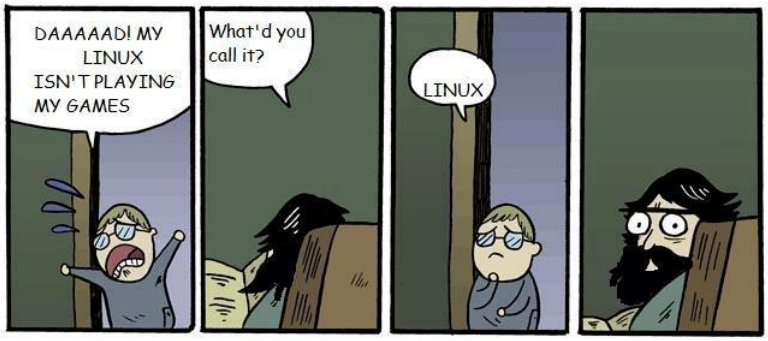
A cura di Karina Chichifoi - Ultima revisione: 27 giugno 2020

SISTEMI OPERATIVI Q&A



https://github.com/TryKatChup/SistemiOperativi_QA

Premessa

Ho scritto questo file in modo da facilitare lo studio e il superamento dell'esame di Sistemi Operativi T. Tuttavia è consigliato integrare questo materiale con le slide della professoressa Anna Ciampolini, disponibili sulla piattaforma *Insegnamenti Online*.

Contribuire alla guida

Se ritieni di poter migliorare la guida, oppure se sono state aggiunte altre domande al di fuori di questo file, o se hai trovato un errore, visita la repository GitHub ed apri una *issue*, oppure inviami un messaggio. Ogni contributo è ben accetto :)

Link Repository: https://github.com/TryKatChup/SistemiOperativi_QA

Figura 1: QR Code alla repository di GitHub

Domanda 1
Confrontare l'organizzazione monolitica con quella basata su microkernel.

Risposta: L'organizzazione monolitica (adottata da GNU/Linux) possiede una struttura interna semplice, e racchiude in un unica componente tutte le procedure, ognuna realizzante una funzione specifica.
In questo modello, i processi interagiscono tra loro per *chiamate a procedura*, rendendo l'esecuzione più veloce, in quanto si evita di passare il controllo al sistema operativo.

Presenta diversi svantaggi: il sistema operativo è complesso e con questa organizzazione risulta:

- difficile da mantenere;
- meno portabile;
- scarsa affidabilità;
- difficilmente estendibile.

Nel modello a microkernel il sistema operativo possiede un ridottissimo nucleo (*kernel*) con poche funzionalità di base, e si interfaccia direttamente con l'hardware.
Il resto del sistema operativo è rappresentato da processi utente.
Questa struttura rende l'esecuzione più lenta poiché un processo deve attendere che il sistema operativo esegua una system call (procedura *interna* al kernel) ma, allo stesso tempo, facilita la manutenzione e rende il sistema generalmente più sicuro.

Domanda 2
Descrivere l'immagine di un processo nel SO Unix.

Risposta: L'immagine di un processo è l'insieme delle aree di memoria e delle strutture dati ad esso dedicate.
In Unix, le immagini sono divise in parti di tipologie diverse:

- In base se è soggetto o meno a **swapping**:
  - ***Swappable***: parte di immagine che può essere spostata in memoria secondaria per liberare spazio ad altre parti di immagini.
  - ***Non swappable***: parte di immagine che non può essere rimossa dalla memoria centrale poiché essenziale al funzionamento.
- In base alla **accessibilità**:
  - ***Kernel***: parte di immagine che viene eseguita solo dal sistema operativo in modalità kernel;
  - ***User***: parte di immagine eseguibile e modificabile dall'utente.

L'immagine viene divisa in sotto parti:

- ***Process structure***: contiene le informazioni per la gestione del processo, come id, stato, informazioni relativi allo scheduling ecc. (non swappable, kernel);
- ***User structure***: contiene le informazioni necessarie solo quando il processo è residente, come registri CPU, informazioni sulle risorse allocate ecc. (swappable, kernel);
- ***Text structure***: contiene il riferimento all'area di memoria dove risiede il codice del processo (non swappable, kernel);
- ***Dati globali***: contiene le variabili globali del processo (swappable, user);
- ***Stack e Heap***: contengono dati dinamici associati al processo (swappable, user);
- ***Stack del kernel***: contiene le system call associate al processo ed è interno al sistema (swappable, kernel).

Domanda 3
Il processo in Unix, caratteristiche e rappresentazione.

Risposta: Un processo è la minima unità di computazione.
In Unix viene definito *pesante* (composto da un singolo task con un unico thread) e ha codice *rientrante* (codice condivisibile da più processi, ma dati non condivisibili). Per permettere la *rientranza* i dati e il codice dei processi sono separati (modello a codice puro).
Nel proprio ciclo di vita, un processo può assumere più stati che ne indicano l'operazione effettuata:

- ***Init***: caricamento in memoria dell'immagine del processo;
- ***Ready***: processo pronto a essere eseguito;
- ***Running***: processo in esecuzione, utilizza la CPU;
- ***Sleeping***: il processo attende un evento per riprendere l'esecuzione;
- ***Swapped***: una parte o tutto il processo è spostato in memoria secondaria;
- ***Zombie***: il processo ha concluso l'esecuzione, ma è in attesa che il proprio padre ne prelevi lo stato di terminazione;
- ***Terminated***: processo deallocato dalla memoria, fine del processo.

Un processo è rappresentato in Unix da diverse parti con relative proprietà:

- ***Swappable***: parte di immagine che può essere spostata in memoria secondaria per liberare spazio ad altre parti di immagini;
- ***Non swappable***: parte di immagine che non può essere rimossa dalla memoria centrale poiché essenziale al funzionamento;
- ***Kernel***: parte di immagine che viene eseguita solo dal sistema operativo in modalità kernel;
- ***User***: parte di immagine eseguibile e modificabile dall'utente.

Il sistema operativo gestisce una struttura dati globale, chiamata *text table*, in cui sono contenuti i puntatori ai codici utilizzati, eventualmente condivisi, dai processi. Ciascun elemento della text table viene chiamato *text structure* e contiene un puntatore al codice e il numero dei processi che lo condividono.

Il *Process Control Block* (PCB) è una struttura dati del nucleo del sistema operativo che contiene le informazioni necessarie per la gestione di un processo, ed è composto da:

- ***Process structure***: contiene le informazioni necessarie al sistema per la gestione del processo, in particolare:
  - id;
  - stato;
  - puntatori alle varie aree dati;
  - riferimento all'elemento della text table associato al codice del processo;
  - informazioni di scheduling;
  - riferimento al processo padre;
  - puntatore alla user structure.

  Più process structure costituiscono la *process table*.
- ***User structure***: contiene le informazioni necessarie solo se il processo è residente in memoria centrale, in particolare:

- registri CPU;
- informazioni sulle risorse allocate;
- informazioni sulla gestione dei segnali;
- ambiente del processo.

Domanda 4
Descrivere i meccanismi del SO all'invocazione di system call.

Risposta: In Unix, per garantire protezione, si hanno due modalità di esecuzione:

- ***User Mode***;
- ***Kernel Mode***.

Il sistema operativo esegue le system call in modalità *kernel*. Quando si tenta di eseguire un'istruzione privilegiata, occorre cedere il controllo al sistema operativo.
Quando un processo tenta di eseguire un'istruzione privilegiata:

- viene inviata un'interruzione al sistema operativo;
- si salva lo stato del programma chiamante;
- si trasferisce il controllo al sistema operativo;
- il sistema operativo esegue in modalità kernel l'istruzione richiesta;
- una volta completata la richiesta, il controllo viene passato nuovamente al programma chiamante, che continua a eseguire istruzioni in modalità user.

Domanda 5
La comunicazione tra processi Unix.

Risposta: La comunicazione tra processi in Unix è vincolata dal fatto che i dati non sono condivisibili.
I processi, quindi, per comunicare tra loro possono ricorrere alla *pipe*, un canale di comunicazione tra processi.
La pipe ha le seguenti caratteristiche:

- unidirezionale, accessibile da un lato di lettura e dall'altro di scrittura;
- più processi della stessa famiglia possono spedire e ricevere messaggi attraverso la stessa pipe;
- ha una capacità limitata, e i messaggi al suo interno vengono gestiti con politica *FIFO* (First In First Out);
- la comunicazione tramite pipe risulta essere **asincrona**.

Per creare una pipe si utilizza la seguente primitiva:

```
int pipe(int fd[2]);
```

Dove `fd` è un puntatore a un vettore di 2 file descriptor:

- `fd[0]` rappresenta il lato di lettura della pipe;
- `fd[1]` rappresenta il lato di scrittura della pipe.

La system call pipe restituisce 0 in caso di creazione della pipe con successo, -1 altrimenti.
Per accedere al lato di lettura della pipe viene utilizzata la system call `read`, per la scrittura viene utilizzata invece la system call `write`.
La comunicazione tramite *pipe* presenta principalmente due svantaggi:

1. La comunicazione avviene solo tra processi della stessa famiglia.
2. Non è persistente: viene distrutta quando terminano tutti i processi che la utilizzano.

La comunicazione tra processi può avvenire anche tramite *fifo*; ha le stesse funzioni della pipe, con il vantaggio che ha validità su tutti i processi in esecuzione sulla macchina.

La fifo presenta le seguenti caratteristiche:

1. Politica di gestione dei messaggi *FIFO*;
2. È rappresentata da un file nel file system, quindi risulta persistente e visibile da tutti i processi;
3. Ha un proprietario, un insieme di diritti e una lunghezza;
4. È aperta e acceduta dalle system call dei file.

La creazione della fifo avviene tramite la seguente system call:

```
int mkfifo(char* pathname, int mode);
```

Dove `pathname` è il nome della fifo, `mode` esprime i permessi.

Domanda 6
Descrivi la sincronizzazione tra processi in Unix

Risposta: La sincronizzazione tra processi permette di imporre dei vincoli sul ordine di esecuzione delle operazioni dei processi interagenti. In Unix, non essendoci condivisione di dati o variabili, risulta difficile imporre vincoli in base all'ordine cronologico delle azioni eseguite o in base alla mutua esclusione di una risorsa.

Il sistema operativo gestisce gli eventi asincroni tramite i *segnali*, ovvero interruzioni asincrone per il processo che li riceve; possono avere origine sia software che hardware.

Il processo che riceve un segnale può decidere di:

- gestirlo con una funzione di sistema (default) o con una funzione definita dal programmatore stesso (handler);
- ignorarlo, senza gestirlo.

Nel primo caso il processo, gestendo il segnale asincrono, interrompe la sua normale esecuzione e una volta terminata la gestione torna ad eseguire il suo codice dal punto in cui era stato interrotto.
Alcuni segnali sono *non bloccabili*: non possono essere gestiti in modo personalizzato, ed è permessa solo l'operazione di default (SIGKILL, SIGSTOP).

Domanda 7
Descrivere l'accesso e la gestione dei file da parte di processi Unix.

Risposta: L'accesso ai file nei sistemi UNIX è di tipo **sequenziale**. L'I/O-Pointer registra di volta in volta la posizione corrente di lettura/scrittura nel file.

Unix mette a disposizione alcune strutture dati fondamentali per la gestione dei file:

- ***Tabella dei file aperti di processo***: è una tabella situata nella user area di ogni processo, contiene una entry per ogni file aperto da quello specifico processo;
- ***Tabella dei file aperti di sistema***: è una tabella di sistema che contiene una entry per ogni file aperto, ma non ancora chiuso. Contiene anche l'IO-Pointer che indica la posizione corrente all'interno del file;
- ***Tabella dei file attivi***: è una tabella di sistema che contiene tutti i descrittori dei file aperti nel sistema (detti anche i-node).

Quando un file viene aperto con `open()`:

1. Viene inserito un nuovo elemento (file descriptor) all'interno della tabella dei file aperti di processo nella prima posizione libera.

2. Viene inserito un nuovo record nella tabella dei file aperti di sistema.
3. Viene copiato l'i-node all'interno della tabella dei file attivi se il file non è già usato e viene effettuato memory mapping.

Quando un file viene chiuso con `close()`:

1. Viene eliminato l'elemento corrispondente al file descriptor nella tabella dei file aperti di processo.
2. Viene eliminato il corrispondente record nella tabella di file aperti di sistema (se il file non è condiviso con altri processi).
3. Viene eliminato l'i-node nella tabella dei file attivi (se il file non è condiviso con altri processi).

Quando viene effettuata una lettura/scrittura da/su file con `read()`/`write()`:

1. Accesso al file tramite file descriptor.
2. Attesa del completamento dell'operazione.

Domanda 8
Descrivere le tecniche di allocazione della memoria centrale.

Risposta: Esistono due metodi di allocazione di codice e dati di un processo in memoria centrale:

1. Allocazione contigua.
2. Allocazione non contigua.

L'allocazione *contigua* presenta varie modalità:

- ***A partizione singola***: la memoria è vista come un unico blocco all'interno del quale è possibile allocare un solo processo per volta, con totale mancanza di multiprogrammazione.
- ***A partizione multipla***: la memoria è suddivisa in più parti, dette partizioni, che possono avere dimensioni fisse o variabili:
  - ***Fisse***: in ogni partizione vengono mappati processi in modo che una determinata partizione sia in grado di contenerli.
    Si ha così il problema della frammentazione interna: all'interno delle partizioni si vengono a creare spazi inutilizzati, poiché non utilizzati dal processo.
    È presente un grado di multiprogrammazione limitato.
  - ***Variabili***: ogni partizione avrà la dimensione del processo allocato.
    Viene risolto così il problema della frammentazione interna e si ha una multiprogrammazione variabile.
    È tuttavia presente il problema della frammentazione esterna, causata dall'allocazione di quantità di memoria maggiori di quelle effettivamente usate dai processi.

L'allocazione *non contigua* presenta diverse tecniche:

- ***Paginazione***: suddivisione dello spazio fisico in pagine (frame) a dimensione costante, dove allocare parti di processi.
  In questo modo viene risolto il problema della frammentazione esterna: a pagine logiche contigue corrispondono pagine fisiche non contigue.
  Viene ridotto il problema della frammentazione interna, grazie alla dimensione fissa delle pagine.
- ***Segmentazione***: estende la tecnica di allocazione a partizioni variabili.
  Lo spazio logico è suddiviso in segmenti caratterizzati da tipo e lunghezza del codice contenuto; ciascun segmento è rappresentato da un intero, che ne permette l'identificazione da parte del sistema operativo.
  Il problema principale della segmentazione, come nella tecnica a partizioni variabili è la frammentazione esterna; può essere risolto con la tecnica di allocazione best-fit, ovvero allocando un segmento solo a processi che più si avvicinano alle dimensioni di quest'ultimo.

- ***Segmentazione paginata***: è una combinazione tra segmentazione e paginazione. Lo spazio logico è diviso in segmenti e questi ultimi, a loro volta, in pagine.
  Viene eliminata la frammentazione esterna, caricando in memoria solo le pagine necessarie.

Domanda 9
Descrivere la paginazione.

Risposta: La paginazione consiste nel partizionamento dello spazio fisico di memoria in cornici (frame) di dimensione *costante* e *limitata* nelle quali mappare porzioni dei processi da allocare. Si ha una distinzione tra lo spazio fisico e quello logico:

- Fisico: costituito da frame, di dimensioni D_f costanti e prefissate.
- Logico: insieme di pagine di dimensione uguale a D_f.

Ogni pagina logica viene mappata su una pagina fisica in memoria centrale.
L'indirizzo fisico ha la seguente struttura:

```
< f, d >
```

Dove `f` è il numero di frame, mentre `d` è l'offset della cella rispetto l'inizio del frame.

L'indirizzo logico invece ha la seguente struttura:

```
< p, d >
```

Dove `p` è il numero della pagina logica, mentre `d` è l'offset della cella rispetto l'inizio della pagina. Gli spazi logici e fisici sono messi in relazione tra loro tramite il **binding**.
Quest'operazione associa a una pagina logica un frame fisico e tiene traccia di queste associazioni in ogni entry della tabella delle pagine, risiedente in memoria o nei registri della CPU.
La tabella delle pagine ha **dimensione fissa**, e non viene sempre utilizzata completamente.
Per distinguere gli elementi significativi da quelli non utilizzati si ricorre al bit di validità per ogni elemento della tabella che può assumere il valore:

- 1: la entry è valida, quindi la pagina appartiene allo spazio logico del processo.
- 0: la entry non è valida.

Per riconoscere gli elementi significativi della tabella delle pagine viene utilizzato anche il **Page Table Length Register**, che contiene il numero degli elementi validi nella tabella delle pagine.
I principali vantaggi di questa tecnica sono:

- eliminazione della frammentazione esterna poiché pagine logiche contigue possono essere mappate su frame fisici non contigui;
- riduzione della frammentazione interna poiché limitata dalla dimensione dei frame;
- possibilità di caricamento in memoria di un sottoinsieme delle pagine logiche di un processo.

Uno svantaggio è la complessità del codice che implementa il paging.

Domanda 10
Descrivere la segmentazione.

Risposta: La segmentazione si basa sul partizionamento dello spazio logico degli indirizzi di un processo in parti (segmenti), ognuno caratterizzati da nome e lunghezza propri.
Ogni segmento è diviso per semantica (scopo del codice in esse contenuto) e a ciascuno è associato un intero per l'identificazione da parte del sistema operativo.
Ogni segmento è allocato in memoria in modo contiguo, ed è possibile applicare diritti di accesso specifici.

L'indirizzo logico è formato da:

- Numero: identificativo del segmento.
- Offset: identificativo della cella di memoria nel segmento.

L'indirizzo fisico è invece formato da:

- Base: indirizzo prima cella del segmento nello spazio fisico.
- Limite: dimensione del segmento.

Al fine di effettuare il binding tra indirizzo logico e fisico dei segmenti, viene utilizzata la **tabella dei segmenti** che risiede in memoria centrale per via delle sue elevate dimensioni (tabella globale).
Essendo la segmentazione un'estensione della tecnica di allocazione a partizioni variabili, presenta il problema della frammentazione esterna: può essere risolto grazie all'uso della tecnica di allocazione ***Best-Fit***, consistente nell'allocazione di un segmento nella porzione di memoria libera più vicina alle sue dimensioni.
Questa tecnica potrebbe richiedere tempi elevati.

Domanda 11
Descrivere la memoria virtuale.

Risposta: La dimensione della memoria centrale può rappresentare una restrizione per grado di multiprogrammazione e dimensione dei processi allocabili. Questa restrizione non è più presente se si sfrutta la memoria virtuale: essa consente di gestire la memoria come se fosse virtualmente illimitata, consentendo l'esecuzione di processi non interamente allocabili in memoria centrale. L'utilizzo della memoria virtuale risulta essere anche *efficiente*, poichè il costo di caricamento di un processo e di swapping è limitato.

La memoria virtuale sfrutta la tecnica della paginazione su richiesta, effettuata dal *pager*, che consiste nel caricare in memoria centrale dalla memoria secondaria solo le pagine del processo che sono necessarie alla sua esecuzione, sostituendo, man mano che l'esecuzione prosegue, le pagine non più utili con quelle attualmente necessarie.
Al fine di distinguere le pagine allocate in memoria centrale da quelle in memoria secondaria, il sistema si avvale di un **bit di validità** che, impostato a 1, indica la presenza della pagina in memoria centrale.
Nell'accesso ad ogni indirizzo di una pagina si consulta il suo bit di validità: se vale 1, allora il programma continua la propria esecuzione, ma, nel caso in cui valga 0, il programma manda un segnale al sistema operativo, che incaricherà il pager di allocare la pagina richiesta in un frame libero della memoria centrale.
Non sempre è presente spazio libero in memoria centrale: quando ciò accade è necessario scegliere una pagina da deallocare per far posto alla nuova pagina necessaria.
Ci sono diversi algoritmi il cui scopo è capire quale pagina sia più corretto deallocare:

- ***LFU*** (Least Frequently Used): dealloca la pagina usata meno di frequente.
- ***FIFO***: dealloca la pagina caricata da più tempo.
- ***LRU*** (Least Recently Used): dealloca la pagina usata meno di recente.

Domanda 12
Descrivere le operazioni svolte dopo un Page Fault.

Risposta: Il Page Fault è un'interruzione di tipo trap che scatta nel momento in cui si sta cercando di accedere a una pagina che non è ancora stata caricata in memoria centrale, oppure quando si effettua un riferimento illegale.
Quando il sistema operativo riceve l'interruzione effettua i seguenti passaggi:

1. Salva lo stato del contesto di esecuzione del processo;
2. Verifica il motivo del Page Fault:
    - Riferimento illegale: violazione delle politiche di protezione e terminazione del processo.

- Riferimento legale: la pagina è memorizzata in memoria secondaria.

3. Copia la pagina dalla memoria secondaria in un frame libero della memoria centrale; nel caso non si trovassero frame liberi si attuerebbe un algoritmo di sovrallocazione, individuando una pagina vittima ed effettuando la sostituzione).

4. Aggiorna la tabella delle pagine.

5. Ripristina il processo, che riparte dall'istruzione interrotta dal page fault.

Domanda 13
Descrivi la sovrallocazione e gli algoritmi di sostituzione.

Risposta: Il problema della sovrallocazione si presenta ogni volta che un processo tenta di accedere ad una pagina non caricata in memoria centrale e non ci sono pagine libere in quest'ultima dove allocare la pagina richiesta.
Per risolvere questo problema, si ricorre ad algoritmi di sostituzione, che hanno lo scopo di individuare una tra le tante pagine allocate in memoria centrale al fine di deallocarla, spostarla nella memoria secondaria e liberare spazio per la pagina necessaria.
Per rendere più efficiente il trattamento di un page fault in caso di sovrallocazione si introduce in ogni elemento della tabella delle pagine un **bit di modifica** (dirty bit):

- se il dirty bit vale 1, la pagina ha subito un aggiornamento da quando è stata caricata in memoria centrale;

- se il dirty bit vale 0, allora la pagina non è stata modificata.

Non sempre è necessario spostare la pagina individuata: nel caso in cui questa non sia stata modificata rispetto alla sua copia in memoria secondaria (dirty bit = 0), il sistema operativo può ridurre i tempi dell'operazione di sostituzione e decidere di eliminarla direttamente, senza doverla trasferire.
Ci sono diversi algoritmi il cui scopo è capire quale pagina sia la migliore da deallocare:

- ***LFU*** (Least Frequently Used): dealloca la pagina usata meno di frequente.
  Per questa tecnica è necessario introdurre un contatore, che tenga conto del numero di accessi per ogni pagina.

- ***FIFO***: dealloca la pagina caricata da più tempo (indipendentemente dal suo uso). In questo caso occorre memorizzare la cronologia dei caricamenti in memoria, o gestire una coda in cui ogni elemento rappresenta una pagina caricata in memoria.

- ***LRU*** (Least Recently Used): dealloca la pagina usata meno di recente. Per implementare tale algoritmo occorre memorizzare la sequenza degli accessi alle pagine in memoria. Due possibili soluzioni:

  - ***Time stamping***: per ciascun elemento della tabella delle pagine esiste un campo che rappresenta l'istante dell'ultimo accesso in memoria.
  - ***Stack***: una struttura dati di tipo stack in cui ogni elemento rappresenta una pagina, ad ogni ogni accesso ad una pagina l'elemento corrispondente si sposta nel top dello stack, mentre il bottom rappresenta la pagina LRU.

  La gestione, in entrambi i casi, risulta essere costosa.

Occorre memorizzare la sequenza degli accessi alle pagine in memoria; viene utilizzato o un time stamping per ciascun elemento della tabella delle pagine, che rappresenta l'istante dell'ultimo accesso in memoria, oppure uno stack, in cui il bottom rappresenta la pagina LRU.
La gestione, in entrambi i casi, risulta essere costosa.

Domanda 14
Descrivere i metodi di accesso e di allocazione nel file system

Risposta: Il file system rappresenta la parte di sistema operativo che fornisce i meccanismi di accesso e memorizzazione delle informazioni in memoria secondaria.
Il file system realizza i concetti astratti di:

- ***File***: unità logica di memorizzazione.
- ***Directory***: insieme di file e direttori.
- ***Partizione***: una suddivisione logica di un dispositivo di memorizzazione fisica a cui viene associato un insieme di file.

Esistono diversi metodi di accesso messi a disposizione dal file system:

- ***Sequenziale***: il file è visto come una sequenza di record logici e viene letto in modo sequenziale. Per accedere a un record bisogna prima accedere a tutti quelli che lo precedono. Ogni operazione di accesso (lettura/scrittura) posiziona il puntatore al file (che indica la posizione corrente all'interno del file) sull'elemento successivo a quello letto/scritto.
- ***Diretto***: il file è visto come un insieme di record logici numerati e si accede specificando il numero del record a cui si vuole accedere.
- ***Ad indice***: a ogni file viene associata una struttura dati contenente l'indice delle informazioni contenute. Per accedere a un record si sfrutta la ricerca a indice, utilizzanod una chiave.

Sono presenti varie tecniche di allocazione nel file system:

- ***Allocazione contigua***: ogni file è mappato su un insieme di blocchi fisicamente contigui.

  Vantaggi:

  1. Possibilità di accedere sia in modo sequenziale che diretto.
  2. Basso costo sulla ricerca di un blocco.

  Svantaggi:

  1. Frammentazione esterna con necessità di compattazione (deframmentazione).
  2. Aumento dinamico della dimensione di un file.

- ***Allocazione a lista concatenata***: i blocchi nei quali viene mappato ogni file sono organizzati in una lista concatenata.

  Vantaggi:

  1. Risolto il problema della frammentazione esterna.
  2. Minor costo di allocazione.

  Svantaggi:

  1. Possibili errori in caso di link danneggiati.
  2. Maggior occupazione dello spazio fisico a causa dei puntatori.
  3. Alto costo nella ricerca.

- ***Allocazione a indice***: i puntatori ai blocchi in cui il file è mappato sono contenuti in un unico blocco (blocco a indice).

  Vantaggi:

1. Risolto il problema della frammentazione esterna.
2. Minor costo di allocazione.
3. Accesso diretto.
4. Maggiore velocità di accesso.

Svantaggi:

1. Scarso utilizzo dei blocchi a indice in caso di file di piccole dimensioni.

Domanda 15
Descrivere l'organizzazione fisica del file system in Unix.

Risposta: In Unix tutto è rappresentato come file.
Questi vengono suddidisi in tre categorie:

- File ordinari (unità logica di memorizzazione).
- Directory (insieme di file e direttori).
- Dispositivi fisici.

A ogni file possono essere associati uno o più nomi simbolici ma un solo descrittore (i-node) identificato da un i-number (intero).
L'i-node contiene gli attributi dei file, come:

- tipo di file;
- proprietario;
- data;
- numero di link.

La superficie del disco dove risiede il File System è partizionata in 4 blocchi:

- ***Boot block***: blocco in cui è memorizzato il codice di avvio del sistema (bootstrap);
- ***Super block***: fornisce i limiti dei 4 blocchi e i puntatori alla lista dei blocchi e degli inode liberi;
- ***Data block***: l'area per la memorizzazione dei file;
- ***I-list***: contiene la lista di tutti gli i-node dei file nel File System.

In Unix, l'allocazione dei file è realizzata con il metodo a indice.
Nell'i-node sono contenuti i puntatori ai blocchi di dati ed eventualmente ad altri blocchi indice (nel caso di file di elevate dimensioni); possono essere presenti più livelli di indirizzamento, ovvero i blocchi puntati possono contenere puntatori ad altri blocchi.
I direttori sono rappresentati come file e contengono un insieme di record logici composti dai file presenti nel direttorio.

Domanda 16
Descrivi lo scheduling della CPU.

Risposta: Lo scheduling della CPU è il meccanismo mediante il quale il sistema operativo assegna l'uso della CPU a un processo, selezionandolo tra tutti i processi che si trovano nella coda ready (pronti a eseguire); questo compito è realizzato dallo **scheduler**.

Il **dispatcher** realizza il cambio di contesto (contest switch), ovvero il caricamento delle variabili locali di un processo che viene schedulato.

I processi nella ready queue che dovranno essere eseguiti verranno scelti in base a vari algoritmi di scheduling, principalmente divisi in due categorie:

1. ***Algoritmi con prelazione*** (pre-emptive): dopo un lasso di tempo fisso per tutti i processi, il sistema operativo può decidere di sottrarre la CPU al processo in esecuzione, assegnandola a un altro. Il processo passa così dallo stato di running allo stato ready.
   I sistemi *time-sharing* hanno sempre uno scheduling **pre-emptive**.

2. ***Algoritmi senza prelazione*** (non pre-emptive): Il sistema operativo non può attuare la sottrazione della CPU e deve aspettare la terminazione (o sospensione) dell'esecuzione del processo che la sta usando.

Per classificare un algoritmo di scheduling bisogna tenere in considerazione alcune variabili. L'obiettivo principale è quello di massimizzare:

- Utilizzo di CPU: percentuale media di utilizzo della CPU nell'unità di tempo.
- Throughput: numero di processi completati nell'unità di tempo.

Si vuole invece minimizzare:

- Tempo di attesa: tempo totale trascorso nella coda dei processi pronti.
- Turnaround: tempo di esecuzione di un processo dall'assegnamento della CPU alla sua terminazione.
- Tempo di risposta: intervallo di tempo tra la sottomissione e l'inizio della prima risposta.

Gli algoritmi di scheduling sono ulteriormente suddivisi per la scelta del processo più opportuno da risvegliare:

- ***FCFS*** (First Come First Served): algoritmo senza prelazione con coda gestita in modo FIFO.
  Non si può influire sull'ordine dei processi. Presenta un tempo di attesa alto se ci sono molti processi che usano in modo intensivo la CPU (CPU-bound).
- ***SJF*** (Shortest Job First): tipicamente senza prelazione, viene effettuata una stima della lunghezza del prossimo CPU-burst per ogni processo e viene schedulato il processo con il CPU-burst minore.
  A ogni processo viene assegnata una priorità (costante o variabile) e viene selezionato il processo con priorità maggiore.
  È presente il rischio di **starvation** per processi con bassa priorità (uso intenso di CPU), dovuto alla costante presenza di processi con priorità minore.
- ***Round robin***: algoritmo con prelazione periodica, la CPU viene assegnata ciclicamente per un intervallo di tempo costante (time slicing).
  Tipicamente usato nei sistemi timesharing.
  Obiettivo principale è la minimizzazione del tempo di risposta.
- ***MLFQ*** (Multiple Level Feedback Queues): effettivamente usato nei sistemi operativi.
  Presenza di più code, ognuna delle quali ha una diversa priorità e viene gestita con un algoritmo di scheduling distinto.
  I processi possono muoversi da una coda all'altra, in base alla loro storia:
  - Da priorità bassa ad alta: processi in attesa da molto tempo.
  - Da priorità alta a bassa: il processo è stato eseguito per molto tempo.

Domanda 17
Descrivere i passaggi di sostituzione ed elaborazione nella Shell Bash.

Risposta: All'esecuzione di un file script contenente istruzioni eseguibili dalla bash shell, vengono effettuati i seguenti passaggi:

1. Sostituzione dei comandi: i comandi contenuti tra backquote (` `) vengono eseguiti e sostituiti dal risultato prodotto.

2. Sostituzione delle variabili e dei parametri: i nomi delle variabili (`$nome`) sono espansi nei valori corrispondenti.
3. Sostituzione dei metacaratteri in nomi di file: i metacaratteri (caratteri speciali a multipla validità come `* ? [ ]`) vengono tradotti in nomi di file.

In alcuni casi è necessario far sì che i caratteri speciali non vengano interpretati come tali.
Per fare ciò si utilizza:

- \ (escape): il carattere che segue è considerato come un normale carattere;
- ‘ ‘ (apici singoli): proteggono da qualsiasi espansione;
- “ ” (apici doppi): proteggono dalle espansioni ad eccezione di `$ \ ‘ ‘`

Domanda 18
Descrivere i Monitor e variabili condizione.

Risposta: Il Monitor è un costrutto sintattico che associa un insieme di operazioni entry (o public) ad una struttura dati comune a più processi, ed è realizzato in modo tale che:

- le operazioni entry siano le uniche permesse sulla struttura;
- le operazioni entry siano **mutuamente esclusive** e quindi un solo processo per volta può essere attivo sul monitor.

All'interno del monitor vengono dichiarate delle variabili locali che definiscono lo stato della risorsa e sono accessibili dall'interno del monitor o dall'esterno, solo da processi che utilizzano metodi entry.
Lo scopo del monitor è quello di controllare l'accesso alla risorsa da parte dei processi concorrenti.

L'accesso alla risorsa avviene mediante due livelli di sincronizzazione:

1. Primo livello: garantisce la **mutua esclusione**, cioè un solo processo alla volta può avere accesso alle variabili del monitor. Se questo dovesse essere occupato, il processo viene sospeso nella entry queue.
2. Secondo livello: regola l'ordine di accesso al monitor. All' invocazione di una procedura viene controllato che la condizione di sincronizzazione sia soddisfatta, assicurandone l'ordinamento.
   Se la condizione non è soddisfatta, il processo viene sospeso nella condition queue associata alla condizione.
   La realizzazione avviene tramite la variabile *condizione* (condition).

La variabile condizione è necessaria solo all'interno del monitor e rappresenta la coda nella quale i processi sospendono la propria esecuzione.
Le operazioni che si possono effettuare sulle variabili condition sono:

- ***wait (condition)***: sospende il processo e lo introduce nella coda individuata dalla variabile condition. Il monitor viene liberato. Al risveglio, il processo riacquisisce l'accesso mutualmente esclusivo al monitor e riprende l'esecuzione.
- ***signal (condition)***: riattiva il processo sospeso nella coda individuata dalla condition.
  Non produce nessun effetto se non ci sono processi in coda.

Domanda 19
Descrivi le operazioni e semantiche delle variabili condizione.

Risposta: Le variabili condizione sono particolari variabili situate all'interno di un monitor, un particolare costrutto sintattico che garantisce, grazie ai suoi metodi entry, la mutua esclusione su una risorsa condivisa. La variabile condizione rappresenta una coda nella quale i processi vengono sospesi nel caso in cui l'accesso alla risorsa sia a loro negato.

Le possibili operazioni sulle variabili condizione sono:

- `await()`: sospende il processo e lo aggiunge nella coda individuata dalla variabile condizione;
- `signal()` / `signalAll()`: risveglia il primo processo/tutti i processi pronto/i nella coda individuata dalla condition.

Semantiche:
Sia P un processo sospeso su una variabile condizione X e sia Q il processo che esegue la `signal()` su tale variabile. Come conseguenza della signal(), entrambi i processi potrebbero proseguire.
Ciò violerebbe la mutua esclusione del monitor: soltanto uno dei due processi può continuare a operare. Le soluzioni per non violare la mutua esclusione sono:

1. ***Signal and Wait***: prevede che il processo P risvegliato riprenda immediatamente la sua esecuzione e che Q venga sospeso affinché non possa cambiare la condizione di sincronizzazione (B). Questa semantica privilegia il processo segnalato rispetto al segnalante: ciò implica che il segnalato (P), proseguendo la sua esecuzione, *è certo* di trovare vera la condizione per la quale è stato risvegliato.

   Lo schema tipico dell'invocazione di wait è, quindi, all'interno di un if:

   ```
   if (!B){
       cond.wait()
       // Accesso alla risorsa
   }
   ```

2. ***Signal and Continue***: questa soluzione, a differenza della prima, privilegia il processo segnalante rispetto al segnalato. Il processo segnalante Q, dopo aver risvegliato P, prosegue la sua esecuzione mediante l'accesso esclusivo al monitor. Il processo P segnalato viene trasferito alla coda associata all'ingresso del monitor (entry-queue).
   Poiché nella entry-queue possono esserci già altri processi, questi potrebbero precedere l'esecuzione di P e modificare il valore della condizione di sincronizzazione (B).
   P dovrà, pertanto, testare nuovamente la condizione di sincronizzazione prima di proseguire utilizzando un while():

   ```
   while (!B){
       cond.wait()
       //accesso alla risorsa
   }
   ```

Domanda 20
Esporre possibili soluzioni al problema della mutua esclusione.

Risposta: L'esigenza della mutua esclusione nasce quando più di un processo alla volta può avere accesso a variabili condivise e/o risorse comuni.
La mutua esclusione impone che le operazioni con le quali i processi accedono alle variabili comuni non si sovrappongano nel tempo.
Se un processo sta utilizzando una certa risorsa o sta accedendo a una variabile, questa non è accessibile a nessun altro processo **in tale istante di tempo**; di conseguenza l'operazione effettuata su tale risorsa o variabile si dice **atomica**.

La sequenza di istruzioni con cui un processo accede e modifica un insieme di variabili comuni prende il nome di sezione critica. A ciascuna variabile interessata possono essere associate una o più sezioni critiche; la regola di mutua esclusione stabilisce che una sola sezione critica alla volta di una classe può essere in esecuzione in un determinato istante.

Per risolvere il problema della mutua esclusione:

1. Le sezioni critiche della stessa classe devono essere eseguite in modo mutuamente esclusivo.

2. Quando un processo si trova **all'esterno** della sezione critica non deve impedire agli altri processi l'accesso alla stessa sezione.

3. Assenza di Deadlock.

Lo schema generale di risoluzione della mutua esclusione è porre un prologo ed un epilogo rispettivamente all'inizio e alla fine della sezione critica.
Il prologo consiste nel chiedere l'autorizzazione ad ogni processo prima di entrare in una sezione critica, affinché venga garantito l'uso esclusivo della risorsa se questa è libera, oppure venga impedito l'acceso se questa è occupata.
Nell'epilogo, invece, al completamento dell'azione il processo deve eseguire una sequenza di istruzioni per dichiarare libera la sezione critica. Le soluzioni possono essere:

- ***Algoritmiche***: non si richiedono particolari meccanismi di sincronizzazione, si sfrutta la possibilità di condividere variabili che indicano lo stato della risorsa.
- ***Hardware-based***: supporto fornito direttamente dall'architettura hardware, come ad esempio la disabilitazione delle interruzioni.
- ***Software-based***: si utilizzano costrutti a livello software (come i semafori) per garantire la mutua esclusione.

Domanda 21
Definizione e uso di semaforo.

Risposta: Il semaforo è uno strumento di sincronizzazione che consente di risolvere qualunque problema di sincronizzazione (come la mutua esclusione) tra thread nel modello ad ambiente globale.
Generalmente è realizzato dal nucleo del sistema operativo ed è un dato astratto rappresentato da un intero non negativo a cui è possibile accedere solo tramite le due operazioni, `p()` e `v()`:

- `p(s)`: blocca il processo fino a che il valore del semaforo diventa > 0 e *successivamente* decrementa tale valore di 1;

```
while(!s);
s--;
```

- `v(s)`: incrementa di 1 il valore del semaforo.

```
s++;
```

Essendo entrambe le operazioni **atomiche** (eseguibili in un solo ciclo di clock), il valore del semaforo viene modificato da un solo processo alla volta. Possiamo, quindi, avere due diversi funzionamenti:

1. `s = 0`: l'esecuzione di `p(s)` provoca l'attesa del processo che la invoca (semaforo rosso).
2. `s > 0`: la chiamata di `p(s)` provoca il decremento di [s] e la continuazione dell'esecuzione (semaforo verde).

Lo scopo dei semafori è quello di:

- eliminare qualsiasi forma di attesa attiva dei processi;
- eliminare qualsiasi forma di Starvation: viene utilizzato un approccio FIFO per scegliere il processo da risvegliare.

Domanda 22
Descrizione e confronto tra gestione a controllo di programma e gestione a interruzioni di dispositivi di I/O.

Risposta: Per gestire dispositivi di I/O vengono utilizzate interfacce, con il compito sia di mascherare la complessità della periferica (astrazione per l'utente), sia di garantire l'accesso alle periferiche in modo omogeneo e il loro corretto funzionamento.

Un altro compito del sottosistema di I/O è quello di naming, relativo alla definizione di uno spazio di nomi tramite i quali i singoli dispositivi vengono identificati univocamente dai processi che su essi devono operare.
Viene definito processo esterno un processo svolto da un'interfaccia fisica che opera in parallelo alla CPU, sui dispositivi di I/O, e come processo applicativo un processo in esecuzione dalla CPU, col compito di inviare comandi all'interfaccia di un dispositivo di I/O.
Normalmente, questo dispositivo è nello stato di stand-by, in attesa che il bit di start del proprio registro di controllo venga impostato a 1.
Una volta attivato, il dispositivo esegue il comando e, una volta completato, segnala l'evento nel registro di stato, ponendo a 1 il corrispondente bit di flag; dopodichè riparte da capo, rimettendosi in attesa.

Gestione a controllo di programma:

- Esecuzione del processo esterno: il processo esterno dell'interfaccia attende l'invio di un comando tramite il registro di controllo. Quando il bit di start in questo registro viene impostato a 1, il dispositivo esegue il comando inviato e segnala la sua fine tramite il registro di stato con un bit di flag a 1.

- Esecuzione del processo applicativo: il processo applicativo prepara il comando e lo invia, impostando contemporaneamente il bit di start a 1; dopodichè attende la fine del comando, segnalato dal bit di flag con valore 1.
  Tale schema risulta inappropriato per sistemi multiprogrammati a causa dei cicli di attesa attiva, nei quali i processi sono schedulati e utilizzano tempo utile della CPU.

Gestione a interruzione:
L'obiettivo è quello di risolvere il problema dell'attesa attiva.
In questa modalità di gestione, si riserva ad ogni dispositivo un semaforo: quando diventa verde, si attiva il dispositivo, abilitandolo a interrompere l'esecuzione della CPU (ponendo nel registro di controllo il bit di abilitazione a 1).
Quando si verifica la transizione del flag da zero a uno il controllore interrompe la CPU, che risponde con una funzione di gestione delle interruzioni del dispositivo.