bacchetta sinistra in contemporanea agli altri, si andrebbe a creare un deadlock poiché ogni filosofo rimarrebbe in attesa che la bacchetta destra sia libera.

- Una **terza** soluzione prevede l'uso dei monitor: prima di prendere in mano una delle due bacchette, è necessario assicurarsi che la seconda sia libera, altrimenti sarà necessario aspettare che entrambe siano libere. Dunque, sarà necessario controllare se il filosofo alla propria destra e il filosofo alla propria sinistra stiano mangiano oppure no (variabili condizionate)

Condizioni necessarie per il deadlock



Affinché un deadlock possa verificarsi, sono necessarie tutte e quattro le seguenti condizioni:

- **Mutua esclusione**, almeno un thread è in possesso di una risorsa non condivisibile (in particolare, solo un thread può essere in possesso della risorsa)
- **Possesso e Attesa**, dove almeno un thread è in possesso di una risorsa non condivisibile ed è in attesa che un'altra risorsa attualmente posseduta da un altro thread sia disponibile
- **No-preemption**, dove un thread può rilasciare una risorsa solo per scelta volontaria, dunque né un altro processo né un altro thread possono forzare il rilascio
- **Attesa circloare**, dove si ha una catena di thread t_1, \dots, t_n in cui ognuno di essi è in attesa che il thread successivo appartenente alla catena stessa rilasci la risorsa richiesta

Rilevare un deadlock

Resource Allocation Graph (RAG)



Un **Resource Allocation Graph (RAG)** è un grafo diretto $G = (V, E)$ dove:

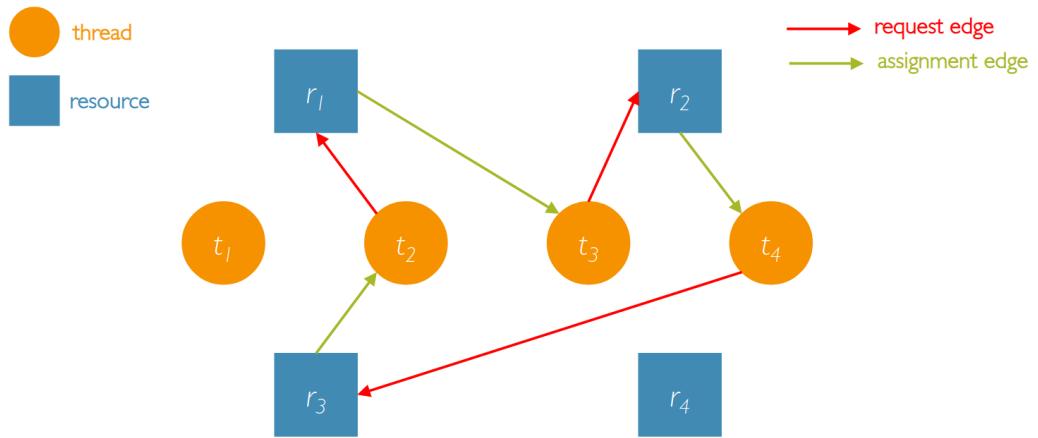
- V è l'insieme dei vertici rappresentanti sia le risorse (r_1, \dots, r_k) sia i thread (t_1, \dots, t_n) considerati
- E è l'insieme degli archi del grafo, i quali possono essere di due tipi:

— **Archi di richiesta** (frecce rosse), indicante che il thread t_i ha richiesto la risorsa r_j

— **Archi di assegnamento** (frecce verdi), indicanti che l'OS ha allocato la risorsa r_k al thread t_h

Esempi:

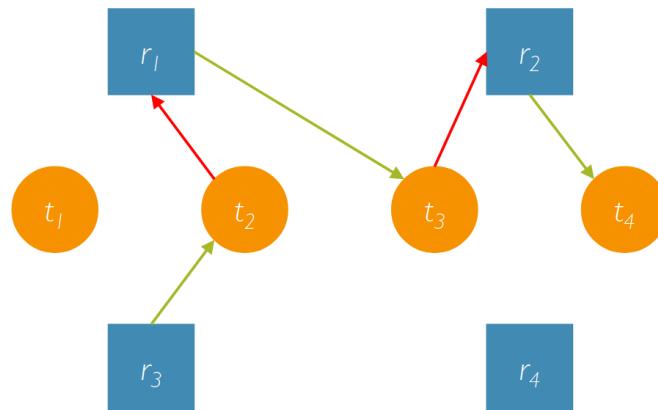
- Consideriamo il seguente RAG:



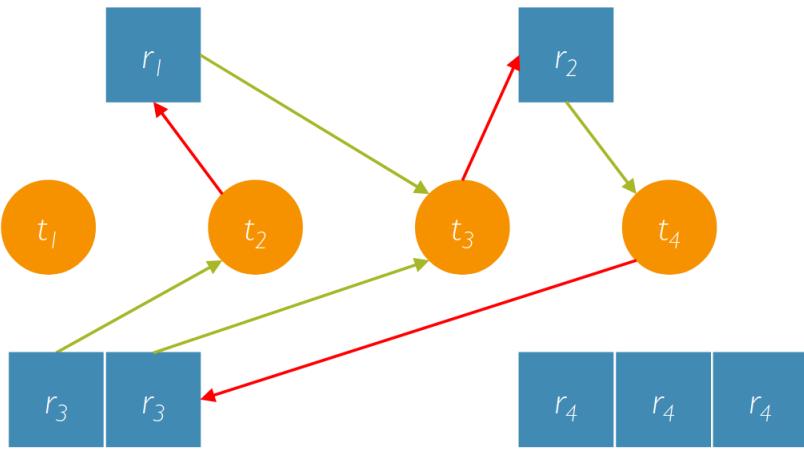
63

- In tal caso, è facilmente individuabile la presenza di un deadlock:
 1. Il thread t_2 , possidente la risorsa r_3 , richiede la risorsa r_1 , rimanendo in attesa che essa sia liberata dal thread t_3

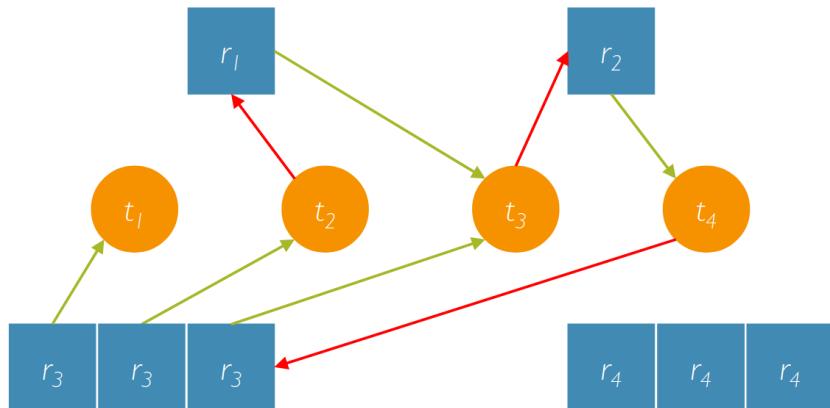
2. A questo punto, il thread t3, possedente la risorsa r1, richiede la risorsa r2, rimanendo in attesa che essa sia liberata dal thread t4
 3. A sua volta, il thread t4, possedente la risorsa r2, richiede la risorsa r3, rimanendo in attesa che essa sia liberata dal thread t2
 4. In definitiva, si è in una situazione dove t2 aspetta t3, il quale sta aspettando t4, il quale a sua volta sta aspettando t2, creando quindi un deadlock
- Se il thread t4 non richiedesse la risorsa r3, le condizioni necessarie affinché possa verificarsi un deadlock non sarebbero soddisfatte, per via dell'assenza dell'attesa circolare



- Consideriamo il caso in cui esistano due istanze della risorsa r3, dove prima istanza è assegnata al thread t2 e la seconda è assegnata al thread t3.



- Anche in tal caso, siamo in una situazione di deadlock:
 1. t4 richiede un'istanza di r3, rimanendo in attesa che t2 o t3 ne rilascino almeno una.
 2. Tuttavia, t3 è in attesa che t4 ceda r2, mentre t2 è in attesa che t3 ceda r1
- Invece, se ci fossero tre istanze di r3 e solo due di esse venissero utilizzate da t2 e t3, non si andrebbe a creare un deadlock, poiché t4 non rimarrebbe in attesa, sbloccando quindi la catena



Prevenire ed evitare un deadlock

Come visto nella sezione precedente, per prevenire un deadlock è sufficiente che una sola delle quattro condizioni necessarie non si verifichi.

Inoltre, è possibile studiare il comportamento di una sequenza di n thread per determinare se tale sequenza sia sicura o non:

- Sia m_i il numero massimo di risorse che il thread i possa richiedere
- Sia c_i il numero di risorse attualmente occupate dal thread i
- Sia $C = \sum_{i=1}^n c_i$ il numero totale di risorse attualmente allocate nel sistema
- Sia R il numero massimo di risorse disponibili
- Definiamo una sequenza di thread come sicura se per ogni thread si verifica che:

$$m_i - c_i \leq R - C + \sum_{j=1}^{i-1} c_j$$

dove:

- $m_i - c_i$ è il numero di risorse ancora richiedibili da t_i
- $R - C$ è il numero di risorse attualmente disponibili
- $\sum_{j=1}^{i-1} c_j$ è il numero di risorse attualmente allocate fino al thread t_j , con $j < i$

Policy a stato sicuro



Definiamo come **stato sicuro** una situazione in cui si ha una sequenza sicura per i thread.

Uno

stato insicuro non implica la presenza di un deadlock, poiché potrebbe verificarsi che alcuni thread non richiedano simultaneamente il massimo numero di risorse necessarie dichiarato.

Tramite tale

policy, una risorsa viene ceduta ad un thread **solo se il nuovo stato è sicuro**, altrimenti tale thread rimarrà in attesa che la risorsa sia disponibile in modo sicuro, impedendo quindi la creazione di possibili attese circolari.

Archi di pretesa

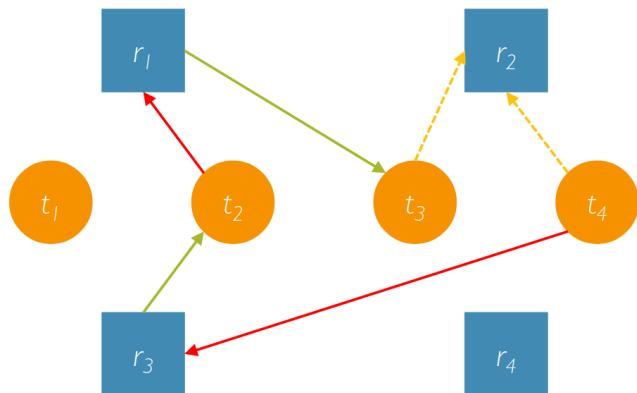
Oltre alla formula numerica precedentemente vista, viene utilizzata una variante del RAG avente una tipologia aggiuntiva di arco, ossia gli **archi di pretesa**, indicanti che il thread ti potrebbe richiedere la risorsa r_j in futuro.

In questa tipologia di RAG,

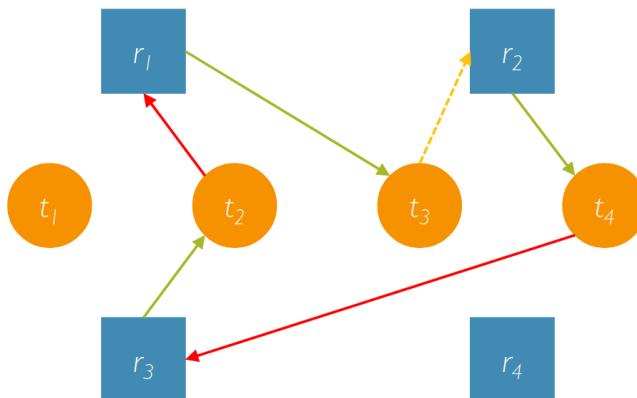
soddisfare una pretesa equivale a trasformare un arco di pretesa in un arco di assegnamento, mentre la presenza di cicli indica **un possibile stato insicuro**. Se un assegnamento generasse uno stato insicuro, allora tale assegnamento non verrà effettuato anche nel caso in cui la risorsa sia effettivamente disponibile.

Esempio:

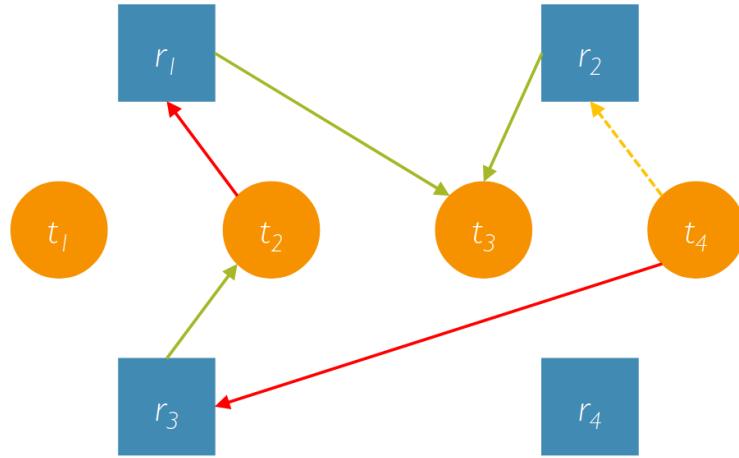
- Consideriamo il seguente RAG



- Supponiamo che venga soddisfatta la pretesa del thread t4. In tal caso, si andrebbe a creare uno stato insicuro poiché, nel caso in cui anche il thread t3 vada a richiedere la risorsa r3 (dunque trasformando l'arco di pretesa in un arco di richiesta) si andrebbe a creare un deadlock. Dunque tale pretesa non verrebbe soddisfatta



- Nel caso in cui invece venga soddisfatta la pretesa del thread t2, si rimarrebbe in uno stato sicuro poiché, nel caso in cui il thread t4 andasse a richiedere la risorsa r3 esso rimarrebbe in attesa di t3, il quale può continuare a lavorare poiché non è in attesa di nessuno. Dunque tale pretesa verrebbe soddisfatta.



Algoritmo del Banchiere

Si fa una simulazione di quale sarebbe lo stato una volta accettata la richiesta, se lo stato risulta sicuro viene accettata la richiesta:

Assunzioni per l'algoritmo :

- n attività (threads) e m tipi di risorse
- max[1,...,n;1,...m] matrice n*m
 - max[i,j] thread i potrebbe richiedere massimo di risorsa j
- allocation[1,...,n;1,...m] matrice n*m
 - allocation[i,j] thread i ha allocate j risorse
- need[1,...,n;1,...m] matrice n*m
 - need[i,j] = max[i,j] - allocation[i,j]

L'algoritmo si divide in 2 procedure principali:

1. isSafeState → dato lo stato corrente di allocazione di risorse controlla se lo stato è safe.
2. resourceRequest → dato un thread e le sue risorse richieste decide se la richiesta può essere soddisfatta.

isSafeState

Siano work e finish 2 vettori di lunghezza m e n

1. Inizializza

- work = avabile;
- finish[i] =False $\forall i$

2. Trova una i tale che:

- finish[i] = False && need[i] \leq work (Se non esiste \rightarrow 4)

3. Se i esiste:

- work = work + allocation[i]
- finisci[i] = true
- Torna a 2)

4. se finisc[i]==true $\forall i$, il sistema è in uno stato safe

resourceRequest

Input: i (thread) e richiede un vettore m-dimensionale di richieste

1. Se richiesta>need[i]

- Verifica se la richiesta è maggiore delle risorse che restano a disposizione (se supera il max), sennò da errore

2. Se richiesta > available

- Il thred i deve aspettare finchè le risorse non saranno disponibili

3. Se ci sono risorse disponibili

- Controlla se questa allocazione condurrà ad uno stato safe simulandolo
- available -= request, allocation[i]+= request, need[i] -=request
- isSafeSTate() ? Se si rollback() e wait()

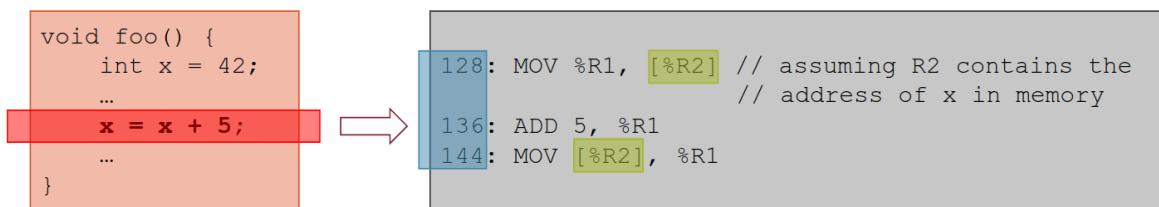
Esempi dell'algoritmo su slide

▼ Lezione del 28/11

Gestione della memoria

Durante l'esecuzione di un programma, la CPU si occupa di ripetere continuamente il ciclo di fetch, decode e execute. La maggior parte delle istruzioni coinvolge il prelevamento e il salvataggio di dati in memoria. I vari chip di memoria, tuttavia, rispondono alle richieste relative solo ad **indirizzi fisici** per la memoria. Dunque, è necessario effettuare una connessione tra i **nomi simbolici** descritti dei programmi utente e i veri e proprio indirizzi fisici della memoria.

Nel programma seguente, la variabile x corrisponde ad un nome simbolico facente riferimento ad un **indirizzo logico** contenente sia istruzioni sia dati



Un **nome simbolico** corrisponde ad un riferimento simbolico alla memoria utilizzato dai programmi utente.

Tale nome simbolico viene poi convertito in un **indirizzo logico**, ossia un indirizzo di memoria generato dal programma utente tramite la CPU, il quale viene a sua volta convertito in un **indirizzo fisico**, ossia l'effettivo indirizzo di memoria contenente il dato.

Ogni indirizzo logico viene **associato** ad un indirizzo fisico tramite tre modalità:

- Associazione in fase di **compilazione**
- Associazione in fase di **avvio**
- Associazione in fase di **esecuzione**

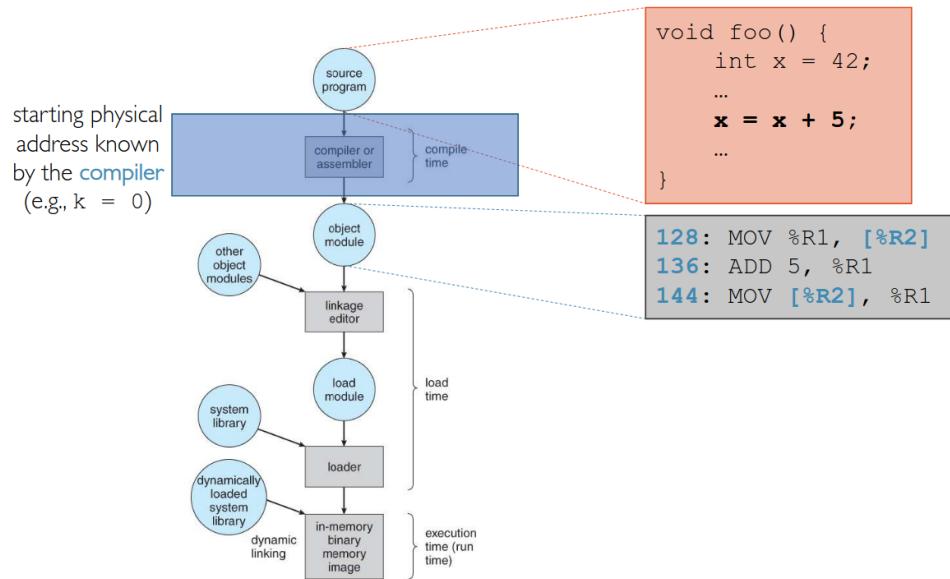
Associazione degli indirizzi

Associazione in fase di compilazione



Se l'indirizzo fisico iniziale dove un programma risiede in memoria **è noto** in fase di compilazione, allora il compilatore genererà del **codice assoluto**, richiedente **nessun intervento dell'OS** e dove l'indirizzo logico e quello fisico **coincidono**.

Se l'indirizzo iniziale viene successivamente modificato, allora il programma dovrà essere
ricompilato.



Associazione in fase di avvio



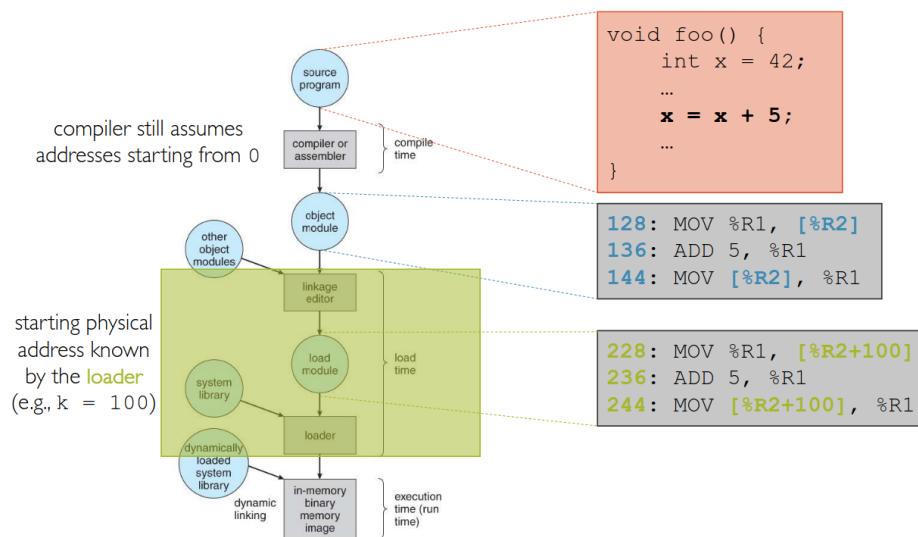
Se l'indirizzo fisico iniziale dove un programma risiede in memoria **non è noto** in fase di compilazione, allora il compilatore genererà del **codice statico rilocabile**, facente riferimento ad indirizzi relativi all'indirizzo iniziale.

L'

avviatore dei processi dell'OS determinerà l'indirizzo fisico iniziale di ogni processo.

Anche in questo caso, l'indirizzo logico e quello fisico **coincidono**.

Se l'indirizzo iniziale viene successivamente modificato, allora il programma dovrà essere **riavviato**, ma non ricompilato.

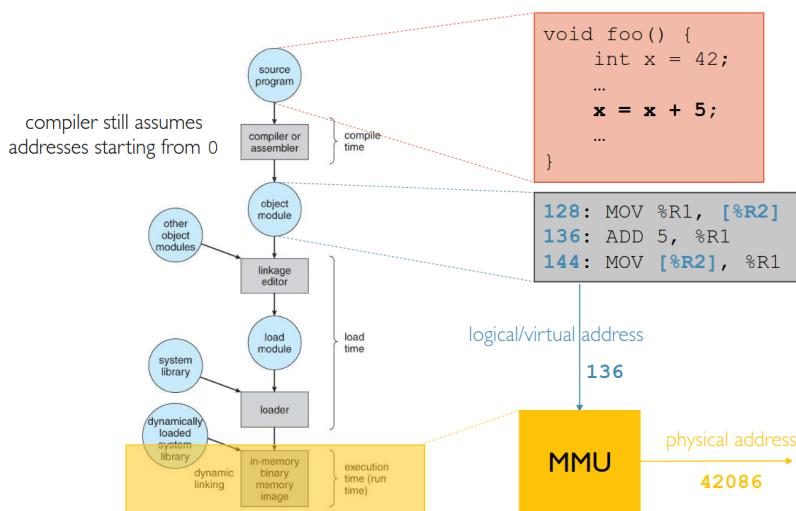


Associazione in fase di esecuzione



Se il programma può essere mosso all'interno della memoria principale durante la sua esecuzione, il compilatore genererà degli **indirizzi virtuali** (o codice dinamico rilocabile)

L'OS si occuperà di associare gli indirizzi fisici della memoria tramite una **Memory Management Unit (MMU)**. In questo caso, quindi, l'indirizzo logico (o virtuale) e l'indirizzo fisico **non coincidono**



A seconda del tipo di architettura che si vuole implementare, vengono utilizzate diverse **tecniche di gestione della memoria**:

- **Gestione della memoria uni-programmabile**, dove all'OS viene assegnata una parte fissa di memoria fisica e dove viene eseguito un singolo processo per volta, il quale possiede sempre lo stesso indirizzo iniziale, implicando che l'associazione tra indirizzo logico e fisico possa avvenire in fase di compilazione.
- **Gestione della memoria multi-programmabile**, dove:
 - Più processi coesistono in contemporanea nella memoria e processi cooperanti condividono porzioni dello spazio di indirizzamento
 - I processi non sono a conoscenza della condivisione della memoria e della porzione di memoria in cui si trovano

- I processi non sono in grado di accedere ai dati di altri processi e del sistema operativo, rendendo impossibile la loro corruzione
- Le performance della CPU e della memoria non diminuiscono di troppo durante la condivisione e vi è poca frammentazione dei dati

Rilocazione dei processi

Durante la rilocazione dei processi, viene assunto che il sistema operativo sia allocato nella parte più alta della memoria e che gli indirizzi logici generati da ogni programma utente siano compresi tra l'indirizzo più basso e l'ultimo indirizzo utilizzabile (dunque

memory_size - os_size - 1).

Ogni processo viene caricato in memoria allocando il primo **segmento di memoria contiguo** in cui il processo può essere contenuto.

Rilocazione statica



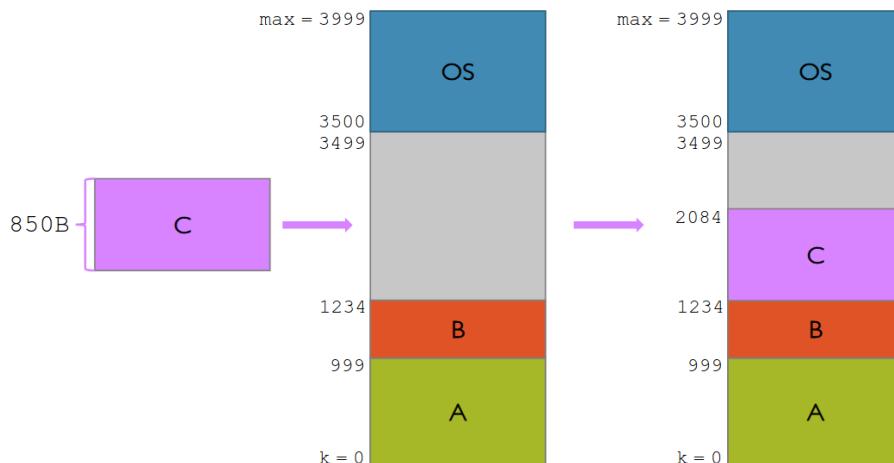
La **rilocazione statica** prevede che l'avviatore dell'OS riscrive gli indirizzi generati da un processo in modo che essi corrispondano la loro posizione della memoria principale (**associazione in fase di avvio**).

Pro:

- Non è necessario ulteriore supporto hardware

Contro:

- Non vi è mutua protezione tra i processi, in quanto ognuno di essi può corrompere l'OS o altri processi stessi
- Lo spazio di indirizzamento deve essere allocato in modo contiguo
- Una volta allocato in memoria, un processo non può essere spostato dall'OS



Rilocazione dinamica



La **rilocazione dinamica** si basa sulla protezione reciproca dell'OS e dei processi, necessitando di una **MMU** tramite cui associare gli indirizzi in fase di esecuzione.

La MMU si occupa di **tradurre** dinamicamente ogni indirizzo logico/virtuale generato da un processo nel corrispettivo indirizzo fisico. Essa contiene almeno un **registro base**, contenente l'indirizzo fisico iniziale dello spazio di indirizzamento, e un **registro limite**, contenente la dimensione massima dello spazio di indirizzamento.

Tramite tali registri, ogni processo può accedere solo alle zone di memoria appartenenti al segmento ad esso assegnato:

- Ogni processo **crede** che il proprio primo indirizzo fisico sia 0
- Durante una richiesta di accesso alla memoria, la MMU controllerà che l'indirizzo richiesto dal processo sia **inferiore** al limite massimo associato allo spazio di indirizzamento del processo stesso
- Nel caso in cui l'indirizzo richiesto **superi** il limite, la MMU restituirà un errore
- Invece, nel caso in cui l'indirizzo richiesto sia **inferiore** al limite, la MMU sommerà tale indirizzo all'indirizzo base associato allo spazio di indirizzamento del processo stesso, per poi inviare l'indirizzo alla memoria

▼ Lezione del 30/11

Policy di allocazione e frammentazione

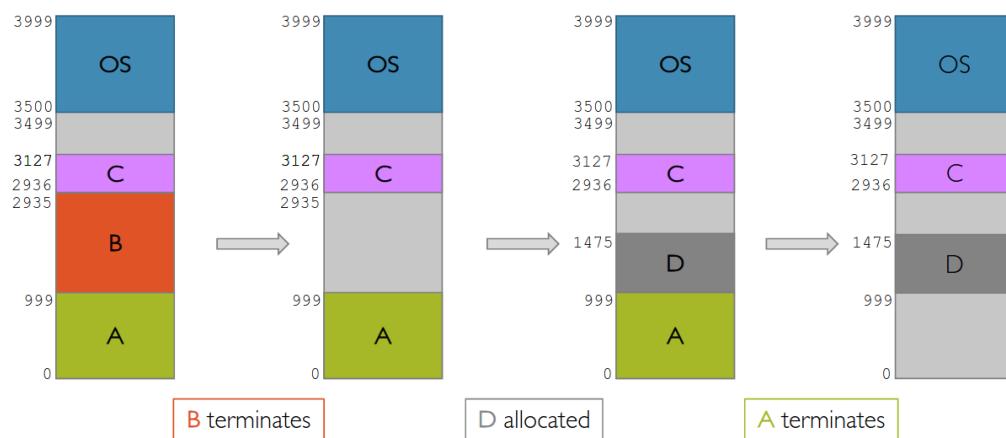
Finora abbiamo dato per assunto che ogni processo venga allocato in uno spazio contiguo di memoria fisica.

Un metodo semplice per poter gestire tale tipologia di allocazione consiste nel dividere preventivamente l'intera memoria dedicata ai processi utente in **partizioni di dimensione equivalente**, assegnando ogni partizione ad un processo.

Un altro approccio consiste nel far tenere traccia al sistema operativo dei segmenti di memoria liberi, ossia inutilizzati. Tali segmenti inutilizzati vengono detti **buchi**.

Consideriamo la seguente situazione:

1. I processi A,B e C vengono allocati nella memoria
2. Successivamente, il processo B termina, lasciando un buco nella memoria al suo posto
3. In seguito, viene allocato il processo D nel buco lasciato dal processo B, creando a sua volta un ulteriore buco, minore rispetto al precedente



Dunque, ogni volta che un processo viene allocato, l'OS deve essere in grado di scegliere adeguatamente quale buco di memoria riempire.

Metodo - Policy first-fit



Prima dell'allocazione viene effettuata una **scansione lineare** della memoria finché non viene trovato un buco abbastanza grande da poter contenere il processo. In altre parole, il processo viene allocato nel **primo buco disponibile**.

Tale policy viene detta **first-fit**.

Esempio:

- Supponiamo di avere tre buchi in memoria, rispettivamente di 20, 400 e 120 byte.
- Supponiamo che il processo X richieda 100 byte di memoria. Tramite la policy first-fit, il processo verrebbe inserito nel primo buco sufficientemente grande, ossia quello da 400 byte, creando così un buco di 300 byte
- Supponiamo ora che il processo Y richieda 350 byte di memoria. L'OS non è in grado di soddisfare tale richiesta, poiché nessun buco è abbastanza grande.

Metodo - Policy best-fit



Prima dell'allocazione viene effettuata una scansione lineare di **tutti i buchi** presenti in memoria. Successivamente, viene selezionato come spazio d'allocazione il **minor buco abbastanza grande** da poter contenere il processo, riservando i buchi di dimensione maggiore per i processi richiedenti più memoria.

Di controparte, tale policy potrebbe portare alla creazione di buchi di dimensione troppo piccola per poter essere allocata da qualsiasi processo, rendendo tali spazi di memoria completamente inutilizzabili.

Tale policy viene detta **best-fit**.

Esempio:

- Supponiamo di avere tre buchi in memoria, rispettivamente di 20, 400 e 120 byte.
- Supponiamo che il processo X richieda 100 byte di memoria. Tramite la policy best-fit, il processo verrebbe inserito nel buco minore sufficientemente grande, ossia quello da 120 byte, creando così un buco di 20 byte
- Supponiamo ora che il processo Y richieda 350 byte di memoria. Tale processo verrà allocato nel buco da 400 byte, creandone uno di 50.
- La somma dei buchi disponibili, quindi, corrisponde a 90 byte. Tuttavia, essendo suddivisi in buchi di piccole dimensioni, difficilmente un processo sarà abbastanza piccolo da poter essere contenuto in uno di tali buchi, portando ad uno spreco di memoria

Metodo - Policy worst-fit



Prima dell'allocazione viene effettuata una scansione lineare di **tutti i buchi** presenti in memoria. Successivamente, viene selezionato come spazio d'allocazione il **maggior buco abbastanza grande** da poter contenere il processo.

Contro intuitivamente, tale policy aumenta la probabilità che i buchi rimanenti dopo un'allocazione siano abbastanza grandi da poter contenere altri processi.

Tale policy viene detta **worst-fit**.

Frammentazione interna ed esterna



Definiamo come **frammentazione della memoria** la presenza di singoli buchi di memoria troppo piccoli per essere utilizzati da un processo, ma abbastanza grandi da poter essere utilizzati nel caso in cui essi fossero combinati.

Distinguiamo due tipi di frammentazione:

-

Frammentazione esterna, dove vi è abbastanza spazio libero per poter allocare il processo, ma tale spazio è suddiviso in buchi di memoria non contigui, impedendo l'allocazione. Si verifica nel caso in cui si utilizzi un'allocazione dinamica della memoria a seguito del frequente allocamento e de-allocamento di processi.

-

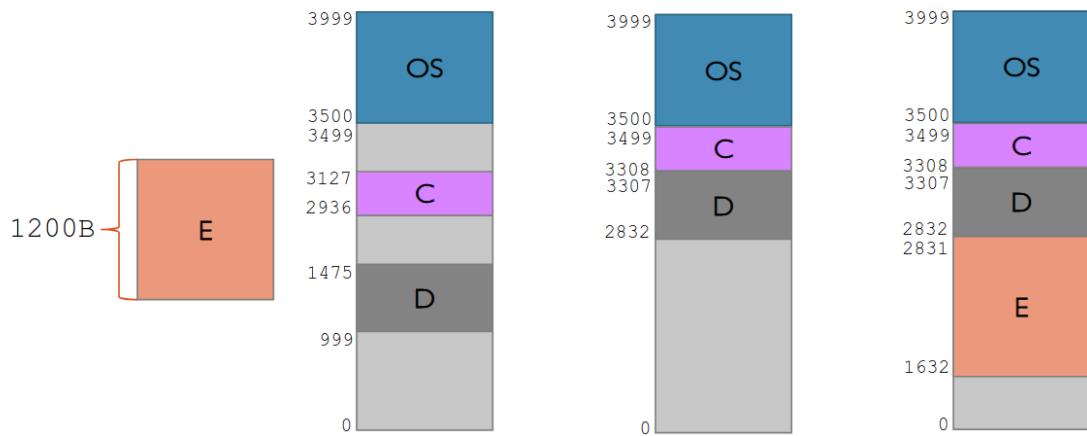
Frammentazione interna, dove viene sprecata parte di memoria a seguito dell'allocazione di settori troppo grandi per un processo richiedente meno spazio di quello fornito. Si verifica nel caso in cui si utilizzi un'allocazione statica della memoria tramite la definizione di partizioni

Risolvere la frammentazione

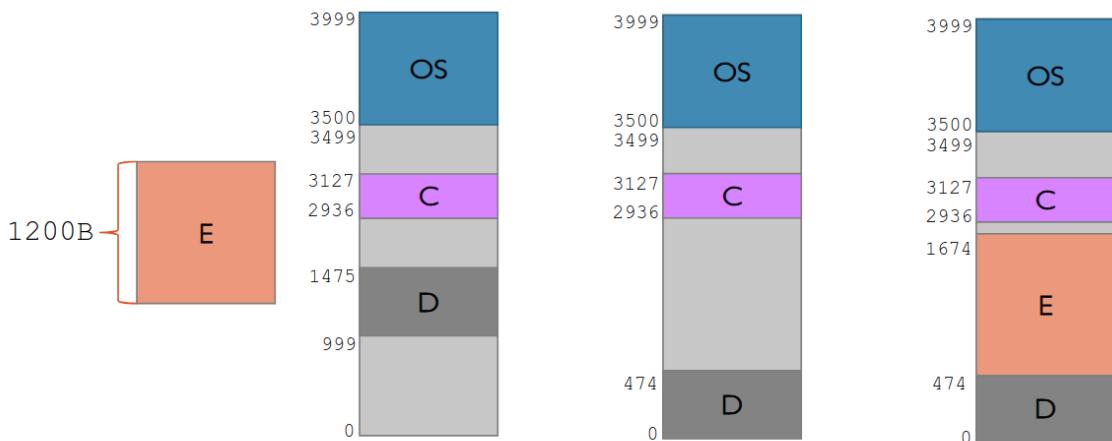
Nel caso in cui venga utilizzata un'allocazione statica dei settori, il problema della **frammentazione interna** risulta essere **irrisolvibile**. Tuttavia, essa può essere parzialmente mitigata creando dei settori di dimensione crescente, permettendo ai processi meno costosi di essere allocati nei settori più piccoli, diminuendo la quantità di memoria sprecata.

Nel caso in cui venga utilizzata un'allocazione dinamica, invece, il problema della **frammentazione esterna** può essere **risolto in due modi**:

- **Compattazione totale**, dove tutti i processi vengono rilocati in modo contiguo agli altri, creando un singolo buco di memoria



- **Compattazione parziale**, dove vengono rilocati solo i processi necessari affinché si crei un buco abbastanza grande per contenere un processo da allocare.



Swapping e paginazione

Metodo - Swapping



Per essere eseguito dalla CPU, un processo necessita di essere caricato in memoria principale.

Tuttavia, nel caso in cui processo si blocchi in attesa che si verifichi un evento, non è necessario che esso si trovi in memoria principale, dunque, è possibile

spostare tale processo dalla memoria primaria a quella secondaria. Una volta che l'evento si verifica, il processo è nuovamente pronto ad essere eseguito, venendo nuovamente spostato in memoria principale.

Tale tecnica viene detta
swapping.

Per via dell'interazione con la memoria secondaria, rispetto alle comuni operazioni lo swapping risulta essere un processo **molto lento**. Tuttavia, esso permette di poter risolvere in modo efficiente la **frammentazione** eseguendo una compattazione prima di spostare il processo in memoria principale.

La

modalità di gestione dello swapping deriva direttamente dalla modalità di gestione dell'associazione degli indirizzi:

- Se viene utilizzata un'associazione in fase di **compilazione** o di **avvio**, il processo deve essere spostato nella stessa identica posizione di memoria in cui si trovava precedentemente
- Nel caso in cui venga utilizzata un'associazione in fase di **esecuzione**, il processo può essere spostato in qualsiasi posizione della memoria

Per via della sua lentezza, i moderni sistemi operativi non utilizzano più la tecnica dello swapping, preferendo invece alternative più rapide, come la **paginazione**.

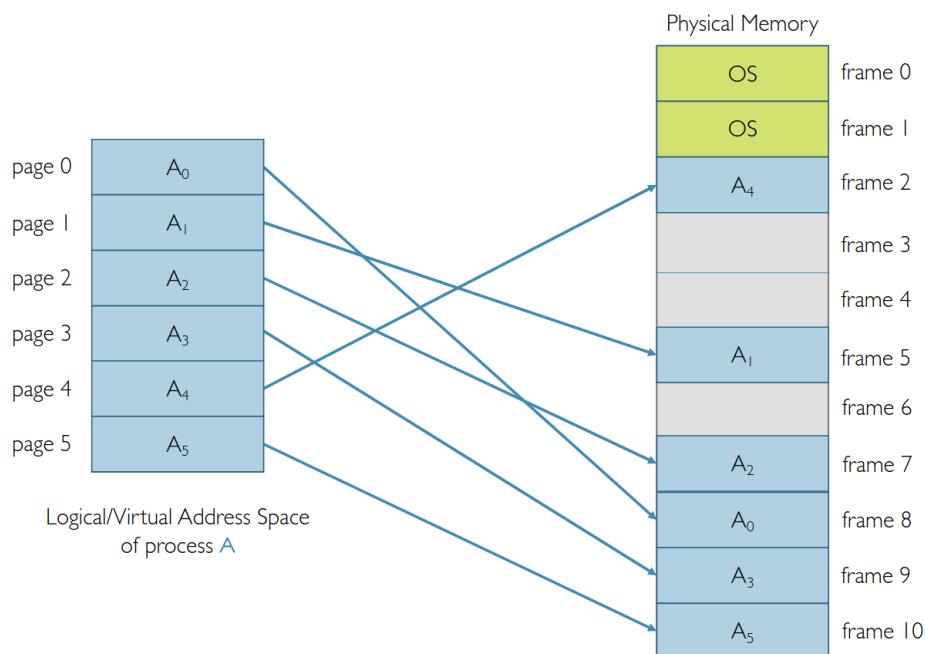
Metodo - Paginazione

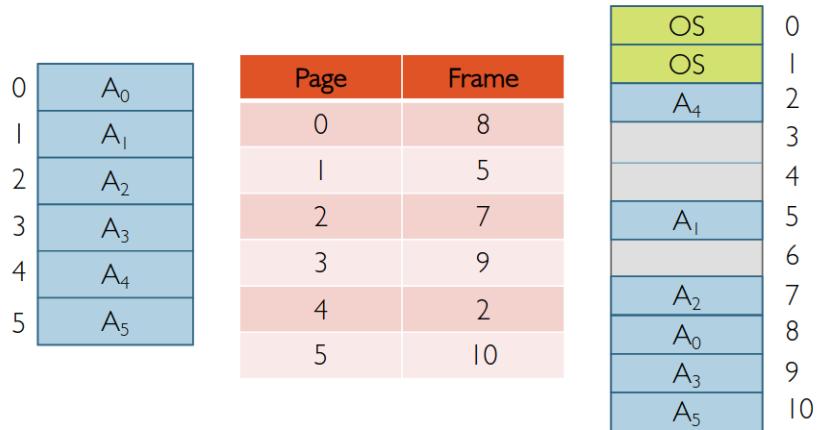


La tecnica della **paginazione** prevede l'uso di uno spazio di indirizzamento logico contiguo ma suddiviso in **blocchi di dimensione fissa**, detti **pagine**, rendendo non più necessario che gli indirizzi fisici siano contigui, poiché le pagine possono essere mappate a **frame fisici non contigui**. Inoltre, viene **eliminata** del tutto la frammentazione esterna.

Il sistema operativo, dunque, dovrà occuparsi di effettuare la mappatura tra le pagine logiche e i frame fisici e di tradurre gli indirizzi logici in indirizzi fisici. Per fare ciò, l'OS necessita di una **page table**.

Esempio:



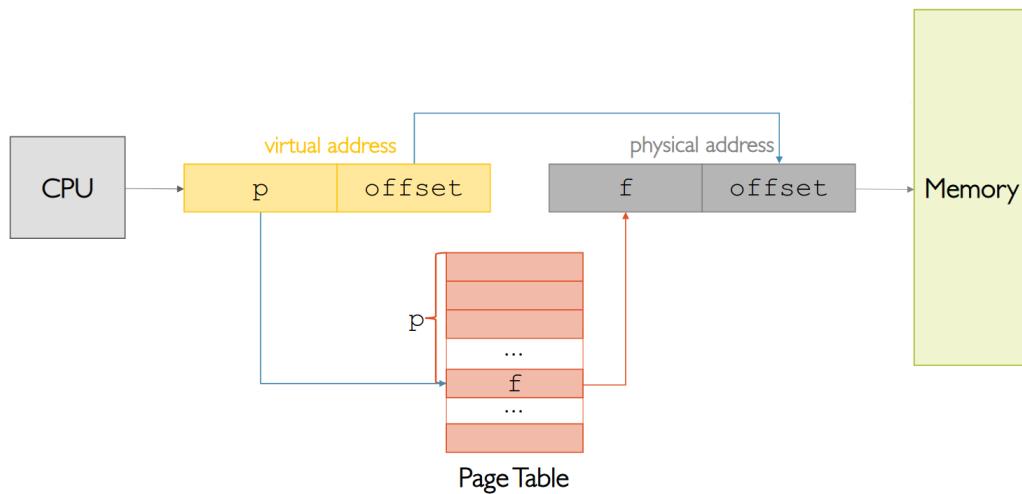


Ogni **indirizzo virtuale** è costituito da un **numero di pagina** p dove gli indirizzi risiedono ed un **offset di word** relativo all'inizio della pagina.

Analogamente, ogni

indirizzo fisico è costituito da un **numero di frame** f e un **offset di word** relativo all'inizio del frame.

Durante la conversione da indirizzo virtuale a fisico, il numero di frame f dell'indirizzo fisico corrisponderà a quello indicato dalla pagina p presente all'interno della **page table**, mentre l'offset corrisponderà all'offset virtuale stesso.



La paginazione, dunque, corrisponde semplicemente ad una forma di **rilocazione dinamica** dove ogni indirizzo virtuale è legato ad un indirizzo fisico tramite la page

table, la quale può essere interpretata come un insieme di registri base (uno per ogni frame).

Esempi:

1

- Supponiamo che lo spazio di memoria riservato ai processi utente sia $M = 50B$ e che la dimensione di ogni pagina/frame sia $S = 10B$.
- Consideriamo l'indirizzo virtuale $x = 27$. Il page number sarà $p = \lfloor \frac{x}{S} \rfloor = \lfloor \frac{27}{10} \rfloor = 2$, mentre l'offset di word sarà $o = 27 \bmod 10 = 7$
- Dunque, l'indirizzo fisico sarà costituito dal numero di frame contenuto nella seconda pagina della page table e un offset pari a 7

2

- Supponiamo di avere una memoria virtuale e una memoria fisica entrambe di dimensione $M = 1024B$ dove il word size è $W = 2B$. Supponiamo inoltre di utilizzare un paginazione con pagine/frame di dimensione $S = 16B$.
- Il numero di pagine/frame contenute page table sarà $T = \frac{M}{S} = \frac{1024}{16} = 64$ dunque saranno necessari $p = 6$ bit per poter indicizzare ogni pagina della page table ($2^6 = 64$)
- Il numero di word contenute in ogni frame sarà $F = \frac{S}{W} = \frac{16}{2} = 8$ dunque saranno necessari $o = 3$ bit per poter indicizzare il word offset ($2^3 = 8$)
- Dei 10 bit costituenti un indirizzo fisico, quindi, 6 saranno utilizzati per il numero di pagina, 3 per l'offset di word e il restante bit per l'offset di byte

▼ Lezione del 05/12

Translation Look-aside Buffer (TLB)



Una **Translation Look-aside Buffer (TLB)** è una cache di primo livello molto veloce utilizzata per memorizzare i numeri di page e di frame calcolati precedentemente, come se fosse una sorta di "barra dei segnalibri" contenente i numeri calcolati più frequentemente o più vicini a numeri già calcolati.

Poiché le cache sono molto più veloci di una memoria normale, ma anche più piccole di essa, la TLB conterrà solo una parte di valori contenuti nella page table.

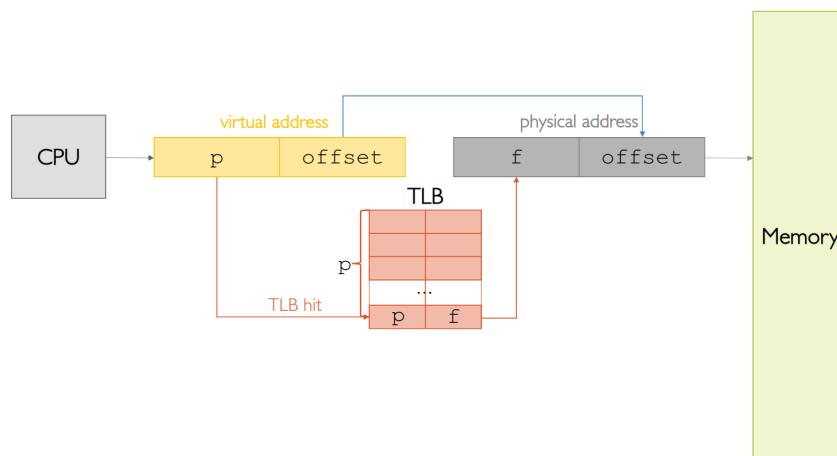
Nel caso in cui venga richiesto un indirizzo virtuale, verrà prima controllato se il numero di pagina associato a tale indirizzo sia già caricato nella TLB.

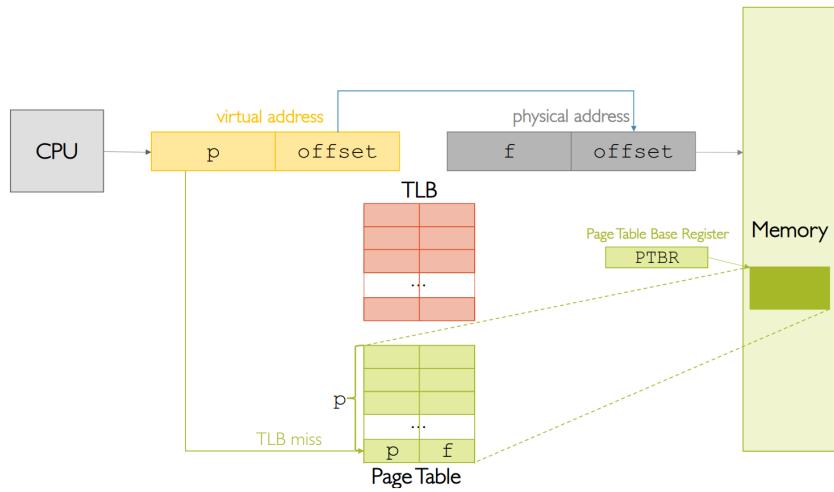
In caso affermativo (

TLB hit), il frame associato verrà **subito restituito**. In caso negativo (**TLB miss**), il controllo verrà effettuato come di norma sulla **page table**, per poi caricare il dato letto all'interno della TLB per futuri utilizzi.

La TLB è condivisa tra

tutti i processi. Per questo motivo, lo stesso numero di pagina può essere mappato ad un **differente numero di frame** in base al processo richiedente.





Segmentazione

Durante lo sviluppo di applicazioni, per i programmatori risulta più comodo immaginare i propri programmi come se fossero salvati in memoria in modo **non contiguo**. In particolare, risulta comodo immaginare la memoria occupata da ogni programma come se fosse suddivisa in segmenti, ognuno di essi dedicato per uno specifico uso (dati, codice, ...).

La tecnica di

segmentazione della memoria facilita tale modalità di ragionamento, basandosi su indirizzi costituiti da un **numero di segmento** ed un **offset** dall'inizio del segmento indicato.

Segment table

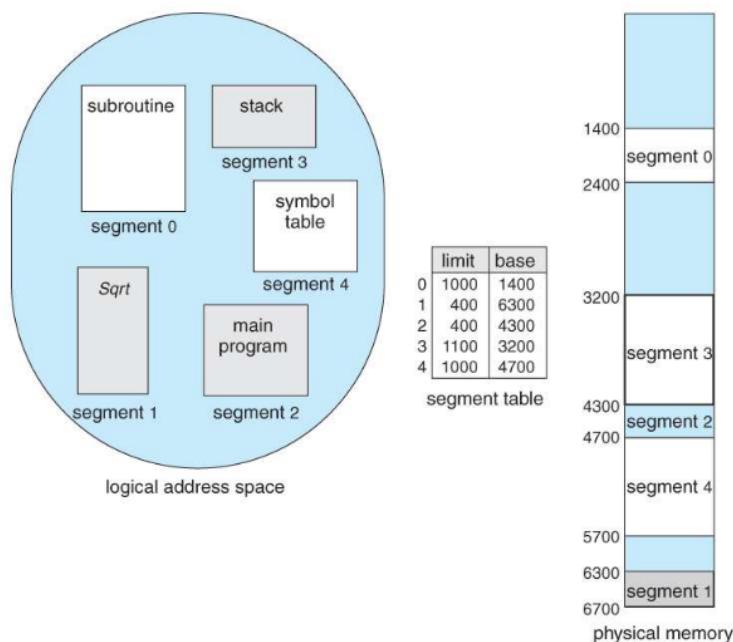


Una **segment table** collega ogni **indirizzo segmento-offset** ad un indirizzo fisico, controllando simultaneamente se tale indirizzo sia invalido o no. Il sistema utilizzato per la segment table risulta molto simile al funzionamento della page table e dei registri base di rilocazione della MMU.

Ogni

entrata della segment table contiene un **indirizzo base** del segmento, la **lunghezza** di tale segmento e **informazioni aggiuntive** per la protezione di esso (ad esempio permessi per la lettura o scrittura).

A differenza della page table (la quale potrebbe contenere troppe pagine), la segment table può essere realizzata tramite pochi registri base e limite

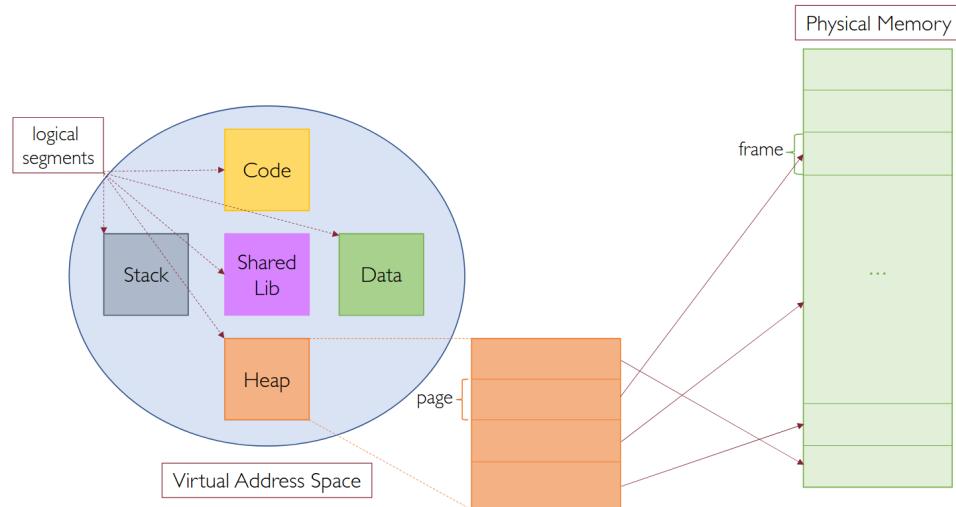


A differenza della paginazione, la segmentazione risulta essere più comoda per la condivisione di dati, rendendo tuttavia inefficiente l'uso della memoria. La soluzione ottimale, quindi, risulta essere quella di applicare la paginazione ai segmenti stessi. Tale tecnica viene detta paginazione segmentata:

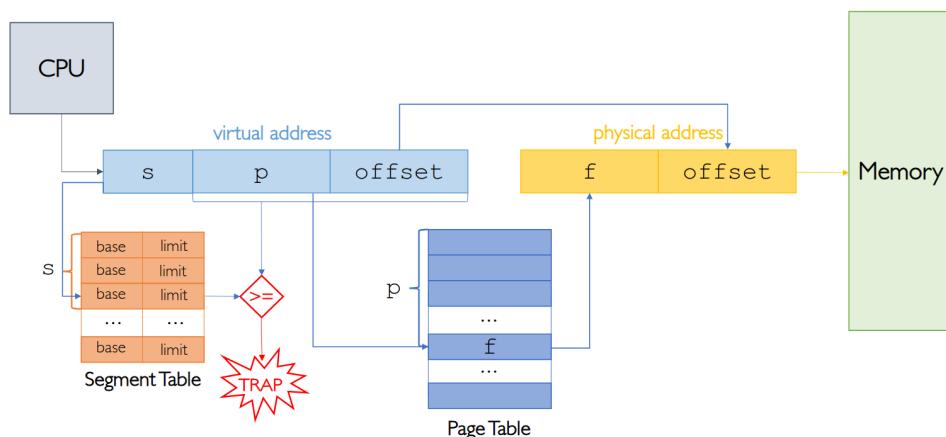
- Lo spazio di indirizzamento virtuale di ogni processo viene visto come una collezione di segmenti, mentre lo spazio di indirizzamento fisico viene visto

come una sequenza di frame di dimensione fissa. Di conseguenza, ogni processo sarà dotato di una propria segment table.

- I segmenti sono generalmente più grandi dei frame fisici, necessitando che ogni segmento logico debba essere mappato a più frame. Di conseguenza, ogni segmento sarà dotato di una propria page table.



- Affinché la traduzione da indirizzo virtuale a fisico sia possibile, quindi, ogni indirizzo deve essere composto da un numero di segmento, un numero di pagina legato alla page table del segmento specifico ed un offset dall'inizio del frame indicato nella page table



- Per tenere traccia delle segment table e delle page table dei vari processi, vengono utilizzate due soluzioni:

- Vengono utilizzati pochi registri per le segment table, mentre le page table vengono salvate nella memoria principale con aggiunta di una TLB. Tale soluzione risulta più veloce, ponendo tuttavia un limite al numero di segmenti
- Sia le segment table sia le page table vengono salvate nella memoria principale con aggiunta di una TLB, i cui controlli vengono effettuati utilizzando un indice dei segmenti ed un indice di pagine. Tale soluzione risulta essere più lenta ma più flessibile.
- Pro:
 - Il compilatore e l'OS vedono la memoria in modo analogo
 - Flessibilità
 - Assenza di frammentazione esterna
 - Condivisione della memoria tra i processi
- Contro
 - Context switch e traduzione degli indirizzi più lenti
 - Frammentazione interna ancora esistente

Oltre alla paginazione segmentata, i moderni sistemi operativi sono dotati di una forme più avanzate di paginazione:

- **Paginazione a due livelli**, dove una page table viene suddivisa in più page table (dunque si tratta di una paginazione della page table)
- **Paginazione con hash**, dove vengono utilizzate delle hash table per indicizzare più page table sparse in modo veloce
- **Page table invertita**, dove vengono conservati i numeri di tutti frame attualmente carichi in memoria

aggiungi esercizi slide 44 lezione del 7/12/2023

▼ Lezione del 12/12

Memoria Virtuale

Metodo - Memoria Virtuale



Poiché la maggior parte dei processi non necessita che tutte le proprie pagine siano caricate in memoria contemporaneamente, le **pagine non utilizzate** possono essere **archiviate temporaneamente** all'interno della memoria secondaria, dando l'illusione di uno spazio di indirizzamento virtuale di dimensione infinita, permettendo a più programmi di poter essere attivi, aumentando l'utilizzo della CPU.

In ogni momento, quindi, ogni pagina potrebbe trovarsi nella memoria principale o nella memoria secondaria, ossia nel disco rigido.

Tale tecnica viene detta
memoria virtuale

Alla base della memoria virtuale quindi, vi è la seguente idea:

- La memoria principale assume il ruolo di "cache" del disco rigido
- Per ogni pagina la page table deve indicare se tale pagina si trovi nella memoria o nel disco (un **singolo bit di appartenenza** è sufficiente). Ogni volta che una pagina viene spostata dal disco alla memoria, e viceversa, l'OS si occuperà di aggiornare l'entrata corrispondente della page table, modificandone il bit indicante la memoria di appartenenza.
- Ogni volta che viene effettuato un riferimento logico, viene eseguito un controllo nella page table. Se il bit di appartenenza è impostato su 1, allora la pagina richiesta si troverà nella memoria principale. Altrimenti, verrà attivata una **page fault trap**, imponendo che la pagina venga caricata dal disco alla memoria.
- Poiché l'accesso al disco è molto più lento dell'accesso alla memoria, gli accessi alla memoria devono fare riferimento a **pagine che con alta probabilità si trovano nella memoria**, evitando di dover effettuare un accesso al disco
- Tramite la **regola del 90/10**, ossia la regola affermante che un processo spende **il 90% del tempo accedendo solo al 10% della sua memoria allocata**, sappiamo che la maggior parte dei riferimenti alla memoria effettuati da un processo si

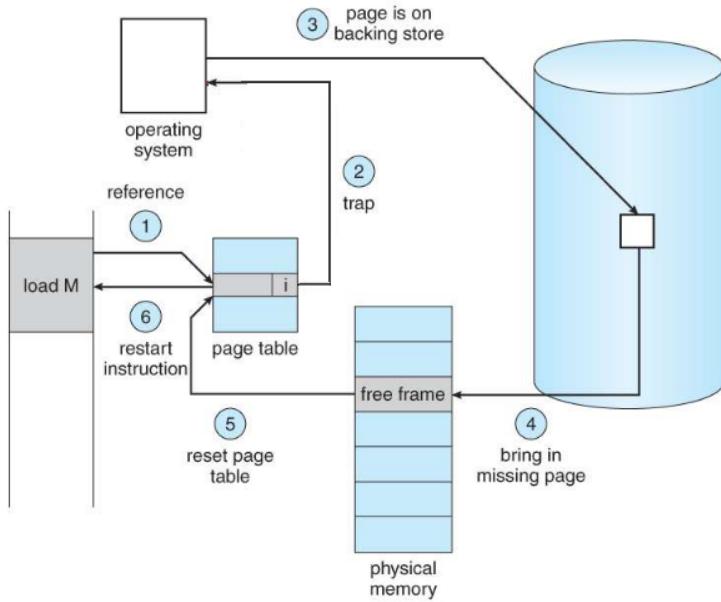
trova in una **piccola zona di memoria**, che definiamo come *working set* del processo.

- Siccome tale zona è molto piccola, la probabilità che essa possa essere caricata in memoria è molto alta. Ovviamente, durante l'esecuzione di un processo, il working set potrebbe variare. Tuttavia, tale variazioni avvengono con **frequenza bassa**, dunque per un determinato lasso di tempo il working set rimarrà lo stesso.

Gestione dei page fault

Nel caso in cui si verifichi un **page fault**, verranno eseguite le seguenti operazioni:

1. Viene controllato nella page table il bit di appartenenza della pagina corrispondente all'indirizzo di memoria richiesto
2. Nel caso in cui sia impostato su 0, viene attivata la **page fault trap**, dunque la pagina deve essere recuperata dal disco
3. Viene individuato un **frame libero** nella memoria principale a cui poter mappare la pagina recuperata dal disco
4. Viene schedulata un'**operazione sul disco** per poter spostare la pagina richiesta, dunque mettendo il processo in attesa di un I/O e permettendo ad un altro processo di essere schedulato
5. Una volta completata l'operazione di I/O, viene **aggiornata** l'entrata della page table con il numero del frame individuato precedentemente e il suo bit di appartenenza viene impostato su 1
6. Infine, il processo attualmente in esecuzione viene interrotto, **riprendendo l'esecuzione** del processo precedente e rieseguendo l'istruzione tramite cui è stato attivato il page fault



Nel caso in cui venga utilizzata anche una TLB, anche essa dovrà essere dotata di un **bit di appartenenza**, valente **1** se la pagina si trova in memoria principale e **0** se si trova nel disco. Il funzionamento della TLB in tal caso risulta essere leggermente diverso:

- Ottenere un **TLB hit** nel caso della memoria virtuale implica che la pagina richiesta si trovi nella cache e che il frame di riferimento sia in memoria
- Se si verifica un **TLB hit** ma il frame di riferimento **non è attualmente caricato in memoria**, sarà necessario recuperare la pagina dal disco.
- Se la pagina richiesta **non è nella cache (TLB miss) ma è in memoria**, l'OS sceglierà un'entrata della TLB da rimpiazzare con la nuova entrata corrispondente alla pagina richiesta
- Se la pagina richiesta **non è nella cache (TLB miss) e non è neanche in memoria**, l'OS sceglierà un'entrata da invalidare, per poi effettuare le operazioni legate alla page fault trap, aggiornando l'entrata della TLB con i dati corrispondenti alla pagina recuperata. Infine, verrà rieseguita l'istruzione generante il miss.

Come Architettura degli Elaboratori

Istruzione idempotente



Un'istruzione viene detta **idempotente** se essa può essere eseguita multiple volte ottenendo lo stesso effetto di una singola esecuzione. Nel caso in cui si verifichi un page fault a seguito dell'esecuzione di un'istruzione idempotente, allora tale istruzione verrà semplicemente rieseguita a seguito della gestione del page fault. Nel caso in cui l'istruzione non sia idempotente, la riesecuzione di tale istruzione risulta molto più difficile.

Teoreticamente, un page fault potrebbe verificarsi a seguito di ogni istruzione eseguita.

Tuttavia, i processi sono soggetti al **principio di località dei riferimenti**, dove se un processo accede ad un indirizzo in memoria, è molto probabile che tale processo accederà nuovamente allo stesso indirizzo (**principio temporale**) e che acceda ad indirizzi molto vicini ad esso (**principio spaziale**).

In particolare, considerando:

- t_{MA} , ossia il tempo impiegato ad accedere alla memoria fisica
- t_{FAULT} , ossia il tempo impiegato a gestire un page fault
- $p \in [0, 1]$, ossia la probabilità che si verifichi un page fault

Il **tempo totale impiegato** per un accesso alla memoria corrisponde a:

$$t_{ACCESS} = (1 - p) * t_{MA} + p * t_{FAULT}$$

Esempio:

- Supponiamo che $t_{MA} = 100\text{ns}$, $t_{FAULT} = 20\text{ms} = 20000000\text{ns}$ e che la probabilità di un page fault sia $p = 0.0001$. In tal caso si ha che:

$$t_{ACCESS} = (1 - \frac{1}{10000}) \cdot 100\text{ns} + \frac{1}{10000} \cdot 20000000\text{ns} = \frac{9999}{100}\text{ns} + 2000\text{ns} \approx 20.1\mu\text{sec}$$

- Nel caso in cui volessimo ottenere un tempo di accesso al massimo il 10% più lento di un normale accesso in memoria, ci basterebbe risolvere la seguente equazione:

$$1.1 \cdot 100 = (1 - p) \cdot 100 + p \cdot 20000000 \iff p = \frac{1}{1999990} \approx 5 \cdot 10^{-7}$$

Di conseguenza, possiamo tollerare un massimo di un page fault ogni circa 2 milioni di accessi

Più generalmente, dati t_{MA}, t_{FAULT} e una soglia $\epsilon > 0$ tale che

$$t_{ACCESS} = (1 + \epsilon) * t_{MA}$$

per trovare il **numero di page fault tollerabili** sarà $\frac{1}{p}$, dove si ha che:

$$\begin{aligned} t_{ACCESS} = (1 + \epsilon) \cdot t_{MA} &\iff (1 - p) \cdot t_{MA} + p \cdot t_{FAULT} = (1 + \epsilon) \cdot t_{MA} \iff \\ &\iff p(t_{FAULT} - t_{MA}) = \epsilon \cdot t_{MA} \iff p = \frac{\epsilon \cdot t_{MA}}{t_{FAULT} - t_{MA}} \end{aligned}$$

▼ Lezione del 14/12

Recupero e rimpiazzo delle pagine

Principalmente, esistono tre strategie per poter recuperare le pagine dal disco:

- **Recupero all'avvio**, dove tutte le pagine del processo vengono caricate contemporaneamente
- **Recupero ad overlay**, dove è il programmatore a scegliere quando e quali pagine da caricare
- **Recupero a richiesta**, dove un processo comunica all'OS quando necessita che una pagina venga recuperata.

Il **recupero a richiesta** è il sistema utilizzato da tutti gli OS moderni. Quando un processo viene avviato, nessuna delle sue pagine è carica in memoria, venendo spostate solamente quando il processo richiede di accedervi (dunque tramite i page

fault). Tale sistema di recupero delle pagine viene detto **lazy swapper o pager**. Un’ulteriore tecnica utilizzata per il recupero delle pagine è il **prefetching**, dove il pager predice quando le pagine verranno richieste, caricandole in anticipo. Tale tecnica, tuttavia, risulta prettamente impraticabile, poiché predire le future pagine risulta molto difficile.

Un possibile approccio consiste nel caricare più di una singola pagina durante un page fault (efficace solo se il programma accede sequenzialmente alla memoria).

Nel caso in cui debba essere caricata una pagina dal disco ma la memoria fisica sia piena, è necessario rimpiazzare una delle pagine caricate. Gli algoritmi più utilizzati per tale compito sono i seguenti:

- **Rimpiazzo casuale**, dove viene rimossa una pagina a caso
- **FIFO (First in, First out)**, dove viene rimossa la pagina rimasta in memoria per il tempo maggiore
- **MIN (OPT)**, dove viene rimossa la pagina che non verrà acceduta per il lasso di tempo maggiore (richiede di predire il futuro, dunque è un algoritmo ottimale ma molto difficile da implementare)
- **LRU (Least Recently Used)**, dove viene rimossa la pagina non utilizzata per il maggior lasso di tempo. È l’algoritmo più utilizzato dagli OS moderni e può essere implementato in tre modi:
 - Tramite un **singolo bit di riferimento**, dove viene mantenuto un singolo bit per ogni entrata della page table, il quale viene impostato a 0 inizialmente (dunque anche dopo ogni pulizia della page table) e viene impostato ad 1 nel caso in cui venga fatto un accesso alla pagina associata.
Durante la rimozione, viene scartata una delle pagine impostate a **0**.
 - Tramite **più bit di riferimento**, dove vengono utilizzati 8 bit per ogni entrata della page table, i quali vengono shiftati di un bit ad ogni colpo di clock.
Dopo ogni accesso alla pagina associata viene impostato ad 1 il bit più a sinistra.
Durante la rimozione, viene scartata la pagina avente il più basso valore negli **8 bit**.

- Tramite l'**algoritmo di seconda chance**, dove viene utilizzato un singolo bit di riferimento ed una politica FIFO: l'OS si occupa di tenere traccia dei frame in una lista FIFO circolare e dove ad ogni accesso alla memoria viene impostato il bit di riferimento ad 1.

Durante un **page fault**, l'OS analizza l'intera lista delle page table, controllando i bit di riferimento del frame: se il bit è impostato su 0, allora la pagina viene rimpiazzata ed esso viene impostato su 1, altrimenti, esso viene impostato su 0 (seconda chance) e viene ripetuto il controllo sul frame successivo.

Una versione più

avanzata di tale algoritmo prevede l'uso di un ulteriore **bit di modifica** (se 1 allora la pagina è diversa da quella salvata sul disco, altrimenti no), classificando le pagine in quattro categorie:

1. Se i bit sono (1,1), allora la pagina è stata utilizzata recentemente ed è stata modificata
2. Se i bit sono (1,0), allora la pagina è stata utilizzata recentemente ma non è stata modificata
3. Se i bit sono (0,1), allora la pagina non è stata utilizzata recentemente ma è stata modificata
4. Se i bit sono (0,0), allora la pagina non è stata utilizzata recentemente e non è stata modificata

Durante la rimozione, viene scartata la prima pagina trovata avente categoria inferiore (dunque prima (0,0), poi (0,1), ...)

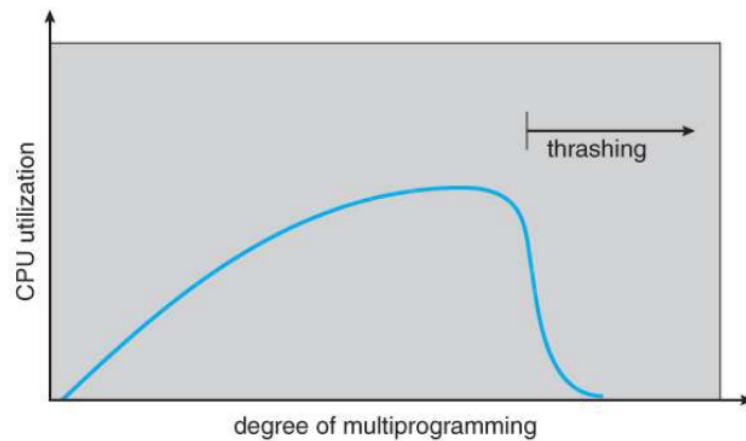
AGGIUNGERE ESEMPI SLIDE DEL 12/12 DA SLIDE 98

Trashing e determinazione del working set

Trashing



Definiamo come **trashing** il fenomeno in cui la memoria è utilizzata da **troppi processi contemporaneamente**, portando le pagine ad essere **continuamente rimpiazzate** nonostante esse siano ancora in uso dai processi stessi e dunque una forte degradazione delle performance



Metodo - Politiche di allocazione e rimpiazzo delle pagine



Per contrastare il trashing, è necessario cedere abbastanza memoria ad ogni processo, rendendo quindi fondamentali le **politiche di allocazione e rimpiazzo delle pagine**:

- **Allocazione e rimpiazzo globale**, dove tutte le pagine di ogni processo sono in una **singola pool**, ossia una queue LRU dove, durante il rimpiazzo di una pagina, anche le pagine non appartenenti allo stesso processo della nuova pagina possono essere rimpiazzate, portando ad una maggiore flessibilità ma anche ad una maggiore quantità di trashing
-
- **Allocazione e rimpiazzo locale**, dove ogni processo possiede la **propria pool**, dunque il rimpiazzo LRU controllerà solo le pagine appartenenti allo stesso processo della nuova pagina, portando ad un totale isolamento dei processi ma anche ad una minore performance
-
- **Allocazione e rimpiazzo proporzionale**, dove viene assunto che **ogni processo di grandi dimensioni** faccia riferimento anche ad una **grande quantità di memoria**. Tuttavia, in alcuni casi tale assunzione si rivela errata (ad esempio, un processo potrebbe allocare un array di 1 GB, per poi utilizzarne solo una piccola porzione). In altre parole, non sempre il **working set** di un processo è correlato al suo impatto sulla memoria.

Working set



Informalmente, il **working set** di un processo è l'insieme delle pagine che il processo sta attualmente utilizzando.

Formalmente, esso corrisponde all'insieme di **tutte le pagine** a cui esso ha effettuato riferimenti durante le **ultime T unità di tempo**.

La scelta della grandezza della **finestra di riferimenti** di cui tenere traccia, indicata con Δ , risulta fondamentale: una finestra troppo piccola non racchiude tutte le pagine della località attualmente richiesta dal processo, mentre una finestra troppo grande include pagine che non sono più accedute frequentemente.

Per evitare di dover gestire una lista delle ultime

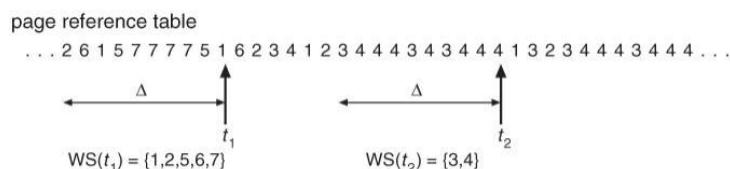
Δ pagine richieste, il working set

viene spesso implementato tramite una semplice

campionatura: ogni k riferimenti in memoria, il working set di un processo viene considerato l'insieme di tutte le pagine a cui è stato fatto riferimento in tale periodo temporale.

L'obiettivo, quindi, è quello di riuscire a dare ad ogni processo

abbastanza frame da poter contenere il proprio working set.



▼ Lezione del 19/12

Storage Management

Dischi magneti



Un **disco magnetico** è una **memoria secondaria** composta da uno o più **piatti** ricoperti di materiale magnetico (metallo rigido per gli HDD, plastica flessibile per i floppy disk), dove:

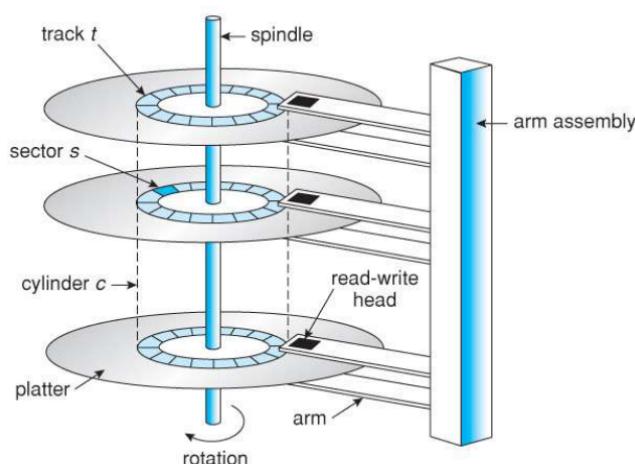
- Ogni piatto possiede **due superfici funzionali**, ossia il fronte e il retro, entrambe suddivise in un numero di anelli concentrici, detti tracce. L'insieme di tutte le tracce di ogni piatto poste alla stessa distanza dal bordo viene detto cilindro.
- Ogni traccia è a sua volta suddivisa in **settori** (solitamente da 512 byte ciascuno), ognuno di essi contenente un'intestazione, un trailer e informazioni di verifica. Una dimensione maggiore per i settori riduce la quantità di spazio sprecata per contenere le intestazioni e i trailer, ma aumenta la frammentazione interna
- Ogni superficie viene letta da una **testina di lettura e scrittura** agganciata alla fine di un **braccio**, a sua volta connesso ad un **attuatore**, il quale muove tutti i bracci simultaneamente da una traccia all'altra

La

capacità totale di memoria di un disco rigido corrisponde a:

$$C = H \cdot T \cdot S \cdot B$$

dove H è il numero di superfici, T il numero di tracce per superficie, S il numero di settori per traccia e B il numero di byte per settore.



Il trasferimento di informazioni dal disco alla memoria è composto da 3 step:

- **Positioning time:**

Il tempo necessario per spostare le testine su una traccia/cilindro specifico

- Comprende il tempo necessario per far posare le testine
- Dipende da quanto velocemente l'hardware muove il braccio
- Tipicamente, è il passo più lento nell'intero processo

- **Rotational delay:**

Il tempo necessario affinché il settore desiderato compia una rotazione e si trovi sotto la testina di lettura-scrittura

- Può variare da 0 fino a una rotazione completa
- 0 à il settore è già sotto la testina
- rotazione completa à il settore è quello precedente, ma nella direzione opposta
- In media, 0,5 rotazioni (r)

Ad esempio, per un disco a 7200 giri al minuto (120 giri al secondo), ciò equivale a 0,5 r/120 giri al secondo ~4 msec

- **Trasfer time:**

Il tempo necessario per spostare i dati (cioè, byte) elettronicamente dal disco alla memoria. Talvolta è espresso come tasso di trasferimento (larghezza di banda) in byte al secondo

La CPU è in grado di interfacciarsi con i dischi magnetici tramite un **bus I/O**, un canale di comunicazione dove la CPU stessa impedisce comandi al disco tramite un controller posto alla fine del bus, ossia **l'host controller**, il quale a sua volta comunica con il **disk controller** interno al disco stesso, trasferendo i dati dalle superfici magnetiche del disco prima ad una cache interna al disco, per poi venir trasferiti alla memoria.

Per via della struttura di un disco magnetico, il

tempo di trasferimento dei dati alla memoria principale, indicato come DTT, equivale alla somma tra:

- **Tempo di posizionamento (Seek time)**, indicato come *ST*, ossia il tempo necessario per spostare le testine su una specifica traccia.
- **Delay di rotazione**, indicato come *ROT*, ossia il tempo necessario affinché il settore desiderato venga ruotato sotto la testina (in media viene effettuata mezza rotazione del disco).
- **Tempo di trasferimento**, indicato come *TT*, ossia il tempo necessario per spostare i dati dal disco alla memoria, spesso espresso come rateo di trasferimento (larghezza di banda), ossia come byte al secondo.

$$\text{DTT} = \text{ST} + \text{ROT} + \text{TT}$$

Rischi del disco magnetico

Le testine del disco "volano" sulla superficie su uno strato molto sottile d'aria
Se entrano accidentalmente in contatto con il disco, si verifica un head crash
Un head crash può danneggiare permanentemente il disco o addirittura distruggerlo
Per evitare tale rischio, le testine vengono "parcheggiate" quando il computer è spento

▼ Lezione del 21/12

Scheduling del disco

Poiché il sistema operativo riceve costanti richieste di lettura e scrittura da parte dei processi, è necessario gestire in modo efficiente gli **accessi al disco**, in particolare cercando di ridurre al minimo il seek time. Dunque, al fine di ridurre il seek time, è necessario **ridurre al minimo la distanza percorsa** dalla testina di lettura e scrittura.

Metodo - First Come First Served (FCFS)



L'algoritmo **First Come First Served (FCFS)** è un algoritmo di scheduling del disco in cui le richieste di accesso al disco vengono effettuate nel loro ordine di arrivo nella coda di richieste.

Pro:

- Facile da implementare
- Utilizzato anche dagli SSD poiché in tali tipologie di memorie non è richiesto alcun movimento meccanico, dunque il seek time è nullo

Contro:

- Poco performante nel caso in cui il sistema abbia poche richieste

Esempio:

- Supponiamo di effettuare le richieste 65, 40, 18, 78 su un disco in cui la testina è inizialmente posizionata sulla traccia 30 utilizzando l'algoritmo FCFS:
 - La testina viene mossa da 30 a 65, dunque vengono traversate 35 tracce
 - La testina viene mossa da 65 a 40, dunque vengono traversate 25 tracce
 - La testina viene mossa da 40 a 18, dunque vengono traversate 22 tracce
 - La testina viene mossa da 18 a 78, dunque vengono traversate 60 tracce
- La distanza percorsa, ossia il numero totale di tracce traversate, corrisponde a
$$35 + 25 + 22 + 60 = 142$$

Metodo - Shortest Seek Time First (SSTF)



L'algoritmo **Shortest Seek Time First (SSTF)** è un algoritmo di scheduling del disco in cui viene eseguita la richiesta di accesso al disco richiedente la **minore quantità di tracce da traversare**.

Pro:

- Può essere implementato mantenendo una semplice lista ordinata delle richieste
- Complessità non troppo elevata

Contro:

- Potrebbe causare starvation (ad esempio, nel caso in cui una richiesta necessiti di una traccia troppo distante dalla testina)
- L'ottimizzazione avviene solo a livello locale (ad esempio, l'esecuzione della richiesta più vicina alla testina potrebbe portare ad una catena di spostamenti peggiore rispetto al caso in cui si vada a selezionare un'altra richiesta)

Esempio:

- Supponiamo di effettuare le richieste 65, 40, 18, 78 su un disco in cui la testina è inizialmente posizionata sulla traccia 30 utilizzando l'algoritmo SSTF:
 - La testina viene mossa da 30 a 40, dunque vengono traversate 10 tracce
 - La testina viene mossa da 40 a 18, dunque vengono traversate 22 tracce
 - La testina viene mossa da 18 a 65, dunque vengono traversate 47 tracce
 - La testina viene mossa da 65 a 78, dunque vengono traversate 13 tracce
- La distanza percorsa, ossia il numero totale di tracce traversate, corrisponde a
$$10 + 22 + 47 + 13 = 92$$

Metodo - SCAN e LOOK



L'algoritmo **SCAN** è un algoritmo di scheduling del disco in cui la testina viene mossa **da un estremo all'altro del piatto** (ad esempio dalla traccia 0 alla traccia 100 e viceversa), dove le richieste vengono **eseguite durante il passaggio della testina**.

L'algoritmo

LOOK corrisponde ad una **versione ottimizzata** dello SCAN, dove durante ogni scansione la testina, piuttosto che proseguire fino all'estremo del piatto, viene fermata alla richiesta **più vicina all'estremo**.

Esempio:

- Supponiamo di effettuare le richieste 65, 40, 18, 78 su un disco avente tracce numerate da 0 a 100 in cui la testina è inizialmente posizionata sulla traccia 30 utilizzando l'algoritmo SCAN:
 - La testina viene mossa da 30 a 18, dunque vengono traversate 12 tracce
 - La testina viene mossa da 18 a 0 per poter raggiungere l'estremo, dunque vengono traversate 18 tracce
 - La testina viene mossa da 0 a 40, poi da 40 a 65 e infine da 65 a 78, dunque vengono traversate 78 tracce
- La distanza percorsa, ossia il numero totale di tracce traversate, corrisponde a

$$12 + 18 + 78 = 108$$

- Se si fosse utilizzato l'algoritmo SCAN ottimizzato, ossia l'algoritmo LOOK, la distanza percorsa corrisponderebbe a:

$$12 + 78 = 90$$

Metodo - C-SCAN e C-LOOK



L'algoritmo **C-SCAN** (o **Circular SCAN**) è un algoritmo di scheduling del disco simile all'algoritmo SCAN in cui la testina viene mossa **dall'estremo finale fino all'estremo iniziale** (ad esempio dalla traccia 100 alla traccia 0), venendo subito dopo "resettata" fino all'estremo finale (venendo spostata nuovamente sulla traccia 100). Le richieste vengono effettuate **solo durante lo spostamento dall'estremo finale fino a quello iniziale**.

L'algoritmo

C-LOOK (o **Circular LOOK**) corrisponde ad una **versione ottimizzata del C-SCAN**, dove durante ogni scansione la testina, piuttosto che proseguire fino all'estremo iniziale del piatto, viene fermata alla richiesta **più vicina all'estremo iniziale**, per poi venire resettata fino alla richiesta più vicina all'estremo finale, piuttosto che essere resettata all'estremo finale stesso.

Esempio:

- Supponiamo di effettuare le richieste 65, 40, 18, 78 su un disco avente tracce numerate da 0 a 100 in cui la testina è inizialmente posizionata sulla traccia 30 utilizzando l'algoritmo C-SCAN:
 - La testina viene mossa da 30 a 18, dunque vengono traversate 12 tracce
 - La testina viene mossa da 18 a 0 per poter raggiungere l'estremo iniziale, dunque vengono traversate 18 tracce
 - La testina viene resettata venendo spostata da 0 a 100, dunque vengono traversate 100 tracce
 - La testina viene mossa da 100 a 78, poi da 78 a 65 e infine da 65 a 40, dunque vengono traversate 60 tracce
- La distanza percorsa, ossia il numero totale di tracce traversate, corrisponde a
$$12 + 18 + 100 + 60 = 190$$
- Utilizzando l'algoritmo C-SCAN ottimizzato, ossia l'algoritmo C-LOOK, si avrebbe che:
 - La testina viene mossa da 30 a 18, dunque vengono traversate 12 tracce

- La testina viene resettata venendo spostata da 18 a 78, dunque vengono traversate 60 tracce
- La testina viene mossa da 78 a 65 e infine da 65 a 40, dunque vengono traversate 38 tracce
- La distanza percorsa, ossia il numero totale di tracce traversate, corrisponde a

$$12 + 60 + 38 = 110$$

Per capire meglio gli esercizi guarda le ultime slide.

Formattazione del disco

Prima che un disco possa essere utilizzato, deve essere formattato a **basso livello**. Ciò significa posizionare tutti gli intestazioni e i piedini che contrassegnano l'inizio e la fine di ciascun settore.

Le intestazioni e i piedini

contengono i numeri lineari dei settori e codici di correzione degli errori (ECC), per scopi di rilevamento/correzione.

La

correzione degli errori (ECC) viene eseguita con ogni lettura/scrittura del disco, e se viene rilevato un danno rilevabile e recuperabile, il controller del disco gestisce autonomamente un **errore soft**.

Una volta che il disco è formattato, il passo successivo è **partizionare** il drive in una o più partizioni separate

Deve essere fatto anche se il disco viene utilizzato come una singola grande partizione, in modo che la tabella delle partizioni sia scritta all'inizio del disco.

Dopo la partizione, è necessario

formattare logicamente i filesystem.

Dischi a stato solido

I **dischi a stato solido** sono una **memoria secondaria** realizzata tramite memoria **flash** o dei chip DRAM ed una batteria a lungo termine, la quale permette a tali memorie volatili di mantenere l'informazione.

Pro:

- Non hanno **meccanismi moventi** al loro interno, risultando quindi **estremamente più veloci** rispetto ai dischi magnetici
- L'accesso ai blocchi viene effettuato in modo diretto tramite un riferimento al numero di blocco, **rimuovendo la necessità di uno scheduling**.
- Le **operazioni di lettura** sono molto **più veloci** rispetto ad un normale disco magnetico

Contro:

- Le **operazioni di scrittura** sono **molto più lente** rispetto ad un disco meccanico, poiché i blocchi non vengono realmente sovrascritti ma solo de-referenziati (ossia resi inaccessibili), rendendo più lento il ciclo di eliminazione dei dati
- Il numero di "scritture" per un blocco è **limitato**

RAID

Un **RAID (Redundant Array of Inexpensive Disks)** è costituito da un **insieme** di molti dischi magnetici di **basso costo** utilizzati come archiviazione di **grandi quantità di dati**.

Vengono dotati di una forma di **duplicazione dei dati**, detto **mirroring**, rendendoli ridondanti nei vari dischi, in modo da poter aumentare l'affidabilità e la velocità delle operazioni di lettura e scrittura, poiché è richiesta una minore quantità di tempo per ricercare i dati rispetto all'uso di pochi dischi magnetici di grandi dimensioni