

# Intro

giovedì 23 gennaio 2025 15:11

Oonestamente ho notato che studiare dagli appunti degli studenti è utile per capire i concetti fondamentali degli argomenti del sistema operativo ma per l'esame (anno di riferimento 2024/2025, prof. De Gaspari) è necessario anche ripassare dalle slide del professore poiché molti argomenti delle domande d'esame sono anche nozioni minori che molti appunti degli studenti (compreso questo) ignorano.

In altri casi (non sempre) invece bisogna conoscere bene l'argomento poiché la domanda potrebbe non essere scritta nelle slide e bisogna trarre risposta dal  
la conoscenza totale dell'argomento.

Altre domande sono su piccoli esercizi di scheduling di processi o di rimpiazzo pagine.

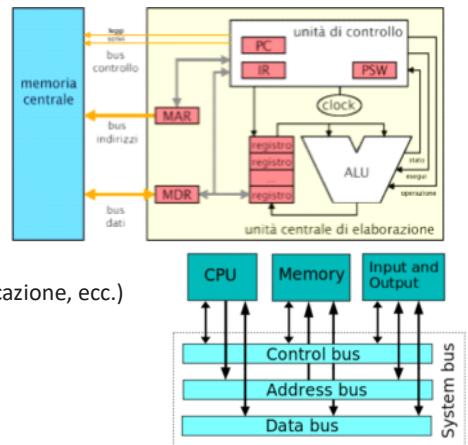
Altra cosa alcuni argomenti (minori) non li ho inseriti o magari ho inserito argomenti che il prof non va a vedere, perche avevo iniziato gli appunti basandomi sugli appunti (si gli appunti degli appunti) di altri studenti (come CasuFrost, SimoneLid e altri).

# Architettura dell'OS

Tuesday, October 1, 2024 7:14 PM

## Componenti OS

- CPU:** processore che effettua i calcoli e opera in base ad un **ciclo di istruzioni**
  - FETCH:** carica l'istr. dalla memoria
  - DECODE:** interpreta l'istr.
  - EXECUTE:** esegue l'istr
- Memoria principale (RAM):** contiene i dati e le istr. usati dalla CPU
- Dispositivi I/O:** periferiche di input e output (dischi, mouse, dispositivi di comunicazione, ecc.)
- Bus di sistema:** mezzo di comunicazione tra i componenti del pc
  - Data Bus:** trasporta i dati
  - Address Bus:** Determina l'indirizzo su cui mandare i dati
  - Control Bus:** Coordina quale componente usare

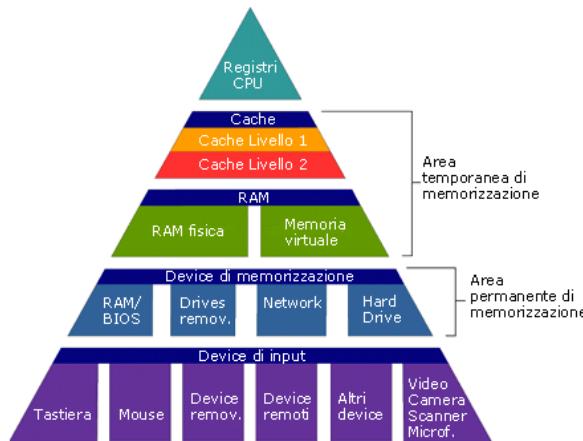


La **CPU** è l'**unità centrale di elaborazione**. Essa esegue le istr. di calcolo e controllo, coordinando gli altri componenti nell'esecuzione delle istruzioni stesse.

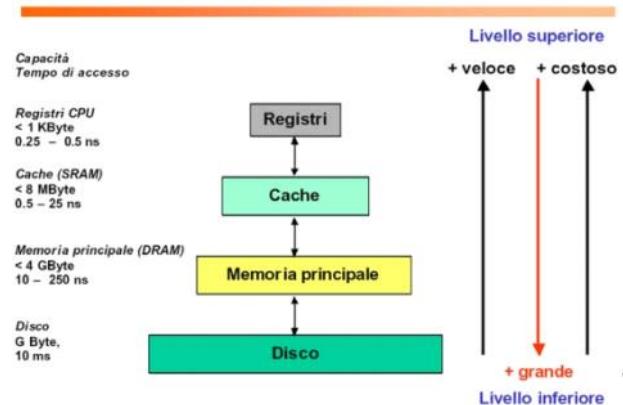
Le componenti principali della CPU sono:

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

## Memoria



Livelli della gerarchia di memoria



La **memoria** è una sequenza di celle grandi 1 Byte (8 bit), o multipli di esso; Ogni cella ha il proprio indirizzo.

Distinguiamo:

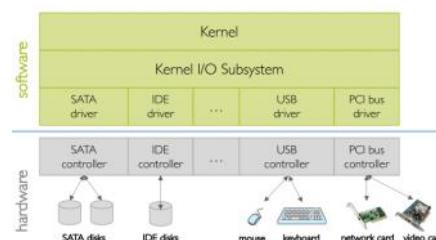
- Memoria Principale**, che si divide in:
  - principalmente **RAM**, una mem. veloce e volatile.
  - Cache**, una memoria di alta qualità che si pone tra CPU e RAM
- Memoria Secondaria**, meno veloce della RAM e usata principalmente per immagazzinare grandi quantità di dati. Tra queste troviamo SSD e Hard Disk

## Dispositivi I/O

- Ogni dispositivo I/O è composto in **dispositivo fisico** e **device controller** (chip che lo gestisce)
- L'OS comunica con il dispositivo tramite **driver** (software per tradurre le richieste dell'OS)
- Ogni dispositivo usa una serie di **registri** per comunicare con l'OS
  - Reg. di stato:** forniscono le info del dispositivo alla CPU
  - Reg. di controllo:** sono usati dalla CPU per controllare il dispositivo
  - Reg. dati:** usati per lo scambio di dati con il dispositivo

Per distinguere tra RAM e dispositivi I/O il control bus è dotato di una **linea speciale** detta **M/#IO** che stabilisce se la CPU vuole comunicare con la RAM o col dispositivo I/O

La CPU può comunicare col device controller in **2 modi**



### Comunicazione Port-Mapped

- I riferimenti ai controller sono effettuati usando uno **spazio di indirizzi aggiuntivo**
- Richiede **istr. speciali (IN/OUT)**
- Quando si usano queste istr., M/#IO non è usata quindi la RAM non risponde ma il dispositivo si

### Comunicazione Memory-Mapped

- Viene usata parte degli indirizzi di mem. ma non si usano istr. speciali
- La CPU tratta i registri dei dispositivi come se fossero dei suoi registri normali di memoria
- M/#IO è usata indicando che l'indir. richiesto dalla CPU si riferisce

### Vantaggio

- Più vantaggioso su CPU con **limitate capacità di indirizzamento**
- Tutto lo spazio di indirizzi può essere usato per la RAM

alla memoria

### Vantaggio

- La CPU richiede meno logica interna
- Usato da CPU moderne (32/64 bit) poiché la mappatura di mem. non è più un problema

# Servizi dell'OS

Tuesday, October 1, 2024 8:54 PM

## Kernel/User Mode e protezione della memoria

Alcune istr. sono più **sensibili** (privilegiate) di altre, es. HLT (arresta il sistema) o INT X (genera un interrupt di sistema). Per fare in modo che solo l'OS possa usare queste istr., la CPU può essere impostata in 2 modalità.

### Kernel Mode, senza restrizioni

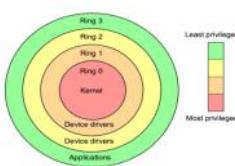
- L'OS può eseguire qualsiasi istr.

### User Mode, non è possibile

- Accedere agli indirizzi I/O
- Manipolare i contenuti di mem
- Arrestare il sistema
- Passare alla Kernel Mode
- Etc...

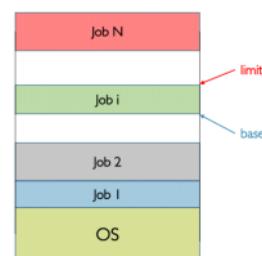
Viene implementato in HW un **bit speciale** salvato in un registro protetto  
0=Kernel, 1=User

Nelle architetture più moderne vengono usati più livelli di protezione detti **protection rings** in cui ogni ring aggiunge restrizioni sulle istr. eseguibili dalla CPU



Invece per la protezione della memoria la tecnica più semplice è di avere **2 registri dedicati**

- **Base** -> contiene l'indirizzo **iniziale** dello spazio utilizzabile
  - **Limite** -> contiene l'indirizzo **finale** dello spazio utilizzabile
- L'OS carica questi 2 registri all'avvio del programma e la CPU controlla che ogni indirizzo di mem. usato dal programma utente rientra tra i valori dei 2 registri



## Trap e System Calls

La **Trap di sistema** è un evento che richiede il **passaggio da user mode a kernel mode**:

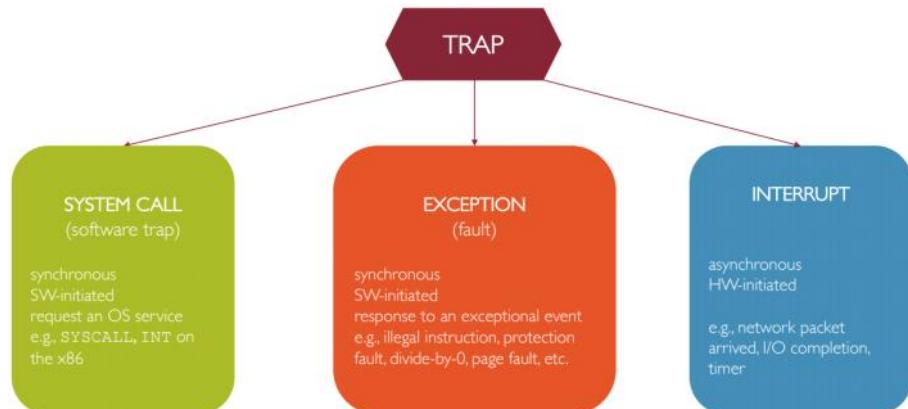
- System calls**: richiesta di un servizio dell'OS
- Exception**: gestione degli errori dovuti a eventi inattesi
- Interrupt**: completamento di una richiesta in attesa

Le **Exceptions** sono generate dal SW (es. divisione per 0)  
Gli **Interrupt** sono generate dal HW (es. chip timer, I/O, ecc.)

La **disabilitazione degli interrupt** non impedisce la creazione di nuove interruzioni, ma solo la loro **gestione**

Le **System calls** sono procedure usate dai programmi utente per accedere a servizi messi a disposizione dall'OS, in particolare l'esecuzione di **istruzioni privilegiate** (es. I/O)

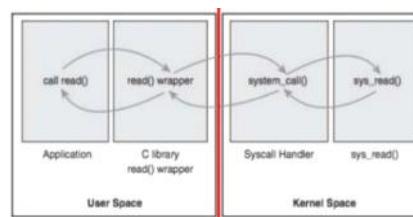
Le Syscall si dividono in **6 categorie**:



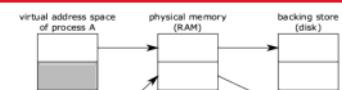
Process Control	File management	Device Management	Information maintenance	Communications	Protection
Procedure per creare e gestire processi.  Quando un processo si ferma, un altro deve ri/partire	Procedure per creare/eliminare/modificare file e i suoi attributi	Procedure per configurare/modificare /rilasciare i dispositivi  I dispositivi possono essere <b>fisici</b> (hard disk) o <b>virtuali</b> (RAM) o risorse HW generali	Procedure per eseguire <b>set/get</b> sulle info dell'OS (es. tempo, processi, system data)	Procedure per creare/eliminare comunicazioni e gestisce i dispositivi connessi (per collegarsi/scollegarsi o mandare informazioni)  <b>2 modelli di comunicazione:</b> <ul style="list-style-type: none"><li>Message passing</li><li>Shared memory</li></ul>	Procedure per controllare quali user/processi hanno accesso a determinate risorse  Gli utenti possono <b>temporaneamente</b> elevati i permessi di accesso sotto specifiche circostanze

### System Call Handler (non nell'esame)

Quando viene invocata una Syscall dall'utente tramite un'API dell'OS, la CPU cambia da User Mode a Kernel Mode e viene chiamata l'**Interrupt Vector Table (IVT)**, tramite cui verrà attivato il **System Call Handler** che contiene una **System Call Table** tramite cui vengono mappate le syscall ai propri codici d'esecuzione per infine eseguire la syscall richiesta.



## Timer, Istruzioni atomiche, Scheduling e Virtualizzazione



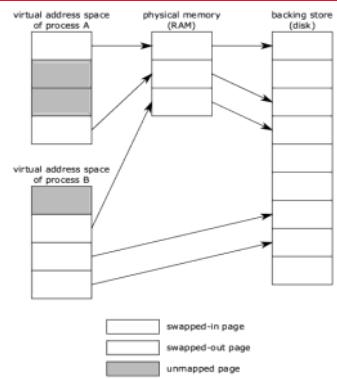
## Timer, Istruzioni atomiche, Scheduling e Virtualizzazione

Il **Timer** è servizio HW che genera un interrupt ogni "tot" microsecondi e ad ogni interrupt il **CPU Scheduler** decide quale processo da eseguire dopo, impedendo alla CPU di essere monopolizzata da un job.

Gli interrupt sono asincroni e possono interferire coi processi in esecuzione. L'OS deve quindi **sincronizzare le operazioni** dei processi concorrenti e cooperanti. L'HW deve assicurarsi che istruzioni corte vengano eseguite **atomicamente** o disabilitando gli interrupt prima delle istr. e riabilitandoli dopo oppure usando istr. speciali che vengano eseguite nativamente in modo atomico

La **Virtualizzazione della memoria** è una **astrazione logica** della memoria fisica dando ad ogni processo l'illusione che la mem. fisica sia uno spazio contiguo di indirizzi.

Permette l'esecuzione dei programmi senza che essi vengano caricati totalmente in memoria (sono caricati totalmente nella mem. virtuale).



# Struttura dell'OS

Thursday, October 3, 2024 6:27 PM

L'OS dovrebbe essere partizionati in sottosistemi separati, ognuno con task, inputs e output caratteristici.

È importante separare le politiche dai meccanismi:

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

Questo disaccoppiamento è un principio fondamentale poiché migliora il sistema dandogli:

- **Flessibilità:** l'aggiunta e modifica delle politiche è più facile
- **Riutilizzabilità:** i meccanismi esistenti possono essere usati per altre politiche
- **Stabilità:** l'aggiunta di nuove policy non destabilizza il sistema

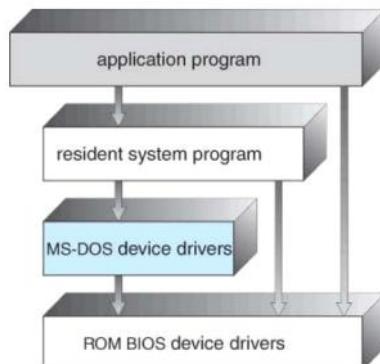
Ci sono vari modi per strutturare un OS:

- **Semplice**

Non ci sono sottosistemi e non c'è separazione tra Kernel e User mode

**PRO:** facile da implementare

**CONTRO:** poco rigido e estremamente insicuro



- **Complesso (Kernel Monolitico) (UNIX)**

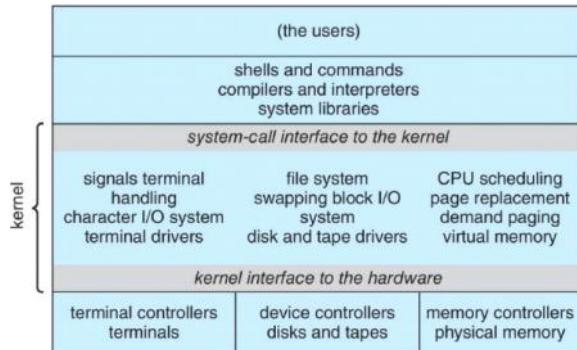
L'intero OS opera in Kernel mode e i software utente in User mode.

L'OS è strutturato a **strati**:

- Hardware di diverso tipo
- **Kernel:** che si occupa di:
  - Interfacciarsi con l'HW
  - Scheduling della CPU
  - Gestione della memoria
  - altre op. di sistema tramite syscall
- Programmi di sistema

**PRO:** efficiente e facile da implementare

**CONTRO:** poco rigido e insicuro



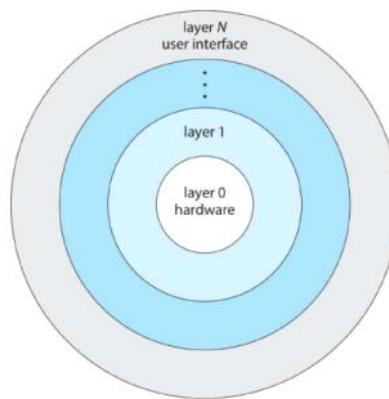
## • A Livelli

L'OS è diviso in N livelli (HW = Liv. 0) e tutti i livelli insieme costituiscono il kernel.

Il vantaggio è che i cambiamento in un componente **influiscono** solo su quello stesso componente  
Ogni livello **L** usa le funzionalità del livello **L-1** per esporre nuove funzionalità del livello **L+1**.

**PRO:** modulare, portabile e facile da debuggere

**CONTRO:** complessa la comunicazione tra livelli, copia extra



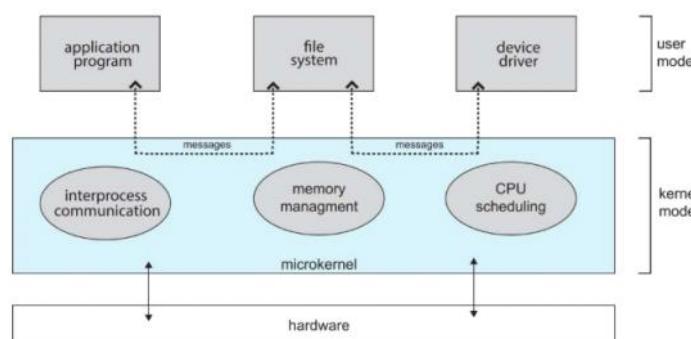
## • Microkernel

Il Kernel contiene **funzionalità di base**, le altre funzionalità dell'OS e i software utente operano in User mode.

La funzione principale del microkernel è **fornire la comunicazione** tra il programma client e i vari servizi che sono in esecuzione nello spazio utente.

**PRO:** più sicurezza, affidabile e estensibile

**CONTRO:** poco efficiente se sovraccaricato



## • Moduli Kernel Caricabili (LKMs)

Forse la miglior metodologia attuale per la progettazione dell'OS.

Il kernel ha una serie di componenti principali e può **collegare servizi aggiuntivi tramite moduli**.

Invece che ricompilare il kernel ad ogni modifica, il collegamento dinamico dei servizi permette l'aggiunta di nuove funzionalità più facilmente

# Cache

giovedì 23 gennaio 2025 22:56

La **cache** è una memoria piccola e **veloce** che sfrutta il principio di località, dove se un proc. **accede** ad un **indir. in mem.**, è molto **probabile** che **accederà** nuovamente allo **stesso indir.** (**principio temporale**) e che acceda a **indir. molto vicini ad esso** (**principio spaziale**). Spesso si usano più livelli di cache (calando di grandezza quando ci si avvicina alla CPU) per velocizzare la ricerca dei dati per la CPU. Contiene delle **copie di porzioni** della RAM.

Il processore prima controlla se un **dato è nella cache** e se non c'è il **corrispondente blocco di RAM** viene **caricato nella cache**. Gestione totalmente controllata dall'HW, quindi il programmatore e l'OS non "vedono" in alcun modo la cache.

Incrementare la misura dei blocchi della cache **aumenta il num. di accessi riusciti** ma incrementarla troppo è **controproducente** poiché molti più dati saranno rimossi abbassano la probabilità di accesso riuscito.

# Programmi e Processi

Friday, October 4, 2024 6:20 PM

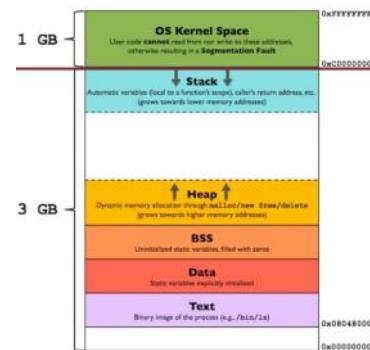
Un **Programma** (statico/passivo) è un file eseguibile salvato nella mem. secondaria (disco) e contiene l'insieme di istr. necessario per completare un compito specifico

Un **Processo** (dinamico/attivo) è una istanza attiva di un programma caricata in mem. principale. Più processi possono corrispondere ad un singolo programma.

È l'astrazione di un programma attivo ed esegue una istr. alla volta in modo sequenziale

L'OS dà la stessa quantità di **Virtual Address Space** (un'astrazione dell'address space della mem. fisica) a ogni processo, la quale è divisa in più sezioni:

- **Text**: istr. dell'eseguibile
- **Data**: dati statici già inizializzati all'avvio
- **BSS**: dati statici non inizializzati all'avvio
- **Stack**: dati utilizzati da ogni funzione, ossia i vari **Stack Frame**
- **Heap**: usato per allocazioni dinamiche (di dimensione non fissa)
- **Spazio libero**: tra Stack ed Heap, usato da entrambi per "espandersi":
  - Heap: si espande tramite **allocazione dinamica**
  - Stack: si espande tramite **chiamate di funzioni consecutive**



Esempio suddivisione del Virtual Address Space di un architettura a 32bit

## Stack

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

Nello Stack sono definite **2 operazioni**:

- **Push**: per aggiungere elementi
- **Pop**: per rimuovere elementi

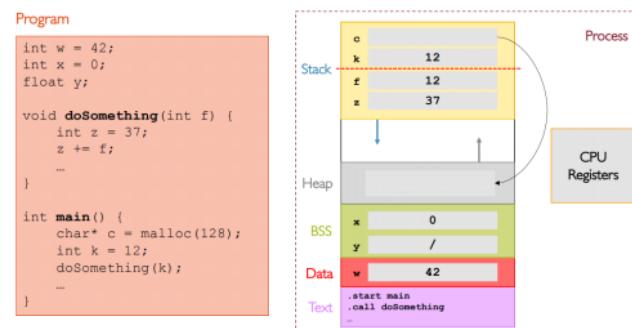
Il registro dedicato **%esp** indica la parte superiore dello stack.

Ogni funzione usa una **porzione** dello stack, chiamata stack frame. Possono esistere simultaneamente più stack frame, ma **solo uno è attivo**.



Esempio:

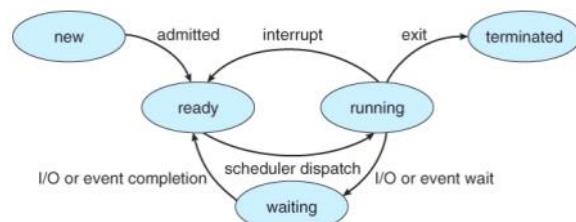
- La var. "w" è salvata in **Data** perché inizializzata con valore (statica)
- La var. "x" e "y" sono in **BSS** perché statiche ma **non inizializzate** (il val di default di un Int è 0 quindi "x" è ignorata dal compiler)
- Le istr. delle funz. "main()" e "doSomething()" sono in **Text**
- Gli **Stack Frame** di "main()" e "doSomething()" sono in **Stack**
- La var "char\* c" è un **puntatore** salvato in **Stack** che fa riferimento ad una zona dell'**Heap** di 128bit(16char)



## Stato di un Processo

Ad ogni momento un processo può essere in uno di questi **5 stati**:

- **New**: l'OS ha appena avviato lo stato del processo
- **Ready**: Il processo è pronto per essere eseguito ma aspetta di essere schedulato nella CPU
- **Running**: Il processo è in esecuzione nella CPU
- **Waiting**: il processo è sospeso in attesa che una risorsa sia disponibile o che un evento sia completato/avvenga (es. keyboard input, accesso al disco, timer, ecc.)
- **Terminated**: Il processo è terminato e l'OS può eliminarlo



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

- Il processo chiamante (user space) non può fare nulla finché la syscall termina la richiesta
- L'OS (kernel space) imposta il processo in **waiting** e schedula il primo processo in **ready** presente nella **state queue**
- Quando la syscall termina la richiesta del processo bloccato in precedenza, quel processo passa a **ready**, pronto per essere schedulato nuovamente

Solo il processo che ha effettuato la chiamata viene bloccato, non tutto l'OS

## PCB

Il **Process Control Block (PCB)** è la struttura dati principale usata dall'OS per tener traccia dello stato e della posizione in mem. di un processo. All'avvio di un processo, l'OS alloca un nuovo PCB, aggiungendolo alla **state queue**, per poi de-allocarlo quando il processo associato termina

Ogni PCB è composto da:

- **Identifieri:**
  - del processo (PID)
  - del processo padre (PPID)
  - dell'user proprietario
- **Info sullo stato del processore:**
  - registri utente
  - program counter (PC)
  - stack pointer
  - registri di stato
- **Info per il controllo del processo:**
  - stato processo (ready, suspended, blocked, ecc)
  - priorità
  - info sullo scheduling
  - evento da attendere per tornare ready
- **Supporto per strutture dati:**
  - puntatori ad altri processi
- **Comunicazione tra processi:** flag, segnali, messaggi
- **Permessi speciali:** non tutti i processi possono accedere a tutto
- **Gestione della mem.**
  - puntatori ad aree di mem. che gestiscono l'uso della mem. virtuale
  - es. page table
- **Uso delle risorse:**
  - file aperti
  - uso di risorse fino ad ora

process state
process number
program counter
registers
memory limits
list of open files
• • •

## Riassunto

- Il processo è l'**unità di esecuzione**
- l'OS dà ad ogni processo l'**illusione** di avere una **sequenza contigua di address** di memoria a cui possono fare riferimento (**Virtual address space**)
- l'OS tiene **traccia** delle **informazioni** relative ai **processi** usando una struttura dati chiamata **Process Control Block (PCB)**
- Un processo può essere in uno di **5 possibili stati**: **new, ready, running, waiting, terminated**

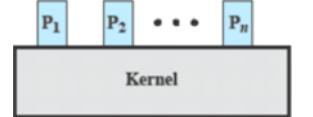
# OS

giovedì 23 gennaio 2025 21:28

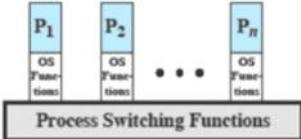
L'OS è un **insieme di programmi** eseguiti dal processore come ogni altro programma.

Esistono tre tipi di implementazioni dell'OS:

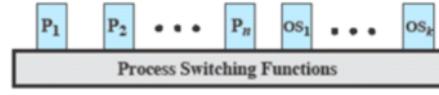
- **Kernel separato** (l'os non è un processo)
  - Kernel eseguito **al di fuori** dei processi
  - Il concetto di processo si **applica solo ai programmi utente**
  - L'OS è eseguito come **un'entità separata**, con privilegi più elevati
  - Ha una sua zona di mem. dedicata sia per i dati che per il codice sorgente che per lo stack



- **Esecuzione all'interno dei processi utente:**
  - L'OS è eseguito nel **contesto del processo utente**
  - **Non** c'è bisogno di un **process switch** per eseguire una funzione dell'OS, **solo del mode switch**
  - Stack delle chiamate separato
  - Dati e codici macchina condiviso coi processi
  - **Process switch solo**, eventualmente, **alla fine**, se lo scheduler decide che tocca ad un altro processo



- **OS basato sui processi**
  - OS implementato come un **insieme di processi di sistema** con privilegi più elevati
  - Partecipano alla **competizione** per il processore insieme ai processi utente
  - Se un processo effettua una syscall, questo crea un nuovo processo di sistema
  - **Il process switch non è un processo**



# Creazione processi

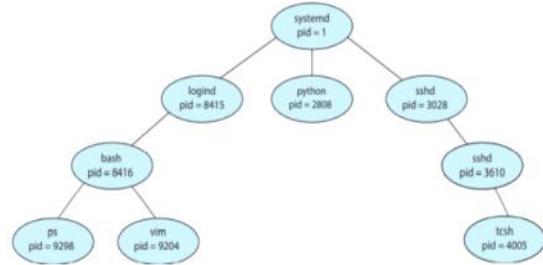
Friday, October 4, 2024 7:28 PM

## Processo padre e figlio

Un processo **"padre"** può creare altri processi **"figlio"** tramite syscall. Ad ogni processo è associato un **process identifier (PID)**, che lo identifica, e un **parent process identifier (PPID)**, che identifica il processo padre

Processo sched e init

In UNIX il processo scheduler è chiamato **sched** e il suo PID è 0. La prima operazione eseguita da **sched** all'avvio dell'OS è avviare il processo **systemd**, avente PID 1. **systemd** si occupa di gestire tutti i processi in background (**processi deamon**), e altri processi di sistema, diventando l'antenato di tutti i processi.



## Fork e Spawn

Per creare i processi figli viene usata la syscall:

- **fork()** (su UNIX):
  - Il figlio è un **duplicato del padre** (con lo stesso programma e dati del padre)
  - ognuno ha il **proprio PCB**
- **spawn()** (su Windows)
  - Il figlio è un processo legato ad un programma **diverso dal padre**
  - avente uno spazio di indirizzamento, quindi con istr., dati e PCB **diversi**

Su UNIX viene usata la syscall **exec()** dopo fork() per ottenere lo stesso effetto di spawn(). exec() **rimancia completamente il processo precedente**, evitando di riprendere l'esecuzione del precedente una volta completato il processo di syscall.

Poiché usando fork() i due processi condividono lo stesso codice e le stesse risorse, nel processo **padre essa ritornerà un PID maggiore a 0**.

Mentre il processo **figlio ritornerà 0 come PID**.

Una volta creato un processo figlio, il processo padre ha 2 opzioni:

- **Attendere** il termine del processo figlio con **wait()**
- Continuare la sua esecuzione in **parallelo** col figlio senza bloccarsi

## Terminazione dei processi

Un processo può essere terminato usando **exit()**. Anche l'OS può terminare il processo, nel caso in cui sia necessario liberare le risorse o se viene eseguito un comando **kill()**.

Quando un processo termina, le **risorse** vengono **liberate** e viene ritornato lo **stato di processo terminato**, il **tempo di esecuzione** e un **codice esito** dell'esecuzione.

## Processo orfano e processo zombie

Se un padre esegue **exit()** e suo figlio non è ancora terminato, quest'ultimo viene detto **orfano**, venendo ereditato da **init** il quale si occuperà di ucciderlo con **kill()**.

Se un figlio tenta di terminare ma suo padre non sta eseguendo **wait()**, viene detto processo **zombie**, venendo ereditato sempre da **init** per poi essere ucciso.

### Es. 1

```
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);
    }
}
```

In main() viene inizializzata la var. **pid**, la quale viene assegnata al val. di ritorno della fork.

Viene così generato un **secondo processo** che esegue lo stesso programma del padre, con l'unica differenza che:

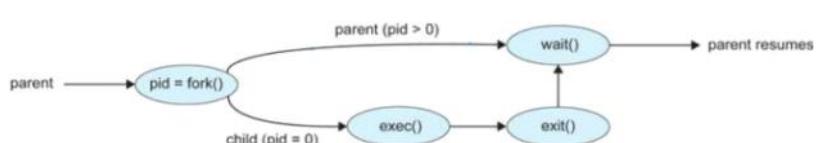
- il padre avrà il **pid uguale al pid del figlio**
- mentre il figlio avrà il **pid uguale a 0**.

Il processo figlio riprende l'esecuzione del programma della fork che lo ha generato.

Se non ci sono errori e il pid è uguale a 0, siamo nel processo figlio, dove viene eseguito **exec()**, che caricherà un nuovo programma nella mem. e il processo figlio inizierà ad eseguirlo.

Altrimenti (se il pid > 0) siamo nel processo padre, il quale attenderà la fine dell'esecuzione di tutti i processi figli.

- L'albero decisionale del programma si sviluppa come tale:



### Es 2

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



```

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

```

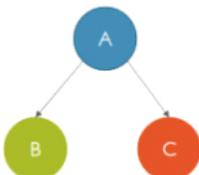


### Es 3

```

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

```

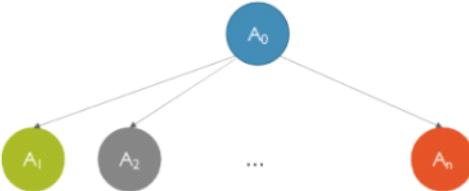


### Es 4

```

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);
}

```



# Thread e Multi-threading

venerdì 22 novembre 2024 14:14

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

Ogni processo definisce le **risorse "globali"** (spazio di indir., istr. da eseguire, dati, risorse, ...) e ogni suo thread definisce un **singolo stream di esecuzione** all'interno del processo stesso.

Poiché lo **spazio di indirizzamento è condiviso tra tutti i suoi thread**, nessuna syscall è richiesta per far cooperare i thread tra di loro, risultando più semplice dello scambio di messaggi o memoria condivisa che si usa per far comunicare i processi.

L'uso dei thread è molto utile per svolgere **più attività indipendenti l'una dall'altra**. In particolare, quando un'attività richiede che il processo si **blocchi**, usando **più thread le altre attività** possono continuare ad essere **svolte liberamente**.

La programmazione **multi-thread** ha senso solo se applicata su **processori multi-core** (altrimenti verrebbero **eseguiti sequenzialmente**), dove i thread vengono **divisi e eseguiti su tutti i core disponibili**, consentendo un reale parallelismo.

I recenti processori possono supportare **più thread virtuali per core fisico**, ad esempio con l'**hyper-threading** consente lo scheduling di **due thread per core fisico**.

L'uso dei thread risulta essere migliore di svolgere un processo su un single-thread poiché:

- La **comunicazione e il context switch** tra thread è **più veloce** di quella tra i processi

I vantaggi dei processi **multi-thread** sono:

- **Reattività**: un processo può fornire una risposta in modo rapido mentre tutti gli altri thread sono bloccati/rallentati dall'uso intenso della CPU
- **Condivisione delle risorse "globali"** tra i thread
- **Economicità**: creare e gestire thread è più veloce e della creazione e gestione dei processi
- **Scalabilità**: Un processo multi-thread può essere suddiviso tra tutti i core disponibili della CPU

Abbiamo due tipi di **parallelismo**:

- **Data parallelism**: dove vengono **distribuiti** gli stessi **dati** su **più thread** (es. divide il compito di sommare un array tra più thread)
- **Task parallelism**: dove **ogni thread** esegue un **compito unico e indipendente**

Un'applicazione può usare un **ibrido** di queste due strategie

I thread possono essere supportati in 2 modi:

## Kernel Thread

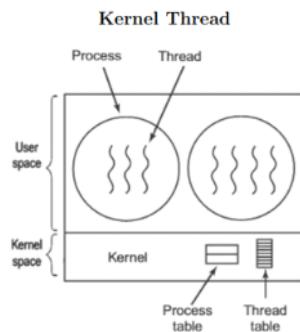
unità di esecuzione più piccola schedulabile dall'OS, responsabile del supporto e gestione di tutti i kernel thread attivi, fornendo syscall per poterli creare/gestire dall'user space. Ogni kernel thread ha un **Thread Control Block (TCB)**

**Pro:**

- Il kernel è a conoscenza di tutti i kernel thread avviati
- Lo scheduler può decidere di dare più tempo d'uso della CPU a un processo con molti thread
- Utile per applicazioni con blocchi frequenti
- Passare da un thread all'altro è più veloce di farlo coi processi

**Contro:**

- Rende il kernel più complesso
- **Lento e inefficiente** (necessita chiamate al kernel)
- Nonostante il context switch è più leggero, la sua gestione tramite kernel non conviene



## User Thread

Gestiti dal sistema run-time tramite **library per gestione dei thread**, gestiti come se fossero processi single-threaded.

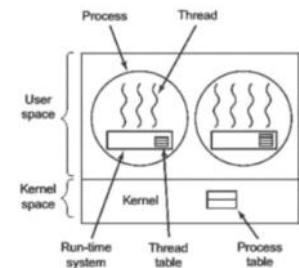
**Pro:**

- **Veloci e leggeri**, le politiche di scheduling sono più flessibili
- Può essere implementato da OS senza supporto nativo del threading
- Nessuna syscall o context switch necessario

**Contro:**

- Il kernel non conosce gli user thread attivi
- Le decisioni prese dallo scheduler sono inefficienti
- Mancanza di coordinazione tra kernel e thread (es. un proc. con 100 thread può competere con un proc. con un solo thread)
- Richiede che tutte le syscall dell'OS siano **non bloccanti**, altrimenti tutti i thread in un processo devono attendere

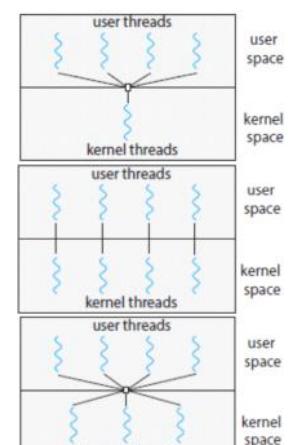
## User Thread



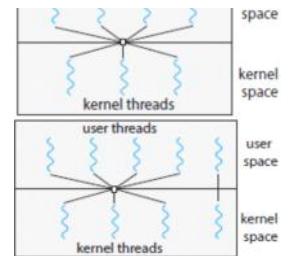
## Modelli di multi-threading

Per gestire il **multithreading** vengono usati più modelli:

- **Many-to-One**
  - Più user thread mappati a **un solo kernel thread**
  - Ogni processo usa un **solo user thread alla volta** poiché vi è un singolo kernel thread associato, che può operare su un solo core, quindi i processi con più user-thread **non** possono essere divisi tra più core
  - Con le syscall bloccante, **l'intero processo viene bloccato**
  - **Puramente basato sul livello user**
- **One-to-One**
  - Ogni user-thread è mappato a **un solo kernel thread**
  - Ogni processo può usare **più user thread alla volta**, poiché sono associati a più kernel thread e possono essere divisi tra più core
  - Con le syscall bloccante, **l'intero processo NON viene bloccato**
  - Richiede un kernel **più complesso**, rallentando l'OS
  - Molti implementazioni fissano un **limite al num. di thread** che possono essere creati
  - **Puramente basato sul livello kernel**
- **Many-to-Many**
  - Più user thread sono mappati a **più kernel thread**
  - Il num. di kernel thread può essere **minore o uguale** al num. di user thread
  - I processi possono essere **suddivisi tra più core**
  - Le syscall bloccanti, **l'intero processo NON viene bloccato**



- Più user thread sono mappati a più kernel thread
- Il num. di kernel thread può essere minore o uguale al num. di user thread
- I processi possono essere suddivisi tra più core
- Le syscall bloccanti, l'intero processo NON viene bloccato
- I num. di thread creabili non è limitato
- Two-level
  - Simile al Many-to-Many, associa più user thread a più kernel thread
  - Ma permette anche ad alcuni user thread di associarsi con un numero uguale di kernel thread, similmente al modello One-to-One
  - Le politiche di scheduling sono più flessibili



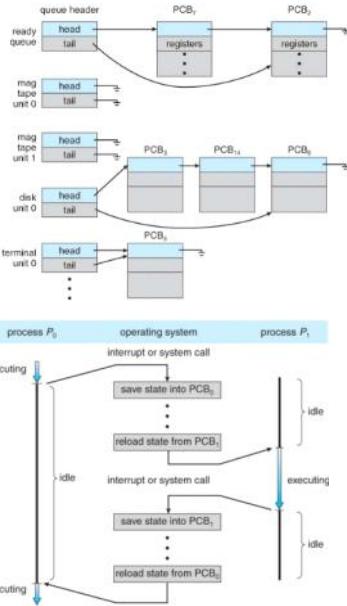
# CPU

giovedì 21 novembre 2024 15:11

**Scheduler CPU:** processo che, a seconda di una **policy** di scheduling, **seleziona un processo ready** quando la **CPU è inattiva** per poi eseguirlo.

Questo processo deve **tenere sempre la CPU occupata** scambiando frequentemente i processi nei core così che tutti i programmi condividano equamente la CPU.

Però quando lo **scheduler** interviene per **scambiare i processi**, viene **impiegato tempo** sulla CPU che viene "perso" per eseguire altro.



## Process State Queue

Per gestire i PCB dei vari processi, l'OS usa una **queue** (coda). Vengono usate **5 code per gestire gli stati dei processi e una coda per ogni dispositivo I/O**

Quando l'OS modifica lo stato di un processo, il PCB viene spostato da una coda all'altra.

La quantità di PCB che si possono inserire nella **running queue** è legata al num. di core usati dall'OS, per le altre non vi è restrizione.

## Context Switch

Occorre distinguere tra **mode switch** e **process switch**:

- **Mode switch:** da modalità utente a sistema e viceversa (es. interrupt)
- **Process switch (context switch):** scambiare l'attuale processo nel processore con un altro processo

Il **context switch** è la procedura usata dalla CPU per **sospendere il processo in esec.** per poter **eseguire un altro in ready**.

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

Quando arriva una **trap**, viene generato un **context switch** e la CPU deve:

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

L'operazione di **context switch** è **molto costosa**, e un time slice minore implica una reattività massimizzata, tuttavia sottraendo tempo all'esec. dei processi per via del costo del context switch, richiedendo un **compromesso tra i due**. Generalmente il **time slice** va dai 10 ai 100 ms, mentre un **context switch** richiede 10 µs

## Scheduler su Architetture Multiprocessore

Ci sono due tipologie di assegnazione del processore ai processi:

- **Assegnamento statico:** quando un processo è creato, gli è assegnato un processore e resterà per tutta la sua durata
  - Si può usare uno scheduler per ogni processore
  - **Pro:** semplice da realizzare, poco overhead
  - **Contro:** un processore può rimanere idle
- **Assegnamento dinamico:** nel corso della sua vita, un processo può cambiare processore
  - **Pro:** più equo e ragionevole
  - **Contro:** complicato da realizzare

Anche l'OS si può basare su uno dei due assegnamenti visti:

- **OS fisso:** rimane eseguito su un unico processore
  - Solo i processi utente possono cambiare processore
  - **Pro:** più semplice da realizzare
  - **Contro:** può diventare il **bottleneck**, se fallisce il processore dell'OS cade l'OS
- **OS mobile:** viene eseguito sul processore che capita
  - **Pro:** ottima flessibilità
  - **Contro:** richiede più overhead

# Scheduling

lunedì 18 novembre 2024 22:00

## Scheduling

Per **scheduling** si intende l'insieme di policy per stabilire quale processo eseguire sulla CPU, consentendo ai processi di intercambiarsi per poter **massimizzare** l'uso della CPU.

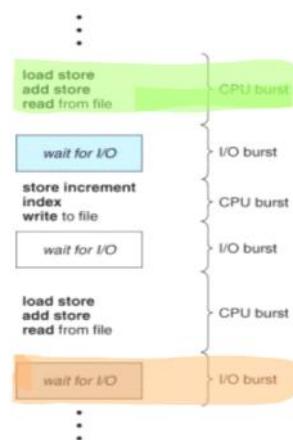
L'obiettivo è quello di rendere il sistema più "efficiente" ed "equo" possibile.

L'esecuzione del processo inizia con un **burst della CPU** (CPU esegue calcoli per il processo), seguito da un **burst di I/O** (attesa che i dati vengano trasferiti dentro/fuori dall'OS), ripetendo poi il ciclo e così via.

Le durate dei **burst della CPU** variano notevolmente in base al processo e al computer.

Lo scheduler andrà quindi a selezionare i **processi liberi** dalla running queue per l'esecuzione di un programma su un core.

Quando un processo viene **spostato da uno stato all'altro**, il suo PCB viene scollegato dalla coda corrente e collegato in quella nuova.



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

- 1) Il processo passa dal **running state** al **waiting state** (es. effettuata richiesta I/O o invocata syscall wait())
- 2) Il processo passa dal **running state** al **ready state** (es. in risposta ad un interrupt)
- 3) Il processo passa dal **waiting state** al **ready state** (es. completata richiesta I/O o terminata syscall wait())
- 4) Il processo viene **creato o terminato**

Per i casi 1 e 4 non c'è scelta in termini di **scheduling**. Un nuovo processo deve essere selezionato per l'esecuzione per tenere occupata la CPU.

Per i casi 2 e 3 invece c'è scelta poiché la CPU può selezionare di nuovo lo stesso processo o sceglierne uno nuovo.

## Preemptive e Non-Preemptive

Uno scheduling **non-preemptive** non ha alcuna scelta se non **selezionare un nuovo processo** (es. condizioni 1 e 4), avviarlo e eseguirlo finché esso non decide di bloccarsi o termina.

Uno scheduling **preemptive** sceglie se continuare a **eseguire il processo attuale** o **selezionare uno nuovo** (tutte le 4 condizioni)

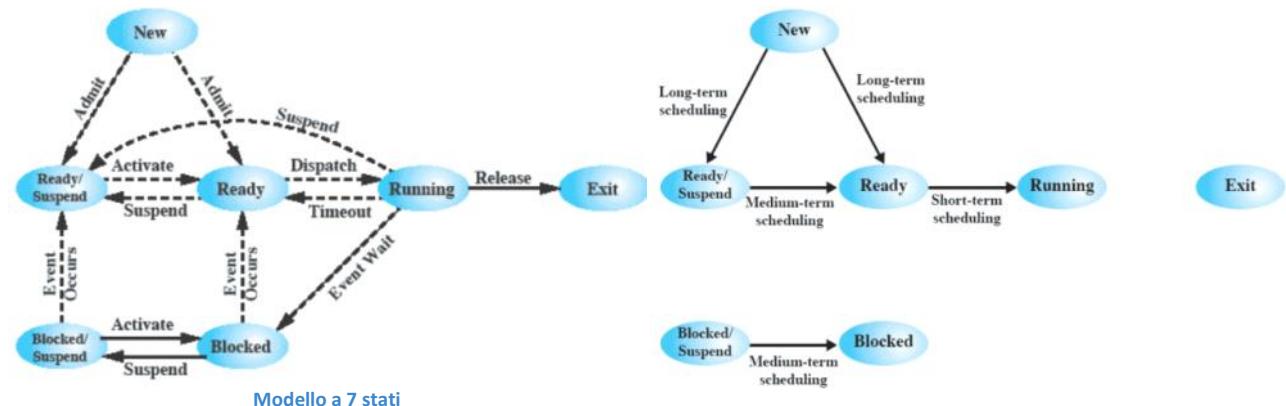
Lo scheduling **preemptive** può causare **problematiche** nel caso in cui avvenga quando il **kernel è occupato**.

Per prevenire tali problematiche si può:

- Far aspettare il processo finché la syscall del kernel non termina (non efficiente)
- Disabilitare gli interrupt prima di entrare nella sezione di codice critica, per poi riabilitarle subito dopo (**mutex locks**)

## Tipi di Scheduling

- **Long-term scheduling**: decide l'aggiunta di processi da essere eseguiti
- **Medium-term scheduling**: decide l'aggiunta di processi che sono in RAM
- **Short-term scheduling (Dispatcher)**: decide quale processo ready va eseguito nella CPU
- **I/O scheduling**: decide a quale processo, tra quelli con richiesta di I/O, va assegnato il corrispondente dispositivo I/O



## Criteri per lo Short-Term Scheduling

- **Criteri Utente:**
  - **Prestazionali**
    - turnaround time (tempo tra la creazione di un processo e il suo completamento (comprende i tempi di attesa)
    - response time (tempo tra la sottomissione di una richiesta e l'inizio della risposta)
    - deadline
  - **Non prestazionali**
    - **predicibilità**
- **Criteri di Sistema:**
  - **Prestazionali**
    - throughput
    - uso del processore
  - **Non prestazionali**

- Fairness
- gestione delle priorità
- bilanciamento dell'uso delle risorse

## Dispatcher

**Dispatcher:** modulo che si occupa di **dare alla CPU il controllo del processo** selezionato dallo scheduler.

Include:

- esecuzione del **context switch**
- **passaggio alla user mode**
- saltare alla pos. corretta nel programma caricato

Esso deve essere il **più veloce possibile** poiché viene **richiamato ad ogni context switch**

## Tempistiche 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:** tempo richiesto da un processo per l'**esecuzione della CPU**
- **turnaround time:** tempo trascorso tra il **completion time e l'arrival time** ( $T_{turnaround} = T_{completion} - T_{arrival}$ )  
È il tempo necessario per il completamento di un processo, dall'avvio al completamento, includendo il tempo di attesa.
- **Waiting time:** la differenza di tempo tra il **turnaround time e il burst time** ( $T_{waiting} = T_{turnaround} - T_{burst} = T_{completion} - T_{arrival} - T_{burst}$ )  
È il tempo di quanto i processi trascorrono in "ready" aspettando di andare in CPU.

**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**

Per la scelta dell'algoritmo di scheduling è necessario considerare più criteri:

- **massimizzazione dell'uso della CPU** (percentuale di tempo in cui è occupata)
- **massimizzazione del throughput**, num. max di processi completati in un'unità di tempo
- **minimizzazione del turnaround time**, tempo del completamento di un processo
- **minimizzazione del waiting time**, tempo trascorso in "ready" ad aspettare l'uso della CPU
- **minimizzazione del response time**, tempo impiegato dall'emissione del comando all'inizio della risposta della richiesta

# Algoritmi di Scheduling

giovedì 21 novembre 2024 18:35

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

## First Come First Served (FCFS)

algoritmo **non-preemptive** dove i job appena creati vengono inseriti in una **FIFO** e lo scheduler esegue il primo job della queue fino al completamento, per poi procedere a quello successivo.

### Pro:

- Estremamente **semplice** da implementare

### Contro:

- Un job potrebbe essere eseguito **indefinitamente** (es. finché non si blocca)
- Il **waiting time medio** è molto variabile perché 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 legati alla CPU forzeranno quelli legati all'I/O ad aspettare

## Esempi

### Es 1

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

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

<b>1</b>	New [A   B   C]	Order   Job   CPU burst (time units)
	Ready [A   B   C]	1   A   5
	Waiting	2   B   2
	Running	3   C   3

<b>2</b>	New [A   B   C]	Order   Job   CPU burst (time units)
	Ready [A   B   C]	1   A   5
	Waiting	2   B   2
	Running [A]	3   C   3

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:  
 $T_{waiting} = T_{turnaround} - T_{burst} = T_{completion} - T_{arrival} - T_{burst} = 5 - 0 - 5 = 0$
- Per il processo B si ha:  
 $T_{waiting} = T_{turnaround} - T_{burst} = T_{completion} - T_{arrival} - T_{burst} = 7 - 0 - 2 = 5$
- Per il processo C si ha:  
 $T_{waiting} = T_{turnaround} - T_{burst} = T_{completion} - T_{arrival} - T_{burst} = 10 - 0 - 3 = 7$
- Il waiting time medio, quindi, sarà:  
 $\bar{T}_{waiting} = \frac{1}{n} \sum_{i=0}^n T_i^{waiting} = \frac{0+5+7}{3} = 4$

### Es 2

- Nel seguente scenario, utilizzando sempre uno scheduling FCFS, si ha:

New [B   C   A]	Order   Job   CPU burst (time units)
	Ready [B   C   A]
	Waiting
	Running [C]
	A   B   C

Il waiting time medio, quindi, sarà:

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

### Es 3

Consideriamo il seguente scenario:

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.

<b>1</b>	New [A   B   C]	Order   Job   CPU burst (time units)
	Ready [A   B   C]	1   A   5
	Waiting	2   B   2
	Running [A]	3   C   3

<b>2</b>	New [A   B   C]	Order   Job   CPU burst (time units)
	Ready [A   B   C]	1   A   5
	Waiting	2   B   2
	Running [A]	3   C   3



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

<b>3</b>	New [B   C   A]	Order   Job   CPU burst (time units)
	Ready [C]	1   A   5
	Waiting [A]	2   B   2
	Running [B]	3   C   3
	A   B	time

New [A   B   C]	Order   Job   CPU burst (time units)
	Order   Job   CPU burst (time units)

**4**



Infine, il waiting time medio sarà:

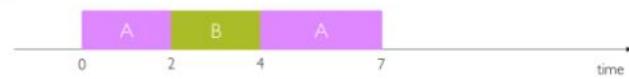
New	A   B   C
Ready	I   A   B   C
Waiting	
Running	

5

ritmo FCFS utilizza una queue basata sull'arrival time dei processi

New	A   B   C
Ready	C
Waiting	
Running	

4



$$\bar{T}_{\text{waiting}} = \frac{1}{n} \sum_{i=0}^n T_i^{\text{waiting}} = \frac{2+2+7}{3} \approx 3.7$$

## Round Robin (RR)

algoritmo **preemptive** dove viene selezionato il **primo processo ready**, con l'aggiunta di un **timer** (limite CPU burst), detto **time slice (time quantum)**:

- Quando un job prende **controllo** della CPU, il **time slice** viene **resetto**
- Se il job **termina prima** del time slice, lo scheduler **sceglierà il primo processo ready**
- Se il **time slice scade prima che termina** il job, lo scheduler **sosterà il job** in esecuzione alla **fine del ready**, poi **seleziona il primo processo ready**

La **ready queue** viene gestita come una **coda circolare**, distribuendo il tempo di uso della CPU equamente tra i processi.

I processi **CPU-bound** sono **favoriti**, mentre quelli **I/O-bound** ne usano solo una porzione

Pro:

- Più equo dell'FCFS, viene fornita la stessa quantità di tempo d'uso della CPU ad ogni job

Contro:

- **Non equo e inefficiente per l'I/O**
- Un **time slice troppo ampio** rischia di degenerare in un FCFS
- Un **time slice troppo stretto** rischia di generare un elevato numero di **context switch**, diminuendo l'uso della CPU

## Esempio

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

New	A   B   C
Ready	A   B   C
Waiting	
Running	

1

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

New	A   B   C
Ready	B   C
Waiting	
Running	

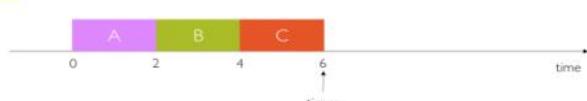
2



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

New	A   B   C
Ready	C
Waiting	
Running	

4



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

New	A   B   C
Ready	
Waiting	
Running	B

3

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

Il waiting time medio, quindi, sarà:

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

New	A   B   C
Ready	
Waiting	
Running	

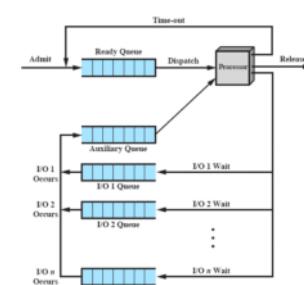
5



## Round-Robin Virtuale

È una soluzione per migliorare l'**equità** per i processi I/O bound rispetto al round-robin semplice.

- Dopo un completamento di I/O, il processo non va in coda ai ready, ma va in una nuova coda che ha priorità su quella dei ready
- però solo per la porzione del quanto che ancora gli rimaneva da completare



## Shortest Job First (SJF) o Shortest Process next (SPN) e Shortest Remaining Time (SRT) o (SRTF)

algoritmo basato sull'esecuzione priorità del job con la **quantità minore stimata di CPU burst**

L'SJF può essere:

- **non-preemptive**, ogni job viene eseguito fino al suo completamento (non funzionale)
- **preemptive**, ogni volta che un **nuovo job** arriva in **ready** e il suo CPU burst stimato è **minore** di quello rimanente del job in esecuzione, allora prende il controllo della CPU

Questa versione viene detta **Shortest Remaining Time First (SRTF)**

**Pro:**

- Ottimale per **minimizzare il waiting time medio**
- Funzionale con scheduler sia preemptive sia non-preemptive

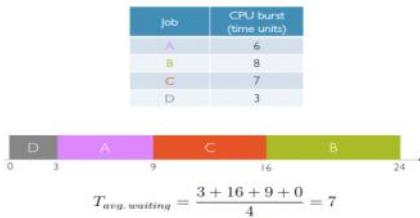
**Contro:**

- è quasi impossibile stimare esattamente il tempo di esecuzione della CPU di un job
- I processi **lunghi** potrebbero andare in **starvation**

### Esempio

#### Non-preemptive

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



#### Preemptive

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

1

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

2



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

3



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

4



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

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.

Una volta completato il processo B, verrà eseguito il processo avente il CPU burst stimato minore:

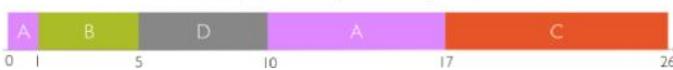
- Il processo A ha un CPU burst rimanente pari a 7
- Il processo C ha un CPU burst rimanente pari a 9
- Il processo D ha un CPU burst rimanente pari a 5

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

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

5

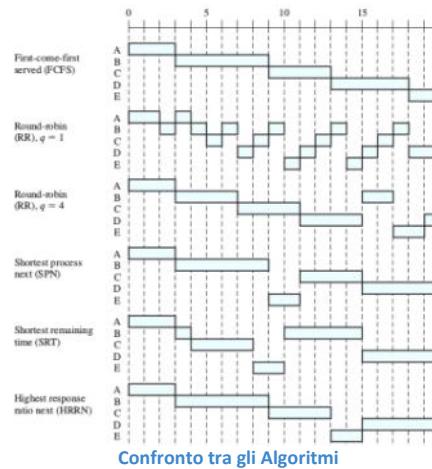


### Highest Response Ratio Next (HRRN)

Massimizza il rapporto tra il **tempo in attesa e richiesto** e il **tempo totale richiesto**

$$\frac{w + s}{s} = \frac{\text{tempo trascorso in attesa} + \text{tempo totale richiesto}}{\text{tempo totale richiesto}}$$





## Priority Scheduling

Algoritmo dove ad ogni job viene assegnato un valore che determina la sua **priorità**. Il job con priorità maggiore viene schedulato per primo.

La priorità viene assegnata in due modi:

- **Internamente**, assegnata dall'OS in base a dei criteri (es. CPU burst medio, risorse usate, ...)
- **Esteramente**, assegnata dall'utente in base all'importanza del job

Il Priority Scheduling può essere sia **preemptive** sia **non-preemptive**

**Contro:**

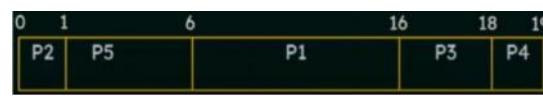
- **Starvation:** un job di **priorità bassa** rimane in **attesa perenne** perché altri job hanno sempre priorità maggiore.  
Come **contromisura** viene usato l'**aging**, l'incremento della priorità del job in base al suo tempo di attesa.

### Esempio

I processi arrivano tutti al **tempo 0**

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

Priorità maggiore chi ha il num. più basso in "Priority"



**Waiting time:**

- $T_{\text{waiting}}$  per P1 = 16 - 0 - 10 = 6
- $T_{\text{waiting}}$  per P2 = 1 - 0 - 1 = 0
- $T_{\text{waiting}}$  per P3 = 18 - 0 - 2 = 16
- $T_{\text{waiting}}$  per P4 = 19 - 0 - 1 = 18
- $T_{\text{waiting}}$  per P5 = 6 - 0 - 5 = 1

**Average waiting time**

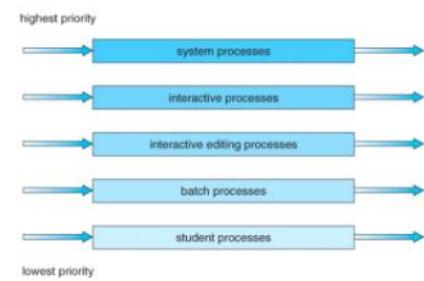
$$T_{\text{waitmedio}} = \frac{6+0+16+18+1}{5} = 8.2$$

## Multilevel Queueing (MLQ)

Algoritmo basato sull'uso di **queue multiple**, ognuna per **categoria** di job, dove ogni queue usa uno scheduling più appropriato per una determinata categoria di job. Una volta inserito in una queue, **nessun job può essere spostato in un'altra**.

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

- **String priority:** nessun job in una queue con priorità bassa viene eseguito finché esiste almeno un job in una queue con priorità alta
- **Round robin:** ogni queue usa un proprio **time slice** che aumenta al diminuire della priorità della coda



## Multilevel Feedback Queueing (MLFQ)

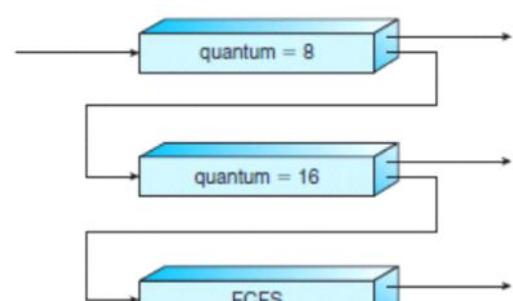
Segue la stessa idea del **MLQ** ma permette **ad ogni job di essere spostato tra le queue**.

Lo spostamento può essere necessario quando:

- Un job passa dall'usare molto la CPU ad usare molto l'I/O e viceversa
- Un job è in **starving**, venendo spostato brevemente in una queue con priorità alta

Per gestire gli spostamenti si usano queste regole:

- Inizialmente ogni job ha la **priorità più alta**
- Se il time slice di un job **scede**, viene spostato in una queue con priorità **inferiore**
- Se il time slice **non scade**, viene spostato in una queue con priorità **superiore**
- La priorità dei processi legati alla **CPU** viene **diminuita** rapidamente
- La priorità di processi legati all'**I/O** rimarrà **alta**



### Esempio Tecnico

Si consideri uno scheduler **MLFQ** con **3 code**, numerate da 0 a 2 (dall'alto verso il basso) come in **figura**. Lo scheduler esegue **prima** tutti i processi nella **coda 0**.

**Solo** quando la coda **0** è **vuota**, lo scheduler eseguirà i processi nella **coda 1**. Allo stesso modo i processi nella **coda 2** verranno eseguiti **solo se le code 0 e 1 sono vuote** e così via.

A un processo nella coda 0 viene assegnato un **time quantum** di 8ms. Se non termina entro questo tempo, viene **spostato** nella **coda 1**.

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

### Esempio Pratico

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	

1

Queue	Time Slice (time units)	jobs
1	1	A <sup>1</sup> , B <sup>1</sup> , C <sup>1</sup>
2	2	
3	4	

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

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	

3

Queue	Time Slice (time units)	jobs
1	1	A <sup>1</sup> , B <sup>1</sup> , C <sup>1</sup>
2	2	A <sup>1</sup> , B <sup>1</sup> , C <sup>1</sup>
3	4	A <sup>1</sup> , B <sup>1</sup> , C <sup>1</sup> , B <sup>2</sup> , C <sup>2</sup>

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	

2

Queue	Time Slice (time units)	jobs
1	1	A <sup>1</sup> , B <sup>1</sup> , C <sup>1</sup>
2	2	A <sup>1</sup> , B <sup>1</sup> , C <sup>1</sup>
3	4	

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	

4

Queue	Time Slice (time units)	jobs
1	1	A <sup>1</sup> , B <sup>1</sup> , C <sup>1</sup>
2	2	A <sup>1</sup> , B <sup>1</sup> , C <sup>1</sup>
3	4	A <sup>1</sup> , B <sup>1</sup> , C <sup>1</sup> , B <sup>2</sup> , C <sup>2</sup> , A <sup>2</sup> , B <sup>2</sup> , C <sup>2</sup>

## Lottery Scheduling

Algoritmo basato sulla **casualità**:

- Ogni job ha un determinato num. di **biglietti**. I job attivi da poco riceveranno **più biglietti**, quelli attivi da molto riceveranno **meno biglietti**. Per evitare la **starvation**, ogni job ha almeno un biglietto.
- Quando un time slice scade viene determinato un **biglietto vincitore** del job che prenderà il controllo della CPU
- Il processo viene ripetuto

Per via della **legge dei grandi numeri**, all'aumentare del num. di estrazioni effettuate, il tempo d'uso della CPU tende a raggiungere la media.

## Exponential Averaging

È una tecnica usata per stimare il val. di una grandezza nel tempo dando **maggior peso a dati più recenti**.

Viene usata per **stimare il tempo di esecuzione dei processi**, in particolare per **SPN e SRT**

# Rilocazione dei processi

martedì 26 novembre 2024

16:20

Per il corretto funzionamento del sistema, dobbiamo **proteggere il sistema operativo dall'accesso da parte dei processi utente**, proteggendo quindi i processi **l'uno dall'altro** per fare in modo che non vadano a scrivere in zone di memoria di altri processi.

Durante la **rilocazione dei processi** viene assunto che l'**OS** sia allocato nella **parte più alta della mem.** e che gli indir. logici generati da ogni programma siano compresi tra l'**indir. più basso e l'ultimo indir. utilizzabile** (dunque **mem\_size - os\_size - 1**).

Ogni processo viene caricato in mem. allocandolo nel **primo segmento di mem. contiguo vuoto** in cui il processo può essere contenuto.

## Rilocazione statica (run-time)

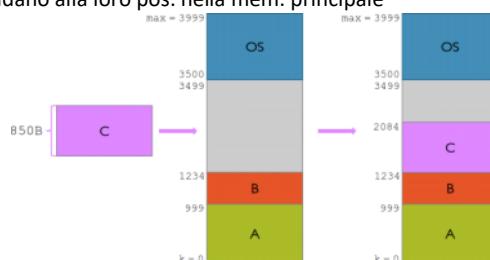
Prevede che l'avviatore dell'OS riscriva gli indir. generati da un processo in modo che essi corrispondano alla loro pos. nella mem. principale (**associazione in fase di avvio**)

**Pro:**

- Non necessita ulteriore supporto HW

**Contro:**

- Troppo overhead
- Non vi è mutua protezione tra i processi (ognuno di essi può corrompere l'OS o altri processi)
- Lo spazio di indir. deve essere allocato in modo **contiguo**
- Una volta allocato in mem., un processo **non può essere spostato** dall'OS



La **rilocazione statica** però non protegge gli spazi di memoria dei processi.

È necessario aggiungere questa protezione a **livello HW**.

## Rilocazione dinamica

Si basa sulla protezione reciproca dell'OS e dei processi, necessitando di una **Memory Management Unit (MMU)**, cioè l'**unità di gestione della memoria** tramite cui associare gli indir. in fase di **esecuzione**.

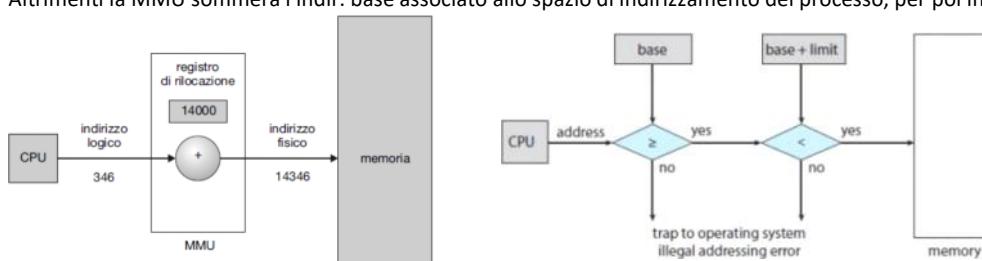
La **MMU traduce dinamicamente** ogni indir. logico/virtuale generato da un processo nel corrispettivo indir. fisico.

Essa contiene almeno:

- **Un registro base:** contenente l'indir. fisico iniziale dello spazio di indir.
- **Un registro limite:** contenente la dim. max. dello spazio di indir.

Tramite tali registri, ogni processo può **accedere solo alle zone** di mem. appartenenti al **segmento assegnato**:

- Ogni processo **crede** che il proprio primo indir. fisico sia 0
- Durante una richiesta di accesso alla mem., la MMU controlla che l'indir. richiesto dal processo **sia all'interno dello spazio** di indir. del processo
  - Se l'indir. richiesto **superà il limite** o è **inferiore al registro base**, la MMU restituisce un errore
  - Altrimenti la MMU sommerà l'indir. base associato allo spazio di indirizzamento del processo, per poi inviare l'indir. alla mem.

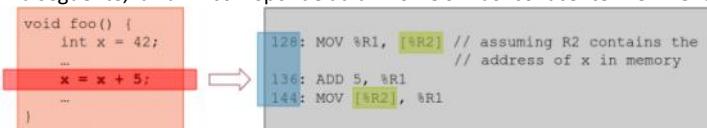


# Gestione Memoria

martedì 26 novembre 2024 13:17

Molte istr. coinvolgono il prelevamento/salvataggio di dati in memoria. I vari chip di mem. rispondono solo ad **indirizzi fisici**, dunque è necessario effettuare una **connessione tra nomi simbolici** descritti dai programmi utente e i **veri indirizzi fisici** di mem.

Nel programma seguente, la var. `x` corrisponde ad un nome simbolico facente riferimento ad un **indirizzo logico** contenente sia istr. che dati



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

Un **nome simbolico** è un **riferimento simbolico alla mem.** usato dai programmi utente.

Tale nome viene poi convertito in **indirizzo logico** (indir. di mem. generato tramite CPU), il quale viene a sua volta convertito in un **indirizzo fisico** (l'effettivo indir. di mem.)

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**

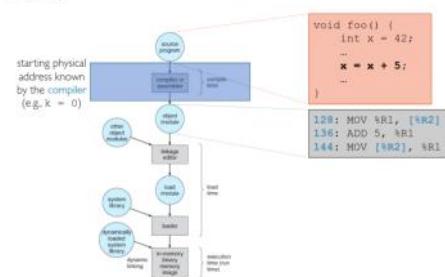
## Associazione degli indirizzi

### Associazione in fase di compilazione

Se l'indir. fisico dove un programma risiede in mem. **è noto** in fase di compilazione, allora il compilatore genera del **codice assoluto**, richiedente **nessun intervento dell'OS** e dove l'indir. logico e fisico **coincidono**.

Se la posizione di partenza cambia, bisogna **ricompilare** il codice.

Esempio:



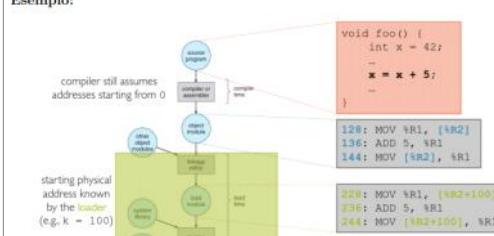
### Associazione in fase di avvio

Se l'indir. fisico dove un programma risiede in mem. **non è noto** in fase di compilazione, allora il compilatore genera del **codice statico rilocabile**, che fa riferimento ad indir. relativi all'indir. iniziale.

L'**avviatore di processi** dell'OS determina l'indir. fisico iniziale di ogni processo. Anche qui, l'indir. logico e fisico **coincidono**.

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

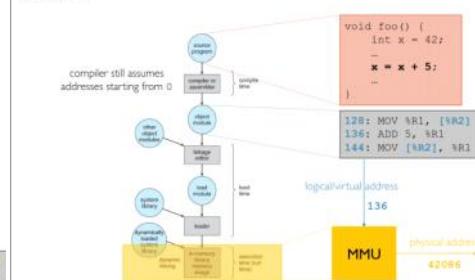
Esempio:



### Associazione in fase di esecuzione

Se il programma può essere mosso nella mem. principale durante l'esecuzione, il compilatore genera degli **indir. virtuali** (o **codice dinamico rilocabile**). L'OS si occuperà di riassociare gli indir. fisici tramite una **MMU**. In questo caso l'indir. logico (o virtuale) e fisico **non coincidono**

Esempio:



A seconda dell'architettura implementata, si usano diverse **tecniche di gestione della mem.:**

- **Gestione della mem. uni-programmabile:**
  - All'OS viene assegnata una **parte fissa di mem. fisica**
  - Viene eseguito un **singolo processo per volta**, il quale possiede sempre lo stesso indir. iniziale (quindi l'associazione tra indir. logico e fisico avviene nella **compilazione**)
- **Gestione della mem. multi-programmabile:**
  - Più processi coesistono in contemporanea nella mem. e processi cooperanti condividono porzioni dello spazio di indir.
  - I processi non conoscono la porzione di mem. in cui si trovano e non sanno della condivisione della mem.
  - I processi non possono accedere ai dati di altri processi e dell'OS, rendendo impossibile la loro corruzione
  - Le performance della CPU non diminuiscono molto durante la condivisione e vi è poca frammentazione dei dati

# Policy di allocazione

martedì 26 novembre 2024 16:33

Un metodo semplice per gestire uno spazio contiguo di mem. fisica consiste nel dividere la mem. dedicata ai processi in **partizioni di dimensione equivalente**, assegnando **ogni partizione ad un processo**.

Esistono 2 tipi di **partizionamento**:

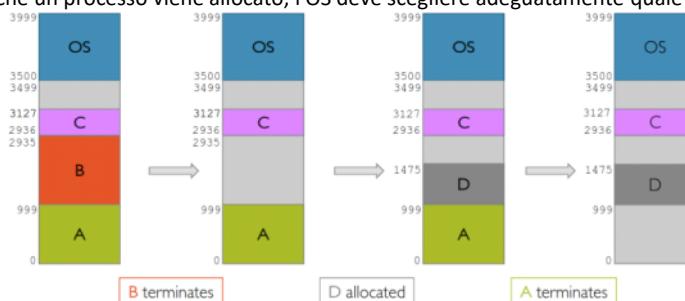
- Partizionamento **fisso**, che si divide in:
  - Partizionamento **fisso statico**: la mem. viene divisa in partizioni di dim. uguali
    - Un programma potrebbe **non entrare** in una partizione
    - **Frammentazione interna**: un programma piccolo occupa un'intera partizione e lo spazio vuoto è inutilizzabile da altri processi
  - Partizionamento **fisso variabile**: la mem. viene divisa in partizioni di grandezza crescente (es. 2M, 4M, 8M, ecc)
    - Mitiga i problemi del partizionamento fisso ma non li risolve
    - Un programma piccolo entra nella partizione minore grande abbastanza per esso
    - **Una coda per ogni partizione o una per tutte**
    - Se ci sono **molti processi piccoli** la mem. è usata in modo inefficiente
- Partizionamento **dinamico**: le partizioni variano in **misura e quantità**, ad ogni processo è allocata esattamente la quantità di mem. che serve
  - **Frammentazione esterna**: risolvibile con la **compattazione** (l'OS sposta i processi per renderli contigui) (elevato overhead)

Un altro approccio è **far tenere traccia** all'OS dei **segmenti di mem. liberi** (inutilizzati), detti **buchi**.

Consideriamo la seguente situazione:

- 1) I processi A, B e C sono allocati nella mem.
- 2) Successivamente, B **termina**, lasciando un **buco** nella mem. al suo posto
- 3) In seguito, viene allocato D nel **buco lasciato da B**, creando a sua volta un **ulteriore buco**, minore rispetto al precedente

Ogni volta che un processo viene allocato, l'OS deve scegliere adeguatamente quale buco di mem. riempire



Quando non c'è sufficiente memoria libera possiamo:

1. **Rifiutare il processo** e fornire un messaggio d'errore
2. **Piazzare i processi in una coda d'attesa**, dalla quale, quando la mem. tornerà libera, il sistema determinerà quale processo può essere caricato.

Quando arriva un nuovo processo, il sistema **scansiona la mem.** alla ricerca di un **buco di mem. abbastanza grande** per il processo.

Quando un processo termina, **rilascia i suoi blocchi di memoria** che ritornano a far parte dell'insieme di buchi di mem.

Esistono molte soluzioni per allocare i processi nella memoria, vediamo i più comuni:

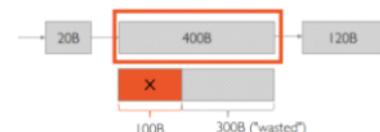
## Policy first-fit

Prima dell'allocazione viene effettuata una **scansione lineare** della mem. finché non viene trovato un **buco abbastanza grande** per il processo.

Il processo viene allocato nel **primo buco disponibile**.

### Esempio

- Abbiamo tre buchi in mem. rispettivamente 20, 400 e 120 byte
- Il processo X richiede 100 byte. Tramite la policy first-fit, il processo viene inserito nel **primo buco sufficiente**, ossia quello da 400 byte, creando un buco da 300 byte
- Il processo Y richiede 350 byte. L'OS non può soddisfare la richiesta perché non c'è buco abbastanza grande



## Policy best-fit

Prima dell'allocazione viene effettuata una **scansione lineare di tutti i buchi** in mem. Viene poi selezionato il **minor buco abbastanza grande** per il processo, **riservando i buchi** di dimensione **maggiori** per i processi richiedenti più mem.

Per tale policy **può** portare alla **creazione di buchi** di dim. **troppo piccola** per poter essere allocata da qualsiasi processo, rendendo quegli **spazi inutilizzabili**.

### Esempio

- Abbiamo tre buchi in mem. rispettivamente 20, 400 e 120 byte
- Il processo X richiede 100 byte. Tramite la policy best-fit, il processo viene inserito nel **buco minore sufficiente**, ossia quello da 20 byte, creando un buco da 100 byte
- Il processo Y richiede 350 byte. Esso verrà allocato nel buco da 400 byte, creandone uno di 50
- La somma dei buchi rimasti è 90 byte. Tuttavia essendo divisi in buchi di **piccoli dimensioni**, difficilmente un processo sarà abbastanza piccolo da poter essere allocato in uno di quei buchi, lasciandoli inutilizzabili



## Policy worst-fit

Prima dell'allocazione viene effettuata una **scansione lineare di tutti i buchi** in mem. Viene poi selezionato il **maggior buco abbastanza grande** per il processo.

Contro intuitivamente, questa policy **aumenta la probabilità** che i **buchi rimanenti** siano **abbastanza grandi** per altri processi

#### Esempio

- Abbiamo tre buchi in mem. rispettivamente 20, 500 e 380 byte
- Il processo X richiede 100 byte. Tramite la policy worst-fit, il processo viene inserito nel **buco maggiore sufficiente**, ossia quello da 500 byte, creando un buco da 400 byte
- Il processo Y richiede 350 byte. Esso verrà allocato nel buco da 400 byte, creandone uno di 50 byte
- I buchi rimasti sono relativamente più grandi rispetto alla best-fit

# Frammentazione

martedì 26 novembre 2024 17:05

## Frammentazione interna ed esterna

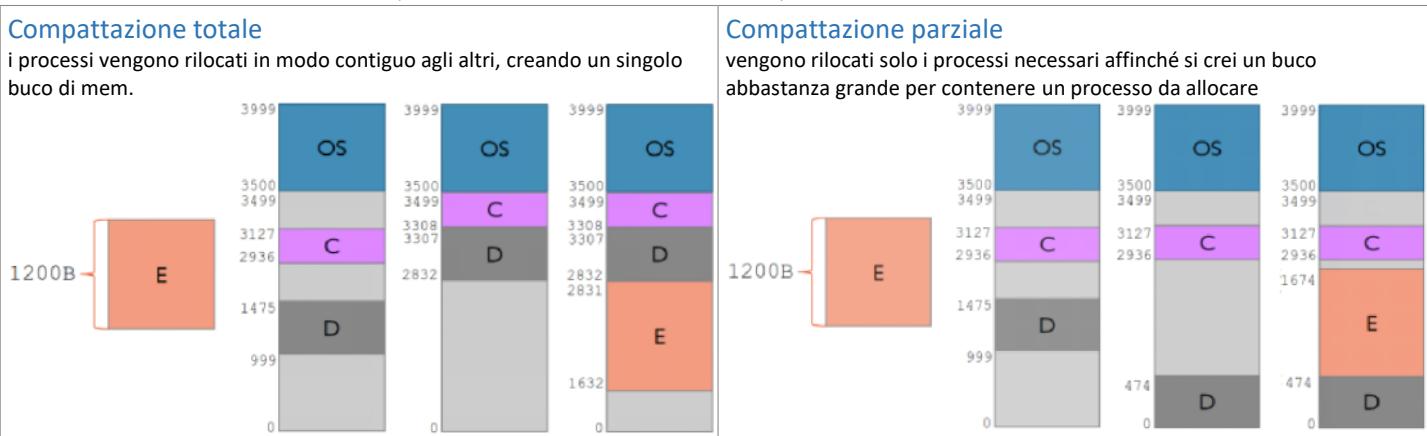
La **frammentazione di mem.** è la presenza di **singoli buchi troppo piccoli** per essere usati da un processo, ma **abbastanza grandi se fossero combinati**. Vi sono due tipi di frammentazione:

- **Frammentazione esterna**, dove vi è abbastanza spazio per allocare il processo, ma è **diviso in buchi di mem. non contigui**, impedendo l'allocazione. Si verifica se si usa un'**allocazione dinamica** della mem. a seguito del frequente allocamento e de-allocamento di processi
- **Frammentazione interna**, dove viene **sprecata** parte di mem. a seguito dell'allocazione di settori **troppo grandi** per un processo richiedente meno spazio di quello fornito. Si verifica se si usa un'**allocazione statica** della mem. tramite la definizione di **partizioni fisse**

## Risolvere la frammentazione

Se viene usata una **allocazione statica dei settori**, il problema della **frammentazione interna** è **irrisolvibile**. Tuttavia può essere **parzialmente mitigata** creando settori di dim. **crescente**, permettendo ai processi meno costosi di essere allocati nei settori più piccoli, diminuendo la quantità di mem. sprecata.

Se viene usata un'**allocazione dinamica**, il problema della **frammentazione esterna** può essere risolta in due modi:



# Swapping e paginazione

martedì 26 novembre 2024 17:18

## Swapping

Per essere eseguito in CPU, un processo deve essere caricato in mem. principale.

Tuttavia, se il processo si blocca in attesa di un evento, è possibile **spostarlo dalla mem. primaria a quella secondaria**. Quando si verifica l'evento, il processo è nuovamente pronto all'esecuzione e viene spostato nella mem. principale.

Tale tecnica è detta **swapping**.

A causa dell'interazione con la mem. secondaria, lo swapping è un processo **molto lento** ma **molto efficiente per risolvere la frammentazione** eseguendo una compattazione prima di spostare il processo in mem. principale.

La **modalità di gestione** dello swapping deriva dalla modalità di gestione dell'associazione di indir.:

- Se viene usata un'associazione in **compilazione** o all'**avvio**, il processo deve essere spostato nella stessa identica pos. di mem. in cui si trovava
- Se viene usata un'associazione in **esecuzione**, il processo può essere spostato in qualsiasi pos. di mem.

A causa della sua lentezza molti OS preferiscono alternative più rapide come la **paginazione**

## Paginazione

La **paginazione divide in sezioni (page) il processo**, eseguendo lo **swap** solo sulle **page** singole invece che sull'intero processo.

Usa uno **spazio di indir. logico contiguo suddiviso in blocchi di dim. fissa**, detti **pagine**, rendendo non più necessario che gli indir. siano contigui, poiché le pagine possono essere **mappate a frame fisici non contigui**. Inoltre, viene **eliminata la frammentazione esterna** e il bisogno di **compattazione**.

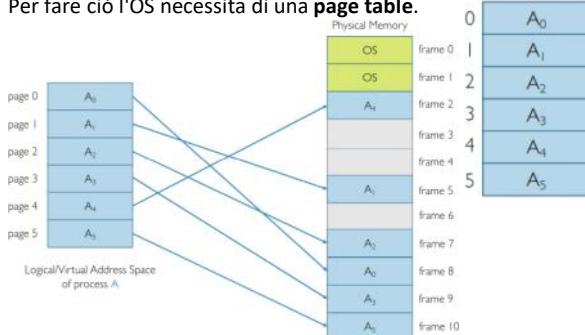
L'implementazione di base divide la mem. **fisica in blocchi di dim. fissa (frame)**, e divide la mem. **logica in blocchi delle stesse dim. (page)**.

Quando un processo deve essere eseguito le sue **page vengono caricate nei frame liberi** dalla sorgente.

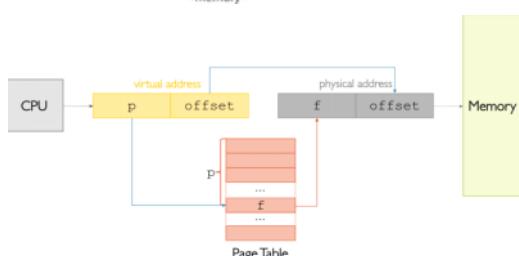
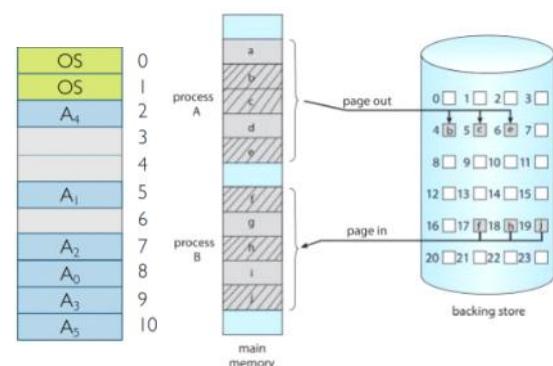
L'OS dovrà occuparsi di:

- **mappare** le pagine logiche e i frame fisici
- **tradurre** gli indir. logici in fisici.

Per fare ciò l'OS necessita di una **page table**.



Page Table	
Page	Frame
0	8
1	5
2	7
3	9
4	2
5	10



Ogni **indir. virtuale** è costituito da:

- **page number p**: num. della pagina dove gli indir. risiedono
- **offset** di word relativo all'inizio della pagina.

Analogamente ogni **indir. fisico** è costituito da un **num. di frame f** e un **offset di word** relativo all'inizio del frame.

Durante la conversione da indir. virtuale a fisico, il num. di frame f dell'indir. fisico corrisponderà a quello indicato dalla pagina p nella page table, mentre l'offset corrisponde all'offset virtuale stesso.

Passaggi della **MMU** per tradurre gli indir. logici in fisici:

- 1) Viene estratto il **num. di pagina p** e usato come **indice nella page table**
- 2) Viene estratto il **corrispettivo num. del frame f** dalla **page table**
- 3) Si **rimancia** il num. di pagina p con il num. del frame f  
(l'offset rimane invariato)

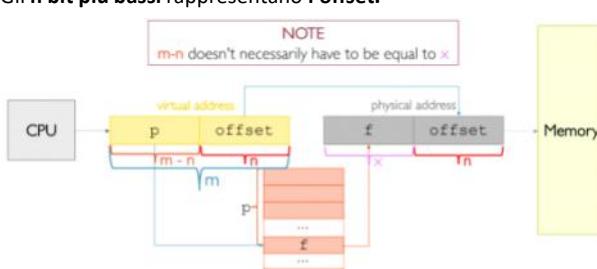
La paginazione è una forma di **rilocazione dinamica** dove ogni indir. virtuale è legato ad uno fisico tramite la page table che può essere interpretata come un insieme di registri base (uno per frame)

La grandezza e il num. di pagine/frame è determinata dall'architettura.

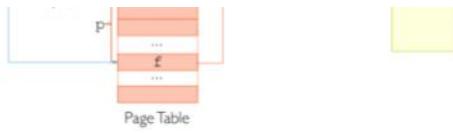
Assumiamo che un indir. **virtuale** è formato da **m** bit (spazio indirizz.  $2^m$ ) e di avere **page di grandezza  $2^n$**  con  $n < m$ .

Gli **m - n** bit più alti dell'indir. virtuale indicano il **num. di pagina**.

Gli **n** bit più bassi rappresentano l'**offset**.



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
  - 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 ( $2^3 = 8$ )
2.
  - 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
  - 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 ( $2^3 = 8$ )



- 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

## Translation Look-aside Buffer (TLB)

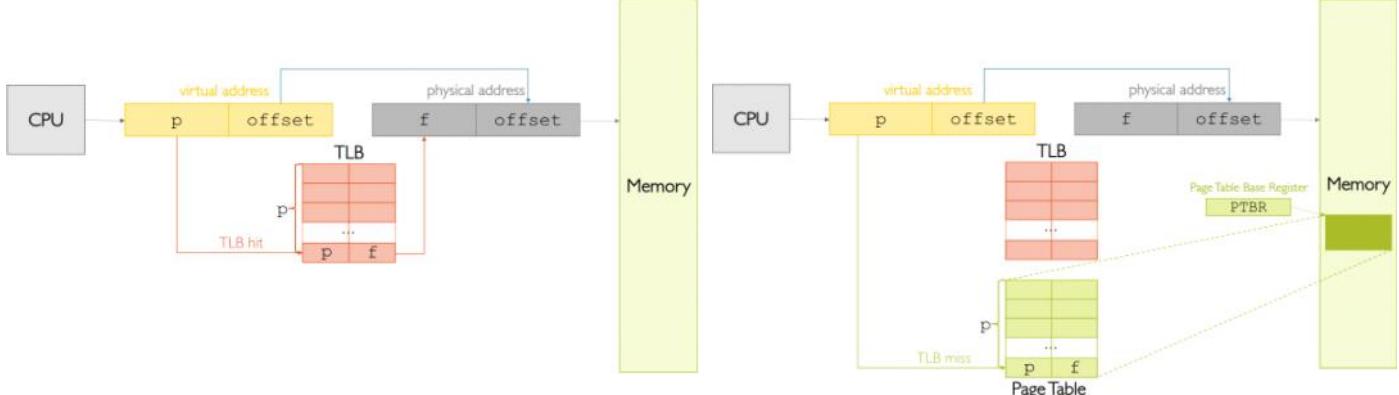
Un TLB è una **cache** di primo livello molto **veloce** usata per memorizzare i num. di page e i frame calcolati **precedentemente** (tipo una "barra dei segnalibri" con numeri calcolati più frequentemente o più vicini a numeri già calcolati)

La cache è **più veloce ma più piccola** di una mem. **normale**, quindi la TLB conterrà solo **una parte** dei val. di una **page table**

Se viene richiesto un indir. virtuale:

- Si controlla se il num. di page associato all'indir. è già nel TLB
  - Se sì (TLB hit)** il frame associato viene **restituito subito**
  - Altrimenti (TLB miss)** il controllo viene effettuato come di norma sulla **page table** per poi caricare il dato letto nella TLB per futuri utilizzi

La TLB è **condivisa tra tutti i processi**, e lo stesso num. di page può essere mappato ad un **differente num. di frame** in base al processo



Il problema però è che i processi potrebbero richiedere molta mem. nella RAM.

## Mapping Associativo

Il TLB contiene solo **alcuni elementi** tratti dalla page table, il num. di pagine non può essere usato come indice per il TLB

Bisogna fare in modo che il TLB contenga solo pagine in RAM, quindi se l'OS swappa una pagina nella mem. secondaria, deve eliminarla dal TLB

Oltre alla paginazione segmentata, gli OS moderni usano altre forme di paginazione più avanzate:

- Paginazione a più livelli o gerarchica:** una page table è divisa in più page table (paginazione della page table)
- Paginazione con hash:** si usano delle hash table per indicizzare più page table sparse in modo veloce
- Page table invertita:** vengono conservati i num. di tutti i frame attualmente caricati in mem

## Paginazione Gerarchica (slide prof. solo a 2 livelli)

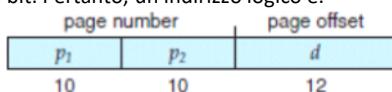
Una soluzione all'allocazione nella RAM è di **dividere la page table in parti più piccole**.

Il modo più semplice per effettuare questa divisione è il **two-level paging algorithm**, in cui viene **paginata la page table stessa**.

Ad esempio, consideriamo il sistema con uno spazio di **indir. logici** a 32 bit e una dim. della **page** 4KB.

Un indir. logico è suddiviso in un **page number** composto da **20 bit** e un **page offset** composto da **12 bit**.

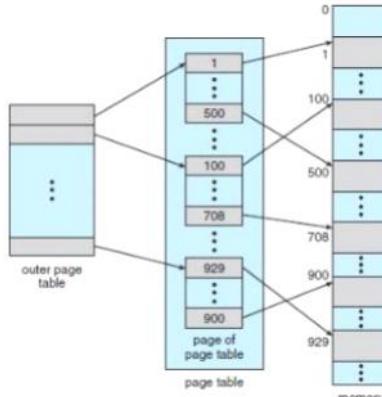
Poiché **impaginiamo la page table**, il num. di pagina è **suddiviso** in un num. di pagina a 10 bit e un offset di pagina a 10 bit. Pertanto, un indirizzo logico è:



Dove:

- $p_1$**  è l'indice all'interno della **page table esterna**
- $p_2$**  è l'**offset** all'interno della **page table interna**

Per le architetture a 64 bit, la **paginazione gerarchica** sono considerate **inappropriate**.



## Paginazione con Hash (non visto nelle slide)

Per una architettura maggiore di 32 bit, viene utilizzata l'**hashed page table**. Con il val. **hash** che è il

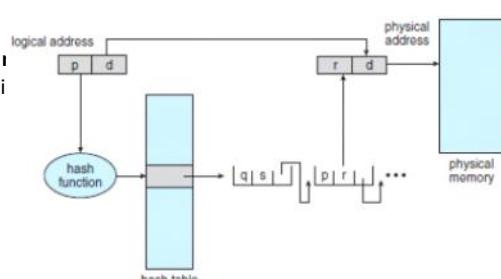
Ogni voce nella **tabella hash** contiene una **linked list** di elementi che hanno l'hash nella stessa posiz

Ogni elemento è composto da:

- num. pagina virtuale**
- val. della page frame mappata**
- puntatore all'elemento successivo nella linked list**

Il num. della **pagina virtuale** nell'indirizzo logico passa per una **funzione hash** e viene cercato nella **hash table** dove viene **confrontato** con il primo campo della **linked list**.

- Se c'è **corrispondenza**, il **page frame** viene usato per formare l'indir. fisico
- Se non c'è **corrispondenza**, vengono controllati gli elementi successivi nella linked list



La **clustered page tables** è una variazione di questo schema, utile per spazi di indirizzi a 64 bit.

Esse sono simili alle **hashed page tables** tranne per il fatto che ogni voce nella tabella hash fa riferimento a diverse pagine anziche una sola. Sono utili per spazi di indirizzi sparsi (riferimenti di mem. non contigui)

## Paginazione Invertita (non visto nelle slide)

Normalmente ogni processo ha una page table associata dove vi è una voce per ogni page usata dal processo riferita tramite indirizzo virtuale.

Il sistema deve tradurre questo riferimento in un indir. fisico e poiché la tabella è ordinata per indir. v.

Il problema principale è che una **page table** può essere composta da **milioni di voci**, che consumano

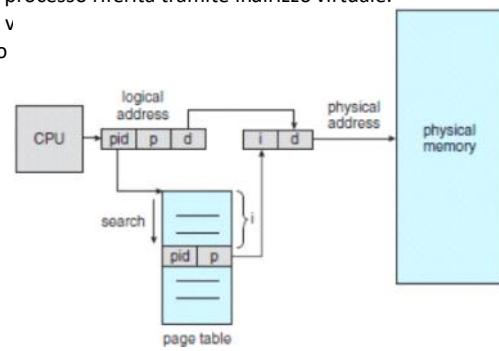
Per risolvere questo problema, si può usare l'**inverted page table**, dove vi è una **voce di mem.** per **ogni pagina reale (frame)**.

Ogni voce è costituita dall'**indir. virtuale** della **pagina memorizzata** in quella posizione di mem. reale, con informazioni sul processo che possiede la pagina.

Nel sistema è presente **solo una page table** e ha una sola voce per ogni page di mem. fisica.

Le inverted page table richiedono che in ciascuna voce della tabella delle pagine sia memorizzato un **identificatore** dello **spazio degli indirizzi**, poiché la tabella di solito **contiene diversi spazi degli indirizzi** diversi che mappano la mem. fisica.

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



### Esempio

Esempio semplificato, dove ogni **indir. virtuale** è costituito da:

$\langle PID, \text{page-number}, \text{offset} \rangle$

Ogni voce della **inverted page table** è una coppia  $\langle PID, \text{page-number} \rangle$ , dove il process-id è l'**identificatore** dello spazio di indirizzi.

Per effettuare un riferimento alla memoria, la parte dell'**indir. virtuale** costituita da  $\langle PID, \text{page-number} \rangle$  viene cercata nella **page table**.

- Se si **trova** una **corrispondenza**, ad esempio alla voce i, viene generato l'**indir. fisico**  $\langle i, \text{offset} \rangle$
- Altrimenti, è stato tentato un accesso illegale all'indir.

# Segmentazione

mercoledì 27 novembre 2024 12:04

È più comodo (per i programmati) immaginare il programma salvato in mem. in modo **non contiguo** e diviso in **segmenti (spazi di indirizzi)**, ognuno dedicato ad un **uso specifico** (dati, codice, ...).

La tecnica di **segmentazione della memoria** facilita questo ragionamento, basandosi su indir. costituiti da un **num. di segmento** e un **offset** dall'inizio del segmento indicato.

## Segment table

Per collegare il segmento al suo spazio fisico vengono **mappati gli indir. bidimensionali** (segmento) definiti dall'utente in **indirizzi fisici unidimensionali**.

Questa mappatura è effettuata da una **segment table** che collega ogni **indir. segmento-offset** ad uno fisico, controllando se l'indir. sia valido o no.

Il sistema usato per la segment table è simile al funzionamento della page table e dei registri base di rilocazione della MMU e ogni processo ha una segment table.

Ogni **entra** della segment table ha un **indir. base** e la **lunghezza del segmento** e informazioni aggiuntive per la protezione di esso (es. permessi r/w).

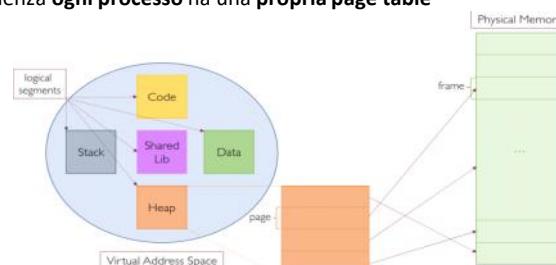
A differenza della page table (che potrebbe contenere troppe pagine), la segment table si può realizzare con pochi registri base e limite.

Rispetto alla paginazione, la segmentazione è più comoda per la condivisione dei dati, rendendo però inefficiente l'uso della mem.

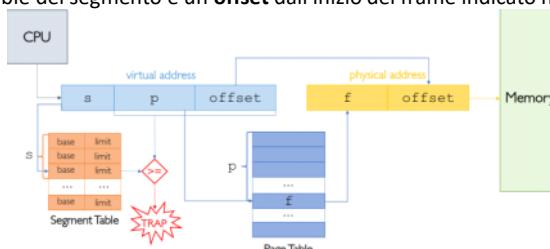
La soluzione ottimale è quella di **applicare la paginazione ai segmenti stessi**.

Questa tecnica è detta **paginazione segmentata**:

- Lo spazio di **indir. virtuale** di ogni processo è visto come una **collezione di segmenti**
- Lo spazio di **indir. fisico** è visto come una **sequenza di frame di dim. fissa**.
- Di conseguenza **ogni processo** ha una **propria segment table**
- I segmenti sono in genere più grandi dei frame fisici, necessitando che ogni segmento logico debba essere mappato a più frame.
- Di conseguenza **ogni processo** ha una **propria page table**



- Affinché la traduzione da indir. virtuale a fisico sia possibile, ogni indir. deve essere composto da un **num. di segmento**, un **num. di pagina** legato alla page table del segmento e un **offset** dall'inizio del frame indicato nella page table.



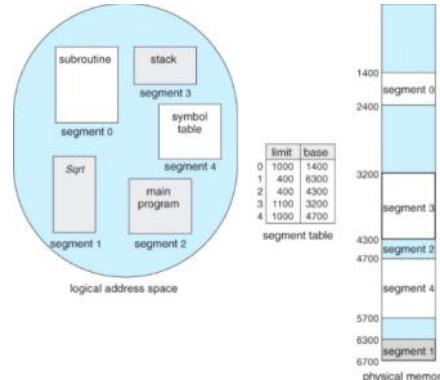
Per tenere traccia della segment table e della page table dei vari processi si usano due soluzioni:

- Vengono usati pochi registri per la segment table e le page table vengono salvate in mem. principale con aggiunta di una TLB.  
Questo soluzioone è **più veloce** ma pone un **limite al num. di segmenti**
- La segment table e la page table vengono salvate in mem. principale con aggiunta una TLB, i cui controlli vengono effettuati usando un indice dei segmenti e uno di pagine.  
Questo soluzioone è **più lenta** ma **più flessibile**

## Vantaggi della Memoria Segmentata

- **Pro**
  - Il compiler e l'OS vedono la mem. in modo analogo
  - Flessibile
  - Assenza di frammentazione esterna
  - Condivisione della mem tra i processi
- **Contro**
  - Context switch e traduzione degli indir. più lenta
  - Frammentazione interna ancora esistente

## Paginazione e Segmentazione



Ad esempio, il segmento 2 è lungo 400 byte e inizia alla posizione 4300. Pertanto, un riferimento al byte 53 del segmento 2 viene mappato sulla posizione  $4300 + 53 = 4353$ .

Un riferimento al segmento 3, byte 852, viene mappato a 3200 (la base del segmento 3) + 852 = 4052.

Un riferimento al byte 1222 del segmento 0 comporterebbe una trap per il sistema operativo, poiché questo segmento è lungo solo 1.000 byte.

La **paginazione** non è visibile al programmatore/compilatore, mentre la **segmentazione** lo è. Ogni segmento viene diviso in più pagine

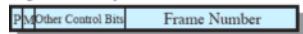
Virtual Address



Segment Table Entry

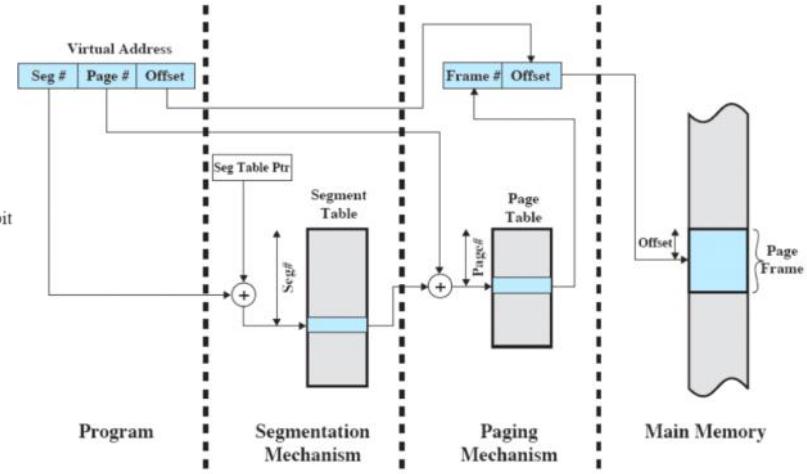


Page Table Entry



Anche il Segment Table Entry contiene i bit P e M

P= present bit  
M = Modified bit



# Memoria virtuale

mercoledì 27 novembre 2024 12:43

Poiché la maggior parte dei processi non necessita tutte le pagine caricate in RAM contemporaneamente, le **pagine non utilizzate** possono essere **archiviate temporaneamente** nella mem. secondaria, dando l'illusione di uno spazio di indirizzamento virtuale di dim. infinita, permettendo a **più programmi** di essere **attivi senza** che venano **caricati interamente** nella **memoria principale**.

In ogni momento, ogni pagina può essere nella mem. principale (RAM) o secondaria (disco rigido). Questa tecnica è detta **memoria virtuale**.

Le pagine che sono nella mem. principale vengono chiamate **resident set (num. di frame di RAM)**.

La dimensione del resident set può essere:

- **allocazione fissa**: il num di frame è deciso al tempo di creazione del processo
- **allocazione dinamica**: il num. di frame varia durante la vita del processo

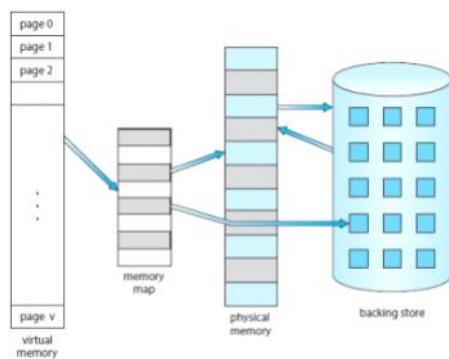
La possibilità di eseguire un programma che è **solo parzialmente in mem.** conferisce molti vantaggi:

- I programmi non sono vincolati alla quantità di mem. fisica
- Esecuzione di più programmi in contemporanea e aumento delle prestazioni
- Necessario meno I/O per caricare/scambiare porzioni di programmi in mem.

Ogni processo ha una **sua page table** e il PCB punta a tale tabella.

Alla base di questa tecnica vi è la seguente idea:

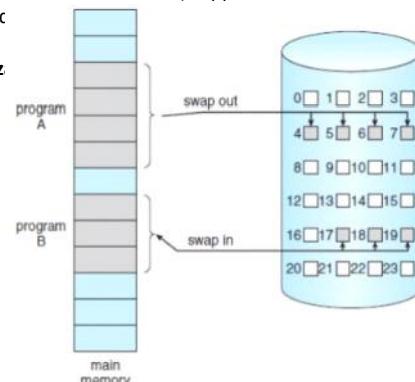
- La mem. principale ha il ruolo di "cache" del disco rigido
- La **page table** deve **indicare** se ogni pagina si **trova nella mem o nel disco** (un singolo **bit di appartenenza** è sufficiente). Quando una pagina si sposta tra Ram e disk (o viceversa), l'OS aggiorna l'entrata corrispondente della page table, modificandone il bit indicante la mem. di appartenenza
- Quando si effettua un **riferimento logico** si controlla la page table
  - Se il bit. di appart. è **1 (valid)**, la page richiesta sta nella RAM
  - Altrimenti il bit è **0 (invalid)** viene attivata una **page fault trap**, imponendo che la pagina venga caricata dal disco
- Poiché l'accesso al disco è più lento, gli accessi alla mem. devono far riferimento a **pagine che con alta probabilità sono in mem.**, evitando di andare sul disk
- Tramite la **regola del 90/10** (che afferma che un processo spende il 90% del tempo ad accedere solo al 10% della sua mem. allocata) sappiamo che la maggior parte dei riferimenti alla mem. si trova in una **piccola zona di mem.** che definiamo come **working set** c
- Tale zona è **moltissima piccola** e la probabilità che possa essere caricata in mem. è molto alta.  
Durante l'esecuzione del processo, il working set può **variare**. Tuttavia, tale variazioni avvengono con **frequenz** working set rimarrà lo stesso



## Demand Paging vs Prepaging

Esistono 2 politiche per recuperare le page dal disco:

- **Prepaginazione**: porta più pagine di quelle richieste, ma sempre vicine a quella richiesta
- **Recupero a richiesta**: un processo comunica all'OS quando necessita che una page venga recuperata
  - molti page fault nei primi momenti di vita del processo



Il **recupero a richiesta** (demand paging) è il sistema usato da OS moderni. Quando un proc. è avviato, **nessuna** delle sue page è caricato in mem., venendo spostate solo quando il **processo richiede di accedervi** (tramite **page fault**).

Tale sistema di recupero viene detto **lazy swapper** o **pager**.

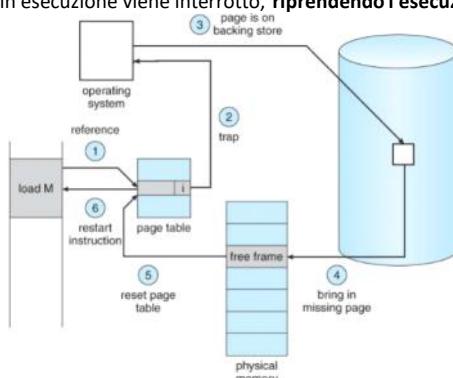
La **placement policy** decide dove mettere una pagina in mem. principale quando c'è un frame libero.

Se non ci sono frame liberi, allora la **replacement policy** si occupa di liberarne uno in base alle policy usate.

## Page Fault

Se si verifica un **page fault**, quindi la pagina non si trova nella page table ma in mem., si esegue:

- 1) Si controlla il **bit di appartenenza** della page corrispondente all'indir. richiesto
- 2) Se è impostato a **0**, si attiva la **page fault trap**, e la page deve essere recuperata dal disco
- 3) Si individua un **frame libero** nella RAM a cui mappare la page recuperata
- 4) Si schedula un'**operazione su disco** per poter **sostituire** la **page** richiesta, mettendo il processo in attesa di un I/O e permettendo ad un altro processo di essere schedulato
- 5) Completata l'op. di I/O, si **aggiorna** l'entrata della page table con il num. del **frame** individuato e il suo bit di appartenenza viene impostato a **1**
- 6) Il processo in esecuzione viene interrotto, **riprendendo l'esecuzione del processo precedente** e rieseguendo l'istr. tramite cui è stato attivato il page fault



Se si usa una **TLB** essa dovrà avere un **bit di appartenenza**, con 1 se la page è nella RAM o 0 se sta nel disco.

Il funzionamento della TLB, in questo caso, funziona in modo leggermente diverso :

- Ottenere una **TLB hit** nella mem. virtuale implica che la **page** si trova nella **cache** e che il **frame** di riferimento è in **mem**
- Se si ottiene una **TLB hit** ma il **frame** di riferimento **non è attualmente caricato in mem**, bisogna recuperare la page dal disco
- Se la **page** richiesta **non è nella cache (TLB miss) ma è in mem.**, l'OS sceglie un'entrata della TLB da rimpiazzare con la **page** richiesta
- Se la **page** richiesta **non è nella cache (TLB miss) e neanche in mem.**, l'OS sceglie un'entrata da invalidare, e poi effettua le op. della **page fault trap**, aggiornando l'entrata della TLB con i dati della **page** recuperata. Infine, viene rieseguita l'istr.

Un'istr. viene detta **idempotente** se può essere eseguita più volte con lo stesso effetto di una singola esecuzione.

Se si verifica un **page fault** a causa di un'istr. **idempotente** allora tale istr. viene semplicemente **rieseguita** dopo la gestione del page fault. Se l'istr. **non è idempotente**, la riesecuzione dell'istr. sarà più difficile.

## Performance

I processi sono soggetti al **principio di località dei riferimenti**, dove se un proc. **accede ad un indir. in mem.**, è molto **probabile** che **accederà** nuovamente allo stesso indir. (**principio temporale**) e che acceda a indir. **molto vicini ad esso (principio spaziale)**.

In particolare:

- **$t_{MA}$ : tempo** impiegato ad accedere alla **mem. fisica**
- **$t_{FAULT}$ : tempo** impiegato a gestire un **page fault**
- **$p \in [0, 1]$ : probabilità** che si verifichi un **page fault** (ci aspetta)

Il **tempo totale impiegato** per un accesso alla mem. è:

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

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  $p = 0.0001 = \frac{1}{10000}$ . In tal caso si ha che:

$$t_{ACCESS} = (1 - \frac{1}{10000}) \cdot 100\text{ns} + \frac{1}{10000} \cdot 20000000\text{ns} = \frac{9999}{100}\text{ns} + 20000\text{ns} \approx 20.1\mu\text{sec}$$

- 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  $\varepsilon > 0$  tale che

$$t_{ACCESS} = (1 + \varepsilon) \cdot t_{MA}$$

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

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

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

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

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

E se volessimo che l'access time sia al massimo il 10% più lento dell'accesso alla memoria di base?

Dobbiamo risolvere per  $p$  la seguente equazione:

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

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

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

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

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

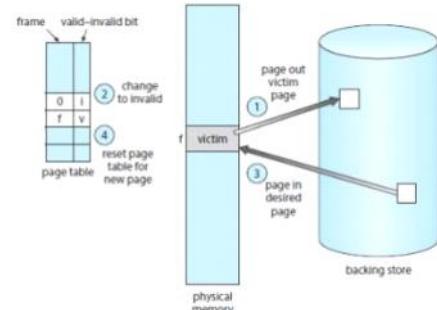
# Recupero e rimpiazzo pagine

venerdì 29 novembre 2024 11:44

Se deve essere caricata più di una page dal disco ma **nessun frame è libero** (mem. fisica è piena), è necessario rimpiazzare delle page caricate.

La routine della **sostituzione della pagina** è:

- 1) Trova la **pos. desiderata** nella mem. secondaria
- 2) Trova un **frame libero**:
  - a. Se c'è un frame libero, **viene usato**
  - b. Se non c'è alcun frame libero, si usa un **algoritmo di sostituzione** per selezionare un **victim frame** dalla mem. principale
  - c. Se necessario **scrivi il victim frame** nella mem. secondaria, modificando le page e le frame table
- 3) **Leggi la pagina** desiderata nel **frame appena liberato**, modificando le page e le frame table
- 4) Continua il processo dal punto in cui si è verificato il **page fault**



## Algoritmi di Sostituzione

Gli algoritmi più usati sono:

### Rimpiazzo casuale

Rimuove una **page a caso**

### FIFO (First In, First Out)

Rimuove la **page rimasta** in mem. **per più tempo**.

Viene usata una coda a cui rimuoviamo la **prima pagina** (la più vecchia) e inseriamo la pagina nuova

Tuttavia, le prestazioni non sono sempre buone, poiché potrebbe **rimuovere una variabile molto utilizzata** pertanto verrebbe riportata molto spesso.

reference string	7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1	page frames
	7 7 7 2 2 2 2 4 4 4 0 0 0 1 7 7 7 2 2 2	
	7 0 0 1 1 1 0 0 0 2 2 0 0 1 2 2 0 0 1 1 1	
	7 0 0 1 1 1 0 0 0 2 2 0 0 1 2 2 0 0 1 1 1	
	7 0 0 1 1 1 0 0 0 2 2 0 0 1 2 2 0 0 1 1 1	

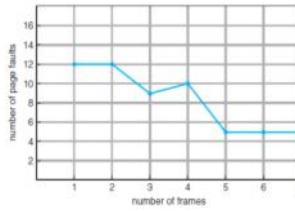
Per la nostra reference string, i nostri tre frame sono inizialmente vuoti. I primi tre riferimenti (7, 0, 1) causano page fault e vengono portati in questi frame vuoti. Il riferimento successivo (2) sostituisce la pagina 7: è stata inserita per prima quindi è la più vecchia. Poiché 0 è il riferimento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

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

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

La Figura mostra la curva dei page fault per questa stringa di riferimento rispetto al numero di frame disponibili.

Si noti che il numero di errori per quattro frame (dieci) è maggiore del numero di errori per tre frame (nove)!



### MIN o OPT (Optimal Page Replacement)

Rimuove la page che **non verrà acceduta** per molto tempo.

È l'algoritmo migliore ma richiede di predire la futura stringa di riferimento, diventando molto difficile da implementare.

reference string	7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1	page frames
	7 7 7 2 2 2 2 4 4 4 0 0 0 1 7 7 7 2 2 2	
	7 0 0 1 1 1 0 0 0 2 2 0 0 1 2 2 0 0 1 1 1	
	7 0 0 1 1 1 0 0 0 2 2 0 0 1 2 2 0 0 1 1 1	
	7 0 0 1 1 1 0 0 0 2 2 0 0 1 2 2 0 0 1 1 1	

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

### LRU (Least Recently Used)

Rimuove la page che **non è stata usata** per il **periodo di tempo più lungo**.

È simile all'OPT ma guarda indietro nel tempo invece che in avanti.

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

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

Esistono 3 modi per implementarlo:

#### Counters

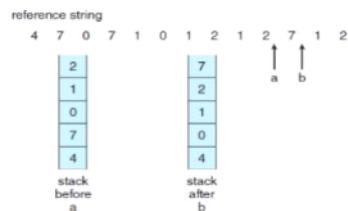
Ogni entry della page table ha un **campo** che indica **quanto tempo è stato usato** e viene aggiunto un **clock** alla CPU che aumenta ad ogni **riferimento di mem.**. Ad ogni **riferimento** ad una **pagina**, il val. nel clock viene **copiato** nel campo della page table, in modo tale di avere sempre il "tempo" dell'ultimo riferimento ad ogni pagina.

#### Stack

Viene mantenuto uno **stack di page number**. Ogni volta che si fa riferimento ad una pagina, questa viene **rimossa** **dallo stack** e messa in cima.

In questo modo la page usata **più recentemente** è sempre **in cima** dello stack e la page usata **meno di recente** è sempre **in fondo**.

#### Reference Bit



Viene utilizzato un **bit di riferimento**, settato dall'HW ogni volta che si fa riferimento a quella pagina

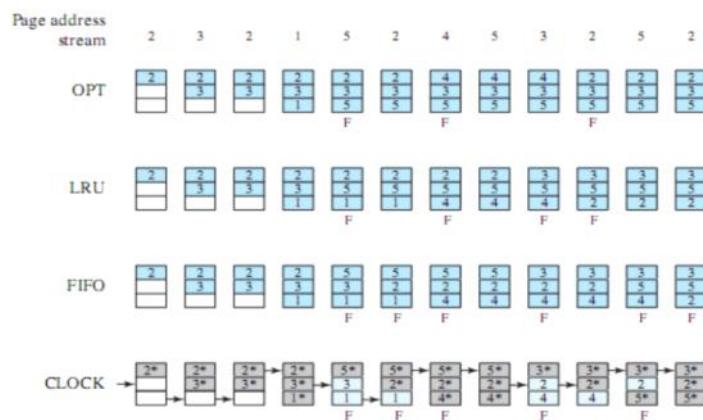
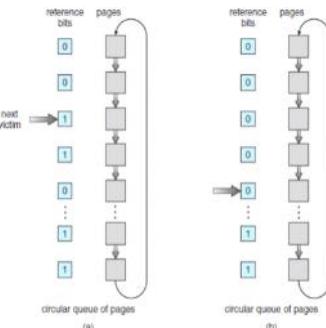
Può essere implementato in 3 modi:

- Tramite un **singolo bit di riferimento**: viene usato un **solo bit per page**, impostato inizialmente a 0 e **impostato a 1** se viene fatto un **accesso** alla page associata.  
Durante la rimozione, si scarta una delle pagine impostate a 0.
- Tramite **più bit di riferimento**: vengono usati **8 bit per page**, i quali vengono **shiftati** (spostati a destra) di un bit ad ogni **intervallo regolare** (es. 100ms), scartando il bit di ordine inferiore. Dopo ogni accesso alla page associata viene impostato ad 1 il bit **più a sinistra**.  
Durante la rimozione, si scarta la page col val. **più basso** nei bit. Si possono scartare tutte le page col val. più piccolo o usare il **FIFO** per scegliere.
- Tramite l'**algoritmo di seconda chance o orologio**: viene usato **un solo bit di riferimento** e una **politica FIFO con coda circolare**: l'OS tiene traccia dei frame nella lista FIFO circolare e ad ogni accesso alla mem. si imposta il bit di riferimento ad 1.  
Durante un **page fault**, l'OS controlla i bit di ogni page:
  - se il bit è 0, la page viene rimpiazzata
  - se il bit è 1, il bit viene reimpostato su 0 (**seconda chance**) e viene ripetuto il controllo sul frame successivo

Una **versione più avanzata** usa un'ulteriore **bit di modifica** (1 se la page è diversa da quella su disco, altrimenti no), classificando la page in quattro categorie:

- a. bit (1,1): **usata recentemente ed è stata modificata**, probabilmente verrà riutilizzata preso e la page dovrà essere scritta nella mem. secondaria prima di poter essere sostituita
- b. bit (1,0): **usata recentemente ma non è stata modificata**, probabilmente verrà riutilizzata preso
- c. bit (0,1): **non usata recentemente ma è stata modificata**, buona da sostituire, ma dovrà essere scritta prima di essere sostituita
- d. bit (0,0): **né usata né modificata di recente**, migliore da sostituire

Durante la rimozione si scarta la prima pagina con categoria inferiore (prima (0,0), poi (0,1), ...)

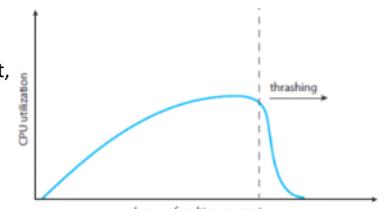


F = page fault occurring after the frame allocation is initially filled

## Trashing

**Trashing** è il fenomeno in cui un processo non dispone di **abbastanza frame** per il numero di page nel working set, portando le pagine ad essere **continuamente rimpiazzate** anche se verranno usate dopo poco tempo.  
dunque una forte degradazione delle performance.

Un processo è in **thrashing** se impiega più tempo a eseguire il paging che l'esecuzione.



## Politiche di allocazione e rimpiazzo delle pagine

Per contrastare il thrashing bisogna cedere abbastanza mem. ad ogni processo, rendendo fondamentale le **politiche di allocazione e rimpiazzo delle page**:

- **Allocazione e rimpiazzo globale**: tutte le page di ogni processo sono in una **singola pool** (queue LRU), durante il rimpiazzo anche le page non appartenenti allo stesso processo della nuova page possono essere rimpiazzate, portando ad una **maggior flessibilità** ma anche ad una **maggior quantità di thrashing**
- **Allocazione e rimpiazzo locale**: ogni processo ha una **propria pool** di frame e il rimpiazzo LRU controlla solo le page appartenenti allo stesso processo della nuova page, portando ad un **isolamento dei processi** ma anche ad una **minore performance**
- **Allocazione e rimpiazzo proporzionale**: viene assunto che ogni **processo di grandi dimensioni** fa riferimento ad una **grande quantità di mem..** Tuttavia in alcuni casi questa assunzione è errata (es. un processo può allocare un array di 1GB, per poi usarne solo una piccola porzione).  
In altre parole, non sempre il **working set** di un processo è correlato al suo impatto sulla mem.

## Working Set

Il **working set** di un processo è l'insieme di page che il processo usa attualmente. Formalmente, corrisponde all'insieme di **tutte le page** a cui ha effettuato riferimenti nelle **ultime T unità di tempo**.

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

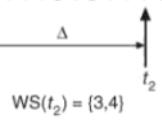
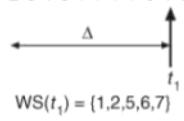
Per evitare di dover gestire una lista delle ultime  $\Delta$  richieste, il **working set** viene spesso implementato tramite una semplice **campionatura**:

ogni **k riferimenti** in mem., il working set di un processo viene considerato l'**insieme di tutte le page** a cui è stato fatto riferimento in **tal periodo temporale**.

L'obiettivo è quello di riuscire a dare ad ogni processo **abbastanza frame** da poter contenere il proprio working set

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



# Dispositivi I/O

lunedì 2 dicembre 2024 12:10

Ci sono 3 categorie di dispositivi:

- **Leggibili dall'utente**: comunicazione diretta con l'utente (es. stampanti, monitor, mouse, ecc.)
- **Leggibili dall'OS**: comunicazione con materiale elettronico (es. sensori, dischi, chiavi USB, ecc.)
- **Dispositivi di comunicazione**: comunicazione con dispositivi remoti (es. modem, Wi-Fi, ecc.)

Dispositivi di Input	Dispositivi di Output
<p>Un disp. di <b>Input</b> deve essere <b>interrogato</b> su certi val. fisici interni (es. tastiera: codice Unicode dei tasti premuti)</p> <p>Un processo che effettua una <b>syscall read</b> su un disp. di Input vuole conoscere questo dato</p>	<p>Un disp. di <b>Output</b> prevede di <b>poter cambiare</b> certi val. fisici interni (es. stampante: PDF di un file da stampare)</p> <p>Un processo che effettua una <b>syscall write</b> su un disp. di Output vuole cambiare qualcosa</p>

Quindi per modificare i disp. I/O si usano le **syscall read/write** e gli argomenti sono l'ID del disp.

Dopo la syscall, il **kernel**, inizializza il trasferimento di informazioni:

- Mette il processo chiamante in **blocked** e passa ad altro
- Usa i **driver** (insieme di procedure per gestire il disp.) del disp. per gestirlo
- A trasferimento completato, termina l'op. e il processo torna in **ready**

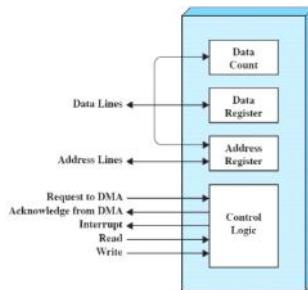
## Tecniche per effettuare I/O

- Programmato
- Guidato dagli interrupt
- Accesso diretto in mem. (**DMA**)

	Senza interruzioni	Con interruzioni
<b>Passando per CPU</b>	I/O programmato	I/O guidato da interruzioni
<b>Direttamente in mem.</b>		DMA

### DMA

- Il proc. delega le op. I/O al modulo **DMA**
- Il modulo DMA **trasferisce i dati** da/verso la **RAM** (mem. principale)
- Completata l'op., il modulo genera un **interrupt** per la **CPU**



# Efficienza

lunedì 2 dicembre 2024 12:54

La maggior parte dei disp. I/O sono molto lenti (in particolare il disco HDD) rispetto alla RAM, e se l'I/O non tiene il passo con la CPU il numero di processi che attendono il completamento di un'op. I/O aumenta.

La soluzione a livello OS è di progettare le funzioni di I/O in modo **modulare, gerarchico e uniforme** e offrendo funzionalità come: read, write, lock, unlock, open, close.

Si implementa quindi una **progettazione gerarchica** usando un **insieme di livelli**:

- Ogni liv. si basa sul fatto che il liv. **sottostante** effettua op. **più primitive**
- Ogni liv. **fornisce servizi** al liv. **superiore**
- Ogni liv. contiene funzionalità simili per complessità, tempi di esec., livello di astrazione
- Teoricamente modificare un liv. **non ha effetti** sugli altri liv.

Per l'I/O ci sono 3 **macrotipi maggiormente usati**:

## Dispositivo Locale

Es. stampante, monitor, tastiera, ecc.

- **Logical I/O**: il disp. è visto come una **risorsa logica** (open, close, read,...)
- **Device I/O**: trasforma richieste logiche in **sequenze di comandi I/O**
- **Scheduling & Control**: esegue e controlla le sequenze di comandi, eventualmente gestendo l'accomodamento



## Dispositivo di Comunicazione

Es. scheda Ethernet, WiFi, ecc.

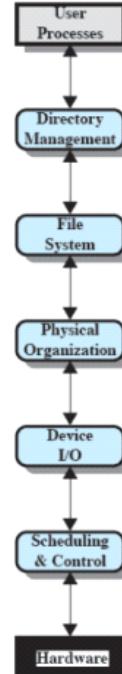
- Come prima ma al posto di Logical I/O c'è una architettura di comunicazione
- **Architettura di comunicazione**: il disp. è visto come una **risorsa logica**
- A sua volta, questa consiste in un certo num. di livelli (es. TCP/IP)



## File System

Es. disco, SSD, CD, chiave USB, ecc.

- **Directory Management**:
  - da nomi di file a ID di file
  - tutte le op. utente che hanno a che fare con i file (creareli, cancellarli, ...)
- **File System**: struttura **logica** e op. (apri, chiudi, leggi, scrivi, ...)
- **Organizzazione fisica**:
  - da ID di file a indir. fisici su disco
  - allocazione/deallocazione



# Buffering

lunedì 2 dicembre 2024 13:49

Il **Buffering I/O** è una tecnica usata per **migliorare l'efficienza** delle op. I/O.

Sfrutta l'uso di un **buffer** (area riservata in mem.) per archiviare temporaneamente i dati per **ridurre il num. di op. I/O** e **gestire il flusso di dati** tra disp. o processi veloci e lenti.

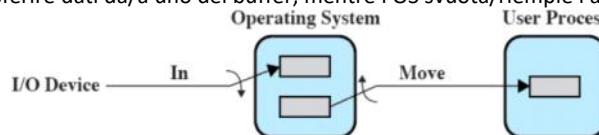


Esistono due tipi di Buffer singoli:

<b>Orientato a Blocchi</b> <ul style="list-style-type: none"><li>I trasferimenti di input sono fatti al buffer in <b>sys. mem.</b></li><li>Il <b>blocco</b> viene mandato nello spazio utente dal processo, che richiede un altro blocco</li><li>Quando un blocco viene elaborato dal processo utente, il blocco successivo viene letto</li><li>L'output viene posticipato (serve la syscall flush)</li></ul>	<b>Orientato agli Stream</b> <ul style="list-style-type: none"><li>I terminali hanno a che fare con linee (l'utente inserisce una riga alla volta, con un ritorno a capo alla fine di una riga)</li><li>Si bufferizza una linea intera di Input o di Output</li><li>Si sa un <b>byte alla volta</b> per dispositivi in cui un singolo carattere premuto va gestito</li></ul>
---	--

## Buffer doppio

Si usano **due buffer**: un processo può trasferire dati da/a uno dei buffer, mentre l'OS svuota/riempie l'altro



Lettura e scrittura nel buffer sono **parallele**: mentre un buffer viene letto, l'altro viene scritto

## Buffer circolare

- Si usano **più di 2 buffer**
- Ogni buffer è un'unità nel buffer circolare
- Usato quando l'op I/O deve tenere il passo del processo
- Soluzione al problema **produttore/consumatore** (vedi in fondo)

## Pro e Contro del Buffer

- Il buffering **smussa i picchi di richieste I/O**
  - ma se la domanda è molta, i buffer si riempiono e il vantaggio è perso
- Utile quando ci sono molti dispositivi I/O da servire

## Problema Produttore/Consumatore

Il problema del **produttore/consumatore** (o problema del buffer limitato) è un problema di **sincronizzazione** in informatica.

Si verifica quando più processi devono **condividere risorse limitate**, come un buffer, e collaborare per evitare conflitti.

### Scenario

- Un **produttore** genera dati e li **inserisce** in un **buffer**
- Un **consumatore** **preleva** i dati dal **buffer** per elaborarli
- Il **buffer** ha **capacità limitata** e può contenere solo un num. max di elementi

### Problemi principali

- Sovraccarico del buffer**: se il produttore **genera** dati **più velocemente di quanto** il consumatore li **prelevi**, il buffer può **riempirsi**, e il produttore deve attendere
- Buffer vuoto**: Se il consumatore **consuma** dati **più velocemente di quanto** il produttore li **generi**, il buffer può **svuotarsi**, e il consumatore deve attendere
- Condizioni di competizione**: se il produttore e il consumatore accedono **contemporaneamente** al buffer **senza sincronizzazione**, possono verificarsi errori (es. lettura/scrittura di dati corrotti)

La **soluzione** al problema si basa su **tecniche di sincronizzazione** come:

- Semafori**
- Monitor**

- **Buffer circolari**

# HDD

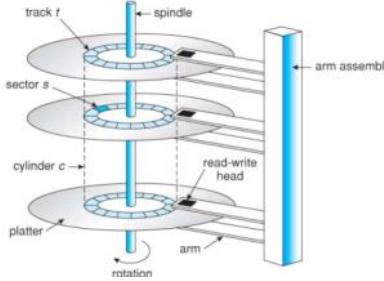
martedì 26 novembre 2024 17:19

Un **HDD** (Hard Disk Drive o Disco rigido) è una **mem. secondaria** composta da un perno sul quale poggiano uno o più **piatti** coperti da materiale magnetico, dove:

- Ogni **piatto** ha **2 superfici** funzionali, **suddivise** in un num. di anelli concentrici, detti **tracce**.
- Ogni **traccia** è **divisa in settori**.
- Per **selezionare** una traccia bisogna:
  - **spostare** una **testina**, se il disco ha **testine mobili**
  - **selezionare** una **testina**, se il disco ha **testine fisse**
- Ogni **superficie** viene letta da una **testina**, mentre il **perno**, ruotando, **seleziona il blocco** di mem. che la **testina leggerà**

La **capacità totale di mem.** è:

$$C = H \cdot T \cdot B = \text{num. di superfici} * \text{tracce per superficie} * \text{byte per traccia}$$



La CPU si interfaccia con i dischi tramite un **bus I/O** e invia comandi tramite un **host controller**, che a sua volta comunica col **disk controller** interno al disco stesso, trasferendo dati dalle superfici ad una cache interna al disco e poi alla mem.

Il **tempo di trasferimento dei dati** alla RAM, indicato come **DTT**, equivale alla somma tra:

- **Tempo di accesso**, che è la **somma di**:
  - **Seek time (ST)**: tempo necessario per spostare le testine su una specifica traccia
  - **Rotation delay (ROT)**: tempo necessario per girare il disco affinché il **settore** venga ruotato **sotto la testina** (in media **mezza rotazione**)
- **Transfer time (TT)**: tempo necessario per **spostare** i dati dal disco alla mem. espresso in **rateo di trasferimento** (byte al secondo)

$$DTT = ST + ROT + TT$$

## Esempi

Supponendo un tempo di trasferimento totale pari a 40ms per effettuare una particolare operazione I/O, dove il seek time è 18ms, il rotational delay è 7ms e che il trasfer rate è 5 Gbit/s, la quantità totale di dati trasferita è:

$$\begin{aligned} 40ms &= 18ms + 7ms + \frac{x}{5\text{Gbit/s}} \iff 15ms = \frac{x}{5\text{Gbit/s}} \iff \\ &\iff x = 15ms \cdot 5\text{Gbit/s} \iff x = 15 \cdot 10^{-3}s \cdot 5 \cdot 10^9\text{bit/s} \iff \\ &\iff x = 75 \cdot 10^6\text{bit} \iff x = 9.375 \cdot 10^6B \iff x = 9.375MB \end{aligned}$$

- Supponendo un tempo di trasferimento totale pari a 36ms per effettuare una particolare operazione I/O, dove il seek time è 13ms, il trasfer rate è 1 Gbit/s e sono stati trasferiti 2MB, il rotational delay è:

$$\begin{aligned} 36ms &= 13ms + x + \frac{2MB}{1\text{Gbit/s}} \iff 23ms = x + \frac{2 \cdot 10^6 \cdot 8\text{bit}}{1 \cdot 10^9\text{bit/s}} \iff \\ &\iff 23ms = x + \frac{16}{10^3}\text{s} \iff 23ms = x + 16ms \iff x = 7ms \end{aligned}$$

## Formattazione

Prima che un disco possa essere usato bisogna **formattarlo fisicamente**. Cioè stabilire l'**inizio (header)** e la **fine (trailer)** di ogni settore, dove verranno inseriti i **codici di correzione degli errori (ECC)**.

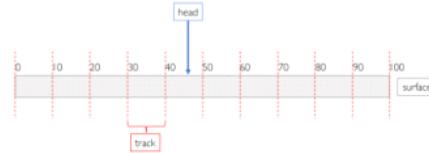
Dopo la formattazione, il disco viene **partizionato** in una o più **partizioni**.

# Scheduling Disco

venerdì 6 dicembre 2024 15:19

È necessario gestire in modo efficiente gli **accessi al disco**, cercando di ridurre al minimo il **seek time**, dunque bisogna ridurre la **distanza percorsa** dalla testina.

Dato il seguente disco



Supponiamo che arrivino delle richieste in ordine delle seguenti tracce: 65, 40, 18, 78.

## First Come First Serve (FCFS)

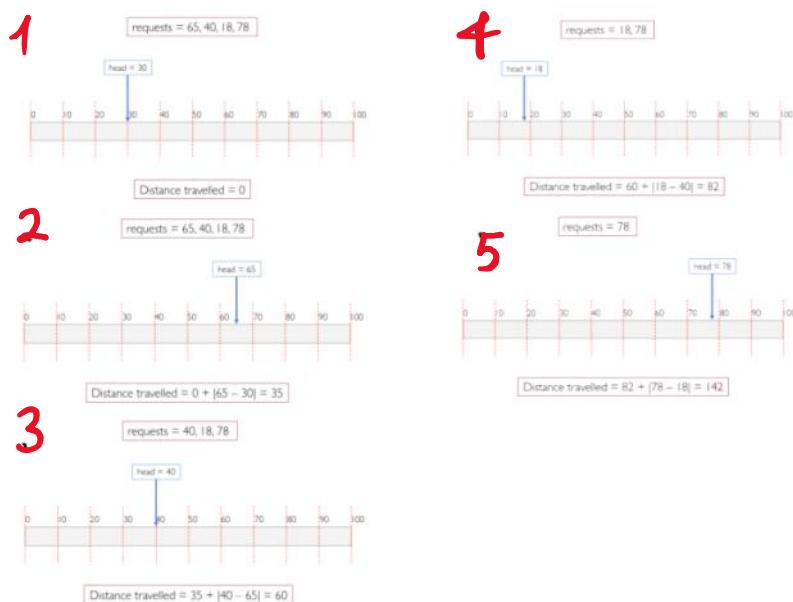
Scheduling in cui le richieste di **accesso al disco** vengono effettuare nel loro **ordine di arrivo**

**Pro:**

- **Facile** da implementare
- Usato anche dagli SSD, poiché lì il seek time è nullo

**Contro:**

- Poco **performante** se arrivano **molte richieste**



## Shortest Service Time First (SSTF)

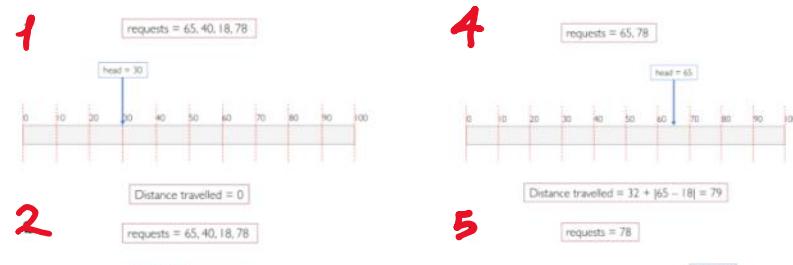
Scheduling in cui viene eseguita la richiesta di accesso al disco richiedente la **minore quantità di tracce da traversare**

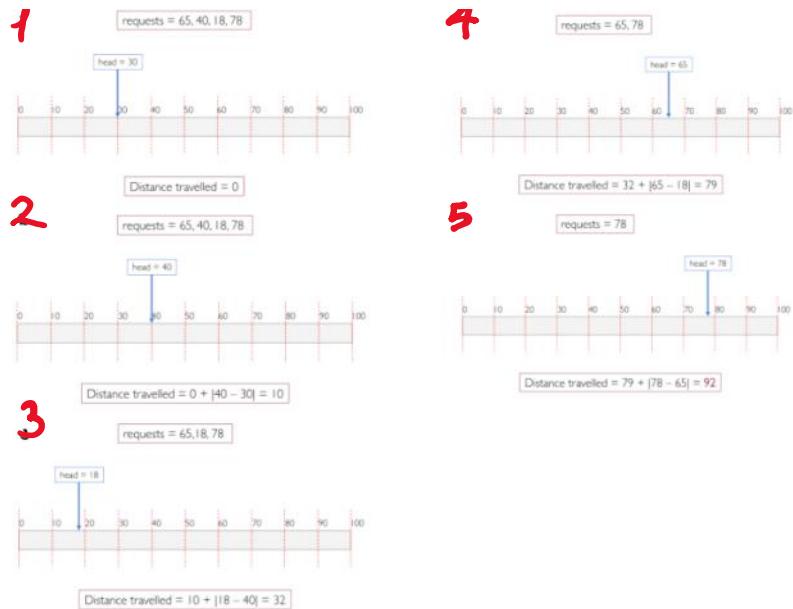
**Pro:**

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

**Contro:**

- Può causare **starvation** (se una richiesta necessita una traccia troppo distante dalla testina)
- L'ottimizzazione avviene a livello locale (es, l'esec. della richiesta più vicina alla testina può portare ad una catena di spostamenti peggiore rispetto al caso in cui si vada a selezionare un'altra richiesta)

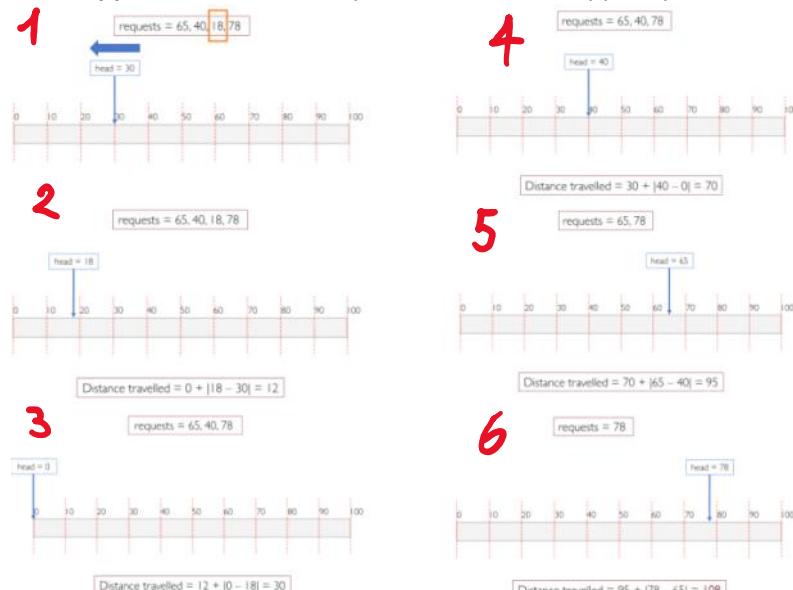




## SCAN e LOOK

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

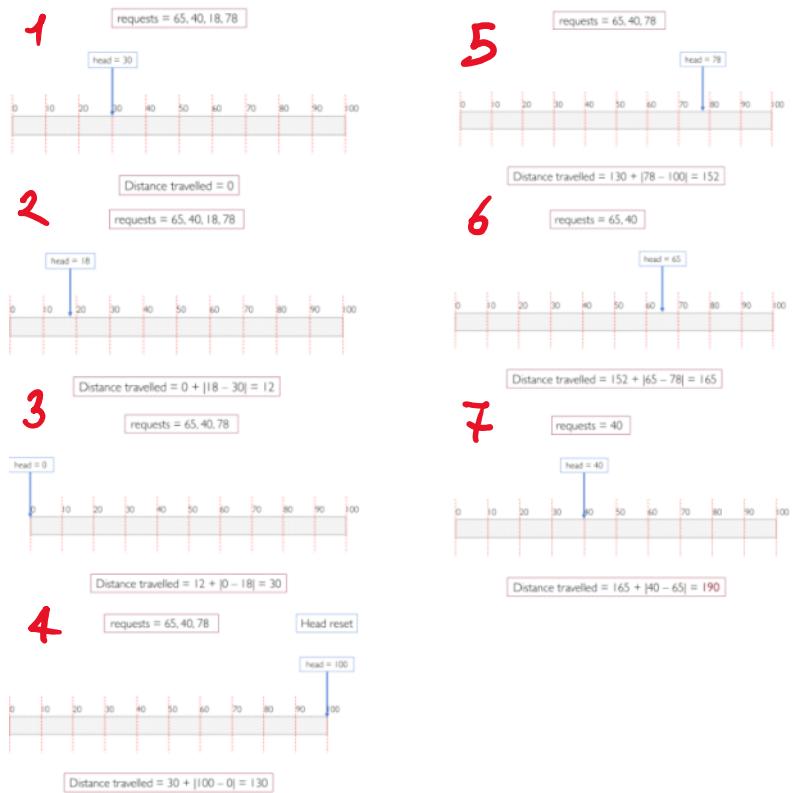
- Niente **starvation**, ma poco "fair":
  - favorisce le richieste **ai bordi** (C-SCAN è stato sviluppato per risolvere questo problema)
  - favorisce le richieste **appena arrivate** (N-step-SCAN è stato sviluppato per risolvere questo problema)



L'algoritmo **LOOK** è la **versione ottimizzata** di SCAN: durante ogni scansione la testina, piuttosto che proseguire fino all'estremo del piatto, si ferma alla richiesta **più vicina all'estremo**.

## C-SCAN e C-LOOK

L'algoritmo **C-SCAN (Circular SCAN)** è uno scheduling simile a SCAN in cui la testina viene mossa **dall'estremo finale a quello iniziale** (es. da traccia 100 a 0), venendo subito dopo "resettata" all'estremo finale (spostata nuovamente a 100). Le richieste vengono effettuate **solo durante** lo spostamento dall'estremo finale a quello iniziale.



L'algoritmo **C-LOOK (Circular LOOK)** è una versione **ottimizzata** del C-SCAN, dove durante ogni scansione la testina, invece che andare all'estremo iniziale, viene fermata alla **richiesta più vicina all'estremo iniziale**, per poi essere resettata alla **richiesta più vicina all'estremo finale**, piuttosto che all'estremo stesso

## FSCAN

Vengono utilizzate **due code** (F e R) dove vengono inserite le richieste scambiandosi durante lo SCAN. Quando SCAN **serve** le richieste in **F**, le **nuove richieste** sono aggiunte ad **R**, e viceversa, scambiandosi quando completano le richieste nella loro coda.

## N-step-SCAN

Simile a FSCAN ma vengono usate **N > 2 code**.

Si accodano le richieste nella coda i-esima finché non si riempie; poi si passa alla i+1.

Se N è **alto**, le **prestazioni** sono quelle di **SCAN**. Se **N=1** si usa **FIFO**.

## Vantaggi degli Scheduling

### Confronto Prestazionale

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed	Next track accessed	Number of tracks traversed
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
<b>Average seek length</b>	55.3	<b>Average seek length</b>	27.5	<b>Average seek length</b>	27.8	<b>Average seek length</b>	35.8

Algoritmo	Descrizione	Note
<b>Selezione a cura del richiedente</b>		
RSS	Scheduling random	Solo per simulazioni e confronti
FIFO	A coda semplice	Il più equo
LIFO	Ultimo utente	A sorpresa, può andare bene
PRI	Priorità del processo	Se la priorità è importante
<b>Selezione sulla base del dato richiesto</b>		
SSTF	Tempo di servizio più corto	Alto uso del disco, piccole code
SCAN	Avanti ed indietro sul disco	Miglior distribuzione del servizio
C-SCAN	Avanti, con ritorno veloce	Minore variabilità di servizio
N-step-SCAN	SCAN su $N$ richieste	Garanzia del servizio
FSCAN	N-step-SCAN con due code	Tiene conto del carico

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

$$35 + 25 + 22 + 60 = 142$$

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

$$10 + 22 + 47 + 13 = 92$$

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:

$$12 + 78 = 90$$

**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
$$12 + 60 + 38 = 110$$

# SSD e RAID

venerdì 29 novembre 2024 17:03

Gli **SSD** (Solid State Drive) sono una **mem. secondaria** realizzata tramite memoria flash o chip DRAM protetti da batteria che permette alle mem. volatili di mantenere l'informazione

## Pro:

- Non hanno meccanismi moventi, risultando estremamente più veloci
- L'accesso ai blocchi è effettuato in modo diretto tramite riferimento al num. di blocco, rimuovendo la necessità di uno scheduling
- Le operazioni di lettura sono molto più veloci

## Contro:

- Le operazioni di scrittura sono molto più lente, perché i blocchi non vengono realmente sovrascritti ma de-referenziati (resi inaccessibili), rendendo più lento il ciclo di eliminazione dei dati
- Il num. di "scritture" per un blocco è limitato

## RAID

RAID (Redundant Array of Inexpensive Disks) è un modello che consiste nel mettere insieme molti dischi fisici insieme a delle forme di **duplicazione dei dati**, detto **mirroring**, rendendoli **ridondanti** nei vari dischi, per aumentare l'**affidabilità** e la **velocità** delle operazioni, poiché è richiesta minor tempo per ricercare i dati rispetto all'uso di pochi dischi di grandi dimensioni.

Il **mirroring** è una tecnica che consiste nel **copiare gli stessi dati** su più dischi. I dati non andranno persi finché tutte le copie non sono danneggiate e diminuisce la probabilità che un singolo disco perda i dati. Aumenta anche le **performance di lettura**, facendo sì che sia possibile leggere blocchi da più dischi simultaneamente.

Un'altra tecnica per aumentare la velocità è lo **striping**, ossia spargere i dati su più dischi per far sì che possano essere acceduti simultaneamente.

## Riassunto tecniche RAID

- **Mirroring:** copia i dati su più dischi per diminuire la probabilità di perdita dei dati e aumentare le performance di lettura
- **Striping:** sparge i dati su più dischi per aumentare la velocità di lettura
- **Parallel access:** se eseguo un'op sul RAID, tutti i dischi eseguono in sincrono quell'op.
- **Independent:** un'op. sul RAID è un op. su un sottoinsieme dei suoi dischi
- **Data availability:** capacità di recuperare dati in caso di fallimento
- **Small I/O request rate:** velocità nel rispondere a piccole richieste I/O

Il mirroring aumenta l'affidabilità ma è costoso e introduce dello spreco di mem., lo striping aumenta le performance ma non l'affidabilità.

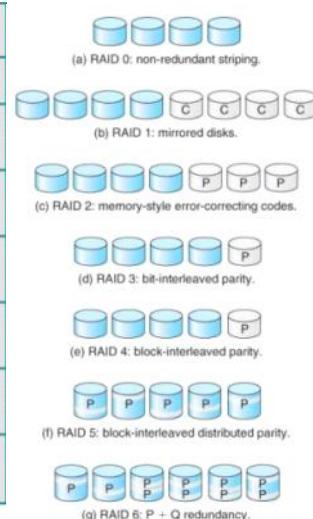
Differenti schemi possono essere costruiti sul RAID combinando diverse ottimizzazioni per bilanciare l'affidabilità e performance.

Sono noti differenti livelli di RAID:

- RAID 0: striping
- RAID 1: mirroring (serve il doppio dei dischi per la copia)
- RAID 2 (non usato): **Hamming** (corregge errori su singoli bit)
- RAID 3 (non usato): Memorizza la **parità dei bit** degli strip
- RAID 4 (non usato): come RAID 3 ma ogni strip è un "blocco"
- RAID 5: come RAID 4 ma la parità salvata è suddivisa tra i dischi
- RAID 6: come RAID 5 ma con 2 dischi di parità indipendenti

Category	Level	Description	Disks Required	Data Availability	Large I/O Data Transfer Capacity	Small I/O Request Rate
Striping	0	Nonredundant	$N$	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

Note:  $N$ , number of data disks;  $m$ , proportional to  $\log N$ .



# File

lunedì 2 dicembre 2024 17:06

I **file** sono l'elem. principale per la maggior parte delle applicazioni, poiché spesso l'input/output di un app è un file ed essi **rimangono** anche quando i processi terminano.

Il **file system** è un insieme di **programmi** eseguiti in **kernel mode** (processi privilegiati) e **librerie** invocate con **syscall**, usati per gestire i file della memoria secondaria e per ogni file vengono mantenuti degli **attributi (metadati)**, come proprietario, data di creazione, etc.

Le operazioni tipiche sui file sono: creazione, cancellazione, apertura, scrittura, lettura, chiusura

## Terminologia

<b>Campi</b> <ul style="list-style-type: none"><li>Dati di base</li><li>Contengono <b>val. singoli</b></li><li>caratterizzati da lunghezza e tipo di dato</li><li>Es: char. ASCII</li></ul>	<b>Record</b> <ul style="list-style-type: none"><li>Insieme di <b>campi correlati</b></li><li>Ognuno trattato come un'<b>unità</b></li><li>Es: un impiegato è caratterizzato dal record <b>nome, cognome, ecc.</b></li></ul>
<b>File</b> <ul style="list-style-type: none"><li>Hanno un nome</li><li>Insieme di <b>record correlati</b></li><li>Ognuno trattato come un'unità con nome proprio</li><li>Possono implementare meccanismi di controllo all'accesso</li></ul>	<b>Database</b> <ul style="list-style-type: none"><li>Collezione di <b>dati correlati</b></li><li>Mantengono anche <b>relazioni</b> tra gli elem. memorizzati</li><li>Realizzati con <b>uno o più file</b></li><li>Ci sono i <b>DBMS</b>, che sono tipicamente processi di un OS</li></ul>

## Obiettivi del File System

- Rispondere alle necessità degli utenti riguardo la **gestione dei dati**
- Ottimizzare le prestazioni**, sia per l'OS (throughput) che per l'utente (tempo di risposta)
- Minimizzare i dati persi o distrutti
- Supportare diversi I/O e permettere l'accesso di più utenti in contemporanea
- Fornire interfacce standard per i processi utente

## Requisiti del File System

Ogni **utente** deve poter:

- 1) **creare, cancellare, leggere, scrivere e modificare** un file
- 2) **accedere**, in modo controllato, ai file di un altro utente
- 3) **leggere e modificare i permessi** di accesso ai propri file
- 4) **ristrutturare** i propri file
- 5) **muovere dati** da un file ad un altro
- 6) **mantenere una copia** di backup dei propri file
- 7) **accedere** ai propri file tramite **nomi simbolici**



- **Directory Management:**
  - da nomi di file a ID di file
  - si occupa delle op. utente che hanno a che fare con i file (crearli, cancellarli, ...)
- **File System:** struttura logica e op. (apri, chiudi, leggi, ...)
- **Organizzazione fisica:**
  - da ID di file a indir. fisico su disco
  - allocazione/deallocazione
- **Scheduling & Control:** vari scheduling e controlli del disco

# Directory

lunedì 2 dicembre 2024 18:06

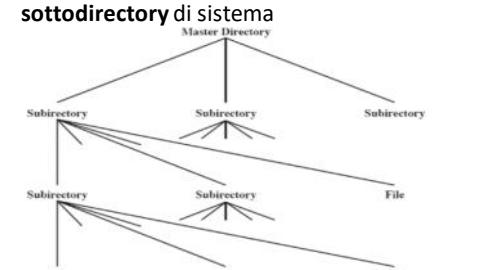
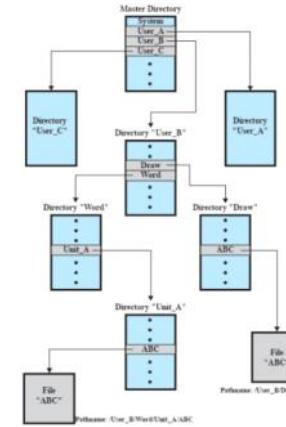
<p>Le <b>directory</b> contengono:</p> <ul style="list-style-type: none"><li>• <b>Informazioni</b> sui file<ul style="list-style-type: none"><li>◦ attributi</li><li>◦ posizione (dove si trovano)</li><li>◦ proprietario</li></ul></li><li>• Una directory è essa stessa un <b>file speciale</b></li><li>• Fornisce il <b>mapping</b> tra nomi dei file e file stessi</li></ul>	<p>Le <b>operazioni</b> sulla directory sono:</p> <ul style="list-style-type: none"><li>• Ricerca</li><li>• Creazione File</li><li>• Cancellazione File</li><li>• Lista dei contenuti della dir.</li><li>• Modifica della dir.</li></ul>
--	--

Gli **elementi** della directory sono:

Info di Base	Info sull'Indirizzo	Controllo di Accesso	Info sull'Uso
<ul style="list-style-type: none"><li>• <b>Nome del file:</b> scelto dal creatore e unico in una directory data</li><li>• <b>Tipo file:</b> eseguibile, testo, binario, ...</li><li>• <b>Organizzazione del file:</b> per OS che supportano diverse possibili organizzazioni</li></ul>	<ul style="list-style-type: none"><li>• <b>Volume:</b> indica il dispositivo su cui il file è memorizzato</li><li>• <b>Indir. di partenza</b> (es. quale settore o traccia di disco)</li><li>• <b>Dimensione attuale</b> (in byte, word o blocchi)</li><li>• <b>Dimensione allocata:</b> dimensione massima del file</li></ul>	<ul style="list-style-type: none"><li>• <b>Proprietario:</b> può <b>modificare o concedere/negare permessi</b> ad altri utenti</li><li>• <b>Info sull'accesso:</b> può contenere <b>username e password</b> di ogni utente autorizzato</li><li>• <b>Azioni permesse:</b> per controllare lettura, scrittura, esecuzione, spedizione tramite rete</li></ul>	<ul style="list-style-type: none"><li>• Data creazione</li><li>• Data dell'ultimo accesso in scrittura/lettura</li><li>• Data dell'ultimo backup</li><li>• Identità del creatore</li><li>• Identità dell'ultimo lettore/scrittore</li><li>• Uso attuale (lock, azione corrente, ...)</li></ul>

## Strutture

Ci sono diversi modi per memorizzare le informazioni di una directory

Struttura semplice	Schema a 2 livelli	Schema Gerarchico ad Albero
<p>Il metodo più semplice è fare per ogni file una <b>lista di entry</b>:</p> <ul style="list-style-type: none"><li>• File <b>sequenziale</b> con il nome del file a fare da <b>chiave</b></li><li>• Non aiuta ad organizzare i file</li><li>• Non si può dare lo stesso nome a due file diversi</li></ul>	<p>Una dir. per ogni utente e una (<b>master</b>) che le contiene</p> <ul style="list-style-type: none"><li>• La <b>master</b> contiene anche l'indir. e le info per il controllo dell'accesso</li></ul> <p>Ogni dir. utente è solo una <b>lista dei file</b> di quell'utente</p> <ul style="list-style-type: none"><li>• Non offre struttura per insiemi di files</li></ul>	<p>• Una <b>dir. master</b> che contiene le dir. utente</p> <ul style="list-style-type: none"><li>• Ogni <b>dir. utente</b> può contenere file o altre dir. utente</li><li>• Dentro la dir. master ci sono anche <b>sottodirectory</b> di sistema</li></ul>  

## Nomi

Gli utenti possono riferirsi ad un file usando il suo **nome**.

La struttura ad **albero** permette agli utenti di trovare un file seguendo un percorso nell'albero (**directory path**).

I nomi duplicati sono possibili purché siano in dir. diverse

# Gestione Memoria Secondaria

lunedì 2 dicembre 2024 18:38

I file si allocano in "porzioni" o "blocchi"

- L'unità minima è il settore del disco
- Ogni porzione o blocco è una sequenza contigua di settori

Poiché l'OS è responsabile dell'assegnamento di **blocchi a file**, questo causa due problemi correlati:

- Occorre allocare **spazio per i file**, e mantenerne traccia una volta allocato
- Occorre tener traccia dello **spazio allocabile**

L'allocazione dello spazio per i file porta ad altri problemi:

- **Preallocazione vs Allocazione dinamica**
- Porzioni di dimensione **fissa o dinamica** e quanto grandi ("porzioni" per dim. dinamica, "blocco" per dim. fissa)
- **Metodo di allocazione**: contiguo, concatenato o indicizzato
- Gestione della **file allocation table** (per ogni file mantiene le info su dove sono sul disco e le porzioni che lo compongono)

## Preallocazione vs Allocazione Dinamica

La **preallocazione** richiede che la **dimensione max sia dichiarata** a tempo di creazione, cosa che a volte può essere complicato stimare creando uno spreco di spazio su disco.

L'**allocazione dinamica** è quasi sempre **preferita** poiché le **dimensioni** possono essere **aggiustate** in base all'append o alle truncate.

## Dimensione delle Porzioni

Esistono due possibilità per decidere le dimensioni:

- Si alloggia una **porzione larga** a sufficienza per **l'intero file**
  - Efficiente per il **processo** che vuole creare il file (l'accesso sequenziale è più veloce)
- Si alloggia un **blocco alla volta**
  - Efficiente per l'**OS**
  - Ciascun blocco è una sequenza di **n settori contigui** (n fisso e piccolo, spesso n=1)

Si deve cercare un **punto di incontro** (trade-off) tra **efficienza del singolo file** e **efficienza dell'OS**.

Per le prestazioni di accesso al file convengono **porzioni contigue** poiché **porzioni piccole** vuol dire grandi tabelle di allocazione (grande overhead) ma maggiore facilità di riuso dei blocchi e **porzioni fisse grandi** vuol dire una frammentazione interna (da evitare).

La **frammentazione esterna** è sempre possibile poiché i file possono essere cancellati.

Quindi esistono due possibilità:

- **Porzioni grandi e dim. variabile**
  - **Efficiente per i file**
  - Ogni allocazione è contigua
  - Tabella di allocazione abbastanza contenuta
  - Complicata la gestione dello spazio libero
- **Porzioni piccole e fisse**
  - **Efficiente per l'OS**
  - 1 blocco per porzione
  - Meno contiguo del precedente
  - Semplice la gestione dello spazio libero

## Allocazione Spazio per File

Esistono 3 metodi di allocazione:

- **Contigua**
- **Concatenata**
- **Indicizzata**

	Contiguous	Chained	Indexed
Preallocation?	Necessary	Possible	Possible
Fixed or variable size portions?	Variable	Fixed blocks	Fixed blocks
Portion size	Large	Small	Small
Allocation frequency	Once	Low to high	High
Time to allocate	Medium	Long	Short
File allocation table size	One entry	One entry	Large

### Allocazione Contigua

Blocchi allocati **tutti insieme** alla creazione del file

- Necessità **preallocazione** (sapere la lunghezza del file)
- Problema di **frammentazione esterna** con necessità di **compattazione**
- **Vantaggio**: gestione semplice con **una sola entry per file** nella tab. di allocazione (**blocco partenza e lunghezza file**)
- **Limite**: difficoltà nel far crescere i file oltre lo **spazio allocato** (potrebbe incontrare blocchi già occupati)

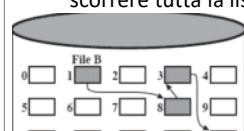


File Allocation Table		
File Name	Start Block	Length
File A	2	3
File B	9	5

### Allocazione Concatenata

Blocchi allocati **uno alla volta** e ogni blocco contiene il **puntatore** al successivo

- **Niente frammentazione esterna** ma necessita il **consolidamento** (simile alla compattazione)
- **Vantaggio**: buona per **accesso sequenziale** e la tab. di allocazione richiede **una sola entry (blocco partenza e lunghezza file)**
- **Limite**: lenta per **accesso diretto** (bisogna scorrere tutta la lista)



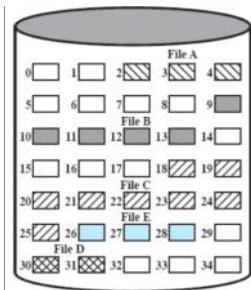
File Name	Start Block	Length
File A	1	5
File B	6	4

### Allocazione Indicizzata

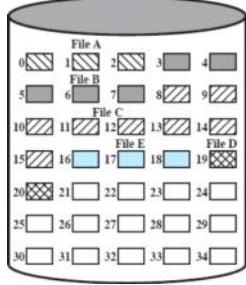
Ogni file ha un **blocco indice** che contiene gli **indirizzi** dei blocchi del file

- Risolve i problemi delle altre due tecniche
- Necessità **una sola entry** (blocco indice)
- Blocchi di **lunghezza fissa**:
  - **Niente frammentazione esterna**
  - Ogni entry del blocco indice punta ad un blocco del file
- Blocchi di **lunghezza variabile**:
  - (come allocazione contigua) Il blocco indice contiene **blocco partenza e lunghezza file**
  - Migliora la località

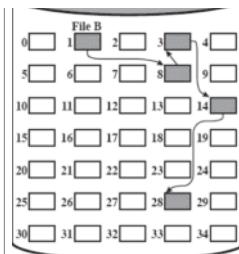
### Porzioni di Lunghezza Fissa



### Compattazione



File Allocation Table		
File Name	Start Block	Length
File A	2	3
File B	6	5
File C	18	3
File D	26	3
File E	-	-

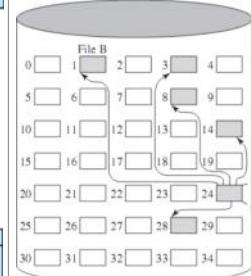


File Name	Start Block	Length
File B	1	5
File C	-	-
File D	-	-
File E	-	-

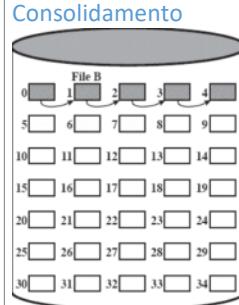
### partenza e lunghezza file

- Migliora la località

### Porzioni di Lunghezza Fissa



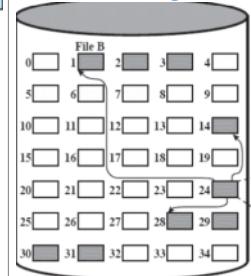
File Name	Index Block
File B	24
File C	-



### Consolidamento

File Name	Start Block	Length
File B	0	5
File C	-	-
File D	-	-
File E	-	-

### Porzioni di Lunghezza Variabile



File Name	Index Block
File B	24
File C	-

Start Block	Length
1	3
28	4
14	1

## Gestione dello Spazio Libero

Per poter allocare nuovo spazio per i file occorre sapere **dov'è lo spazio libero**. Quindi serve una **tavella di allocazione di disco** (oltre che a quella per i file) e ogni volta che si **alloca/cancella un file**, lo **spazio libero** va aggiornato.

Abbiamo 4 metodi usati per tenere traccia degli spazi liberi:

### Vettore di Bit

- Vettore con un **bit** per ogni **blocco** (0: libero, 1: occupato)
- Minimizza lo spazio richiesto** alla tabella di allocazione
- Se il disco è quasi pieno, la ricerca di uno spazio libero può richiedere molto tempo (risolvibile con tabelle riassuntive)

### Porzioni Libere Concatenate

- Similmente all'allocazione concatenata, si usa, per ogni blocco libero, **un puntatore ed un intero per la dimensione**
- Senza overhead** di spazio

Problemi:

- Se c'è frammentazione, le porzioni sono tutte da un blocco e la lista si allunga
- Bisogna leggere un blocco libero per sapere qual è il prossimo
- è lungo cancellare file molto frammentati

### Indicizzazione

- Similmente all'allocazione indicizzata, tratta lo **spazio libero** come un **file** e usa un **indice** per ogni blocco libero
- Per efficienza, l'indice gestisce le porzioni come se fossero di **lunghezza variabile** (c'è una entry per ogni porzione libera)

### Lista dei Blocchi Liberi

- Ad ogni blocco viene assegnato un **num. sequenziale** e la **lista** di questi numeri viene memorizzata in una parte dedicata del **disco**
- Se per ogni blocco servono 4 byte e i blocchi sono da 512 byte, richiede meno dell'1% dello spazio
- Per avere parti della lista in RAM, si può:
  - organizzare la lista come **pila**, e tenere solo la parte alta
  - pop per **allocare** spazio libero, push per **deallocare** spazio occupato
  - Quando la parte nella RAM finisce, si prende una nuova parte dal disco

# Informazioni Importanti del File System

venerdì 6 dicembre 2024 12:13

## Volumi

Il **Volume** è essenzialmente un **disco "logico"**

- È la **partizione di un disco**
- Oppure è **più dischi** messi insieme e visti come un disco solo **dall'LVM** (Gestore logico dei volumi)

Esso è un insieme di **settori** in mem. secondaria, che possono essere usati dall'OS o dalle applicazioni. I settori non devono necessariamente essere contigui, ma appariranno come tali all'OS o ai processi.

Un volume può essere anche un **unione di volumi** più piccoli.

## Dati e Metadati

**Dati:** Contenuto di un file | **Metadati:** informazioni di un file (es. lista blocchi liberi, lista blocchi in un file, data di ultima modifica)

I **metadati** devono rimanere su **disco** perché devono essere persistenti, però per efficienza, vengono anche tenuti in RAM

## Journaling

Il **journaling** è una tecnica usata per **preservare l'integrità** dei dati nel caso di imprevisti (es. reboot dopo un crash, mancanza di corrente, rimozione di un disco senza un appropriato comando).

Questa tecnica consiste nel scrivere le informazioni in una **zona di disco dedicata chiamata "log"** prima di farne il commit nel file system

Si usa un **bit di shutdown** all'inizio del disco, che dice se l'OS è stato spento correttamente.

Se il bit è 0 bisogna eseguire il ripristino del disco:

- 1) **Confronta il journal** allo stato corrente del file system
- 2) **Corregge inconsistenze** nel file system basandosi sulle op. salvate nel **log**
  - a. Se la scrittura del journal è **completa**, in caso di crash durante la scrittura nel file system si può **recuperare l'errore**
  - b. Se c'è un **crash** durante la scrittura nel journal, il file system rimane **integro**

Esistono due tipi di journaling:

### Journaling Fisico (tutti i dati)

- **Copia** nel journal **tutti i blocchi** che dovranno essere scritti nel file system (anche i metadati)
- Se c'è un crash durante la scrittura nel file system, basta copiare il contenuto dal journal al file system

### Journaling Logico (solo metadati)

- **Copia** nel journal **solo i metadati** delle op. effettuate (es. modifica blocchi liberi dopo cancellazione file)
- Se c'è un crash, si copiano i metadati dal journal al file system, ma questo può causare **corruzione dei dati**

## Journaling: Alternative

### Soft Updates File System

- **Riordina** scritture sul file system per **non avere mai inconsistenze** che portino a **perdita dei dati (storage leaks)**

### Log-Structured File System

- L'intero file system è strutturato come un **buffer circolare**, detto **log**
- Dati e metadati sono scritti in modo **sequenziale**, alla fine del log
- Ci possono essere **diverse versioni dello stesso file**

### Copy-on-Write File System

- Evita **sovrascritture** dei contenuti dei file
- Scrive **nuovi contenuti** in blocchi vuoti, poi aggiorna i metadati per puntare a i nuovi contenuti

# UNIX Gestione File

mercoledì 22 gennaio 2025 11:20

UNIX gestisce **sei tipi di file**:

- **normali**
- **directory**
- **file speciale** (mappano su nomi di file i dispositivi di I/O)
- **named pipe** (per la comunicazione tra processi)
- **hard link** (collegamenti)
- **link simbolici** (il suo contenuto è il nome del file cui si riferisce)

## Inode

Struttura dati **fondamentale** ispirata all'allocazione indicizzato, con dimensione fissa dei blocchi.

Contiene le **informazioni essenziali** di un file e vengono mantenuti:

- In una **tabella** gli inode di file **aperti in RAM**
- Gli altri inode sono in una **zona dedicata sul disco (i-list)**

Contengono diverse informazioni come il tipo e i permessi di accesso, l'id del proprietario, data di creazione e accesso, flag, dimensione, ecc.

Un inode può essere **associato a più nomi di file**:

- Hard link
- Ma un inode **attivo** è associato ad **un solo file**
- Ogni file è **controllato da un solo inode**

## File condivisi

Invece di creare una copia di un file per condividerlo si possono usare due soluzioni:

- **Symbolik links**: contengono il **percorso del file originale** e possono puntare a file non più esistenti
  - Comando: **In -s <fileOrigine> <fileDestinazione>**
- **Hard links**: **puntano all'inode del file originale**, incrementando un contatore nell'inode che indica quanti file lo referenziano  
Il file condiviso non può essere cancellato finché esiste un link remoto ad esso
  - Comando: **In <fileOrigine> <fileDestinazione>**

# Windows Gestione File

mercoledì 22 gennaio 2025 11:33

Esistono due metodi per la gestione dei file su Windows:

- File system **vecchio FAT**: allocazione concatenata, con blocchi di dim. fissa
- File system **nuovo NTFS**: allocazione con **bitmap**, con blocchi di dim. fissa

## File Allocation Table (FAT)

La FAT usa una **tabella di puntatori** per ogni **cluster** (blocco) del disco.

Ogni file è una **catena di indici**, che permette di accedere ai blocchi del file **sequenzialmente**.

Le **regioni del volume** in FAT includono le regioni:

- **Boot Sector**: informazioni necessarie per l'accesso al volume
- **FAT**: due copie della **file allocation table**, mappa il contenuto della regione dati
- **Root Directory**: directory table con tutti i file entry per la directory root del sistema
- **Regione dati**: dove sono contenuti i dati dei file e directory

## Svantaggi

- Dimensione massima di 4GB
- **Non implementa journaling**
- Non consente meccanismi di controllo di accesso ai file

## New Technology File System (NTFS)

Nuovo File System adottato da Windows NT dove viene usata una **bitmap** per l'allocazione dei blocchi.

I file sono definiti da un **insieme di attributi**, rappresentati come **byte stream**, e supporta **hard e soft link**

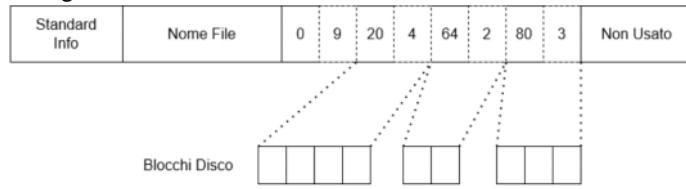
Le **regioni del volume** in NTFS includono le regioni:

- **Boot Sector**: basato su FAT
- **Master File Table (MFT)**: implementata come un file dove ogni **record** descrive un file ed è identificato con un puntatore
  - Ogni **record** contiene una lista di **attributi (attributo, valore)**
    - Il contenuto di un file è anche esso un attributo (\$DATA)
  - Il val. dell'attributo può essere incluso direttamente nel **record** (attributo **residente**)
  - Oppure può essere un **puntatore** ad un **record remoto** (attributo **non residente**)
  - I primi 27 record sono **riservati per i metadati del File System**
  - Dal 28 in poi ci sono i descrittori dei file normali

NTFS cerca di allocare sequenze contigue di blocchi ai file.

- I file piccoli (<1KB) sono salvati direttamente nell'MFT
- Per i file grandi, il val. di attributo indica la sequenza ordinata dei blocchi sul disco dove risiede il file

Per ogni file esiste un **record base** nell'MFT contenente **blocchi divisi in sezioni contigue (run)**



- ▶ File di 9 blocchi, divisi in 3 run

File di **large dimensioni** possono necessitare di **più record**. In questo caso NTFS usa una tecnica simile agli **i-node di UNIX**:

- Il **record base** è un **puntatore a N record secondari**
- Eventuale spazio rimanente nel record base contiene le prime sequenze del file

Se i record estesi **non rientrano in MFT** per mancanza di spazio, vengono **trattati come attributo non residente**, e quindi salvati in un file dedicato, con un apposito record puntatore salvato in MFT

# Sincronizzazione processi e thread

venerdì 22 novembre 2024 17:02

è necessaria la **sincronizzazione** tra processi e thread per via della presenza di **settori critici** (parte di codice che accede a risorse condivise). Per evitare che tali settori generino problemi, vengono usate delle **primitive** che si assicurano che **un solo thread possa lavorare sul settore stesso**.

## Atomico

il termine "**atomico**" si riferisce a operazioni o azioni che vengono eseguite in modo **indivisibile e ininterrotto**. È un'azione che viene **completata senza essere interrotta** o influenzata da altre operazioni in esecuzione contemporaneamente

## Proprietà della sincronizzazione

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

- **Mutua esclusione**, solo un processo/thread per volta può usare il sett. critico
- **Liveness**, possibilità per ogni processo di accedere al sett. critico nel caso in cui sia libero
- **Attesa limitata**, possibilità ad ogni processo richiedere l'accesso al sett. critico di potervi accedere eventualmente, limitando il num. di processi che vi accederanno prima di esso

Per la sincronizzazione, i linguaggi di programmazione forniscono **primitive atomiche** basate su **tre soluzioni**:

- **Lock**, prima di accedere ad sett. critico un processo **acquisisce** tale **lock** per poi **rilasciarlo** una volta uscito
- **Semaforo**, generalizzazione del lock
- **monitor**, connette dei dati condivisi alle primitive di sincronizzazione

## Esempio

Scenario con 2 coinvolti: **Bob** e **Carla**

Obiettivi della sincronizzazione:

- Garantire la **mutua esclusione** significa che non verrà acquistato più latte di quanto necessario (cioè solo uno dei due comprerà il latte se necessario)
- Garantire il **progresso** significa che qualcuno dovrà comprare del latte
- Garantire l'**attesa limitata** significa che eventualmente uno dei due entrerà nella sezione critica

Abbiamo 3 possibili soluzioni:

### ESEMPIO 1

```
# Thread Bob
if (!milk and !note)
    leave_note()
    buy_milk()
    remove_note()

# Thread Carla
if (!milk and !note)
    leave_note()
    buy_milk()
    remove_note()
```

Does this solution work regardless of the scheduling?  
[Not mutual exclusion can be violated]

Questa soluzione funziona?  
**Nol** La **mutua esclusione** può essere violata.

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

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

### ESEMPIO 2

```
# Thread Bob
leave_note(Bob)
if (!note(Carla)):
    if (!milk):
        buy_milk()
    remove_note()

# Thread Carla
leave_note(Carla)
if (!note(Bob)):
    if (!milk):
        buy_milk()
    remove_note()
```

Does this solution work regardless of the scheduling?  
[No! Liveness property can be violated]

Questa soluzione funziona?  
**Nol** Il **progresso** può essere violato.

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

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

### ESEMPIO 3

```
Use 2 (labeled) notes... more cleverly

# Thread Bob
leave_note(Bob)

while (!note(Carla)):
    do_nothing()
if (!milk):
    buy_milk()
remove_note()

# Thread Carla
leave_note(Carla)

if (!note(Bob)):
    if (!milk):
        buy_milk()
    remove_note()
```

Does this solution work regardless of the scheduling?  
[Yes!]

Questa soluzione funziona?

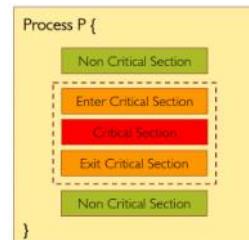
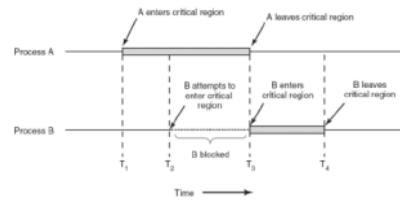
**Sil** Tutte le proprietà sono rispettate.

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

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

Pero la terza soluzione non è una soluzione valida poiché:

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



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

# Lock

venerdì 22 novembre 2024 18:06

Un **lock** fornisce la **mutua esclusione** tra processi usando 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** acquisito, inviando un **segnaletico** a tutti i **processi** che stanno eseguendo **acquire()**

Per poter usare correttamente il lock, è necessario seguire regole:

- 1) È necessario **acquisire il lock prima** di accedere a dati condivisi
- 2) È necessario **rilasciare il lock dopo** aver finito di usare 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 interviene solo in caso di **eventi interni** (quando il thread in esecuzione lascia il controllo della CPU) o **esterni**, per evitare problemi nei sett. critici è necessario **impedire lo scheduling** durante il rilascio/acquisizione di un lock.

Per implementare le primitive ad alto livello serve il **supporto hardware a basso livello** che garantisca **operazioni atomiche**, tramite la **disabilitazione degli interrupt** o l'uso di **istruzioni atomiche**.

## Disabilitazione degli interrupt

La **prima soluzione a supporto HW** per la sincronizzazione è la **disabilitazione degli interrupt** mentre un thread **acquisisce/rilascia un lock**

Necessitiamo che sia **acquire()** che **release()** siano implementati come **system calls**

Esempio Implementazione

```
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.isEmpty()) {  
        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();  
}
```

- All'inizio della classe Lock vengono **definiti** metodi e attributi, mentre nel costruttore vengono **inizializzati** gli attributi (**lock libero e coda vuota**)
- In **acquire()** si controlla che:
  - Se il **lock è libero**, viene occupato (impostato a 1 il val)
  - Se il **lock è già occupato**, il **thread** che ha chiamato la funz. si aggiunge alla **coda di attesa** e si mette in **attesa**.
- In **release()** si controlla che:
  - Se la **coda è vuota**, **value** viene **resetto** (**lock è libero**)
  - Altrimenti viene **estratto** un **thread** dalla **coda di attesa** e lo si manda in **esecuzione**
- In entrambe le funzioni gli **interrupt** sono **disattivati all'inizio** e **riattivati alla fine** tramite system call

Tale implementazione è **inattuabile su sistemi multi-core** poiché il messaggio della disabilitazione degli interrupts deve passare tra tutti i core, rallentando il sistema.

## Istruzioni atomiche

### test&set (compare and swap)

La **seconda sol. a supporto HW** è l'uso di un'**istruzione atomica** in grado di leggere un val. dalla mem. e modificarlo in un singolo istante.

Nella maggior parte delle architetture si usa l'istr. **test&set (compare and swap)**, che prende in **input** un val., lo legge (restituendolo al chiamante) e lo imposta a 1 in mem.:

- **Acquisizione del lock**: se il lock è **libero** (val. 0), l'istr. **test&set(value)** legge 0 per poi settare il val. a 1, restituendo 0 (**acquisizione del lock**)
- **Attesa del lock**: se il lock è **occupato** (val. 1), l'istr. **test&set(value)** legge 1, restituendo 1 (**attesa del lock**)

Esempio Implementazione

```
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;  
}  
  
int compare_and_swap (int word, int testval,  
                     int newval)  
{  
    int oldval;  
    oldval = word;  
    if (word == testval) word = newval;  
    return oldval;  
}  
test&set(L) = compare_and_swap(L, 0, 1)
```

Abbiamo 2 casi possibili:

- 1) Se il **lock è libero** (val. 0), **test&set** restituirà 0 e occuperà il lock (val. 1), l'espr. **bool** nel **while** è **False** e l'**acquire termina**
- 2) Se il **lock è occupato** (val. 1), **test&set** restituirà 1, l'espr. **bool** nel **while** è **True** e l'**acquire continua in loop** fin quando non si esegue un **release**.

## Exchange

Un altro metodo è l'uso di una funzione **exchange**, che prende in **input** due val., e li **scambia**.

Il processo funziona solo se **bolt è globale** tra i processi mentre **keyi è locale** per ogni processo, poiché il lock deve essere condiviso tra tutti i processi.

- **Acquisizione del lock**: se il **lock è libero** (bolt = 0), il processo scambia keyi (1) con bolt (0), impostando così il lock (bolt = 1) e uscendo dal while, entrando quindi nella **zona critica**
- **Attesa del lock**: se il lock è **occupato** (bolt = 1), il processo rimane in un loop in cui scambia keyi (1) con bolt (1)

Esempio Implementazione

```
/* program mutual exclusion */  
int const n = /* number of processes */;  
int bolt;  
  
void exchange (int register, int memory)  
{  
    ...  
}
```

- **Attesa del lock.** Se il lock è occupato (`bolt = 1`), il processo rimane in un loop in cui scambia `keyi` (`1`) con `bolt` (`1`)

### Esempio Implementazione

```
/* program mutual exclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
    int keyi;
    while (true) {
        keyi = 1;
        do exchange (keyi, bolt);
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), ..., P(n));
}
```

```
void exchange (int register, int memory)
{
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

Il processo può entrare nella sezione critica se `bolt = 0`, poiché:

- Se `bolt = 0`, l'`exchange(keyi, bolt)` scambia i val. **bloccando il lock** (`bolt = 1`) e permettendo al processo di **uscire dal while** (`keyi (0) != 0`), e dopo aver lavorato nella zona critica **rilascia il lock** (`bolt = 1`)
- Se `bolt = 1`, l'`exchange(keyi, bolt)` scambia i val. ma poiché entrambi sono 1, il processo **rimane nel while** (`keyi (1) != 0`) aspettando che un altro processo **rilasci il lock**

## Contro

### Contro della **disabilitazione degli interrupt**:

- Necessita di **invocare il kernel**, rende il tutto più complesso
- Inutilizzabile in sistemi multi-core

### Contro delle **istruzioni atomiche**:

- **Busy waiting:** ogni thread che non ha acquisito il lock è in attesa che esso si liberi, dunque la CPU non sta realmente svolgendo delle atti vità
- **Iniquità:** quando il lock è libero, i thread in attesa si contenderanno il lock, e non è possibile determinare chi otterrà il controllo

Però il tempo di attesa delle **istruzioni atomiche** può essere **minimizzato** e l'uso di una **queue** permette di gestire meglio quale thread vada ad ottenere il lock quando diventa libero

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

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

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

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

In questa implementazione, a differenza della prima, aggiungiamo due parametri: `Guard` e `Queue`. Supponiamo che un thread(`T1`) entri nell'`acquire()`, `T1` leggerà il valore 0, imposterà la `Guard` a 1 e ritornerà 0. Il valore booleano restituito dal `while` sarà `False`, quindi `T1` passerà direttamente al controllo dell'`if`. L'`if` darà come valore booleano `False` perché `value` all'inizio è uguale a 0. Supponiamo ora che `T1` venga interrotto e un altro thread(`T2`) provi ad acquisire la lock. `T2` leggerà il valore 1, imposterà la `Guard` a 1 e ritornerà 1. A questo punto il valore booleano del `while` ritornerà `True` e quindi `T2` entrerà nel `while` causando un attesa (**busy waiting**). Ora supponiamo che `T2` venga interrotto e che `T1` riprenda il controllo della CPU. `T1` ripartirà dall'`else`, e imposterà il valore di `value` a 1 e quello di `Guard` a 0. Supponiamo ora che `T2` riprenda il controllo. Nel momento della lettura del valore all'interno del `while`, il valore di `Guard` letto, ora sarà 0, di conseguenza imposterà il valore di `Guard` a 1 e ritornerà 0, uscendo così dall'attesa(**busy waiting**). Dopo ciò, passerà all'istruzione `if`, che restituirà il valore `True`, dato che ora `value` è uguale a 1, e `T2` verrà inserito in una coda di attesa e verrà messo a "dormire". `T2` verrà "svegliato" solo quando `T1` eseguirà l'istruzione `push_onto_ready_queue()` della funzione `release()`, che servirà a "svegliare" `T2` per metterlo nella ready queue.

Questa implementazione è più efficace della prima, perché a differenza della prima, `T2` rimane in attesa fino a quando `T1` non completa la funzione di `acquire()`, dopodiché può uscire dal `while` e può entrare in una coda, smettendo di occupare la CPU inutilmente. Nella prima implementazione invece `T2` deve attendere fino a quando `T1` non esegue tutta la funzione di `release()`, quindi fino a quando non rilascia totalmente la lock.

# Semafori

domenica 24 novembre 2024 19:02

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

- **wait() (o P()):**
  - **Decrementa la var.**
  - Se il val è negativo (non ci sono più risorse disponibili) il processo viene messo in **attesa nella queue**
  - Se la var è  $\geq 0$  il processo continua
- **signal() (o V()):**
  - **Incrementa la var.**
  - Sblocca un processo in attesa (se ci sono)
  - Se non ci sono processi, il segnale viene **ricordato** per dopo

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

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

I semafori si **differenziano** in due tipi:

- **Semafori binari (Mutex):**
  - Usati per gestire una singola risorsa condivisa.
  - Valore della var. è 1(libera) o 0(occupata). (simile al **lock**)
- **Semafori di conteggio:**
  - Usati per gestire risorse multiple.
  - Val. iniziale della var. associata è il num. 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;  
}
```

```
semWait(s)  
{  
    while (compare_and_swap(s.flag, 0 , 1) == 1)  
        /* do nothing */;  
    s.count--;  
    if (s.count < 0) {  
        /* place this process in s.queue*/;  
        /* block this process (must also set  
        s.flag to 0) */;  
    }  
    else  
        s.flag = 0;  
}  
semSignal(s)  
{  
    while (compare_and_swap(s.flag, 0 , 1) == 1)  
        /* do nothing */;  
    s.count++;  
    if (s.count >= 0) {  
        /* remove a process P from s.queue */;  
        /* place process P on ready list */;  
    }  
    s.flag = 0;  
}
```

```
semWait(s)  
{  
    inhibit interrupts;  
    s.count--;  
    if (s.count < 0) {  
        /* place this process in s.queue */;  
        /* block this process and allow inter-  
        rupts */;  
    }  
    else  
        allow interrupts;  
}  
semSignal(s)  
{  
    inhibit interrupts;  
    s.count++;  
    if (s.count <= 0) {  
        /* remove a process P from s.queue */;  
        /* place process P on ready list */;  
    }  
    allow interrupts;  
}
```

Sia con Istr. Atomiche che con Disabilitazione degli Interrupt

I semafori sono utili anche per risolvere problemi di sincronizzazione.

Ad esempio, consideriamo due processi in esecuzione:

- **P<sub>1</sub>** con istruzione S<sub>1</sub>
- **P<sub>2</sub>** con istruzione S<sub>2</sub>

Per richiedere che S<sub>2</sub> venga **eseguito solo dopo** che S<sub>1</sub> sia **completato** possiamo implementare questo schema con **P<sub>1</sub>** e **P<sub>2</sub>** che **condividono** una sincronizzazione del **semaforo comune**, inizializzato a 0

In **P<sub>1</sub>** inseriamo:

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

In **P<sub>2</sub>**, inseriamo:

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

## Esempi

### Semafori binari

```
# Variabile semaforo inizializzata a 1  
semaforo_stampante = 1  
  
def processo_usa_stampante():  
    wait(semaforo_stampante) # Decrementa il semaforo: risorsa  
    occupata  
    print("Stampante in uso")  
    signal(semaforo_stampante) # Incrementa il semaforo:  
    risorsa libera
```

Se un processo chiama **wait()** quando il semaforo è 0, dovrà aspettare che un altro processo lo sblocca con **signal()**

### Semafori di conteggio

```
# Variabile semaforo inizializzata al numero di risorse  
connessioni_disponibili = 3  
  
def processo_connessione():  
    wait(connessioni_disponibili) # Decrementa il semaforo:  
    una connessione occupata  
    print("Connessione al database in uso")  
    signal(connessioni_disponibili) # Incrementa il semaforo:  
    una connessione rilasciata
```

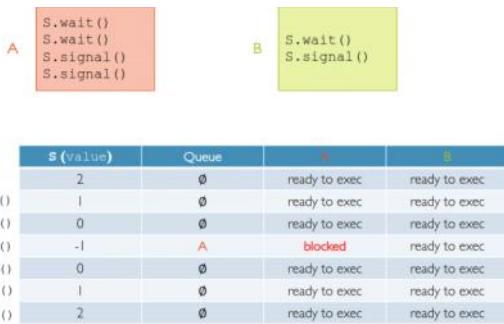
Se 4 processi cercano di accedere contemporaneamente, il quarto sarà messo in attesa finché una connessione non è rilasciata

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



```
    /* 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 var. **counter** tiene traccia del num. di elementi nel buffer
- Potrebbe crearsi una **race condition**: in assenza di sincronizzazione, i due processi potrebbero lavorare contemporaneamente sulla var.
- Ad es., ipotizziamo inizialmente che **counter = 5**. In tal caso, una possibile esecuzione del programma potrebbe essere:

#### Producer:

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

#### Consumer:

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

#### Interleaving:

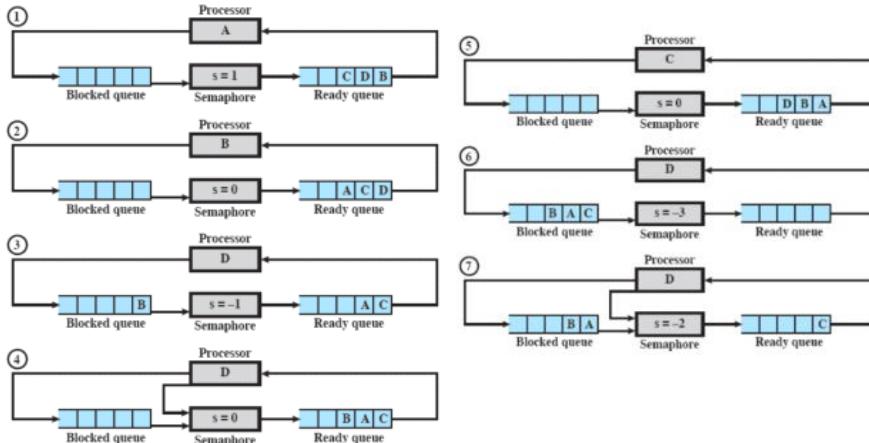
```
T0: producer execute register1 = counter {register1 = 5}
T1: producer execute register1 = register1 + 1 {register1 = 6}
T2: consumer execute register2 = counter {register2 = 5}
T3: consumer execute register2 = register2 - 1 {register2 = 4}
T4: producer execute counter = register1 {counter = 6}
T5: consumer execute counter = register2 {counter = 4}
```

## Semafori Deboli e Forti

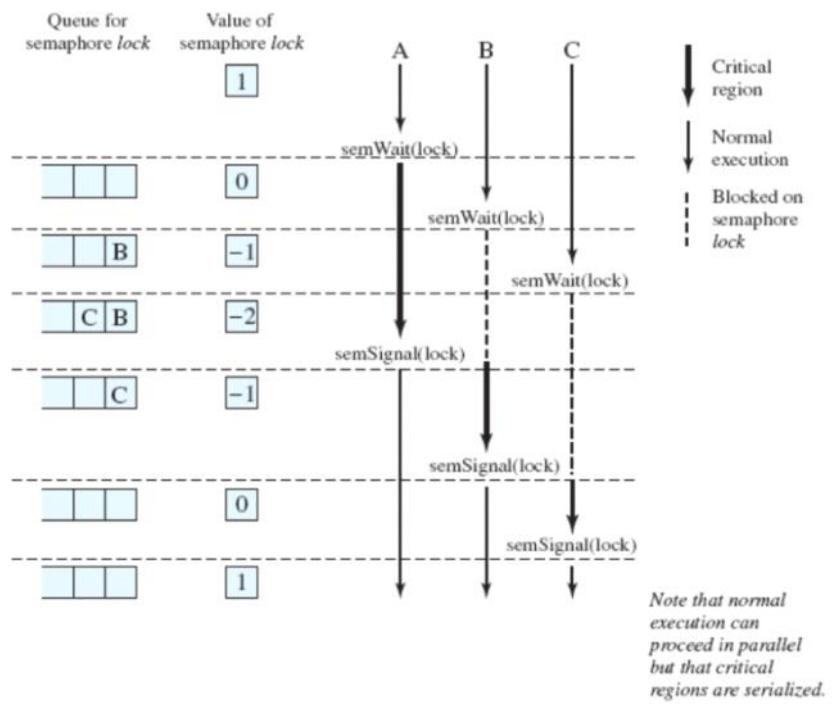
Il SO usa una coda per memorizzare i processi in attesa su un semaforo. In base alla politica usata per prendere il prossimo processo in coda si ha:

- Se è la politica **FIFO**, allora si parla di **semaforo forte**  
I semafori forti, se usati bene, possono evitare la **starvation**
- Se non viene specificata la politica, allora si parla di **semaforo debole**  
I semafori deboli, non possono evitare la starvation

## Esempi Semafori Forti



- 1) A.wait() | s = 1  $\Rightarrow$  s = 0 | A  $\Rightarrow$  ready
- 2) B.wait() | s = 0  $\Rightarrow$  s = -1 | B  $\Rightarrow$  blocked
- 3) D.signal() | s = -1  $\Rightarrow$  s = 0 | B  $\Rightarrow$  ready
- 4) D  $\Rightarrow$  ready
- 5) C.wait(), A.wait(), B.wait() | s = 0  $\Rightarrow$  s = -3 | C, A, B  $\Rightarrow$  blocked
- 6) D.signal() | s = -3  $\Rightarrow$  s = -2 | C  $\Rightarrow$  ready



# Produttore/Consumatore

martedì 21 gennaio 2025 18:03

Gia visto nella sezione I/O dei Buffer ([Problema Produttore/Consumatore](#))

Situazione generale:

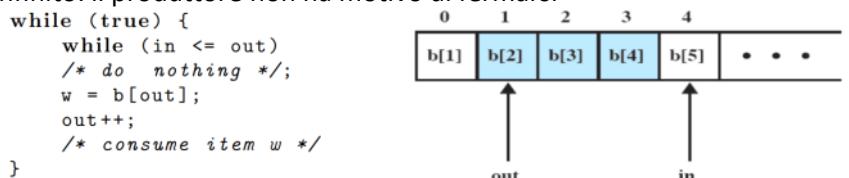
- Uno o più (processi) **produttori creano dati** e li mettono in un buffer
- Un **consumatore prende i dati** dal buffer **uno alla volta**
- Al buffer può **accedere solo un processo alla volta**

Problema:

- Assicurare che il produttore non inserisca dati quando il buffer è pieno
- Assicurare che il consumatore non prenda dati quando il buffer è vuoto
- **Mutua esclusione** del buffer

Per ora facciamo finta che il buffer si infinito: il produttore non ha motivo di fermarsi

```
while (true) {
    while (in <= out)
        /* do nothing */;
    w = b[out];
    out++;
    /* consume item w */
}
```



```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

**Soluzione Sbagliata**

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	if (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	if (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		if (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		if (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	-1	0
21		semSignalB(s)	1	-1	0

**Soluzione Sbagliata: Possibile Scenario**

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    int m; /* a local variable */
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
}
```

```
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
```

```
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

Soluzione Corretta

```
        take();
        semSignal(s);
        consume();
    }
void main()
{
    parbegin (producer, consumer);
}
```

Soluzione con Semafori Generali

# Monitor (non all'esame)

domenica 24 novembre 2024 19:38

Un **monitor** è un **costrutto** (struttura di controllo) dei linguaggi di programmazione in grado di controllare l'accesso ai dati condivisi. I processi che tentano di accedere al monitor vengono messi in una **queue**.

Un monitor è costituito da un **lock**, usato per assicurare la mutua esclusione sui dati e che un solo thread sia attivo nel monitor, e **zero o più var. di condizione**, usate per gestire gli accessi concorrenti ai dati.

Ogni var. di condizione supporta tre op.:

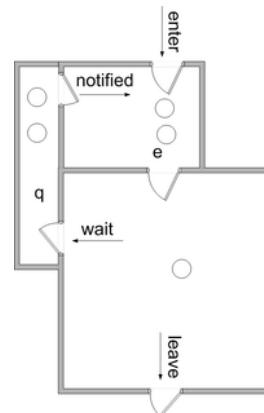
- **wait()**: il lock viene **liberato** e il thread viene messo a dormire nella queue
- **signal (o notify())**: un processo nella queue (se esistente) viene svegliato, altrimenti nulla
- **broadcast (o notifyAll())**: tutti i thread vengono svegliati

Durante lo svolgimento di op. su var condizionali, è necessario che il lock sia **chiuso**

## Esempio

Le var. 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, quindi serve che il thread stia già accedendo al monitor. Se nessun thread è nella queue(), il segnale **signal()** viene perso
- Nei semafori, **wait()** blocca solo il thread nella queue, mentre il segnale **signal()** viene preservato



Comunemente i monitor vengono implementati in due stili:

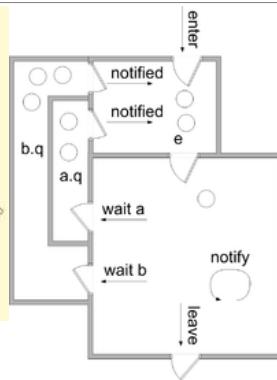
### Stile Mesa

Dove il thread segnalante **inserisce un thread in attesa** in **ready** mentre il primo continua l'esec. nel 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;
    }
}
```



### Codice Mesa Monitor

```
enter the monitor:
    enter the method
    if the monitor is locked
        add this thread to e
        block this thread
    else
        lock the monitor
leave the monitor:
    schedule
    return from the method
wait c:
    add this thread to c.q
    schedule
    block this thread
notify c:
    if there is a thread waiting on c.q
        select and remove one thread t from c.q
        (t is called "the notified thread")
        move t to e
notify all c:
    move all threads waiting on c.q to e
schedule :
    if there is a thread on e
        select and remove one thread from e and restart it
    else
        unlock the monitor
```

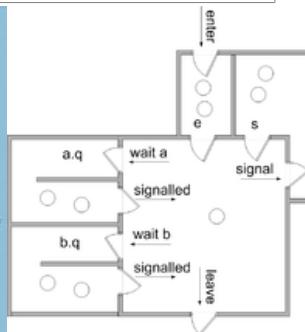
### Stile Hoare

Dove il thread segnalante viene **trasformato** in un thread di **attesa** e dove la condizione anticipata dal thread in attesa è **garantita** quando 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;
    }
}
```



### Codice Hoare Monitor

```
enter the monitor:
    enter the method
    if the monitor is locked
        add this thread to e
        block this thread
    else
        lock the monitor
leave the monitor:
    schedule
    return from the method
wait c:
    add this thread to c.q
    schedule
    block this thread
signal c:
    if there is a thread waiting on c.q
        select and remove one such thread t from c.q
        (t is called "the signaled thread")
        add this thread to s
        restart t
        (so t will occupy the monitor next)
        block this thread
schedule:
    if there is a thread on s
        select and remove one thread from s and restart it
        (this thread will occupy the monitor next)
    else if there is a thread on e
        select and remove one thread from e and restart it
        (this thread will occupy the monitor next)
    else
        unlock the monitor
        (the monitor will become unoccupied)
```

(this thread will occupy the monitor next)  
else  
    unlock the monitor  
    (the monitor will become unoccupied)

# Dekker/Peterson

mercoledì 22 gennaio 2025 23:37

(Purtroppo sto scrivendo questi argomenti sotto esame e non metto molto in dettaglio il codice degli algoritmi in s)  
L'algoritmo di Dekker e l'algoritmo di Peterson sono due soluzioni **software** al problema della **mutua esclusione** tra processi.

## Algoritmo di Dekker

Necessario inizializzare Flag = 0 e turn = 0 o 1

- **Vantaggio:** evita race condition, deadlock e starvation e non richiede supporto OS
- **Svantaggio:** usa busy waiting

```
p0:  
    wants_to_enter[0] ← true  
    while wants_to_enter[1] {  
        if turn ≠ 0 {  
            wants_to_enter[0] ← false  
            while turn ≠ 0 {  
                // busy wait  
            }  
            wants_to_enter[0] ← true  
        }  
    }  
  
    // critical section  
    ...  
    turn ← 1  
    wants_to_enter[0] ← false  
    // remainder section  
  
p1:  
    wants_to_enter[1] ← true  
    while wants_to_enter[0] {  
        if turn ≠ 1 {  
            wants_to_enter[1] ← false  
            while turn ≠ 1 {  
                // busy wait  
            }  
            wants_to_enter[1] ← true  
        }  
    }  
  
    // critical section  
    ...  
    turn ← 0  
    wants_to_enter[1] ← false  
    // remainder section
```

## Algoritmo di Peterson

- **Vantaggi:** come Dekker per starvation, deadlock e mutua esclusione
- **Svantaggi:** anche lui usa **busy waiting**

```
boolean flag [2];  
int turn;  
void p0()  
{  
    while (true) {  
        flag [0] = true;  
        turn = 1;  
        while (flag [1] && turn == 1) /* do nothing */;  
        /* critical section */;  
        flag [0] = false;  
        /* remainder */;  
    }  
}  
void p1()  
{  
    while (true) {  
        flag [1] = true;  
        turn = 0;  
        while (flag [0] && turn == 0) /* do nothing */;  
        /* critical section */;  
        flag [1] = false;  
        /* remainder */;  
    }  
}  
void main()  
{  
    flag [0] = false;  
    flag [1] = false;  
    parbegin (p0, p1);  
}
```

# Messaggi

giovedì 23 gennaio 2025 13:54

Quando un processo interagisce con un altro devono essere **soddisfatti** due **requisiti**:

- **Sincronizzazione** (mutua esclusione)
- **Comunicazione**

Lo **scambio di messaggi** (message passing) è una soluzione al secondo requisito, funziona sia con mem. condivisa che distribuita

Può anche essere utilizzato per la **sincronizzazione**.

Vengono usate **due primitive atomiche**:

- **send(destination, message)**: (message è un input)
- **receive(source, message)**: (message è un output)

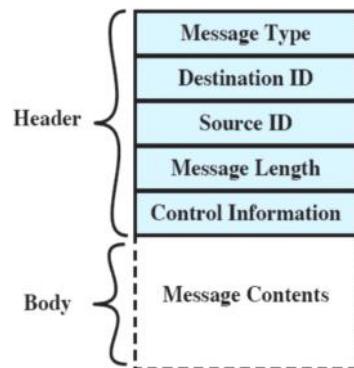
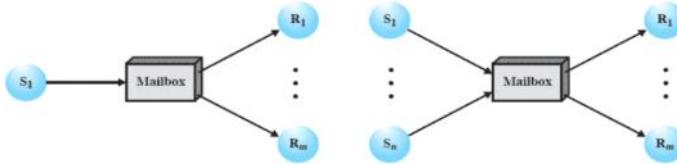
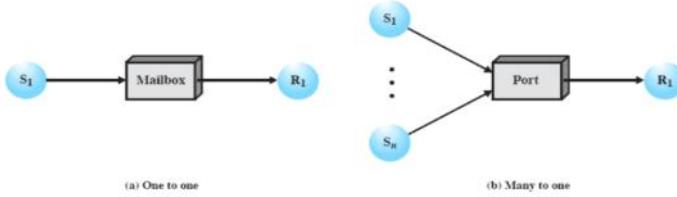
È richiesta anche la **sincronizzazione** (il mittente deve inviare prima che il ricevente riceva)

Possiamo applicare in due modi queste primitive:

- **Send e Receive bloccanti**: dove sia il mittente che il destinatario sono **bloccati** finché il messaggio **non è ricevuto completamente**
  - Sincronizzazione molto stretta
- **Send non bloccante (nbsend)**:
  - **Ricezione bloccante** (caso molto comune): il **mittente continua** e il **destinatario è bloccato** finché all'arrivo del messaggio
  - **Ricezione non bloccante** (nbreceive): se il messaggio c'è allora viene ricevuto, altrimenti si va avanti
    - può settare un **bit** dentro al messaggio per indicare se la **ricezione è avvenuta o no**
    - se la ricezione è non bloccante, tipicamente neanche l'invio lo è

Per indicare a quale destinatario (processo/i) si vuole mandare il messaggio, si può usare:

- **Indirizzamento diretto**:
  - send include un ID per il destinatario
  - per la receive ci può essere (receive(sender, msg) oppure no (receive(null, msg) così che riceva da chiunque)
- **Indirizzamento indiretto**:
  - I messaggi sono inviati ad una **casella di posta (mailbox) condivisa**
  - Il mittente manda messaggi alla mailbox, dove il destinatario li va a prendere  
La prima riga, in realtà, è la comunicazione diretta



## Lettori/Scrittori

giovedì 23 gennaio 2025 14:09

- Un'area dati è condivisa tra molti processi
- Alcuni la leggono, altri la scrivono
- Condizioni da soddisfare:
  - più lettori possono leggere l'area contemporaneamente
  - solo uno scrittore può scrivere nell'area
  - se uno scrittore e all'opera sull'area, nessun lettore può effettuare letture
- Differenza con i produttori/consumatori: l'area condivisa si accede per intero
  - niente problemi di buffer pieno o vuoto
  - ma è importante permettere ai lettori di accedere contemporaneamente

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
void writer()
{
    while (true) {
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
    }
}
void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

Sol. precedenza lettori

```
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        semSignal (rsem);
        semSignal (z);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
void writer ()
{
    while (true) {
        semWait (y);
        writecount++;
        if (writecount == 1) semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount--;
        if (writecount == 0) semSignal (rsem);
        semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Sol. precedenza scrittori

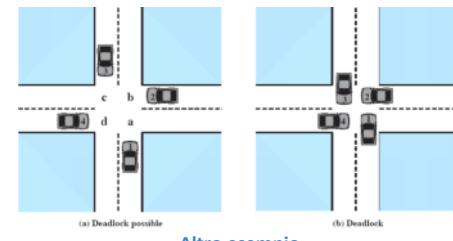
# Deadlock

lunedì 25 novembre 2024 16:09

Il **Deadlock** è una situazione in cui **due o più processi** che **competono per le stesse risorse**, creando un **blocco perenne**.

## Esempio

Thread A	Thread B
<pre>printer.wait(); disk.wait();  // copy from disk to printer  printer.signal(); disk.signal();</pre>	<pre>disk.wait(); printer.wait();  // copy from disk to printer  printer.signal(); disk.signal();</pre>



Altro esempio

- **printer e disk** sono regolate da un **mutex**, dunque c'è un'unica risorsa per entrambi i tipi
- A esegue l'istr. **acquisisce** la risorsa **printer**, decrementando il mutex e eseguendo un context switch, cedendo il controllo sulla CPU a B
- Successivamente, B **acquisisce** la risorsa **disk**, effettuando il context switch e cedendo il controllo della CPU ad A
- Ora, A richiederà di **acquisire il disk**. Tuttavia, poiché il mutex è 0, **non** vi sono più risorse **disk** disponibili, A rimarrà in attesa che la risorsa venga liberata (da B), cedendo il controllo della CPU a B
- Tuttavia B richiede di **acquisire il printer**, pero anche B rimarrà in attesa che **printer** venga liberata (da A)
- Infine, si crea una situazione dove A è in **attesa** che B **liberi disk**, che è in **attesa** che A **liberi printer**, creando un **deadlock**

## Esempio 2: Filosofi a cena

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

Soluzioni:

- **lock globale**: un singolo filosofo alla volta può prendere in mano due bacchette. Tuttavia **l'assenza di concorrenza** implica che non possono mangiare contemporaneamente
- **Semaforo**:

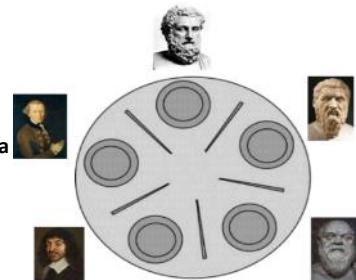
```
Semaphore chopsticks[5];

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 se ogni filosofo prende la bacchetta sinistra contemporaneamente, si creerebbe un **deadlock** perché ogni filosofo sarebbe in attesa che la bacchetta destra si liberi

- **Monitor**: Prima di prendere in mano una delle due bacchette, è necessario assicurarsi che la seconda sia libera, altrimenti bisogna aspettare che si liberi.  
Dunque, sarà necessario controllare se il filosofo alla destra e alla sinistra stiano mangiando oppure no (**var. condizionate**)

## Condizioni Deadlock (risorse consumabili)

Usando **risorse consumabili** affinché un **deadlock possa verificarsi**, sono necessarie **tutte e quattro le seguenti condizioni**:

- **Mutua esclusione**: la risorsa **va a chi riesce a fare richiesta per primo**
- **Possesso e Attesa (hold-and-wait)**: richiesta di una **risorsa** quando **se ne ha già una**.  
diventa: **si può richieder, in modo bloccante, una risorsa** quando ancora **non sia stata creata**
- **No-preemption**: non appena viene concessa una risorsa, essa viene distrutta
- **Attesa circolare**: si ha una **catena di thread t1, ..., tn** in cui ognuno di essi deve **creare una risorsa richiesta** dal processo che lo segue nella catena

## Condizioni Deadlock (risorse riusabili)

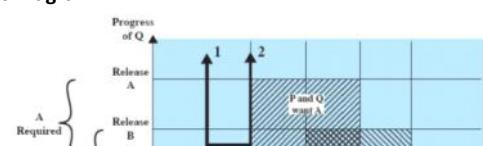
Usando **risorse riusabili** affinché un **deadlock possa verificarsi**, sono necessarie **tutte e quattro le seguenti condizioni**:

- **Mutua esclusione**: almeno un thread **possiede** una risorsa **non condivisibile** (solo un thread può possedere la risorsa)
- **Possesso e Attesa (hold-and-wait)**: almeno un thread **possiede** una risorsa **non condivisibile** ed è in **attesa** che un'altra risorsa posseduta sia **disponibile**
- **No-preemption**: un thread **può rilasciare** una risorsa solo per **scelta volontaria**, dunque né un altro processo né un altro thread possono **forzare il rilascio**
- **Attesa circolare**: si ha una **catena di thread t1,...,tn** in cui ognuno di essi è in **attesa che il thread successivo** appartente alla catena stessa **rilasci la risorsa richiesta**

## Joint Progress Diagram

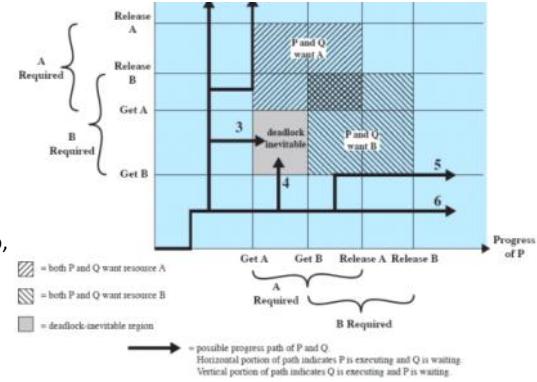
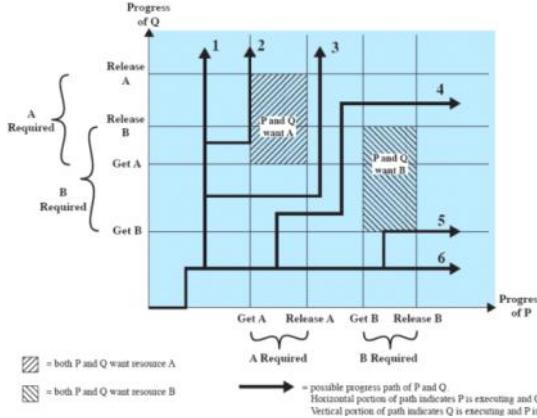
Le situazioni di **deadlock** tra pochi processi possono essere **analizzati** tramite il **Joint Progress Diagram**

- Viene disegnato un diagramma della timeline di due processi (P e Q nel nostro caso)
- Sia P che Q dovranno accedere alle risorse A e B
- Quando chiedono una risorsa, chiedono anche l'altra prima di rilasciare la prima risorsa (es. P chiede A, chiede B e poi rilascia A e infine rilascia B)
- Col grafico possiamo **identificare le zone** in cui:



- Quando chiedono una risorsa, chiedono anche l'altra prima di rilasciare la prima risorsa (es. P chiede A, chiede B e poi rilascia A e infine rilascia B)
- Col grafico possiamo **identificare le zone** in cui:
  - il deadlock non c'è
  - il deadlock è possibile
  - il deadlock è inevitabile
- In questo caso il deadlock è causato dal fatto che P chiede B, mentre Q chiede A ma entrambe le risorse sono tenute dall'altro processo (Q tiene B e P tiene A)  
Quindi un processo ha già una risorsa, vuole l'altra che però è tenuta dall'altro processo, stessa cosa per l'altro processo
- Le frecce sono le **possibili evoluzioni dei processi**:
  - Le frecce orizzontali indicano che P è in esecuzione e Q è in attesa
  - Le frecce verticali indicano che Q è in esecuzione e P è in attesa

Altro esempio in cui non c'è il deadlock poiché il processo P richiede le due risorse separatamente



## Risorse Riusabili

Sono risorse usabili da **un solo processo alla volta** ma non vengono "consumate" quindi quando vengono rilasciate un altro processo può utilizzarle a loro volta (es. processori, mem. dispositivi).

Il deadlock esiste solo se un processo ha già una risorsa e ne richiede un'altra.

Esempi

**Process P**

Step	Action
p <sub>0</sub>	Request (D)
p <sub>1</sub>	Lock (D)
p <sub>2</sub>	Request (T)
p <sub>3</sub>	Lock (T)
p <sub>4</sub>	Perform function
p <sub>5</sub>	Unlock (D)
p <sub>6</sub>	Unlock (T)

**Process Q**

Step	Action
q <sub>0</sub>	Request (T)
q <sub>1</sub>	Lock (T)
q <sub>2</sub>	Request (D)
q <sub>3</sub>	Lock (D)
q <sub>4</sub>	Perform function
q <sub>5</sub>	Unlock (T)
q <sub>6</sub>	Unlock (D)

Supponiamo che ci siano 200 KB di memoria disponibili, e che ci sia la seguente sequenza di richieste

P1  
...  
Request 80 Kbytes;  
...  
Request 60 Kbytes;

P2  
...  
Request 70 Kbytes;  
...  
Request 80 Kbytes;

Il deadlock avverrà quando uno dei due processi farà la seconda richiesta

## Risorse Non Riusabili

Vengono **create** e poi **distrutte** (consumate). Il deadlock è possibile solo se si fa una richiesta (bloccante) di una risorsa non ancora creata

Esempio

P1  
...  
Receive(P2);  
...  
Send(P2, M1);

P2  
...  
Receive(P1);  
...  
Send(P1, M2);

Deadlock inevitabile, processi scritti molto male...

Approccio	Politica di Allocazione	Possibili Schemi	Vantaggi Principali	Svantaggi Principali
Prevenire	Conservativa: concede meno risorse di quelle richieste	Richiesta contemporanea di tutte le risorse	OK per processi con singolo <i>burst</i> di computazione; Non richiede preemption	Inefficiente: ritarda l'inizializzazione dei processi; Un processo deve conoscere tutte le sue richieste future, difficile per gli interattivi
		Preemption	OK se le risorse hanno uno stato facile da salvare e ripristinare	La preemption può avvenire troppo spesso
		Ordinamento delle risorse	Possibile con controlli a tempo di compilazione; Niente controlli a run-time, risolto con il progetto del SO	Non possibile per processi interattivi
Evitare	A metà tra prevenire e rilevare	Si cerca di trovare almeno un cammino sicuro	Niente prevenzione	Necessità di risorse future da conoscere in anticipo
Rilevare	Molto liberale: concede più risorse di quelle possibili	Controllo del deadlock da fare periodicamente	Niente ritardo sull'inizializzazione; Facilita la gestione delle risorse online	Gesione del deadlock quando avviene

# Rilevare e Prevenire Deadlock

martedì 26 novembre 2024 12:13

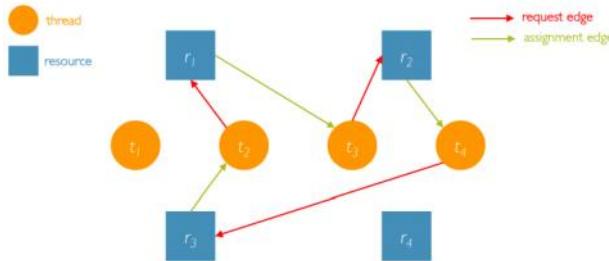
## Resource Allocation Graph (RAG)

È un modello astratto per **determinare** e rappresentare eventuali situazioni di **deadlock**

Un **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 di archi nel grafo, di due tipi:
  - **Archi di richiesta (rossi)**, indicante che il thread  $t_i$  ha richiesto  $r_j$
  - **Archi di assegnamento (verdi)**, indicanti che l'OS ha assegnato  $r_k$  a  $t_h$

### Esempi

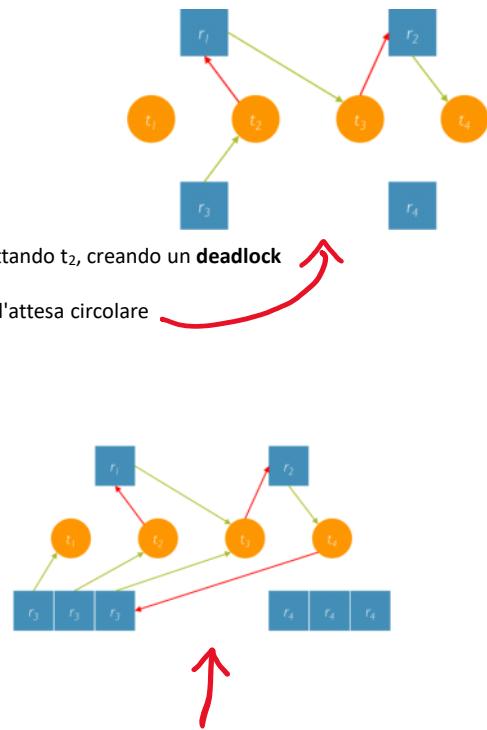
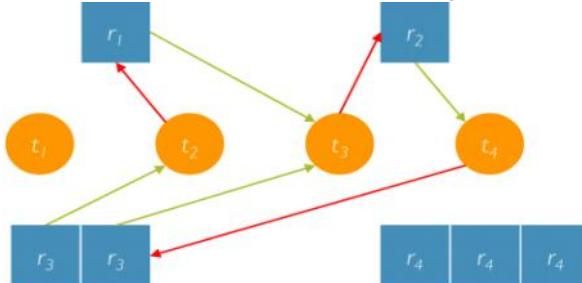


È facile individuare il deadlock:

- 1)  $t_2$ , possidente  $r_3$ , richiede  $r_1$ , rimanendo in attesa che essa sia liberata da  $t_3$
- 2) A questo punto  $t_3$ , possidente  $r_1$ , richiede  $r_2$ , rimanendo in attesa che sia liberata da  $t_4$
- 3) A sua volta  $t_4$ , possidente  $r_2$ , richiede  $r_3$ , rimanendo in attesa che sia liberata da  $t_2$
- 4) Infine, si è in una situazione in cui  $t_2$  aspetta  $t_3$ , il quale sta aspettando  $t_4$ , che a sua volta sta aspettando  $t_2$ , creando un **deadlock**

Se  $t_4$  non richiedesse  $r_3$ , le condizioni per il deadlock **non sarebbero soddisfatte**, per via dell'assenza dell'attesa circolare

Consideriamo il caso in cui ci sono due istanze di  $r_3$ , una assegnata a  $t_2$  e una a  $t_3$



Anche in questo caso siamo in una situazione di deadlock:

- 1)  $t_4$  richiede  $r_3$ , rimanendo in attesa che  $t_2$  o  $t_3$  ne rilascino almeno una
- 2) Tuttavia  $t_3$  è in attesa che  $t_4$  ceda  $r_2$ , mentre  $t_2$  è in attesa che  $t_3$  ceda  $r_1$

Invece se ci fossero **tre istanze di  $r_3$**  e solo due fossero usate da  $t_2$  e  $t_3$ , non si creerebbe un deadlock, poiché una volta che  $t_1$  rilascia  $r_3$ , l'attesa di  $t_4$  terminerebbe rompendo il ciclo

## Prevenire ed evitare un deadlock

Un deadlock si verifica quando si verificano queste quattro condizioni:

- Mutua esclusione
- Hold-and-wait: richiesta di una risorsa quando se ne ha già una
- Niente preemption: un thread rilascia la risorsa solo su volontà propria
- Attesa circolare: si ha una catena di thread in cui ognuno attende che il prossimo thread rilasci la risorsa e così via

Nel caso di deadlock possiamo:

- **Prevenire**: è sufficiente che una sola delle quattro condizioni necessarie non si verifichi
  - Mutua esclusione: non si può prevenire
  - Hold-and-wait: si impone ad un processo di richiedere tutte le risorse in una volta sola
  - Niente preemption: l'OS richiede ad un processo di rilasciare le risorse
  - Attesa circolare: una risorsa viene data solo se segue l'ordinamento che il processo già detiene
- **Evitare**: decidere di volta in volta cosa fare con l'assegnazione di risorse
- **Rilevare**: L'OS lascia che ci sia un deadlock e ogni tanto decide che fare quando incontra un deadlock

Inoltre, è possibile **studiare il comportamento** di una sequenza n di thread per determinare se sia sicura o no

- 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:

$C = \text{somma del numero di risorse occupate da tutti i thread}$

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

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  $t_j$ , dove  $j < i$

Verifica che  $t_i$  possa essere eseguito con le risorse attualmente disponibili e quelle che verranno liberate dai thread precedenti ( $t_1, \dots, t_{i-1}$ )

### Esempio

Immaginiamo:

- 3 thread:  $t_1, t_2, t_3$ .
- Numero massimo di risorse:  $R = 10$ .
- Risorse attualmente allocate:
  - $c_1 = 2, c_2 = 3, c_3 = 1 \rightarrow C = 2 + 3 + 1 = 6$ .
- Richieste massime:
  - $m_1 = 4, m_2 = 6, m_3 = 3$ .

Verifichiamo la sequenza  $t_1, t_2, t_3$ :

1. Per  $t_1$ :

$$m_1 - c_1 = 4 - 2 = 2$$

Risorse disponibili:

$$R - C + \sum_{j=1}^0 c_j = 10 - 6 + 0 = 4$$

$2 \leq 4$ , quindi  $t_1$  è sicuro.

2. Per  $t_2$ :

$$m_2 - c_2 = 6 - 3 = 3$$

Risorse disponibili dopo  $t_1$ :

$$R - C + \sum_{j=1}^1 c_j = 10 - 6 + 2 = 6$$

$3 \leq 6$ , quindi  $t_2$  è sicuro.

3. Per  $t_3$ :

$$m_3 - c_3 = 3 - 1 = 2$$

Risorse disponibili dopo  $t_1$  e  $t_2$ :

$$R - C + \sum_{j=1}^2 c_j = 10 - 6 + (2 + 3) = 9$$

$2 \leq 9$ , quindi  $t_3$  è sicuro.

### Policy a stato sicuro

Lo **stato sicuro** è 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 max. num. di risorse necessarie dichiarato.

Tramite tale **policy**, una risorsa viene ceduta al thread **solo se il nuovo stato è sicuro** (se quando viene ceduta la risorsa non ci sia un'altra risorsa che potrebbe richiedere la stessa risorsa creando un deadlock), altrimenti tale thread rimarrà in attesa che la risorsa sia disponibile in modo sicuro

### Algoritmo per Rilevare Deadlock

- Occorre dichiarare il massimo uso di risorse
- I processi devono essere liberi di andare in esecuzione in qualsiasi ordine, l'unica sincronizzazione è quella sulla richiesta di risorse
- Non va bene per risorse consumabili
- Nessun processo deve terminare senza rilasciare le sue risorse
- **Algoritmo:**
  - 1) marca tutti i processi che **non hanno allocato nulla**
  - 2)  $w \leftarrow V$
  - 3) sia  $i$  un **processo non marcato** t.c.  $Q_{ik} \leq w_k \forall 1 \leq k \leq m$ 
    - le sue risorse **possono essere accordate**
  - 4) se  $i$  è **non esiste**, vai al passo 6
  - 5) marca  $i$  e aggiorna  $w \leftarrow w + A_i$ ; poi torna al passo 3
  - 6) c'è un **deadlock** solo se **esiste un processo non marcato**

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
        claim [k,*] = alloc [k,*] <= currentavail>
        if (found) {
            /* simulate execution of Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

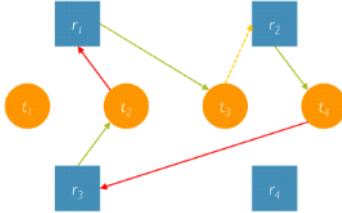
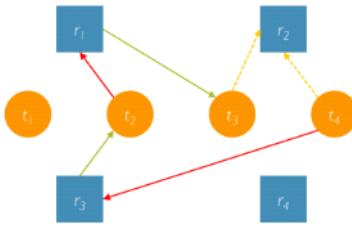
### Archi di pretesa

Viene usata una variante del RAG avente un tipo di arco in più, ossia gli **archi di pretesa**, indicanti che  $t_i$  **potrebbe richiedere**  $r_j$  in futuro.

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

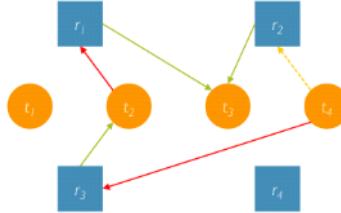
### Esempio:

Consideriamo il seguente RAG



Se viene **soddisfatta** la pretesa di  $t_4$  si andrebbe a creare uno **stato insicuro** (perché se  $t_4$  richiede  $r_3$  si creerebbe un deadlock).

Dunque tale pretesa **non viene soddisfatta**



Se viene **soddisfatta** la pretesa di  $t_3$  si rimarrebbe in uno **stato sicuro** (perché se  $t_4$  richiede  $r_3$  esso rimarrebbe in attesa di  $t_3$ , che può continuare a lavorare poiché non è in attesa di nessuno).

Dunque tale pretesa **viene soddisfatta**

Approccio	Politica di Allocazione	Possibili Schemi	Vantaggi Principali	Svantaggi Principali
Prevenire	Conservativa: concede meno risorse di quelle richieste	Richiesta contemporanea di tutte le risorse	OK per processi con singolo <i>burst</i> di computazione; Non richiede preemption	Inefficiente: ritarda l'inizializzazione dei processi; Un processo deve conoscere tutte le sue richieste future, difficile per gli interattivi
		Preemption	OK se le risorse hanno uno stato facile da salvare e ripristinare	La preemption può avvenire troppo spesso
		Ordinamento delle risorse	Possibile con controlli a tempo di compilazione; Niente controlli a run-time, risolto con il progetto del SO	Non possibile per processi interattivi
Evitare	A metà tra prevenire e rilevare	Si cerca di trovare almeno un cammino sicuro	Niente prevenzione	Necessità di risorse future da conoscere in anticipo
Rilevare	Molto liberale: concede più risorse di quelle possibili	Controllo del deadlock da fare periodicamente	Niente ritardo sull'inizializzazione; Facilita la gestione delle risorse online	Gestione del deadlock quando avviene

# Sicurezza

venerdì 6 dicembre 2024 15:13

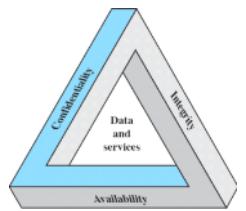
La **sicurezza** è la protezione offerta da un OS al fine di **conservare integrità, disponibilità e confidenzialità** delle risorse del sistema stesso.

Ci sono tre **obiettivi** principali che costituiscono il cuore della sicurezza

- **Integrità:** riferita ai **dati**, che **non** devono essere **modificati senza** le dovute **autorizzazioni**
- **Disponibilità:** riferita ai **servizi**, che devono essere **disponibili senza interruzioni**
- **Confidenzialità:** riferita ai **dati**, che **non** devono essere **letti senza** le dovute **autorizzazioni**

E altri due che vengono aggiunti alla sicurezza informatica

- **Autenticità:** riferita agli **utenti**, che devono essere **chi dichiarano di essere** (vale anche per messaggi e dati)
- **Tracciabilità (accountability)**



## Minacce (Threats)

L'**RFC 2828** (glossario di sicurezza dell'Internet) descrive quattro conseguenze delle minacce informatiche:

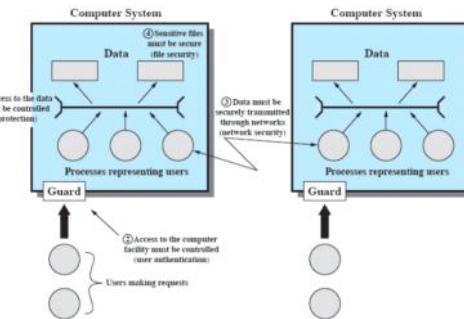
<b>Accesso non autorizzato</b> <ul style="list-style-type: none"><li>• Un'entità ottiene l'<b>accesso</b> a dati per i quali <b>non ha autorizzazione</b></li><li>• Minaccia alla <b>confidenzialità</b></li></ul> <b>Attacchi</b> <ul style="list-style-type: none"><li>• <b>Esposizione</b> (intenzionale o per errore)</li><li>• <b>Intercettazione</b> (es. man in the middle)</li><li>• <b>Inferenza</b></li><li>• <b>Intrusione</b></li></ul>	<b>Imbroglino</b> <ul style="list-style-type: none"><li>• Un'entità <b>autorizzata</b> riceve <b>dati falsi</b> e pensa siano <b>veri</b></li><li>• Minaccia all'<b>integrità dell'OS o dei dati</b></li></ul> <b>Attacchi</b> <ul style="list-style-type: none"><li>• <b>Mascheramento:</b> l'attaccante riesce ad entrare in possesso delle credenziali di un utente autorizzato (trojan)</li><li>• <b>Falsificazione</b> (es. uno studente che modifica i suoi voti)</li><li>• <b>Ripudio:</b> un utente nega di aver ricevuto o inviato dei dati</li></ul>
<b>Interruzione</b> <ul style="list-style-type: none"><li>• Impedimento al corretto <b>funzionamento</b> dei <b>servizi</b></li><li>• Minaccia all'<b>integrità o disponibilità</b> dell'OS</li></ul> <b>Attacchi</b> <ul style="list-style-type: none"><li>• <b>Incapacitazione:</b> rompe qualche componente dell'OS</li><li>• <b>Ostruzione: Denial of Service (DoS)</b> (es. riempiendo l'OS di richieste)</li><li>• <b>Corruzione:</b> alterazione dei servizi</li></ul>	<b>Usurpazione</b> <ul style="list-style-type: none"><li>• L'OS viene <b>controllato</b> da chi non ne ha l'autorizzazione</li><li>• Minaccia all'<b>integrità dell'OS</b></li></ul> <b>Attacchi</b> <ul style="list-style-type: none"><li>• <b>Appropriazione indebita:</b> diventare admin di una macchina non propria (es. botnet)</li><li>• <b>Uso non appropriato</b> (es. virus che cancella file)</li></ul>

## Asset

Gli **asset (risorsa)** di un OS possono essere categorizzati come:

- **HW**
- **SW**
- **Dati**
- **Linee di comunicazione e reti**

	Disponibilità	Confidenzialità	Integrità
<b>Hardware</b>	Workstation rubate o rese inutilizzabili		
<b>Software</b>	Programmi cancellati	Copia non autorizzata dei programmi	Modifica dei programmi (per non farli funzionare o per fargli fare compiti indesiderati)
<b>Dati</b>	File cancellati	File letti senza autorizzazione. Dati infetti da analisi statistica	Modifica di file esistenti o creazione di file
<b>Comunicazione</b>	Messaggi distrutti. Linee di comunicazione rese inutilizzabili	Lettura dei messaggi o osservazione dei pattern	Modifica, ritardo, ribordo o duplicazione di messaggi esistenti, creazione di messaggi falsi



# Autenticazione

venerdì 6 dicembre 2024 16:19

L'autenticazione è la **base** per la maggior parte dei tipi di **controllo di accesso e tracciabilità**.

Essa si divide in due passi:

- **Identificazione**
- **Verifica**

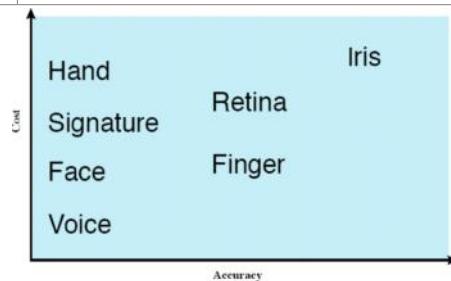
E determina se un **utente** è **abilitato** ad accedere al sistema e quali **privilegi** l'utente ha

Rende possibile il **discretionary control access** (controllo di accesso discrezionale), in cui un utente può decidere a quali utenti concedere determinati permessi.

## Mezzi per l'Autenticazione

Tradizionalmente si divide in 3 **fattori**. Almeno uno deve essere presente ma è meglio due contemporaneamente (auth. a 2 fattori)

Qualcosa che sai	Qualcosa che hai	Qualcosa che sei
<p>Esempio</p> <p><b>Password</b></p> <ul style="list-style-type: none"><li>• Quella più nota ed usata</li><li>• Spesso anche l'unica</li><li>• È importante che le password siano memorizzate non in chiaro</li></ul>	<p>Oggetti fisici da un utente per l'autenticazione vengono chiamati <b>token</b></p> <p>Esempio</p> <p><b>Memory Card</b></p> <ul style="list-style-type: none"><li>• Possono memorizzare dati, ma senza elaborarli (bancomat tradizionali)</li><li>• Spesso usati insieme a password o PIN</li><li>• Svantaggi<ul style="list-style-type: none"><li>◦ Serve un lettore apposito</li><li>◦ Può essere perso</li></ul></li></ul> <p><b>Smartcard</b></p> <ul style="list-style-type: none"><li>• Hanno un microprocessore, memoria e porte I/O</li><li>• Esistono di diversi tipi, a seconda degli aspetti:<ul style="list-style-type: none"><li>◦ Caratteristiche fisiche (carta di credito o USB)</li><li>◦ Interfaccia (lettore apposito o tastierino)</li><li>◦ Protocollo di autenticazione (generatore di password o domanda-risposta)</li></ul></li></ul>	<p>Esempio</p> <p><b>Biometrica</b> che si divide in <b>Biometrica statica</b> Basata su riconoscimento di pattern<ul style="list-style-type: none"><li>• Caratteristiche facciali</li><li>• Impronte digitali</li><li>• Geometria della mano</li><li>• retina</li><li>• iride</li></ul><p><b>Biometrica dinamica</b> I pattern possono cambiare<ul style="list-style-type: none"><li>• Firma</li><li>• Voce</li><li>• Ritmo di battitura</li></ul></p></p>



# Accesso

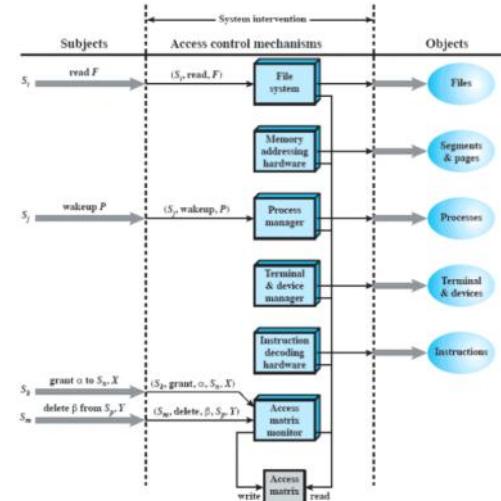
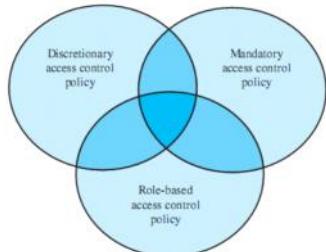
venerdì 6 dicembre 2024 16:31

**Il controllo d'accesso** determina quali tipi di accesso **sono ammessi**, sotto quali **circostanze** e **da chi**

Può essere:

- **Discrezionale**: un utente può **concedere** i suoi stessi **privilegi** ad **altri utenti**
- **Obbligatorio**: un utente **non** può **concedere** i suoi stessi privilegi ad altri utenti
- **Basato su ruoli**

Le 3 modalità possono essere presenti contemporaneamente ovviamente applicate a diverse classi di risorse



Organizzazione del Controllo di Accesso

		subjects		files		processes		disk drives		OBJECTS	
		S <sub>1</sub>	S <sub>2</sub>	S <sub>3</sub>	F <sub>1</sub>	F <sub>2</sub>	P <sub>1</sub>	P <sub>2</sub>	D <sub>1</sub>	D <sub>2</sub>	
SUBJECTS	S <sub>1</sub>	control	owner	owner control	read *	read owner	wakeup	wakeup	seek	owner	
	S <sub>2</sub>		control		write *	execute			owner	seek *	
	S <sub>3</sub>			control		write	stop				

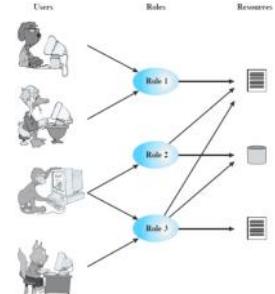
\* = copy flag set

Controllo di Accesso Discrezionale

## Controllo di Accesso basato sui Ruoli

Implementa il principio di **minimo privilegio** in cui ciascun ruolo deve contenere il **minimo insieme di diritti d'accesso** per il ruolo stesso.

Un utente viene assegnato ad un ruolo, che lo abilita ad effettuare le op. richieste per quel ruolo



## Matrice del controllo di Accesso: Rappresentazione RBAC

	R <sub>1</sub>	R <sub>2</sub>	• • •	R <sub>n</sub>
U <sub>1</sub>	X			
U <sub>2</sub>	X			
U <sub>3</sub>		X		X
U <sub>4</sub>				X
U <sub>5</sub>				X
U <sub>6</sub>				X
•				
U <sub>m</sub>	X			

ROLES	OBJECTS								
	R <sub>1</sub>	R <sub>2</sub>	R <sub>n</sub>	F <sub>1</sub>	F <sub>1</sub>	P <sub>1</sub>	P <sub>2</sub>	D <sub>1</sub>	D <sub>2</sub>
R <sub>1</sub>	control	owner	owner control	read *	read owner	wakeup	wakeup	seek	owner
R <sub>2</sub>		control		write *	execute			owner	seek *
•									
R <sub>n</sub>			control		write	stop			

# UNIX: Meccanismi di Protezione

venerdì 6 dicembre 2024 16:39

I meccanismi principali sono:

- Autenticazione dell'utente (User-Oriented Access Control)
- Diritti o permessi di accesso ai dati (Data-Oriented Access Control)
- Altri come
  - NIS
  - LDAP
  - Kerberos

## Utenze e Gruppi

- Per ogni utente c'è un **username** (alfanumerico) e un **uid** (numerico intero)
- L'**uid** è usato ogni volta che occorre dare un proprietario ad una risorsa (file, processi, ...)
- Ogni utente appartiene ad un **gruppo**, analogamente identificato da **groupname** e **gid**
- **File di sistema:** /etc/group e /etc/passwd

## Login

- Può essere fatto sul **terminale** della macchina (proc. getty) o tramite **rete** (telnet, ssh)
- Richiede una coppia **username+password**
- Se corrisponde ad una **entry** in /etc/passwd, viene eseguita la shell **ivi** indicata, a partire dalla dir. di home ivi indicata
- Quando la shell esegue **exit**, o si ritorna al getty o si chiude la connessione di rete
- All'interno di una shell si può **cambiare l'identità** col comando **su**

## Accesso ai File

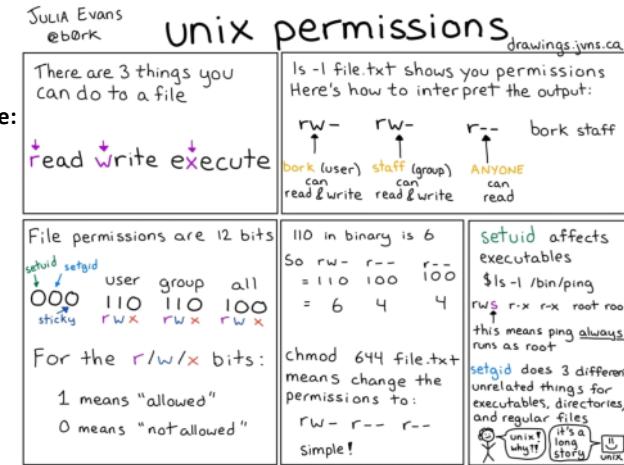
- Per ogni file ci sono **tre terne di permessi di lettura, scrittura, esecuzione**:
  - La **prima terna** è per il **proprietario** del file
  - La **seconda** è per il **gruppo** in cui il proprietario appartiene
  - La **terza** è per **tutti gli altri utenti**
- Il proprietario è lo stesso del **processo che ha creato il file**, ma si può cambiare con **chown**
- I diritti si possono cambiare con **chmod**:

Esempio:

```
-rwxr-xr-x 1 federico em 5120 Nov 7 11:03 a.out  
-rw-r--r-- 1 federico em 233 Nov 7 11:03 test.c
```

## SETUID e STGID

- Le terne di diritti sono usate ogni volta che un processo richiede l'accesso ad un file.
  - Se il proprietario del file e del processo coincidono, si guarda la prima terna
  - Altrimenti, se almeno appartengono allo stesso gruppo, si guarda la seconda terna
  - Altrimenti la terza terna
- Si prende poi l'elemento della terna corrispondente all'accesso richiesto
- I comandi come **passwd** hanno il **permesso speciale SETUID e/o SETGID**
- Tale permesso può essere accordato solo da un utente **admin** con **chmod u+s nomefile e/o chmod g+s nomefile**
- Vuol dire che l'**uid** o il **gid** del processo non sono quelli dell'utente che lo ha lanciato, ma del proprietario del file eseguibile



# Robe esame

mercoledì 27 novembre 2024 17:18

In base ai due esami sul classroom del De Gaspari del 2023-24, questi sono gli argomenti dell'esame

- OS
  - Kernel del Sistema Operativo
    - Funzioni principali del kernel ✓
    - Gestione dei processori e multitasking ✓
    - Differenze tra kernel monolitico e modulare
- Processi
  - Process Control Block (PCB)
    - Struttura e contenuti del PCB ✓
    - Informazioni sul contesto del processo e loro aggiornamento ✓
    - Ruolo del PCB nella gestione dei processi ✓
- Scheduling
  - Preemption e Scheduling
    - Scheduler preemptive e non-preemptive ✓
    - Ruolo del dispatcher nei process switch ✓
    - Trattamento equo dei processi e monopolizzazione della CPU ✓
  - Scheduling dei Dischi
    - Algoritmi SCAN e C-SCAN: funzionamento e caratteristiche ✓
    - Equità degli algoritmi di scheduling ✓
    - Differenze tra scheduling random e ottimo
  - Algoritmi di Scheduling e Gestione della Concorrenza
    - Algoritmi per evitare starvation ✓
    - Uso di semafori deboli e forti ✓
    - Comparazione tra disabilitazione degli interrupt e meccanismi più complessi
- Gestione della Memoria
  - Memoria Virtuale e Paginazione
    - Traduzione degli indirizzi virtuali in fisici ✓
    - Tabelle delle pagine a uno o due livelli ✓
    - Impatto della dimensione delle pagine sulla frammentazione e sulla multiprogrammazione
  - Gestione della Memoria
    - Indirizzi fisici, logici e relativi ✓
    - Requisiti di rilocazione e paginazione ✓
    - Tabella delle pagine e suo utilizzo ✓
- Gestione I/O
  - Dispositivi di I/O
    - Funzionamento dell'I/O buffering ✓
    - Modalità di trasferimento dati tra memoria e dispositivi ✓
    - Velocità di trasferimento (data rate) e differenze tra dispositivi
- File System
  - File System
    - Struttura della directory entry in Linux
    - File system journaling
    - Metadati e il loro utilizzo
    - Requisiti di un file management system
- Concorrenza
  - Concetti di Concorrenza
    - Sezioni critiche e operazioni atomiche ✓
    - Mutua esclusione e prevenzione della starvation
    - Uso di semafori e istruzioni macchina per la sincronizzazione
- Deadlock
  - Deadlock
    - Joint progress diagram per analisi del deadlock (rivedere la lezione video)
    - Condizioni necessarie per il deadlock
    - Algoritmi per rilevamento e gestione del deadlock
- Sicurezza
  - Controllo di Accesso
    - Controllo di accesso discrezionale (DAC) e basato su ruoli (RBAC)
    - Tabelle soggetti-oggetti e soggetti-ruoli-oggetti
    - Funzionamento dei modelli di controllo di accesso

## Tipologia esame 24/25

In base ai commenti degli studenti che l'hanno passato l'esame è diviso in:

- 1° parte: 20 domande a risposta multipla (simili o uguali alle domande del bot di telegram)
- 2° parte (se superato 80% della 1° parte): 1/2 domande risposta aperta su argomenti generali possibili domande:
  - spiegare come vengono salvate le password su linux, quale metodo tra hashing e encryption viene usato e perché e spiegare un attacco per il cracking delle password elencando vantaggi e svantaggi
  - Quali sono le condizioni necessarie per un deadlock in un OS con risorse riusabili? una volta rilevato quali sono le opzioni per risolverlo? elencare descrivendo vantaggi e svantaggi
  - La disabilitazione e l'abilitazione degli interrupt è un possibile approccio per la mutua esclusione, quali sono i vantaggi e gli svantaggi?

Non so quanto sia cambiato dal 2014/2015/2016 ma un ragazzo (Alessandro Francucci) ha riassunto in un pdf i 3 tipi di domande che possono essere all'esame (ricordo che è di 10 anni fa circa)

### Livello 1

Il livello uno tratta semplicemente domande a scelta multipla.(True o False) E il livello piu` semplice, e richiede un minimo del 50 % di risposte esatte ` per essere superato. Consigli: Avere ben chiari i vari tipi di scheduler e il significato di preemption (ovvero di preelazione, che significa che il processo puo' essere momentaneamente interrotto, per dar spazio ad altri processi.)

- Definizione e ruolo del sistema operativo
  - Differenza tra software applicativo e di sistema.
- Scheduling dei processi
  - Algoritmi di scheduling (FCFS, SPN, Round Robin, SRT, HRRN).
  - Concetti di starvation, preemption e non-preemption.
- Sistemi multilivello
  - Condivisione di risorse e sistemi operativi in ambiente multilivello.
- Disk scheduling
  - Politiche di scheduling (C-Scan) e loro funzionamento.
- Kernel del sistema operativo
  - Responsabilità del kernel (gestione processore, I/O).
- DMA (Direct Memory Access)
  - Funzione e ruolo del DMA nei sistemi di I/O.
- Gestione della memoria
  - Partizionamento fisso e frammentazione (interna ed esterna).
  - Concetto di memoria virtuale (paginazione, traduzione indirizzi, tavola delle pagine).
  - Problemi di thrashing e località.
- Mutua esclusione e sincronizzazione dei processi
  - Uso di semafori e primitive atomiche (compare&swap, exchange).
  - Problemi di deadlock e starvation.
  - Problema dei lettori-scrittori.
- Descrittore di processo (PCB)
  - Elementi contenuti nel Process Control Block (identificatori, tabella delle pagine, stato del processo).
- Sistemi operativi Linux
  - Notazione simbolica e ottime dei permessi (esempio: RW-, R--).
  - File eseguibili e processi.
- Politiche di sostituzione delle pagine
  - Principio di località e algoritmo LRU (Least Recently Used).

### Livello 2

Il livello due, a differenza del primo, tratta domande a risposta semi-aperta, oltre a quelle a risposta multipla. (Che sono pressoché lo stesso stile del livello 1) Occorre infatti, per alcuni esercizi di questo livello, capire la metodologia da applicare. (cosa per nulla difficile e alquanto meccanica.)

#### 1. Gestione della memoria

- Translation Lookaside Buffer (TLB) e associative mapping.
- Segmentazione: vantaggi e svantaggi (frammentazione interna ed esterna).
- Paginazione e segmentazione combinate.
- Frammentazione esterna e interna in diversi sistemi di gestione della memoria (partizionamento dinamico e partizioni fisse).
- Traduzione degli indirizzi (da logico a fisico) in memoria virtuale.
- Numero massimo di elementi nelle tabelle delle pagine in un sistema a memoria paginata.

#### 2. Processi e scheduling

- Transizioni degli stati di processo (ready, running, blocked, ecc.).
- Funzioni dello scheduler nei modelli di processo.
- Prerilascio del processore a causa della riattivazione di processi a priorità superiore.
- Scheduler Highest Response Ratio Next (calcolo del rapporto R e comportamento non preemptivo).
- Modelli di processo a 7 stati.

#### 3. Deadlock

- Prevenzione e gestione del deadlock.
- Mutua esclusione e comunicazione tra processi per evitare il deadlock.

#### 4. Chiamate di sistema in Linux

- Famiglia di chiamate exec() (exec(), execv(), ecc.):
  - Sostituzione del processo in esecuzione.
  - Assegnazione del Process ID (PID).
  - Valore restituito dalla funzione main.

#### 5. Calcolo delle performance del sistema

- Calcolo della durata di una singola istruzione sulla base del numero di istruzioni eseguite al secondo.

#### 6. Input/output (I/O)

- Operazioni di I/O in un sistema multiprogrammato:
  - Tempo di turnaround di processi concorrenti con scheduler round robin.

#### 7. DMA (Direct Memory Access)

- Tempo di trasferimento dati su disco basato su velocità di rotazione e dimensioni dei blocchi.

#### 8. Sistemi di memoria

- Organizzazione della memoria centrale:
  - Eliminazione della frammentazione esterna con partizioni fisse.
  - Vincoli di contiguità della memoria fisica.

### Livello 3

L'ultimo livello è la parte più difficile dell'esame (rispetto alle altre...). Questa parte sarà infatti strutturata da 3 domande, tutte sulla concorrenza. Le prime 2 saranno per/o a risposta multipla, con tanto di penalty nel caso di risposta errata, mentre l'ultima riguarda la stesura di codice. Arriviamoci per/o gradualmente.

**1. Concorrenza e variabili condivise**

- Esecuzione concorrente di processi che accedono a variabili condivise.
- Problemi di race condition (condizioni di gara).
- Esempio di processi con istruzioni atomiche e ordine di esecuzione variabile.

**2. Sincronizzazione tramite semafori**

- Uso dei semafori per sincronizzare l'accesso a risorse condivise.
- Problemi legati al coordinamento di più processi (esempio: stampa di caratteri con semafori).

**3. Produttore-consumatore**

- Modello classico con buffer finito.
- Sincronizzazione con semafori (semWait e semSignal).
- Gestione della mutua esclusione tra produttore e consumatore.

**4. Test-and-set con attesa attiva**

- Implementazione del produttore-consumatore usando l'istruzione test-and-set per la mutua esclusione.
- Utilizzo dell'attesa attiva (busy waiting) in assenza di semafori.

**5. Doppi consumatori**

- Sincronizzazione tra un produttore e due consumatori.
- Gestione di più code FIFO condivise tra i processi.
- Scelta della coda da cui consumare in base alla disponibilità di dati.

**6. Doppio buffer**

- Comunicazione tra processi tramite doppio buffer.
- Sincronizzazione del passaggio di dati tra buffer usando semafori.

**7. Barriere di sincronizzazione**

- Implementazione di barriere per coordinare processi in esecuzione concorrente.
- Sblocco dei processi una volta raggiunta la barriera.

Ogni esercizio tratta concetti fondamentali di programmazione

# Robe ChatGPT esame

mercoledì 22 gennaio 2025 23:13

Certamente. Ecco una lista degli argomenti principali relativi ai sistemi operativi, basata sulle domande e risposte del file "so1.txt", con una breve descrizione e i relativi vantaggi e svantaggi quando disponibili:

- **Directory di un file system:** Le directory sono usate per organizzare i file all'interno di un file system.
  - Non è sempre necessario identificare un file con il suo percorso assoluto; si può usare il percorso relativo.
- **Disabilitazione delle interruzioni:** La disabilitazione delle interruzioni può essere utilizzata per la gestione della concorrenza, ma non è efficace in sistemi con più processori o core.
  - Questo metodo è utile per garantire l'atomicità di alcune operazioni, ma non è scalabile.
- **Scheduler in sistemi monoprocesso:** Lo scheduler ha lo scopo di minimizzare il tempo di inattività del processore.
  - Uno degli obiettivi dello scheduler è di massimizzare l'uso della CPU.
- **Stati dei processi in UNIX SVR4:** In UNIX SVR4, un processo può trovarsi in diversi stati, tra cui lo stato Zombie, in cui il processo è terminato ma il suo Process Control Block (PCB) è ancora in memoria.
  - Lo stato "Asleep in Memory" corrisponde allo stato "Blocked".
- **Memoria virtuale con paginazione:** La memoria virtuale con paginazione è una tecnica per gestire la memoria.
  - Quando un indirizzo non si trova nel Translation Lookaside Buffer (TLB), è necessario consultare la tabella delle pagine.
- **Obiettivi di sicurezza di un sistema operativo:** Gli obiettivi di sicurezza includono la disponibilità, la confidenzialità e l'integrità dei dati.
  - La "disponibilità" si riferisce alla protezione contro l'interruzione del servizio.
  - La "confidenzialità" garantisce che gli utenti non accedano a informazioni non autorizzate.
  - L'integrità protegge i dati da modifiche non autorizzate.
- **Buffering dell'I/O:** Il buffering dell'I/O è una tecnica per migliorare le prestazioni.
  - Il completamento di un'istruzione di output può avvenire anche dopo l'esecuzione di alcune istruzioni successive.
- **Gerarchia della memoria:** La gerarchia della memoria è organizzata in base alla velocità, al costo e alla frequenza di accesso.
  - Andando dall'alto verso il basso, la frequenza di accesso alla memoria da parte del processore diminuisce.
- **Process Control Block (PCB):** Il PCB contiene informazioni necessarie per la gestione di un processo, ma non include l'identificatore del thread.
  - Il PCB contiene informazioni come il puntatore alla tabella delle pagine, lo stato e l'identificatore del processo.
- **Algoritmi di scheduling in sistemi monoprocesso:** Diversi algoritmi di scheduling hanno diversi vantaggi e svantaggi.
  - Nessuna delle opzioni fornite nella domanda è corretta.
- **Segmentazione della memoria:** La segmentazione della memoria permette di avere segmenti di diverse lunghezze.
  - Il programmatore assembler non interagisce direttamente con la gestione dei segmenti.
- **Algoritmo di rilevamento del deadlock:** Questo algoritmo individua i processi coinvolti in un deadlock.
  - I processi marcati non sono coinvolti in un deadlock.
- **Long-term scheduler in sistemi monoprocesso:** Il long-term scheduler decide quali processi ammettere in memoria principale per l'esecuzione.
  - Non viene chiamato in causa esclusivamente quando viene creato un nuovo processo.
- **Memoria virtuale con paginazione (prepaging vs. paging on demand):** Il prepaging può caricare pagine non necessarie, mentre il paging on demand può causare molti page fault all'inizio.
  - Nessuna delle opzioni fornite nella domanda è corretta.
- **Requisiti della mutua esclusione:** Un meccanismo di mutua esclusione deve evitare assunzioni sulla velocità di esecuzione dei processi.
  - Non è detto che un processo che richiede l'entrata nella sezione critica debba entrare subito.
- **Principio di località e thrashing:** Il principio di località afferma che le pagine di un processo che vengono accedute spesso sono relativamente poche. Il thrashing si verifica quando l'overhead della paginazione è alto.
  - Nessuna delle opzioni fornite è corretta.
- **Scambio messaggi per la gestione della concorrenza:** Lo scambio di messaggi può essere usato per la gestione della concorrenza.
  - Se un processo chiama receive, finché il messaggio non viene ricevuto, tutti gli altri processi che proveranno a chiamare receive verranno bloccati.
- **File system:** I file system possono usare il journaling per la gestione delle scrittture.
  - I file system che adottano il metodo journaling mantengono un log per le operazioni di sola scrittura da effettuare, realizzandole in seguito.
- **Dispositivi di I/O:** Non tutti i dispositivi di I/O scambiano informazioni con la CPU in blocchi.
  - Nessuna delle opzioni fornite è corretta.
- **Metodi di gestione dello spazio libero su disco:** La lista di blocchi liberi ha un overhead di spazio.
  - Nessuna delle opzioni fornite è corretta.
- **Process switch e mode switch:** In un SO dove le funzioni di sistema sono eseguite all'interno dei processi utente, il salvataggio del contesto del programma è necessario sia per il process switch che per il mode switch.
- **Page cache:** In un'algoritmo di sostituzione basato su frequenza a 2 segmenti, un blocco passa da un segmento all'altro esclusivamente per scorrimento.
  - Quando un settore viene acceduto spesso, ma tra il primo accesso e quelli successivi ci sono N accessi ad altri settori, con N pari alla dimensione del segmento nuovo, l'algoritmo potrebbe non avere buone performance.
- **Kernel di un sistema operativo:** Il kernel è responsabile della gestione dei processi.
  - Il kernel è la parte di sistema operativo che si trova sempre in memoria principale.
- **Controllo di accesso:** Nel controllo di accesso discrezionale, è necessario consultare una tabella soggetti-oggetti.
- **Preeemption in sistemi monoprocesso:** Se uno scheduler è preemptive, non è possibile che un processo monopolizzi il processore.
- **Scheduler round-robin virtuale:** Lo scheduler round-robin virtuale migliora il round-robin classico, facendo sì che i processi I/O-bound non vengano sfavoriti.
- **Indirizzi di memoria principale:** Gli indirizzi relativi sono usati nella paginazione.
- **Termini della concorrenza:** Una sezione critica non è semplicemente una porzione di memoria con variabili condivise.
  - Una race condition non è una violazione della mutua esclusione
- **Process Control Block (PCB):** Il PCB non contiene l'intera immagine del processo in memoria.
  - Nessuna delle altre opzioni contiene elementi del process control block.
- **Directory di un file system:** Le directory non memorizzano il gruppo cui appartiene l'utente che ha creato il file.
- **Algoritmi di scheduling per dischi:** L'algoritmo C-SCAN deriva da SCAN ed è stato sviluppato per evitare di favorire le richieste di tracce ai bordi del disco.
- **Allocazione contigua dei file:** La compattazione permette di memorizzare file che altrimenti non potrebbero esserlo.
- **Paginazione della memoria:** Frame e pagine devono avere la stessa dimensione.
- **Controllo di accesso:** Nel controllo di accesso discrezionale, prima di stabilire se un'operazione è legittima, è necessario consultare una tabella soggetti-oggetti.
- **Grafo dell'allocazione delle risorse:** Un grafo dell'allocazione delle risorse non è acillico.
- **Disponibilità di un sistema operativo:** La disponibilità di un sistema operativo consiste nel fatto che il sistema operativo deve essere sempre pronto a rispondere alle richieste di un utente.
- **Memoria virtuale con paginazione:** Il difetto principale del prepaging è che potrebbe portare in memoria pagine cui poi non si fa riferimento.
- **Dispositivi di memoria di massa:** Nessuna delle opzioni fornite è corretta.
- **Semafori per la gestione della concorrenza:** Per implementare le primitive sui semafori, servono un contatore ed una coda, che non saranno condivisi da tutti i semafori usati.
- **Algoritmi di scheduling per dischi:** L'algoritmo LIFO non è il più equo.
- **Mecanismi per la gestione della concorrenza:** Non è possibile garantire la mutua esclusione senza usare semafori, scambi messaggi o istruzioni atomiche.
- **Long-term scheduler in sistemi monoprocesso:** Il long-term scheduler non viene chiamato in causa esclusivamente quando viene creato un nuovo processo.
- **Metodi di gestione dello spazio libero su disco:** Se viene usata la lista di blocchi liberi, c'è un overhead di spazio, contrariamente alla concatenazione di blocchi liberi.
- **Directory di un file system:** Nessuna delle opzioni fornite è corretta.
- **Memoria cache:** È possibile che, in un dato istante, la cache e la memoria RAM non siano coerenti tra loro.
- **Problemi dei produttori/consumatori e dei lettori/scrittori:** Nessuna delle opzioni è corretta.
- **(Vecchi) metodi per il partizionamento della memoria:** Con il partizionamento dinamico si manifesta il problema della frammentazione esterna.

Certamente, ecco una lista degli argomenti principali delle domande e risposte presenti nel file "so1.txt", con descrizioni tratte dalle slide di sistemi operativi 1 e integrazioni da altre fonti, ove necessario, e con eventuali vantaggi e svantaggi:

- **File System e Directory:** Un file system organizza i dati su un dispositivo di memoria di massa. Una directory è una struttura che contiene informazioni sui file.
  - **Path Assoluto vs. Relativo:** Un path assoluto specifica la posizione di un file a partire dalla radice del file system, mentre un path relativo specifica la posizione a partire dalla directory corrente.
  - **Descrizione:** Il file system gestisce l'organizzazione dei file e delle directory, consentendo agli utenti di accedere e manipolare i dati memorizzati.
- **Interruzioni (Interrupt):** Mecanismi che interrompono l'esecuzione normale del processore per gestire eventi esterni o errori.
  - **Le interruzioni:** Le interruzioni permettono all'sistema operativo di reagire tempestivamente a eventi come l'arrivo di dati da un dispositivo I/O o un errore hardware. Le interruzioni sono gestite tramite degli interrupt handler.
  - **Disabilitazione Interruzione:** La disabilitazione delle interruzioni impedisce al processore di essere interrotto. Questo può essere utile in sezioni critiche ma non funziona in sistemi con più processori/core.
  - **Vantaggi:** Permettono una gestione efficiente degli eventi, senza dover attendere attivamente lo stato di un dispositivo.
  - **Svantaggi:** L'uso improprio della disabilitazione delle interruzioni può causare la diminuzione della multiprogrammazione.
- **Scheduler:** Componente del sistema operativo che decide quale processo deve essere eseguito dal processore in un dato momento.
  - **Descrizione:** Gli scheduler sono fondamentali nei sistemi multiprogrammati per massimizzare l'uso del processore, minimizzare i tempi di risposta e garantire l'equità tra i processi. Esistono diverse tipologie di scheduler (long-term, medium-term, short-term).
  - **Obiettivi:** Minimizzare il tempo di inattività del processore. Gli scheduler devono essere efficienti e avere un basso overhead.
  - **Algoritmi di Scheduling:**
    - **First Come First Served (FCFS):** Semplice, ma può portare a starvation.
    - **Round Robin (RR):** Equo, ma le performance dipendono dal quanto di tempo.
    - **Shortest Process Next (SPN):** Può portare a starvation dei processi con grande immagine in RAM.
    - **Shortest Remaining Time (SRT):**
    - **Virtual Round Robin:** Migliora il RR per processi I/O-bound.
    - **Highest Response Ratio Next:** Considera sia il tempo di esecuzione stimato che il tempo trascorso in attesa.
- **Processi:** Un programma in esecuzione, caratterizzato da uno stato e da risorse.
  - **Descrizione:** I processi sono gestiti dal sistema operativo, che si occupa di allocare loro memoria, schedulerli e monitorarne il loro stato . Il **Process Control Block (PCB)** contiene le informazioni necessarie per la gestione di un processo.
  - **Stati:** I processi possono trovarsi in diversi stati (ad esempio, Ready, Running, Blocked, Zombie, Asleep in memory). In particolar e, il modello a 7 stati si sposta i vari stati del modello a 5 con stati "suspend".
    - Uno stato "Zombie" indica che un processo è terminato ma il suo PCB è ancora in memoria.
  - **Transizioni:** I processi passano da uno stato all'altro (es. Running ->Ready, Blocked->Ready).
  - **Process Switch:** Il passaggio dall'esecuzione di un processo all'esecuzione di un altro.
- **Memoria Virtuale con Paginazione:** Tecnica per gestire la memoria, permettendo di eseguire processi più grandi della RAM.
  - **Descrizione:** La memoria è divisa in pagine e frame. La tabella delle pagine serve per tradurre gli indirizzi virtuali in indirizzi fisici i.
  - **Translation Lookaside Buffer (TLB):** Cache per accelerare la traduzione degli indirizzi.
  - **Page Fault:** Si verifica quando una pagina non è presente in memoria solo quando necessario.
    - **Paging on Demand:** Le pagine vengono caricate in memoria solo quando necessarie.
  - **Preparation:** Le pagine vengono precaricate in memoria, anche se non immediatamente necessarie.
  - **Replacement Policy:** Per scegliere quale pagina sostituire (es. LRU).
  - **Placement Policy:** Come e dove allocare le pagine.
  - **Resident Set:** Insieme delle pagine di un processo presenti in memoria principale.
  - **Thrashing:** Situazione in cui il sistema passa la maggior parte del tempo a scambiare pagine, senza eseguire codice utile.
  - **Vantaggi:** Permette di eseguire processi più grandi della RAM, favorisce la multiprogrammazione.
  - **Svantaggi:** Overhead dovuto alla gestione delle tabelle delle pagine, possibilità di page fault e thrashing.
- **Segmentazione:** La memoria è divisa in segmenti di lunghezza variabile.
  - **Differenza:** A differenza della paginazione, i segmenti corrispondono a parti logiche del programma.
  - **Vantaggi:** Permette di gestire la memoria in modo più flessibile rispetto alla paginazione semplice.
  - **Svantaggi:** Frammentazione esterna.
- **Mutua Esclusione:** Mecanismo per garantire che solo un processo alla volta acceda a risorse condivise.
  - **Descrizione:** La mutua esclusione è fondamentale per evitare race condition e garantire la correttezza dei programmi concurrenti.
  - **Requisiti:** Non fare assunzioni sulla velocità dei processi, accesso esclusivo alle sezioni critiche.
  - **Algoritmi per la Mutua Esclusione:**
    - **Algoritmo di Dekker e Peterson:** Algoritmo software per la mutua esclusione.
    - **Semafori:** Variabili che possono essere modificate tramite operazioni atomiche (semWait e semSignal).
    - **Scambio Messaggi:** Primitiva per la comunicazione tra processi.
    - **Istruzioni Macchina Atomiche:** Istruzioni come exchange e compare\_and\_swap, che sono eseguite in modo atomico dal processore.
- **Deadlock:** Situazione in cui due o più processi sono bloccati in attesa l'uno dell'altro.
  - **Descrizione:** Il deadlock si verifica quando ci sono attesa circolare, hold-and-wait, mutua esclusione e no preemption.
  - **Metodi per la gestione del Deadlock:** Prevenzione, evitamento (algoritmo del banchiere) e rilevamento.
  - **Grafico dell'Allocazione delle Risorse:** Rappresenta lo stato di allocazione delle risorse ai processi.
  - **Input/Output (I/O):** Operazioni di trasferimento dati tra la memoria principale e i dispositivi esterni.
    - **Buffering:** Tecnica per migliorare l'efficienza dell'I/O.
    - **Direct Memory Access (DMA):** Permette ai dispositivi di I/O di accedere direttamente alla memoria, senza passare dal processore.
    - **Scheduling del Disco:** Algoritmi per ottimizzare l'ordine in cui le richieste di I/O vengono servite.
    - **Algoritmi:** SCAN, C-SCAN, FSCAN, N-steps SCAN, Minimum Service Time.
- **File System:** Componente del sistema operativo che gestisce la memorizzazione e l'accesso ai file.
  - **Metadata:** Informazioni sui file, come nome, dimensione, data di creazione.
  - **Allocazione dei File:**
    - **Contigui:** I file occupano blocchi contigui sul disco.
    - **Concatenati:** I blocchi dei file sono collegati tramite puntatori.
    - **Individuati:** Usa tabelle di indici per allocare i blocchi.
  - **Gestione dello Spazio Libero:** Metodi per tenere traccia dei blocchi liberi su disco.
  - **Directory:** Organizzano i file in modo gerarchico.
  - **i-node:** Struttura dati in Unix che contiene informazioni sui file.
  - **Hard Link:** Puntatori diretti a un i-node.
  - **Journaling:** Tecnica per garantire la consistenza del file system.
  - **FAT:** File Allocation Table, sistema di allocazione contigua.
  - **NTFS:** Nuovo file system di Windows.
- **Sicurezza:** Protezione da accessi non autorizzati e attacchi.
  - **Controllo degli Accessi:** Mecanismi per definire chi può accedere a quali risorse.
    - **Discrezionale:** Basato su permessi definiti dagli utenti.
    - **Basato su Ruoli:** Basato su permessi assegnati ai ruoli.
  - **Attacchi:** Dizionario.
  - **Confidencialità:** Protezione della riservatezza dei dati.
  - **Integrità:** Protezione da modifiche non autorizzate.
  - **Disponibilità:** Garanzia che il sistema sia sempre pronto a rispondere alle richieste.

Spero che questa lista sia utile. Se hai domande o necessiti ulteriori chiarimenti, fammelo sapere.

- **Preemption in sistemi monoprocesso:** Per avere un trattamento equo sui processi, non è sufficiente usare uno scheduler preemptive.
- **Transizioni di stato dei processi:** Nessuna delle opzioni fornite è corretta.
- **Allocazione indirizzata dei file:** Se usato con porzioni di dimensione variabile, i blocchi indice devono contenere anche la lunghezza di ogni porzione.
- **Requisito di riclazione:** Nessuna delle opzioni fornite è corretta.
- **Requisiti della mutua esclusione:** Non deve essere fatta alcuna assunzione sulla velocità di esecuzione dei processi coinvolti.
- **Dispatcher:** La predictability non è un criterio utente prestazionale.
- **Interrupt (o eccezioni):** Una volta gestito l'interrupt o l'eccezione, quando (e se) si torna ad eseguire il processo interrotto, l'esecuzione ripartirà sempre dall'istruzione successiva a quella dove è stato ricevuto l'interrupt o

l'eccezione.

- **Istruzioni macchina speciali per la gestione della concorrenza**: Nessuna delle altre opzioni è vera.
- **Processi**: Per la terminazione normale di un processo, è tipicamente prevista un'apposita system call.
- **Algoritmo software per la gestione della concorrenza**: Nell'algoritmo di Dekker, se la variabile turn è inizializzata ad 1, allora il processo 1 sarà sicuramente il primo ad entrare nella sezione critica nella prima iterazione.
- **I-node di Unix**: Per modificare una directory, un utente non deve aprire il file speciale corrispondente e poi modificarlo opportunamente.
- **Sistemi operativi**: Nel caso di un sistema operativo a kernel separato, la gestione dei process switch non è un processo.
- **Paginazione della memoria**: La differenza tra paginazione semplice e paginazione con memoria virtuale è che nella prima viene richiesto che tutte le pagine di un processo siano in memoria principale, affinché il processo stesso possa essere eseguito.
- **Allocazione concatenata del file**: Nessuna delle altre opzioni è vera.
- **Memoria virtuale con paginazione**: Il numero di bit di una entry di una tabella delle pagine di ultimo livello è uguale al numero di bit di controllo più il logaritmo (arrotondato all'intero superiore) del massimo numero di frame in memoria principale.
- **Deadlock**: Affinché ci sia un deadlock, sono necessarie le condizioni di attesa circolare, hold -and-wait, mutua esclusione e no preemption.
- **Modalità dei processi**: La modalità di un processo utente è sempre la modalità utente.
- **Memoria virtuale con paginazione**: Nessuna delle altre opzioni è corretta.
- **Concorrenza tra processi o thread**: Nessuna delle altre opzioni è vera.
- **Dispatcher**: Il throughput non è un criterio di sistema non prestazionale.
- **File system FAT**: Nessuna delle altre opzioni è vera.
- **Memoria virtuale con paginazione**: Il mapping associativo non permette al translation lookaside buffer di trovare una data pagina semplicemente sommando il numero della pagina con l'indirizzo di partenza del translation lookaside buffer stesso.
- **Processo**: Le informazioni sullo stato delle risorse non sono una parte che definisce un processo.
- **Risorse di un sistema operativo e deadlock**: Nel caso delle risorse consumabili, se c'è un deadlock allora è stata richiesta almeno una risorsa già detenuta da un altro processo.
- **Scheduler round-robin**: Per realizzare correttamente un process switch, il SO avrà necessità di usare le informazioni sul contesto contenute nel process control block.
- **Traduzione di un indirizzo virtuale in fisico**: L'hardware non deve cercare il numero di pagina nelle entries della tabella delle pagine del processo in esecuzione.
- **Operazioni su file**: La "connessione" non è un'operazione tipica sui file.
- **Thread**: Diversi thread di uno stesso processo non condividono lo stesso thread identifier.
- **Page cache**: Nell'algoritmo di sostituzione basato su frequenza a 3 segmenti della page cache, i contatori vengono sempre incrementati, tranne quando sono nel segmento vecchio.
- **Efficienza di un sistema operativo**: Un sistema operativo non deve dare accesso alle risorse in modo equo ed egualitario tra tutti i processi.
- **Metodi di gestione del deadlock**: L'unico metodo che non prevede mai la preemption delle risorse è quello che evita il deadlock.
- **Scheduler**: Lo scheduler non ha, tra i suoi obiettivi, quello di evitare il deadlock.
- **Scheduler per architetture multiprocessore**: Nessuna delle altre opzioni è vera.
- **Algoritmo del banchiere**: La matrice C - A può contenere elementi negativi, ma le matrici C ed A contengono elementi non negativi.
- **Processor utilization**: Il processor utilization è definito come il rapporto tra il tempo in cui il processore viene usato ed il tempo totale del sistema.
- **I-nodi di Unix**: I puntatori a tripla indirezione di un i-node vengono usati solo se la dimensione del file lo richiede.
- **Modello dei processi a 5 stati**: Nessuna delle altre opzioni è vera.
- **Dischi magnetici a testina mobile**: Il tempo di accesso ad un disco magnetico a testina mobile non tiene conto del tempo che occorre per effettuare effettivamente il trasferimento di dati.
- **Algoritmi di scheduling**: L'exponential averaging non è applicabile dal solo scheduler Short Process Next.
- **Memoria virtuale con paginazione a segmentazione**: Sia la tabella dei segmenti che quella delle pagine di un processo non contengono, in ciascuna entry, un bit per indicare se la pagina o il segmento sono stati modificati.
- **Memoria virtuale con paginazione**: Per avere un overhead accettabile, occorre demandare la traduzione degli indirizzi all'hardware, mentre al software resta da gestire prelievo, posizionamento e sostituzione delle pagine.
- **Sistemi batch e sistemi time sharing**: I sistemi time-sharing non puntavano a minimizzare l'uso del processore.
- **Algoritmi di scheduling**: Non ci sono sufficienti informazioni per determinare come si comporterebbe l'algoritmo di scheduling SRT.
- **Algoritmi di scheduling**: Il tempo medio di attesa non è tra 10 ed 11 ms.
- **Mutua esclusione**: La soluzione fornita non implementa correttamente la mutua esclusione.
- **Memoria virtuale con paginazione**: Aumentare la dimensione delle pagine ha effetti positivi sulla frammentazione interna.
- **Concorrenza tra processi o thread**: Se un processo utente può disabilitare le interruzioni tramite un'istruzione macchina dedicata, allora può far diminuire l'uso utile del processore.
- **Kernel**: Il kernel non è costituito da vari moduli che non possono essere caricati nel sistema operativo in esecuzione.
- **Esecuzione istruzioni**: La CPU può eseguire un'istruzione soltanto quando gli operandi si trovano nei registri della CPU.
- **Process Control Block (PCB)**: Il PCB è una struttura dati mantenuta dal sistema operativo che contiene tutte le informazioni necessarie all'esecuzione, sospensione e ripresa dell'esecuzione di un processo.
- **Chiamate di sistema**: Il sistema operativo deve effettuare soltanto un mode switch per eseguire la funzione richiesta.
- **Dispatcher**: Il dispatcher si occupa di scambiare i processi in esecuzione sulla CPU (process switch).
- **Buffering dell'I/O**: Il sistema operativo non copia quando più' opportuno il contenuto della scrittura dalla memoria del processo direttamente alla memoria del dispositivo di I/O.
- **Algoritmo di scheduling C-SCAN**: L'algoritmo C-SCAN non favorisce le richieste ai bordi rispetto a SCAN.
- **File Management System**: Tutte le opzioni sono requisiuti per un File Management System.
- **Sezione critica**: Una sezione critica è un segmento di programma in cui si accede a risorse condivise.
- **Algoritmo di Dekker**: L'algoritmo di Dekker richiede supporto dal SO.
- **Translation lookaside buffer**: Tutte le opzioni sono vere riguardo al Translation lookaside buffer per la gestione della memoria.
- **Long-term scheduler**: Il long-term scheduler si occupa della decisione di quali processi debbano essere ammessi all'esecuzione nel sistema.
- **Modello dei processi a 5 stati**: Un processo può essere spostato dallo stato blocked solo allo stato ready.
- **Preallocazione vs allocazione dinamica**: Nessuna delle opzioni è corretta riguardo alla preallocazione rispetto all'allocazione dinamica dello spazio per i file.
- **File system NTFS**: NTFS può, ove possibile, includere direttamente i dati di un file nella master file table.
- **Rilocazione degli indirizzi di memoria**: Nei sistemi con hardware dedicato per la rilocazione, il base register (registro base) viene impostato una sola volta, quando il programma viene caricato in memoria per la prima volta.
- **Thrashing**: Tutte le opzioni sono vere riguardo il concetto di Thrashing.
- **Partizionamento fisso della memoria**: Il partizionamento fisso della memoria impone un numero massimo di processi che possono essere in memoria principale.
- **Attacco dizionario**: La velocità di computazione in real time degli hash non è un vantaggio dell'attacco dizionario.
- **Scheduler a breve e lungo termine**: La distinzione principale tra lo scheduler a breve e a lungo termine è la frequenza di esecuzione.
- **Multiprogrammazione**: La riduzione dell'overhead del sistema operativo non è un vantaggio della multiprogrammazione.
- **Program Counter**: Il program counter fornisce l'indirizzo della prossima istruzione da eseguire.
- **Prevenzione del deadlock**: Non chiedere mai una risorsa dopo averne rilasciate altre non è un valido schema di prevenzione del deadlock.
- **Algoritmo di Dekker**: L'algoritmo di Dekker non usa busy waiting.
- **Tabelle di controllo del sistema operativo**: La tabella dei processi sospesi non è una tabella di controllo del sistema operativo.
- **Interrupt annidati**: La cpu sospende l'esecuzione del codice corrente, ed avvia l'handler del nuovo interrupt ricevuto.
- **Throughput**: Il throughput non è la capacità del sistema operativo.
- **File Management System**: Tutte le opzioni elencate sono obiettivi per un File Management System.
- **Allocazione dei file indicizzata**: La tabella di allocazione contiene soltanto l'indirizzo di un blocco, e questo blocco contiene le entry delle porzioni di file allocate oppure l'indirizzo di altri blocchi usati a loro volta per indicizzare le porzioni di file allocate.
- **Algoritmi di scheduling del disco**: N-step-SCAN è una generalizzazione di FSCAN che è fair e può avere prestazioni molto simili a quelle di SCAN.
- **Scheduling**: Il file scheduling non è un tipo di scheduling dei sistemi operativi.
- **Paginazione SEMPLICE**: Nessuna delle altre opzioni è corretta per la paginazione SEMPLICE.
- **Journaling logico**: Il sistema operativo usa il journal solo per copiare i metadati prima di aggiornare le strutture del file system, ma non lo utilizza per i dati.
- **Linux e gestione dei file**: Nessuna delle altre opzioni è corretta.
- **Sistemi Unix**: Gli hard link sono puntatori diretti al descrittore di un file (inode).

Spero che questa lista ti sia utile.

# Obiettivi e problemi dell'OS

giovedì 23 gennaio 2025 12:30

- OS
- OS
  - Convenienza
  - Efficienza
- Processi
- 
- Scheduling
- Tipi di scheduling
  - Long-Term: decide i programmi da ammettere nel sistema per eseguirli
  - Medium-Term: parte dello swapping dei processi tra mem. secondaria e RAM
  - Short-Term: chiamato dispatcher
    - eseguito più frequentemente
    - alloca tempo di esec. su un processore
    - minimizzare il tempo di risposta medio (response time)
    - massimizzare il num. di utenti
    - massimizzare il num. di scadenze rispettate (se ci sono)
    - massimizzare num. processi completati in un lasso di tempo (throughput)
    - massimizzare l'uso della CPU
    - massimizzare l'uso delle risorse di sistema
    - equità e niente starvation
- Criteri per lo Short-Term Scheduling
  - Criteri Utente:
    - Prestazionali
      - turnaround time (tempo tra la creazione di un processo e il suo completamento (comprende i tempi di attesa)
      - response time (tempo tra la sottomissione di una richiesta e l'inizio della risposta)
      - deadline
    - Non prestazionali
      - predicitività
  - Criteri di Sistema:
    - Prestazionali
      - throughput
      - uso del processore
    - Non prestazionali
      - Fairness
      - gestione delle priorità
      - bilanciamento dell'uso delle risorse

- Algoritmi di scheduling:

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s - e]	$\max\left(\frac{w + s}{s}\right)$	(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Throughput	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

- Scheduling
  - ottimizzare tempo di risposta, throughput, efficienza processore
  - distribuire tempo di esecuzione equamente tra processi
  - gestire priorità processi quando necessario
  - evitare starvation
  - overhead basso
  - massimizzare num. utenti

- Memoria

- Requisiti gestione mem.:
  - Rilocazione
  - Protezione
  - Condivisione
  - Organizzazione logica
  - Organizzazione fisica
- Partizionamento:

scopo X  
starvationX  
overhead X  
requisiti

- Fisso uniforme: uso inefficiente della mem, framm. interna
- Fisso variabile: mitiga i problemi del fisso uniforme
- Problemi fisso: num. max di processi corrispondente al num. di partizioni, uso inefficiente di mem. se ci sono molti processi piccoli
- Dinamico: frammentazione esterna (risolvibile con compattazione ma elevato overhead)
- Algoritmi assegnazione:
  - best-fit: risultati peggiori, lascia frammenti molto piccoli, compattazione frequente
  - first-fit: molto veloce, riempie solo la prima parte di mem.
  - next-fit: assegna spesso alla fine della mem.
  - buddy system: compromesso tra partizionamento fisso e dinamico, dimezza lo spazio per ogni esponente di 2(es.  $1024 \Rightarrow 512+512 \Rightarrow 512+128+128+256$ ) fino a trovare una porzione sufficiente (una delle 2 porz. è usata dal processo, l'altra è libera)
- Segmentazione:
  - Permette al programmatore di vedere la mem. come un insieme di spazi (segmenti) di indirizzi
  - Dim. degli indirizzi può essere variabile e dinamica
  - Permette di modificare e ricompilare i programmi in modo indipendente
  - Permette di condividere i dati
  - Permette di proteggere i dati
- Memoria virtuale:
  - multiprogrammazione elevata
  - più processi in mem.
- **Gestione IO**
- Scheduling:
  - FIFO: equo, se ci sono molti processi le prestazioni sono simili al random
  - LIFO: possibile starvation
  - SSTF: minimizza il movimento del braccio (dalla pos. attuale), possibile starvation (arrivano richieste vicine)
  - SCAN: niente starvation, poco equo poiché favorisce la richiesta ai bordi e le richieste appena arrivate
  - C-SCAN: più equo di SCAN, non favorisce la richiesta ai bordi, minore variabilità del servizio
  - FSCAN:
  - N-step-SCAN: non favorisce le richieste appena arrivate
- Buffer:
  - smussa i picchi di richieste IO, ma se la domanda è molta il buffer si riempie e il vantaggio si perde
  - Utile quando ci sono molti IO da servire, migliora efficienza dei processi e dell'OS
  - Introduce overhead per la copia intermedia in user buffer, si può risolvere facendo un trasferimento da un kernel buffer ad un altro (Zero Copy)
- **File System**
- File Management System
  - garantire validità dei dati nei file
  - ottimizzare throughput e tempo di risposta
  - minimizzare dati persi o distrutti
  - supporto IO per più utenti in contemporanea
- Allocazione file
  - Contigua:
    - Pro: gestione semplice, **una sola entry** per file
    - Contro: frammentazione esterna, difficoltà nel far crescere i file oltre lo spazio allocato
  - Concatenata:
    - Pro: buona per accesso sequenziale, **una sola entry**
    - Contro: lenta per accesso diretto (devi scorrere tutta la lista)
  - Indicizzata
    - Pro: risolve i problemi delle altre due tecniche, **una sola entry**
      - Lunghezza fissa: niente frammentazione esterna
      - Lunghezza var.: migliora la località
- Gestione Spazio Libero
  - Vettore bit: **minimizza** lo spazio richiesto alla tab. di allocazione ma se il disco è quasi pieno, la ricerca di uno spazio libero richiede molto tempo
  - Porzioni concatenate: senza overhead di spazio ma:
    - se c'è frammentazione, la lista si allunga
    - bisogna leggere un blocco libero per sapere quale è il prossimo
    - è lungo cancellare file molto frammentati
  - Indicizzazione:
    - Lista blocchi liberi:
- **Concorrenza**
- Semafori:
  - Deboli: non si evita starvation
  - Forti: se usati bene, si può evitare la starvation
- Problema produttore/consumatore: risolvibile con buffer circolari, con semafori usati bene o con messaggi
- Requisiti mutua esclusione:
  - Niente deadlock o starvation
  - Solo un processo alla volta nella sezione critica (codice che accede a risorse condivise) per una risorsa
  - Nessuna assunzione sullo scheduling dei processi
  - Se nessun altro processo usa la risorsa, un processo deve entrare **subito** nella sezione critica
  - Un processo che entra nella sua sezione critica ne deve uscire prima o poi uscire
- Metodi mutua esclusione:
  - Disabilitazione interrupt:
    - funziona solo su sistemi con un solo core
    - cala la multiprogrammazione e quindi l'uso del processore

- Istruzioni macchina speciali: l'HW garantisce che solo un processo per volta possa chiamare tali istr.
  - Vantaggi:
    - applicabili a qualsiasi numero di processi (sia a sistemi multi-core che con un solo core)
    - facili da verificare
    - usati per gestire sezioni critiche multiple
  - Svantaggi:
    - basate sul busy-waiting: risolvibile con una coda di attesa
    - possibile starvation
    - possibile deadlock se abbinata la priorità
- Algoritmi di Dekker e Peterson:
  - vale solo per 2 processi; estensione a N processi possibile ma non banale
  - niente starvation
  - niente deadlock, tranne con priorità fisse
  - usa busy-waiting
  - non richiede supporto OS
- **Deadlock**
- 
- **Sicurezza**
-

## RISPOSTE A CUI HO SBAGLIATO O INSICURO

15) Quale delle seguenti affermazioni sull'algoritmo per il rilevamento del deadlock visto a lezione è vera?  
 1. Richiede in input, per ogni processo  $p$  e per ogni risorsa  $r$ , il numero massimo di istanze di  $r$  che  $p$  chiederà nel corso della sua esecuzione  
 2. Se al passo 3 viene trovato un processo non marcato che soddisfa la condizione  $Qik \leq w_{ik}$ , allora c'è un deadlock  
 3X. I processi marcati sono quelli che non sono coinvolti in un deadlock  
 4. Nessuna delle altre opzioni è vera

30) Quale delle seguenti affermazioni sugli indirizzi di memoria principale è vera?

1. Un indirizzo fisico fa sempre riferimento alla memoria secondaria
2. Per rispettare il requisito di rilocazione, occorre trasformare indirizzi fisici in logici
3. Gli indirizzi relativi sono usati nella paginazione
- 4X. Nessuna delle altre opzioni è corretta

17) Quale delle seguenti affermazioni sulla memoria virtuale con paginazione è vera?

- 1X. Il difetto principale del prepaging è che potrebbe portare in memoria pagine cui poi non si fa riferimento
2. Placement policy e replacement policy sono sinonimi ed indicano lo stesso insieme di metodologie
3. Nessuna delle altre opzioni è corretta
4. Il difetto principale del paging on demand è che causa molti page fault dopo alcuni secondi di esecuzione

150) Quale delle seguenti affermazioni sulla concorrenza tra processi o thread è vera?

1. La disabilitazione delle interruzioni non impedisce la creazione di nuove interruzioni, ma solo la loro gestione
2. Se un processo utente può disabilitare le interruzioni tramite un'istruzione macchina dedicata, allora può far diminuire l'uso utile del processore
3. La disabilitazione delle interruzioni non funziona ai fini della concorrenza (gestione sezioni critiche) su sistemi con più processori o più core
- 4X. Tutte le opzioni sono vere

52) Quale delle seguenti affermazioni, riguardanti il joint progress diagram di 2 processi, è vera?

1. Può essere usato per determinare quando uno dei due processi sperimenta un page fault
2. Può essere usato per visualizzare le possibilità di deadlock, ma solo se i processi richiedono al massimo 2 risorse
- 3X. Nessuna delle altre opzioni è vera
4. Può essere usato per determinare quando uno dei due processi manda un segnale all'altro

19) Quale delle seguenti affermazioni sulla memoria virtuale con paginazione è vera?

1. Il principio di località afferma che poche pagine saranno sempre sufficienti per eseguire ogni processo senza thrashing
2. Il thrashing si verifica quando l'overhead dovuto alla gestione della paginazione è molto basso
- 3X. Nessuna delle altre opzioni è corretta
4. La paginazione con memoria virtuale funziona bene nonostante il principio di località

28) Assumendo un sistema monoprocesso, quale delle seguenti affermazioni sulla preemption è vera?

1. Se uno scheduler è non-preemptive, permette sempre ai suoi processi di essere eseguiti sul processore, senza interruzioni, fino al loro completamento
- 2X. Se uno scheduler è non-preemptive, è possibile che un processo monopolizzi il processore, anche in presenza di altri processi ready
3. Se uno scheduler è preemptive, non è possibile che un processo monopolizzi il processore, anche in presenza di altri processi ready
4. Per avere un trattamento equo sui processi, è sufficiente usare uno scheduler preemptive

20) Quale delle seguenti affermazioni sullo scambio messaggi per la gestione della concorrenza è vera?

- 1X. Nessuna delle altre opzioni è vera
2. L'implementazione delle primitive per lo scambio messaggi non è garantita atomica dal sistema operativo
3. Se un processo chiama receive, finché il messaggio non viene ricevuto, tutti gli altri processi che proveranno a chiamare receive verranno bloccati
4. Per garantire la mutua esclusione, occorre ricorrere al busy waiting se sia invio che ricezione sono non bloccanti

116) Quale delle seguenti affermazioni sul long-term scheduler è vera:

- 1X. Si occupa della decisione di quali processi debbano essere ammessi all'esecuzione nel sistema
2. Si occupa dell'organizzazione di lungo termine dell'ordine di esecuzione dei processi nella CPU
3. Si occupa dell'implementazione della funzione di swapping dei processi alla memoria secondaria
4. Si occupa della transizione dei processi tra gli stati running ed exit

66) Quale delle seguenti affermazioni è falsa?

- 1X. Nel caso di un sistema operativo a kernel separato, la gestione del process switch è a sua volta un processo
2. Nel caso di un sistema operativo in cui le funzioni del sistema operativo vengono eseguite all'interno dei processi utente, non c'è bisogno di un process switch per eseguire una funzionalità del sistema operativo
3. Nel caso di un sistema operativo in cui le funzioni del sistema operativo vengono eseguite all'interno dei processi utente, se un processo effettua una syscall e poi può continuare ad essere eseguito, non avviene alcun process switch
4. Nel caso di un sistema operativo in cui le funzioni del sistema operativo vengono eseguite come processi separati, c'è sempre bisogno di un process switch per eseguire una funzionalità del sistema operativo

65) Quale delle seguenti affermazioni sugli i-node di Unix è falsa?

1. Ogni directory è identificata da un i-node
- 2X. Per modificare una directory, un utente deve aprire il file speciale corrispondente e poi modificarlo opportunamente
3. Ogni directory è un file speciale, organizzato come una lista di entry, ciascuna delle quali contiene il nome di un file ed il relativo i-node number
4. Ogni directory può contenere molti i-node

99) Quale delle seguenti affermazioni sulla memoria virtuale con paginazione è vera?

- 1X. Per avere un overhead accettabile, occorre demandare la traduzione degli indirizzi all'hardware, mentre al software resta da gestire prelievo, posizionamento e sostituzione delle pagine
2. Per avere un overhead accettabile, occorre demandare la traduzione degli indirizzi e la politica di sostituzione delle pagine all'hardware, mentre al software resta da gestire prelievo e posizionamento delle pagine
3. Per avere un overhead accettabile, occorre demandare all'hardware la traduzione degli indirizzi ed il prelievo, il posizionamento e la sostituzione delle pagine
4. Per avere un overhead accettabile, occorre demandare al software anche la traduzione degli indirizzi

62) Quale delle seguenti affermazioni sulle istruzioni macchina speciali per la gestione della concorrenza è vera?

1. Sono basate sul busy-waiting, ovvero sul fatto che un processo si mette autonomamente in stato blocked
- 2X. Nessuna delle altre opzioni è vera
3. Non riescono ad evitare il manifester del deadlock, a meno che non sia presente un sistema a priorità
4. Come per la disabilitazione delle interruzioni, non funzionano per architetture con più processori o core

138) Nei sistemi operativi che usano paginazione SEMPLICE per la gestione della memoria

1. ai processi devono essere allocati frame di memoria necessariamente contigui per poter consentire l'esecuzione del processo
2. il sistema operativo deve utilizzare la tabella delle pagine per tradurre gli indirizzi. Qualora una pagina non sia presente in memoria principale, il sistema la deve caricare dinamicamente per consentire il proseguimento dell'esecuzione di un processo
3. non c'è necessità di traduzione degli indirizzi, in quanto tutte le pagine di un processo sono sempre caricate in un frame nella memoria principale
- 4X. nessuna delle altre opzioni è corretta

16) Assumendo un sistema monoprocesso, quale delle seguenti affermazioni sul long-term scheduler è falsa?

- 1X. Viene chiamato in causa esclusivamente quando viene creato un nuovo processo
2. Avendo le necessarie informazioni, una tipica strategia è mantenere una giusta proporzione, stabilita a priori, tra processi I/O-bound e CPU-bound
3. Avendo le necessarie informazioni, una tipica strategia è ammettere in memoria principale i processi che richiedono dispositivi di I/O diversi da quelli richiesti dai processi già attivi
4. Decide quali processi, tra quelli appena creati, possono essere ammessi in memoria principale per l'esecuzione

15) Algoritmo di rilevamento del deadlock

Risposta corretta è 3 (I processi marcati sono quelli che non sono coinvolti in un deadlock)  
 L'algoritmo segna i processi che possono completare la loro esecuzione e liberare risorse. I processi non marcati al termine dell'algoritmo sono coinvolti in un deadlock.

Perché non le altre

- Opzione 1) L'algoritmo richiede il numero massimo di risorse che ogni processo può chiedere, ma non necessariamente per ogni istanza specifica.
- Opzione 2) Se un processo non soddisfa la condizione al passo 3, non implica automaticamente un deadlock.
- Opzione 4) Non è vera perché l'opzione corretta esiste.

30) Indirizzi di memoria principale

Risposta corretta è 4 (Nessuna delle altre opzioni è corretta)

Tutte le altre affermazioni sono errate rispetto alla gestione degli indirizzi di memoria principale.

Perché non le altre

- Opzione 1) Un indirizzo fisico non fa riferimento alla memoria secondaria, ma alla memoria principale.
- Opzione 2) Gli indirizzi logici vengono trasformati in fisici, non al contrario.
- Opzione 3) Gli indirizzi relativi non sono usati nella paginazione.

17) Memoria virtuale con paginazione

Risposta corretta è 1 (Il difetto principale del prepaging è che potrebbe portare in memoria pagine cui poi non si fa riferimento entro il prepaging cerca di anticipare le richieste, ma a volte carica inutilmente pagine non necessarie).

Perché non le altre

- Opzione 2) Placement e replacement policy sono concetti diversi.
- Opzione 3) Esistono opzioni corrette, quindi questa è errata.
- Opzione 4) Il paging on-demand causa molti page fault inizialmente, ma non necessariamente dopo alcuni secondi.

150) Concorrenza tra processi o thread

Risposta corretta è 4 (Tutte le opzioni sono vere)

Tutte le affermazioni descrivono aspetti corretti della gestione della concorrenza.

Perché non le altre Non applicabile, poiché l'opzione scelta include tutte le risposte.

52) Joint Progress Diagram

Risposta corretta è 3 (Nessuna delle altre opzioni è vera)

Nessuna delle altre descrizioni è corretta per l'uso del diagramma.

Perché non le altre

- Opzione 1 e 4) Il diagramma non è utilizzato per determinare page fault o segnali.
- Opzione 2) Il deadlock non dipende solo dal numero di risorse richieste.

19) Memoria virtuale con paginazione

Risposta corretta è 3 (Nessuna delle altre opzioni è corretta)

Tutte le altre affermazioni sono errate.

Perché non le altre

- Opzione 1) Il principio di località non garantisce l'assenza di thrashing.
- Opzione 2) Il thrashing è causato da un overhead alto.
- Opzione 4) La paginazione funziona bene proprio grazie al principio di località.

28) Preemption

Risposta corretta è 2 (Se uno scheduler è non-preemptive, è possibile che un processo monopolizzi il processore)

Uno scheduler non-preemptive consente a un processo di mantenere il controllo fino al completamento.

Perché non le altre

- Opzione 1) Non tutti i processi ottengono sempre risorse senza interruzioni.
- Opzione 3) Anche con uno scheduler preemptive, un processo può monopolizzare il processore in certe condizioni.
- Opzione 4) L'equità non dipende esclusivamente dalla preemption.

20. Quale delle seguenti affermazioni sullo scambio messaggi per la gestione della concorrenza è vera?

Risposta corretta è 1. Nessuna delle altre opzioni è vera

Motivo) Nessuna delle affermazioni rimanenti è effettivamente corretta.

- Opzione 2) L'implementazione delle primitive di scambio messaggi è garantita atomica dal sistema operativo, quindi l'affermazione è errata.
- Opzione 3) Se un processo chiama 'receive', non blocca necessariamente gli altri processi che chiamano 'receive'; questo dipende dall'implementazione.
- Opzione 4) Non è sempre necessario il busy waiting per garantire la mutua esclusione.

116. Quale delle seguenti affermazioni sul long-term scheduler è vera?

Risposta corretta è 1. Si occupa della decisione di quali processi debbano essere ammessi all'esecuzione nel sistema

Motivo) Il long-term scheduler decide quali processi passano dalla coda di input allo stato di ready per l'esecuzione.

- Opzione 2) L'organizzazione a lungo termine dei processi è tipica dello short-term scheduler.
- Opzione 3) Lo swapping è gestito dalla memoria virtuale, non dal long-term scheduler.
- Opzione 4) La transizione tra gli stati 'running' ed 'exit' è gestita dal sistema operativo.

66. Quale delle seguenti affermazioni è falsa?

Risposta corretta è 1. Nel caso di un sistema operativo a kernel separato, la gestione del process switch è a sua volta un processo

Motivo) La gestione del process switch non è un processo separato, ma una funzione del kernel.

- Le altre affermazioni descrivono correttamente comportamenti dei sistemi operativi basati su diverse architetture.

65. Quale delle seguenti affermazioni sugli i-node di Unix è falsa?

Risposta corretta è 2. Per modificare una directory, un utente deve aprire il file speciale corrispondente e poi modificarlo opportunamente

Motivo) Le directory in Unix sono modificate indirettamente attraverso comandi specifici come 'mkdir' o 'rmdir'.

- Le altre opzioni descrivono correttamente il comportamento degli i-node.

99. Quale delle seguenti affermazioni sulla memoria virtuale con paginazione è vera?

Risposta corretta è 1. Per avere un overhead accettabile, occorre demandare la traduzione degli indirizzi all'hardware, mentre al software resta da gestire prelievo, posizionamento e sostituzione delle pagine

Motivo) Il sistema operativo si occupa delle politiche di gestione della memoria, ma l'hardware gestisce la traduzione degli indirizzi per efficienza.

- Le altre opzioni attribuiscono all'hardware o al software compiti non corretti.

62. Quale delle seguenti affermazioni sulle istruzioni macchina speciali per la gestione della concorrenza è vera?

Risposta corretta è 2. Nessuna delle altre opzioni è vera

Motivo) Le altre affermazioni contengono errori

- Opzione 1) Le istruzioni speciali non richiedono che un processo si metta autonomamente in stato 'blocked'.
- Opzione 3) La gestione del deadlock non dipende solo da un sistema a priorità.
- Opzione 4) Le istruzioni possono funzionare anche in architetture multiprocessore.

138. Nei sistemi operativi che usano paginazione semplice per la gestione della memoria

Risposta corretta è 4. Nessuna delle altre opzioni è corretta

Motivo)

- Opzione 1) I frame non devono essere contigui.
- Opzione 2) Non è sempre necessario caricare dinamicamente una pagina per continuare l'esecuzione.
- Opzione 3) La traduzione degli indirizzi è fondamentale per la paginazione.

16) Assumendo un sistema monoprocesso, quale delle seguenti affermazioni sul long-term scheduler è falsa?

Risposta corretta è 1. Viene chiamato in causa esclusivamente quando viene creato un nuovo processo

Motivo) Il long-term scheduler non viene attivato solo quando un processo è creato, ma anche in base alla necessità di bilanciare i carichi d el sistema.

Perché le altre non sono errate)

- 2) È vero che il long-term scheduler cerca un bilanciamento tra processi I/O-bound e CPU-bound.
- 3) È vero che può considerare i dispositivi di I/O usati dai processi.
- 4) Decide quali processi possono entrare in memoria principale.

93) Quale delle seguenti affermazioni sugli i-node di Unix è vera?

Risposta corretta è 2. I puntatori a tripla indirezione di un i-node vengono usati solo se la dimensione del file lo richiede

Motivo) I puntatori a tripla indirezione servono per file molto grandi.

Perché le altre non sono corrette)

- 1) Gli i-node possono essere memorizzati anche nella cache di memoria.
- 3) Esistono opzioni vere.
- 4) È falso che ogni file ha un solo numero di i-node.

117) Nel modello dei processi a 5 stati, quale affermazione è falsa?

Risposta corretta è 1. Un processo può essere spostato allo stato suspended dallo stato blocked e ready

Motivo) Un processo può essere sospeso solo dallo stato blocked.

Perché le altre non sono errate)

- 2, 3, 4) Tutte descrivono transizioni valide.

2. Avendo le necessarie informazioni, una tipica strategia è mantenere una giusta proporzione, stabilità a priori, tra processi I/O-bound e CPU-bound
3. Avendo le necessarie informazioni, una tipica strategia è ammettere in memoria principale i processi che richiedono dispositivi di I/O diversi da quelli richiesti dai processi già attivi
4. Decidere quali processi, tra quelli appena creati, possono essere ammessi in memoria principale per l'esecuzione

93) Quale delle seguenti affermazioni sugli i-node di Unix è vera?

1. Per ogni file-system su disco organizzato con i-node, tutti gli i-node di tutti i file su tale file-system sono memorizzati esclusivamente su disco
- 2X. I puntatori a tripla indirezione di un i-node vengono usati solo se la dimensione del file lo richiede
3. Nessuna delle altre opzioni è vera
4. Ad ogni file effettivamente memorizzato su disco può essere associato un solo numero di i-node

117) Nel modello dei processi a 5 stati, quale affermazione è falsa?

- 1X. Un processo può essere spostato allo stato suspended dallo stato blocked e ready
2. Un processo può essere spostato dallo stato running allo stato ready o exit
3. Un processo può essere spostato dallo stato blocked solo allo stato ready
4. Un processo può essere spostato dallo stato ready allo stato running, blocked o exit (da ready a blocked?)

134) Quale dei seguenti sono obiettivi per un file Management System?

- 1X. Tutte le opzioni elencate
2. Fornire supporto per l'I/O da più utenti in contemporanea
3. Minimizzare i dati persi o distrutti
4. Fornire un insieme di interfacce standard per i processi utente

95) Quale delle seguenti affermazioni sui dischi magnetici a testina mobile è vera?

1. Per selezionare un settore su una traccia di un disco magnetico a testina mobile, bisogna prima far ruotare il disco fino ad arrivare alla giusta traccia, e poi posizionare la testina sul giusto settore
2. Una traccia di un disco è l'area compresa tra 2 raggi del disco stesso
- 3X. Il tempo di accesso ad un disco magnetico a testina mobile tiene conto sia del tempo che occorre per posizionare la testina che del tempo che occorre per far ruotare il disco, ma non del tempo che occorre per effettuare effettivamente il trasferimento di dati
4. Nessuna delle altre opzioni è corretta

110) In un sistema operativo con I/O buffering, quando c'è una scrittura su dispositivo di I/O quale delle seguenti affermazioni è vera?

1. Il sistema operativo copia immediatamente il contenuto della scrittura dalla memoria del processo direttamente alla memoria del dispositivo di I/O
- 2X. Il sistema operativo copia immediatamente il contenuto della scrittura dalla memoria utente alla memoria del sistema operativo, e dalla memoria del sistema operativo alla memoria del dispositivo di I/O quando più' opportuno
3. Il sistema operativo copia quando più' opportuno il contenuto della scrittura dalla memoria del processo direttamente alla memoria del dispositivo di I/O
4. Nessuna delle altre opzioni è corretta

38) Quale delle seguenti affermazioni sul controllo di accesso è vera?

1. Nel controllo di accesso basato su ruoli, ad ogni ruolo è assegnato un utente
2. Nel controllo di accesso basato su ruoli, prima di stabilire se un'operazione è lecita, è necessario consultare una tabella soggetti-ruoli-oggetti
- 3X. Nel controllo di accesso discrezionale, prima di stabilire se un'operazione è lecita, è necessario consultare una tabella soggetti-oggetti
4. Nessuna delle altre opzioni è vera

44) Quale delle seguenti affermazioni sui semafori per la gestione della concorrenza è falsa?

1. Semafori generali e semafori binari hanno lo stesso potere computazionale (ovvero, permettono di risolvere gli stessi problemi)
2. Le primitive sui semafori sono in grado di mettere un processo in blocked, senza usare, a tal proposito, il busy-waiting
- 3X. Per implementare le primitive sui semafori, servono un contatore ed una coda, che saranno condivisi da tutti i semafori usati
4. L'implementazione delle primitive sui semafori è garantita atomica dal sistema operativo

100) Quale delle seguenti affermazioni è vera sulla memoria virtuale con paginazione a segmentazione?

- 1X. Sia la tabella dei segmenti che quella delle pagine di un processo contengono, in ciascuna entry, un bit per indicare se la pagina o il segmento sono stati modificati
2. Un indirizzo virtuale contiene anche un bit per indicare se la pagina corrispondente è o no in memoria principale
3. La tabella delle pagine di un processo contiene una pagina speciale dove è memorizzato il process control block del processo stesso
4. Ogni entry di una tabella delle pagine contiene un numero di pagina ed un offset

71) Quale delle seguenti affermazioni è vera?

1. La modalità di un processo utente è sempre la modalità di sistema
2. La modalità di un processo utente è inizialmente la modalità utente; può diventare modalità sistema nel momento in cui va in esecuzione il dispatcher
- 3X. Nessuna delle altre opzioni è vera
4. La modalità di un processo utente è sempre la modalità utente

112) Quale dei seguenti sono requisiti per un File Management System?

1. Ogni utente dev'essere in grado di creare, cancellare, leggere, scrivere e modificare un file
2. Ogni utente deve poter accedere, in modo controllato, ai file di un altro utente
3. Ogni utente deve poter mantenere una copia di backup dei propri file
- 4X. Tutte le opzioni sono requisiti

98) Assumendo un sistema monoprocesso, quale delle seguenti affermazioni sugli algoritmi di scheduling è vera?

1. L'exponential averaging permette di stimare la dimensione dell'immagine di un processo, a partire dalle precedenti immagini di quello stesso processo
- 2X. La funzione di decisione dello scheduler Highest Response Ratio Next considera tanto il tempo di esecuzione stimato quanto il tempo trascorso in attesa
3. L'exponential averaging è una tecnica applicabile dal solo scheduler Short Process Next
4. La funzione di decisione dello scheduler Shortest Remaining Time considera tanto il tempo di esecuzione richiesto quanto il tempo trascorso in attesa

122) Quale delle seguenti affermazioni riguardo la rilocazione degli indirizzi di memoria è vera?

1. Nei sistemi con hardware dedicato per la rilocazione, il base register (registro base) viene impostato una sola volta, quando il programma viene caricato in memoria per la prima volta
2. In un sistema con rilocazione a run time, i sistemi di protezione che verificano che un processo non vada ad accedere alla memoria di un altro processo possono essere eseguiti a tempo di compilazione, prima di eseguire il programma
3. In un sistema a rilocazione con indirizzi logici, non è necessario avere hardware dedicato per effettuare la rilocazione
- 4X. In un sistema a rilocazione con indirizzi assoluti, se si conosce l'indirizzo di memoria dove verrà caricato il programma, il compilatore può inserire direttamente gli indirizzi di memoria corretti nel codice oggetto (programma compilato)

41) Quale delle seguenti affermazioni sulla memoria virtuale con paginazione è vera?

- 1X. Il difetto principale del prepaging è che potrebbe portare in memoria pagine cui poi non si fa riferimento
2. Nessuna delle altre opzioni è corretta
3. Il difetto principale del paging on demand è che, dopo una prima fase di assestamento, causa molti page fault
4. Placement policy e replacement policy sono sinonimi ed indicano lo stesso insieme di metodologie

5) Quale delle seguenti affermazioni sulla memoria virtuale con paginazione è falsa?

1. Quando un indirizzo non viene trovato nel translation lookaside buffer, è necessario consultare la normale tabella delle pagine
2. Il translation lookaside buffer è una particolare cache, ma non è completamente trasparente al sistema operativo
- 3X. Il translation lookaside buffer permette di accedere direttamente al contenuto degli indirizzi di memoria virtuali usati più di recente (Contenuto?)
4. In assenza di translation lookaside buffer, l'accesso ad un indirizzo virtuale può richiedere almeno 2 accessi in memoria

144) Quale dei seguenti elementi non è contenuto nel Process Control Block (PCB)?

1. Stack del processo
2. Radice del programma

117) Nel modello dei processi a 5 stati, quale affermazione è falsa?

- Risposta corretta è 1. Un processo può essere spostato allo stato suspended dallo stato blocked e ready  
 Motivo) Un processo può essere sospeso solo dallo stato blocked.  
 Perché le altre non sono errate)  
 - 2, 3, 4) Tutte descrivono transizioni valide.

134) Quale dei seguenti sono obiettivi per un file Management System?

- Risposta corretta è 1. Tutte le opzioni elencate  
 Motivo) Tutti gli obiettivi menzionati sono effettivamente parte di un File Management System.  
 Perché le altre non sono errate) Sono parte delle funzioni di un File Management System.

95) Quale delle seguenti affermazioni sui dischi magnetici a testina mobile è vera?

- Risposta corretta è 3. Il tempo di accesso tiene conto del posizionamento della testina e della rotazione del disco  
 Motivo) È una caratteristica fondamentale del tempo di accesso.  
 Perché le altre non sono errate) Sono descrizioni errate del funzionamento dei dischi.

110) In un sistema operativo con I/O buffering, quale delle seguenti affermazioni è vera?

- Risposta corretta è 2. Il contenuto della scrittura è copiato nella memoria del sistema operativo prima di essere inviato al dispositivo  
 Motivo) Questa è la definizione di buffering.  
 Perché le altre non sono errate) Non rispecchiano il funzionamento del buffering.

38) Quale delle seguenti affermazioni sul controllo di accesso è vera?

- Risposta corretta è 3. Nel controllo di accesso discrezionale si usa una tabella soggetti-oggetti  
 Motivo) Il controllo discrezionale si basa su una matrice di accesso.  
 Perché le altre non sono errate) Descrivono altri tipi di controllo o sono errate.

44) Quale delle seguenti affermazioni sui semafori per la gestione della concorrenza è falsa?

- Risposta corretta è 3. Per implementare i semafori servono contatori e code condivisi da tutti i semafori  
 Motivo) Ogni semaforo ha i propri contatori e code.  
 Perché le altre non sono errate) Sono caratteristiche reali dei semafori.

100) Quale delle seguenti affermazioni è vera sulla memoria virtuale con paginazione a segmentazione?

- Risposta corretta è 1. Entrambe le tabelle contengono un bit per indicare modifiche  
 Motivo) Le tabelle memorizzano questo bit per ottimizzare il recupero dati.  
 Perché le altre non sono errate) Contengono inesattezze.

71) Quale delle seguenti affermazioni è vera?

- Risposta corretta è 3. Nessuna delle altre opzioni è vera  
 Motivo) Le altre opzioni contengono errori concettuali.

112) Quale dei seguenti sono requisiti per un File Management System?

- Risposta corretta è 4. Tutte le opzioni  
 Motivo) Sono requisiti fondamentali per un File Management System.  
 Perché le altre non sono errate) Sono parte integrante del sistema.

98) Assumendo un sistema monoprocesso, quale delle seguenti affermazioni sugli algoritmi di scheduling è vera?

- Risposta corretta è 2. HRNN considera tempo di esecuzione stimato e attesa  
 Motivo) HRNN bilancia equità e utilizzo.  
 Perché le altre non sono errate) Descrivono algoritmi diversi o in modo errato.

122) Rilocazione degli indirizzi di memoria

- Risposta corretta è 4 (In un sistema a rilocazione con indirizzi assoluti, se si conosce l'indirizzo di memoria dove verrà caricato il programma, il compilatore può inserire direttamente gli indirizzi di memoria corretti nel codice oggetto)

- Motivo) In un sistema con rilocazione a indirizzi assoluti, il compilatore può calcolare gli indirizzi esatti per il codice o getto, poiché la posizione in memoria è già nota.  
 Perché non le altre  
 - Opzione 1) Nei sistemi con registro base, la rilocazione può essere aggiornata più volte, ad esempio durante la rilocazione dinamica.  
 - Opzione 2) La protezione della memoria non può essere eseguita a tempo di compilazione, poiché dipende dall'esecuzione e dal contesto del programma.  
 - Opzione 3) La rilocazione con indirizzi logici richiede hardware dedicato (ad esempio, l'MMU per tradurre indirizzi logici in fisici).

41) Memoria virtuale con paginazione

- Risposta corretta è 1 (Il difetto principale del prepaging è che potrebbe portare in memoria pagine cui poi non si fa riferimento)  
 Motivo) Il prepaging carica preventivamente pagine nella memoria, ma alcune potrebbero non essere mai utilizzate, causando un inutile utilizzo di memoria.  
 Perché non le altre  
 - Opzione 2) È falso che nessuna delle opzioni sia corretta; l'opzione 1 è chiaramente corretta.

- Opzione 3) Il paging on demand non provoca un numero elevato di page fault dopo la fase di assestamento, ma piuttosto all'inizio.  
 - Opzione 4) Placement policy e replacement policy non sono sinonimi; si occupano di aspetti diversi della gestione della memoria.

5) Memoria virtuale con paginazione (falsa)

- Risposta corretta è 3 (Il translation lookaside buffer permette di accedere direttamente al contenuto degli indirizzi di memoria usati più di recente)  
 Motivo) Il TLB non consente di accedere direttamente ai contenuti degli indirizzi di memoria; si limita a velocizzare la traduzione degli indirizzi virtuali in indirizzi fisici.

Perché non le altre

- Opzione 1) È vero che, in caso di mancata corrispondenza nel TLB, bisogna consultare la normale tabella delle pagine.  
 - Opzione 2) È corretto che il TLB sia una cache non completamente trasparente al sistema operativo.

- Opzione 4) È vero che, senza TLB, l'accesso agli indirizzi virtuali può richiedere almeno due accessi in memoria.

144) Process Control Block (PCB)

- Risposta corretta è 4 (Nessuna delle opzioni è contenuta nel PCB)

- Motivo) Il PCB contiene informazioni di controllo del processo, come stato, registri, contatori e informazioni sulla memoria. Non include né lo stack, né il codice del programma, né il bootstrap.  
 Perché non le altre  
 - Opzione 1) Lo stack è mantenuto nella memoria principale o nella pila separata per il processo, non nel PCB.  
 - Opzione 2) Il codice del programma è conservato nella memoria principale, non nel PCB.  
 - Opzione 3) Il programma bootstrap è parte del caricamento del sistema operativo, non del PCB del processo.

135) Allocazione dei file indicizzata

- Risposta corretta è 3 (La tabella di allocazione contiene soltanto l'indirizzo di un blocco, e questo blocco contiene le entry delle porzioni di file allocate oppure l'indirizzo di altri blocchi usati a loro volta per indicizzare le porzioni di file allocate)

- Motivo) Questo descrive accuratamente il metodo di allocazione indicizzata che utilizza un sistema a più livelli per gestire i blocchi di file.

Perché non le altre

- Opzione 1) Non è vero che un blocco contenga sempre tutte le entry allocate; può puntare ad altri blocchi.  
 - Opzione 2) Descrive l'allocazione a lista concatenata, non quella indicizzata.

- Opzione 4) Non è vero che la lista completa dei blocchi è contenuta direttamente nella tabella di allocazione.

39) Grafo dell'allocazione delle risorse

- Risposta corretta è 3 (Un grafo dell'allocazione delle risorse è un grafo diretto aciclico)

- Motivo) Un grafo dell'allocazione delle risorse può essere ciclico in presenza di deadlock; non è sempre aciclico.

Perché non le altre

- Opzione 1) È vero che possono esserci più archi tra lo stesso processo e la stessa risorsa in presenza di più istanze.  
 - Opzione 2) È vero che ci possono essere archi sia dai processi alle risorse (richieste) sia viceversa (assegnazioni).

- Opzione 4) È corretto che i nodi risorsa abbiano un pallino per ogni istanza disponibile.

88) Metodi di gestione del deadlock

- Risposta corretta è 4 (L'unico metodo che non prevede mai la preemption delle risorse è quello che evita il deadlock)

- Motivo) La strategia di prevenzione evita situazioni di deadlock impedendo richieste di risorse che potrebbero causare cicli nel grafo delle risorse, senza mai preempire risorse già allocate.

Perché non le altre

- Opzione 1) È falso che nessuna delle opzioni sia vera, poiché l'opzione 4 lo è.  
 - Opzione 2) La rilevazione richiede il monitoraggio continuo dello stato delle risorse, ma non evita sempre la preemption.

- Opzione 3) La prevenzione è restrittiva, non permisiva.

152) Il file system NTFS

- Risposta corretta è 3

- NTFS utilizza un sistema simile all'allocazione indicizzata basato su record di attributi. Il tipo di attributo DATA contiene puntatori a sequenze di blocchi contigui (runs), permettendo di minimizzare la frammentazione e migliorare le prestazioni.

Perché non le altre

- Opzione 1) La lista concatenata è utilizzata in alcuni file system più semplici, ma non in NTFS. Questo sistema non è efficiente per file di grandi dimensioni.  
 - Opzione 2) NTFS non usa puntatori individuali per ogni blocco, ma raggruppa blocchi contigui in "runs" per migliorare l'efficienza.  
 - Opzione 4) Gli inode di UNIX sono diversi dal sistema NTFS, in quanto usano strutture specifiche per indirizzare blocchi, non sequenze contigue.

7) Quale delle seguenti affermazioni sul buffering dell'I/O è vera?

- Risposta corretta è 4

- Il buffering può consistere nel completare un'istruzione di output dopo che alcune istruzioni successive siano già state eseguite, migliorando l'efficienza.

Perché non le altre

- Opzione 1) Esistono opzioni corrette, quindi questa non è valida.

4. In assenza di translation lookaside buffer, l'accesso ad un indirizzo virtuale può richiedere almeno 2 accessi in memoria

- 144) Quale dei seguenti elementi non è contenuto nel Process Control Block (PCB)?  
 1. Stack del processo  
 2. Codice del programma  
 3. Programma Bootstrap  
 4X. Nessuna delle opzioni è contenuta nel PCB

135) In un sistema operativo con allocazione dei file indicizzata, quale delle seguenti opzioni è vera:

1. La tabella di allocazione contiene soltanto l'indirizzo di un blocco, e questo blocco contiene sempre tutte le entry per ogni porzione allocata al file  
 2. La tabella di allocazione contiene l'indirizzo del primo blocco del file, e ciascun blocco contiene l'indirizzo del prossimo blocco del file  
 3X. La tabella di allocazione contiene soltanto l'indirizzo di un blocco, e questo blocco contiene le entry delle porzioni di file allocate oppure l'indirizzo di altri blocchi usati a loro volta per indicizzare le porzioni di file allocato  
 4. La tabella di allocazione dei file contiene l'indirizzo di un blocco e la lista dei blocchi del file

39) Quale delle seguenti affermazioni è falsa?

1. Nel caso delle risorse riusabili, in un grafo dell'allocazione delle risorse ci possono essere più archi tra lo stesso nodo-processo e lo stesso nodo-risorsa  
 2. Nel caso delle risorse riusabili, in un grafo dell'allocazione delle risorse ci possono essere archi sia da nodi - processi a nodi-risorse che viceversa  
 3X. Un grafo dell'allocazione delle risorse è un grafo diretto aciclico  
 4. In un grafo dell'allocazione delle risorse, all'interno di un nodo rappresentante una risorsa, c'è un pallino per ogni istanza di quella risorsa

88) Quale delle seguenti affermazioni sui metodi di gestione del deadlock è vera?

1. Nessuna delle altre opzioni è vera  
 2. L'unico metodo, che richiede di conoscere in anticipo il massimo numero di risorse che un processo dovrà chiedere, è quello per rilevare il deadlock  
 3. Il metodo più permisivo nei confronti delle richieste di risorse è quello che consiste nel prevenire il deadlock  
 4X. L'unico metodo che non prevede mai la preemption delle risorse è quello che evita il deadlock

152) Il file system NTFS

1. Utilizza un sistema di lista concatenata per tenere traccia dei blocchi del disco assegnati ai diversi file  
 2. Utilizza un sistema di allocazione simile all'allocazione indicizzata basato su record di attributi per ciascun file, dove un tipo di attributo DATA contiene puntatori individuali a tutti i blocchi che compongono un dato file  
 3X. Utilizza un sistema di allocazione simile all'allocazione indicizzata basato su record di attributi per ciascun file, dove un tipo di attributo DATA contiene puntatori a sequenze di blocchi contigui (runs) che compongono un dato file  
 4. Utilizza un sistema di allocazione simile all'allocazione indicizzata simile agli inode di UNIX, in cui un campo DATA all'interno dell'inode del file contiene la sequenza completa dei blocchi che compongono un dato file

7) Quale delle seguenti affermazioni sul buffering dell'I/O è vera?

1. Nessuna delle altre opzioni è corretta  
 2. Avviene direttamente su disco, altrimenti si rischia il deadlock per interferenze con il DMA  
 3. Nel caso ci siano più buffer, vanno gestiti come nel problema dei lettori/scrittori  
 4X. Può consistere nel completare un'istruzione di output I dopo che alcune istruzioni successive ad I siano state eseguite

129) Quale dei seguenti NON è un valido schema di prevenzione del deadlock?

1. Rilasciare tutte le risorse prima di richiederne una nuova  
 2X. Non chiedere mai una risorsa dopo averne rilasciate altre  
 3. Si definisce un ordinamento crescente delle risorse, una risorsa viene data solo se esegue quelle che il processo già detiene  
 4. Richiedere e allocare tutte le risorse necessarie prima dell'esecuzione

48) Quale delle seguenti affermazioni sui metodi di gestione dello spazio libero su disco è vera?

1. Se ci sono blocchi da 1KB, e il disco contiene 1TB, l'occupazione dovuta alla lista di blocchi liberi è dell'1%  
 2. Se viene usata la lista di blocchi liberi, tale lista viene interamente mantenuta in memoria principale  
 3. Nessuna delle altre opzioni è vera  
 4X. Se viene usata la lista di blocchi liberi, c'è un overhead di spazio, contrariamente alla concatenazione di blocchi liberi

153) Nei sistemi operativi con memoria virtuale paginata, tipicamente

1. Il sistema usa sempre e comunque l'allocatore di memoria standard a pagine, senza consentire richieste speciali come ad esempio allocazioni contigue di memoria. Questo per assicurarsi di evitare i problemi di frammentazione interna od esterna  
 2X. Il sistema operativo, in casi speciali, può usare allocator di memoria specializzati che consentano ad esempio allocazioni di porzioni di memoria contigue per richieste particolari (come ad esempio per il DMA)  
 3. Il sistema può modificare dinamicamente la dimensione delle pagine in base alla quantità di memoria richiesta da un processo, in modo tale da minimizzare la frammentazione interna  
 4. nessuna delle altre opzioni è corretta

7) Quale delle seguenti affermazioni sul buffering dell'I/O è vera?

Risposta corretta è 4

Il buffering può consistere nel completare un'istruzione di output dopo che alcune istruzioni successive siano già state eseguite, migliorando l'efficienza.  
 Perché non le altre

- Opzione 1) Esistono opzioni corrette, quindi questa non è valida.
- Opzione 2) Il buffering non avviene direttamente su disco, ma in memoria principale per ridurre il tempo di accesso.
- Opzione 3) Il problema dei lettori/scrittori riguarda la gestione concorrente, non è direttamente collegato al buffering.

129) Quale dei seguenti NON è un valido schema di prevenzione del deadlock?

Risposta corretta è 2

Non chiedere mai una risorsa dopo averne rilasciate altre non è una tecnica valida per prevenire i deadlock, in quanto non garantisce il rispetto delle condizioni necessarie per evitarli.

Perché non le altre

- Opzione 1) Rilasciare tutte le risorse prima di richiederne di nuove è una strategia valida per prevenire i deadlock.
  - Opzione 2) Ordinare le risorse in modo crescente ed evitare richieste fuori ordine è una tecnica valida.
  - Opzione 4) Richiedere tutte le risorse necessarie in una sola volta evita situazioni di attesa circolare, quindi è valida.
- 48) Quale delle seguenti affermazioni sui metodi di gestione dello spazio libero su disco è vera?

Risposta corretta è 4

Se viene usata la lista di blocchi liberi, c'è un overhead di spazio. Questo sistema mantiene una struttura per tracciare i blocchi liberi, a differenza della concatenazione che minimizza l'overhead.

Perché non le altre

- Opzione 1) L'occupazione dovuta alla lista dei blocchi liberi è molto più bassa dell'1%.
- Opzione 2) La lista di blocchi liberi non viene interamente mantenuta in memoria principale, ma solo una parte viene caricata quando necessario.
- Opzione 4) Esistono opzioni corrette, quindi questa non è valida.

153) Nei sistemi operativi con memoria virtuale paginata, tipicamente

Risposta corretta è 2

In casi speciali, il sistema operativo può utilizzare allocatori specializzati per richieste particolari (come DMA), consentendo allocazioni contigue.

Perché non le altre

- Opzione 1) Il sistema operativo non usa sempre e solo l'allocator standard, poiché alcune applicazioni richiedono gestioni particolari.
- Opzione 3) La dimensione delle pagine è fissa e non può essere modificata dinamicamente.

- Opzione 4) Esistono opzioni corrette, quindi questa non è valida.