Appunti di Sistemi Operativi e Reti

Domande e risposte – Parte di Sistemi Operativi (OS)

Capitolo 1 – Introduzione ai Sistemi Operativi

1) Cosa si intende per sistema operativo?

Un sistema operativo è una collezione di programmi che consente di eseguire le operazioni fondamentali del calcolatore, inizializza il sistema e assicura il corretto funzionamento della macchina.

2) A cosa serve il sistema operativo?

Il sistema operativo serve per:

→ fornire una piattaforma di sviluppo per le applicazioni, che permette loro di condividere ed astrarre le risorse hardware;

→ agire da intermediario tra utenti e computer, permettendo agli utenti di controllare l'esecuzione dei programmi applicativi e l'assegnazione delle risorse hardware ad essi;

→ proteggere le risorse degli utenti (e dei loro programmi) dagli altri utenti (e dai loro programmi) e da eventuali attori esterni.

Tali funzionalità permettono ai programmi di poter usare in maniera conveniente le risorse hardware, e di condividerle:

→ da un lato, il sistema operativo astrae le risorse hardware, presentando agli sviluppatori dei programmi applicativi una visione delle risorse hardware più facile da usare e più potente rispetto alle risorse hardware «native»;

→ dall'altro, il sistema operativo condivide le risorse hardware tra molti programmi contemporaneamente in esecuzione, suddividendole tra i programmi in maniera equa ed efficiente e controllando che questi le usino correttamente.

3) Quali sono i componenti di un sistema di elaborazione?

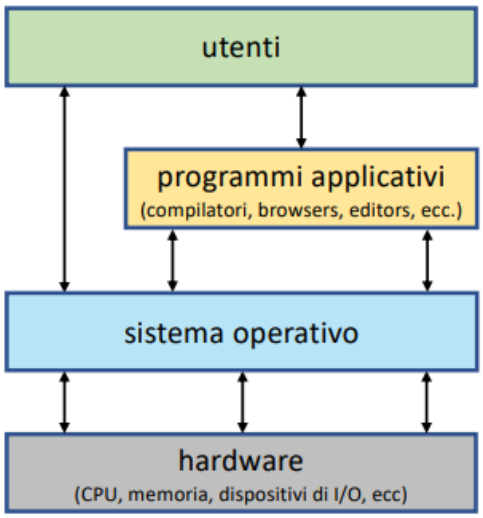
I componenti di un sistema di elaborazione sono:

→ Utenti: persone, macchine, altri computer…

→ Programmi Applicativi: risolvono i problemi di calcolo degli utenti

→ Sistema Operativo: coordina e controlla l'uso delle risorse hw;

→ Hardware: risorse di calcolo (CPU, RAM, HDD).



4) Che cosa si richiede ad un sistema operativo per supportare un certo scenario applicativo?

Per quanto riguarda:

→ Server, mainframe: massimizzare la performance, rendere equa la condivisione delle risorse tra molti utenti

→ Laptop, PC, tablet: massimizzare la facilità d'uso e la produttività del singolo utente che lo usa;

→ Dispositivi mobili: ottimizzare i consumi energetici e la connettività;

→ Sistemi embedded: funzionare senza, o con minimo, intervento umano e reagire in tempo reale agli stimoli esterni (interrupt).

5) Che cosa afferma la maledizione della generalità?

La maledizione della generalità afferma che, se un sistema operativo deve supportare un insieme di scenari applicativi troppo ampio, non sarà in grado di supportare nessuno di tali scenari particolarmente bene. Questo fenomeno si è verificato con OS/360, il primo sistema operativo che doveva supportare una famiglia di computer diversi (la linea 360 dell'IBM).

6) Quali sono i componenti di un sistema operativo?

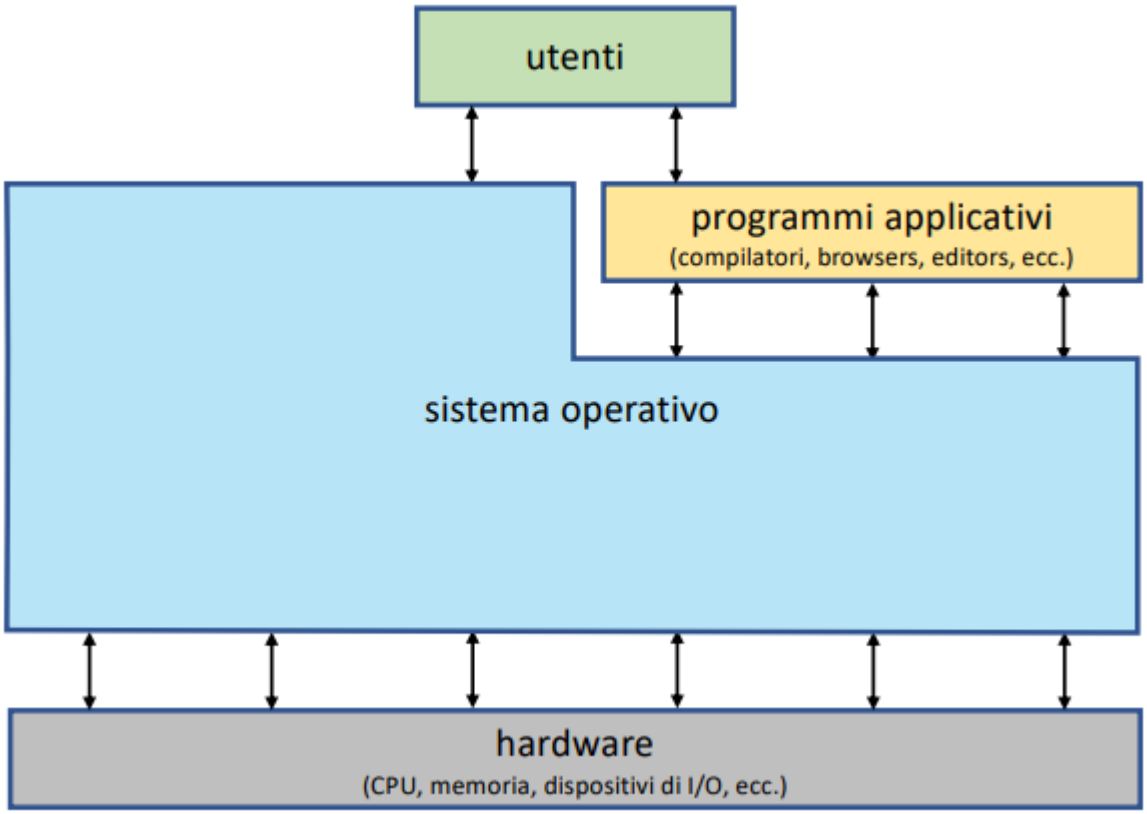
Un sistema operativo comprende almeno:

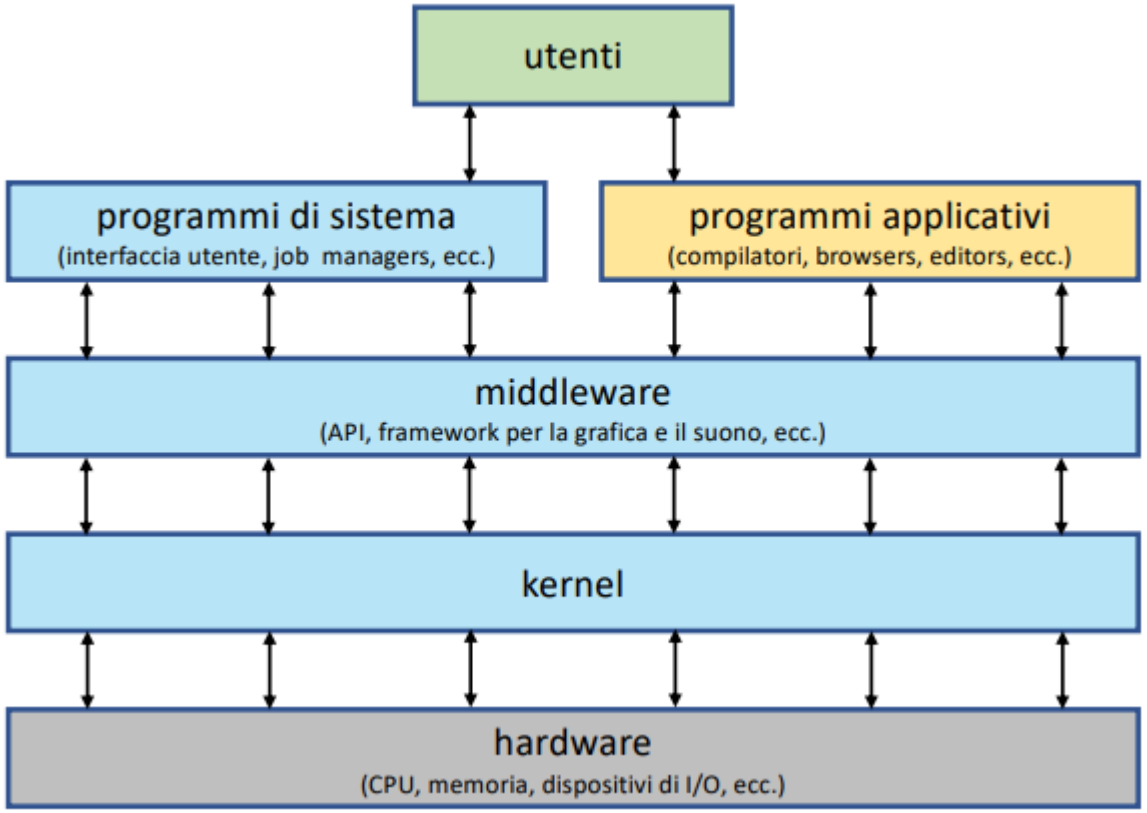
→ Kernel: il quale «si impadronisce» dell'hardware, lo gestisce, ed offre ai programmi i servizi per poterlo usare in maniera condivisa ed astratta;

→ Middleware: servizi di alto livello che astraggono ulteriormente i servizi del kernel e semplificano la programmazione di applicazioni (API, framework per grafica e per suono…);

→ Programmi di sistema: non sempre in esecuzione, offrono ulteriori

funzionalità di supporto e di interazione utente con il sistema (gestione di jobs e processi, interfaccia utente…)





Un sistema operativo offre un certo numero di servizi:
• per i programmi applicativi: perché possano eseguire sul sistema di elaborazione usando le risorse astratte esposte dal sistema operativo;
• per gli utenti: per gestire l'esecuzione dei programmi e stabilire a quali risorse hardware i programmi (e gli altri utenti) hanno diritto;
• per garantire che il sistema di elaborazione funzioni in maniera efficiente.

7) Quali sono i principali servizi di un sistema operativo?

I principali servizi di un sistema operativo sono:
• Controllo processi: questi servizi permettono di caricare in memoria un programma, eseguirlo, identificare la sua terminazione e registrarne la condizione di terminazione (normale o erronea);
• Gestione file: questi servizi permettono di leggere, scrivere, e manipolare files e directory;
• Gestione dispositivi: questi servizi permettono ai programmi di effettuare operazioni di input/output, ad esempio leggere da/scrivere su un terminale;
• Comunicazione tra processi: i programmi in esecuzione possono collaborare tra di loro scambiandosi informazioni: questi servizi permettono ai programmi in esecuzione di comunicare;
• Protezione e sicurezza: permette ai proprietari delle informazioni in un sistema multiutente o in rete di controllarne l'uso da parte di altri utenti e di difendere il sistema dagli accessi illegali;
• Allocazione delle risorse: alloca le risorse hardware (CPU, memoria, dispositivi di I/O) ai programmi in esecuzione in maniera equa ed efficiente;
• Rilevamento errori: gli errori possono avvenire nell'hardware o nel software (es. divisione per zero); quando avvengono il sistema operativo deve intraprendere opportune azioni (recupero, terminazione del programma o segnalazione della condizione di errore al programma);
• Logging: mantiene traccia di quali programmi usano quali risorse, allo scopo di contabilizzarle.

8) Definizioni e differenze tra System Call e API?

La System Call è un meccanismo usato da un processo a livello utente o livello applicativo, per richiedere un servizio a livello kernel del sistema operativo del computer in uso. Le API (Application Programming Interface) sono librerie middleware implementate invocando le chiamate di sistema. Sono fortemente

legate con le librerie standard del linguaggio di implementazione. Le differenze tra System Call e API vengono riportate nella seguente tabella:

System Call	API (Application Programming Interface)
esposte dal kernel	esposte dal middleware
vengono usate dalle API nella loro implementazione	usano le chiamate di sistema nella loro implementazione
non sono standardizzate	sono standardizzate
dipendenti dalla versione del sistema operativo	indipendenti dalla versione del sistema operativo
offrono funzionalità più elementari e più complesse da usare	offrono funzionalità di più alto livello e più semplici da usare

Esempi di API:

- Win32 (sistemi Windows-like)
- POSIX (sistemi Unix-like, inclusi Linux e macOS)

	Win32	POSIX
Controllo processi	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
Gestione file	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Gestione dispositivi	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Comunicazione tra processi	CreatePipe()	pipe()
	CreateFileMapping()	shm_open()
	MapViewOfFile()	mmap()
Protezione e sicurezza	SetFileSecurity()	chmod()
	InitializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

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

ssize_t read(int fd, void *buf, size_t count);
```

9) Programmi di sistema: definizione e caratteristiche

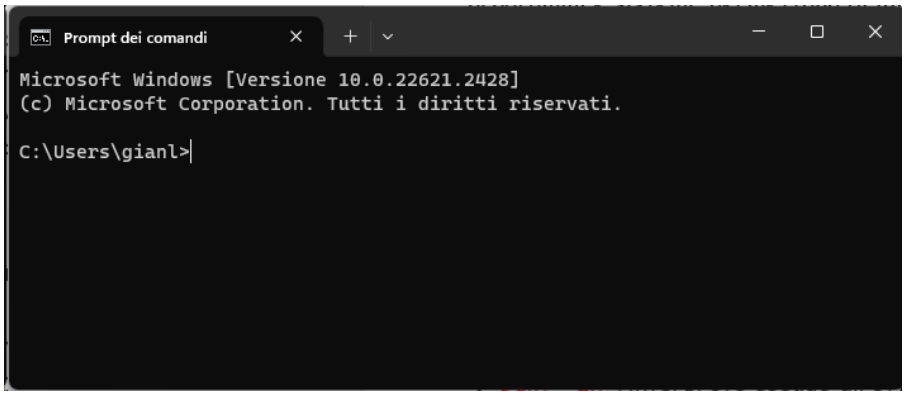
Un programma di sistema è quel programma in grado di gestire tutte le risorse hardware (quindi fisiche) della macchina sulla quale si sta lavorando. Questi permettono agli utenti di avere un ambiente più conveniente per l'esecuzione dei programmi, il loro sviluppo, e la gestione delle risorse del sistema.

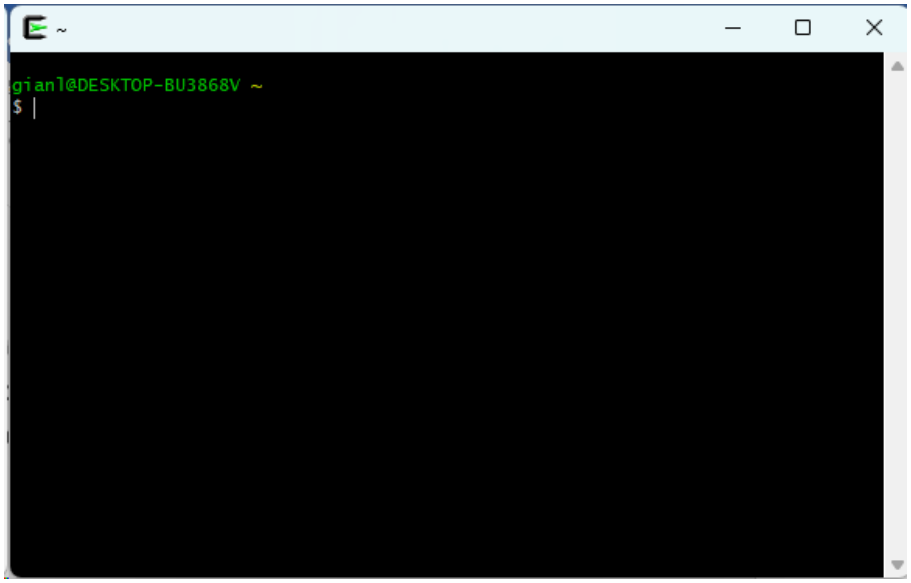
10) Quali sono le tipologie di Programmi di sistema?

Le tipologie di programmi di sistema sono:
→ Interfaccia utente (UI): permette agli utenti di interagire con il sistema stesso; può essere grafica (GUI) o a riga di comando (CLI); i sistemi mobili hanno un'interfaccia touch.
→ Gestione file: creazione, modifica, e cancellazione file e directory.
→ Modifica dei file: editor di testo, programmi per la manipolazione del contenuto dei file.
→ Visualizzazione e modifica informazioni di stato: data, ora, memoria disponibile, processi, utenti… fino informazioni complesse su prestazioni, accessi al sistema e debug. Alcuni sistemi implementano un registry, ossia un database delle informazioni di configurazione.
→ Caricamento ed esecuzione dei programmi: loader assoluti e rilocabili, linker, debugger.
→ Ambienti di supporto alla programmazione: compilatori, assemblatori, debugger, interpreti per diversi linguaggi di programmazione
→ Comunicazione: forniscono i meccanismi per creare connessioni tra utenti, programmi e sistemi; permettono di inviare messaggi agli schermi di un altro utente, di navigare il web, di inviare messaggi di posta elettronica, di accedere remotamente ad un altro computer, di trasferire file…
→ Servizi in background: lanciati all'avvio, alcuni terminano, altri continuano l'esecuzione fino allo shutdown. Forniscono servizi quali verifica dello stato dei dischi, scheduling di jobs, logging…

11) CLI: Definizione e caratteristiche

Una Command Line Interface (CLI) è un'interfaccia a riga di comando, ovvero un'interfaccia basata su comandi testuali per eseguire azioni su un sistema operativo. Esistono due modi per implementare un comando:
→ Built – In: l'interprete esegue direttamente il comando (tipico nell'interprete di comandi di Windows).
→ Come programma di sistema: l'interprete manda in esecuzione il programma (tipico delle shell Unix e Unix-like).
Spesso riconosce un vero e proprio linguaggio di programmazione con variabili, condizionali, cicli…





12) GUI: Definizione e caratteristiche

La Graphical User Interface (GUI) è l'interfaccia grafica con cui gli utenti interagiscono con il sistema operativo. Si tratta quindi di una rappresentazione visiva che include elementi come icone, cursori e pulsanti.

13) Che cosa si intende per multitasking?

Il termine multitasking si indica la capacità di un software di eseguire più programmi contemporaneamente.

14) Cosa rappresenta la macchina di Von Neumann?

Fu la prima proposta di architettura hardware per un calcolatore. Questa architettura vede il calcolatore composto da più elementi:
→ CPU (Central Processing Unit): in grado di acquisire, interpretare ed eseguire programmi/istruzioni;
→ Memoria, per il salvataggio delle informazioni;
→ Periferiche, come la memoria di massa e le periferiche input/output (chiavette e stampanti), permettono lo scambio di informazioni con l'esterno;
→ Bus di sistema, ovvero i collegamenti tra i vari elementi;

Capitolo 2 – Processi e Thread

15) Cosa è un processo?

Un processo è un'entità attiva astratta definita dal sistema operativo allo scopo di eseguire un programma.

16) Differenza tra programma e processo

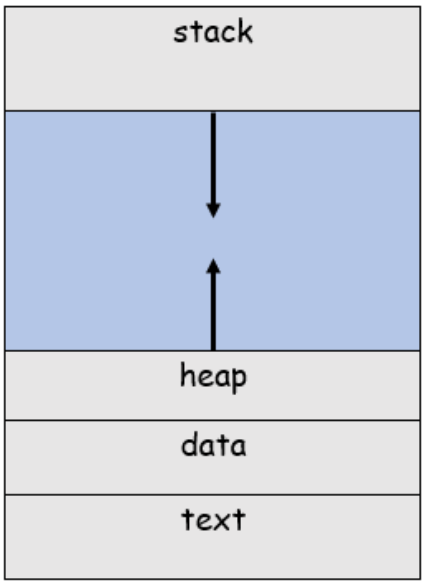
Un programma è un'entità passiva (un insieme di istruzioni, tipicamente contenuto in un file sorgente o eseguibile), mentre il processo è un'entità attiva (è un esecutore di un programma, o un programma in esecuzione).

17) Struttura di un processo: descrivere la sua struttura

Un processo è composto da diverse parti:
→ lo stato dei registri del processore che esegue il programma, incluso il program counter;
→ lo stato della immagine del processo, ossia della regione di memoria centrale usata dal programma;
→ le risorse del sistema operativo in uso al programma (files, lock…);
→ diverse informazioni sullo stato del processo per il sistema operativo.
Attenzione: I processi distinti hanno immagini distinte! Due processi operano su zone di memoria centrale separate!

18) Cosa si intende per spazio di indirizzamento del processo?

Lo spazio di indirizzamento del processo è l'intervallo degli indirizzi di memoria occupati dall'immagine del processo.



19) Da cosa è costituita l'immagine del processo?

L'immagine di un processo di norma contiene:

→ text section: contenente il codice macchina del programma;

→ data section: contenente le variabili globali;

→ heap: contenente la memoria allocata dinamicamente durante l'esecuzione;

→ stack di chiamate: contenente parametri, variabili locali, return address delle varie procedure che vengono invocate durante l'esecuzione del programma.

La text section e data section hanno dimensioni costanti, mentre heap e stack variano durante la vita del processo.

20) Cosa si intende per processo padre e per processo figlio?

Il processo padre è il processo che crea uno o più figli, mentre il processo figlio è il processo che viene creato dal padre. La differenza tra il processo padre e il processo figlio è il valore di ritorno della primitiva `fork()`: il processo padre ottiene un valore intero positivo, mentre il figlio ottiene il valore 0.

21) Relazioni tra padre e figlio: quali sono i possibili scenari?

La relazione padre/figlio è di norma importante per le politiche di condivisione risorse e di coordinazione tra processi. I possibili scenari possono essere:

→ padre e figlio condividono tutte le risorse, o un sottoinsieme delle risorse o nessuna;

→ il figlio può essere un duplicato del padre (stessa memoria e stesso programma), oppure no e bisogna specificare quali programma deve eseguire il figlio;

→ il padre è in sospeso finché i figli non terminano l'esecuzione oppure eseguono in maniera concorrente;

22) Come può avvenire la terminazione di un processo?

Di norma un processo richiede la propria terminazione al sistema operativo. Un processo padre può attendere o meno la terminazione di un figlio, potendo in alcuni casi forzarne la terminazione. Le possibili ragioni di terminazione forzata sono:

→ il figlio utilizza in maniera eccessiva le risorse;
→ le funzionalità del figlio non sono più richieste;
→ il padre termina prima che il figlio termini.

23) Differenza tra processo zombie e processi orfano

Il processo è detto zombie se il processo figlio termina senza l'attesa del padre e le sue risorse non possono essere deallocate. Il processo è detto orfano se il processo padre termina prima del processo figlio.

24) API POSIX: operazioni tra processi

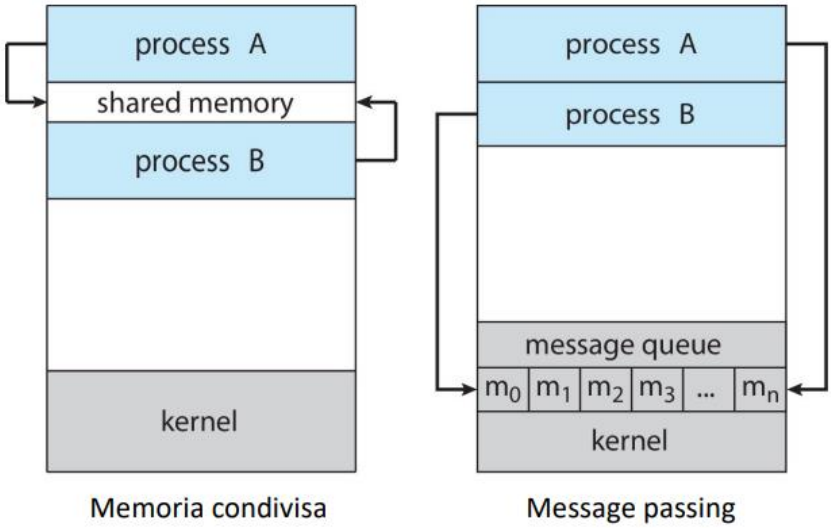
Le primitive sono:

→ `fork()`: crea un nuovo processo figlio, tale che il figlio è un duplicato del padre e ritorna al padre un numero identificatore (PID) del processo figlio, e al figlio il PID 0;
→ `exec()`: sostituisce il programma in esecuzione da un processo con un altro programma, che viene eseguito dall'inizio; viene tipicamente usata dopo una `fork()` dal figlio per iniziare ad eseguire un programma diverso da quello del padre;
→ `wait()`: viene chiamata dal padre per attendere la fine dell'esecuzione di un figlio; ritorna:
a) PID del figlio;
b) il codice di ritorno del figlio.
→ `exit()`: fa terminare il processo che la invoca e accetta come parametro un codice di ritorno numerico, il processo viene eliminato dal sistema operativo, restituisce al processo padre il codice di ritorno.
→ `abort()`: forza la terminazione di un processo figlio.

25) Comunicazione Inter Processo: IPC

La comunicazione inter processo (IPC) permette lo scambio di informazioni e dati tra più processi in evoluzione. Per permettere ciò il sistema operativo deve

fornire primitive di comunicazione inter processo (IPC). Vi sono due tipologie di primitive:
→ memoria condivisa;
→ message passing.



26) IPC memoria condivisa: definizione e caratteristiche

Le caratteristiche di IPC memoria condivisa sono:
→ viene stabilita una zona di memoria condivisa tra i processi che intendono comunicare;
→ la comunicazione è controllata dai processi che comunicano, non dal sistema operativo;
→ sincronizzazione tra i processi (un processo A non deve leggere la zona di memoria condivisa mentre il processo B sta scrivendo o viceversa).

27) IPC message passing: definizione e caratteristiche

Il meccanismo di IPC message passing permette ai processi sia di comunicare che di sincronizzarsi.
Le caratteristiche di questo meccanismo sono:
→ permette ai processi sia di comunicare che di sincronizzarsi;
→ vi sono a disposizione le primitive `send(message)`, con la quale un processo può inviare un messaggio ad un altro processo, e `receive(message)` con la quale un processo può ricevere un messaggio da un altro processo.

28) Pipe: definizione e caratteristiche

Pipe è una sezione della memoria condivisa che elabora l'uso per la comunicazione. Il processo che crea una pipe è il server di pipe. Un processo che si connette ad una pipe è un client di pipe. Un processo scrive le

informazioni nella pipe, quindi l'altro processo legge le informazioni dalla pipe.

- In Unix:
  - Half-duplex
  - Solo sulla stessa macchina
  - Solo dati byte-oriented
- In Windows:
  - Full-duplex
  - Anche tra macchine diverse
  - Anche dati message-oriented

29) Segnali: definizione e caratteristiche

Nei sistemi Unix – like (POSIX, Linux), si dice segnale una notifica che un processo manda in modo asincrono ad un altro processo in modo da causare l'esecuzione di un blocco di codice («callback»), visto come un interrupt. Nelle API Win32 esiste un meccanismo simile, detto Asynchronous Procedure Call (APC), che però richiede che il ricevente si metta esplicitamente in uno stato di attesa, e che esponga un servizio che il mittente possa invocare.

30) API POSIX per la comunicazione inter processo

Le primitive API POSIX sono:

→ `shm_open`: un processo crea un segmento di memoria condivisa, usato anche anche per aprire un segmento precedentemente creato;

```
int shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

→ `ftruncate`: imposta la dimensione del segmento;

```
ftruncate(shm_fd, 4096);
```

→ `mmap`: mappa la memoria condivisa nello spazio di memoria del processo;

```
void *shm_ptr = mmap(0, 4096, PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

→ `pipe`: crea due sezioni di memoria condivisa che elabora l'uso per la comunicazione e ritorna due descrittori, uno per il punto di lettura e uno di scrittura;

```
int p_fd[2];
int res = pipe(p_fd);
```

→ Le funzioni `read` e `write` permettono di leggere e scrivere:

```
char buffer[256];
ssize_t n_wr = write(p_fd[1], "Hello, World!", 14);
ssize_t n_rd = read(p_fd[0], buffer, sizeof(buffer) - 1);
```

31) Named Pipes: definizione e caratteristiche

Una pipe denominata è una pipe unidirezionale o duplex per la comunicazione tra il server pipe e uno o più client pipe. Nei sistemi POSIX le pipe vengono

nominate FIFO e la creazione avviene mediante la seguente istruzione:

```
int res = mkfifo("/home/gianl/myfifo", 0640);
```

Lettura e scrittura:

```
char buffer[256];
ssize_t n_wr = open("/home/gianl/myfifo", O_RDONLY);
ssize_t n_rd = read(fd, buffer, sizeof(buffer) - 1);
```

Chiusura

```
close(fd);
```

Eliminazione

```
unlink("/home/gianl/myfifo");
```

32) Segnali POSIX

Invio segnale POSIX:

```
int ok = kill(1000, SIGTERM);
```

Per registrare una callback per un determinato segnale viene utilizzata l'API sigaction:

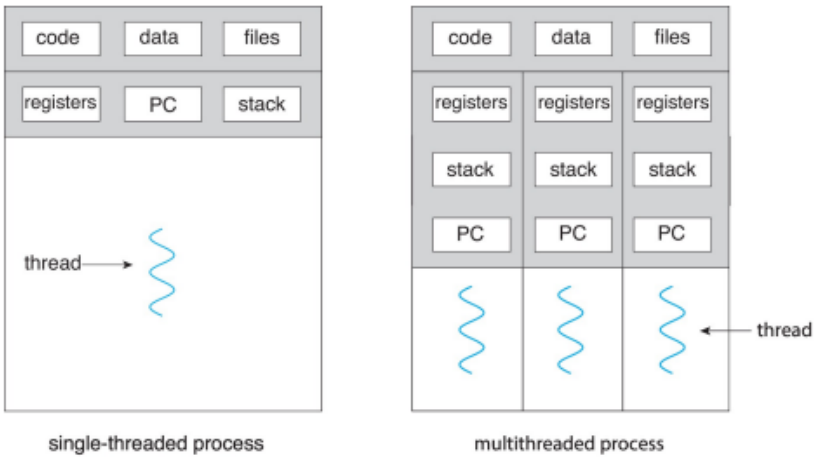
```
struct sigaction act;
sigemptyset(&act.sa_mask);
act.sa_flags = SA_SIGINFO;
act.sa_sigaction = sigterm_handler;
int ok = sigaction(SIGTERM, &act, NULL);
```

33) Che cosa si intende per multithreading?

Il termine multitasking si indica la capacità di un software di eseguire più thread contemporaneamente.

34) Che cosa si intende per thread?

Si dice thread l'unità di base di utilizzo della CPU caratterizzata da identificatore, PC, stack e registri. I thread di uno stesso processo condividono la memoria globale (data), la memoria contenente il codice (code) e le risorse ottenute dal sistema operativo (ad esempio i file aperti). Ogni thread di uno stesso processo però deve avere un proprio stack, altrimenti le chiamate a subroutine di un thread interferirebbero con quelle di un altro thread concorrente.



35) Come avviene la creazione di un thread?

Per creare un nuovo thread si utilizza l'API `pthread_create`, per attendere la fine dell'esecuzione di un thread si utilizza l'API `pthread_join`:

```
pthread_id tid1, tid2;
int ok1 = pthread_create(&tid1, NULL, thread_code, "thread 1");
int ok2 = pthread_create(&tid2, NULL, thread_code, "thread 2");

void *ret1, *ret2;
ok1 = pthread_join(tid1, &ret1);
ok2 = pthread_join(tid2, &ret2);
```

36) Quali sono le due tipologie di cancellazione di thread?

Vi sono due approcci di cancellazione dei thread:

→ Cancellazione asincrona: il thread che riceve la cancellazione viene terminato immediatamente. Il vantaggio di questo approccio è la nessuna necessità di controllare periodicamente se ci sono richieste di cancellazione pendenti.

→ Cancellazione differita: un thread che supporta la cancellazione differita deve controllare periodicamente se esiste una richiesta di cancellazione pendente, e in tal caso terminare la propria esecuzione. Il vantaggio di questo approccio è che dal momento che un thread controlla il momento della propria cancellazione, può effettuare una terminazione ordinata.

Capitolo 3 – Gestione della memoria

37) Come fa, pertanto, un'istruzione macchina di un programma a far riferimento ad una certa locazione di memoria, se il suo indirizzo non è noto a priori ma dipende da dove il programma viene caricato?

Vi sono due possibilità:

→ il compilatore produce codice indipendente dalla posizione (position-independent code (PIC)), ossia codice macchina che utilizza solo indirizzi di

memoria relativi, permettendo così un corretto funzionamento in qualsiasi locazione di memoria in cui il codice viene caricato;
→ produrre codice dipendente dalla posizione e tradurre gli indirizzi dipendenti dalla posizione negli indirizzi corretti (associazione (binding) degli indirizzi).

38) Differenza tra linker e loader

Il linker (link editor) è un software di sistema che combina più file oggetto (diversi file sorgente e librerie) per formare un file eseguibile. Il loader si occupa di caricare in memoria i file eseguibili nel momento in cui devono essere eseguiti.

39) Librerie dinamiche: definizione e vantaggio

Una libreria si dice dinamica se tale libreria viene collegata quando il programma è caricato o durante l'esecuzione del programma stesso. Il vantaggio delle librerie dinamiche è che queste possono essere condivise tra diversi programmi, riducendo le dimensioni dei programmi stessi e risparmiando memoria.

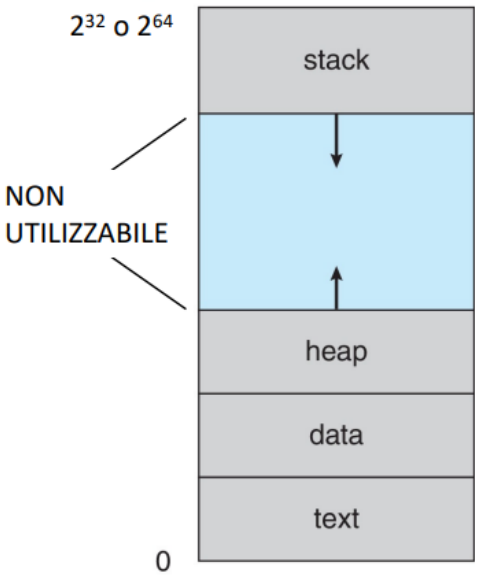
40) Quali sono le tre tipologie di associazione degli indirizzi? Vantaggi e svantaggi

L'associazione degli indirizzi può essere fatta in tre momenti diversi:
→ In compilazione: il linker, a partire dall'indirizzo di caricamento, effettua il binding e genera codice assoluto;
→ In caricamento: il linker genera codice rilocabile e il loader, a partire dall'indirizzo di caricamento, effettua il binding al momento del caricamento in memoria del codice;
→ In esecuzione: il binding viene effettuato dall'hardware dinamicamente mentre il codice viene eseguito.
I vantaggi e svantaggi delle tre tipologie di associazione sono:
→ In compilazione: soluzione semplice, ma se cambia l'indirizzo di caricamento il codice va ricompilato (si possono, ad esempio, avere n versioni per n diversi indirizzi di caricamento);
→ In caricamento: permette di variare liberamente l'indirizzo di caricamento da esecuzione ad esecuzione, ma è una soluzione lenta che non permette di rilocare (spostare) l'immagine di un processo durante la sua esecuzione; inoltre l'eseguibile deve contenere delle opportune tabelle che indichino le istruzioni macchina da modificare;

→ In esecuzione: soluzione rapida che permette di rilocare l'immagine di un processo anche durante la sua esecuzione, e di proteggere la memoria centrale non assegnata ad un processo, ma richiede il supporto dell'hardware.

41) Cosa si intende per spazio di indirizzamento virtuale?

Si dice spazio di indirizzamento virtuale, uno spazio di indirizzamento indipendente dagli indirizzi fisici della memoria centrale nella quale l'immagine del processo è caricata che si estende dall'indirizzo 0 al massimo indirizzo consentito dall'architettura del processore.
Le caratteristiche dello spazio di indirizzamento virtuale sono:
→ lo spazio di indirizzamento virtuale di un processo è di regola molto più ampio della memoria centrale;
→ buona parte dello spazio di indirizzamento virtuale non è utilizzabile dal processo perché non è associato a nessuna regione di memoria centrale;
→ inutilizzabile tra stack e heap che possono essere dinamicamente estesi e ridotti.



42) API POSIX: Gestione memoria

```
//Per mappare 4 KB di memoria a partire dall'indirizzo virtuale 0xa0000000
void *ptr = mmap(0xa0000000, 4096, PROT_READ|PROT_WRITE, MAP_ANONYMOUS, 0, 0);

//Per mappare, a partire dall'indirizzo virtuale 0xb0000000, 8192 bytes del file /usr/foo a partire dal byte 100
int fd = open("/usr/foo", O_RDWR);
void *ptr = mmap(0xb0000000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 100);

// Per sincronizzare le modifiche in memoria con il file
int ok = msync(0xb0000000, 8192, MS_SYNC|MS_INVALIDATE);
```

Capitolo 4 – File system

43) Cosa si intende per file system?

Il file system è il modo attraverso il quale il sistema operativo memorizza in linea i dati e i programmi. Il file system è costituito da un insieme di file e da una struttura delle directory, che organizza i file.

44) Cosa si intende per file?

Un file è una unità di memorizzazione logica, un insieme di informazioni correlate, registrate in memoria secondaria, alle quali è stato dato un nome. A sua volta è costituito da una sequenza di record, righe, bit o byte, il cui significato è definito dal creatore del file.

45) Attributi di un file

Un file possiede un insieme di attributi:
→ nome: è di solito l'unica informazione in forma umanamente leggibile;
→ identificatore: un'etichetta unica fornita dal file system per distinguere i file;
→ tipo: tipo di dati contenuti nel file (alcuni sistemi operativi non hanno questo attributo);
→ locazione: dispositivo di memoria secondaria e posizione nel dispositivo dove l'informazione del file è memorizzata;
→ dimensione: in byte, parole, record…
→ protezione: informazione di controllo accessi;
→ ora, data e utente che ha creato, letto o modificato per ultimo il file;
→ attributi estesi: checksum, codifica caratteri, applicazioni correlate…

46) Operazioni dei processi sui file

Le operazioni effettuabili sui file sono:
→ creazione: viene riservato spazio nel filesystem per i dati, e viene aggiunto un elemento nella directory;
→ apertura: effettuata prima dell'utilizzo di un file;
→ lettura: a partire dalla posizione determinata da un puntatore di lettura;
→ scrittura: a partire dalla posizione determinata da un puntatore di scrittura (di solito coincide con il puntatore di lettura);
→ riposizionamento (seek): spostamento del puntatore all'interno del file;
→ chiusura: effettuata alla fine dell'utilizzo di un file;
→ cancellazione e troncamento: il troncamento cancella i dati ma non il file con i suoi attributi.

47) Lock dei file

Esistono quattro tipologie di lock dei file:
→ lock condiviso: detto anche lock di lettura; più processi possono acquisirlo, proibisce l'acquisizione di un lock esclusivo;
→ lock esclusivo: detto anche lock di scrittura; solo un processo alla volta può acquisirlo, proibisce l'acquisizione di un lock condiviso;
→ lock obbligatori (mandatory): il sistema operativo impedisce l'accesso al file ai processi che non lo detengono;
→ lock consultivi (advisory): il sistema operativo offre il lock ma non regola l'accesso al file: sono i processi che devono evitare di accedere al file se non hanno il lock.

48) Tipi di file

Esistono due tipologie di file:
→ file di testo, binari o numerici;
→ programmi.
Le possibili tecniche per riconoscere un file sono:
→ schema del nome;
→ attributi nei file;
→ "magic number" all'inizio del file.

49) Struttura dei file

Le possibilità sono:
→ Nessuna struttura: nei sistemi Unix-like un file è una sequenza di byte;
→ Sequenza di record: righe di testo o record binari, a struttura e lunghezza fissa o variabile;
→ Strutture più complesse e standardizzate: esempio formato PE in Windows, a.out ed ELF nei sistemi Unix-like, Mach-O in macOS;

50) Metodi di accesso

Esistono 3 tipologie di metodi di accesso:
→ Metodo di accesso sequenziale: il file è una sequenza di record a lunghezza fissa e le operazioni effettuabili sono `read_next()` e `write_next()` che leggono/scrivono il successivo record dalla posizione corrente;
→ Metodo di accesso diretto: le operazioni ammesse sono read(n) e write(n) per accedere direttamente all'n-esimo record;
→ Metodo di accesso indicizzato: in alcuni sistemi operativi, ad esempio quelli

per mainframe IBM, i file possono essere sequenze di record ordinate secondo un determinato campo chiave del record.

51) Directory: struttura

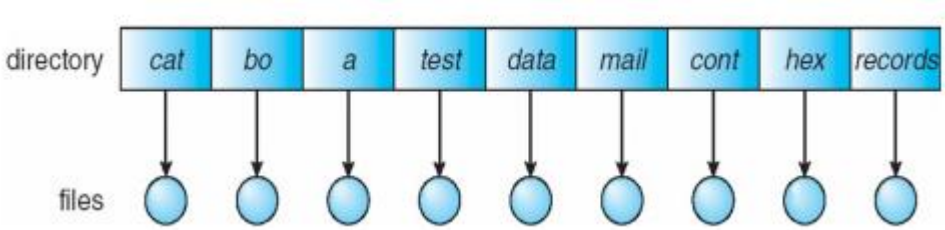
Una directory è una tabella che permette di associare il nome di un file ai dati (e metadati) contenuti nel file stesso. Sia i file che le directory risiedono su disco: deve esservi almeno una directory nel file system.

52) Operazioni sulle directory

Le operazioni su una directory sono:
→ creazione di un file in una directory;
→ cancellazione di un file in una directory;
→ ridenominazione di un file o spostamento da una directory ad un'altra;
→ elenco dei file in una directory;
→ ricerca di un file: basata sul nome, o su uno schema di possibili nomi attraversamento del file system, ad esempio per backup.

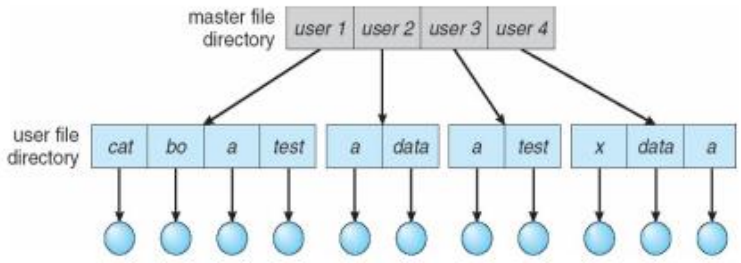
53) Struttura di una directory

La struttura di una directory può essere:
→ Ad un livello: una sola directory per tutti i file.
Vantaggio: semplicità;
Svantaggio:
a) difficoltà nel naming file quando sono molti;
b) difficoltà riguardo al raggruppamento file di utenti diversi.



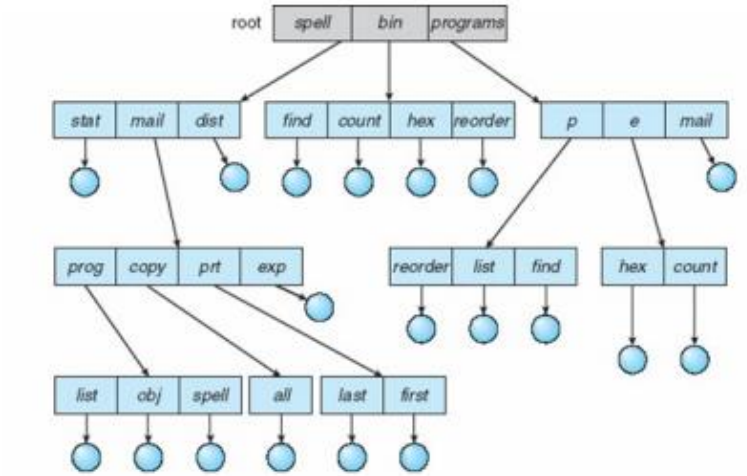
→ A due livelli: la directory principale contiene delle sottodirectory, una per ogni utente, che contiene i file dell'utente. Utenti diversi possono dare lo stesso nome a file diversi. Occorre quindi usare dei nomi di percorso (path name) per identificare un file univocamente:

- /user2/data (separatori Unix-like)
- \user2\data (separatori Windows-like)
- >user2>data (separatori MULTICS-like)



→ Ad albero: ogni directory contiene ricorsivamente files e altre directory. Permette agli utenti di raggruppare i propri file e per semplificare l'accesso ad ogni programma è assegnata una directory corrente, dalla quale si possono specificare path relativi.

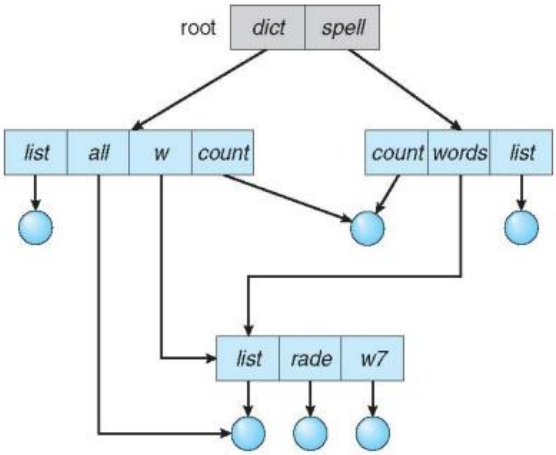
Esempio: se la directory corrente è /programs/mail, un path relativo potrebbe essere prt/first



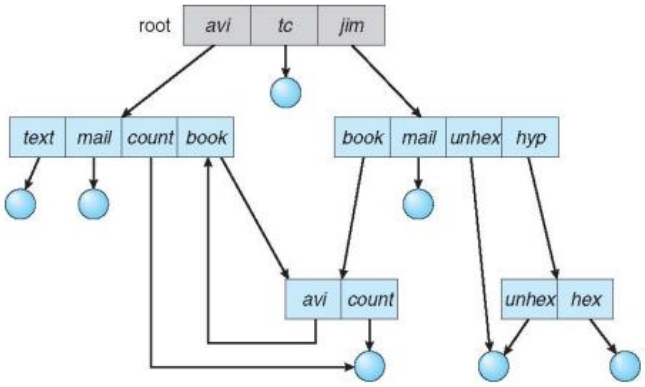
→ A grafo aciclico: Permette l'aliasing (più di un nome per lo stesso file). Questi vengono gestiti mediante:

a) Hard links: duplicazione voci di directory; viene introdotto un contatore ai riferimenti, quando è a zero viene cancellato il file;

b) Link simbolici: riferimenti simbolici ad un path assoluto, quando questo è cancellato restano dangling; non sono aggiunti al contatore dei riferimenti.



→ A grafo generale: Possibilità di hard link anche a directory a livelli superiori, persino che contengono ricorsivamente il link stesso. Per determinare se un file non è più referenziato occorre un algoritmo di garbage collection.



54) Cosa si intende per access control list (ACL)?

Si dice access control list (ACL), un elenco di controllo degli accessi è un elenco di autorizzazioni associate a una risorsa di sistema. Specifica a quali utenti o processi di sistema è concesso l'accesso alle risorse, nonché quali operazioni sono consentite su determinate risorse.

Esempio:

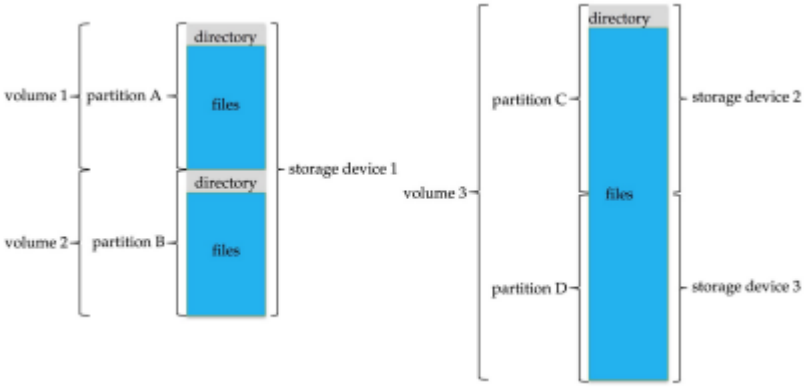
```
                              r w x
• Accesso proprietario: 7 ⇒ 1 1 1
• Accesso gruppo:       6 ⇒ 1 1 0
• Accesso pubblico:     4 ⇒ 1 0 0
```

Supponiamo di avere un file game, di volergli assegnare un gruppo G, e di volergli attribuire i permessi di cui sopra:

```
• chgrp G game #cambia gruppo al file game
• chmod 764 game #cambia i permessi al file game
```

55) Cosa si intende per volume?

Si dice volume è una zona di archiviazione contenente un filesystem. Di solito è contenuto in una partizione, su un solo dispositivo. Un dispositivo di archiviazione può essere suddiviso in partizioni.



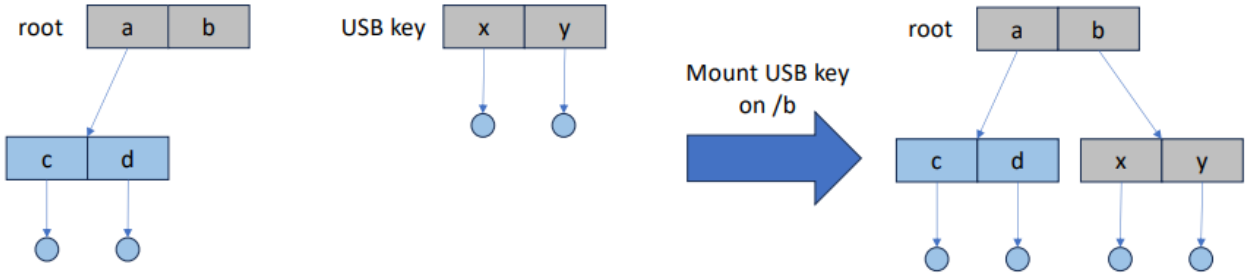
56) Cosa si intende per montare un volume?

Si dice montare un volume, il processo logico di preparazione di un file system (di un dispositivo esterno di memorizzazione o di una partizione del disco rigido) teso a renderlo accessibile al sistema operativo della macchina in uso. Ciò avviene leggendo dalla memoria certi indici delle strutture dati letti dal dispositivo di memoria di massa.

Per montare un volume occorre fornire al sistema operativo:

- l'identificazione del dispositivo/partizione dove risiede il volume;
- il punto di montaggio, ossia la locazione nella struttura di file e directory alla quale «agganciare» il filesystem contenuto nel volume (tipicamente una directory vuota).

Esempio:



57) API POSIX per gestione dei file

- Per creare un file ciao.txt con diritti di lettura e scrittura per il proprietario, e di sola lettura per i membri del gruppo e gli altri:

```
int fd = creat("ciao.txt", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
```

- Per aprire un file ciao.txt in sola lettura:

```
int fd = open("ciao.txt", O_RDONLY);
```

- Per leggere da un file:

```
char buf[NBYTES];
int tot_read = read(fd, buf, NBYTES);
```

- Per scrivere su un file:

```
off_t offset = lseek(fd, 10, SEEK_CUR);     /* 10 bytes dopo la posizione corrente */
```

- Per riposizionarsi all'interno di un file:

```
char buf[] = "Hello";
int tot_written = write(fd, buf, 5);
```

- Per chiudere un file:

```
int ok = close(fd);
```

- Per cancellare il file ciao.txt:

```
int ok = unlink("ciao.txt");
```

• Per troncare il file ciao.txt alla lunghezza n:

```
int ok = truncate("ciao.txt", n);
```

58) API POSIX per gestione delle directory

Per visitare il contenuto di una directory:

```
#include <dirent.h>

DIR *dir;
struct dirent *dp;

if ((dir = opendir(".")) == NULL) {
    perror("Cannot open .");
    exit(1);
}
while ((dp = readdir(dir)) != NULL) {
    ...
}
```